67. Conversion

Objectives

- Use constructors for conversion
- Overload conversion operators
- Define constructors explicitly

Conversion using a constructor

C++ will generate an automatic conversion using a constructor call if necessary in order to find a best match for a method.

```
class Int
{
public:
    Int(int a)
        : x_(a)
    {}

    void m(Int c) { std::cout << x_ << '\n'; }

private:
    int x_;
};
```

```
int main()
{
    Int c(0), d(1);
    c.m(d);
    c.m(2);

    return 0;
}
```

However, there is an exception: The constructor is **not** invoked for the left-hand side of `.` And `->`: `3.m(1)` will not become `C(3).m(1)`. You still might want to write your own `m(int)` for efficiency.

But there are problems with conversion using constructors:

- You cannot convert **to** built-in types (int, double, etc.) because they are not objects
- You cannot convert from an object of class `C` to an object of class `D` (without changing the declaration of `D`)

So ...

Conversion operators

Suppose `D` is a type (class or built-in type like int, double, ...) and `C` is a class. You can define a **conversion operator** from `C` to `D`. This should look like

```
class C
{
public:
  ...
  operator D();
  ...
};
```

Type conversion from `C` to `D`

WARNING: Do **not** specify a return type. C++ knows that the return type will be `D`. Also, conversion operators are usually `const`.

Let's make an addition to our `Int` class

```
class Int {

public:
    Int(int a)
        : x_(a)
    {}

    operator int() const { return x_; }

private:
    int x_;
};
```

```
int main()
{
    Int c(0);
    std::cout << (int) c << " " << int(c);

    return 0;
}
```

`(int) c` and `int(c)` are the same as `c.operator int()`.

Here's another example:

```
class D
{
public:
    D(int a, int b)
        : x_(a), y_(b)
    {}

    void print()
    {
        std::cout << "(" << x_ << "," << y_ << ")\n";
    }

private:
    int x_, y_;
};
```

```
class Int
{
public:
    Int(int a)
        : x_(a)
    {}

    operator D() { return D(x_,x_); }

private:
    int x_;
};
```

```
int main()
{
    Int c(2);

    ((D)c).print();
    D(c).print();

    return 0;
}
```

Automatic Conversion

C++ will automatically generate a call to conversion operators if necessary to find a best match for a method.

WARNING: C++ will only generate one automatic conversion for each value.

WARNING: Default conversions are used first (example `double()`). C++ will use user-defined conversion only when necessary. No errors during compilation.

```
class D
{
public:
    D(int a, int b)
        : x_(a), y_(b)
    {}

private:
    int x_, y_;
};
```

```
class Int
{
public:
    Int(int a)
        : x_(a)
    {}

    operator D()
    {
        std::cout << "Int::D()\n";
        return D(x_,x_);
    }

    void m(D d)
    {
        std::cout << "m(D)\n";
    }

private:
    int x_;
};
```

`c1` is converted to a `D` object automatically

```
int main()
{
    Int c0(2), c1(3);
    c0.m(c1);

    return 0;
}
```

```
class E {};

class D
{
public:
    operator E()
    {
        ...
    }
};

class C
{
public:
    operator D()
    {
        ...
    }
};

int main()
{
    E e1 = D(C());
    E e2 = D();
    E e3 = C();

    return 0;
}
```

ERROR: Two automatic conversions needed, `C` to `D` to `E`. i.e. C++ will **not** convert

```
E e3 = C()
```

to

```
E e3 = E(D(C()))
```

Here's another example:

```
class D {};

class C
{
public:
    operator D() { std::cout << "C::D()\n"; }
};

void f(D d) { std::cout << "f(D)\n"; }
void f(double d) { std::cout << "f(double)\n"; }

int main()
{
    f(1);
    return 0;
}
```

Notice that the default conversion to a `double()` was used instead of the `C::D()` conversion.

Advice

Do not have too many conversion operators. Doing so might lead to ambiguities. For instance, suppose you have conversion operators from `C` to `D` and vice versa; you also have `operator+` in `C` and `D`. If you run the following code

```
C c;
D d;
c + d;
```

you'll see that your compiler does not know if it should do

`D(c) + d` using `D::operator+`

or

`c + C(d)` using `C::operator+`

Here's another tip: If you have two classes `C` and `D` where `C` is "included" in D, then it's better to have automatic conversion from `C` to `D`. Consider the following example:

Let's say you have an `Int` class and a `Fraction` class. You should have a `operator Fraction()` declared in the `Int` class. Note that you can achieve the same thing by having a constructor of the form:

```
Fraction(const Int &) const;
```

Explicit

You can prevent automatic type conversion via constructors. Here's an example:

```
explicit Fraction(int);
```

In this case, C++ will **not** perform automatic type conversion from an `int` to a `Fraction`.

Here's an example of what you've already seen before:

```
class C
{
public:
    C(int i)
    {}
};

void f(C c)
{
}

int main()
{
    f(1);
    return 0;
}
```

You should already know that this will not produce any errors; `f(1)` automatically becomes `f(C(1))`. However, if you run this

```
class C
{
public:
    explicit C(int i)
    {}
};

void f(C c)
{
}

int main()
{
    f(1);

    return 0;
}
```

You'll get an error! The compiler does not know what to do because we did not define a way for an `int` to be converted to a `C` object.

Note that

- Only constructors can be made explicit.
- You cannot make type conversion operators which are not a constructor explicit.

Here's an example of the second restriction:

```
class C
{
    explicit C(int); // OK
    explicit operator int() const; // BAD!
};
```

In the future, C++ will allow all type conversion to be made `explicit`. (NOTE: C++11 (i.e., C++ 2011) supports `explicit` for type conversion.)

Now, suppose the `Fraction` class has the constructor `Fraction(int)` and `operator int()`. If we run

```
Fraction r(1, 2);
std::cout << r + 1 << '\n';
```

Which is used? `int(r) + 1 or r + Fraction(1)`? **If** `operator int()` is explicit. Then the above would execute

```
r + Fraction(1)
```