Chapter 21

MULTIPLE INHERITANCE (MI)

***Listing 21-1. Illustrating Multiple Interface Inheritance***

**//: C21:Interfaces.cpp**

**// Multiple interface inheritance.**

**#include <iostream>**

**#include <sstream>**

**#include <string>**

**using namespace std;**

**class Printable {**

**public:**

**virtual ~Printable() {}**

**virtual void print(ostream&) const = 0;**

**};**

**class Intable {**

**public:**

**virtual ~Intable() {}**

**virtual int toInt() const = 0;**

**};**

**class Stringable {**

**public:**

**virtual ~Stringable() {}**

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

**};**

**class Able : public Printable, public Intable,**

**public Stringable {**

**int myData;**

**public:**

**Able(int x) { myData = x; }**

**void print(ostream& os) const { os << myData; }**

**int toInt() const { return myData; }**

**string toString() const {**

**ostringstream os;**

**os << myData;**

**return os.str();**

**}**

**};**

**void testPrintable(const Printable& p) {**

**p.print(cout);**

**cout << endl;**

**}**

**void testIntable(const Intable& n) {**

**cout << n.toInt() + 1 << endl;**

**}**

**void testStringable(const Stringable& s) {**

**cout << s.toString() + "th" << endl;**

**}**

**int main() {**

**Able a(7);**

**testPrintable(a);**

**testIntable(a);**

**testStringable(a);**

**} ///:~*****Listing 21-2. Illustrating Implicit Interface Inheritance***

***(using templates)***

**//: C21:Interfaces2.cpp**

**// Implicit interface inheritance via templates.**

**#include <iostream>**

**#include <sstream>**

**#include <string>**

**using namespace std;**

**class Able {**

**int myData;**

**public:**

**Able(int x) { myData = x; }**

**void print(ostream& os) const { os << myData; }**

**int toInt() const { return myData; }**

**string toString() const {**

**ostringstream os;**

**os << myData;**

**return os.str();**

**}**

**};**

**template<class Printable>**

**void testPrintable(const Printable& p) {**

**p.print(cout);**

**cout << endl;**

**}**

**template<class Intable>**

**void testIntable(const Intable& n) {**

**cout << n.toInt() + 1 << endl;**

**}**

**template<class Stringable>**

**void testStringable(const Stringable& s) {**

**cout << s.toString() + "th" << endl;**

**}**

**int main() {**

**Able a(7);**

**testPrintable(a);**

**testIntable(a);**

**testStringable(a);**

**} ///:~**

***Listing 21-3. Implementing A Database Class***

**//: C21:Database.h**

**// A prototypical resource class.**

**// A prototypical resource class.**

**#ifndef DATABASE\_H**

**#define DATABASE\_H**

**#include <iostream>**

**#include <stdexcept>**

**#include <string>**

**struct DatabaseError : std::runtime\_error {**

**DatabaseError(const std::string& msg)**

**: std::runtime\_error(msg) {}**

**};**

**class Database {**

**std::string dbid;**

**public:**

**Database(const std::string& dbStr) : dbid(dbStr) {}**

**virtual ~Database() {}**

**void open() throw(DatabaseError) {**

**std::cout << "Connected to " << dbid << std::endl;**

**}**

**void close() {**

**std::cout << dbid << " closed" << std::endl;**

**}**

**// Other database functions...**

**};**

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

**/\* We’re leaving out actual database functionality (storing, retrieving, and so on), but that’s not important here. Using this class requires a database connection string and that you call Database::open( ) to connect and Database::close( ) to disconnect: \*/**/\* We’re leaving out actual database functionality (storing, retrieving, and so on), but that’s not important here. Using this class requires a database connection string and that you call **Database::open( )** to connect and **Database::close( )** to disconnect: \*/

