Chapter 12

OPERATOR OVERLOADING

***Listing 12-1. Illustrating the Syntax for Operator Overloading***

**//: C12:OperatorOverloadingSyntax.cpp**

**#include <iostream>**

**using namespace std;**

**class Integer {**

**int i;**

**public:**

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

**const Integer**

**operator+(const Integer& rv) const {**

**cout << "operator+" << endl;**

**return Integer(i + rv.i);**

**}**

**Integer&**

**operator+=(const Integer& rv) {**

**cout << "operator+=" << endl;**

**i += rv.i;**

**return \*this;**

**}**

**};**

**int main() {**

**cout << "built-in types:" << endl;**

**int i = 1, j = 2, k = 3;**

**k += i + j;**

**cout << "user-defined types:" << endl;**

**Integer ii(1), jj(2), kk(3);**

**kk += ii + jj;**

**} ///:~**

***Listing 12-2. Illustrating the Syntax for Overloading Unary Operators***

**//: C12:OverloadingUnaryOperators.cpp**

**#include <iostream>**

**using namespace std;**

**// Non-member functions:**

**class Integer {**

**long i;**

**Integer\* This() { return this; }**

**public:**

**Integer(long ll = 0) : i(ll) {}**

**// No side effects takes const& argument:**

**friend const Integer&**

**operator+(const Integer& a);**

**friend const Integer**

**operator-(const Integer& a);**

**friend const Integer**

**operator~(const Integer& a);**

**friend Integer\***

**operator&(Integer& a);**

**friend int**

**operator!(const Integer& a);**

**// Side effects have non-const& argument:**

**// Prefix:**

**friend const Integer&**

**operator++(Integer& a);**

**// Postfix:**

**friend const Integer**

**operator++(Integer& a, int);**

**// Prefix:**

**friend const Integer&**

**operator--(Integer& a);**

**// Postfix:**

**friend const Integer**

**operator--(Integer& a, int);**

**};**

**// Global operators:**

**const Integer& operator+(const Integer& a) {**

**cout << "+Integer\n";**

**return a; // Unary + has no effect**

**}**

**const Integer operator-(const Integer& a) {**

**cout << "-Integer\n";**

**return Integer(-a.i);**

**}**

**const Integer operator~(const Integer& a) {**

**cout << "~Integer\n";**

**return Integer(~a.i);**

**}**

**Integer\* operator&(Integer& a) {**

**cout << "&Integer\n";**

**return a.This(); // &a is recursive!**

**}**

**int operator!(const Integer& a) {**

**cout << "!Integer\n";**

**return !a.i;**

**}**

**// Prefix; return incremented value**

**const Integer& operator++(Integer& a)**

**cout << "++Integer\n";**

**a.i++;**

**return a;**

**}**

**// Postfix; return the value before increment:**

**const Integer operator++(Integer& a, int) {**

**cout << "Integer++\n";**

**Integer before(a.i);**

**a.i++;**

**return before;**

**}**

**// Prefix; return decremented value**

**const Integer& operator--(Integer& a) {**

**cout << "--Integer\n";**

**a.i--;**

**return a;**

**}**

**// Postfix; return the value before decrement:**

**const Integer operator--(Integer& a, int) {**

**cout << "Integer--\n";**

**Integer before(a.i);**

**a.i--;**

**return before;**

**}**

**// Show that the overloaded operators work:**

**void f(Integer a) {**

**+a;**

**-a;**

**~a;**

**Integer\* ip = &a;**

**!a;**

**++a;**

**a++;**

**--a;**

**a--;**

**}**

**// Member functions (implicit "this"):**

**class Byte {**

**unsigned char b;**

**public:**

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

**// No side effects: const member function:**

**const Byte& operator+() const {**

**cout << "+Byte\n";**

**return \*this;**

**}**

**const Byte operator-() const {**

**cout << "-Byte\n";**

**return Byte(-b);**

**}**

**const Byte operator~() const {**

**cout << "~Byte\n";**

**return Byte(~b);**

**}**

**Byte operator!() const {**

**cout << "!Byte\n";**

**return Byte(!b);**

**}**

**Byte\* operator&() {**

**cout << "&Byte\n";**

**return this;**

**}**

**// Side effects: non-const member function:**

**const Byte& operator++() { // Prefix**

**cout << "++Byte\n";**

**b++;**

**return \*this;**

**}**

**const Byte operator++(int) { // Postfix**

**cout << "Byte++\n";**

**Byte before(b);**

**b++;**

**return before;**

**}**

**const Byte& operator--() { // Prefix**

**cout << "--Byte\n";**

**--b;**

**return \*this;**

**}**

**const Byte operator--(int) { // Postfix**

**cout << "Byte--\n";**

**Byte before(b);**

**--b;**

**return before;**

**}**

**};**

**void g(Byte b) {**

**+b;**

**-b;**

**~b;**

**Byte\* bp = &b;**

**!b;**

**++b;**

**b++;**

**--b;**

**b--;**

**}**

**int main() {**

**Integer a;**

**f(a);**

**Byte b;**

**g(b);**

**} ///:~*****Listing 