Chapter 14

INHERITANCE AND COMPOSITION

***Listing 14-1. A Valuable and Useful Reusable Class***

**//: C14:Useful.h**

**// A class to reuse**

**#ifndef USEFUL\_H**

**#define USEFUL\_H**

**class X {**

**int i;**

**public:**

**X() { i = 0; }**

**void set(int ii) { i = ii; }**

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

**int permute() { return i = i \* 47; }**

**};**

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

***Listing 14-2. Reusing Code with Composition***

**//: C14:Composition.cpp**

**// Reuse code with composition**

**#include "Useful.h" // To be INCLUDED from Header FILE above**

**class Y {**

**int i;**

**public:**

**X x; // Embedded object**

**Y() { i = 0; }**

**void f(int ii) { i = ii; }**

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

**};**

**int main() {**

**Y y;**

**y.f(47);**

**y.x.set(37); // Access the embedded object**

**} ///:~**

***Listing 14-3. A Composition with Private Embedded Objects***

**//: C14:Composition2.cpp**

**// Private embedded objects**

**#include "Useful.h"**

**class Y {**

**int i;**

**X x; // Embedded object**

**public:**

**Y() { i = 0; }**

**void f(int ii) { i = ii; x.set(ii); }**

**int g() const { return i \* x.read(); }**

**void permute() { x.permute(); }**

**};**

**int main() {**

**Y y;**

**y.f(47);**

**y.permute();**

**} ///:~**

***Listing 14-4. Illustrating Simple Inheritance***

**//: C14:Inheritance.cpp**

**// Simple inheritance**

**#include "Useful.h"**

**#include <iostream>**

**using namespace std;**

**class Y : public X {**

**inti; // Different from X's i**

**public:**

**Y() { i = 0; }**

**int change() {**

**i = permute(); // Different name call**

**return i;**

**}**

**void set(int ii) {**

**i = ii;**

**X::set(ii); // Same-name function call**

**}**

**};**

**int main() {**

**cout << "sizeof(X) = " << sizeof(X) << endl;**

**cout << "sizeof(Y) = "**

**<< sizeof(Y) << endl;**

**Y D;**

**D.change();**

**// X function interface comes through:**

**D.read();**

**D.permute();**

**// Redefined functions hide base versions:**

**D.set(12);**

**} ///:~**

***Listing 14-5. Demonstrating Pseudo-constructor***

**//: C14:PseudoConstructor.cpp**

**class X {**

**int i;**

**float f;**

**char c;**

**char \*s;**

**public:**

**X() : i(7), f(1.4), c('x'), s("howdy") {}**

**};**

**int main() {**

**X x;**

**int i(100); // Applied to ordinary definition**

**int \*ip = new int(47);**

**} ///:~*****Listing 14-6. Illustrating Combined Composition and Inheritance***

**//: C14:Combined.cpp**

**// Inheritance & composition**

**class A {**

**int i;**

**public:**

**A(int ii) : i(ii) {}**

**~A() {}**

**void f() const {}**

**};**

**class B {**

**int i;**

**public:**

**B(int ii) : i(ii) {}**

**~B() {}**

**void f() const {}**

**};**

**class C : public B {**

**A a;**

**public:**

**C(int ii) : B(ii), a(ii) {}**

**~C() {} // Calls ~A() and ~B()**

**void f() const { // Redefinition**

**a.f();**

**B::f();**

**}**

**};**

**int main() {**

**C c(47);**

**} ///:~**

***Listing 14-7. Demonstrating Order of Constructor/Destructor Calls***

**//: C14:Order.cpp**

**// Constructor/destructor order**

**#include <fstream>**

**using namespace std;**

**ofstream out("order.out");**

**#define CLASS(ID) class ID { \**

**public: \**

**ID(int) { out << #ID " constructor\n"; } \**

**~ID() { out << #ID " destructor\n"; } \**

**};**

**CLASS(Base1);**

**CLASS(Member1);**

**CLASS(Member2);**

**CLASS(Member3);**

**CLASS(Member4);**

**class Derived1 : public Base1 {**

**Member1 m1;**

**Member2 m2;**

**public:**

**Derived1(int) : m2(1), m1(2), Base1(3) {**

**out << "Derived1 constructor\n";**

**}**

**~Derived1() {**

**out << "Derived1 destructor\n";**

**}**

**};**

**class Derived2 : public Derived1 {**

**Member3 m3;**

**Member4 m4;**

**public:**

**Derived2() : m3(1), Derived1(2), m4(3) {**

**out << "Derived2 constructor\n";**

**}**

**~Derived2() {**

**out << "Derived2 destructor\n";**

**}**

**};**

**int main() {**

**Derived2 d2;**

**} ///:~**

***Listing 14-8. Illustrating Hiding of Overloaded Names (during Inheritance)***

**//: C14:NameHiding.cpp**

**// Hiding overloaded names during inheritance**

**#include <iostream>**

**#include <string>**

**using namespace std;**

**class Base {**

**public:**

**int f() const {**

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

