Member Operators

- Overload operators to form expressions involving objects of a class
- Describe the syntax for overloading operators that are members of a class
- Describe casting and conversion operations on instances of a class

> "Programmers hate surprises: Overload operators only for good reason, and preserve natural semantics; if that's difficult, you might be misusing operator overloading" **Sutter, Alexandrescu, 2005.**

An important feature of object-oriented programs is support for expressions composed of objects. An expression consists of an operator and a set of operands. The expression evaluates to a value of specific type. In languages like C++, all operators are built-in. The core language defines the logic for the operands of fundamental type. To support expressions with operands of class type, we need to overload the built-in operators for those operands. Overloading an operator entails declaring a corresponding function in the class definition and defining its logic in the implementation file.

This chapter lists the C++ operators that we may overload and describes the syntax for overloading operators using member functions. These functions cover unary and binary operations on the current object. This chapter also describes how to define casting operations and how to use temporary object effectively.

Operations

In the C++ language, the keyword `operator` identifies an overloaded operation. We follow the keyword by the operator's symbol. The signature of a member function that overloads an operator consists of the keyword, the symbol and the type of the right operand, if any, within parentheses. The left operand of any member operator is the current object.

For example, an overloaded assignment operator for a `Student` right operand takes the form

```
Student& operator=(const Student&);
```

Candidates for Overloading

C++ lets us overload the following operators (amongst others):

- **Binary arithmetic:** `+` `-` `*` `/` `%`
- **Assignment (simple and compound):** `=` `+=` `-=` `*=` `/=` `%=`
- **Unary (pre-fix post-fix plus minus):** `++` `--` `+` `-`
- **Relational:** `==` `<` `>` `<=` `>=` `!=`
- **Logical:** `&&` `||` `!`
- **Insertion, extraction:** `<<` `>>`

C++ **DOES NOT ALLOW** overloading of the following operators (amongst others):

- **The scope resolution operator:** `::`
- **The member selection operator:** `.`
- **The member selection through pointer to member operator:** `.*`
- **The conditional operator:** `?:`

C++ **DOES NOT** let us introduce or define new operators.

Classifying Operators

We classify operators by the number of operands that they take:

- **Unary (one operand):** Post-fix increment/decrement, pre-fix increment/decrement, pre-fix plus, pre-fix minus
- **Binary (two operand):** Assignment, compound assignment, arithmetic, relational, logical
- **Ternary (three operands):** Conditional operator

Members and Helpers

We overload operators in either of two ways, as:

- **Member Operators:** Part of the class definition with direct access to the class representation
- **Helper Operators:** Supporting the class, without direct access to its representation

We prefer to declare operators that change the state of their left operand as member operators. Helper operators are described separately in the chapter entitled Helper Functions.

Overloading a Member Operator

Signature

The signature of an overloaded member operator consists of:

- The `operator` keyword
- The operation symbol
- The type of its right operand, if any
- The `const` status of the operation

The compiler binds an expression to the member function with the signature that matches the operator symbol, the operand type and the `const` status.

Promotion or Narrowing of Arguments

If the compiler cannot find an exact match to an operation's signature, the compiler will attempt a rather complicated selection process to find an optimal fit, promoting or narrowing the operand value into a related type if necessary.

Type of the Evaluated Expression

The return type of the member function declaration identifies the type of the evaluated expression.

Good Design Practice

Programmers expect an operator to perform its operation in a way similar if not identical to the way that the operator performs its operation on any fundamental type as defined by the core language. For instance, + implies addition of two values in a binary operation (not subtraction). In defining a member operator we code its logic to be consistent with operations on other types.

Binary Operators

A binary operation consists of one operator and two operands. In a binary member operator, the left operand is the current object and the member function takes one explicit parameter: the right operand.

The declaration of a binary member operator takes the form

```
return_type operator symbol (type [identifier])
```

`return_type` is the type of the evaluated expression. `operator` identifies the function as an operation. `symbol` specifies the kind of operation. `type` is the type of the right operand. `identifier` is the right operand's name.