12-3. Illustrating the Syntax for Overloading Binary Operators***

***(for Non-member Overloaded Operators)***

**//: C12:Integer.h**

**// Non-member overloaded operators**

**#ifndef INTEGER\_H**

**#define INTEGER\_H**

**#include <iostream>**

**// Non-member functions:**

**class Integer {**

**long i;**

**public:**

**Integer(long ii = 0) : i(ii) {}**

**// Operators that create new, modified value:**

**friend const Integer**

**operator+(const Integer& left,**

**const Integer& right);**

**friend const Integer**

**operator-(const Integer& left,**

**const Integer& right);**

**friend const Integer**

**operator\*(const Integer& left,**

**const Integer& right);**

**friend const Integer**

**operator/(const Integer& left,**

**const Integer& right);**

**friend const Integer**

**operator%(const Integer& left,**

**const Integer& right);**

**friend const Integer**

**operator^(const Integer& left,**

**const Integer& right);**

**friend const Integer**

**operator&(const Integer& left,**

**const Integer& right);**

**friend const Integer**

**operator|(const Integer& left,**

**const Integer& right);**

**friend const Integer**

**operator<<(const Integer& left,**

**const Integer& right);**

**friend const Integer**

**operator>>(const Integer& left,**

**const Integer& right);**

**// Assignments modify & return lvalue:**

**friend Integer&**

**operator+=(Integer& left,**

**const Integer& right);**

**friend Integer&**

**operator-=(Integer& left,**

**const Integer& right);**

**friend Integer&**

**operator\*=(Integer& left,**

**const Integer& right);**

**friend Integer&**

**operator/=(Integer& left,**

**const Integer& right);**

**friend Integer&**

**operator%=(Integer& left,**

**const Integer& right);**

**friend Integer&**

**operator^=(Integer& left,**

**const Integer& right);**

**friend Integer&**

**operator&=(Integer& left,**

**const Integer& right);**

**friend Integer&**

**operator|=(Integer& left,**

**const Integer& right);**

**friend Integer&**

**operator>>=(Integer& left,**

**const Integer& right);**

**friend Integer&**

**operator<<=(Integer& left,**

**const Integer& right);**

**// Conditional operators return true/false:**

**friend int**

**operator==(const Integer& left,**

**const Integer& right);**

**friend int**

**operator!=(const Integer& left,**

**const Integer& right);**

**friend int**

**operator<(const Integer& left,**

**const Integer& right);**

**friend int**

**operator>(const Integer& left,**

**const Integer& right);**

**friend int**

**operator<=(const Integer& left,**

**const Integer& right);**

**friend int**

**operator>=(const Integer& left,**

**const Integer& right);**

**friend int**

**operator&&(const Integer& left,**

**const Integer& right);**

**friend int**

**operator||(const Integer& left,**

**const Integer& right);**

**// Write the contents to an ostream:**

**void print(std::ostream& os) const { os << i; }**

**};**

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

**//: C12:Integer.cpp {O}**

**// Implementation of overloaded operators**

**#include "Integer.h" // TO be INCLUDED from Header FILE above // above**

**#include "../require.h" // TO be INCLUDED From Header FILE in *Chapter 9***

**const Integer**

**operator+(const Integer& left,**

**const Integer& right) {**

**return Integer(left.i + right.i);**

**}**

**const Integer**

**operator-(const Integer& left,**

**const Integer& right) {**

**return Integer(left.i - right.i);**

**}**

**const Integer**

**operator\*(const Integer& left,**

**const Integer& right) {**

**return Integer(left.i \* right.i);**

**}**

**const Integer**

**operator/(const Integer& left,**

**const Integer& right) {**

**require(right.i != 0, "divide by zero");**

**return Integer(left.i / right.i);**

**}**

**const Integer**

**operator%(const Integer& left,**

**const Integer& right) {**

**require(right.i != 0, "modulo by zero");**

**return Integer(left.i % right.i);**

**}**

**const Integer**

**operator^(const Integer& left,**

**const Integer& right) {**

**return Integer(left.i ^ right.i);**

**}**

**const Integer**

**operator&(const Integer& left,**

**const Integer& right) {**

**return Integer(left.i & right.i);**

**}**

**const Integer**

**operator|(const Integer& left,**

**const Integer& right) {**

**return Integer(left.i | right.i);**

**}**

**const Integer**

**operator<<(const Integer& left,**

**const Integer& right) {**

**return Integer(left.i << right.i);**

**}**

**const Integer**

**operator>>(const Integer& left,**

**const Integer& right) {**