**return 1;**

**}**

**int f(string) const { return 1; }**

**void g() {}**

**};**

**class Derived1 : public Base {**

**public:**

**void g() const {}**

**};**

**class Derived2 : public Base {**

**public:**

**// Redefinition:**

**int f() const {**

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

**return 2;**

**}**

**};**

**class Derived3 : public Base {**

**public:**

**// 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");**

**14: Inheritance & Composition 627**

**Derived1 d1;**

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

**d1.f(s);**

**Derived2 d2;**

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

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

**Derived3 d3;**

**//! x = d3.f(); // return int version hidden**

**Derived4 d4;**

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

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

**} ///:~*Listing 14-9. Specializing the General Stack class Using Inheritance***

**//: C14:InheritStack.cpp**

**// Specializing the Stack class**

**#include "../C09/Stack4.h" // Refer Chapter 9**

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

**#include <iostream>**

**#include <fstream>**

**#include <string>**

**using namespace std;**

**class StringStack : public Stack {**

**public:**

**void push(string\* str) {**

**Stack::push(str);**

**}**

**string\* peek() const {**

**return (string\*)Stack::peek();**

**}**

**string\* pop() {**

**return (string\*)Stack::pop();**

**}**

**~StringStack() {**

**string\* top = pop();**

**while(top) {**

**delete top;**

**top = pop();**

**}**

**}**

**};**

**int main() {**

**ifstream in("InheritStack.cpp");**

**assure(in, "InheritStack.cpp");**

**string line;**

**StringStack textlines;**

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

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

**string\* s;**

**while((s = textlines.pop()) != 0) { // No cast!**

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

**delete s;**

**}**

**} ///:~**

***Listing 14-10. Illustrating Synthesized Functions***

**//: C14:SynthesizedFunctions.cpp**

**// Functions that are synthesized by the compiler**

**#include <iostream>**

**using namespace std;**

**classGameBoard {**

**public:**

**GameBoard() { cout << "GameBoard()\n"; }**

**GameBoard(const GameBoard&) {**

**cout << "GameBoard(const GameBoard&)\n";**

**}**

**GameBoard& operator=(const GameBoard&) {**

**cout << "GameBoard::operator=()\n";**

**return \*this;**

**}**

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

**};**

**class Game {**

**GameBoard gb; // Composition**

**public:**

**// Default GameBoard constructor called:**

**Game() { cout << "Game()\n"; }**

**// You must explicitly call the GameBoard**

**// copy-constructor or the default constructor**

**// is automatically called instead:**

**Game(const Game& g) : gb(g.gb) {**

**cout << "Game(const Game&)\n";**

**}**

**Game(int) { cout << "Game(int)\n"; }**

**Game& operator=(const Game& g) {**

**// You must explicitly call the GameBoard**

**// assignment operator or no assignment at**

**// all happens for gb!**

**gb = g.gb;**

**cout << "Game::operator=()\n";**

**return \*this;**

**}**

**class Other {}; // Nested class**

**// Automatic type conversion:**

**operator Other() const {**

**cout << "Game::operator Other()\n";**

**return Other();**

**}**

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

**};**

**class Chess : public Game {};**

**void f(Game::Other) {}**

**class Checkers : public Game {**

**public:**

**// Default base-class constructor called:**

**Checkers() { cout << "Checkers()\n"; }**

**// You must explicitly call the base-class**

**// copy constructor or the default constructor**

**// will be automatically called instead:**

**Checkers(const Checkers& c) : Game(c) {**

**cout << "Checkers(const Checkers& c)\n";**

**}**

**Checkers& operator=(const Checkers& c) {**

**// You must explicitly call the base-class**

**// version of operator=() or no base-class**

**// assignment will happen:**

**Game::operator=(c);**

**cout << "Checkers::operator=()\n";**

**return \*this;**

**}**

**};**

**int main() {**

**Chess d1; // Default constructor**

**Chess d2(d1); // Copy-constructor**

**//! Chess d3(1); // Error: no int constructor**

**d1 = d2; // Operator= synthesized**

**f(d1); // Type-conversion IS inherited**

**Game::Other go; /\* This declaration is only for the purpose of**

**demonstrating to you the next line of code which has been commented out for obvious reasons!(otherwise, the program will not compile!!)\*/**

