**Operator Overloading in C++**

# Polymorphism

Polymorphism means one name many forms. It has two types

1. Static Polymorphism
2. Dynamic Polymorphism

Static polymorphism is achieved using method overloading.

Dynamic polymorphism using method overriding. It is closely related to inheritance. We can write a code that works on the superclass, and it will work with any subclass type as well.

This chapter deals with Static polymorphism using operator overloading (because operator overloads are also member function).

# Operator overloading

In C++, we can make operators to work for user defined classes. For example, we can overload an operator ‘+’ in a class like String so that we can concatenate two strings by just using +.

**What is the difference between operator functions and normal functions?**

Operator functions are same as normal functions. The only differences are, **name of an operator function is always operator keyword followed by symbol of operator** and operator functions are called when the corresponding operator is used.

**Can we overload all operators?**

Almost all operators can be overloaded except few. Following is the list of operators that cannot be overloaded.

1. . (dot)
2. .\* (dereferencing operator)
3. ::
4. ?:
5. sizeof
6. typeid Object type Operator

Reason <http://www.stroustrup.com/bs_faq2.html#overload-dot>

**Operators which can be overloaded using only member functions and not any global function**

= Assignment operator

() Function call operator

[] Subscripting operator

-> Class member access operator

**Important points about operator overloading**

1. For operator overloading to work, at least one of the operands must be a user defined class object.

Only an expression containing a user-defined type can have an overloaded operator.

1. Assignment Operator: Compiler automatically creates a default assignment operator with every class.
2. Conversion Operator: We can also write conversion operators that can be used to convert one type to another type.

Overloaded conversion operators must be a member method. Other operators can either be member method or global method.

#include <iostream>

using namespace std;

class Fraction {

int num, den;

public:

Fraction(int n, int d) : num{n}, den{d} {}

// conversion operator: return float value of fraction

operator float() const { return float(num) / float(den); }

};

int main() {

Fraction f(2, 5);

float val = f;

cout << val;

return 0;

}

Output:

0.4

1. Any constructor that can be called with a single argument works as a conversion constructor, means it can also be used for implicit conversion to the class being constructed.

#include <iostream>

using namespace std;

class Point {

int x, y;

public:

Point(int i) : x{i}, y{0} {} // conversion constructor

Point(int i, int j) : x{i}, y{j} {}

void display() { cout << " x = " << x << ", y = " << y << '\n'; }

};

int main() {

Point p1(20, 10);

p1.display();

p1 = 30; // p1 = Point(30);

p1.display();

return 0;

}

Output:

x = 20, y = 10

x = 30, y = 0

To stop implicit conversion, declare single parameter constructor explicit

explicit Point(int i) : x{i}, y{0} {} // conversion constructor

…

// error: no match for 'operator=' (operand types are 'Point' and 'int')

// p1 = 30; // p1 = Point(30);

# Operator overloading function syntax

The name of that function is operator@, in which @ represents the operator that’s being overloaded. The number of arguments in the overloaded operator’s argument list depends on two factors:

1. Whether it’s a unary operator (one argument) or a binary operator (two arguments).
2. Whether the operator is defined as a

**global function** (one argument for unary, two for binary) or a

**member function** (zero arguments for unary, one for binary – the object becomes the left-hand argument).

# Overloadable operators

* You cannot combine operators that currently have no meaning in C (such as \*\* to represent exponentiation)
* You cannot change the evaluation precedence of operators
* You cannot change the number of arguments required by an operator

## Unary operators

The following example shows the syntax to overload all the unary operators, in the form of both global functions (non-member friend functions) and as member functions.

