Explicit Conversion Operators

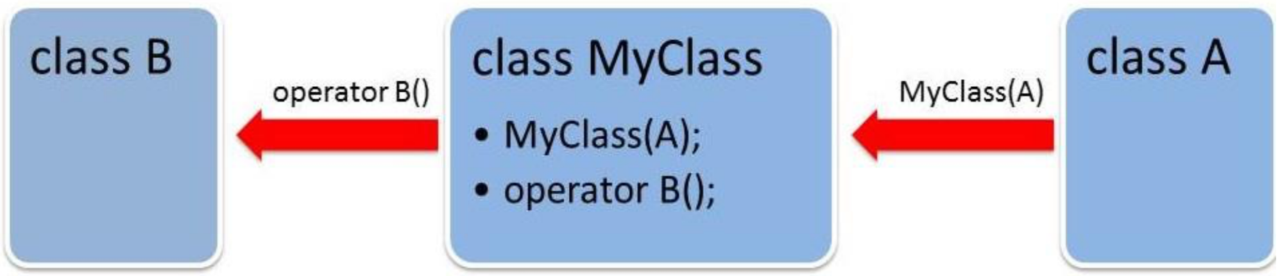
This lesson explains how conversion operators can be overloaded explicitly in C++.

Asymmetry in C++98

In C++98, the `explicit` keyword was only supported for conversion constructors. Conversion operators converted user-defined objects *implicitly*.

All this changed in C++11. Now, we can overload conversion operators to explicitly prevent and permit conversions.

Let's suppose that a class called `MyClass` can perform conversions from `class A` to `MyClass` and from `MyClass` to `class B`.



Here is what `myClass` would look like:

```
class MyClass{
  public:
    explicit MyClass(A){}        // C++98
    explicit operator B(){}      // C++11
};
```

- `MyClass(A)`: Converting constructor

- `operatorB()`: Converting operator

As we can see, the `explicit` keyword can now be used when overloading the conversion operator, `B()`.

One thing to keep in mind is that implicit conversions to `bool` are still possible, so be careful.

```
class MyBool{
public:
  explicit operator bool(){return true;}
};


...
MyBool myB;
if (myB){};
int a = (myB)? 3: 4;
int b = myB + a; // ERROR
```

We have defined that a `MyBool` object can be converted to `bool` but not to anything else.

Because of this, `int b = myB + a;` causes an error, since it is trying to implicitly convert `myB` to `int`.

Example

```
#include <iostream>

class A{};

class B{};

class MyClass{
  public:
    MyClass(){}
    explicit MyClass(A){}                // since C++98
    explicit operator B(){return B();}   // new with C++11
};

void needMyClass(MyClass){};
void needB(B){};

struct MyBool{
  explicit operator bool(){return true;}
};

int main(){
  // A -> MyClass
  A a;

  // explicit invocation
  MyClass myClass1(a);
  // implicit conversion from A to MyClass
  MyClass myClass2=a;
  needMyClass(a);

  // MyClass -> B
  MyClass myCl;

  // explicit invocation
  B b1(myCl);
  // implicit conversion from MyClass to B
  B b2= myCl;
  needB(myCl);

  // MyBool -> bool conversion
  MyBool myBool;
  if (myBool){};
  int myNumber = (myBool)? 1998: 2011;
  // implicit conversion
  int myNewNumber = myBool + myNumber;
  auto myTen = (20*myBool -10*myBool)/myBool;

  std::cout << myTen << std::endl;

}
```

- We have defined an explicit conversion constructor from `A` to `MyClass` in line 10.
- The constructor call works fine in line 27, but the implicit conversions in lines 29 and 30 are rejected by the compiler.
- `needMyClass(a)` will not be able to implicitly convert `a` to `MyClass`. This functionality has been available since C++98.
- We have defined an explicit conversion operator from `MyClass` to `B` in line 11.
- Lines 38 and 39 use an implicit conversion. Due to the explicit conversion operator `B` in line 11, this is not valid.
- Because of this explicit definition, implicit conversions through the operator are rejected by the compiler, as seen in lines 46 and 47.
- The explicit conversion feature was introduced in C++11.

- The explicit conversion feature was introduced in C++11.

To understand explicit conversions better, we can try out the exercise in the next lesson.