CS 520 Advanced Programming Languages
Fall Semester, 2009
Doc 11 C++ Virtual, Multiple Inheritance, Templates
Oct 12, 2009

Polymorphism

Non-Polymorphic

```
class Top {
      public:
            void name()
            {     std::cout << "Roger\n"; }
};

class Bottom :  public Top {
      public:
            void name()
            {     std::cout << "Whitney\n";}
};
```

```
int main() {
      Top topStack;
      topStack.name();              //Roger
      topStack = *(new Bottom);
      topStack.name();              //Roger

      Top* topHeap  = new Top;
      topHeap->name();              // Roger
      topHeap = new Bottom;
      topHeap->name();              // Roger
}
```

Polymorphism & Virtual

```
class Top {
      public:
            virtual void name()
            {     std::cout << "Roger\n"; }
};

class Bottom :  public Top {
      public:
            void name()
            {     std::cout << "Whitney\n";}
};
```

```
int main() {
      Top topStack;
      topStack.name();              //Roger
      topStack = *(new Bottom);
      topStack.name();              //Roger

      Top* topHeap  = new Top;
      topHeap->name();              // Roger
      topHeap = new Bottom;
      topHeap->name();              // Whitney
}
```

Pure Virtual = Abstract Class

```
class Abstract {
        public:
                virtual void a() = 0;
                virtual void b() = 0;
};
```

```
class Concrete : public Abstract {
public:
                virtual void a() { put real code here}
                void b() {more real code}
                };
```

What Happens Here?

```
class Top {
	public:
		virtual void name() {std::cout << "Top\n";}
};

class Middle : public Top {
	private:
		virtual void name() {std::cout << "Middle\n";}
};

class Bottom : public Middle {
	public:
		virtual void name() {std::cout << "Bottom\n";}
};
```

```
int main()
{
	Top* topHeap  = new
Middle;
	topHeap->name();
	topHeap = new Bottom;
	topHeap->name();
	Middle* middleHeap =
		new Middle;
	middleHeap->name();
}
```

Problem

```
#include <iostream>
using namespace std;

class Top {
      public:          ~Top()
                {cout << "Top\n";}
};


class Bottom : public Top{
      public:          ~Bottom()
          {cout << "Bottom\n";}
};
```

```
int main() {
      Top* pointer = new Bottom;
      delete pointer;
}
```

Output

Top

Solution

```
#include <iostream>
using namespace std;

class Top {
      public:          virtual ~Top()
                  {cout << "Top\n";}
};


class Bottom : public Top{
      public:          ~Bottom()
          {cout << "Bottom\n";}
};
```

```
int main() {
      Top* pointer = new Bottom;
      delete pointer;
}
```

Output

Bottom

Top

Rule of Thumb

If you have virtual functions

You need a virtual destructor

When Virtual is not Virtual

When virtual function member is called

Through an object

Using class scope operator

Called in constructor or destructor of base class

When Virtual is not Virtual

```
class Top {
public:
      virtual void  name() { std::cout << "Top\n"; }
      Top() {name();}
};

class Bottom :  public Top {
public:
      void name()  { std::cout << "Bottom\n";}
      Bottom() {name();}
};
```

```
int main() {
      Top topStack;
      Top* topHeap  = new Top;
      topHeap = new Bottom;
      topHeap->name();
      topHeap->Top::name();
}
```

Output

```
Top
Top
Top
Bottom
Bottom
Top
```

Virtual & Parameters

```
class Top {
public:
      virtual void  name() { std::cout << "Top\n"; }
};

class Bottom :  public Top {
public:
      void name()  { std::cout << "Bottom\n";}
};

void value( Top   aTop)   { aTop.name();    };

void reference(  Top&   aTop )  { aTop.name();   };
```

```
int main()
{
      Bottom test;
      value(test);          //Top
      reference(test);    //Bottom
}
```

Multiple Inheritance

Function Name Collision

```
class Left {
public:
	virtual void name() {cout << "left";}
};

class Right {
public:
	virtual void name() {cout << "Right";}
};

class Bottom : public Left, public Right {

};
```

```
int main() {
	Bottom *test = new Bottom;
	test->name();
	test->Left::name();
}
```

Data Member Name Collision

```
class Left {
public:
      int value;
      virtual int getValue() {return value;}
      Left() {value = -1;}
};

class Right {
public:
      int value;
      virtual int getValue() {return value;}
      Right() {value = 1;}
};

class Bottom : public Left, public Right {
public:
      int value;
      virtual int getValue() {return value;}
      Bottom() {value = 0;}
      };
```

```
int main() {
      Bottom *test = new Bottom;
      cout << test->getValue() << '\n';
      cout << test->Left::getValue() << '\n';
      cout << test->Right::getValue() << '\n';
}
```

