Chapter 20

RUNTIME TYPE IDENTIFICATION (RTTI)

***Listing 20-1. Checking Casts at Runtime***

**//: C20:CheckedCast.cpp**

**// Checks casts at runtime.**

**#include <iostream>**

**#include <vector>**

**#include "../purge.h" // SEE ahead in this Section**

**using namespace std;**

**class Security {**

**protected:**

**enum { BASEID = 0 };**

**public:**

**virtual ~Security() {}**

**virtual bool isA(int id) { return (id == BASEID); }**

**};**

**class Stock : public Security {**

**typedef Security Super;**

**protected:**

**enum { OFFSET = 1, TYPEID = BASEID + OFFSET };**

**public:**

**bool isA(int id) {**

**return id == TYPEID || Super::isA(id);**

**}**

**static Stock\* dynacast(Security\* s) {**

**return (s->isA(TYPEID)) ? static\_cast<Stock\*>(s) : 0;**

**}**

**};**

**class Bond : public Security {**

**typedef Security Super;**

**protected:**

**enum { OFFSET = 2, TYPEID = BASEID + OFFSET };**

**public:**

**bool isA(int id) {**

**return id == TYPEID || Super::isA(id);**

**}**

**static Bond\* dynacast(Security\* s) {**

**return (s->isA(TYPEID)) ? static\_cast<Bond\*>(s) : 0;**

**}**

**};**

**class Investment : public Security {**

**typedef Security Super;**

**protected:**

**enum { OFFSET = 3, TYPEID = BASEID + OFFSET };**

**public:**

**bool isA(int id) {**

**return id == TYPEID || Super::isA(id);**

**}**

**static Investment\* dynacast(Security\* s) {**

**return (s->isA(TYPEID)) ?**

**static\_cast<Investment\*>(s) : 0;**

**}**

**void special() {**

**cout << "special Investment function" << endl;**

**}**

**};**

**class Metal : public Investment {**

**typedef Investment Super;**

**protected:**

**enum { OFFSET = 4, TYPEID = BASEID + OFFSET };**

**public:**

**bool isA(int id) {**

**return id == TYPEID || Super::isA(id);**

**}**

**static Metal\* dynacast(Security\* s) {**

**return (s->isA(TYPEID)) ? static\_cast<Metal\*>(s) : 0;**

**}**

**};**

**int main() {**

**vector<Security\*> portfolio;**

**portfolio.push\_back(new Metal);**

**portfolio.push\_back(new Investment);**

**portfolio.push\_back(new Bond);**

**portfolio.push\_back(new Stock);**

**for(vector<Security\*>::iterator it = portfolio.begin();**

**it != portfolio.end(); ++it) {**

**Investment\* cm = Investment::dynacast(\*it);**

**if(cm)**

**cm->special();**

**else**

**cout << "not an Investment" << endl;**

**}**

**cout << "cast from intermediate pointer:" << endl;**

**Security\* sp = new Metal;**

**Investment\* cp = Investment::dynacast(sp);**

**if(cp) cout << "  it's an Investment" << endl;**

**Metal\* mp = Metal::dynacast(sp);**

**if(mp) cout << "  it's a Metal too!" << endl;**

**purge(portfolio);**

**} ///:~**

**//: :purge.h**

**// Delete pointers in an STL sequence container**

**#ifndef PURGE\_H**

**#define PURGE\_H**

**#include <algorithm>**

**template<class Seq> void purge(Seq& c) {**

**typename Seq::iterator i;**

**for(i = c.begin(); i != c.end(); i++) {**

**delete \*i;**

**\*i = 0;**

**}**

**}**

**// Iterator version:**

**template<class InpIt>**

**void purge(InpIt begin, InpIt end) {**

**while(begin != end) {**

**delete \*begin;**

**\*begin = 0;**

**begin++;**

**}**

**}**

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

***Listing 20-2. Modifying Listing 20-1 Using dynamic\_cast***

**//: C20:Security.h**

**#ifndef SECURITY\_H**

**#define SECURITY\_H**

**#include <iostream>**

**class Security {**

**public:**

**virtual ~Security() {}**

**};**

**class Stock : public Security {};**

**class Bond : public Security {};**

**class Investment : public Security {**

**public:**

**void special() {**

**std::cout << "special Investment function” << std::endl;**

**}**

**};**

**class Metal : public Investment {};**

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

