Chapter 15

POLYMORPHISM AND VIRTUAL FUNCTIONS

***Listing 15-1. Illustrating Inheritance and the Upcasting Problem***

**//: C15:Instrument2.cpp**

**// Inheritance & upcasting**

**#include <iostream>**

**using namespace std;**

**enum note { middleC, Csharp, Eflat }; // Etc.**

**class Instrument {**

**public:**

**void play(note) const {**

**cout << "Instrument::play" << endl;**

**}**

**};**

**// Wind objects are Instruments**

**// because they have the same interface:**

**class Wind : public Instrument {**

**public:**

**// Redefine interface function:**

**void play(note) const {**

**cout << "Wind::play" << endl;**

**}**

**};**

**void tune(Instrument &i) {**

**// ...**

**i.play(middleC);**

**}**

**int main() {**

**Wind flute;**

**tune(flute); // Upcasting**

**} ///:~**

***Listing 15-2. Illustrating Late Binding with the virtual Keyword***

**//: C15:Instrument3.cpp**

**// Late binding with the virtual keyword**

**#include <iostream>**

**using namespace std;**

**enum note { middleC, Csharp, Cflat }; // Etc.**

**class Instrument {**

**public:**

**virtual void play(note) const {**

**cout << "Instrument::play" << endl;**

**}**

**};**

**// Wind objects are Instruments**

**// because they have the same interface:**

**class Wind : public Instrument {**

**public:**

**// Override interface function:**

**void play(note) const {**

**cout << "Wind::play" << endl;**

**}**

**};**

**void tune(Instrument &i) {**

**// ...**

**i.play(middleC);**

**}**

**int main() {**

**Wind flute;**

**tune(flute);**

**// Upcasting**

**} ///:~**

***Listing 15-3. Illustrating Extensibility in OOP***

**//: C15:Instrument4.cpp**

**/// Extensibility in OOP**

**#include <iostream>**

**using namespace std;**

**enum note { middleC, Csharp, Cflat }; // Etc.**

**class Instrument {**

**public:**

**virtual void play(note) const {**

**cout << "Instrument::play" << endl;**

**}**

**virtual char\* what() const {**

**return "Instrument";**

**}**

**// Assume this will modify the object:**

**virtual void adjust(int) {}**

**};**

**class Wind : public Instrument {**

**public:**

**void play(note) const {**

**cout << "Wind::play" << endl;**

**}**

**char\* what() const { return "Wind"; }**

**void adjust(int) {}**

**};**

**15: Polymorphism & Virtual Functions 667**

**class Percussion : public Instrument {**

**public:**

**void play(note) const {**

**cout << "Percussion::play" << endl;**

**}**

**char\* what() const { return "Percussion"; }**

**void adjust(int) {}**

**};**

**class Stringed : public Instrument {**

**public:**

**void play(note) const {**

**cout << "Stringed::play" << endl;**

**}**

**char\* what() const { return "Stringed"; }**

**void adjust(int) {}**

**};**

**class Brass : public Wind {**

**public:**

**void play(note) const {**

**cout << "Brass::play" << endl;**

**}**

**char\* what() const { return "Brass"; }**

**};**

**class Woodwind : public Wind {**

**public:**

**void play(note) const {**

**cout << "Woodwind::play" << endl;**

**}**

**char\* what() const { return "Woodwind"; }**

**};**

**// Identical function from before:**

**void tune(Instrument& i) {**

**// ...