CS 520 Advanced Programming Languages
Fall Semester, 2009
Doc 10 C++ Operators & Inheritance
Oct 7, 2009

Reading

Chapter 14 & 15 of C++ Primer, 4 Ed, Lippman, Lajoie, Moo

Overloadable Operators

+	-	*	/	%	^	&	|
~	!	,	=	<	>	<=	>=
++	--	<<	>>	==	!=	&&	||
+=	-=	/=	%=	^=	&=	|=	*=
<<=	>>=	[ ]	( )	->	->*	new	delete

Non-Overloadable Operators

::	.*	.	?:

Can Not Create New Operators

Precedence and Associativity

Programmer defined operator

Same precedence & associativity as built in operators

Operator as Member

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

class Foo {
private:
    int first;
public:
    void set_first(int value) { first = value;}
    int get_first() const  {return first;}
    int operator+(int value) {return first + value;}
    Foo(): first(2) {}
};
```

```
int main() {
    Foo test;
    int result = test + 10;
    cout << result << endl;
}
```

Operator as Function

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

class Foo {
private:
      int first;
public:
      void set_first(int value) { first = value;}
      int get_first() const  {return first;}
      Foo(): first(2) {}
};

int operator+(Foo leftArg, int value) {return leftArg.get_first() + value;}

int main() {
      Foo test;
      int result = test + 10;
      cout << result << endl;
}
```

Both?

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

class Foo {
private:
      int first;
public:
      void set_first(int value) { first = value;}
      int get_first() const  {return first;}
      int operator+(int value) {return first + value;}
      Foo(): first(2) {}
};

int operator+(Foo leftArg, int value) {return leftArg.get_first() + value;}

int main() {
      Foo test;
      int result = test + 10;  //compile error
      cout << result << endl;
}
```

Assignment Operator

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

class Foo {
private:
	int first;
public:
	void set_first(int value) { first = value;}
	int get_first() const  {return first;}
	Foo operator=(int value) {first = value; return *this;}
};

int main() {
	Foo test;
	test = 10;
	cout << test.get_first() << endl;
}
```

What happens here?

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

class Foo {
private:
      int first;
public:
      void set_first(int value) { first = value;}
      int get_first() const  {return first;}
      Foo operator=(int value) {first = value; return *this;}
};
```

```
int main() {
      Foo test = 10;
      cout << test.get_first() << endl;
}
```

Conversion Operator

```
class Foo {
private:
      int first;
public:
      void set_first(int value) { first = value;}
      int get_first() const  {return first;}

      operator int() const { return first;}

      Foo(): first(2) {}
};
```

```
int main() {
      Foo test;
      int result = test;
      cout << result << endl;
}
```

Implicit Conversion

```
class Foo {
private:
      int first;
public:
      void set_first(int value) { first = value;}
      int get_first() const  {return first;}

      operator int() const { return first;}

      Foo(): first(2) {}
};
```

```
int main() {
      Foo test;
      int fun = 5;
      int result;
      result = fun + test;
      double wild = test * 2.4;
      if (test)
            cout << "how\n";
      cout << wild << endl;
}
```

Ambiguities

```
class Foo {
private:
        int first;
public:
        void set_first(int value) { first = value;}
        int get_first() const  {return first;}

        operator int() const { return first;}
        operator double() const { return first;}

        Foo(): first(2) {}
};
```

```
int main() {
        Foo test;
        int fun = 5;
        int result = fun + test; //compile error
        double wild = test * 2.4; //compile
error
        if (test)                   /// compile error
                cout << "how\n";
        cout << wild << endl;
}
```

Casting

```
int main() {
        Foo test;
        int fun = 5;
        int result =  fun + (int) test;
        double wild = ((int)test) * 2.4;
        if ((double)test)
                cout << "how\n";
        cout << wild << endl;
}
```

More Ambiguities

```
class Foo {
private:
        int first;
public:
        void set_first(int value) { first = value;}
        int get_first() const  {return first;}
        operator int() const { return first;}
        operator double() const { return first;}
        Foo(): first(2) {}
};

void int_test(int bar) {}
void double_test(double bar) {}
void long_test(long double bar) {}
```

```
int main() {
        Foo test;
        int_test(test);
        double_test(test);
        long_test(test);            // compile
Error
}
```

Overload Functions

```
class Foo {
private:
        int first;
public:
        void set_first(int value) { first = value;}
        int get_first() const  {return first;}
        operator double() const { return first;}
        Foo(): first(2) {}
};

void test_this(int bar) {cout << "int\n";}
void test_this(double bar) {cout << "double\n";}
void test_this(long double bar) {cout << "long\n";}
```

```
int main() {
        Foo test;
        test_this(test);
}
```

Trouble

```
class Foo {
private:
      int first;
public:
      void set_first(int value) { first = value;}
      int get_first() const  {return first;}
      operator int() const { return first;}
      operator double() const { return first;}
      Foo(): first(2) {}
};

void test_this(int bar) {cout << "int\n";}
void test_this(double bar) {cout << "double\n";}
void test_this(long double bar) {cout << "long\n";}
```