**//: C20:CheckedCast2.cpp**

**// Uses RTTI’s dynamic\_cast.**

**#include <vector>**

**#include "../purge.h"**

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

**using namespace std;**

**int main() {**

**vector<Security\*> portfolio;**

**portfolio.push\_back(new Metal);**

**portfolio.push\_back(new Investment);**

**portfolio.push\_back(new Bond);**

**portfolio.push\_back(new Stock);**

**for(vector<Security\*>::iterator it =**

**portfolio.begin();**

**it != portfolio.end(); ++it) {**

**Investment\* cm = dynamic\_cast<Investment\*>(\*it);**

**if(cm)**

**cm->special();**

**else**

**cout << "not a Investment" << endl;**

**}**

**cout << "cast from intermediate pointer:” << endl;**

**Security\* sp = new Metal;**

**Investment\* cp = dynamic\_cast<Investment\*>(sp);**

**if(cp) cout << "  it's an Investment” << endl;**

**Metal\* mp = dynamic\_cast<Metal\*>(sp);**

**if(mp) cout << "  it's a Metal too!” << endl;**

**purge(portfolio);**

**} ///:~**

***Listing 20-3. Catching a bad\_cast Exception***

**//: C20:CatchBadCast.cpp**

**#include <typeinfo>**

**#include "Security.h"**

**using namespace std;**

**int main() {**

**Metal m;**

**Security& s = m;**

**try {**

**Investment& c = dynamic\_cast<Investment&>(s);**

**cout << "It's an Investment" << endl;**

**} catch(bad\_cast&) {**

**cout << "s is not an Investment type" << endl;**

**}**

**try {**

**Bond& b = dynamic\_cast<Bond&>(s);**

**cout << "It's a Bond" << endl;**

**} catch(bad\_cast&) {**

**cout << "It's not a Bond type" << endl;**

**}**

**} ///:~**

***Listing 20-4. Illustrating use of the typeid Operator***

**//: C20:TypeInfo.cpp**

**// Illustrates the typeid operator.**

**#include <iostream>**

**#include <typeinfo>**

**using namespace std;**

**struct PolyBase { virtual ~PolyBase() {} };**

**struct PolyDer : PolyBase { PolyDer() {} };**

**struct NonPolyBase {};**

**struct NonPolyDer : NonPolyBase { NonPolyDer(int) {} };**

**int main() {**

**// Test polymorphic Types**

**const PolyDerpd;**

**const PolyBase\* ppb = &pd;**

**cout << typeid(ppb).name() << endl;**

**cout << typeid(\*ppb).name() << endl;**

**cout << boolalpha << (typeid(\*ppb) == typeid(pd))**

**<< endl;**

**cout << (typeid(PolyDer) == typeid(const PolyDer))**

**<< endl;**

**// Test non-polymorphic Types**

**const NonPolyDernpd(1);**

**const NonPolyBase\* nppb = &npd;**

**cout << typeid(nppb).name() << endl;**

**cout << typeid(\*nppb).name() << endl;**

**cout << (typeid(\*nppb) == typeid(npd)) << endl;**

**// Test a built-in type**

**int i;**

**cout << typeid(i).name() << endl;**

**} ///:~**

***Listing 20-5. Illustrates RTTI and Nesting***

**//: C20:RTTIandNesting.cpp**

**#include <iostream>**

**#include <typeinfo>**

**using namespace std;**

**class One {**

**class Nested {};**

**Nested\* n;**

**public:**

**One() : n(new Nested) {}**

**~One() { delete n; }**

**Nested\* nested() { return n; }**

**};**

**int main() {**

**One o;**

**cout << typeid(\*o.nested()).name() << endl;**

**} ///:~**

***Listing 20-6. Illustrates Intermediate Casting***

**//: C20:IntermediateCast.cpp**

**#include <cassert>**

**#include <typeinfo>**

**using namespace std;**

**class B1 {**

**public:**

**virtual ~B1() {}**

**};**

**class B2 {**

**public:**

**virtual ~B2() {}**

**};**

**class MI : public B1, public B2 {};**

**class Mi2 : public MI {};**

**int main() {**

**B2\* b2 = new Mi2;**

**Mi2\* mi2 = dynamic\_cast<Mi2\*>(b2);**

**MI\* mi = dynamic\_cast<MI\*>(b2);**

**B1\* b1 = dynamic\_cast<B1\*>(b2);**

**assert(typeid(b2) != typeid(Mi2\*));**

**assert(typeid(b2) == typeid(B2\*));**

**delete b2;**

**} ///:~**

***Listing 20-7. Illustrates RTTI and void Pointers***

**//: C20:VoidRTTI.cpp**

**// RTTI & void pointers.