Example

Let us overload the `+=` operator for a `float` as the right operand, in order to add a single grade to a `Student` object:

```
// Overloading Operators
// operators.cpp

#include <iostream>
using namespace std;
const int NG = 20;

class Student {
    int no;
    float grade[NG];
    int ng;
    void set(int, const float*, int);
public:
    Student();
    Student(int, const float*, int);
    void display() const;
    Student& operator+=(float g);
};

Student::Student() {
    no = 0;
    ng = 0;
}
```

```
Student::Student(int sn, const float* g, int ng_) {
    set(sn, g, ng_);
}

void Student::set(int sn, const float* g, int ng_) {
    bool valid = sn > 0 && g != nullptr && ng_ >= 0;
    if (valid)
        for (int i = 0; i < ng_ && valid; i++)
            valid = g[i] >= 0.0f && g[i] <= 100.0f;

    if (valid) {
        // accept the client's data
        no = sn;
        ng = ng_ < NG ? ng_ : NG;
        for (int i = 0; i < ng; i++)
            grade[i] = g[i];
    } else {
        no = 0;
        ng = 0;
    }
}

void Student::display() const {
    if (no > 0) {
        cout << no << ":\n";
        cout.setf(ios::fixed);
        cout.precision(2);
        for (int i = 0; i < ng; i++) {
            cout.width(6);
            cout << grade[i] << endl;
        }
        cout.unsetf(ios::fixed);
        cout.precision(6);
    } else {
        cout << "no data available" << endl;
    }
}

Student& Student::operator+=(float g) {
    if (no != 0 && ng < NG && g >= 0.f && g <= 100.f)
        grade[ng++] = g;
    return *this;
}
```

```
int main () {
    float gh[] = {89.4f, 67.8f, 45.5f};
    Student harry(1234, gh, 3);
    harry.display();
    harry += 78.23f;
    harry.display();
}
```

```
1234:
 89.40
 67.80
 45.50
1234:
 89.40
 67.80
 45.50
 78.23
```

Unary Operators

A unary operation consists of one operator and one operand. The left operand of a unary member operator is the current object. The operator does not take any explicit parameters (with one exception - see post-fix operators below).

The header for a unary member operator takes the form

```
return_type operator symbol()
```

`return_type` is the type of the evaluated expression. `operator` identifies an operation. `symbol` identifies the kind of operation.

Pre-Fix Operators

We overload the pre-fix increment/decrement operators to increment/decrement the current object and return a reference to its updated value. The header for a pre-fix operator takes the form

```
Type& operator++()  or  Type& operator--()
```

Example

Let us overload the pre-fix increment operator for our `Student` class so that a pre-fix expression increases all of the `Student`'s grades by one mark, if possible:

```
// Pre-Fix Operators
// preFixOps.cpp

#include <iostream>
using namespace std;
const int NG = 20;

class Student {
    int no;
    float grade[NG];
    int ng;
    void set(int, const float*, int);
public:
    Student();
    Student(int, const float*, int);
    void display() const;
    Student& operator++();
};

Student::Student() {
    no = 0;
    ng = 0;
}

Student::Student(int sn, const float* g, int ng_) {
    set(sn, g, ng_);
}

void Student::set(int sn, const float* g, int ng_) {
    bool valid = sn > 0 && g != nullptr && ng_ >= 0;
    if (valid)
        for (int i = 0; i < ng_ && valid; i++)
            valid = g[i] >= 0.0f && g[i] <= 100.0f;
```

```
    if (valid) {
        // accept the client's data
        no = sn;
        ng = ng_ < NG ? ng_ : NG;
        for (int i = 0; i < ng; i++)
            grade[i] = g[i];
    } else {
        no = 0;
        ng = 0;
    }
}

void Student::display() const {
    if (no > 0) {
        cout << no << ":\n";
        cout.setf(ios::fixed);
        cout.precision(2);
        for (int i = 0; i < ng; i++) {
            cout.width(6);
            cout << grade[i] << endl;
        }
        cout.unsetf(ios::fixed);
        cout.precision(6);
    } else {
        cout << "no data available" << endl;
    }
}

Student& Student::operator++() {
    for (int i = 0; i < ng; i++)
        if (grade[i] < 99.0f) grade[i] += 1.f;
    return *this;
}

int main () {
    float gh[] = {89.4f, 67.8f, 45.5f};
    Student harry(1234, gh, 3), backup;
    harry.display();
    backup = ++harry;
    harry.display();
    backup.display();
}
```

```
1234:
 89.40
 67.80
 45.50
1234:
 90.40
 68.80
 46.50
1234:
 90.40
 68.80
 46.50
```

Post-Fix Operators

We overload the post-fix operators to increment/decrement the current object after returning its value. The header for a post-fix operator takes the form

```
return_type operator++(int)  or  Type operator--(int)
```

The `int` type in the header distinguishes the post-fix operators from their pre-fix counterparts.