### Global Function

#include <iostream>

using namespace std;

// Non-member functions as friend

class MyInteger {

long m\_val;

MyInteger\* MyThis() { return this; }

public:

MyInteger(long val = 0) : m\_val(val) {}

void display() { cout << "m\_val " <<m\_val << '\n'; }

// No side effects takes const& argument:

friend const MyInteger& operator+(const MyInteger& a);

friend const MyInteger operator-(const MyInteger& a);

friend const MyInteger operator~(const MyInteger& a);

friend MyInteger\* operator&(MyInteger& a);

friend int operator!(const MyInteger& a);

// Side effects have non-const& argument:

friend const MyInteger& operator++(MyInteger& a); // Prefix

friend const MyInteger operator++(MyInteger& a, int); // Postfix

friend const MyInteger& operator--(MyInteger& a); // Prefix

friend const MyInteger operator--(MyInteger& a, int); // Postfix

};

// Global operators:

const MyInteger& operator+(const MyInteger& a) {

cout << "+MyInteger ";

return a; // Unary + has no effect

}

const MyInteger operator-(const MyInteger& a) {

cout << "-MyInteger ";

return MyInteger(-a.m\_val);

}

const MyInteger operator~(const MyInteger& a) {

cout << "~MyInteger ";

return MyInteger(~a.m\_val);

}

MyInteger\* operator&(MyInteger& a) {

cout << "&MyInteger ";

return a.MyThis(); // &a is recursive!

}

int operator!(const MyInteger& a) {

cout << "!MyInteger ";

return !a.m\_val;

}

// Prefix; return incremented m\_value

const MyInteger& operator++(MyInteger& a) {

cout << "++MyInteger ";

a.m\_val++;

return a;

}

// Postfix; return the m\_value before increment:

const MyInteger operator++(MyInteger& a, int) {

cout << "MyInteger++ ";

MyInteger before(a.m\_val);

a.m\_val++;

return before;

}

// Prefix; return decremented m\_value

const MyInteger& operator--(MyInteger& a) {

cout << "--MyInteger ";

a.m\_val--;

return a;

}

// Postfix; return the m\_value before decrement:

const MyInteger operator--(MyInteger& a, int) {

cout << "MyInteger-- ";

MyInteger before(a.m\_val);

a.m\_val--;

return before;

}

int main() {

MyInteger a{5};

{ MyInteger val = +a; val.display(); }

{ MyInteger val = -a; val.display(); }

{ MyInteger val = ~a; val.display(); }

{ MyInteger\* ip = &a; ip->display(); }

{ MyInteger val = !a; val.display(); }

{ MyInteger val = ++a; val.display(); }

{ MyInteger val = a++; val.display(); }

a.display();

{ MyInteger val = --a; val.display(); }

{ MyInteger val = a--; val.display(); }

a.display();

return 0;

}

Output:

+MyInteger m\_val 5

-MyInteger m\_val -5

~MyInteger m\_val -6

&MyInteger m\_val 5

!MyInteger m\_val 0

++MyInteger m\_val 6

MyInteger++ m\_val 6

m\_val 7

--MyInteger m\_val 6

MyInteger-- m\_val 6

m\_val 5

### Member Function

#include <iostream>

using namespace std;

// member functions

class MyInteger {

long m\_val;

public:

MyInteger(long val = 0) : m\_val(val) {}

void display() { cout << "m\_val " <<m\_val << '\n'; }

// No side effects takes const& argument:

const MyInteger& operator+();

const MyInteger operator-();

const MyInteger operator~();

MyInteger\* operator&();

int operator!();

// Side effects have non-const& argument:

const MyInteger& operator++(); // Prefix

const MyInteger operator++(int); // Postfix

const MyInteger& operator--(); // Prefix

const MyInteger operator--(int); // Postfix

};

const MyInteger& MyInteger::operator+() {

cout << "+MyInteger ";

return \*this; // Unary + has no effect

}

const MyInteger MyInteger::operator-() {

cout << "-MyInteger ";

return MyInteger(-m\_val);

}

const MyInteger MyInteger::operator~() {

cout << "~MyInteger ";

return MyInteger(~m\_val);

}

MyInteger\* MyInteger::operator&() {

cout << "&MyInteger ";

return this; // &a is recursive!

