Inheritance

In this Week

- Inheritance: Extending classes
- Base class and Derived Class
- The IS-A Relationship
- Type Casting
- Member Function Overriding
- Arrays and Inheritance
- Pointers, References and Inheritance

Motivation

- Consider the following class designed to represent employees working in a company
- The design of the class consists of member variables, member functions and friend functions as usual
- In addition, we introduce the protected access specifier which will be discussed later
- We also have readInfo and printInfo member functions that will be called from their respective istream and ostream friend operator functions for reasons that will be discussed later
- Moreover since there are no any pointer member variables, we wouldn't really need any destructor, copy constructor or assignment operator overloaded. However we will have these functions for reasons that will be discussed later
- The implementation of the class together with a test main program and a sample run output are also provided in order to demonstrate the class design and its application

Motivation

```
class Employee
{
private:
    string firstName, lastName;
protected:
    double salary;
public:
    //Constructors
    Employee();
    Employee(const string&, const string&, const double&);
    Employee(const Employee&);
    //Destructor
    ~Employee();
    //Assignment operator
    Employee& operator=(const Employee&);
    //Getters
    string getFirstName() const;
    string getLastName() const;
    double getSalary() const;
    //Setters
    void setFirstName(const string&);
    void setLastName(const string&);
    void setSalary(const double&);
    //Additional member functions
    void readInfo(istream&);
    void printInfo(ostream&) const;
    //Friend functions
    friend istream& operator>>(istream&, Employee&);
    friend ostream& operator<<(ostream&, const Employee&);
};
```

Motivation

```
Employee::Employee() : firstName("N/A"), lastName("N/A"), salary(0.00)
{
    cout << "Inside employee default constructor"<< endl;
}
Employee::Employee(const string& f, const string& l, const double& s) : firstName(f), lastName(l), salary(s)
{
    cout << "Inside employee non-default constructor"<< endl;
}
Employee::Employee(const Employee& e) : firstName(e.firstName), lastName(e.lastName), salary(e.salary)
{
    cout << "Inside employee copy constructor"<< endl;
}
Employee::~Employee()
{
    cout << "Employee object destructed" << endl;
}
Employee& Employee::operator=(const Employee& e)
{
    firstName = e.firstName;
    lastName = e.lastName;
    salary = e.salary;
    return *this;
}
string Employee::getFirstName() const
{
    return firstName;
}
string Employee::getLastName() const
{
    return lastName;
}
double Employee::getSalary() const
{
    return salary;
}
```

Motivation

```
void Employee::setFirstName(const string& f)
{
    firstName = f;
}
void Employee::setLastName(const string& l)
{
    lastName = l;
}
void Employee::setSalary(const double& s)
{
    salary = s;
}
void Employee::readInfo(istream& in)
{
    cout << endl;
    cout << "\tEnter first name: ";
    in >> firstName;
    cout << "\tEnter last name: ";
    in >> lastName;
    cout << "\tEnter salary: ";
    in >> salary;
}
void Employee::printInfo(ostream& out) const
{
    out << endl;
    out << "\tFull Name = " << firstName << " " << lastName << endl;
    out << "\tSalary = " << salary << endl;
}
istream& operator>>(istream& in, Employee& e)
{
    e.readInfo(in);
    return in;
}
```

Motivation

```
ostream& operator<<(ostream& out, const Employee& e)
{
    e.printInfo(out);
    return out;
}

int main()
{
    Employee e1, e2("Jack", "Malcom", 1500.00);
    Employee e3(e2), e4;
    e4 = e2;
    cout << "Enter an employee ";
    cin >> e1;
    cout << "Employee 1 " << e1 << endl;
    cout << "Employee 2 " << e2 << endl;
    cout << "Employee 3 " << e3 << endl;
    cout << "Employee 4 " << e4 << endl;

    e1.~Employee();
    e2.~Employee();
    e3.~Employee();
    e4.~Employee();

    system("Pause");
    return 0;
}
```

```
Inside employee default constructor
Inside employee non-default constructor
Inside employee copy constructor
Inside employee default constructor
Enter an employee
        Enter first name: Tom
        Enter last name: Mark
        Enter salary: 1200
Employee 1
        Full Name = Tom Mark
        Salary = 1200

Employee 2
        Full Name = Jack Malcom
        Salary = 1500

Employee 3
        Full Name = Jack Malcom
        Salary = 1500

Employee 4
        Full Name = Jack Malcom
        Salary = 1500

Employee object destructed
Employee object destructed
Employee object destructed
Employee object destructed
Press any key to continue . . .
```