```
int main() {
      Foo test;
      test_this(test);
}
```

Function Objects

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

class Foo {
private:
        int first;
public:
        void set_first(int value) { first = value;}
        Foo(): first(2) {}
        int operator() (int value) { return first + value;}
};
```

```
int main() {
        Foo test;
        int result = test(5);
        cout << result << endl;
}
```

Inheritance

Inheritance

```
class Account  {
public :
        float balance;
        transaction(float amount);
};

Account::transaction(float amount)
{     balance += amount;}
```

```
class Checking : public Account
{
      public:
            check(float amount);
};

Checking::check(float amount)
{
      transaction(-amount);
}
```

Terms

```
class Top {};
class Bottom : public Top {};
```

Base (Parent, or Super) Class

Class which is inherited from

Derived (Child, or Sub) Class

Class which inherits from one or more Base classes

What you Inherit

A derived class inherits all members of its base class except:

constructors

destructors

assignment operators

friends

Constructors

```
class Top {
public:
	int a;

	Top() {
		a = 5;
		cout << "In Top\n";
	}
};

class Bottom : public Top {
public:
	float b;
};
```

```
int main() {
	Bottom rung; // prints In Top
}
```

Calling Base Constructor

```
class Top {
public:
	int a;

	Top(int value) {
		a = value;
		cout << "In Top\n";
	}
};
```

```
class Bottom : public Top {
public:
	float b;
	Bottom(int first, int second):Top(first), b(second)
{
		cout << "In Bottom";
	}
};
```

Generates Warning

```
class Bottom : public Top {
public:
        float b;
        Bottom(int first, int second): b(second), Top(first) {
                cout << "In Bottom";
        }
};
```

Public, Protected, Private Base Classes

```
class Top{
        public:    int seeMe;
};

class Middle : private Top {
        public:
                void watchThis() {
                        seeMe = 10;
                }
};

class Bottom :  public Middle {
        public:
                void tryThisOut()  {
                        seeMe = 1;          // compile error
                }
};
```

```
int main() {
        Middle test;
        test.seeMe;  //compile error
}
```

Public, Protected, Private Base Classes

Public base class

inherited members maintain same access level

Protected base class

inherited public and protected members become protected members

Private base class

inherited public and protected members become private members

Exempting Individual Members

```
class Top{
public:
        int seeMe;
        int hideMe;
protected:
        int protectMe;
};

class Middle : private Top {
        public:
                Top::seeMe;
        protected:
                Top::protectMe;
};
```

```
class Bottom :  public Middle {
        public:
                void tryThisOut()  {
                        seeMe = 1;
                        hideMe = 2;  // compile error
                        protectMe = 3;
                }
};


int main() {
        Bottom test;
        test.seeMe = 4;   // ok
}
```

Inheritance is not Overloading

```
class Top{
public:
        void different(char *string)
        {       cout << "Roger\n"; }
};


class Bottom : public Top {
public:
        void different(int a)
        {       cout << "Whitney\n";}

        void same(char *string)
        {       cout << "Here\n";}

        void same(int a)
        {       cout << "There\n";}
};
```

```
int main() {
        Bottom test;
        test.same(5);                   // prints
There
        test.same("Hi Mom");        // prints Here
        test.different(5);          // prints Whitney
        test.different("Hi Dad");       // illegal
}
```

Inheritance and Types

```
class Top     {
public :
      int Hi;
};

class Bottom :  public Top {
public:
      int Bye;
};
```

```
int main() {
      Top        a;
      Bottom z;

      a = z;
      z = a;  // compile error
}
```

With Pointers

```
int main() {
        Top *a = new Top;
        Bottom *z = new Bottom;

        a = z;
        z = a;                          //Still compile error
        z = static_cast<Bottom*>(a);
        z = dynamic_cast<Bottom*>(a);
        z = reinterpret_cast<Bottom*>(a);
        z = (Bottom*) a;
}
```

Inherited Methods

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


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

```
int main() {
       Top a;
       Bottom z;
       a.name();            //Roger
       z.name();            //
Whitney
       a = z;
       a.name();            //Roger
}
```

With Pointers

```
int main() {
        Top*      a = new Top;
        Bottom* z = new Bottom();
        a->name();                      //Roger
        z->name();                      //Whitney
        a = z;
        a->name();                      //Roger
}
```

Virtual

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

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

```
int main() {
      Top*       a = new Top;
      Bottom* z = new Bottom();
      a->name();                  //Roger
      z->name();                  //Whitney
      a = z;
      a->name();                  //Whitney
      Top b;
      Bottom y;
      b = y;
      b.name();                        //Roger
}
```

Be Careful

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

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

void javaLike(Top& value) {value.name();}
void notJavaLike(Top value) {value.name();}
```

```
int main() {
        Bottom z;
        javaLike(z);          //Whitney
        notJavaLike(z);    //Roger
}
```