**return Integer(left.i >> right.i);**

**}**

**// Assignments modify & return lvalue:**

**Integer& operator+=(Integer& left,**

**const Integer& right) {**

**if(&left == &right) {/\* self-assignment \*/}**

**left.i += right.i;**

**return left;**

**}**

**Integer& operator-=(Integer& left,**

**const Integer& right) {**

**if(&left == &right) {/\* self-assignment \*/}**

**left.i -= right.i;**

**return left;**

**}**

**Integer& operator\*=(Integer& left,**

**const Integer& right) {**

**if(&left == &right) {/\* self-assignment \*/}**

**left.i \*= right.i;**

**return left;**

**}**

**Integer& operator/=(Integer& left,**

**const Integer& right) {**

**require(right.i != 0, "divide by zero");**

**if(&left == &right) {/\* self-assignment \*/}**

**left.i /= right.i;**

**return left;**

**}**

**Integer& operator%=(Integer& left,**

**const Integer& right) {**

**require(right.i != 0, "modulo by zero");**

**if(&left == &right) {/\* self-assignment \*/}**

**left.i %= right.i;**

**return left;**

**}**

**Integer& operator^=(Integer& left,**

**const Integer& right) {**

**if(&left == &right) {/\* self-assignment \*/}**

**left.i ^= right.i;**

**return left;**

**}**

**Integer& operator&=(Integer& left,**

**const Integer& right) {**

**if(&left == &right) {/\* self-assignment \*/}**

**left.i &= right.i;**

**return left;**

**}**

**Integer& operator|=(Integer& left,**

**const Integer& right) {**

**if(&left == &right) {/\* self-assignment \*/}**

**left.i |= right.i;**

**return left;**

**}**

**Integer& operator>>=(Integer& left,**

**const Integer& right) {**

**if(&left == &right) {/\* self-assignment \*/}**

**left.i >>= right.i;**

**return left;**

**}**

**Integer& operator<<=(Integer& left,**

**const Integer& right) {**

**if(&left == &right) {/\* self-assignment \*/}**

**left.i <<= right.i;**

**return left;**

**}**

**// Conditional operators return true/false:**

**int operator==(const Integer& left,**

**const Integer& right) {**

**return left.i == right.i;**

**}**

**int operator!=(const Integer& left,**

**const Integer& right) {**

**return left.i != right.i;**

**}**

**int operator<(const Integer& left,**

**const Integer& right) {**

**return left.i < right.i;**

**}**

**int operator>(const Integer& left,**

**const Integer& right) {**

**return left.i > right.i;**

**}**

**int operator<=(const Integer& left,**

**const Integer& right) {**

**return left.i <= right.i;**

**}**

**int operator>=(const Integer& left,**

**const Integer& right) {**

**return left.i >= right.i;**

**}**

**int operator&&(const Integer& left,**

**const Integer& right) {**

**return left.i && right.i;**

**}**

**int operator||(const Integer& left,**

**const Integer& right) {**

**return left.i || right.i;**

**} ///:~**

**//: C12:IntegerTest.cpp**

**//{L} Integer**

**#include "Integer.h"**

**#include <fstream>**

**using namespace std;**

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

**void h(Integer& c1, Integer& c2) {**

**// A complex expression:**

**c1 += c1 \* c2 + c2 % c1;**

**#define TRY(OP) \**

**out << "c1 = "; c1.print(out); \**

**out << ", c2 = "; c2.print(out); \**

**out << "; c1 " #OP " c2 produces "; \**

**(c1 OP c2).print(out); \**

**out << endl;**

**TRY(+) TRY(-) TRY(\*) TRY(/)**

**TRY(%) TRY(^) TRY(&) TRY(|)**

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

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

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

**// Conditionals:**

**#define TRYC(OP) \**

**out << "c1 = "; c1.print(out); \**

**out << ", c2 = "; c2.print(out); \**

**out << "; c1 " #OP " c2 produces "; \**

**out << (c1 OP c2); \**

**out << endl;**

**TRYC(<) TRYC(>) TRYC(==) TRYC(!=) TRYC(<=)**

**TRYC(>=) TRYC(&&) TRYC(||)**

**}**

**int main() {**

**cout << "friend functions" << endl;**

**Integer c1(47), c2(9);**

**h(c1, c2);**

**} ///:~**

***Listing 12-4. Illustrating the Syntax for Overloading Binary Operators***

***(for Member Overloaded Operators)***

**//: C12:Byte.h**

**// Member overloaded operators**

**#ifndef BYTE\_H**

**#define BYTE\_H**

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

**#include <iostream>**

**// Member functions (implicit "this"):**

**class Byte {**

**unsigned char b;**

**public:**

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