}

int MyInteger::operator!() {

cout << "!MyInteger ";

return !m\_val;

}

// Prefix; return incremented m\_value

const MyInteger& MyInteger::operator++() {

cout << "++MyInteger ";

m\_val++;

return \*this;

}

// Postfix; return the m\_value before increment:

const MyInteger MyInteger::operator++(int) {

cout << "MyInteger++ ";

MyInteger before(m\_val);

m\_val++;

return before;

}

// Prefix; return decremented m\_value

const MyInteger& MyInteger::operator--() {

cout << "--MyInteger ";

m\_val--;

return \*this;

}

// Postfix; return the m\_value before decrement:

const MyInteger MyInteger::operator--(int) {

cout << "MyInteger-- ";

MyInteger before(m\_val);

m\_val--;

return before;

}

int main() {

MyInteger a{5};

{ MyInteger val = +a; val.display(); }

{ MyInteger val = -a; val.display(); }

{ MyInteger val = ~a; val.display(); }

{ MyInteger\* ip = &a; ip->display(); }

{ MyInteger val = !a; val.display(); }

{ MyInteger val = ++a; val.display(); }

{ MyInteger val = a++; val.display(); }

a.display();

{ MyInteger val = --a; val.display(); }

{ MyInteger val = a--; val.display(); }

a.display();

return 0;

}

Output:

+MyInteger m\_val 5

-MyInteger m\_val -5

~MyInteger m\_val -6

&MyInteger m\_val 5

!MyInteger m\_val 0

++MyInteger m\_val 6

MyInteger++ m\_val 6

m\_val 7

--MyInteger m\_val 6

MyInteger-- m\_val 6

m\_val 5

## Increment & decrement

When the compiler sees, for example, ++a (a pre-increment), it generates a call to operator++(a); but when it sees a++, it generates a call to operator++(a, int.) That is, the compiler differentiates between the two forms by making calls to different overloaded functions. For the member function versions, if the compiler sees ++b, it generates a call to B::operator++( ); if it sees b++ it calls B::operator++(int).

All the user sees is that a different function gets called for the prefix and postfix versions. Underneath, however, the two functions calls have different signatures, so they link to two different function bodies. The compiler passes a dummy constant value for the int argument (which is never given an identifier because the value is never used) to generate the different signature for the postfix version.

## Binary operators

Again, both global versions and member function versions are shown.

### Global Function

#include <iostream>

using namespace std;

// global friend functions

class MyInteger {

long m\_val;

public:

MyInteger(long val = 0) : m\_val(val) {}

long get\_val() const { return m\_val; }

void display() const { cout << "value " << m\_val << '\n'; }

// Operators that create new, modified value:

friend const MyInteger operator+(const MyInteger& left, const MyInteger& right);

friend const MyInteger operator-(const MyInteger& left, const MyInteger& right);

// Assignments modify & return lvalue:

friend MyInteger& operator+=(MyInteger& left, const MyInteger& right);

friend MyInteger& operator-=(MyInteger& left, const MyInteger& right);

};

const MyInteger operator+(const MyInteger& left, const MyInteger& right) {

return MyInteger(left.m\_val + right.m\_val);

}

const MyInteger operator-(const MyInteger& left, const MyInteger& right) {

return MyInteger(left.m\_val - right.m\_val);

}

// Assignments modify & return lvalue:

MyInteger& operator+=(MyInteger& left, const MyInteger& right) {

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

left.m\_val += right.m\_val;

return left;

}

MyInteger& operator-=(MyInteger& left, const MyInteger& right) {

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

left.m\_val -= right.m\_val;

return left;

}

int main() {

MyInteger c1(47), c2(9);

MyInteger t1, t2;

t1 = c1 + c2; t1.display();

t2 = c1 - c2; t2.display();

t1 += c1; t1.display();

t2 -= c2; t2.display();

t1 += t1; t1.display();

t2 -= t2; t2.display();

return 0;

}

Output:

value 56

value 38

value 103

value 29

value 206

value 0

### Member Function

#include <iostream>

using namespace std;

// member functions

class MyInteger {

long m\_val;

public:

MyInteger(long val = 0) : m\_val(val) {}

long get\_val() const { return m\_val; }

void display() const { cout << "value " << m\_val << '\n'; }

const MyInteger operator+(const MyInteger& right);

const MyInteger operator-(const MyInteger& right);

