

## Conversion Constructors

---

### Introduction

When designing classes, you might find that certain data types can logically be converted into objects of the type you're creating. For example, when designing the C++ `string` class, you might note that `char * C` strings could have a defined conversion to `string` objects. In these situations, it may be useful to define *implicit conversions* between the two types. To define implicit conversions, C++ uses *conversion constructors*, constructors that accept a single parameter and initialize an object to be a copy of that parameter.

While useful, conversion constructors have several major idiosyncrasies, especially when C++ interprets normal constructors as conversion constructors. This handout explores implicit type conversions, conversion constructors, and how to prevent coding errors stemming from inadvertent conversion constructors.

### Implicit Conversions

In C++, an *implicit conversion* is a conversion from one type to another that doesn't require an explicit typecast. Perhaps the simplest example is the following conversion from an `int` to a `double`:

```
double myDouble = 137 + 2.71828;
```

Here, even though `137` is an `int` while `2.71828` is a `double`, C++ will implicitly convert it to a `double` so the operation can proceed smoothly.

When C++ performs implicit conversions, it does not “magically” figure out how to transform one data type into another. Rather, it creates a temporary object of the correct type that's initialized to the value of the implicitly converted object. Thus the above code is equivalent to

```
double temp = (double)myInt;  
double myDouble = temp + 2.71828;
```

It's important to remember that when using implicit conversions you are creating temporary objects. With primitive types this is hardly noticeable, but makes a difference when working with classes. For example, consider the following code:

```
string myString = "This ";  
string myOtherString = myString + "is a string";
```

Note that in the second line, we're adding a C++ `string` to a C `char * string`. Thus C++ will implicitly convert “is a string” into a C++ `string` by storing it in a temporary object. The above code, therefore, is equivalent to

```
string myString = "This ";
string tempStr = "is a string";
string myOtherString = myString + tempStr;
```

Notice that in both of the above examples, at some point C++ needed a way to initialize a temporary object to be equal to an existing object of a different type. In the first example, we made a temporary `double` that was equal to an `int`, and in the second, a temporary `string` equal to a `char *`. When C++ performs these conversions, it uses a special function called a *conversion constructor* to initialize the new object. Conversion constructors are simply class constructors that accept a single parameter and initialize the new object to a semantically-equivalent copy of the parameter. In the case of the `double` example, the newly created `double` had the same value as the `int` parameter. With the C++ `string`, the temporary `string` was equivalent to the C string it was initialized to.

C++ will invoke conversion constructors in all of the same places where it invokes copy constructors.\* Thus, if you pass a `char *` to a function accepting a C++ `string`, the `string` will be initialized to the `char *` in its conversion constructor. Similarly, if you have a function like this one:

```
string MyFunction()
{
    return "This is a string!";
}
```

The temporary object created for the return value will be initialized to the C string “This is a string!” using the conversion constructor.

## Writing Conversion Constructors

To see how to write conversion constructors, we'll use the example of a `CString` class that's essentially our own version of the C++ `string` class. Internally, `CString` stores the string as a C string called `theString`. Since we'd like to define an implicit conversion from `char *` to `CString`, we'll declare a conversion constructor, as shown below:

```
class CString
{
public:
    CString(const char *other);
    /* Other member functions. */
private:
    char *theString;
};
```

Then we'd implement the conversion constructor as

```
CString::CString(const char *other)
{
    /* Allocate space and copy over the string. */
    theString = new char[strlen(other) + 1];
    strcpy(theString, other);
}
```

---

\* In fact, technically speaking, a copy constructor is a special case of the conversion constructor.

Now, whenever we have a `char * C` string, we can implicitly convert it to a `CString`.

In the above case, we defined an implicit conversion from `char * C` strings to our special class `CString`. However, it's possible to define a second conversion from a C++ `string` to our new `CString` class. In fact, C++ allows you to provide conversion constructors for any number of different types that may or may not be primitive types.

Here's a modified `CString` interface that provides a copy constructor and two conversion constructors from `string` and `char *`:

```
class CString
{
    public:
        CString(const string &other);
        CString(const char *other);

        /* The rule-of-three says we need these functions. */
        CString(const CString &other);
        CString &operator =(const CString &other);
        ~CString();
        /* Other member functions. */
    private:
        char *theString;
};
```

### A Word on Readability

When designing classes with conversion constructors, it's easy to get carried away by adding too many implicit conversions. For example, suppose that for the `CString` class we want to define a conversion constructor that converts `ints` to their string representations. This is completely legal, but can result in some confusing or unreadable code. For example, if there's an implicit conversion from `ints` to `CStrings`, then we can write code like this:

```
CString myStr = myInt + 137;
```

The resulting `CString` would then hold a string version of the value of `myInt + 137`, not the string composed of the concatenation of the value of `myInt` and the string "137." This can be a bit confusing and can lead to counterintuitive code. Worse, since C++ does not normally define implicit conversions between numeric and string types, people unfamiliar with the `CString` implementation might get confused by lines assigning `ints` to `CStrings`.