**//! d1 = go; // Operator= not synthesized**

**// for differing types**

**Checkers c1, c2(c1);**

**c1 = c2;**

**} ///:~*****Listing 14-11. Illustrating A Public Composition***

**//: C14:Car.cpp**

**// Public composition**

**class Engine {**

**public:**

**void start() const {}**

**void rev() const {}**

**void stop() const {}**

**};**

**class Wheel {**

**public:**

**void inflate(int psi) const {}**

**};**

**class Window {**

**public:**

**void rollup() const {}**

**void rolldown() const {}**

**};**

**class Door {**

**public:**

**Window window;**

**void open() const {}**

**void close() const {}**

**};**

**class Car {**

**public:**

**Engine engine;**

**Wheel wheel[4];**

**Door left, right; // 2-door**

**};**

**int main() {**

**Car car;**

**car.left.window.rollup();**

**car.wheel[0].inflate(72);**

**} ///:~**

***Listing 14-12. Embedding both an ifstream and a string(a File Name) using Composition***

**//: C14:FName1.cpp**

**// An ifstream with a file name**

**#include "../require.h"**

**#include <iostream>**

**#include <fstream>**

**#include <string>**

**using namespace std;**

**class FName1 {**

**ifstream file;**

**string fileName;**

**bool named;**

**public:**

**FName1() : named(false) {}**

**FName1(const string &fname)**

**: fileName(fname), file(fname.c\_str()) {**

**assure(file, fileName);**

**named = true;**

**}**

**string name() const { return fileName; }**

**void name(const string &newName) {**

**if(named) return; // Don't overwrite**

**fileName = newName;**

**named = true;**

**}**

**operator ifstream&() { return file; }**

**};**

**int main() {**

**FName1 file("FName1.cpp");**

**cout << file.name() << endl;**

**// Error: close() not a member:**

**//! file.close();**

**} ///:~**

***Listing 14-13. Illustrating that Subtyping Solves the Problem in Listing 14-12***

**//: C14:FName2.cpp**

**// Subtyping solves the problem**

**#include "../require.h"**

**#include <iostream>**

**#include <fstream>**

**#include <string>**

**using namespace std;**

**class FName2 : public ifstream {**

**string fileName;**

**bool named;**

**public:**

**FName2() : named(false) {}**

**FName2(const string &fname)**

**: ifstream(fname.c\_str()), fileName(fname) {**

**assure(\*this, fileName);**

**named = true;**

**}**

**string name() const { return fileName; }**

**void name(const string &newName) {**

**if(named) return; // Don't overwrite**

**fileName = newName;**

**named = true;**

**}**

**};**

**int main() {**

**FName2 file("FName2.cpp");**

**assure(file, "FName2.cpp");**

**cout << "name: " << file.name() << endl;**

**string s;**

**getline(file, s); // These work too!**

**file.seekg(-200, ios::end);**

**file.close();**

**} ///:~**

***Listing 14-14. Demonstrating Private Inheritance***

**//: C14:PrivateInheritance.cpp**

**class Pet {**

**public:**

**char eat() const { return 'a'; }**

**int speak() const { return 2; }**

**float sleep() const { return 3.0; }**

**float sleep(int) const { return 4.0; }**

**};**

**class Goldfish : Pet { // Private inheritance**

**public:**

**using Pet::eat; // Name publicizes member**

**using Pet::sleep; // Both overloaded members exposed**

**};**

**int main() {**

**Goldfish bob;**

**bob.eat();**

**bob.sleep();**

**bob.sleep(1);**

**//! bob.speak();// Error: private member function**

**} ///:~**

***Listing 14.15. Illustrating Use of The protected Keyword***

**//: C14:Protected.cpp**

**// The protected keyword**

**#include <fstream>**

**using namespace std;**

**class Base {**

**int i;**

**protected:**

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

**void set(int ii) { i = ii; }**

**public:**

**Base(int ii = 0) : i(ii) {}**

**int value(int m) const { return m\*i; }**

**};**

**class Derived : public Base {**

**int j;**

**public:**

**Derived(int jj = 0) : j(jj) {}**

**void change(int x) { set(x); }**

**};**

**int main() {**

**Derived d;**

**d.change(10);**

**} ///:~**

***Listing 14-16. Illustrating Inheritance of Overloaded Operators***

**//: C14:OperatorInheritance.cpp**

**// Inheriting overloaded operators**

**#include "../C12/Byte.h" // Refer Chapter 12**

**#include <fstream>**

**using namespace std;**

**ofstream out("ByteTest.out");**

**class Byte2 : public Byte {**

**public:**

**// Constructors don't inherit:**

**Byte2(unsigned char bb = 0) : Byte(bb) {}**

**// operator= does not inherit, but**

**// is synthesized for memberwise assignment.