Extending Classes (Chapter 15)

Issues:

- C++: Constraints that must be observed by a subclass constructor and destructor in relation to the base class constructor and destructor.
- C++: Special syntax of the subclass copy constructor, the subclass copy assignment operator, and any operators that we may wish to overload for a subclass.
- C++: Virtual functions.
- C++: Restrictions on overriding (virtual) functions.
- C++: Virtual Destructors
- C++: Abstract classes
- C++: Constructor order dependencies
• Java: Restrictions on overriding functions.
• Java: Abstract Classes
• Java: Interfaces
• Java: Extending interfaces
• Java: Constructor order dependencies
Public Derivation of a Subclass in C++

When we say

```cpp
class Y : public X { /* ..... */ }
```

we are declaring X to be a public base of Y.
A public derivation of a subclass buys us two things:

1. An object of derived-class type can at any time be treated as an object of base-class type without the use of explicit type conversion, provided both are manipulated through pointers or references.

Therefore, $Y^*$ can be substituted anywhere $X^*$ is needed.
2. With a public derivation, all of the public and protected data members and methods of the base class become available in the derived class.

It’s as if they were declared in the derived class itself.
Although public and protected data members and member functions of a base class are directly visible in a derived class, the identifiers used for them can be used again in a derived class.
The name conflict is avoided by using the following rule:

Say the same identifier is used to define a member function in a derived class as a public member function in a base class. The meaning of the identifier in the derived class will correspond to its definition in the derived class.

The derived class definition of a name hides all base class definitions of the same name.

If it is desired to also avail of the definition associated with the identifier in the base class, a scope operator must be used.
Speaking of function names in particular, the derived-class definition of a function hides *all* base-class functions of the *same name*, even when their signatures are different from that of the derived-class function.

Such hidden base-class function definitions can always be accessed in the derived class via the scope operator.
class FourLegged {
    double weight;

public:
    void print() { cout << weight << endl; }
};

class Cat : public FourLegged {
    int numClaws;

public:
    int getNumClaws();
    void print() {
        FourLegged::print();
        cout << getNumClaws() << endl;
    }
};
Since the private members of a base class are not visible in a derived class, one would naturally expect to be able to use those identifiers in a derived class. That is indeed the case.
```cpp
#include <iostream>
#include <string>

class User {
    string name; // given name
    int age; // actual age
public:
    User(string nam, int yy) : name(nam), age(yy) {}
    string getName() { return name; }
};

class UserStudent : public User {
    string name; // nick name at school
    int age; // assumed age for partying
public:
    UserStudent(string str1, int yy1, string str2, int yy2)
        : User(str1, yy1), name(str2), age(yy2) {}
    string getName() { return name; }
};

int main() {
    UserStudent us( "maryjo", 19, "jojo", 21);
    cout << us.getName() << endl;
    cout << us.User::getName() << endl;
}
```
Constructors for Derived Classes in C++

A derived class constructor must call a base call constructor before it does anything else.

If the base class constructor is not called explicitly in a derived-class constructor, the system will try to invoke the base-class’s no-arg constructor.

But, remember, a base class will have a no-arg constructor only if you provide one, or, by default, if you have defined no constructors at all for the base class.
class User {
    string name;
    int age;
public:
    User(string nm, int a) {name=nm; age=a;}
    void print() { cout << "Name: " << name << " Age: " << age; }
};

class UserStudent : public User {
    string schoolEnrolled;
public:
    UserStudent( string nam, int y, string school ) : User( nam, y ) {
        schoolEnrolled = school;
    }

    void print() {
        User::print();
        cout << " School Enrolled: " << schoolEnrolled << endl;
    }
};

int main()
{
    UserStudent us( "Pollyanna", 20, "ece" );
    us.print();
}
Through the initialization syntax, the constructor of a derived-class object first constructs the base-class portion of the derived class object.