**

**i.play(middleC);**

**}**

**// New function:**

**void f(Instrument& i) { i.adjust(1); }**

**// Upcasting during array initialization:**

**668 Thinking in C++ www.BruceEckel.com**

**Instrument\* A[] = {**

**new Wind,**

**new Percussion,**

**new Stringed,**

**new Brass,**

**};**

**int main() {**

**Wind flute;**

**Percussion drum;**

**Stringed violin;**

**Brass flugelhorn;**

**Woodwind recorder;**

**tune(flute);**

**tune(drum);**

**tune(violin);**

**tune(flugelhorn);**

**tune(recorder);**

**f(flugelhorn);**

**} ///:~*****Listing 15-4. Illustrating Comparison of Object Sizes***

***(with and without virtual functions)***

**//: C15:Sizes.cpp**

**/// Object sizes with/without virtual functions**

**#include <iostream>**

**using namespace std;**

**class NoVirtual {**

**int a;**

**public:**

**void x() const {}**

**int i() const { return 1; }**

**};**

**class OneVirtual {**

**int a;**

**public:**

**virtual void x() const {}**

**int i() const { return 1; }**

**};**

**class TwoVirtuals {**

**int a;**

**public:**

**virtual void x() const {}**

**virtual int i() const { return 1; }**

**};**

**int main() {**

**cout << "int: " << sizeof(int) << endl;**

**cout << "NoVirtual: "**

**<< sizeof(NoVirtual) << endl;**

**cout << "void\* : " << sizeof(void\*) << endl;**

**cout << "OneVirtual: "**

**<< sizeof(OneVirtual) << endl;**

**cout << "TwoVirtuals: "**

**<< sizeof(TwoVirtuals) << endl;**

**} ///:~*****Listing 15-5. Illustrating Early Binding and Virtual Functions***

**//: C15:Early.cpp**

**// Early binding & virtual functions**

**#include <iostream>**

**#include <string>**

**using namespace std;**

**class Pet {**

**public:**

**virtual string speak() const { return ""; }**

**};**

**class Dog : public Pet {**

**public:**

**string speak() const { return "Bark!"; }**

**};**

**int main() {**

**Dog ralph;**

**Pet\* p1 = &ralph;**

**Pet& p2 = ralph;**

**Pet p3;**

**// Late binding for both:**

**cout << "p1->speak() = " << p1->speak() << endl;**

**cout << "p2.speak() = " << p2.speak() << endl;**

**// Early binding (probably):**

**cout << "p3.speak() = " << p3.speak() << endl;**

**} ///:~**

***Listing 15-6. Illustrating a Pure Abstract Class***

**//: C15:Instrument5.cpp**

**// Pure abstract base classes**

**#include <iostream>**

**using namespace std;**

**enum note { middleC, Csharp, Cflat }; // Etc.**

**class Instrument {**

**public:**

**// Pure virtual functions:**

**virtual void play(note) const = 0;**

**virtual char\* what() const = 0;**

**// Assume this will modify the object:**

**virtual void adjust(int) = 0;**

**};**

**// Rest of the file is the same ...**

**class Wind : public Instrument {**

**public:**

**void play(note) const {**

**cout << "Wind::play" << endl;**

**}**

**char\* what() const { return "Wind"; }**

**void adjust(int) {}**

**};**

**class Percussion : public Instrument {**

**public:**

**void play(note) const {**

**cout << "Percussion::play" << endl;**

**}**

**char\* what() const { return "Percussion"; }**

**void adjust(int) {}**

**};**

**class Stringed : public Instrument {**

**public:**

**void play(note) const {**

**cout << "Stringed::play" << endl;**

**}**

**char\* what() const { return "Stringed"; }**

**void adjust(int) {}**

**};**

**class Brass : public Wind {**

**public:**

**void play(note) const {**

**cout << "Brass::play" << endl;**

**}**

**char\* what() const { return "Brass"; }**

**};**

**class Woodwind : public Wind {**

**public:**

**void play(note) const {**

**cout << "Woodwind::play" << endl;**

**}**

**char\* what() const { return "Woodwind"; }**

**};**

**// Identical function from before:**

**void tune(Instrument&i) {**

**// ...