MyInteger& operator+=(const MyInteger& right);

MyInteger& operator-=(const MyInteger& right);

// operator= is only allowed to be a member function

MyInteger& operator=(const MyInteger& right);

};

const MyInteger MyInteger::operator+(const MyInteger& right) {

return MyInteger(m\_val + right.m\_val);

}

const MyInteger MyInteger::operator-(const MyInteger& right) {

return MyInteger(m\_val - right.m\_val);

}

// Assignments modify & return lvalue:

MyInteger& MyInteger::operator+=(const MyInteger& right) {

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

m\_val += right.m\_val;

return \*this;

}

MyInteger& MyInteger::operator-=(const MyInteger& right) {

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

m\_val -= right.m\_val;

return \*this;

}

MyInteger& MyInteger::operator=(const MyInteger& right) {

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

m\_val = right.m\_val;

return \*this;

}

int main() {

MyInteger c1(47), c2(9);

MyInteger t1, t2;

t1 = c1 + c2; t1.display();

t2 = c1 - c2; t2.display();

t1 += c1; t1.display();

t2 -= c2; t2.display();

t1 += t1; t1.display();

t2 -= t2; t2.display();

t2 = t1; t2.display();

return 0;

}

Output:

value 56

value 38

value 103

value 29

value 206

value 0

value 206

**operator= is only allowed to be a member function.**

Notice that all of the assignment operators have code to check for self-assignment; this is a general guideline. In some cases, this is not necessary; for example, with operator+= you often want to say A+=A and have it add A to itself. The most important place to check for self-assignment is operator= because with complicated objects disastrous results may occur. (In some cases, it’s OK, but you should always keep it in mind when writing operator=.)

## Arguments & return values

The different ways that arguments are passed and returned

1. As with any function argument, if you only need to read from the argument and not change it, default to passing it as a const reference. Only with the operator-assignments (like +=) and the operator=, which change the left-hand argument, is the left argument not a constant, but it’s still passed in as an address because it will be changed.
2. The type of return value you should select depends on the expected meaning of the operator. For example, MyInteger::operator+ must produce an MyInteger object that is the sum of the operands. This object is returned by value as a const, so the result cannot be modified as an lvalue.
3. All the assignment operators modify the lvalue. To allow the result of the assignment to be used in chained expressions, like a=b=c, it’s expected that you will return a reference to that same lvalue that was just modified. But should this reference be a const or nonconst? Although you read a=b=c from left to right, the compiler parses it from right to left, so you’re not forced to return a nonconst to support assignment chaining. However, people do sometimes expect to be able to perform an operation on the thing that was just assigned to, such as (a=b).func( ) ;to call func( ) on a after assigning b to it. Thus, the return value for all of the assignment operators should be a nonconst reference to the lvalue.
4. For the logical operators, everyone expects to get at worst an int back, and at best a bool. (Libraries developed before most compilers supported C++’s built-in bool will use int or an equivalent typedef.)

Prefix and Postfix both versions change the object and so cannot treat the object as a const. The prefix version returns the value of the object after it was changed, so you expect to get back the object that was changed. Thus, with prefix you can just return \*this as a reference. The postfix version is supposed to return the value before the value is changed, so you’re forced to create a separate object to represent that value and return it.

Now the question is: Should these be returned as const or nonconst? If you allow the object to be modified and someone writes (++a).func( ), func( ) will be operating on a itself, but with (a++).func( ), func( ) operates on the temporary object returned by the postfix operator++. **Temporary objects are automatically const,** so this would be flagged by the compiler, but for consistency’s sake it may make more sense to make them both const, as was done here.

Or you may choose to make the prefix version non-const and the postfix const. Because of the variety of meanings, you may want to give the increment and decrement operators, they will need to be considered on a case-by-case basis.

### Return by value as const

Consider the binary operator+. If you use it in an expression such as f(a+b), the result of a+b becomes a temporary object that is used in the call to f( ). Because it’s a temporary, it’s automatically const, so whether you explicitly make the return value const or not has no effect.

