Chapter 8

OBJECT MANIPULATION - INHERITANCE

Chapter 8

- **Advanced constructors**
- **Destructors**
- **Constant Objects**
- **Static class members**
- **Inheritance**

Advanced constructors

- **Constructors can do more than initializing data members.**
- **They can execute member functions and perform other type of initialization routines that a class may require when it first starts.**

Parameterized Constructors

- **Constructors can accept *parameters* that a client can use to pass initialization values to the class.**

- **Example: We can have a constructor function definition in the implementation file as follows:**

```
Payroll::Payroll(double dFed, double dState){
    dFedTax = dFed;
    dStateTax = dState;
};
```

- **Once you create a parameterized constructor, you have to supply parameters when you instantiate a new object.**

```
//Payroll.h
class Payroll{
public:
   Payroll(double, double);
private:
   double dFedTax;
   double dStateTax;
}
//Payroll.cpp
#include “Payroll.h
#include <iostream.h>
Payroll::Payroll(double dFred, double dState){
    dFedTax = dFed;
    dStateTax = dState;
};
void main( ){
    Payroll employee;          //illegal because of no parameter values
          ……
}
```

Overloading constructor functions

- **Constructor functions can be *overloaded*. You can instantiate different versions of a class, depending on the supplied parameters**

- **Being able to overload a constructor function allows you to instantiate an object in multiple ways.**

- **Example:**

```
//Payroll.h
class Payroll{
public:
   Payroll();                              //three version of constructors
   Payroll(double dFed);
   Payroll(double dFed, double dState);
private:
   double dFedTax;
   double dStateTax;
}
```

```
//Payroll.cpp
#include “Payroll.h
#include <iostream.h>
Payroll::Payroll(){
    dFedTax = 0.28;
    dStateTax = 0.05;
};

Payroll::Payroll(double dFed){
    dFedTax = dFed;
};

Payroll::Payroll(double dFred, double dState){
    dFedTax = dFed;
    dStateTax = dState;
};
```

```
void main( ){
    Payroll employeeFL(0.28);
    Payroll employeeMA(0.28, 0.0595);
}
```

Initialization Lists

- ***Initialization lists*, or member initialization lists, are another way of assigning initial values to a class's data members.**
- **An initialization list is placed after a function header's closing parenthesis, but before the function's opening curly braces.**

- **<u>Example</u>:**

```
Payroll::Payroll(double dFed, double dState){
    dFedTax = dFed;
    dStateTax = dState;
};
```

- **You can use initialization list to rewrite the above constructor.**

```
Payroll::Payroll(double dFed, double dState)
    :dFedTax(dFed), dStateTax(dState){
};
```

Parameterized Constructors that Uses Default Arguments

- **To create a parameterized constructor that uses default arguments, we can put the *default values* at the constructor prototype.**

- **Example: The class *Employee* has a constructor with the prototype:**

  Employee(const int id = 999, const double hourly = 5.65);

- **This format provides the constructor function with default values for two arguments. When we create an *Employee* object, the default values in the constructor prototype are assigned to the class variables.**

Example 8.1.2

```
#include<iostream.h>
class Employee{
private:
    int idNum;
    double hourlyRate;
public:
    Employee(const int id = 9999, const double hourly = 5.65);
    void setValues(const int id, const double hourly);
    void displayValues();
};
Employee::Employee(const int id, const double hourly)
{
    idNum = id;
    hourlyRate = hourly;
}
void Employee::displayValues()
{
    cout<<"Employee #<< idNum<<" rate $"<<
                hourlyRate<< " per hour "<<endl;
}
```

```
void Employee::setValues(const int id, const double hourly)
{
    idNum = id;
    hourlyRate = hourly;
}
void main(){
    Employee assistant;
   cout<< “Before setting values with setValues()”<< endl;
    assistant.displayValues();
    assistant.setValues(4321, 12.75);
    cout<< “After setting values with setValues()”<< endl;
    assistant.displayValues();
}
```

The output of the above program:

```
Before setting values with setValues()
Employee #9999 rate $5.65 per hour
After setting values with setValues()
Employee #4321 rate $12.75 per hour
```

DESTRUCTORS

- **A *default destructor* cleans up any resources allocated to an object once the object is destroyed.**

- **To delete any heap variables declared by your class, you must write your own destructor function.**

- **You create a destructor function using the name of the class, the same as a constructor function, preceded by a tilde ~. Destructor functions cannot be overloaded or accept parameters.**

- **A destructor is called in two ways;**

  - when a stack object loses scope when the function in which it is declared ends.