**//: C21:UseDatabase.cpp**

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

**// above**

**int main() {**

**Database db("MyDatabase");**

**db.open();**

**// Use other db functions...**

**db.close();**

**}**

**/\* Output:**

**connected to MyDatabase**

**MyDatabase closed**

**\*/ ///:~**

***Listing 21-4. Illustrating the Countable “mixin” Class***

**//: C21:Countable.h**

**// A "mixin" class.**

**#ifndef COUNTABLE\_H**

**#define COUNTABLE\_H**

**#include <cassert>**

**class Countable {**

**long count;**

**protected:**

**Countable() { count = 0; }**

**virtual ~Countable() { assert(count == 0); }**

**public:**

**long attach() { return ++count; }**

**long detach() {**

**return (--count > 0) ? count : (delete this, 0);**

**}**

**long refCount() const { return count; }**

**};**

**#endif // COUNTABLE\_H ///:~*****Listing 21-5. Using the Countable “mixin” Class***

**//: C21:DBConnection.h**

**// Uses a "mixin" class.**

**#ifndef DBCONNECTION\_H**

**#define DBCONNECTION\_H**

**#include <cassert>**

**#include <string>**

**#include "Countable.h" // To be INCLUUDED from Header FILE**

**// above**

**#include "Database.h"**

**using std::string;**

**class DBConnection : public Database, public Countable {**

**DBConnection(const DBConnection&); // Disallow copy**

**DBConnection& operator=(const DBConnection&);**

**protected:**

**DBConnection(const string& dbStr) throw(DatabaseError)**

**: Database(dbStr) { open(); }**

**~DBConnection() { close(); }**

**public:**

**static DBConnection\***

**create(const string& dbStr) throw(DatabaseError) {**

**DBConnection\* con = new DBConnection(dbStr);**

**con->attach();**

**assert(con->refCount() == 1);**

**return con;**

**}**

**// Other added functionality as desired...**

**};**

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

***Listing 21-6. Testing Out the Countable “mixin” Class***

**//: C21:UseDatabase2.cpp**

**// Tests the Countable "mixin" class.**

**#include <cassert>**

**#include "DBConnection.h" // To be INCLUUDED from Header FILE**

**// above**

**class DBClient {**

**DBConnection\* db;**

**public:**

**DBClient(DBConnection\* dbCon) {**

**db = dbCon;**

**db->attach();**

**}**

**~DBClient() { db->detach(); }**

**// Other database requests using db…**

**};**

**int main() {**

**DBConnection\* db = DBConnection::create("MyDatabase");**

**assert(db->refCount() == 1);**

**DBClient c1(db);**

**assert(db->refCount() == 2);**

**DBClient c2(db);**

**assert(db->refCount() == 3);**

**// Use database, then release attach from original create**

**db->detach();**

**assert(db->refCount() == 2);**

**} ///:~**

***Listing 21-7. Illustrating a Parameterized ”mixin” Class***

***(using Templates)***

**//: C21:DBConnection2.h**

**// A parameterized mixin.**

**#ifndef DBCONNECTION2\_H**

**#define DBCONNECTION2\_H**

**#include <cassert>**

**#include <string>**

**#include "Database.h"**

**using std::string;**

**template<class Counter>**

**class DBConnection : public Database, public Counter {**

**DBConnection(const DBConnection&); // Disallow copy**

**DBConnection& operator=(const DBConnection&);**

**protected:**

**DBConnection(const string& dbStr) throw(DatabaseError)**

**: Database(dbStr) { open(); }**

**~DBConnection() { close(); }**

**public:**

**static DBConnection\* create(const string& dbStr)**

**throw(DatabaseError) {**

**DBConnection\* con = new DBConnection(dbStr);**

**con->attach();**

**assert(con->refCount() == 1);**

**return con;**

**}**

**// Other added functionality as desired...**

**};**

**#endif // DBCONNECTION2\_H ///:~**

***Listing 21-8. Testing Out the Parametrized “mixin” Class***

**//: C21:UseDatabase3.cpp**

**// Tests a parameterized "mixin" class.