Motivation

- Next, consider a manager working in the same company
- We note that a manager in a company is also an employee of the company and thus we may be tempted to use the Employee class to represent manager objects as well
- However a manager may have additional attributes (member variables or member functions) in addition to those of the employee object
- For example, a manager may have certain number of employees under his/her leadership
- Thus we may design a class named Manager to represent manager objects as follows
- Only the class declaration is shown because its implementation is straight forward
- Once again, although there is no need for copy constructor, destructor and assignment operator member functions, they are still implemented for reasons to be discussed later

Motivation

```
class Manager
{
private:
    string firstName, lastName;
    int num; //number of subordinates
protected:
    double salary;
public:
    //Constructors
    Manager();
    Manager(const string&, const string&, const double&, const int&);
    Manager(const Manager&);
    //Destructor
    ~Manager();
    //Assignment operator
    Manager& operator=(const Manager&);
    //Getters
    string getFirstName() const;
    string getLastName() const;
    double getSalary() const;
    int getSubordinates() const;
    //Setters
    void setFirstName(const string&);
    void setLastName(const string&);
    void setSalary(const double&);
    void setSubordinates(const int&);
    //Additional member functions
    void readInfo(istream&);
    void printInfo(ostream&) const;
    //Friend functions
    friend istream& operator>>(istream&, Manager&);
    friend ostream& operator<<(ostream&, const Manager&);
};
```

Motivation

- We observe that in designing the Manager class, we are duplicating the Employee class code entirely with a few additional code that are specific to the Manager class
- We therefore ask ourselves: wouldn't it be more efficient to use the existing Employee class when we design the Manager class?
- Doing so would save us coding time and more importantly debugging time as well
- This is where **inheritance** comes to play its role and to help us reuse the Employee class code in the Manager class

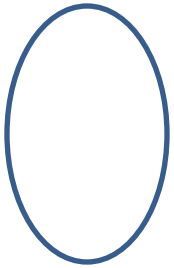
Inheritance

- **Inheritance** is the process by which an existing class is **EXTENDED** in order to create a new class
- For example, inheritance allows us to extend the Employee class in order to design a Manager class
- We say the new class (Manager class) inherits from the existing class (Employee class)
- The existing class is known as the **base class** or the **parent class** or the **super class**
- While the new class is known as the **derived class** or the **child class** or the **subclass**
- The most important feature of inheritance is that the derived class will **automatically inherit** all the member variables and all the member functions (with the exception of non-default constructors) of its base class without retyping them
- Inheritance has its own syntax as shown below

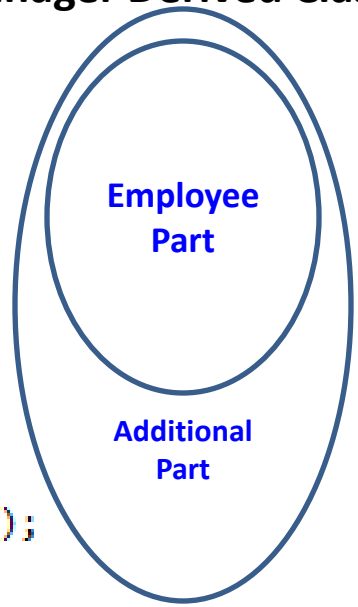
Inheritance

```
class Manager : public Employee
{
private:
    int num; //number of subordinates
public:
    //Constructors
    Manager();
    Manager(const string&, const string&, const double&, const int&);
    Manager(const Manager&);
    //Destructor
    ~Manager();
    //Assignment operator
    Manager& operator=(const Manager&);
    //Getters
    int getSubordinates() const;
    //Setters
    void setSubordinates(const int&);
    //Additional member functions
    void readInfo(istream&);
    void printInfo(ostream&) const;
    friend istream& operator>>(istream& in, Manager& m);
    friend ostream& operator<<(ostream& out, const Manager& m);
};
```