  - when a heap object is destroyed with the delete operator.

Example 8.2.1

```
//Stocks02.h
class Stocks {
public:
    Stocks(char* szName);
    ~Stocks();                    //destructor
    void setStockName(char* szName);
    char* getStockName();
    void setNumShares(int);
    int getNumShares(int);
    void setPricePerShare(double);
    double getPricePerShar();
    double calcTotalValue();
private:
    char* szStockName;
    int iNumShares;
    double dCurrentValue;
    double dPricePerShare;
};
```

```
//Stocks.cpp
#include “stocks_02.h”
#include <string.h>
#include <iostream.h>
Stocks::Stocks(char* szName){
    szStockName = new char[25];
    strcpy(szStockName, szName);
};

Stocks::~Stocks(){
    delete[] szStockName;
    cout <<”Destructor called” << endl;
}
…
…
```

```
void main(){
    Stocks stockPick1("Cisco");
    stockPick1.setNumShares(100);
    stockPick1.setPricePerShare(68.875);
    Stocks* stockPick2 = new Stocks("Lucent");   //heap object
    stockPick2->setNumShares(200);
    stockPick2->setPricePerShare(59.5);
    cout << "The current value of your stock in "
         << stockPick1.getStockName() << " is $"
         << stockPick1.calcTotalValue()
         << "." << endl;
    cout << "The current value of your stock in "
         << stockPick2->getStockName() << " is $"
         << stockPick2->calcTotalValue()
         << "." << endl;
    delete stockPick2;
}
```

- **The output of the above program:**

  The current value of your stock in Cisco is $6887.5
  The current value of your stock in Lucent is $11900
  Destructor called.
  Destructor called.

- **The *stockPick1* object calls the destructor when it is destroyed by the *main()* function going out of scope.**
- **The *stockPick2* object does not call the destructor since it is declared on the heap and must be deleted manually.**
- **To delete the *stockPick2* object manually, we add the statement *delete stockPick2;* to the *main()* function**

CONSTANT OBJECTS

- **If you have any type of variable in a program that does not change, you should always use the *const* keyword to declare the variable as a constant.**

- **To declare an object as contant, place the *const* keyword in front of the object declaration.**

- **Examle:**

  const Date currentDate;

- **Note: *Constant data members* in a class can not be assigned values using a standard assignment statement. You must use an *initialization list* to assign initial values to constant data members.**

Example:

```
//Payroll.h
class Payroll{
public:
   Payroll();
private:
   const double dFedTax;
   const double dStateTax;
};
//Payroll.cpp
#include “Payroll.h”
#include <iostream.h>
Payroll::Payroll( ){
    dFedTax = 0.28;          //illegal
    dStateTax = 0.05;        //illegal
};
```

Using an initialization list

```
Payroll::Payroll()
:dFedTax(0.28), dStateTax(0.05){
};
```

Constant Functions

- **Another good programming technique is to use the *const* keyword to declare *get* functions as *constant function*.**

- **The *const* keyword makes your programs more reliable by ensuring that functions that are not supposed to modify data *cannot modify data*.**

- **To declare a function as constant, you add the *const* keyword after a function's parentheses in both the function declaration and definition.**

Example:

```
//Payroll.h
double getStateTax(Payroll* pStateTax) const;
//Payroll.cpp
double Payroll::getStateTax(Payroll* pStateTax)
const {
  return pStateTax->dStateTax;
};
```

INHERITANE

- ***Inheritance*** **is a form of software reusability in which new classes are created from existing classes by inheriting their attributes and behaviors, and overriding these with capabilities the new classes require.**

  Software *reusability* saves time in programming development.