**// No side effects: const member function:**

**const Byte**

**operator+(const Byte& right) const {**

**return Byte(b + right.b);**

**}**

**const Byte**

**operator-(const Byte& right) const {**

**return Byte(b - right.b);**

**}**

**const Byte**

**operator\*(const Byte& right) const {**

**return Byte(b \* right.b);**

**}**

**const Byte**

**operator/(const Byte& right) const {**

**require(right.b != 0, "divide by zero");**

**return Byte(b / right.b);**

**}**

**const Byte**

**operator%(const Byte& right) const {**

**require(right.b != 0, "modulo by zero");**

**return Byte(b % right.b);**

**}**

**const Byte**

**operator^(const Byte& right) const {**

**return Byte(b ^ right.b);**

**}**

**const Byte**

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

**return Byte(b & right.b);**

**}**

**const Byte**

**operator|(const Byte& right) const {**

**return Byte(b | right.b);**

**}**

**const Byte**

**operator<<(const Byte& right) const {**

**return Byte(b << right.b);**

**}**

**const Byte**

**operator>>(const Byte& right) const {**

**return Byte(b >> right.b);**

**}**

**// Assignments modify & return lvalue.**

**// operator= can only be a member function:**

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

**// Handle self-assignment:**

**if(this == &right) return \*this;**

**b = right.b;**

**return \*this;**

**}**

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

**if(this == &right) {/\* self-assignment \*/}**

**b += right.b;**

**return \*this;**

**}**

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

**if(this == &right) {/\* self-assignment \*/}**

**b -= right.b;**

**return \*this;**

**}**

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

**if(this == &right) {/\* self-assignment \*/}**

**b \*= right.b;**

**return \*this;**

**}**

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

**require(right.b != 0, "divide by zero");**

**if(this == &right) {/\* self-assignment \*/}**

**b /= right.b;**

**return \*this;**

**}**

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

**require(right.b != 0, "modulo by zero");**

**if(this == &right) {/\* self-assignment \*/}**

**b %= right.b;**

**return \*this;**

**}**

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

**if(this == &right) {/\* self-assignment \*/}**

**b ^= right.b;**

**return \*this;**

**}**

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

**if(this == &right) {/\* self-assignment \*/}**

**b &= right.b;**

**return \*this;**

**}**

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

**if(this == &right) {/\* self-assignment \*/}**

**b |= right.b;**

**return \*this;**

**}**

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

**if(this == &right) {/\* self-assignment \*/}**

**b >>= right.b;**

**return \*this;**

**}**

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

**if(this == &right) {/\* self-assignment \*/}**

**b <<= right.b;**

**return \*this;**

**}**

**// Conditional operators return true/false:**

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

**return b == right.b;**

**}**

**int operator!=(const Byte& right) const {**

**return b != right.b;**

**}**

**int operator<(const Byte& right) const {**

**return b < right.b;**

**}**

**int operator>(const Byte& right) const {**

**return b > right.b;**

**}**

**int operator<=(const Byte& right) const {**

**return b <= right.b;**

**}**

**int operator>=(const Byte& right) const {**

**return b >= right.b;**

**}**

**int operator&&(const Byte& right) const {**

**return b && right.b;**

**}**

**int operator||(const Byte& right) const {**

**return b || right.b;**

**}**

**// Write the contents to an ostream:**

**void print(std::ostream& os) const {**

**os << "0x" << std::hex << int(b) << std::dec;**

**}**

**};**

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

**//: C12:ByteTest.cpp**

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

**#include <fstream>**

**using namespace std;**

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

**void k(Byte& b1, Byte& 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:**

**Byte b3 = 92;**

**b1 = b2 = b3;**

**}**

**int main() {**

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

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

**k(b1, b2);**

**} ///:~**

***Listing 12-5. Overloading The Comma Operator***

**//: C12:OverloadingOperatorComma.cpp**

**#include <iostream>**

**using namespace std;**

**class After {**

**public:**

**const After& operator,(const After&) const {**

**cout << "After::operator,()" << endl;**

**return \*this;**

**}**

**};**

**class Before {};**

**Before& operator,(int, Before& b) {**

**cout << "Before::operator,()" << endl;**

**return b;**

**}**

**int main() {**

**After a, b;**

**a, b; // Operator comma called**

**Before c;**

**1, c; // Operator comma called**

**} ///:~**

***Listing 12-6. A Smart Pointer Example***

**//: C12:SmartPointer.cpp**

**#include <iostream>**

**#include <vector>**

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