**

**#include <cassert>**

**#include "Countable.h"**

**#include "DBConnection2.h" // To be INCLUDED from Header FILE**

**// above**

**class DBClient {**

**DBConnection<Countable>\* db;**

**public:**

**DBClient(DBConnection<Countable>\* dbCon) {**

**db = dbCon;**

**db->attach();**

**}**

**~DBClient() { db->detach(); }**

**};**

**int main() {**

**DBConnection<Countable>\* db =**

**DBConnection<Countable>::create("MyDatabase");**

**assert(db->refCount() == 1);**

**DBClient c1(db);**

**assert(db->refCount() == 2);**

**DBClient c2(db);**

**assert(db->refCount() == 3);**

**db->detach();**

**assert(db->refCount() == 2);**

**} ///:~**

***Listing 21-9. Demonstrating Layout of Subobjects with MI***

**//: C21:Offset.cpp**

**// Illustrates layout of subobjects with MI.**

**#include <iostream>**

**using namespace std;**

**class A { int x; };**

**class B { int y; };**

**class C : public A, public B { int z; };**

**int main() {**

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

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

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

**C c;**

**cout << "&c == " << &c << endl;**

**A\* ap = &c;**

**B\* bp = &c;**

**cout << "ap == " << static\_cast<void\*>(ap) << endl;**

**cout << "bp == " << static\_cast<void\*>(bp) << endl;**

**C\* cp = static\_cast<C\*>(bp);**

**cout << "cp == " << static\_cast<void\*>(cp) << endl;**

**cout << "bp == cp? " << boolalpha << (bp == cp) << endl;**

**cp = 0;**

**bp = cp;**

**cout << bp << endl;**

**} ///:~**

***Listing 21-10. Demonstrating Duplicate Subobjects***

**//: C21:Duplicate.cpp**

**// Shows duplicate subobjects.**

**#include <iostream>**

**using namespace std;**

**class Top {**

**int x;**

**public:**

**Top(int n) { x = n; }**

**};**

**class Left : public Top {**

**int y;**

**public:**

**Left(int m, int n) : Top(m) { y = n; }**

**};**

**class Right : public Top {**

**int z;**

**public:**

**Right(int m, int n) : Top(m) { z = n; }**

**};**

**class Bottom : public Left, public Right {**

**int w;**

**public:**

**Bottom(int i, int j, int k, int m)**

**: Left(i, k), Right(j, k) { w = m; }**

**};**

**int main() {**

**Bottom b(1, 2, 3, 4);**

**cout << sizeof b << endl; // 20**

**} ///:~**

***Listing 21-11. Demonstrating True Diamond Inheritance***

**//: C21:VirtualBase.cpp**

**// Shows a shared subobject via a virtual base.**

**#include <iostream>**

**using namespace std;**

**class Top {**

**protected:**

**int x;**

**public:**

**Top(int n) { x = n; }**

**virtual ~Top() {}**

**friend ostream&**

**operator<<(ostream& os, const Top& t) {**

**return os << t.x;**

**}**

**};**

**class Left : virtual public Top {**

**protected:**

**int y;**

**public:**

**Left(int m, int n) : Top(m) { y = n; }**

**};**

**class Right : virtual public Top {**

**protected:**

**int z;**

**public:**

**Right(int m, int n) : Top(m) { z = n; }**

**};**

**class Bottom : public Left, public Right {**

**int w;**

**public:**

**Bottom(int i, int j, int k, int m)**

**: Top(i), Left(0, j), Right(0, k) { w = m; }**

**friend ostream&**

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

**return os << b.x << ',' << b.y << ',' << b.z**

**<< ',' << b.w;**

**}**

**};**

**int main() {**

**Bottom b(1, 2, 3, 4);**

**cout << sizeof b << endl;**

**cout << b << endl;**

**cout << static\_cast<void\*>(&b) << endl;**

**Top\* p = static\_cast<Top\*>(&b);**

**cout << \*p << endl;**

**cout << static\_cast<void\*>(p) << endl;**

**cout << dynamic\_cast<void\*>(p) << endl;**

**} ///:~**

***Listing 21-12. Demonstrating a Wrong Way to Implement operator<<( )***

**//: C21:VirtualBase2.cpp**

**// How NOT to implement operator<<.**

**#include <iostream>**

**using namespace std;**

**class Top {**

**int x;**

**public:**

**Top(int n) { x = n; }**

**virtual ~Top() {}**

**friend ostream& operator<<(ostream& os, const Top& t) {**

**return os << t.x;**

**}**

**};**

**class Left : virtual public Top {**

**int y;**

**public:**

**Left(int m, int n) : Top(m) { y = n; }**

**friend ostream& operator<<(ostream& os, const Left& l) {**

**return os << static\_cast<const Top&>(l) << ',' << l.y;**

**}**

**};**

**class Right : virtual public Top {**

**int z;**

**public:**

**Right(int m, int n) : Top(m) { z = n; }**

**friend ostream& operator<<(ostream& os, const Right& r) {**

**return os << static\_cast<const Top&>(r) << ',' << r.z;**

**}**

**};**

**class Bottom : public Left, public Right {**

**int w;**

**public:**

**Bottom(int i, int j, int k, int m)**

**: Top(i), Left(0, j), Right(0, k) { w = m; }**

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