BASIC INHERITANCE

Inheritance **refers to the ability of one class to take on the characteristics of another class.**

Base Classes and Derived Classes

- **When you write a new class that inherits the characteristics of another class, you are *deriving* or *subclassing* a class.**
- **An inherited class is called the *base class*, or *superclass* and the class that inherits a base class is called a *derived class* or *subclass*.**
- **A class that inherits the characteristics of a base class is said to be *extending* the base class since you often extend the class by adding your own class members.**

Base Classes and Derived Classes

- **When a class is derived from a base class, the derived class inherits all of the base class members and all of its member functions, with the exception of: *constructor functions, copy constructor functions, destructor functions* and *overloaded assignment (=) functions***

- **A derived class must provide its own implementation of these functions.**

- **The class header declaration for a derived class *Customer* which inherits the characteristics of the *Person* class is as follows:**

```
class Customer: public Person
{
 ……
}
```

- **Once you extend a base class, you can access its class member directly through objects instantiated from the derived class.**

- **Example 8.4.1**

```
#include<iostream.h>
#include<string.h>
class Person
{
private:
   int idnum;
   char lastName[20];
   char firstName[15];
public:
   void setFields(int, char[], char[]);
   void outputData( );
};
```

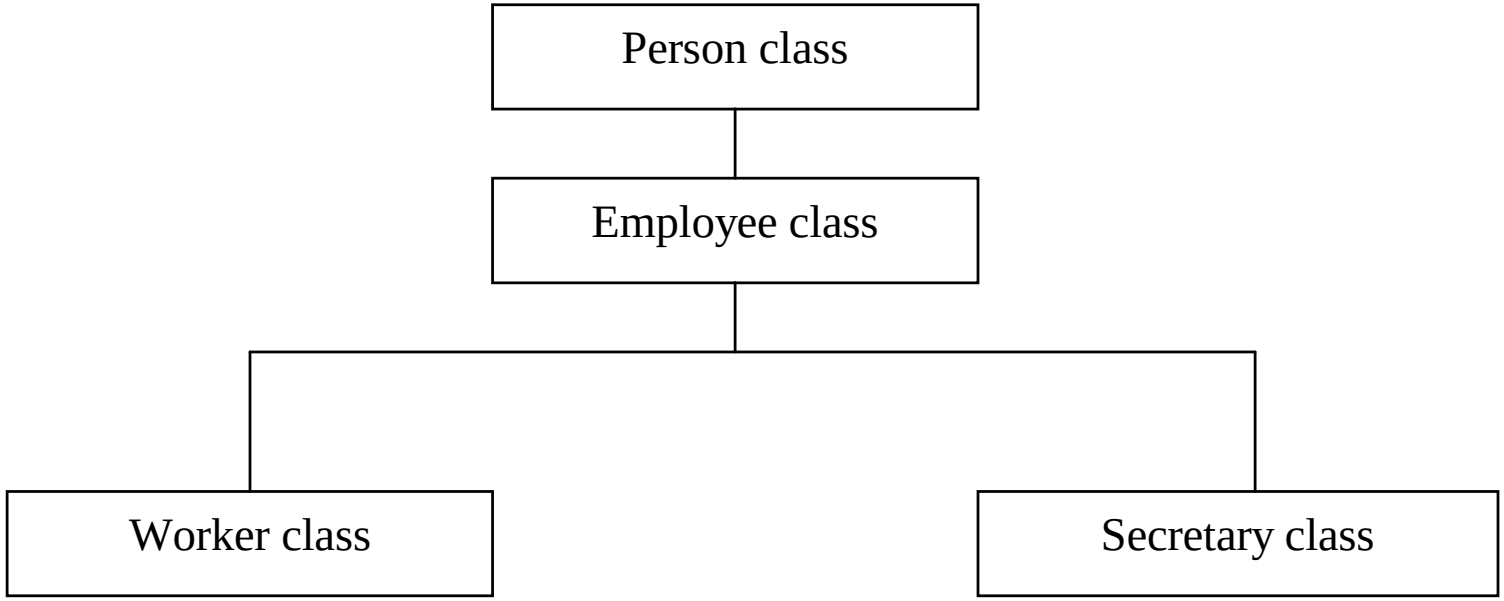
```
void Person::setFields(int num, char last[], char first[])
{
    idnum = num;
    strcpy(lastName, last);
    strcpy(firstName, first);
}
void Person::outputData( ){
    cout<< "ID#"<< idnum << " Name: "<< firstName << "  "<<
    lastName << endl;
}
class Customer:public Person
{
private:
   double balanceDue;
public:
   void setBalDue;
   void outputBalDue( );
};
```

```
void Customer::setBalDue(double bal)
{
      balanceDue = bal;
}
void Customer::outputBalDue()
{
     cout<< “Balance due $” << balanceDue<< endl;
}
void main()
{
    Customer cust;
    cust.setFields(215, “Santini”, “Linda”);
    cust.outputData();
    cust.setBalDue(147.95);
    cust.outputBalDue();
}
```