After that, the derived class constructor proceeds to bring into existence what is defined explicitly for the derived class.
```cpp
#include<iostream>

class X {
    int m;

public:
    X():m(10){ cout << "inside X’s default constructor" << endl; }
    X(int mm): m(mm){}
    int getm() { return m; }
};

class Y : public X {
    int n;

public:
    Y(int nn) : n(nn) {}
    int getn() { return n; }
};

int main()
{
    Y yobj( 100 );
    int j = yobj.getm();
    int k = yobj.getn();
    cout << "m = " << j << " n = " << k << endl;
}
```
class X { /* */ };

class Y : public X { /* */ };

class Z : public Y { /* */ };

---

<p>| | |
| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>X sub-object</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td></td>
<td>Y sub-object</td>
</tr>
</tbody>
</table>

---

| Z Object |
Copy Constructors for Derived Classes in C++

#include <iostream>

class X {
    int m;
public:
    X( int mm ) : m( mm ) {}
    X( const X& other ) : m( other.m ) {}
    void print() { cout << "printing value of m of X obj: " << m << endl; }
};

class Y : public X {
    int n;
public:
    Y( int mm, int nn ) : X( mm ), n( nn ) {}
    Y( const Y& other ) : X( other ), n( other.n ) {}
    void print()
    {
        X::print();
        cout << "printing value of n of Y obj: " << n << endl;
    }
};
int main()
{
    X* xptr1 = new X(5);
    xptr1->print();

    cout << endl;

    Y y1(2, 3);
    y1.print();

    cout << endl;

    Y y2 = y1; // invokes copy constructor for Y
    y2.print();
}
class Y : public X1, public X2 {
    int n;
    //
};

Y( const Y& other ) : X1( other ), X2( other ), n( other.n ) { ... }
Assignment Operators for Derived Classes in C++

The assignment operator of a derived class must invoke the assignment operator of the base class.
#include <iostream>

class X {
    int m;
public:
    X( int mm ) : m( mm ) {}  
    X( const X& other ) : m( other.m ) {}  // copy constructor
    X& operator=( const X& other ) // assignment constructor
    {
        if ( this == &other ) return *this;
        m = other.m;
        return *this;
    }
    void print() { cout << "printing value of m of X obj: " << m << endl; }
};

class Y : public X {
    int n;
public:
    Y( int mm, int nn ) : X( mm ), n( nn ) {}  
    Y( const Y& other ) : X( other ), n( other.n ) {}  // copy constructor
    Y& operator=( const Y& other ) // assignment constructor
    {
        if ( this == &other ) return *this;
        X::operator=( other );
        n = other.n;
        return *this;
    }
    void print() {
    
}
X::print();
cout << "printing value of n of Y obj: " << n << endl; }
}

int main()
{
    X* xptr1 = new X( 5 );
xptr1->print();

cout << endl;

Y* yptr1 = new Y( 10, 11 );
yptr1->print();

cout << endl;

Y yobj = *yptr1;
yobj.print();
}
class Y : public X1, public X2 {
    int n;
    //
};

Y& operator=( const Y& other )
{
    if ( this == &other ) return *this;
    X1::operator=( other );          // (A)
    X2::operator=( other );          // (B)
    n = other.n;
    return *this;
}
Overloading Operators for Derived Classes in C++

Does anything different need to be done in the overload definition of an operator for a derived class if the same operator has been overloaded for the base class?

If you want the derived-class overload definition to use the base-class overload definition for doing part of the work, then you may have to use what is known as *upcasting*. 
```cpp
#include <iostream>
#include <string>
using namespace std;

class Person {
    string name;

public:
    Person( string nom ) : name( nom ) {} 
    Person( const Person& p ) : name( p.name ) {} 
    Person& operator=( const Person& p ) {
        if ( this != &p ) name = p.name;
        return *this;
    }
    virtual ~Person() {}
    friend ostream& operator<<( ostream& os, const Person& p );
};

// overload << for base class Person:
ostream& operator<<( ostream& os, const Person& p ) {
    os << p.name;
    return os;
}
```
class Employee: public Person {
    string department;
    double salary;

public:
    Employee( string name, string dept, double s ) : Person( name ), department( dept ), salary( s ) {}
    Employee( const Employee& e ) : Person( e ), department( e.department ),
                                   salary( e.salary ) {} 
    Employee& operator=( const Employee& e ) {
        if ( this != &e ) {
            Person::operator=( e );
            department = e.department;
            salary = e.salary;
        }
        return *this;
    }
    ~Employee() {}
    friend ostream& operator<<( ostream& os, const Employee& p );
};