**

**i.play(middleC);**

**}**

**// New function:**

**void f(Instrument&i) { i.adjust(1); }**

**int main() {**

**Wind flute;**

**Percussion drum;**

**Stringed violin;**

**Brass flugelhorn;**

**Woodwind recorder;**

**tune(flute);**

**tune(drum);**

**tune(violin);**

**tune(flugelhorn);**

**tune(recorder);**

**f(flugelhorn);**

**} ///:~**

***Listing 15-7. Illustrating Pure Virtual Definitions***

**//: C15:PureVirtualDefinitions.cpp**

**// Pure virtual base definitions**

**#include <iostream>**

**using namespace std;**

**class Pet {**

**public:**

**virtual void speak() const = 0;**

**virtual void eat() const = 0;**

**// Inline pure virtual definitions illegal:**

**//! virtual void sleep() const = 0 {}**

**};**

**// OK, not defined inline**

**void Pet::eat() const {**

**cout << "Pet::eat()" << endl;**

**}**

**void Pet::speak() const {**

**cout << "Pet::speak()" << endl;**

**}**

**class Dog : public Pet {**

**public:**

**// Use the common Pet code:**

**void speak() const { Pet::speak(); }**

**void eat() const { Pet::eat(); }**

**};**

**int main() {**

**Dog simba; // Richard's dog**

**simba.speak();**

**simba.eat();**

**} ///:~**

***Listing 15-8. Illustrating Addition of Virtual Functions in the Derived Class***

**//: C15:AddingVirtuals.cpp**

**// Adding virtuals in derivation**

**#include <iostream>**

**#include <string>**

**using namespace std;**

**class Pet {**

**string pname;**

**public:**

**Pet(const string &petName) : pname(petName) {}**

**virtual string name() const { return pname; }**

**virtual string speak() const { return ""; }**

**};**

**class Dog : public Pet {**

**string name;**

**public:**

**Dog(const string &petName) : Pet(petName) {}**

**// New virtual function in the Dog class:**

**virtual string sit() const {**

**return Pet::name() + " sits";**

**}**

**string speak() const { // Override**

**return Pet::name() + " says 'Bark!'";**

**}**

**};**

**int main() {**

**Pet\* p[] = {new Pet("generic"),new Dog("bob")};**

**cout << "p[0]->speak() = "**

**<< p[0]->speak() << endl;**

**cout << "p[1]->speak() = "**

**<< p[1]->speak() << endl;**

**//! cout << "p[1]->sit() = "**

**//! << p[1]->sit() << endl; // Illegal**

**} ///:~**

***Listing 15-9.Illustrating Object Slicing***

**//: C15:ObjectSlicing.cpp**

**#include <iostream>**

**#include <string>**

**using namespace std;**

**class Pet {**

**string pname;**

**public:**

**Pet(const string& name) : pname(name) {}**

**virtual string name() const { return pname; }**

**virtual string description() const {**

**return "This is " + pname;**

**}**

**};**

**class Dog : public Pet {**

**string favoriteActivity;**

**public:**

**Dog(const string& name, const string& activity)**

**: Pet(name), favoriteActivity(activity) {}**

**string description() const {**

**return Pet::name() + " likes to " +**

**favoriteActivity;**

**}**

**};**

**void describe(Pet p) { // Slices the object**

**cout << p.description() << endl;**

**}**

**int main() {**

**Pet p("Bob");**

**Dog d("Peter", "sleep");**

**describe(p);**

**describe(d);**

**} ///:~**

***Listing 15-10. Demonstrating that Virtual Functions Restrict Overloading***

**//: C15:NameHiding2.cpp**

**// Virtual functions restrict overloading**

**#include <iostream>**

**#include <string>**

**using namespace std;**

**class Base {**

**public:**

**virtual int f() const {**

**cout << "Base::f()\n";**

**return 1;**

**}**

**virtual void f(string) const {}**

**virtual void g() const {}**

**};**

**class Derived1 : public Base {**

**public:**

**void g() const {}**

**};**

**class Derived2 : public Base {**

**public:**

**// Overriding a virtual function:**

**int f() const {**

**cout << "Derived2::f()\n";**

**return 2;**

**}**

**};**

**class Derived3 : public