You can say (a+b).g( ), in which g( ) is some member function of Integer, in this case. By making the return value const, you state that only a const member function can be called for that return value. This is const-correct, because it prevents you from storing potentially valuable information in an object that will most likely be lost.

### The return optimization

When new objects are created to return by value, notice the form used. In operator+, for example:

**return Integer(left.i + right.i); // returns temporary object**

This may look at first like a “function call to a constructor,” but it’s not. The syntax is that of a temporary object; the statement says “make a temporary Integer object and return it.” Because of this, you might think that the result is the same as creating a named local object and returning that. However, it’s quite different. If you were to say instead:

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

**return tmp; // returning local object**

three things will happen. First, the tmp object is created including its constructor call. Second, the copy-constructor copies the tmp to the location of the outside return value. Third, the destructor is called for tmp at the end of the scope.

In contrast, the **“returning a temporary**” approach works quite differently. When the compiler sees you do this, it knows that you have no other need for the object it’s creating than to return it. The compiler takes advantage of this by building the object directly into the location of the outside return value. This **requires only a single ordinary constructor call (no copy-constructor is necessary) and there’s no destructor call** because you never actually create a local object. Thus, while it doesn’t cost anything but programmer awareness, it’s significantly more efficient. This is often called the return value optimization (RVO).

## Reflexivity

In the global versions, automatic type conversion may be applied to either operand, whereas with member objects, the left-hand operand must already be the proper type.

If you want both operands to be converted, the global versions can save a lot of coding.

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

}

# Unusual operators

Several additional operators have a slightly different syntax for overloading.

1. [] subscript operator
2. , comma operator
3. -> Pointer-to-member operator
4. ->\* pointer-to-member dereference operator
5. new new operator
6. delete delete operator

## [] subscript operator

The subscript, operator[ ], must be a member function and it requires a single argument. Because operator[ ]implies that the object it’s being called for acts like an array, you will often return a reference from this operator, so it can be conveniently used on the left-hand side of an equal sign.

## , comma operator

The comma operator is called when it appears next to an object of the type the comma is defined for. However, “operator,” is not called for function argument lists, only for objects that are out in the open, separated by commas. There doesn’t seem to be a lot of practical uses for this operator; it’s in the language for consistency.

## -> Pointer-to-member operator

The operator–> is generally used when you want to make an object appear to be a pointer. Since such an object has more “smarts” built into it than exist for a typical pointer, an object like this is often called a smart pointer.

A pointer dereference operator must be a member function. It has additional, atypical constraints: It must return an object (or reference to an object) that also has a pointer dereference operator, or it must return a pointer that can be used to select what the pointer dereference operator arrow is pointing at.

## ->\* pointer-to-member dereference operator

The operator–>\* is a binary operator that behaves like all the other binary operators. It is provided for those situations when you want to mimic the behaviour provided by the built-in pointer-to-member syntax, described in the previous chapter.

Just like operator->, the pointer-to-member dereference operator is generally used with some kind of object that represents a “smart pointer,”. The trick when defining operator->\* is that it must return an object for which the operator( ) can be called with the arguments for the member function you’re calling.

## new and delete

**new new operator**

**delete delete operator**

Why overload new and delete?

You might be creating and destroying so many objects of a particular class that it has become a speed bottleneck.

Heap fragmentation. By allocating objects of different sizes, it’s possible to break up the heap so that you effectively run out of storage. That is, the storage might be available, but because of fragmentation no piece is big enough to satisfy your needs. By creating your own allocator for a particular class, you can ensure this never happens.

#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) {

if(x >= 0 && x < sz)

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;

return 0;

}

Output:

47, 34, 56, 92, -1

# Operators you can’t overload

1. . (dot)
2. .\* (dereferencing operator)
3. ::
4. ?:
5. sizeof
6. typeid Object type Operator

* The member selection operator. Currently, the dot has a meaning for any member in a class, but if you allow it to be overloaded, then you couldn’t access members in the normal way; instead you’d have to use a pointer and the arrow operator->.
* The pointer to member dereference operator.\*, for the same reason as operator.
* There are no user-defined operators. That is, you can’t make up new operators that aren’t currently in the set. Part of the problem is how to determine precedence, and part of the problem is an insufficient need to account for the necessary trouble.
* You can’t change the precedence rules. They’re hard enough to remember as it is without letting people play with them.