Example

Let us overload the incrementing post-fix operator for our `Student` class so that a post-fix expression increases all of the Stude`nt's grades by one mark, if possible:

```
// Post-Fix Operators
// postFixOps.cpp

#include <iostream>
using namespace std;
const int NG = 20;

class Student {
    int no;
    float grade[NG];
    int ng;
```

```
    void set(int, const float*, int);
public:
    Student();
    Student(int, const float*, int);
    void display() const;
    Student& operator++();
    Student operator++(int);
};

Student::Student() {
    no = 0;
    ng = 0;
}

Student::Student(int sn, const float* g, int ng_) {
    set(sn, g, ng_);
}

void Student::set(int sn, const float* g, int ng_) {
    bool valid = sn > 0 && g != nullptr && ng_ >= 0;
    if (valid)
        for (int i = 0; i < ng_ && valid; i++)
            valid = g[i] >= 0.0f && g[i] <= 100.0f;

    if (valid) {
        // accept the client's data
        no = sn;
        ng = ng_ < NG ? ng_ : NG;
        for (int i = 0; i < ng; i++)
            grade[i] = g[i];
    } else {
        no = 0;
        ng = 0;
    }
}

void Student::display() const {
    if (no > 0) {
        cout << no << ":\n";
        cout.setf(ios::fixed);
        cout.precision(2);
        for (int i = 0; i < ng; i++) {
            cout.width(6);
            cout << grade[i] << endl;
```

```
        }
        cout.unsetf(ios::fixed);
        cout.precision(6);
    } else {
        cout << "no data available" << endl;
    }
}

Student& Student::operator++() {
    for (int i = 0; i < ng; i++)
        if (grade[i] < 99.0f) grade[i] += 1.f;
    return *this;
}

Student Student::operator++(int) {
    Student s = *this;  // save the original
    ++(*this);          // call the pre-fix operator
    return s;           // return the original
}

int main () {
    float gh[] = {89.4f, 67.8f, 45.5f};
    Student harry(1234, gh, 3), backup;
    harry.display();
    backup = harry++;
    harry.display();
    backup.display();
}
```

```
1234:
 89.40
 67.80
 45.50
1234:
 90.40
 68.80
 46.50
1234:
 89.40
 67.80
 45.50
```

We avoid duplicating logic by calling the pre-fix operator from the post-fix operator.

Return Types

The return types of the pre-fix and post-fix operators differ. The post-fix operator returns a copy of the current object as it was **before** any changes took effect. The pre-fix operator returns a reference to the current object, which accesses the data **after** the changes have taken effect.

Type Conversion Operators

Type conversion operators define implicit conversions to different types, including fundamental types.

For the following code to compile, the compiler needs information on how to convert a `Student` object to a `bool` value:

```
Student harry;

if (harry)
    harry.display();
```

`bool` operator

Let us define a conversion operator that returns `true` if the `Student` object has valid data and `false` if the object is in a safe empty state.

We add the following declaration to the class definition:

```
const int NG = 20;

class Student {
    int no;
    float grade[NG];
    int ng;
    void set(int, const float*, int);
public:
    Student();
```

```
    Student(int, const float*, int);
    void display() const;
    operator bool() const;
};
```

We define the conversion operator in the implementation file

```
#include "Student.h"

// ...

Student::operator bool() const { return no != 0; }
```

Good Design Tip

Conversion operators easily lead to ambiguities. Good design uses them quite sparingly and keeps their implementations trivial.

Cast Operator

C++ defines the casting operation for a class type in terms of a single-argument constructor. This overloaded constructor defines the rule for casting a value of its parameter type to the class type, as well as constructing an object from an argument of the parameter type.

The following program demonstrates both uses of a single-argument constructor on an `int` argument:

```
// Casting
// casting.cpp

#include <iostream>
using namespace std;
const int NG = 20;

class Student {
    int no;
    float grade[NG];
    int ng;
```

```
    void set(int, const float*, int);
public:
    Student();
    Student(int);
    Student(int, const float*, int);
    void display() const;
};

Student::Student() {
    no = 0;
    ng = 0;
}

Student::Student(int sn) {
    float g[] = {0.0f};
    set(sn, g, 0);
}

Student::Student(int sn, const float* g, int ng_) {
    set(sn, g, ng_);
}

void Student::set(int sn, const float* g, int ng_) {
    bool valid = sn > 0 && g != nullptr && ng_ >= 0;
    if (valid)
        for (int i = 0; i < ng_ && valid; i++)
            valid = g[i] >= 0.0f && g[i] <= 100.0f;

    if (valid) {
        // accept the client's data
        no = sn;
        ng = ng_ < NG ? ng_ : NG;
        for (int i = 0; i < ng; i++)
            grade[i] = g[i];
    } else {
        no = 0;
        ng = 0;
    }
}

void Student::display() const {
    if (no > 0) {
        cout << no << ":\n";
        cout.setf(ios::fixed);
```

```
        cout.precision(2);
        for (int i = 0; i < ng; i++) {
            cout.width(6);
            cout << grade[i] << endl;
        }
        cout.unsetf(ios::fixed);
        cout.precision(6);
    } else {
        cout << "no data available" << endl;
    }
}

int main () {
    Student harry(975), nancy;

    harry.display();
    nancy = (Student)428;
    nancy.display();
}
```

```
975:

428:
```

The first use converts `975` to the `Student` object `harry`. The second use casts `428` to a `Student` object containing the number `428`. Both objects hold empty grade lists.