The output of the above program:

ID#215 Name: Linda Santini
Balance due $147.95

Class Hierarchy



Derived classes themselves can serve as base classes for other derived classes. When you build a series of base classes and derived classes, the chain of inherited classes is known as a *class hierarchy*

Class hierarchy (cont.)

- **Each class in a class hierarchy *cummulatively inherits* the class members of all classes that precede it in the hierarchy chain.**

- **A class that directly precedes another class in a class hierarchy, and is included in the derived class's base list, is called the *direct base class*.**

- **A class that does not directly precedes another class in a class hierarchy, and that not included in the derived class's base list, is called the *indirect base class*.**

Access Specifiers and Inheritance

- **Even though a derived class inherits the class members of a base class, the base class's members are still *bound* by its access specifiers.**

- **Private class members in the base class can be accessed only by the base class's member functions.**

- **For example, the *idNum* data member in the *Person* class is private. If you write the following member function in the *Customer* class, which attempts to directly access to the *idNum* data member, you will get a compiler error.**

```
void Customer::outputBalDue(){
    cout << " ID #"<< idNum<< " Balance due $"
         << balanceDue<<endl;
}
```

Proteced access specifier

- **You can declare the *idNum* data member with the *protected* access specifier.**

- **The *protected* access modifier restricts class member access to**
  1. **the class itself**
  2. **to classes derived from the class.**

- **The following code shows a modified version of the *Person* class declaration in which the private access modifier has been changed to *protected*.**

Example:

```
class Person {
protected:
    int idNum;
    char lastName[20];
    char firstName[15];
public:
    void setFields(int num, char last[], char first[]);
    void outputData();
};
```

- **A member function in *Customer* class that attempts to directly access to the *idNum* data member will work correctly since the *Customer* class is a derived class of the Person class and the *idNum* data member is now declared as *protected*.**

OVERRIDDING BASE CLASS MEMBER FUNCTIONS

- **Derived classes are not required to use a base class's member functions. You can write a more suitable version of a member function for a derived class when necessary.**

- **Writing a member function in a derived class to replace a base class member function is called *function overridding*.**

- **To overrid a base class function, the derived member function declaration must exactly *match* the base class member function declaration, including the function name, return type and parameters.**

- **To force an object of a derived class to use the base class version of an overriden function, you precede the function name with the base class name and the *scope resolution operator* using the syntax:**

  object.base_class::function();

