Chapter 8

OBJECT MANIPULATION - INHERITANCE

In the previous chapter, the declaration, initialization, and display of objects were presented. In this chapter, we continue our construction of classes by discussing how to write advanced constructors and destructors.

Besides, this chapter also discusses one of C++ most powerful features, inheritance. Inheritance permits the reuse and extension of existing code in a way that ensures the new code does not adversely affect what has already been written.

8.1 ADVANCED CONSTRUCTORS

In the last chapter, we already notice that constructors can provide a mechanism for initializing data members. However, 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.

8.1.1 Parameterized Constructors

Although constructor functions do not return values, they 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.

Example:

```
//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;
};
int main( ){
        Payroll employee;           //illegal
        ……
        return 0;
}
```

<u>Example 8.1.1</u>
The program in this example finds the distance between two points using the pointer to class objects technique.

```
//points
#include <iostream.h>
#include<math.h>
class point {
private:
   int x,y;
public:
    point( int xnew, int ynew);
    inline int getx(){
        return(x);
        }
    inline int gety(){
        return(y);
        }
    double finddist(point a, point b);
};
point::point(int xnew, ynew)          //parameterized constructor
{
  x = xnew;
  y = ynew;
}
double point::finddist(point a, point b)
{
        double temp;
        temp = ((b.y – a.y)*(b.y – a.y) + (b.x – a.x)*(b.x – a.x));
        return(sqrt(temp)):
}
int main()
{
        double value;
        point aobj(4,3), bobj(0, -1);

        value = aobj.finddist(aobj, bobj);
        cout << “Distance between two points = “<< value << endl;
        return 0;
}
```

The output of the above program:

```
Distance between two points = 5.656855
```

Constructor functions can be *overloaded*, just like other functions. This means that 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();
   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;
};
int main( ){
        Payroll employeeFL(0.28);
        Payroll employeeMA(0.28, 0.0595);
        return 0;
}
```

In the above example, the *Payroll* class has two parameterized constructor functions: one for states that have a state income tax and one for states that do not have a state income tax.

8.1.2 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.

Example: Given the simple constructor that assigns parameter values to the Payroll class.

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

You can use initialization list to rewrite the above constructor function.

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

8.1.3 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.

In the following 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 *Employe*e object, the default values in the constructor function 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;
}
int 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();
        return 0;
}
```

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
```

8.2 DESTRUCTORS

A *default destructor* cleans up any resources allocated to an object once the object is destroyed. The default constructor is sufficient for most classes, except when you have allocated memory on the heap.

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. A destructor accepts no parameter and returns no value.

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.

The destructor itself does not actually destroy the object – it performs termination house keeping before the system reclaims the object's memory so that memory may be reused to hold new objects.

Example 8.2.1

```
//Stocks_02.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 Stocks::setNumShares(int iShares){
        iNumShares = iShares;
}
int Stocks::getNumShares()const{
        return iNumShares;
}
void Stocks::setPricePerShare(double dPrice){
        dPricePerShare = dPrice;
}
int Stocks::getPricePerShare() const{
        return dPricePerShare;
}
void Stocks::setStockName(char* szName){
        strcpy(szStockName, szName);
}
char* Stock::getStockName() const{
        return szStockName;
}
double Stocks::calcTotalValue(){
        dCurrentValue = iNumShares*dPricePerShare;
        return dCurrentValue;
}
int 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;
        return 0;
}
```

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.
```

Notice that in the above program, the destructor function is called only once. 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, add the statement *delete stockPick2;* to the *main()* function as in the following program:

```
int 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;
        return 0;
}
```

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.
```

8.3 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 constant, place the *const* keyword in front of the object declaration.

Example:

```
        const Date currentDate;
```

Note: Constant data members in a class can not be assigned values using a standard assignment statement. Therefore, 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), dStateTax(0.05){
};
```

In contrast, the following code raises several compiler errors since constant data members must be initialized in an initialization list:

