Overloading, Conversions, and Operators

A `nonexlicit` constructor that can be called with one argument defines an implicit conversion. That is, consturctors convert an object from the argument's type to the class type. We can also define conversion from the class type. We define a conversion from a class type by defining a conversion operator. Conversion constructors and conversion operators define **class-type conversions**, which are also referred to as **user-defined conversion**.

Conversion Operators

A conversion operator is a special kind of member function that converts a value of class type to a value of some other type. A conversion function typically has the general form

```
operator type() const;
```

Conversion operators can be defined for any type (other than `void`) that can be a function return type.

- conversion to an array or a funtion type are not allowed
- conversion operators have no explicitly stated return type and no parameters
- conversion operators must be defined as member functions
- conversion operations ordinarily should not change the object they are converting. (should be defined as `const` members)

Definiing a Class with a Conversion Operator

```
class SmallInt{
    public:
        SmallInt(int i=0):val(i){
            if(i<0||i>255)
                throw std::out_of_range("bad smallint value");
        }

        operator int() const {return val;}
    private:
        std::size_t val;
};
```

This `SmallInt` class defines conversions to and from its type. The constructor converts values of arthmetic type to a `SmallInt`.

```
// uses the consturctor to convert values of arithmetic type to a SmallInt

// equivalent calls
SmallInt f(123);
```

```
SmallInt f=123;// not assignment, calls the constructor to initialize

SmallInt g;
g=123;//  123 wil be implictly converted to a SmallInt;
// then calls SmallInt::operator=
```

The conversion operator converts `SmallInt` objects to `int`:

```
SmallInt f;// f.val will be set to be 0
int a=f;// calls the conversion operator
```

What' going on if the constructor was defined as `exlicit`?

The `nonexlicit` constructor:

```
SmallInt si=3.14; // calls the SmallInt(int) constructor
// the double argument is converted to int suing the built-in conversion

si+3.14; // the SmallInt conversion operator converts si to int
// then that int is converted to double using the built-in conversion
```

Notes:
Because conversion operators are implicitly appied, there is no way to pass arguments to these functions. Hence, conversion operators may not be defined to take parameters. Altough a conversion function does not specify a return type, each conversion function must return a value of its corresponding type:

```
class SmllInt;
operator int(SmallInt &);// error: nonmember
class SmallInt{
    public:
        int operator int()const;// error: return type
        operator int(int =0) const;// error: parameter list
        operator int *()const {return 42;}// error: 42 is not a pointer
};
```

Conversion Operators Can Yield Surprising Results

Too often users are more likely to be surprised if a conversion happens automatically than to be helped by the existence of the conversion. However, there is one important exception to this rule of thumb: It is common for classes to define conversions to `bool`.

An example:

```
int i=42;
std::cin<<i;// this code would be legal if the conversion to bool were not
```

```
explicit
```

This program attempts to use the outoperator (<<) on an input stream. There is no << defined for istream, so the code is almost surely in error. However, this code could use the bool conversion operator to convert cin to bool. The resulting bool value would then be promoted to int and used as the left-hand operand to the built-in version of the left-shift operator. The promoted bool value (either 1 or 0) would be shifted left 42 positions.

explicit Conversion Operators

To prevent such problems, the new standard introduced explicit conversion operators:

```
class SmallInt{
    public:
        // the compiler won't automatically apply this conversion
        explicit operator int()const{return val;}

};
```

As with an explicit constructor, the compiler won't use an explicit conversion operator for implicit conversion:

```
SmallInt si=3; // ok, the SmallInt constructor is not explict
si+3;// error, implicit is conversion required, but operator is explicit.
static_cast<int>(si)+3;// ok, explicitly request the conversion
```

> An explicit conversion will be used implicitly to convert an expression used as a condition.

Conversion to bool

We use the operator bool that is defined for the IO types. For example,

```
while(std::cin>>value)
```

The condition executes the input operator, which reads into value and returns cin. To evaluate the condition, cin is implicitly converted by the istream operator bool conversion function. That function returns true if the condition state of cin is good, and false otherwise.

> Conversion to bool is usually intended for use in conditions. As a result, operator bool ordinarily should be defined as explicit.

Avoiding Ambiguous Conversions

Argument Matching and Mutual Conversions

In the following example, we've defined two ways to obtain an `A` from a `B`: either by using `B`'s conversion operator or by using the `A` constructor that takes a `B`:

```
struct B;
struct A{
    A()=default;
    A(const B&);// converts a B to an A
};

struct B{
    operator A()const;// also converts a B to an A
};

A f(const A &);
B b;
A a=f(b);// error ambiguous: f(B::operator A())
            //              or f(A::A(const B &))
```

If we want to make this call, we have to explicitly call the conversion operator or the constructor:

```
A a1=f(b.operator A());// ok, use B's conversion operator
A a2=f(A(b));// ok, use A's nonexplicit constructor
```

Ambiguities and Multiple Conversions to Built-in Types

Ambiguities also occur when a class defines multiple conversions to (or from) types that are themselves related by conversions.

```
struct A{
    // a bad idea to have two conversions from arithmetic types
    A(int =0);
    A(double);

    // a bad idea to have two conversions to arithmetic types
    operator int()const;
    operator double()const;
};

void f2(long double);

A a;

f2(a); // error ambiguous: f(A::operator int())
                        // or f(A::operator double())

long lg;
A a2(lg);// error ambiguous: A::A(int) or A::A(double)
```

In the call to `f2`, neither conversion is an exact match to `long double`. However, either conversion can be used, followed by a standard conversion to get to `long double`.