**using namespace std;**

**class Obj {**

**static int i, j;**

**public:**

**void f() const { cout << i++ << endl; }**

**void g() const { cout << j++ << endl; }**

**};**

**// Static member definitions:**

**int Obj::i = 47;**

**int Obj::j = 11;**

**// Container:**

**class ObjContainer {**

**vector<Obj\*> a;**

**public:**

**void add(Obj\* obj) { a.push\_back(obj); }**

**friend class SmartPointer;**

**};**

**class SmartPointer {**

**ObjContainer& oc;**

**int index;**

**public:**

**SmartPointer(ObjContainer& objc) : oc(objc) {**

**index = 0;**

**}**

**// Return value indicates end of list:**

**bool operator++() { // Prefix**

**if(index >= oc.a.size()) return false;**

**if(oc.a[++index] == 0) return false;**

**return true;**

**}**

**bool operator++(int) { // Postfix**

**return operator++(); // Use prefix version**

**}**

**Obj\* operator->() const {**

**require(oc.a[index] != 0, "Zero value "**

**"returned by SmartPointer::operator->()");**

**return oc.a[index];**

**}**

**};**

**int main() {**

**const int sz = 10;**

**Obj o[sz];**

**ObjContainer oc;**

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

**oc.add(&o[i]); // Fill it up**

**SmartPointer sp(oc); // Create an iterator**

**do {**

**sp->f(); // Pointer dereference operator call**

**sp->g();**

**} while(sp++);**

**} ///:~**

***Listing 12-7. A Nested Smart Pointer/Iterator***

**//: C12:NestedSmartPointer.cpp**

**#include <iostream>**

**#include <vector>**

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

**using namespace std;**

**class Obj {**

**static int i, j;**

**public:**

**void f() { cout << i++ << endl; }**

**void g() { cout << j++ << endl; }**

**};**

**// Static member definitions:**

**int Obj::i = 47;**

**int Obj::j = 11;**

**// Container:**

**class ObjContainer {**

**vector<Obj\*> a;**

**public:**

**void add(Obj\* obj) { a.push\_back(obj); }**

**class SmartPointer;**

**friend class SmartPointer;**

**class SmartPointer {**

**ObjContainer& oc;**

**unsigned int index;**

**public:**

**SmartPointer(ObjContainer& objc) : oc(objc) {**

**index = 0;**

**}**

**// Return value indicates end of list:**

**bool operator++() { // Prefix**

**if(index >= oc.a.size()) return false;**

**if(oc.a[++index] == 0) return false;**

**return true;**

**}**

**bool operator++(int) { // Postfix**

**return operator++(); // Use prefix version**

**}**

**Obj\* operator->() const {**

**require(oc.a[index] != 0, "Zero value "**

**"returned by SmartPointer::operator->()");**

**return oc.a[index];**

**}**

**};**

**// Function to produce a smart pointer that**

**// points to the beginning of the ObjContainer:**

**SmartPointer begin() {**

**return SmartPointer(\*this);**

**}**

**};**

**int main() {**

**const int sz = 10;**

**Obj o[sz];**

**ObjContainer oc;**

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

**oc.add(&o[i]); // Fill it up**

**ObjContainer::SmartPointer sp = oc.begin();**

**do {**

**sp->f(); // Pointer dereference operator call**

**sp->g();**

**} while(++sp);**

**} ///:~**

***Listing 12-8. The Pointer-to-Member Operator***

**//: C12:PointerToMemberOperator.cpp**

**#include <iostream>**

**using namespace std;**

**class Dog {**

**public:**

**int run(int i) const {**

**cout << "run\n";**

**return i;**

**}**

**int eat(int i) const {**

**cout << "eat\n";**

**return i;**

**}**

**int sleep(int i) const {**

**cout << "ZZZ\n";**

**return i;**

**}**

**typedef int (Dog::\*PMF)(int) const;**

**// operator->\* must return an object**

**// that has an operator():**

**class FunctionObject {**

**Dog\* ptr;**

**PMF pmem;**

**public:**

**// Save the object pointer and member pointer**

**FunctionObject(Dog\* wp, PMF pmf)**

**: ptr(wp), pmem(pmf) {**

**cout << "FunctionObject constructor\n";**

**}**

**// Make the call using the object pointer**

**// and member pointer**

**int operator()(int i) const {**

**cout << "FunctionObject::operator()\n";**

**return (ptr->\*pmem)(i); // Make the call**

**}**

**};**

**FunctionObject operator->\*(PMF pmf) {**

**cout << "operator->\*" << endl;**

**return FunctionObject(this, pmf);**

**}**

**};**

**int main() {**

**Dog w;**

**Dog::PMF pmf = &Dog::run;**

**cout << (w->\*pmf)(1) << endl;**

**pmf = &Dog::sleep;**

**cout << (w->\*pmf)(2) << endl;**

**pmf = &Dog::eat;**

**cout << (w->\*pmf)(3) << endl;**

**} ///:~**

***Listing 12-9. Iostream Operator Overloading***

**//: C12:IostreamOperatorOverloading.cpp**

**// Example of non-member overloaded operators**

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