Employee Base Class



Manager Derived Class



Inheritance

- Now, the Manager class will automatically have the following member variables:
  - **firstName**, **lastName**, and **salary** (inherited from the Employee class) and
  - **num** (declared in the Manager class)
- Moreover, the Manager class will automatically have all the member functions of the Employee class except for the non-default constructors
- Last but not least, the Manager class redefines some of its inherited member functions to have its own different and correct implementation
- Now we proceed to implementing the member functions of the Manager class

Inheritance

- Before we proceed to implementation, let us first investigate the relationship between the derived class (Manager) and the base class (Employee)
- First of all we note that our Manager class is inheriting from the Employee class with a **public** access specifier
- This means any member variable or member function that is inherited from the base class and that was public in the base class will still be public in the base class and in the derived class (for example all the **getter** and **setter** member functions)
- Similarly any inherited member variable or function that was private in the base class will still remain private in the base class (for example **firstName** and **lastName**). This means these member variables, although inherited in the Manager class, are not accessible inside the Manager class
- Finally any inherited member variable or function that was protected in the base class will still remain protected in both the base and the derived classes (for example **salary**). This means such member variables, not only are they inherited in the derived class, but are also directly accessible in the derived class
- The implementation of the Manager class will therefore be as follows. A test main program and its output are also included in order to demonstrate the class design and some relationships between the derived class and the base class

Inheritance

```
Manager::Manager() : num(0)
{
    cout << "Inside manager default constructor" << endl;
    this->setFirstName("N/A");
    this->setLastName("N/A");
    this->salary = 0.00;
}
Manager::Manager(const string& f, const string& l, const double& s, const int& n) : num(n)
{
    cout << "Inside manager non-default constructor" << endl;
    this->setFirstName(f);
    this->setLastName(l);
    this->salary = s;
}
Manager::Manager(const Manager& m) : num(m.num)
{
    cout << "Inside manager copy constructor" << endl;
    this->setFirstName(m.getFirstName());
    this->setLastName(m.getLastName());
    this->salary = m.salary;
}
Manager::~Manager()
{
    cout << "Manager object destructed" << endl;
}
Manager& Manager::operator=(const Manager& m)
{
    this->setFirstName(m.getFirstName());
    this->setLastName(m.getLastName());
    this->salary = m.salary;
    num = m.num;
    return *this;
}
```

Inheritance

```
int Manager::getSubordinates() const
{
    return num;
}
void Manager::setSubordinates(const int& n)
{
    num = n;
}
void Manager::readInfo(istream& in)
{
    string temp;
    cout << endl;
    cout << "\tEnter first name: ";
    in >> temp;
    this->setFirstName(temp);
    cout << "\tEnter last name: ";
    in >> temp;
    this->setLastName(temp);
    cout << "\tEnter salary: ";
    in >> this->salary;
    cout << "\tEnter number of subordinates: ";
    in >> this->num;
}
void Manager::printInfo(ostream& out) const
{
    out << endl;
    out << "\tFull Name = " << this->getFirstName() << " " << this->getLastName() << endl;
    out << "\tSalary = " << this->salary << endl;
    out << "\tNumber of subordinates = " << this->num << endl;
}
```