Check examples of these operators in book mentioned in [References](#_References)

# Non-member operators

The operators may be members or non-members, and it doesn’t seem to make much difference. This usually raises the question, “Which should I choose?”

In general, if it doesn’t make any difference, they should be members, to emphasize the association between the operator and its class. When the left-hand operand is always an object of the current class, this works fine.

Guidelines for choosing between members and non-members:

|  |  |
| --- | --- |
| **Operator** | **Recommended use** |
| All unary operators | member |
| = () [ ] –> –>\* | must be member |
| += –= /= \*= ^= &= |= %= >>= <<= | member |
| All other binary Operators | non-member |

# Overloading assignment

= sign is such a fundamental operation in programming, right down to copying a register at the machine level. In addition, the copy-constructor is also sometimes invoked when the = sign is used:

MyType b;

MyType a = b; // copy constructor

a = b; // assignment

Any time you’re initializing an object using an = instead of the ordinary function-call form of the constructor, the compiler will look for a constructor that accepts whatever is on the right-hand side.

When dealing with the = sign, it’s important to keep this distinction in mind: If the object hasn’t been created yet, initialization is required; otherwise, the assignment operator= is used.

It’s even better to avoid writing code that uses the = for initialization; instead, always use the explicit constructor form.

## Behaviour of operator=

operator= can be only a member function. It is intimately connected to the object on the left side of the ‘=’. If it was possible to define operator= globally, then you might attempt to redefine the built-in ‘=’ sign:

int operator= (int, MyType); // Global = not allowed!

The compiler skirts this whole issue by forcing you to make operator= a member function.

When you’re assigning two objects of the same type, you should always check first for self-assignment: is the object being assigned to itself? In some cases, such as this one, it’s harmless if you perform the assignment operations anyway, but if changes are made to the implementation of the class, it can make a difference, and if you don’t do it as a matter of habit, you may forget and cause hard-to-find bugs.

### Pointers in classes

If the object contains pointers to other objects, simply copying a pointer means that you’ll end up with two objects pointing to the same storage location.

There are two common approaches to this problem

1. Deep Copy
2. Reference counting

**Deep Copy**

Copy whatever the pointer refers to when you do an assignment or a copy-construction

### Reference Counting

If your object requires a lot of memory or a high initialization overhead, you may want to avoid copying. A common approach to this problem is called reference counting.

Copy-construction or assignment means attaching another pointer to an existing object and incrementing the reference count. Destruction means reducing the reference count and destroying the object if the reference count goes to zero.

But what if you want to write to the object? You’d be modifying someone else’s object as well as yours. To solve this “aliasing” problem, an additional technique called **copy-on-write** is used. Before writing to a block of memory, you make sure no one else is using it. If the reference count is greater than one, you must make yourself a personal copy of that block before writing it, so you don’t disturb someone else’s turf.

### Automatic operator= creation

If programmer don’t write, the compiler will automatically create a type::operator=(type ). The behaviour of this operator mimics that of the automatically created copy-constructor; if the class contains objects (or is inherited from another class), the operator= for those objects is called recursively. This is called **member wise assignment**.

If you really don’t want people to perform assignment, declare operator= as a private function (or delete since C++11).

# When should we write our own assignment operator in C++?

The answer is same as Copy Constructor. If a class contain pointers, then there is need to write assignment operator and copy constructor.

Problem: Program without assignment operator

#include <iostream>

using namespace std;

// A class without user defined assignment operator

class Test {

int \*ptr;

public:

Test (int i = 0) { ptr = new int(i); }

void setValue (int i) { \*ptr = i; }

void print() { cout << \*ptr << endl; }

};

int main(void) {

Test t1(5);

Test t2;

t2 = t1;

t1.print(); t2.print(); cout << endl;

t1.setValue(10);

t1.print(); t2.print();

return 0;

}

Output:

5

5

10

10

Since there is no user defined assignment operator in the above program, compiler creates a default assignment operator, which copies ‘ptr’ of right-hand side to left hand side. So, both ptr’s start pointing to the same location.