Base {**

**public:**

**// Cannot change return type:**

**//! void f() const{ cout<< "Derived3::f()\n";}**

**};**

**class Derived4 : public Base {**

**public:**

**// Change argument list:**

**int f(int) const {**

**cout << "Derived4::f()\n";**

**return 4;**

**}**

**};**

**int main() {**

**string s("hello");**

**Derived1 d1;**

**int x = d1.f();**

**d1.f(s);**

**Derived2 d2;**

**x = d2.f();**

**//! d2.f(s); // string version hidden**

**Derived4 d4;**

**x = d4.f(1);**

**//! x = d4.f(); // f() version hidden**

**//! d4.f(s); // string version hidden**

**Base &br = d4; // Upcast**

**//! br.f(1); // Derived version unavailable**

**br.f(); // Base version available**

**br.f(s); // Base version available**

**} ///:~**

***Listing 15-11. Illustrating Variant Return Type***

**//: C15:VariantReturn.cpp**

**// Returning a pointer or reference to a derived**

**// type during overriding**

**#include <iostream>**

**#include <string>**

**using namespace std;**

**class PetFood {**

**public:**

**virtual string foodType() const = 0;**

**};**

**class Pet {**

**public:**

**virtual string type() const = 0;**

**virtual PetFood\* eats() = 0;**

**};**

**class Bird : public Pet {**

**public:**

**string type() const { return "Bird"; }**

**class BirdFood : public PetFood {**

**public:**

**string foodType() const {**

**return "Bird food";**

**}**

**};**

**// Upcast to base type:**

**PetFood\* eats() { return &bf; }**

**private:**

**BirdFood bf;**

**};**

**class Cat : public Pet {**

**public:**

**string type() const { return "Cat"; }**

**class CatFood : public PetFood {**

**public:**

**string foodType() const { return "Birds"; }**

**};**

**// Return exact type instead:**

**CatFood\* eats() { return &cf; }**

**private:**

**CatFood cf;**

**};**

**int main() {**

**Bird b;**

**Cat c;**

**Pet\* p[] = { &b, &c, };**

**for(int i = 0; i < sizeof p / sizeof \*p; i++)**

**cout << p[i]->type() << " eats "**

**<< p[i]->eats()->foodType() << endl;**

**// Can return the exact type:**

**Cat::CatFood\* cf = c.eats();**

**Bird::BirdFood\* bf;**

**// Cannot return the exact type:**

**//! bf = b.eats();**

**// Must downcast:**

**bf = dynamic\_cast<Bird::BirdFood\*>(b.eats());**

**} ///:~**

***Listing 15-12. Illustrating Behavior of Virtual vs. Non-Virtual Destructors***

**//: C15:VirtualDestructors.cpp**

**// Behavior of virtual vs. non-virtual destructor**

**#include <iostream>**

**using namespace std;**

**class Base1 {**

**public:**

**~Base1() { cout << "~Base1()\n"; }**

**};**

**class Derived1 : public Base1 {**

**public:**

**~Derived1() { cout << "~Derived1()\n"; }**

**};**

**class Base2 {**

**public:**

**virtual ~Base2() { cout << "~Base2()\n"; }**

**};**

**class Derived2 : public Base2 {**

**public:**

**~Derived2() { cout << "~Derived2()\n"; }**

**};**

**int main() {**

**Base1\* bp = new Derived1; // Upcast**

**delete bp;**

**Base2\* b2p = new Derived2; // Upcast**

**delete b2p;**

**} ///:~**

***Listing 15-13. Illustrating Pure Virtual Destructors***

**//: C15:UnAbstract.cpp**

**// Pure virtual destructors**

**// seem to behave strangely**

**class AbstractBase {**

**public:**

**virtual ~AbstractBase() = 0;**

**};**

**AbstractBase::~AbstractBase() {}**

**class Derived : public AbstractBase {};**

**// No overriding of destructor necessary?**

**int main() { Derived d; } ///:~*Listing 15-14. Illustrating that Pure Virtual Destructors Require a Function Body(also, shows that the Virtual Function Body is Called after the Derived Class Version)***