Inheritance

```
istream& operator>>(istream& in, Manager& m)
{
    m.readInfo(in);
    return in;
}
ostream& operator<<(ostream& out, const Manager& m)
{
    m.printInfo(out);
    return out;
}

int main()
{
    Manager m1, m2("Sam", "Smith", 2800.00, 5);
    Manager m3(m2), m4;
    m4 = m2;
    cout << "Enter a manager ";
    cin >> m1;
    cout << "Manager 1 " << m1 << endl;
    cout << "Manager 2 " << m2 << endl;
    cout << "Manager 3 " << m3 << endl;
    cout << "Manager 4 " << m4 << endl;

    m1.~Manager();
    m2.~Manager();
    m3.~Manager();
    m4.~Manager();

    system("Pause");
    return 0;
}
```

```
Inside employee default constructor
Inside manager default constructor
Inside employee default constructor
Inside manager non-default constructor
Inside employee default constructor
Inside manager copy constructor
Inside employee default constructor
Inside manager default constructor
Enter a manager
        Enter first name: Jack
        Enter last name: Jones
        Enter salary: 3200
        Enter number of subordinates: 10
Manager 1
        Full Name = Jack Jones
        Salary = 3200
        Number of subordinates = 10

Manager 2
        Full Name = Sam Smith
        Salary = 2800
        Number of subordinates = 5

Manager 3
        Full Name = Sam Smith
        Salary = 2800
        Number of subordinates = 5

Manager 4
        Full Name = Sam Smith
        Salary = 2800
        Number of subordinates = 5

Manager object destructed
Employee object destructed
Manager object destructed
Employee object destructed
Manager object destructed
Employee object destructed
Manager object destructed
Employee object destructed
Press any key to continue . . .
```

Inheritance

- Observation
  - Every time a derived class object is constructed,
    - ✓ It will first automatically call the default constructor of its base class (unless we override this rule which will be discussed later) and then
    - ✓ It executes the constructor of its own class
  - Every time a derived class object is destructed,
    - ✓ It will first call its own destructor and then
    - ✓ It will automatically call the destructor of its base class
- This is useful from design point of view because every time an object of a derived class is constructed, it should construct both its inherited part and its own part in order to perform complete construction
- Similarly every time an object of a derived class is destructed, it should destruct both its own part and its inherited part in order to perform complete destruction

Calling Base Class Constructors From Derived Class

- Looking back on the Manager class constructors, we notice that we are duplicating the code in the base class which assign values to the inherited member variables in the derived class
- In fact, we could avoid such code duplications by calling the base class constructors from within the derived class constructors
- Base class constructors are called outside the block of derived class constructors
- The Manager class constructors modified to call the Employee base class constructors is shown below

Calling Base Class Constructors From Derived Class

```
Manager::Manager() : Employee(), num(0)
{
    cout << "Inside manager default constructor" << endl;
}
Manager::Manager(const string& f, const string& l, const double& s, const int& n) : Employee(f, l, s), num(n)
{
    cout << "Inside manager non-default constructor" << endl;
}
Manager::Manager(const Manager& m) : Employee(m.getFirstName(), m.getLastName(), m.salary), num(m.num)
{
    cout << "Inside manager copy constructor" << endl;
}
```

- With this modification, we run the same main program to see the effect of these changes
- This time the constructors of the derived class will not automatically call the default constructor of the base class because we are specifying which constructor of the base class we would like to invoke. See the output shown below

Calling Base Class Constructors From Derived Class

```
int main()
{
    Manager m1, m2("Sam", "Smith", 2800.00, 5);
    Manager m3(m2), m4;
    m4 = m2;
    cout << "Enter a manager ";
    cin >> m1;
    cout << "Manager 1 " << m1 << endl;
    cout << "Manager 2 " << m2 << endl;
    cout << "Manager 3 " << m3 << endl;
    cout << "Manager 4 " << m4 << endl;

    m1.~Manager();
    m2.~Manager();
    m3.~Manager();
    m4.~Manager();

    system("Pause");
    return 0;
}
```

```
Inside employee default constructor
Inside manager default constructor
Inside employee non-default constructor
Inside manager non-default constructor
Inside employee non-default constructor
Inside manager copy constructor
Inside employee default constructor
Inside manager default constructor
Enter a manager
        Enter first name: Jack
        Enter last name: Jones
        Enter salary: 3200
        Enter number of subordinates: 10
Manager 1
        Full Name = Jack Jones
        Salary = 3200
        Number of subordinates = 10

Manager 2
        Full Name = Sam Smith
        Salary = 2800
        Number of subordinates = 5

Manager 3
        Full Name = Sam Smith
        Salary = 2800
        Number of subordinates = 5

Manager 4
        Full Name = Sam Smith
        Salary = 2800
        Number of subordinates = 5

Manager object destructed
Employee object destructed
Manager object destructed
Employee object destructed
Manager object destructed
Employee object destructed
Manager object destructed
Employee object destructed
Press any key to continue . . .
```