We encounter the same problem when we try to initialize `a2` from a `long`. Neither constructor is an exact match for `long`. Each would require that the argument be converted before using the constructor:

- standard `long` to `double` conversion followed by `A(double)`
- standard `long` to `int` conversion followed by `A(int)`

When a user-defined conversion is used, the rank of the standard conversion, if any, is used to select the best match:

```
short s=42;
// promoting short to int is better than converting short to double
A a3(s);// uses A::A(int)
```

In this case, promoting a `short` to an `int` is preferred to converting the `short` to a `double`. Hence `a3` is constructed using the `A::A(int)` constructor, which is run on the (promoted) value of `s`.

Overloaded Functions and Converting Constructor

Choosing among multiple conversions is further complicated when we call an overloaded function. If two or more conversions provide a viable match, then the conversions are considered equally good.

One example:

```
struct C{
    C(int);
};

struct D{
    D(int);
};

void manip(const C&);
void manip(const D&);

manip(10);// error ambiguous: manip(C(10)) or manip(D(10))
```

Here both `C` and `D` have constructors that take an `int`. The caller can disambiguate by explicitly constructing the correct type:

```
manip(C(10));
manip(D(10));
```

Overloaded Functions and User-Defined Conversion

In a call to an overloaded function, if two (or more) user-defined conversions provide a viable match, the conversions are considered equally good. The rank of any standard conversions that might or might not be required is not considered.

Whether a built-in conversion is considered only if the overload set can be matched using the same conversion function.

```
struct E{
    E(double);
};

void foo(const E&);
foo(10);// built-in conversion is implicitly executed like foo(double(10));
```

As an example, even if one of the classes defined a constructor that required a standard conversion for the argument.

```
struct E{
    E(double);
};
void manip2(const C&);
void manip2(const E&);

// error ambiguous: two different user-defined conversions could be used
manip2(10);// manip2(C(10)) or manip2(E(double(10)))
```

For the call `manip2(10)`, both `manip2` functions are viable:

- `manip2(const C &)` is viable because `C` has a converting constructor that takes an `int`. That constructor is an exact match for the argument.
- `manip2(const E &)` is viable because `E` has a converting constructor that takes a `double` and we can use a standard conversion to convert the `int` argument in order to use that converting constructor.

Function Matching and Overloaded Operators**

If `a` has a class type, the expression `a sym b` might be

```
a.operatorsym(b);// a has operatorsym as a member function
operatorsym(a,b);// operatorsym is an ordinary function
```

> When an operator function is used in an expression, the set of candidate functions is broader than when we call a function using the call operator.

When we use an overloaded operator with an operand of class type, the candidate functions include ***ordinary nonmember versions*** of that operator, as well as ***the built-in versions*** of the operator. Moveover, if the left-hand operand has class type, ***the overloaded versions*** of the operator, if any, defined by that class are also included.

When we call a named function, member and nonmember functions with the same name do not overload one another. There is on overloading because the syntax we use to call a named function distinguishes between member and nonmember functions.

When a call is through an object of a class type (or through a reference of pointer to such an object), then only the member functions of that class are considered.

```
Foo f;
f.func();// call the member version
```

When we use an overloaded **operator** in an expression, there is nothing to indicate whether we're using a member or nonmember function. Therefore, both member and nonmember versions must be considered.

```
class Foo{
    // member version, the first parameter is implicitly passed by the object to
which this pointer points.
    Foo operator+(const Foo &f);// overloaded plus operator
};

// nonmember version
Foo operator+(const Foo &f,const Foo &g);
Foo f;
f+g;// nonmember version and member version are considered.
//      but, it seems the member version is preferred to the nonmember version

f.operator+(g);// only member version is considered

operator+(f,g);// only nonmember version is considered
```

Another example:

```
class SmallInt{
    friend SmallInt operator+(const SamllInt &,const SmallInt &);
    public:
    SamllInt(int =0);
    operator int() const{return val;}

private:
    std::size_t val;
};
When we call,
```c++
SamllInt s1,s2;
SamllInt s3=s1+s2;// uses overloaded operator+
int i=s3+0;// error: ambiguous
```
```

The second addition is ambiguous, because we can convert `0` to a `SmallInt` and use the `SmallInt` version of `+`, or convert `s3` to `int` and use the built-in addition operator on `int`s.