**#include <iostream>**

**#include <sstream>**

**// "String streams"**

**#include <cstring>**

**using namespace std;**

**class IntArray {**

**enum { sz = 5 };**

**int i[sz];**

**public:**

**IntArray() { memset(i, 0, sz\* sizeof(\*i)); }**

**int& operator[](int x) {**

**require(x >= 0 && x < sz,**

**"IntArray::operator[] out of range");**

**return i[x];**

**}**

**friend ostream&**

**operator<<(ostream& os, const IntArray& ia);**

**friend istream&**

**operator>>(istream& is, IntArray& ia);**

**};**

**ostream&**

**operator<<(ostream& os, const IntArray& ia) {**

**for(int j = 0; j < ia.sz; j++) {**

**os << ia.i[j];**

**if(j != ia.sz -1)**

**os << ", ";**

**}**

**os << endl;**

**return os;**

**}**

**istream& operator>>(istream& is, IntArray& ia){**

**for(int j = 0; j < ia.sz; j++)**

**is >> ia.i[j];**

**return is;**

**}**

**int main() {**

**stringstream input("47 34 56 92 103");**

**IntArray I;**

**input >> I;**

**I[4] = -1; // Use overloaded operator[]**

**cout << I;**

**} ///:~**

***Listing 12-10. Copying vs. Initialization***

**//: C12:CopyingVsInitialization.cpp**

**class Fi {**

**public:**

**Fi() {}**

**};**

**class Fee {**

**public:**

**Fee(int) {}**

**Fee(const Fi&) {}**

**};**

**int main() {**

**Fee fee = 1; // Fee(int)**

**Fi fi;**

**Fee fum = fi; // Fee(Fi)**

**} ///:~**

***Listing 12-11. Simple Assignment***

**//: C12:SimpleAssignment.cpp**

**// Simple operator=()**

**#include <iostream>**

**using namespace std;**

**class Value {**

**int a, b;**

**float c;**

**public:**

**Value(int aa = 0, int bb = 0, float cc = 0.0)**

**: a(aa), b(bb), c(cc) {}**

**Value& operator=(const Value& rv) {**

**a = rv.a;**

**b = rv.b;**

**c = rv.c;**

**return \*this;**

**}**

**friend ostream&**

**operator<<(ostream& os, const Value& rv) {**

**return os << "a = " << rv.a << ", b = "**

**<< rv.b << ", c = " << rv.c;**

**}**

**};**

**int main() {**

**Value a, b(1, 2, 3.3);**

**cout << "a: " << a << endl;**

**cout << "b: " << b << endl;**

**a = b;**

**cout << "a after assignment: " << a << endl;**

**} ///:~**

***Listing 12-12. Copying with Pointers***

**//: C12:CopyingWithPointers.cpp**

**// Solving the pointer aliasing problem by**

**// duplicating what is pointed to during**

**// assignment and copy-construction.**

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

**#include <string>**

**#include <iostream>**

**using namespace std;**

**class Dog {**

**string nm;**

**public:**

**Dog(const string& name) : nm(name) {**

**cout << "Creating Dog: " << \*this << endl;**

**}**

**// Synthesized copy-constructor & operator= are correct.**

**// Create a Dog from a Dog pointer:**

**Dog(const Dog\* dp, const string& msg)**

**: nm(dp->nm + msg) {**

**cout << "Copied dog " << \*this << " from "**

**<< \*dp << endl;**

**}**

**~Dog() {**

**cout << "Deleting Dog: " << \*this << endl;**

**}**

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

**nm = newName;**

**cout << "Dog renamed to: " << \*this << endl;**

**}**

**friend ostream&**

**operator<<(ostream& os, const Dog& d) {**

**return os << "[" << d.nm << "]";**

**}**

**};**

**class DogHouse {**

**Dog\* p;**

**string houseName;**

**public:**

**DogHouse(Dog\* dog, const string& house)**

**: p(dog), houseName(house) {}**

**DogHouse(const DogHouse& dh)**

**: p(new Dog(dh.p, " copy-constructed")),**

**houseName(dh.houseName**

**+ " copy-constructed") {}**

**DogHouse& operator=(const DogHouse& dh) {**

**// Check for self-assignment:**

**if(&dh != this) {**

**p = new Dog(dh.p, " assigned");**

**houseName = dh.houseName + " assigned";**

**}**

**return \*this;**

**}**

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

**houseName = newName;**

**}**

**Dog\* getDog() const { return p; }**

**~DogHouse() { delete p; }**

**friend ostream&**

**operator<<(ostream& os, const DogHouse& dh) {**

**return os << "[" << dh.houseName**

**<< "] contains " << \*dh.p;**

**}**

**};**

**int main() {**

**DogHouse fidos(new Dog("Fido"), "FidoHouse");**

**cout << fidos << endl;**

**DogHouse fidos2 = fidos; // Copy construction**

**cout << fidos2 << endl;**

**fidos2.getDog()->rename("Spot");**

**fidos2.renameHouse("SpotHouse");**

**cout << fidos2 << endl;**

**fidos = fidos2; // Assignment**

**cout << fidos << endl;**

**fidos.getDog()->rename("Max");**

**fidos2.renameHouse("MaxHouse");**

**} ///:~**

***Listing 12-13. Illustrating Reference Counting and Copy-on-Write***

**//: C12:ReferenceCounting.cpp**

**// Reference count, copy-on-write**

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