The IS-A Relationship

- Every object of a derived class IS-A object of its base class
- Thus every Manager object is a Employee object
- This means a derived class object has more than one type: It is the derived class type as well as it is the base class type
- Thus every Manager object is a Manager type as well as Employee type
- This means a Manager object can be used in every place where an Employee object can be used
- Of course a base class object is not a derived class type

The IS-A Relationship

- The following program shows correct and incorrect use of the IS-A relationship between derived class and its base class

```
int main()
{
    Employee e1;
    Manager m1("Sam", "Smith", 2800.00, 5);

    e1 = m1; //Employee class assignment operator invoked
    Employee e2(m1); //Employee class copy constructor invoked
    e2.printInfo(cout); //Employee class printInfo member function invoked
    cout << e1 << endl; //Employee class overloaded ostream operator invoked

    cout << e2.getSubordinates() << endl; //Syntax error. Employee object can not be used in place of Manager object
    m1 = e1; //Syntax error. Employee object can not be used in place of Manager object

    system("Pause");
    return 0;
}
```

The IS-A Relationship

- With the help of automatic type casting, we may therefore rewrite the copy constructor of the Manager class as follows

```
Manager::Manager(const Manager& m) : Employee(m), num(m.num)
{
    cout << "Inside manager copy constructor" << endl;
}
```

- The same test main program and its sample run output is shown below in order to demonstrate the effect of this modification

The IS-A Relationship

```
int main()
{
    Manager m1, m2("Sam", "Smith", 2800.00, 5);
    Manager m3(m2), m4;
    m4 = m2;
    cout << "Enter a manager ";
    cin >> m1;
    cout << "Manager 1 " << m1 << endl;
    cout << "Manager 2 " << m2 << endl;
    cout << "Manager 3 " << m3 << endl;
    cout << "Manager 4 " << m4 << endl;

    m1.~Manager();
    m2.~Manager();
    m3.~Manager();
    m4.~Manager();

    system("Pause");
    return 0;
}
```

```
Inside employee default constructor
Inside manager default constructor
Inside employee non-default constructor
Inside manager non-default constructor
Inside employee copy constructor
Inside manager copy constructor
Inside employee default constructor
Inside manager default constructor
Enter a manager
        Enter first name: Jack
        Enter last name: Jones
        Enter salary: 3200
        Enter number of subordinates: 10
Manager 1
        Full Name = Jack Jones
        Salary = 3200
        Number of subordinates = 10

Manager 2
        Full Name = Sam Smith
        Salary = 2800
        Number of subordinates = 5

Manager 3
        Full Name = Sam Smith
        Salary = 2800
        Number of subordinates = 5

Manager 4
        Full Name = Sam Smith
        Salary = 2800
        Number of subordinates = 5

Manager object destructed
Employee object destructed
Manager object destructed
Employee object destructed
Manager object destructed
Employee object destructed
Manager object destructed
Employee object destructed
Press any key to continue . . .
```

Type Casting

- The IS-A relationship shows an automatic casting of a derived class object to a base class type
- In addition, we may explicitly perform type casting using static_cast
- The following program demonstrates type casting with static_cast
- The output of the program demonstrates the fact that a static_cast **constructs a new temporary object using copy constructor** for computation purposes

```
int main()
{
    Employee e1;
    Manager m1("Sam", "Smith", 2800.00, 5);

    e1 = static_cast<Employee>(m1); //Employee class assignment operator invoked
    Employee e2(static_cast<Employee>(m1)); //Employee class copy constructor invoked
    m1.printInfo(cout); //Manager class printInfo member function invoked
    static_cast<Employee>(m1).printInfo(cout); //Employee class printInfo member function invoked
    e2.printInfo(cout); //Employee class printInfo member function invoked
    cout << static_cast<Employee>(m1) << endl; //Employee class overloaded ostream operator invoked

    system("Pause");
    return 0;
}
```

