

Rule To Follow Using Inheritance (Chapter 4)

Inheritance is private by default, which is most often not what you want. So be sure to specify the public keyword:

```
class Derived : public Base { ... };
```

Private data in a base class is not visible in a derived class. Use protected data and member functions to make data and functions visible to a derived class. You can put constructors and destructors in a protected class, which means that only a derived class can cause the Base class to be constructed.

```
class Base {
private:
    String name;
protected:
    Base(const String& s) { name = s; }
    ~Base();

public:
    ... // other methods
};

class Derived : public Base {
public:
    Derived(const String& s) : Base(s) {}
    ~Derived();
};
```

Rules To Follow Using Inheritance

A class with one or more pure virtual functions is called an “abstract” class. It is abstract because a pure virtual function has no implementation, so you can never instantiate an object of an abstract class. Some derived class must provide the implementation. Note that a virtual function in a derived class must have the same name, return type, and number and type of arguments as the overridden virtual function.

```
class Base {
public:
    ...
    virtual int f() = 0; // implementation must be provided by a derived class
};

class Derived_1 : public Base {
public:
    ... // no implementation for f(), so Derived_1 is still abstract
};

class Derived_2 : public Derived_1 {
public:
    // Note that if you leave off the virtual modifier, f() defaults to virtual anyway
    int f() { /* some implementation for f()*/ }
};
```

Rule To Follow Using Inheritance (Chapter 4)

The reason for declaring a function virtual in a class is so that a derived class can override the implementation of the function and have that implementation called when using a pointer of the type of the base class, which points to an instance of the derived class.

```
Shape* s = new Circle(1.0);
s->draw(); // call the Circle::draw method using a Shape pointer
```

When constructing an object that is an instance of a derived class, the base class constructor will be executed first, followed by the derived class constructor. Constructors cannot be virtual, but if you declare one or more member functions virtual, you should also declare the destructor as virtual.

```
class Base {
public:
    Base();
    virtual ~Base(); // destructor
    virtual int f();
};

class Derived : public Base {
public:
    Derived() : Base() {}
    ~Derived(); // virtual by default since ~Base is virtual

    virtual int f(); // override the implementation of f
};
```

Destructors are executed in bottom up fashion normally. If you destruct an object using a base class pointer, then the base class destructor will be executed unless it is declared virtual.

```
Base* p = new Derived();
...
delete p; // will call Derived::~Derived, but only if Base::~Base is declared virtual
```

Rules To Follow Using Inheritance

If the base class defines a friend function, then the friend access only applies to the base class, not a derived class.

```
class BankAccount {
private:
    int acct_id;
public:
    friend int customerID(BankAccount& ba);
    ...
};

class CheckingAccount : public BankAccount {
private:
    double balance;
public:
    ...
    void deposit (int key, double m);
    double withdraw (int key, double m);
    ...
};

int customerID(BankAccount& ba) {
    CheckingAccount& ca = (CheckingAccount&)ba;
    ca.balance += 10000000; // HA HA! I'm rich, rich, rich!!
    return ba.acct_id;
}

CheckingAccount you(0);
int id = customerID(you);
```

Rules To Follow Using Inheritance

Assignment operators are not inherited. You must define a new one in the derived class that has the appropriate type signature for the class.

```
class Shape {
protected:
    String name;
public:
    ...
    Shape& operator=(const Shape& s) {
        if (this != &s)
            name = s.name;
        return *this;
    }
    ...
};

class Rectangle : public Shape {
    double x,y;
public:
    ...
    Rectangle& operator=(const Rectangle& r) {
        if (this != &r) {
            x = r.x;
            y = r.y;
            Shape::operator=(r); // do base class assignment, or just name = r.name since
                                // name is protected
        }
        return *this;
    }
    ...
};
```

8/5/00

5 of 8

IS-A versus HAS-A Relationships

Consider the following:

```
Stack<int> s;
s.push(5);
s.push(6);
int x = s.remove_back();
int y = s.pop();
```

Question: What does it mean to call `remove_back()` on a stack object?

You could use private inheritance to hide the public interface of the inherited `List<T>`

```
template<class T> class Stack : private List<T> { ... };
```

Ask yourself whether or not it makes sense to ever use a `List<T>*` to refer to an instance of a `Stack<T>`?

```
List<int> *list = new Stack<T>();
list->insert_front(5); // a confusing way to push something on the stack, but it works.
```

When considering whether or not to inherit from some class, you also need to consider the methods that are already implemented in the interface of the class. Ask yourself whether or not it makes sense to have those methods called on an instance of the derived class. If not, then you should not use inheritance. You should use composition and define an instance of the type in the private section of your new class.

```
template<class T> class Stack {
private:
    List<T> list;
public:
    ...
};
```

8/5/00

7 of 8

IS-A versus HAS-A Relationships

Inheritance is not always the best way to reuse another class. You want to consider inheritance when the relationship between two classes can be expressed as an “is-a” relationship (e.g., a Circle is-a Shape, a Car is-a Vehicle, etc.) Most often, we define has-a relationships (e.g., a Shape has-a name, a Circle has-a radius). Sometimes, it is not clear whether or not to define an is-a or has-a relationship.

What about the following example where a `Stack<T>` template object inherits from a `List<T>` template:

```
template<class T> class List {
    T* list;
    int n;
public:
    List(int size) { list = new T[size]; }
    ~List() { delete list; }

    void insert_front(const T& x); // insert into front if not full
    T remove_front(); // remove from front if not empty
    void insert_back(const T& x); // insert into the back if not full
    T remove_back(); // remove from the back if not empty
};

template<class T> class Stack : public List<T> {
public:
    Stack(int size) : List<T>(size) {}
    ~Stack();

    void push (T x) { List<T>::insert_front(x); }
    T pop() { List<T>::remove_front(); }
};
```

Question: Does it make more sense to say that a `Stack<T>` is-a `List<T>` or that a `Stack<T>` has-a `List<T>` as part of the stack implementation?

8/5/00

6 of 8

Tomorrow - Question and Answer Session

Exam covers:

1. Appendix A and Chapters 1-4 in the book
2. All course lecture notes (get them online if you missed a class)
3. Homework assignments
4. Quizzes

I will give a handout with a number of questions that are not exam questions, but represent the kind of knowledge you should be able to demonstrate on the exam. Another good source of information are the Objects of the Game sections and the Common Errors sections at the end of each chapter.

You should bring your questions and I will provide the answers.

8/5/00

8 of 8