**#include <string>**

**#include <iostream>**

**using namespace std;**

**class Dog {**

**string nm;**

**int refcount;**

**Dog(const string& name)**

**: nm(name), refcount(1) {**

**cout << "Creating Dog: " << \*this << endl;**

**}**

**// Prevent assignment:**

**Dog& operator=(const Dog& rv);**

**public:**

**// Dogs can only be created on the heap:**

**static Dog\* make(const string& name) {**

**return new Dog(name);**

**}**

**Dog(const Dog& d)**

**: nm(d.nm + " copy"), refcount(1) {**

**cout << "Dog copy-constructor: "**

**<< \*this << endl;**

**}**

**~Dog() {**

**cout << "Deleting Dog: " << \*this << endl;**

**}**

**void attach() {**

**++refcount;**

**cout << "Attached Dog: " << \*this << endl;**

**}**

**void detach() {**

**require(refcount != 0);**

**cout << "Detaching Dog: " << \*this << endl;**

**// Destroy object if no one is using it:**

**if(--refcount == 0) delete this;**

**}**

**// Conditionally copy this Dog.**

**// Call before modifying the Dog, assign**

**// resulting pointer to your Dog\*.**

**Dog\* unalias() {**

**cout << "Unaliasing Dog: " << \*this << endl;**

**// Don't duplicate if not aliased:**

**if(refcount == 1) return this;**

**--refcount;**

**// Use copy-constructor to duplicate:**

**return new Dog(\*this);**

**}**

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

**nm = newName;**

**cout << "Dog renamed to: " << \*this << endl;**

**}**

**friend ostream&**

**operator<<(ostream& os, const Dog& d) {**

**return os << "[" << d.nm << "], rc = "**

**<< d.refcount;**

**}**

**};**

**class DogHouse {**

**Dog\* p;**

**string houseName;**

**public:**

**DogHouse(Dog\* dog, const string& house)**

**: p(dog), houseName(house) {**

**cout << "Created DogHouse: "<< \*this << endl;**

**}**

**DogHouse(const DogHouse& dh)**

**: p(dh.p),**

**houseName("copy-constructed " +**

**dh.houseName) {**

**p->attach();**

**cout << "DogHouse copy-constructor: "**

**<< \*this << endl;**

**}**

**DogHouse& operator=(const DogHouse& dh) {**

**// Check for self-assignment:**

**if(&dh != this) {**

**houseName = dh.houseName + " assigned";**

**// Clean up what you're using first:**

**p->detach();**

**p = dh.p; // Like copy-constructor**

**p->attach();**

**}**

**cout << "DogHouse operator= : "**

**<< \*this << endl;**

**return \*this;**

**}**

**// Decrement refcount, conditionally destroy**

**~DogHouse() {**

**cout << "DogHouse destructor: "**

**<< \*this << endl;**

**p->detach();**

**}**

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

**houseName = newName;**

**}**

**void unalias() { p = p->unalias(); }**

**// Copy-on-write. Anytime you modify the**

**// contents of the pointer you must**

**// first unalias it:**

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

**unalias();**

**p->rename(newName);**

**}**

**// ... or when you allow someone else access:**

**Dog\* getDog() {**

**unalias();**

**return p;**

**}**

**friend ostream&**

**operator<<(ostream& os, const DogHouse& dh) {**

**return os << "[" << dh.houseName**

**<< "] contains " << \*dh.p;**

**}**

**};**

**int main() {**

**DogHouse**

**fidos(Dog::make("Fido"), "FidoHouse"),**

**spots(Dog::make("Spot"), "SpotHouse");**

**cout << "Entering copy-construction" << endl;**

**DogHouse bobs(fidos);**

**cout << "After copy-constructing bobs" << endl;**

**cout << "fidos:" << fidos << endl;**

**cout << "spots:" << spots << endl;**

**cout << "bobs:" << bobs << endl;**

**cout << "Entering spots = fidos" << endl;**

**spots = fidos;**

**cout << "After spots = fidos" << endl;**

**cout << "spots:" << spots << endl;**

**cout << "Entering self-assignment" << endl;**

**bobs = bobs;**

**cout << "After self-assignment" << endl;**

**cout << "bobs:" << bobs << endl;**

**// Comment out the following lines:**

**cout << "Entering rename(\"Bob\")" << endl;**

**bobs.getDog()->rename("Bob");**

**cout << "After rename(\"Bob\")" << endl;**

**} ///:~**

***Listing 12-14. Illustrating Member-Wise Assignment***

**//: C12:AutomaticOperatorEquals.cpp**

**#include <iostream>**

**using namespace std;**

**class Cargo {**

**public:**

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