**return os << static\_cast<const Left&>(b)**

**<< ',' << static\_cast<const Right&>(b)**

**<< ',' << b.w;**

**}**

**};**

**int main() {**

**Bottom b(1, 2, 3, 4);**

**cout << b << endl;  // 1,2,1,3,4**

**} ///:~**

***Listing 21-13. Demonstrating a Correct Stream Inserter***

**//: C21:VirtualBase3.cpp**

**// A correct stream inserter.**

**#include <iostream>**

**using namespace std;**

**class Top {**

**int x;**

**public:**

**Top(int n) { x = n; }**

**virtual ~Top() {}**

**friend ostream& operator<<(ostream& os, const Top& t) {**

**return os << t.x;**

**}**

**};**

**class Left : virtual public Top {**

**int y;**

**protected:**

**void specialPrint(ostream& os) const {**

**// Only print Left's part**

**os << ',' << y;**

**}**

**public:**

**Left(int m, int n) : Top(m) { y = n; }**

**friend ostream& operator<<(ostream& os, const Left& l) {**

**return os << static\_cast<const Top&>(l) << ',' << l.y;**

**}**

**};**

**class Right : virtual public Top {**

**int z;**

**protected:**

**void specialPrint(ostream& os) const {**

**// Only print Right's part**

**os << ',' << z;**

**}**

**public:**

**Right(int m, int n) : Top(m) { z = n; }**

**friend ostream& operator<<(ostream& os, const Right& r) {**

**return os << static\_cast<const Top&>(r) << ',' << r.z;**

**}**

**};**

**class Bottom : public Left, public Right {**

**int w;**

**public:**

**Bottom(int i, int j, int k, int m)**

**: Top(i), Left(0, j), Right(0, k) { w = m; }**

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

**os << static\_cast<const Top&>(b);**

**b.Left::specialPrint(os);**

**b.Right::specialPrint(os);**

**return os << ',' << b.w;**

**}**

**};**

**int main() {**

**Bottom b(1, 2, 3, 4);**

**cout << b << endl;  // 1,2,3,4**

**} ///:~**

***Listing 21-14. Illustrating Initialization Order with Virtual Base Classes***

**//: C21:VirtInit.cpp**

**// Illustrates initialization order with virtual bases.**

**#include <iostream>**

**#include <string>**

**using namespace std;**

**class M {**

**public:**

**M(const string& s) { cout << "M " << s << endl; }**

**};**

**class A {**

**M m;**

**public:**

**A(const string& s) : m("in A") {**

**cout << "A " << s << endl;**

**}**

**virtual ~A() {}**

**};**

**class B {**

**M m;**

**public:**

**B(const string& s) : m("in B")  {**

**cout << "B " << s << endl;**

**}**

**virtual ~B() {}**

**};**

**class C {**

**M m;**

**public:**

**C(const string& s) : m("in C")  {**

**cout << "C " << s << endl;**

**}**

**virtual ~C() {}**

**};**

**class D {**

**M m;**

**public:**

**D(const string& s) : m("in D") {**

**cout << "D " << s << endl;**

**}**

**virtual ~D() {}**

**};**

**class E : public A, virtual public B, virtual public C {**

**M m;**

**public:**

**E(const string& s) : A("from E"), B("from E"),**

**C("from E"), m("in E") {**

**cout << "E " << s << endl;**

**}**

**};**

**class F : virtual public B, virtual public C, public D {**

**M m;**

**public:**

**F(const string& s) : B("from F"), C("from F"),**

**D("from F"), m("in F") {**

**cout << "F " << s << endl;**

**}**

**};**

**class G : public E, public F {**

**M m;**

**public:**

**G(const string& s) : B("from G"), C("from G"),**

**E("from G"),  F("from G"), m("in G") {**

**cout << "G " << s << endl;**

**}**

**};**

**int main() {**

**G g("from main");**

**} ///:~**

***Listing 21-15. Illustrating Ambiguous Function Names***

**//: C21:AmbiguousName.cpp {-xo}**

**class Top {**

**public:**

**virtual ~Top() {}**

**};**

**class Left : virtual public Top {**

**public:**

**void f() {}**

**};**

**class Right : virtual public Top {**

**public:**

**void f() {}**

**};**

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