**

**//!#include <iostream>**

**#include <typeinfo>**

**using namespace std;**

**classStimpy {**

**public:**

**virtual void happy() {}**

**virtual void joy() {}**

**virtual ~Stimpy() {}**

**};**

**int main() {**

**void\* v = new Stimpy;**

**// Error:**

**//!  Stimpy\* s = dynamic\_cast<Stimpy\*>(v);**

**// Error:**

**//!  cout<<typeid(\*v).name() <<endl;**

**} ///:~**

***Listing 20-8. Printing the Order of Constructor/Destructor Calls***

**//: C20:ConstructorOrder.cpp**

**// Order of constructor calls.**

**#include <iostream>**

**#include <typeinfo>**

**using namespace std;**

**template<int id> class Announce {**

**public:**

**Announce() {**

**cout << typeid(\*this).name() << " constructor" << endl;**

**}**

**~Announce() {**

**cout << typeid(\*this).name() << " destructor" << endl;**

**}**

**};**

**class X : public Announce<0> {**

**Announce<1> m1;**

**Announce<2> m2;**

**public:**

**X() { cout << "X::X()" << endl; }**

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

**};**

**int main() { X x; } ///:~**

***Listing 20-9. Illustrates RTTI and Multiple Inheritance***

**//: C20:RTTIandMultipleInheritance.cpp**

**#include <iostream>**

**#include <typeinfo>**

**using namespace std;**

**class BB {**

**public:**

**virtual void f() {}**

**virtual ~BB() {}**

**};**

**class B1 : virtual public BB {};**

**class B2 : virtual public BB {};**

**class MI : public B1, public B2 {};**

**int main() {**

**BB\* bbp = new MI; // Upcast**

**// Proper name detection:**

**cout << typeid(\*bbp).name() << endl;**

**// Dynamic\_cast works properly:**

**MI\* mip = dynamic\_cast<MI\*>(bbp);**

**// Can't force old-style cast:**

**//! MI\* mip2 = (MI\*)bbp; // Compile error**

**} ///:~**

***Listing 20-10. Simulating a Trash Recycler***

**//: C20:Trash.h**

**// Describing trash.**

**#ifndef TRASH\_H**

**#define TRASH\_H**

**#include <iostream>**

**class Trash {**

**float \_weight;**

**public:**

**Trash(float wt) : \_weight(wt) {}**

**virtual float value() const = 0;**

**float weight() const { return \_weight; }**

**virtual ~Trash() {**

**std::cout << "~Trash()" << std::endl;**

**}**

**};**

**class Aluminum : public Trash {**

**static float val;**

**public:**

**Aluminum(float wt) : Trash(wt) {}**

**float value() const { return val; }**

**static void value(float newval) {**

**val = newval;**

**}**

**};**

**class Paper : public Trash {**

**static float val;**

**public:**

**Paper(float wt) : Trash(wt) {}**

**float value() const { return val; }**

**static void value(float newval) {**

**val = newval;**

**}**

**};**

**class Glass : public Trash {**

**static float val;**

**public:**

**Glass(float wt) : Trash(wt) {}**

**float value() const { return val; }**

**static void value(float newval) {**

**val = newval;**

**}**

**};**

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

***Listing 20-11. Implementing the header file in Listing 20-10 (Trash.h)***

**//: C20:Trash.cpp {O}**

**// A Trash Recycler.**

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