**cout << "inside Cargo::operator=()" << endl;**

**return \*this;**

**}**

**};**

**class Truck {**

**Cargo b;**

**};**

**int main() {**

**Truck a, b;**

**a = b; // Prints: "inside Cargo::operator=()"**

**} ///:~**

***Listing 12-15. Illustrating Automatic Type Conversion***

**//: C12:AutomaticTypeConversion.cpp**

**// Type conversion constructor**

**class One {**

**public:**

**One() {}**

**};**

**class Two {**

**public:**

**Two(const One&) {}**

**};**

**void f(Two) {}**

**int main() {**

**One one;**

**f(one); // Wants a Two, has a One**

**} ///:~**

***Listing 12-16. Illustrating Use of the Explicit Keyword***

**//: C12:ExplicitKeyword.cpp**

**// Using the "explicit" keyword**

**class One {**

**public:**

**One() {}**

**};**

**class Two {**

**public:**

**explicit Two(const One&) {}**

**};**

**void f(Two) {}**

**int main() {**

**One one;**

**//! f(one); // No auto conversion allowed**

**f(Two(one)); // OK -- user performs conversion**

**} ///:~**

***Listing 12-17. Illustrating Operator Overloading Conversion***

**//: C12:OperatorOverloadingConversion.cpp**

**class Three {**

**int i;**

**public:**

**Three(int ii = 0, int = 0) : i(ii) {}**

**};**

**class Four {**

**int x;**

**public:**

**Four(int xx) : x(xx) {}**

**operator Three() const { return Three(x); }**

**};**

**void g(Three) {}**

**int main() {**

**Four four(1);**

**g(four);**

**g(1); // Calls Three(1,0)**

**} ///:~**

***Listing 12-18. Illustrating Reflexivity in Overloading***

**//: C12:ReflexivityInOverloading.cpp**

**class Number {**

**int i;**

**public:**

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

**const Number**

**operator+(const Number& n) const {**

**return Number(i + n.i);**

**}**

**friend const Number**

**operator-(const Number&, const Number&);**

**};**

**const Number**

**operator-(const Number& n1,**

**const Number& n2) {**

**return Number(n1.i - n2.i);**

**}**

**int main() {**

**Number a(47), b(11);**

**a + b; // OK**

**a + 1; // 2nd arg converted to Number**

**//! 1 + a; // Wrong! 1st arg not of type Number**

**a - b; // OK**

**a - 1; // 2nd arg converted to Number**

**1 - a; // 1st arg converted to Number**

**} ///:~**

***Listing 12-19. Using No Automatic Type Conversion***

**//: C12:Strings1.cpp**

**// No auto type conversion**

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

**#include <cstring>**

**#include <cstdlib>**

**#include <string>**

**using namespace std;**

**class Stringc {**

**string s;**

**public:**

**Stringc(const string& str = "") : s(str) {}**

**int strcmp(const Stringc& S) const {**

**return ::strcmp(s.c\_str(), S.s.c\_str());**

**}**

**// ... etc., for every function in string.h**

**};**

**int main() {**

**Stringc s1("hello"), s2("there");**

**s1.strcmp(s2);**

**} ///:~**

***Listing 12-20. Using Automatic Type Conversion***

**//: C12:Strings2.cpp**

**// With auto type conversion**

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

**#include <cstring>**

**#include <cstdlib>**

**#include <string>**

**using namespace std;**

**class Stringc {**

**string s;**

**public:**

**Stringc(const string& str = "") : s(str) {}**

**operator const char\*() const {**

**return s.c\_str();**

**}**

**};**

**int main() {**

**Stringc s1("hello"), s2("there");**

**strcmp(s1, s2); // Standard C function**

**strspn(s1, s2); // Any string function!**

**} ///:~**

***Listing 12-21. Illustrating Ambiguity in Automatic Type Conversion***

**//: C12:TypeConversionAmbiguity.cpp**

**class Orange; // Class declaration**

**class Apple {**

**public:**

**operator Orange() const; // Convert Apple to Orange**

**};**

**class Orange {**

**public:**

**Orange(Apple); // Convert Apple to Orange**

**};**

**void f(Orange) {}**

**int main() {**

**Apple a;**

**//! f(a); // Error: ambiguous conversion**

**} ///:~**

***Listing 12-22. Illustrating “Fan-out”***

**//: C12:TypeConversionFanout.cpp**

**class Orange {};**

**class Pear {};**

**class Apple {**

**public:**

**operator Orange() const;**

**operator Pear() const;**

**};**

**// Overloaded eat():**

**void eat(Orange);**

**void eat(Pear);**

**int main() {**

**Apple c;**

**//! eat(c);**

**// Error: Apple -> Orange or Apple -> Pear ???**

**} ///:~**

***Listing 12-23. Illustrating Hidden Activities in Automatic Type Conversion***

**//: C12:CopyingVsInitialization2.cpp**

**class Fi {**

**public:**

**Fi() {}**

**};**

**class Fee {**

**public:**

**Fee(int) {}**

**Fee(const Fi&) {}**

**};**

**int main() {**

**Fee fee = 1; // Fee(int)**

**Fi fi;**

**Fee fum = fi; // Fee(Fi)**

**} ///:~**