// overload << for derived class Employee:
ostream& operator<<( ostream& os, const Employee& e ) {  // (E)
    const Person* ptr = &e;  // upcast
    os << *ptr;
    os << " " << e.department << " " << e.salary;
    return os;
}
class Manager: public Employee {
    string title;
public:
    Manager( string name, string dept, double salary, string atitle )
        : Employee( name, dept, salary),
        title( atitle ) {} 
    Manager( const Manager& m ) : Employee( m ), title( m.title ) {} 
    Manager& operator=( const Manager& m ){
        if ( this != &m ) {
            Employee::operator=( m );
            title = m.title;
        }
        return *this;
    }
    ~Manager() {} 
    friend ostream& operator<<( ostream& os, const Manager& m );
};

ostream& operator<<( ostream& os, const Manager& m ) { //(F)
    const Employee* ptr = &m; //upcast
    os << *ptr;
    os << " " << m.title;
    return os;
}

int main()
{
    Manager m1( "Zahpod", "assembly", 100, "director" );
}
Manager m2( m1 ); // invokes copy construct
cout << m2 << endl; // Zaphod assembly 100 director
Manager m3( "Trillion", "sales", 200, "vice_pres" );
m2 = m3; // invokes assignment oper
cout << m2 << endl; // Trillion sales 200 vice_pres
return 0;
}
Destructors for Derived Classes in C++

A destructor defined in a base class cannot be inherited by a derived class.

However, the base class destructor can be overridden in a derived class. The usefulness of overriding a base class destructor in a derived class is discussed separately in Section 15.10.
If resources are appropriated in a derived class in C++, then a destructor must be explicitly defined for the derived class in order to free up those resources when an object corresponding to the derived class goes out of scope or is explicitly deleted from the free store.
X sub-object

Memory appropriated by X

Y object
#include <iostream>
using namespace std;

class X {
public:
    int* x_data;
    int x_size;

    X( int* ptr, int sz ) : x_size(sz) {
        cout << "X’s constructor invoked" << endl;
        x_data = new int[ x_size ];
        int i=0;
        int* temp = x_data;
        while (i++<x_size)
            *temp++ = *ptr++;
    }

    ~X() {
        cout << "X’s destructor invoked" << endl;
        delete [] x_data;
    };
};
class Y : public X {
    int y;
public:
    Y( int* xptr, int xsz, int yy ) : X( xptr, xsz ), y( yy ) {
        cout << "Y's constructor invoked" << endl;
    }
};

int main()
{
    int freshData[100] = {0};
    Y* yptr = new Y( freshData, 100, 1000 );       // (A)
    delete yptr;                                  // (B)
}
X’s constructor invoked
Y’s constructor invoked
X’s destructor invoked

This output lends corroboration to the fact mentioned earlier that an object of a derived-class type is constructed base up, meaning that first its base sub-object is constructed and then the rest of the Y object is created. That’s why X’s constructor is invoked before Y’s constructor even though we are constructing a Y object.
Case 2:

X sub-object

Memory appropriated by X

Y object

Memory Appropriated by Y
//DestCase2.cc

#include <iostream>
using namespace std;

class X {
public:
    int* x_data;
    int x_size;
    X( int* ptr, int sz ) : x_size(sz) {
        cout << "X’s constructor invoked" << endl;
        x_data = new int[ x_size ];
        int i=0;
        int* temp = x_data;
        while (i++<x_size)
            *temp++ = *ptr++;
    }

    ~X() {
        cout << "X’s destructor invoked" << endl;
        delete [] x_data;
    }
};
class Y : public X {
    int* y_data;
    int y_size;
public:
    Y( int* xptr, int xsz, int* yptr, int yz)
    : X( xptr, xsz ), y_size( yz ) {
        cout << "Y’s constructor invoked" << endl;
        y_data = new int[ y_size ];
        int i=0;
        int* temp = y_data;
        while (i++<x_size)
            *temp++ = *yptr++;
    }
};

int main()
{
    int freshData[100] = {0};
    int moreFreshData[1000] = {1};
    Y* yptr = new Y( freshData, 100, moreFreshData, 1000 );  // (A)
    delete yptr;  // (B)
}
X’s constructor invoked
Y’s constructor invoked
X’s destructor invoked
Case 3:

//DestCase3.cc

#include <iostream.h>
using namespace std;

class X {
public:
    int* x_data;
    int x_size;
    X( int* ptr, int sz ) : x_size(sz) {
        cout << "X’s constructor invoked" << endl;
        x_data = new int[ x_size ];
        int i=0;
        int* temp = x_data;
        while (i++<x_size)
            *temp++ = *ptr++;
    }