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
};
```

Constant Functions

Another good programming technique is to use the *const* keyword to declare *get* functions as *constant function*. *Get functions* are function members which do not modify data members.

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.

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

8.4 INHERITANCE

Inheritance is a form of software reusability in which new classes are created from existing classes by absorbing their attributes and behaviors, and overriding or embellishing these with capabilities the new classes require. Software reusability saves time in programming development. It encourages the reuse of proven and debugged high quality software, thus reducing problems after a system becomes functional.

8.4.1 <u>Basic Inheritance</u>

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

Often, classes do not have to be created "from scratch". Rather, they may be derived from other classes that provide attributes and behaviors the new classes can use. Such *software reuse* can greatly enhance programmer productivity.

Base Classes and Derived Classes

When you write a new class that inherits the characteristics of another class, you are said to be *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.

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
- overloaded assignment (=) functions

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

Consider a class originally developed by a company to hold an individual's data, such as an ID number and name. The class, named *Person*, contains three fields and two member functions.

```
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;
}
.
```
The company that uses the *Person* class soon realizes that the class can be used for all kinds of individuals – customers, full-time employees, part-time employees, and suppliers all have names and numbers as well. Now, the company wants to define the *Customer* class which inherits the members of the *Person* class.

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

```
class Customer: public Person

{

  ……// other statements go here
}
```

The access modifiers and base class names following the colon in a class's header declaration statement are called the *base list*. Here, the *public* inheritance is used since it is most common.

The *Customer* class contains all the members of *Person* because it inherits them. In other words, every *Customer* object has an *idNum*, l*astName* and *firstName*, just as a *Person* object does. Additionally, you can define the *Customer* class to include an additional data member, *balanceDue*, and two more functions: *setBalDue()* and *outputBalDue()*.

The base class *Person* and the derived class *Customer* can be graphically represented as in Figure 8.1.

The arrow in the above class diagram points from the derived class to the base class.

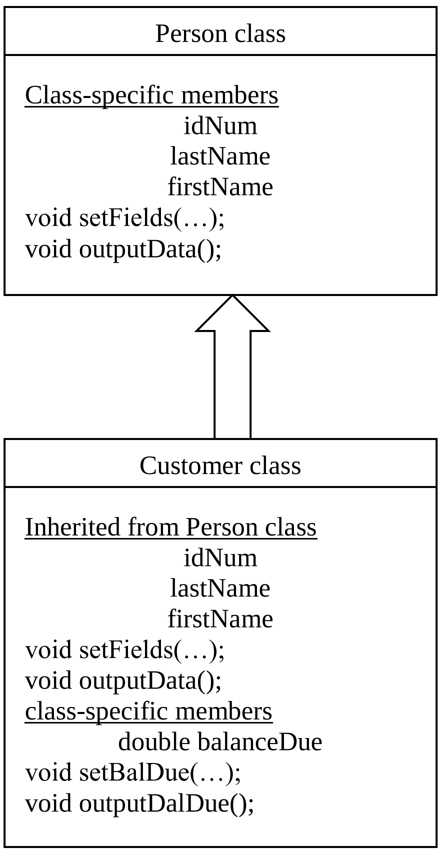


Figure 8.1 Base class and derived class

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

Example:

```
int main(){
        customer cust;
        cust.setField(123, “Richard”, “Leakey”);

        cust.outputData( );

        return 0;
}
```

The object *cust* which belongs to the class *Customer* can call the member functions *setFields()* and *outputData()* that belongs to the base class *Person*.

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;
}
int main()
{
    Customer cust;
    cust.setFields(215, “Santini”, “Linda”);
    cust.outputData();
    cust.setBalDue(147.95);
    cust.outputBalDue();
    return 0;
}
```

The output of the above program:

```
ID#215 Name: Linda Santini
```

```
Balance due $147.95
```

Of course, a *Customer* object can use its own class' member functions, *setBalDue()* and *outputBalDue().* Additionally, it can use the *Person* functions, *setFields()* and *outputData(),* as if they were its own.

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.*

Figure 8.2 shows a simple inheritance hierarchy. A typical company has hundreds of persons. An important subset of these persons is a set of employees. Employees are either workers or secretaries.