**int main() {**

**Bottom b;**

**b.f(); // Error here**

**} ///:~**

***Listing 21-16. Resolving the Ambiguity in Listing 21-15***

**//: C21:BreakTie.cpp**

**class Top {**

**public:**

**virtual ~Top() {}**

**};**

**class Left : virtual public Top {**

**public:**

**void f() {}**

**};**

**class Right : virtual public Top {**

**public:**

**void f() {}**

**};**

**class Bottom : public Left, public Right {**

**public:**

**using Left::f;**

**};**

**int main() {**

**Bottom b;**

**b.f(); // Calls Left::f()**

**} ///:~**

***Listing 21-17. Illustrating the Dominance Principle to Resolve Function Name Ambiguities in a Class Hierarchy***

**//: C21:Dominance.cpp**

**class Top {**

**public:**

**virtual ~Top() {}**

**virtual void f() {}**

**};**

**class Left : virtual public Top {**

**public:**

**void f() {}**

**};**

**class Right : virtual public Top {};**

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

**int main() {**

**Bottom b;**

**b.f(); // Calls Left::f()**

**} ///:~**

***Listing 21-18. Illustrating the Dominance Principle (again) to Resolve More Ambiguities***

**//: C21:Dominance2.cpp**

**#include <iostream>**

**using namespace std;**

**class A {**

**public:**

**virtual ~A() {}**

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

**};**

**class B : virtual public A {**

**public:**

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

**};**

**class C : public B {};**

**class D : public C, virtual public A {};**

**int main() {**

**B\* p = new D;**

**p->f(); // Calls B::f()**

**delete p;**

**} ///:~**

***Listing 21-19. Illustrating a Vendor-Supplied Class Header***

**//: C21:Vendor.h**

**// Vendor-supplied class header**

**// You only get this & the compiled Vendor.obj.**

**#ifndef VENDOR\_H**

**#define VENDOR\_H**

**class Vendor {**

**public:**

**virtual void v() const;**

**void f() const; // Might want this to be virtual...**

**~Vendor(); // Oops! Not virtual!**

**};**

**class Vendor1 : public Vendor {**

**public:**

**void v() const;**

**void f() const;**

**~Vendor1();**

**};**

**void A(const Vendor&);**

**void B(const Vendor&);**

**// Etc.**

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

***Listing 21-20. Implementing the header file in Listing 21-19 (Vendor.h)***

**//: C21:Vendor.cpp {O}**

**// Assume this is compiled and unavailable to you.**

**#include "Vendor.h" // To be INCLUUDED from Header FILE**

**// above**

**#include <iostream>**

**using namespace std;**

**void Vendor::v() const { cout << "Vendor::v()" << endl; }**

**void Vendor::f() const { cout << "Vendor::f()" << endl; }**

**Vendor::~Vendor() { cout << "~Vendor()" << endl; }**

**void Vendor1::v() const { cout << "Vendor1::v()" << endl; }**

**void Vendor1::f() const { cout << "Vendor1::f()" << endl; }**

**Vendor1::~Vendor1() { cout << "~Vendor1()" << endl; }**

**void A(const Vendor& v) {**

**// ...**

**v.v();**

**v.f();**

**// ...**

**}**

**void B(const Vendor& v) {**

**// ...**

**v.v();**

**v.f();**

**// ...**

**} ///:~**

***Listing 21-21. Illustrates Fixing of the Mess in Listing 21-20 using MI***

**//: C21:Paste.cpp**

**//{L} Vendor**

**// Fixing a mess with MI.**

**#include <iostream>**

**#include "Vendor.h"**

**using namespace std;**

**class MyBase { // Repair Vendor interface**

**public:**

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

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

**// New interface function:**

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

**virtual ~MyBase() { cout << "~MyBase()" << endl; }**

**};**

**class Paste1 : public MyBase, public Vendor1 {**

**public:**

**void v() const {**

**cout << "Paste1::v()" << endl;**

**Vendor1::v();**

**}**

**void f() const {**

**cout << "Paste1::f()" << endl;**

**Vendor1::f();**

**}**

**void g() const { cout << "Paste1::g()” << endl; }**

**~Paste1() { cout << "~Paste1()” << endl; }**

**};**

**int main() {**

**Paste1& p1p = \*new Paste1;**

**MyBase& mp = p1p; // Upcast**

**cout << "calling f()” << endl;**

**mp.f();  // Right behavior**

**cout << "calling g()” << endl;**

**mp.g(); // New behavior**

**cout << "calling A(p1p)” << endl;**

**A(p1p); // Same old behavior**

**cout << "calling B(p1p)” << endl;**

**B(p1p);  // Same old behavior**

**cout << "delete mp” << endl;**

**// Deleting a reference to a heap object:**

**delete &mp; // Right behavior**

**} ///:~**