Calling Base Class Member Functions From Derived Class

- Observe that the **assignment operator**, the **readInfo** and **printInfo** member functions of the Manager class are duplicating some code segments that already exist in their respective member functions in the Employee class
- It therefore makes sense to invoke the member functions in the base class to avoid such code duplication
- The **readInfo** member function of the Manager class modified to make use of the **readInfo** member function in the Employee class is shown below

```
void Manager::readInfo(istream& in)
{
    //this->readInfo(in); //This is runtime error. Explain.
    //static_cast<Employee>(*this).readInfo(in); //This is semantic error. Explain.
    //this->Employee.readInfo(in); //This is syntax error. Explain.
    this->Employee::readInfo(in); //This is correct
    cout << "\tEnter number of subordinates: ";
    in >> this->num;
}
```

Calling Base Class Member Functions From Derived Class

- Similarly, the modified **printInfo** member function and the modified **assignment operator** of the Manager class are shown below

```
void Manager::printInfo(ostream& out) const
{
    //this->printInfo(out); //This is runtime error. Explain.
    //static_cast<Employee>(*this).printInfo(out); //This is semantic error. Explain.
    //this->Employee.printInfo(out); //This is syntax error. Explain.
    this->Employee::printInfo(out); //This is correct
    out << "\tNumber of subordinates = " << this->num << endl;
}
Manager& Manager::operator=(const Manager& m)
{
    //*this = m; //This is runtime error. Explain.
    //this->operator=(m); //This is runtime error. Explain.
    //static_cast<Employee>(*this) = m; //This is semantic error. Explain.
    //this->Employee = m; //This is syntax error. Explain.
    //this->Employee.operator=(m); //This is syntax error. Explain.
    this->Employee::operator=(m); //This is correct
    num = m.num;
    return *this;
}
```

Member Function Overriding

- Recall that the Manager class automatically inherits the following member functions from the Employee class

  void readInfo(istream&);

  void printInfo(ostream&) const;

- But also the Manager class redefines the following member functions of its own

  void readInfo(istream&);

  void printInfo(ostream&) const;

- Now we observe that the newly redefined member functions have the same signature as the inherited member functions
- This means we have multiple functions with the same signature in the Manager class
- Is this allowed?
- Yes! Because whenever say for example **readInfo** member function is invoked, then the calling object will determine which **readInfo** member function is invoked
- If the calling object is an Employee object then the **readInfo** member function in the Employee class is invoked; if on the other hand the calling object is a Manager object then the **readInfo** member function in the Manager class is invoked
- If the Manager class had not redefined the **readInfo** member function then both the Employee objects and Manager objects would always have invoked the **readInfo** member function in the Employee class
- This is known as **member function overriding**

Arrays and Inheritance

- Since a derived class type object can be used in places where a base class object can be used, then we could create a static or dynamic array of base class type and populate it with base class type objects, derived class type objects or mixture of them
- However, the moment a derived class type object is stored in an element of the array, then it will automatically be casted to the base class type and at the end the array will store only base class type objects
- The following program demonstrates arrays and inheritance. The output of the program shows the array is filled with default objects during declaration and all the Manager objects assigned to elements of the array are automatically casted to Employee type. Thus each element will print an Employee type object

Arrays and Inheritance

```
int main()
{
    Employee E[10];
    cout << "Populating the array" << endl;
    for (int i = 0; i < 10; i++)
    {
        if (i % 2 == 0)
            E[i] = Employee();
        else
            E[i] = Manager();
    }
    cout << "Printing the elements of the array" << endl;
    for (int i = 0; i < 10; i++)
        cout << E[i] << endl;

    system("Pause");
    return 0;
}
```

Arrays and Inheritance

- Similarly we could use dynamic array and we will still get the same effect as demonstrated below

```
int main()
{
    Employee* E = new Employee[10];
    cout << "Populating the array" << endl;
    for (int i = 0; i < 10; i++)
    {
        if (i % 2 == 0)
            E[i] = Employee();
        else
            E[i] = Manager();
    }
    cout << "Printing the elements of the array" << endl;
    for (int i = 0; i < 10; i++)
        cout << E[i] << endl;

    delete[] E;

    system("Pause");
    return 0;
}
```