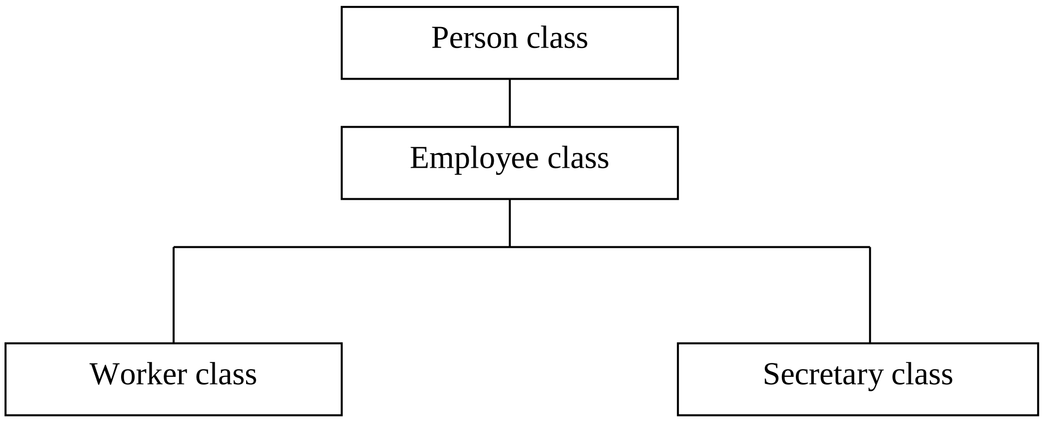


Figure 8.2 Person class hierarchy

Each class in a class hierarchy *cumulatively 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 precede 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 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;
        }
```

Instead, to access the *idNum* data member you must call the *Person* class's *outputData()* member function, which is public. Alternatively, 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, or

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*.

8.4.2 <u>Overriding Base Class Member Functions</u>

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 overriding*.

To override 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 overridden function, you precede the function name with the base class name and the *scope resolution operator* using the syntax:

object.base_class::function();

Example 8.4.2

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       // derived class
{
  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;
}
int main()
{
        Person aPerson;
        aPerson.setFields(123, “Kroening”, “Ginny”);
        aPerson.outputData();
        cout<< endl<<endl;
        Employee worker;
        worker.Person::setFields(777,”John”, “Smith”);
        worket.outputData();
        worker.setFields(987,”Lewis”, “Kathy”, 6, 23.55);
        worker.outputData();
        return 0;
}
```

The output of the above program:

```
ID # 123 Name:  Ginny Kroening

ID # 777 Name: John Smith

ID # 987 Name: Kathy Lewis
```

In the above program, when you use the Employee class to instantiate an Employee object with a statement such as *Employee worker;* and then the statement *worker.setFields();* uses the child function with the name *setFields()*. When used with a child class object, the child class function overrides the parent class version. On the other hand, the statement *worker.outputData();* uses the parent class function because no child class function has the name *outputData()*.

Overriding a base class member functions with a derived member function demonstrates the concept of polymorphism. Recall that polymorphism permits the same function name to take many forms.

8.4.3 <u>Constructors and Destructors in Derived Classes</u>

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.

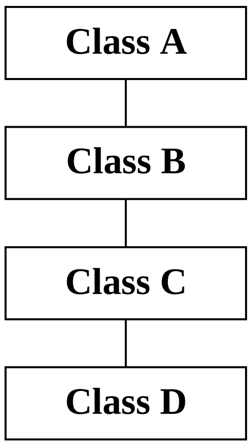
During the instantiating process, the base class portion of the object is instantiated, and then the derived class portion of the object is instantiated.

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. First, the derived class's destructor is called, then the destructors for any indirect base classes, and finally, the destructor for the base class. Figure 8.4 illustrates this process using a class hierarchy with four levels.

The order of construction makes sense, since it allows base classes to perform any initialization on class members that may be used by derived classes. And the order of destruction ensures that any base class members required by derived classes are not destroyed until all objects of any derived classes are destroyed first.



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**

Figure 8.3 Execution of constructors and destructors in a class hierarchy.

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;
}
int main()
{
      Customer cust;
      cust.setFields(215, “Santini”, “Linda”);
      cust.outputData();
      cust.setBalDue(147.95);
      cust.outputBalDue();
      return 0;
}
```

The output of the above program is:

```
Base class constructor called
Derived class constructor called

ID #215 Name: Linda Santini
```

```
Balance due $147.95
```

The output shows that both the constructor of *Person* class and the constructor of *Customer* class involve in creating the object *Cust*.