**float Aluminum::val = 1.67;**

**float Paper::val = 0.10;**

**float Glass::val = 0.23;**

**///:~**

***Listing 20-12. Illustrates Recycling using the sumValue( ) Template***

**//: C20:Recycle.cpp**

**//{L} Trash**

**// A Trash Recycler.**

**#include <cstdlib>**

**#include <ctime>**

**#include <iostream>**

**#include <typeinfo>**

**#include <vector>**

**#include "Trash.h"**

**#include "../purge.h"**

**using namespace std;**

**// Sums up the value of the Trash in a bin:**

**template<class Container>**

**void sumValue(Container& bin, ostream&os) {**

**typename Container::iterator tally = bin.begin();**

**floatval = 0;**

**while(tally != bin.end()) {**

**val += (\*tally)->weight() \* (\*tally)->value();**

**os << "weight of " << typeid(\*\*tally).name()**

**<< " = " << (\*tally)->weight() << endl;**

**++tally;**

**}**

**os << "Total value = " << val << endl;**

**}**

**int main() {**

**srand(time(0)); // Seed the random number generator**

**vector<Trash\*> bin;**

**// Fill up the Trash bin:**

**for(int i = 0; i < 30; i++)**

**switch(rand() % 3) {**

**case 0 :**

**bin.push\_back(new Aluminum((rand() % 1000)/10.0));**

**break;**

**case 1 :**

**bin.push\_back(new Paper((rand() % 1000)/10.0));**

**break;**

**case 2 :**

**bin.push\_back(new Glass((rand() % 1000)/10.0));**

**break;**

**}**

**// Note: bins hold exact type of object, not base type:**

**vector<Glass\*> glassBin;**

**vector<Paper\*> paperBin;**

**vector<Aluminum\*> alumBin;**

**vector<Trash\*>::iterator sorter = bin.begin();**

**// Sort the Trash:**

**while(sorter != bin.end()) {**

**Aluminum\* ap = dynamic\_cast<Aluminum\*>(\*sorter);**

**Paper\* pp = dynamic\_cast<Paper\*>(\*sorter);**

**Glass\* gp = dynamic\_cast<Glass\*>(\*sorter);**

**if(ap) alumBin.push\_back(ap);**

**else if(pp) paperBin.push\_back(pp);**

**else if(gp) glassBin.push\_back(gp);**

**++sorter;**

**}**

**sumValue(alumBin, cout);**

**sumValue(paperBin, cout);**

**sumValue(glassBin, cout);**

**sumValue(bin, cout);**

**purge(bin);**

**} ///:~**

***Listing 20-13. Illustrates Recycling using a map***

**//: C20:Recycle2.cpp**

**//{L} Trash**

**// Recyling with a map.**

**#include <cstdlib>**

**#include <ctime>**

**#include <iostream>**

**#include <map>**

**#include <typeinfo>**

**#include <utility>**

**#include <vector>**

**#include "Trash.h"**

**#include "../purge.h"**

**using namespace std;**

**// Comparator for type\_info pointers**

**struct TInfoLess {**

**bool operator()(const type\_info\* t1, const type\_info\* t2)**

**const { return t1->before(\*t2); }**

**};**

**typedef map<const type\_info\*, vector<Trash\*>, TInfoLess>**

**TrashMap;**

**// Sums up the value of the Trash in a bin:**

**void sumValue(const TrashMap::value\_type& p, ostream& os) {**

**vector<Trash\*>::const\_iterator tally = p.second.begin();**

**floatval = 0;**

**while(tally != p.second.end()) {**

**val += (\*tally)->weight() \* (\*tally)->value();**

**os << "weight of "**

**<< p.first->name()  // type\_info::name()**

**<< " = " << (\*tally)->weight() << endl;**

**++tally;**

**}**

**os << "Total value = " << val << endl;**

**}**

**int main() {**

**srand(time(0)); // Seed the random number generator**

**TrashMap bin;**

**// Fill up the Trash bin:**

**for(int i = 0; i < 30; i++) {**

**Trash\* tp;**

**switch(rand() % 3) {**

**case 0 :**

**tp = new Aluminum((rand() % 1000)/10.0);**

**break;**

**case 1 :**

**tp = new Paper((rand() % 1000)/10.0);**

**break;**

**case 2 :**

**tp = new Glass((rand() % 1000)/10.0);**

**break;**

**}**

**bin[&typeid(\*tp)].push\_back(tp);**

**}**

**// Print sorted results**

**for(TrashMap::iterator p = bin.begin();**

**p != bin.end(); ++p) {**

**sumValue(\*p, cout);**

**purge(p->second);**

**}**

**} ///:~**