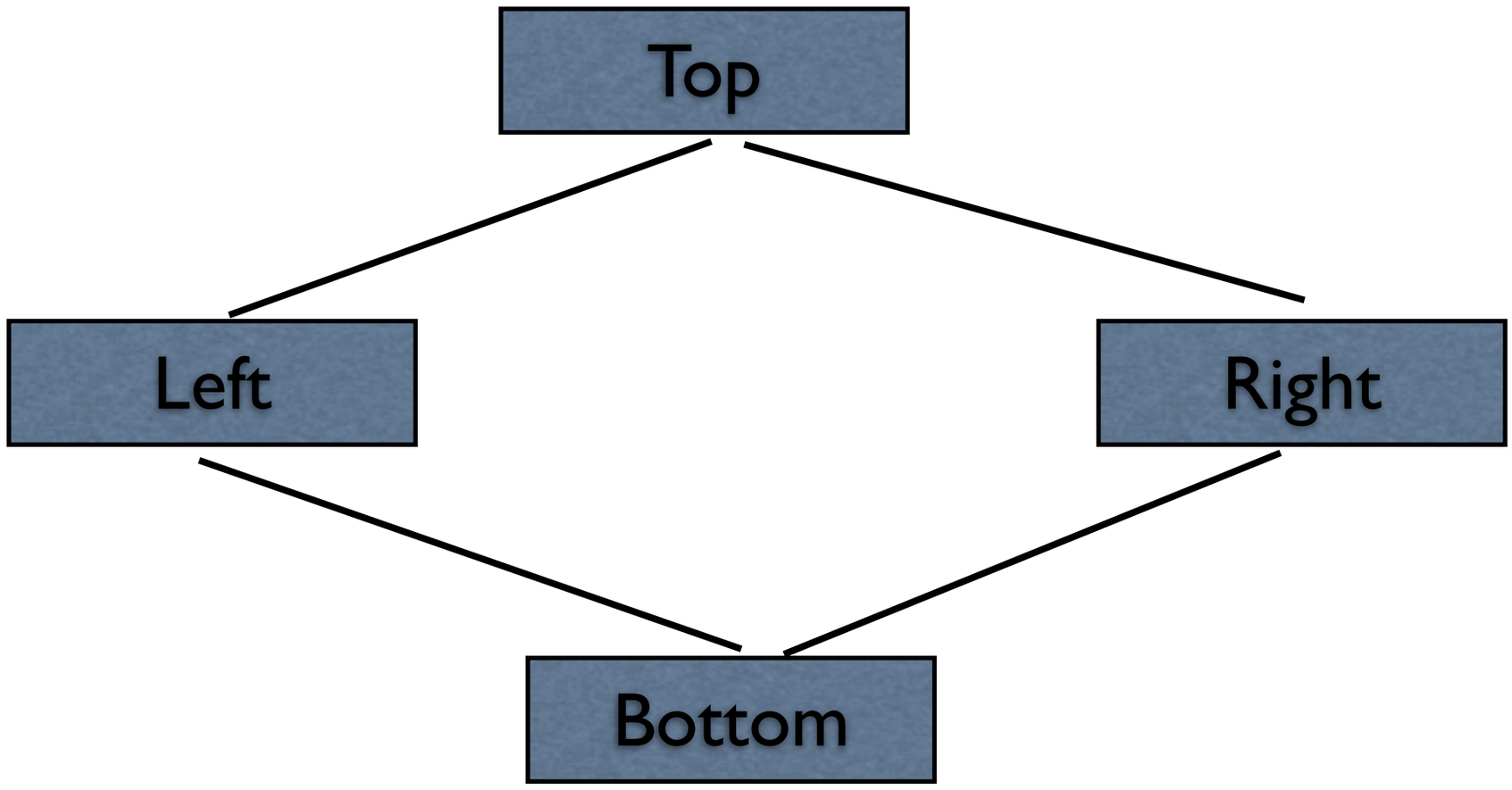
Ouput

0

-1

1

Common Ancesters



Inherit two Tops

```
class Top {
public:
      Top() {cout << "Top\n";}
};

class Left : public Top {
};

class Right  : public Top {
};

class Bottom : public Left, public Right {
};
```

```
int main() {
      Bottom *test = new Bottom;
}
```

Output

Top

Top

Virtual Base Class

```
class Top {
public:
      Top() {cout << "Top\n";}
};

class Left : virtual public Top {
};

class Right  : virtual public Top {
};

class Bottom : public Left, public Right {
};
```

```
int main() {
      Bottom *test = new Bottom;
}
```

Output

Top

All immediate subclasses of Top must declare themselves as inheriting Top virtually

Constructor Order

```
class Top {
public:         Top() {cout << "Top\n";}
};

class Left : virtual public Top {
public:         Left() {cout << "Left\n";}
};

class Right  : virtual public Top {
public:         Right() {cout << "Right\n";}
};

class Bottom : public Left, public Right {
public:
      Bottom():Right(), Left() {cout << "Bottom\n";}
};
```

```
int main() {
       Bottom test;
}
```

Output

```
Top
Left
Right
Bottom
```

Constructor Order

Virtual base class constructors are invoked first

Multiple Virtual base class constructors are invoked in the order they appear in the class definition.