We can handle the above problem in two ways.

1. Do not allow assignment of one object to another object. We can create our own dummy assignment operator and make it private.
2. Write your own assignment operator that does deep copy.

Solution : 01 // Do not allow assignment

#include <iostream>

using namespace std;

// A class without user defined assignment operator

class Test {

int \*ptr;

Test & operator = (const Test &t) {} // assignment private

public:

Test (int i = 0) { ptr = new int(i); }

void setValue (int i) { \*ptr = i; }

void print() { cout << \*ptr << endl; }

};

int main(void) {

Test t1(5);

Test t2;

t2 = t1;

t1.print(); t2.print(); cout << endl;

t1.setValue(10);

t1.print(); t2.print();

return 0;

}

// CE: 'Test& Test::operator=(const Test&)' is private

Solution : 02 // own assignment operator

#include <iostream>

using namespace std;

// A class without user defined assignment operator

class Test {

int \*ptr;

public:

Test (int i = 0) { ptr = new int(i); }

void setValue (int i) { \*ptr = i; }

void print() { cout << \*ptr << endl; }

Test & operator = (const Test &t);

};

Test & Test::operator = (const Test &t) {

if(this != &t) // Check for self-assignment

\*ptr = \*(t.ptr);

return \*this;

}

int main(void) {

Test t1(5);

Test t2;

t2 = t1;

t1.print(); t2.print(); cout << endl;

t1.setValue(10);

t1.print(); t2.print();

return 0;

}

Output:

5

5

10

5

# Default Assignment Operator and References

When we don’t write our own assignment operator, compiler created assignment operator does shallow copy and that cause problems. What happens when we have references in our class and there is no user defined assignment operator.

For example, predict the output of following program:

#include <iostream>

using namespace std;

class Test {

int x;

int &ref;

public:

Test (int i):x(i), ref(x) {}

void print() { cout << ref; }

void setX(int i) { x = i; }

};

int main() {

Test t1(10);

Test t2(20);

t1.print(); t2.print(); cout << endl;

t2 = t1;

t1.print(); t2.print(); cout << endl;

t1.setX(40);

t1.print(); t2.print(); cout << endl;

return 0;

}

Compilation Error:

prog.cpp: In function 'int main()':

prog.cpp:17:5: error: use of deleted function 'Test& Test::operator=(const Test&)'

t2 = t1;

^

prog.cpp:4:7: note: 'Test& Test::operator=(const Test&)' is implicitly deleted because the default definition would be ill-formed:

class Test {

^

prog.cpp:4:7: error: non-static reference member 'int& Test::ref', can't use default assignment operator

**Compiler doesn’t create default assignment operator in following cases**

1. Class has a nonstatic data member of a const type or a reference type
2. Class has a nonstatic data member of a type which has an inaccessible copy assignment operator
3. Class is derived from a base class with an inaccessible copy assignment operator

When any of the above conditions is true, user must define assignment operator.

For example, if we add an assignment operator to the above code, the code works fine without any error.

#include <iostream>

using namespace std;

class Test {

int x;

int &ref;

public:

Test (int i):x(i), ref(x) {}

void print() { cout << ref << " "; }

void setX(int i) { x = i; }

Test &operator = (const Test &t) { x = t.x; return \*this; }

};

int main() {

Test t1(10);

Test t2(20);

t1.print(); t2.print(); cout << endl;

t2 = t1;

t1.print(); t2.print(); cout << endl;

t1.setX(40);

t1.print(); t2.print(); cout << endl;

return 0;

}

Output:

10 20

10 10

40 10

# Is assignment operator inherited?

In C++, like other functions, assignment operator function is inherited in derived class.

For example, in the following program, base class assignment operator function can be accessed using the derived class object.