- **<u>Example 8.4.2</u>**

  In the following code, the base class *Person* and the derived class *Employee* have their own function member with the same name *setFields().*

```
#include<iostream.h>
#include<string.h>
class Person
{
private:
   int idnum;
   char lastName[20];
   char firstName[15];
public:
   void setFields(int, char[], char[]);
   void outputData( );
};

void Person::setFields(int num, char last[], char first[])
{
    idnum = num;
    strcpy(lastName, last);
    strcpy(firstName, first);
}
```

```
void Person::outputData( )
{
    cout<< “ID#”<< idnum << “ Name: “<< firstName << “  “<< lastName <<
    endl;
}
class Employee: public Person
{
 private:
       int dept;
       double hourlyRate;
public:
       void setFields(int, char[], char[], int, double);
};
void Employee::setFields(int num, char last[], char first[], int dept,
                                                              double sal)
{
    Person::setFields(num, last, first);
    dept = dep; hourlyRate = sal;
}
```

```
void main()
{
    Person aPerson;
    aPerson.setFields(123, “Kroening”, “Ginny”);
    aPerson.outputData();
    cout<< endl<<endl;
    Employee worker;
    worker.Person::setFields(777, “Smith”, “John”);
    worket.outputData();
    worker.setFields(987,”Lewis”, “Kathy”, 6, 23.55);
    worker.outputData();
}
```

The output of the above program:

ID # 123 Name: Ginny Kroening
ID # 777 Name: John Smith
ID # 987 Name: Kathy Lewis

CONSTRUCTORS AND DESTRUCTORS IN DERIVED CLASSES

- **When you derive one class from another class, you can think of any instantiated object of the derived class as having two portions:**

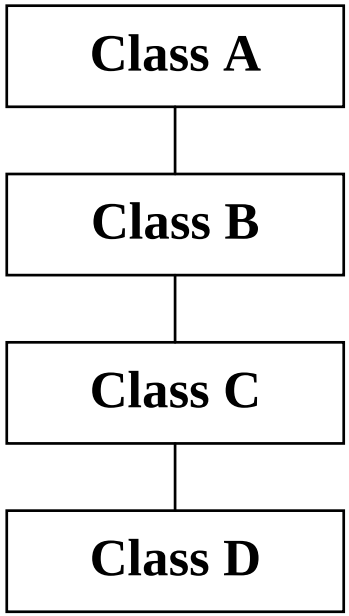
  - **the base class portion and**
  - **the derived class portion.**

- **During the instantiating process, the base class portion of the object is instantiated, and then the derived class portion of the object is instantiated.**

- **So, two constructors execute for a single derived class object: the base class constructor and the derived class constructor.**

- **When a derived class object instantiates, constructors begin executing at the top of the class hierarchy. First, the base constructor executes, then any indirect base class's constructors execute. Finally, the derived class' constructor executes.**

- **When an object is destroyed, class destructors are executed in the reverse order**



Class D object Instantiated

Order of Construction	Order of Destruction
1.Class A	1. Class D
2. Class B	2. Class C
3. Class C	3. Class B
4. Class D	4. Class A

Example 8.4.3

```
#include<iostream.h>
#include<string.h>
class Person
{
private:
   int idnum;
   char lastName[20];
   char firstName[15];
public:
   Person();
   void setFields(int, char[], char[]);
   void outputData( );
};
Person::Person(){
      cout << “Base class constructor call “<< endl;
}
```

```
void Person::setFields(int num, char last[], char first[])
{
    idnum = num;
    strcpy(lastName, last);
    strcpy(firstName, first);
}
void Person::outputData( )
{
    cout<< “ID#”<< idnum << “ Name: “<< firstName << “  “<< lastName <<
    endl;
}
class Customer: public Person
{
private:
   double balanceDue;
public:
   Customer();
   void setBalDue;
   void outputBalDue( );
};
```

```
Customer::Customer(){
    cout << “Derived constructor called” <<
     endl;
}
void Customer::setBalDue(double bal)
{
       balanceDue = bal;
}
void Customer::outputBalDue()
{
      cout<< “Balance due $”
     << balanceDue<< endl;
}
void main()
{
     Customer cust;
     cust.setFields(215, “Santini”, “Linda”);
     cust.outputData();
     cust.setBalDue(147.95);
     cust.outputBalDue();
}
```

The output of the above program:

Base class constructor called

Derived class constructor called

ID #215 Name: Linda Santini

Balance due $147.95