**

**// However, only the SameType = SameType**

**// operator= is synthesized, so you have to**

**// make the others explicitly:**

**Byte2& operator=(const Byte& right) {**

**Byte::operator=(right);**

**return \*this;**

**}**

**Byte2& operator=(inti) {**

**Byte::operator=(i);**

**return \*this;**

**}**

**};**

**// Similar test function as in C12:ByteTest.cpp:**

**void k(Byte2& b1, Byte2& b2) {**

**b1 = b1 \* b2 + b2 % b1;**

**#define TRY2(OP) \**

**out << "b1 = "; b1.print(out); \**

**out << ", b2 = "; b2.print(out); \**

**out << "; b1 " #OP " b2 produces "; \**

**(b1 OP b2).print(out); \**

**out << endl;**

**b1 = 9; b2 = 47;**

**TRY2(+) TRY2(-) TRY2(\*) TRY2(/)**

**TRY2(%) TRY2(^) TRY2(&) TRY2(|)**

**TRY2(<<) TRY2(>>) TRY2(+=) TRY2(-=)**

**TRY2(\*=) TRY2(/=) TRY2(%=) TRY2(^=)**

**TRY2(&=) TRY2(|=) TRY2(>>=) TRY2(<<=)**

**TRY2(=) // Assignment operator**

**// Conditionals:**

**#define TRYC2(OP) \**

**out << "b1 = "; b1.print(out); \**

**out << ", b2 = "; b2.print(out); \**

**out << "; b1 " #OP " b2 produces "; \**

**out << (b1 OP b2); \**

**out << endl;**

**b1 = 9; b2 = 47;**

**TRYC2(<) TRYC2(>) TRYC2(==) TRYC2(!=) TRYC2(<=)**

**TRYC2(>=) TRYC2(&&) TRYC2(||)**

**// Chained assignment:**

**Byte2 b3 = 92;**

**b1 = b2 = b3;**

**}**

**int main() {**

**out << "member functions:" << endl;**

**Byte2 b1(47), b2(9);**

**k(b1, b2);**

**} ///:~**

***Listing 14-17. Illustrating Inheritance and Upcasting***

**//: C14:Instrument.cpp**

**// Inheritance & upcasting**

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

**class Instrument {**

**public:**

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

**};**

**// Wind objects are Instruments**

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

**class Wind : public Instrument {};**

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

**// ...**

**i.play(middleC);**

**}**

**int main() {**

**Wind flute;**

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

**} ///:~**

***Listing 14-18. Demonstrating Correct Creation of the copy-constructor***

**//: C14:CopyConstructor.cpp**

**// Correctly creating the copy-constructor**

**#include <iostream>**

**using namespace std;**

**class Parent {**

**int i;**

**public:**

**Parent(int ii) : i(ii) {**

**cout << "Parent(int ii)\n";**

**}**

**Parent(const Parent& b) : i(b.i) {**

**cout<< "Parent(const Parent&)\n";**

**}**

**Parent() : i(0) { cout << "Parent()\n"; }**

**friend ostream&**

**operator <<(ostream& os, const Parent& b) {**

**return os << "Parent: " << b.i << endl;**

**}**

**};**

**class Member {**

**int i;**

**public:**

**Member(int ii) : i(ii) {**

**cout << "Member(int ii)\n";**

**}**

**Member(const Member& m) : i(m.i) {**

**cout << "Member(const Member&)\n";**

**}**

**friend ostream&**

**operator <<(ostream& os, const Member& m) {**

**return os<< "Member: " << m.i<< endl;**

**}**

**};**

**class Child : public Parent {**

**int i;**

**Member m;**

**public:**

**Child(int ii) : Parent(ii), i(ii), m(ii) {**

**cout << "Child(int ii)\n";**

**}**

**friend ostream&**

**operator <<(ostream& os, const Child& c){**

**return os << (Parent&)c << c.m**

**<< "Child: " << c.i << endl;**

**}**

**};**

**int main() {**

**Child c(2);**

**cout << "calling copy-constructor: " << endl;**

**Child c2 = c; // Calls copy-constructor**

**cout << "values in c2:\n" << c2;**

**} ///:~**

***Listing 14-19. Comparing Compositing with Inheritance***

**//: C14:InheritStack2.cpp**

**// Composition vs. inheritance**

**#include "../C09/Stack4.h"**

**#include "../require.h"**

**#include <iostream>**

**#include <fstream>**

**#include <string>**

**using namespace std;**

**class StringStack {**

**Stack stack; // Embed instead of inherit**

**public:**

**void push(string\* str) {**

**stack.push(str);**

**}**

**string\* peek() const {**

**return (string\*)stack.peek();**

**}**

**string\* pop() {**

**return (string\*)stack.pop();**

**}**

**};**

**int main() {**

**ifstream in("InheritStack2.cpp");**

**assure(in, "InheritStack2.cpp");**

**string line;**

**StringStack textlines;**

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

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

**string\* s;**

**while((s = textlines.pop()) != 0) // No cast!**

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

**} ///:~**