#include <iostream>

using namespace std;

class Base {

public:

Base &operator=(Base &a) {

cout << " Base class assignment operator called ";

return \*this;

}

};

class Derived : public Base {};

int main() {

Derived d1, d2;

d1.Base::operator=(d2); // calling base class assignment op, function using derived class

return 0;

}

Output:

Base class assignment operator called

# Automatic type conversion

In C++, you can achieve automatic type conversion for user-defined types by defining automatic type conversion functions. These functions come in two forms:

1. Conversion constructor
2. Overloaded operator

## Conversion constructor

If you define a constructor that takes as its single argument an object (or reference) of another type, that constructor allows the compiler to perform an automatic type conversion.

#include <iostream>

class ConvertThis {

public:

ConvertThis() { std::cout << "ConvertThis()\n"; }

};

class ConvertToThis {

public:

ConvertToThis(ConvertThis) { std::cout << "ConvertToThis(ConvertThis)\n"; }

};

void fun(ConvertToThis) {}

int main() {

ConvertThis cthis;

fun(cthis);

return 0;

}

Output:

ConvertThis()

ConvertToThis(ConvertThis)

### Preventing constructor conversion

Creating a single-argument constructor always defines an automatic type conversion (even if it’s got more than one argument, if the rest of the arguments are defaulted).

To prevent automatic type conversion, you modify the constructor by prefacing with the keyword explicit (which only works with constructors).

#include <iostream>

class ConvertThis {

public:

ConvertThis() { std::cout << "ConvertThis()\n"; }

};

class ConvertToThis {

public:

explicit ConvertToThis(ConvertThis) { std::cout << "ConvertToThis(ConvertThis)\n"; }

};

void fun(ConvertToThis) {}

int main() {

ConvertThis cthis;

// implicit cast not allowed for explicit constructor

//fun(cthis); // error: could not convert 'cthis' from 'ConvertThis' to 'ConvertToThis'

fun(static\_cast<ConvertToThis>(cthis)); // only explicit cast allowed

return 0;

}

Output:

ConvertThis()

ConvertToThis(ConvertThis)

## Conversion operator

You can create a member function that takes the current type and converts it to the desired type using the operator keyword followed by the type you want to convert to.

This form of operator overloading is unique because you don’t appear to specify a return type – the return type is the name of the operator you’re overloading.

With the constructor technique, the destination class is performing the conversion, but with operators, the source class performs the conversion.

#include <iostream>

class ConvertToThis;

class ConvertThis {

public:

ConvertThis() { std::cout << "ConvertThis()\n"; }

operator ConvertToThis() const;

};

class ConvertToThis {

public:

ConvertToThis() { std::cout << "ConvertToThis()\n"; }

};

ConvertThis::operator ConvertToThis() const {

std::cout << "ConvertThis::operator ConvertToThis() const\n";

return ConvertToThis{};

}

void fun(ConvertToThis) {}

int main() {

ConvertThis cthis;

fun(cthis);

return 0;

}

Output:

ConvertThis()

ConvertThis::operator ConvertToThis() const

ConvertToThis()

## Pitfalls in automatic type conversion

Pitfalls in automatic type conversion Because the compiler must choose how to quietly perform a type conversion, it can get into trouble if you don’t design your conversions correctly. A simple and obvious situation occurs with a class X that can convert itself to an object of class Y with an operator Y( ). If class Y has a constructor that takes a single argument of type X, this represents the identical type conversion. The compiler now has two ways to go from X to Y, so it will generate an ambiguity error when that conversion occurs.

class ConvertToThis;

class ConvertThis {

public:

operator ConvertToThis() const;

};

class ConvertToThis {

public:

ConvertToThis() {}

ConvertToThis(ConvertThis) {}

};

ConvertThis::operator ConvertToThis() const {

return ConvertToThis{};

}

void fun(ConvertToThis) {}

int main() {

ConvertThis cthis;

fun(cthis);

return 0;

}

Compilation Error:

<source>:22:9: error: conversion from 'ConvertThis' to 'ConvertToThis' is ambiguous

22 | fun(cthis);

| ^~~~~

# References

Chapter 12 Operator Overloading | Thinking in C++, Volume 1, 2nd Edition