**//: C15:PureVirtualDestructors.cpp**

**// Pure virtual destructors**

**// require a function body**

**#include <iostream>**

**using namespace std;**

**class Pet {**

**public:**

**virtual ~Pet() = 0;**

**};**

**Pet::~Pet() {**

**cout << "~Pet()" << endl;**

**}**

**class Dog : public Pet {**

**public:**

**~Dog() {**

**cout << "~Dog()" << endl;**

**}**

**};**

**int main() {**

**Pet\* p = new Dog; // Upcast**

**delete p; // Virtual destructor call**

**} ///:~**

***Listing 15-15. Illustrating Virtual Calls Inside Destructors***

**//: C15:VirtualsInDestructors.cpp**

**// Virtual calls inside destructors**

**#include <iostream>**

**using namespace std;**

**class Base {**

**public:**

**virtual ~Base() {**

**cout << "Base1()\n";**

**f();**

**}**

**virtual void f() { cout << "Base::f()\n"; }**

**};**

**class Derived : public Base {**

**public:**

**~Derived() { cout << "~Derived()\n"; }**

**void f() { cout << "Derived::f()\n"; }**

**};**

**int main() {**

**Base\* bp = new Derived; // Upcast**

**delete bp;**

**} ///:~*****Listing 15-16. Illustrating a Singly-Rooted Hierarchy***

***(also, Known as an Object-based Hierarchy)***

**//: C15:OStack.h**

**// Using a singly-rooted hierarchy**

**#ifndef OSTACK\_H**

**#define OSTACK\_H**

**class Object {**

**public:**

**virtual ~Object() = 0;**

**};**

**// Required definition:**

**inline Object::~Object() {}**

**class Stack {**

**struct Link {**

**Object\* data;**

**Link\* next;**

**Link(Object\* dat, Link\* nxt) :**

**data(dat), next(nxt) {}**

**}\* head;**

**public:**

**Stack() : head(0) {}**

**~Stack(){**

**while(head)**

delete **pop();**

**}**

**void push(Object\* dat) {**

**head = new Link(dat, head);**

**}**

**Object\* peek() const {**

**return head ? head->data : 0;**

**}**

**Object\* pop() {**

**if(head == 0) return 0;**

**Object\* result = head->data;**

**Link\* oldHead = head;**

**head = head->next;**

**delete oldHead;**

**return result;**

**}**

**};**

**#endif // OSTACK\_H ///:~**

***Listing 15-17. Testing Out the OStack in Listing 15-16***

**//: C15:OStackTest.cpp**

**//{T} OStackTest.cpp**

**#include "OStack.h" // To be INCLUDED from above**

**#include "../require.h" // To be INCLUDED from *Chapter 9***

**#include <fstream>**

**#include <iostream>**

**#include <string>**

**using namespace std;**

**// Use multiple inheritance. We want**

**// both a string and an Object:**

**class MyString: public string, public Object {**

**public:**

**~MyString() {**

**cout << "deleting string: " << \*this << endl;**

**}**

**MyString(string s) : string(s) {}**

**};**

**int main(int argc, char\* argv[]) {**

**requireArgs(argc, 1); // File name is argument**

**ifstream in(argv[1]);**

**assure(in, argv[1]);**

**Stack textlines;**

**string line;**

**// Read file and store lines in the stack:**

**while(getline(in, line))**

**textlines.push(new MyString(line));**

**// Pop some lines from the stack:**

**MyString\* s;**

**for(int i = 0; i < 10; i++) {**

**if((s=(MyString\*)textlines.pop())==0) break;**

**cout << \*s << endl;**

**delete s;**

**}**

**cout << "Letting the destructor do the rest:” << endl;**

**} ///:~**

***Listing 15-18. Illustrating Polymorphism with Overloaded Operators***

**//: C15:OperatorPolymorphism.cpp**

**// Polymorphism with overloaded operators**

**#include <iostream>**

**using namespace std;**

**class Matrix;**

**class Scalar;**

**class Vector;**

**class Math {**

**public:**

**virtual Math& operator\*(Math& rv) = 0;**

**virtual