Arrays and Inheritance

- Here we should emphasize that even though the Manager objects have been lost during the assignment of Manager objects to the array elements (due to the automatic casting), but still the fact that we could create a base class type container (such as an array) and store in it both base class and derived class objects is a big achievement. This fact added to the fact that lots of code duplication have been avoided in derived classes is the main purpose of inheritance
- Next, we see a way to avoid automatic casting…

Pointers, References and Type Casting

- Now consider the following program and determine its output

```
int main()
{
    Manager m("Sam", "Smith", 2800.00, 1);

    Employee e1 = m;    //m is casted to Employee type. e1 is Employee object
    Employee* e2 = &m;  //m is not casted. e2 is pointing to a Manager object m on the main stack
    Employee* e3 = new Manager; //e3 is pointing to a default Manager object on the heap
    Employee* e4 = new Manager(); //e4 is pointing to a default Manager object on the heap
    Employee* e5 = new Manager(m); //e5 is pointing to a Manager object on the heap copied from m
    Employee& e6 = m;   //m is not casted. e3 is a reference to a Manager object m on the main stack

    cout << endl;
    cout << "e1 is ";   e1.printInfo(cout);
    cout << "*e2 is ";  e2->printInfo(cout);
    cout << "*e3 is ";  e3->printInfo(cout);
    cout << "*e4 is ";  e4->printInfo(cout);
    cout << "*e5 is ";  e5->printInfo(cout);
    cout << "e6 is ";   e6.printInfo(cout);

    cout << endl;
    cout << "e1 is " << e1 << endl;
    cout << "*e2 is " << *e2 << endl;
    cout << "*e3 is " << *e3 << endl;
    cout << "*e4 is " << *e4 << endl;
    cout << "*e5 is " << *e5 << endl;
    cout << "e6 is " << e6 << endl;

    delete e3; delete e4; delete e5;

    system("Pause");
    return 0;
}
```

Memory Leak!!! Only Employee class destructor is invoked

Pointers, References and Type Casting

- The object e1 is an Employee object and therefore calls the Employee class member and friend functions
- How about *e2, *e3, *e4, *e5 and e6 objects?
- Although e2, e3, e4, e5 and e6 are respectively pointing to and referencing Manager objects, they are calling the Employee class member and friend functions. Why?
- The reason is that e2, e3, e4, e5 and e6 are declared as Employee data types and therefore will always blindly call Employee class member and friend functions; irrespective of what actual objects they are pointing to or referencing
- But the worst part yet is the fact that destructing Manager objects with Employee pointers causes memory leak. Why? Because the compiler knows the pointers as Employee type and therefore will invoke the destructor of the Employee class only
- We conclude that whenever base class type pointers or references are used with derived class objects, they will always work on the base class type due to their **early binding** to base class data type

Parameter Passing to Functions

- We could also pass base class or derived class objects to functions and make all the parameters base class type
- In this case,
  - If parameter passing by value is used, then arguments will be copied and therefore casting will take place
  - If parameter passing by pointer or reference is used however, then no casting will take place but still the parameters will blindly call the base class member functions
- The following example demonstrates this effect

Parameter Passing to Functions

- Analyze the following program and determine its output

```
void foo(Employee e1, Employee e2, Employee* e3, Employee& e4)
{
    e1.printInfo(cout);
    e2.printInfo(cout);
    e3->printInfo(cout);
    e4.printInfo(cout);
}
int main()
{
    Employee a("a", "a", 1000);
    Manager b("b", "b", 1000, 5);
    Manager c("c", "c", 2000, 8);
    Manager d("d", "d", 3000, 10);
    foo(a, b, &c, d);

    system("Pause");
    return 0;
}
```

Inheritance Lineage

- It is perfectly fine for a class to be a subclass of another class which itself may be a subclass of another class
- For example suppose we have a class **A**. Then we can have a class **B** that is a subclass of **A**
- Moreover we can have a class **C** that is a subclass of **B** etc
- In that case a **B** object has two data types (**B** type and **A** type)
- Moreover a **C** object has three data types (**C** type, **B** type, and **A** type)
- That is every **B** object is an **A** object and every **C** object is a **B** object and is also an **A** object