Promotion (Optional)

For the same result as the above cast, we may omit the cast operator and defer to the compiler promoting the `int` value `428` to a `Student` object before assigning the object to `nancy`:

```
int main () {
    Student harry(975), nancy;

    harry.display();
    cout  << endl;
    nancy = 428; // promotes an int to a Student
    nancy.display();
```

```
    cout  << endl;
}
```

```
975

428
```

The compiler inserts code that creates a temporary `Student` object using the single-argument constructor. The constructor receives the value `428` and initializes `no` to `428` and ng to `0`. Then, the assignment operator copies the temporary object to `nancy`. Finally, the compiler inserts code that destroys the temporary object removing it from memory.

Explicit (Optional)

Declaring several single-argument constructors raise the possibility of potential ambiguities in automatic conversions form one type to another. Limiting the number of single-argument constructors in a class definition helps avoid such potential ambiguities.

To prohibit the compiler from using a single-argument constructor for any implicit conversion, we declare that constructor `explicit`:

```
class Student {
    int no;
    char grade[M+1];
    void set(int, const float*, int);
public:
    Student();
    explicit Student(int);
    Student(int, const float*, int);
    void display() const;
};
```

With such a declaration, the second invocation in the example at the start of the section above (`nancy = 428`) would generate a compiler error.

Temporary Objects

C++ compilers create temporary objects in a variety of situations. A temporary object has no name and is destroyed as the last step in evaluating the expression that contains its creation point.

Consider the assignment expression below:

```
int main () {
    Student harry(975), nancy;

    harry.display();
    nancy = Student(428); // temporary Student object
    nancy.display();
}
```

```
975:

428:
```

Localizing Constructor Logic

We can use temporary objects to access validation logic localized within one constructor. Note the temporary object assignments to the current object (*this) in the one-argument and three-argument constructors below:

```
// Localized Validation
// localize.cpp

#include <iostream>
using namespace std;
const int NG = 20;

class Student {
    int no;
    float grade[NG];
    int ng;
public:
    Student();
    Student(int);
    Student(int, const float*, int);
    void display() const;
```

```
};

Student::Student() {
    // safe empty state
    no = 0;
    ng = 0;
}

Student::Student(int sn) {
    float g[] = {0.0f};
    *this = Student(sn, g, 0);
}

Student::Student(int sn, const float* g, int ng_) {
    bool valid = sn > 0 && g != nullptr && ng_ >= 0;
    if (valid)
        for (int i = 0; i < ng_ && valid; i++)
            valid = g[i] >= 0.0f && g[i] <= 100.0f;

    if (valid) {
        // accept the client's data
        no = sn;
        ng = ng_ < NG ? ng_ : NG;
        for (int i = 0; i < ng; i++)
            grade[i] = g[i];
    } else {
        *this = Student();
    }
}

void Student::display() const {
    if (no > 0) {
        cout << no << ":\n";
        cout.setf(ios::fixed);
        cout.precision(2);
        for (int i = 0; i < ng; i++) {
            cout.width(6);
            cout << grade[i] << endl;
        }
        cout.unsetf(ios::fixed);
        cout.precision(6);
    } else {
        cout << "no data available" << endl;
    }
```

```
}

int main () {
    float gh[] = {89.4f, 67.8f, 45.5f};
    Student harry(1234, gh, 3), josee(1235), empty;
    harry.display();
    josee.display();
    empty.display();
}
```

```
1234:
89.40
67.80
45.50
1235:
no data available
```

The three-argument constructor validates all data received from client code. If the validation fails, this constructor creates a temporary object in a safe empty state and assigns that temporary object to the current object. Note that the single-argument constructor uses the temporary object created by the three-argument constructor to initialize the current object.

Good Design Tip

Using temporary objects to avoid repeated logic is good programming practice. If we update the logic later, there is no chance that we will update the logic in one part of the source code and neglect to update identical logic in another part of the code.

Summary

- C++ allows overloading of most of the operators for operands of class type
- We cannot define new operators or redefine operations on the fundamental types
- The keyword operator followed by a symbol identifies an operation
- The left operand in an overloaded member operator is the current object
- We use member operators to overload operations that modify the left operand

- The int keyword in the signature for increment/decrement operator identifies the post-fix operation distinguishing it from the pre-fix operation
- We use temporary objects to localize logic, which improves maintainability