Math& multiply(Matrix\*) = 0;**

**virtual Math& multiply(Scalar\*) = 0;**

**virtual Math& multiply(Vector\*) = 0;**

**virtual ~Math() {}**

**};**

**class Matrix : public Math {**

**public:**

**Math& operator\*(Math& rv) {**

**return rv.multiply(this); // 2nd dispatch**

**}**

**Math& multiply(Matrix\*) {**

**cout << "Matrix \* Matrix" << endl;**

**return \*this;**

**}**

**Math& multiply(Scalar\*) {**

**cout << "Scalar \* Matrix" << endl;**

**return \*this;**

**}**

**Math& multiply(Vector\*) {**

**cout << "Vector \* Matrix" << endl;**

**return \*this;**

**}**

**};**

**class Scalar : public Math {**

**public:**

**Math& operator\*(Math& rv) {**

**return rv.multiply(this); // 2nd dispatch**

**}**

**Math& multiply(Matrix\*) {**

**cout << "Matrix \* Scalar" << endl;**

**return \*this;**

**}**

**Math& multiply(Scalar\*) {**

**cout << "Scalar \* Scalar" << endl;**

**return \*this;**

**}**

**Math& multiply(Vector\*) {**

**cout << "Vector \* Scalar" << endl;**

**return \*this;**

**}**

**};**

**class Vector : public Math {**

**public:**

**Math& operator\*(Math& rv) {**

**return rv.multiply(this); // 2nd dispatch**

**}**

**Math& multiply(Matrix\*) {**

**cout << "Matrix \* Vector" << endl;**

**return \*this;**

**}**

**Math& multiply(Scalar\*) {**

**cout << "Scalar \* Vector" << endl;**

**return \*this;**

**}**

**Math& multiply(Vector\*) {**

**cout << "Vector \* Vector" << endl;**

**return \*this;**

**}**

**};**

**int main() {**

**Matrix m; Vector v; Scalar s;**

**Math\* math[] = { &m, &v, &s };**

**for(int i = 0; i < 3; i++)**

**for(int j = 0; j < 3; j++) {**

**Math& m1 = \*math[i];**

**Math& m2 = \*math[j];**

**m1 \* m2;**

**}**

**} ///:~**

***Listing 15-19. Illustrating a dynamic\_cast***

**//: C15:DynamicCast.cpp**

**#include <iostream>**

**using namespace std;**

**class Pet { public: virtual ~Pet(){}};**

**class Dog : public Pet {};**

**class Cat : public Pet {};**

**int main() {**

**Pet\* b = new Cat; // Upcast**

**// Try to cast it to Dog\*:**

**Dog\* d1 = dynamic\_cast<Dog\*>(b);**

**// Try to cast it to Cat\*:**

**Cat\* d2 = dynamic\_cast<Cat\*>(b);**

**cout << "d1 = " << (long)d1 << endl;**

**cout << "d2 = " << (long)d2 << endl;**

**} ///:~**

***Listing 15-20. Illustrating Navigation of Class Hierarchies with static\_cast***

**//: C15:StaticHierarchyNavigation.cpp**

**/// Navigating class hierarchies with static\_cast**

**#include <iostream>**

**#include <typeinfo>**

**using namespace std;**

**class Shape { public: virtual ~Shape() {}; };**

**class Circle : public Shape {};**

**class Square : public Shape {};**

**class Other {};**

**int main() {**

**Circle c;**

**Shape\* s = &c; // Upcast: normal and OK**

**// More explicit but unnecessary:**

**s = static\_cast<Shape\*>(&c);**

**// (Since upcasting is such a safe and common**

**// operation, the cast becomes cluttering)**

**Circle\* cp = 0;**

**Square\* sp = 0;**

**// Static Navigation of class hierarchies**

**// requires extra type information:**

**if(typeid(s) == typeid(cp)) // C++ RTTI**

**cp = static\_cast<Circle\*>(s);**

**if(typeid(s) == typeid(sp))**

**sp = static\_cast<Square\*>(s);**

**if(cp != 0)**

**cout << "It's a circle!" << endl;**

**if(sp != 0)**

**cout << "It's a square!" << endl;**

**// Static navigation is ONLY an efficiency hack;**

**// dynamic\_cast is always safer. However:**

**// Other\* op = static\_cast<Other\*>(s);**

**// Conveniently gives an error message, while**

**Other\* op2 = (Other\*)s;**

**// does not**

**} ///:~**