Nonvirtual base class constructors are invoked in the order they appear in the class definition.

Member class constructors are invoked in the order of member class objects are declared.

Destructors are called in reverse order of the constructors.

Templates

Template Class

```
template <class Money>
class BankAccount  {
public :
      Money balance;
      BankAccount(Money amount = 0.0);
};

template <class Money>
BankAccount<Money>::BankAccount(Money amount) {
      balance = amount;
}
```

Using the Class

```
class Yen {
        friend ostream&  operator<<(ostream& , const Yen&);
public:
        Yen ( float StartAmount = 0) {value = StartAmount ;};
private:
        float value ;
};

ostream&  operator<<(ostream& out, const Yen& money) {
        out << "Yen: " << money.value;
        return out;
};
```

```
int main()
{
        BankAccount<int> me(10);
        BankAccount<Yen> you(200.0);
        std::cout
            << me.balance
            << you.balance;
}
```

Implicit Requirements

```
template <class Money>
class BankAccount  {
public :
      Money balance;
      void deposit(Money&);
      BankAccount(Money amount = 0.0);
};

template <class Money>
void BankAccount<Money>::deposit(Money& amount) {
      balance = balance + amount;
}
template <class Money>
BankAccount<Money>::BankAccount(Money amount) {
      balance = amount;
}
```

Multiple Parameters

```
template <class TypeA, class TypeB>
class Foo
{
        public :
                TypeA balance;
                TypeB amount;

                TypeB    bar( TypeA in );
};

template <class TypeA, class TypeB>
TypeB Foo<TypeA, TypeB> :: bar( TypeA in )
{
        balance = in;
        return amount;
}
```

Instantiation

```
template <class Money>
class BankAccount  {
public :
        Money balance;
        static int test;
        BankAccount(Money amount = 0.0);
};

template <class Money>
BankAccount<Money>::BankAccount(Money amount) {
        balance = amount;
        test = amount;
}

template <class Money>
int BankAccount<Money>::test = 0;
```

```
int main(){
        BankAccount<int> a(10);
        BankAccount<float> b(20);
        BankAccount<int> c(30);
        cout << a.test
                << '\n'
                << b.test
                << '\n'
                << c.test;
}
```

Output

```
30
20
30
```

Special Instances of Operations

```
template <class Type>
class Foo  {
public :
      void bar(Type input);
};

template <class Type>
void Foo<Type>::bar(Type input) {
      cout << "In general bar\n";
}
template<>
void Foo<int>::bar(int input) {
      cout << "In int bar\n";
}
```

```
int main(){
      Foo<int> x;
      Foo<float> y;
      x.bar(10);
      y.bar(10);
}
```

Output

```
In int bar
In general bar
```

Non-class Templates

```
template <int Size>
class Board
{
      public :
            int Squares[Size];
};
```

```
int main(){
      Board<100> large;          // OK

      const int y = 8;
      Board<y> normal;           // OK
      int x = 10;
      Board<x> normal;           // error  needs constant
}
```

Template Inheritance

```
template <class Type>
class Top {
      public:
            Type data;
            Top( Type value ) {
                  data = value;
                  cout << "Top Construct\n";
            };

            void setData( Type value) {
                  data = value;
                  cout << "In top\n";
            };
};

template <class Type>
class Bottom : public Top<Type>{
      public:
            Bottom( Type value ) : Top<Type>( value) {
                  cout << "Bottom Construct\n";};
};
```

```
int main(){
      Top<int> A( 1);
      Bottom<float> B( 3.3);
      B.setData( 5.5);
}
```

Regular Base and Template Derived Class

```
class Top {
      public:
            int data;
            Top( int value ) {
                  data = value;
                  cout << "Top Construct\n";
            };

            void setData( int value) {
                  data = value;
                  cout << "In top\n";};
};
```

```
template <class Type>
class Bottom : public Top   {
      public:
            Type NewData;

            Bottom( int A, Type value ) : Top(A) {
                  cout << "Bottom Construct\n";
                  NewData = value;};

            void setNewData( Type value) {
                  NewData = value;
                  cout << "In Bottom\n";};
};
```

Template Base and Regular Derived Class

```
template <class Type>
class Top {
      public:
            Type data;
            Top( Type value ) {
                  data = value;
                  cout << "Top Construct\n";
            };

            void setData( Type value) {
                  data = value;
                  cout << "In top\n"; };
};
```

```
class Bottom : public Top<int>   {
      public:
            float NewData;

            Bottom( int A, float value ) : Top<int> (A) {
                  cout << "Bottom Construct\n";
                  NewData = value;};

                        void setNewData( float value) {
                              NewData = value;
                              cout << "In Bottom\n";};
            };
```