    ~X() {
        cout << "X’s destructor invoked" << endl;
        delete [] x_data;
    }
};
class Y : public X {
    int* y_data;
    int y_size;
public:
    Y( int* xptr, int xsz, int* yptr, int yz) :
        X( xptr, xsz ), y_size( yz ) {
        cout << "Y’s constructor invoked" << endl;
        y_data = new int[ y_size ];
        int i=0;
        int* temp = y_data;
        while (i++<x_size)
            *temp++ = *yptr++;
    }

    ~Y() {
        cout << "Y’s destructor invoked" << endl;
        delete [] y_data;
    }
};

int main()
{
    int freshData[100] = {0};
    int moreFreshData[1000] = {1};
    Y* yptr = new Y( freshData, 100, moreFreshData, 1000 );
    delete yptr; // (A)
}
X’s constructor invoked
Y’s constructor invoked
Y’s destructor invoked
X’s destructor invoked
Virtual Functions in C++
class Employee {
    string firstName, lastName;
    //.....
public:
    Employee( string fnam, string lnam ) {
        firstName = fnam; lastName = lnam;
    }

    void print() const {
        cout << firstName << " " << lastName << endl;
    }
};

class Manager : public Employee {
    short level;
    //..
public:
    Manager( string fnam, string lnam, short lvl )
        : Employee( fnam, lnam ), level( lvl ) {}

    void print() const {
        Employee::print();
        cout << "level: " << level << endl;
    }
};
vector<Employee*> empList;

Employee* e1 = new Employee( "john", "doe" );
Employee* e2 = new Employee( "jane", "doe" );
Manager* e3 = new Manager( "mister", "bigshot", 2);
Manager* e4 = new Manager( "ms", "importante", 10);

Note that for the last two employees, we could also have said

Employee* e3 = new Manager( "mister", "bigshot", 2);
Employee* e4 = new Manager( "ms", "importante", 3);
empList.push_back(e1);
empList.push_back(e2);
empList.push_back(e3);
empList.push_back(e4);

vector<Employee*>::iterator p = empList.begin();
while (p < empList.end())
    (*p++)->print();

john doe
jane doe
mister bigshot
ms importante
If we want the system to automatically use `Manager::print()` for items of type `Manager*`, we have to declare the function `print()` as a `virtual` function in the base class.

```cpp
class Employee {
    string firstName, lastName;
    //.....

public:
    Employee( string fnam, string lnam ) {
        firstName = fnam;
        lastName = lnam;
    }

    virtual void print() const {
        cout << firstName << " " << lastName << endl;
    }
};
```

john doe
jane joe
mister bigshot at level: 2
ms importante at level: 3
Restrictions on Virtual Function Declarations

A requirement on a virtual function is that such a function must be defined for the class in which it is first declared, unless it is declared to be a pure virtual function.

Additionally, although a virtual function will typically be in the protected or the public sections of a class, since only those members are visible in a derived class, *it is legal and sometimes very useful to define a virtual function in the private section of a class.*
Virtual Functions in Multi-level Hierarchies:

Person
  /\  
 /   \  
----
  |
  |
  |
  |
  |
Employee
  /\  
 /   \  
----
  |
  |
  |
  |
  |
Manager
vector<Employee*> empList;

Employee* e1 = new Employee("mister", "bigshot", 2);
Employee* e2 = new Employee("ms", "importante", 3);
Manager* m3 = new Manager("mister", "biggun", 5, 2);
Manager* m4 = new Manager("ms", "shiningstar", 5, 2);

empList.push_back(e1);
empList.push_back(e2);
empList.push_back(m3);
empList.push_back(m4);
Will the `print()` function exhibit polymorphic behavior on this list of `Employee` objects even though the virtual declaration was made for the `Person` class?

The answer is yes. Once a function is declared to be virtual at any depth in a hierarchy, it will exhibit polymorphic behavior with respect to all classes at and below that level in the hierarchy.
Can Operators be Made to Behave Polymorphically?

Employee* e_ptr = new Manager( "ms", "importante", 3);
cout << *e_ptr;

Employee* e_ptr = new Manager( "ms", "importante", 3);
Manager* m_ptr = static_cast<Manager*>( e_ptr );
cout << *m_ptr;
Polymorphic Types:

When a class has at least one virtual function, the class defines a *polymorphic type*. 