In general, when working with conversion constructors, make sure that the conversion is intuitive and consistent with major C++ conventions. If not, consider using non-constructor member functions. For example, if we would like `CString` to be able to convert `int` values into their string representations, we might want to make a member function `intToString` that performs the conversion. This way, someone reading the code could explicitly see that we're converting an `int` to a `CString`.

## Conversion Assignment Operators

One question that might have crossed your mind is whether there's a need for a “conversion assignment operator” to pair with the conversion constructor. After all, with our `CString` class, since we've only provided a single assignment operator, it would seem that we shouldn't be able to assign a regular `char *` to a `CString`.

It turns out, however, that the following code is legal:

```
CString myStr;
myStr = "This is a C string!"; // Invokes assignment operator
```

The reason that this code is legal has to do with how C++ interprets the statement `myStr = "This is a C string!"`. Note that in our `CString` class we provided an assignment operator as the function `CString &operator =(const CString &other)`. This means that when we write the line `myStr = "This is a C string!"`, we're actually calling the `operator =` function. Thus the above code snippet is equivalent to

```
CString myStr;
myStr.operator =("This is a C string!");
```

We'll go into code like this in more detail next week when discussing operator overloading, but for now just be aware that the above two lines are completely legal C++ code.

If you'll recall, since we're passing the value “This is a C string!” to a function that accepts an object of type `CString`, C++ will automatically invoke the `CString` conversion constructor to fill in the `CString` parameter. The above code, therefore, is equivalent to

```
CString myStr;
CString tempStr("This is a C string!");
myStr = tempStr;
```

And the assignment will work out correctly. In short, since we defined conversion constructors to convert `char *`s into `CStrings`, we're free to assign `char *`s to `CStrings` without defining a new assignment operator.

## Problems with Conversion Constructors

While conversion constructors are quite useful in a wide number of circumstances, the fact that C++ automatically treats all single-parameter constructors as conversion constructors can lead to convoluted or nonsensical code.

One of my favorite examples of “conversion-constructors-gone-wrong” comes from an older version of the CS106 ADT class libraries. Originally, the CS106 `Vector` was defined as

```
template<typename ElemType>
class Vector
{
public:
    Vector(int sizeHint = 10); // Hint about the size of the Vector
    /* ... */
};
```

Nothing seems all that out-of-the-ordinary here – we have a `Vector` template class that lets you give the class a hint about the number of elements you will be storing in it. However, because the constructor accepts a single parameter, C++ will interpret it as a conversion constructor and thus will let us implicitly convert from `ints` to `Vectors`. This can lead to some very strange behavior. For example, given the above class definition, consider the following code:

```
Vector<int> myVector = 15;
```

This code, while nonsensical, is legal and equivalent to `Vector<int> myVector(15)`. Fortunately, this probably won't cause any problems at runtime – it just doesn't make sense in code.

However, suppose we have the following code:

```
void DoSomething(Vector<int> &myVector)
{
    myVector = 0;
}
```

This code is totally legal even though it makes no logical sense. Since the code expands to a call to `myVector.operator = (0)`, C++ will create a new `Vector<int>` initialized with the parameter 0 and then assign it to `myVector`. In other words, the above code is equivalent to

```
void DoSomething(Vector<int> &myVector)
{
    Vector<int> tempVector(0);
    myVector = tempVector;
}
```

`tempVector` is empty when it's created, so when we assign `tempVector` to `myVector`, we'll set `myVector` to the empty vector. Thus the nonsensical line `myVector = 0` is effectively an obfuscated call to `myVector.clear()`.

This is a quintessential example of why conversion constructors can be dangerous. When writing single-argument constructors, you run the risk of letting C++ interpret your constructor as a conversion constructor.

## **explicit**

To prevent problems like the one described above, C++ provides the `explicit` keyword to indicate that a constructor cannot be interpreted as a conversion constructor.

For example, let's look at the current version of the CS106 `Vector`, which has its constructor marked `explicit`:

```
template<typename ElemType>
class Vector
{
public:
    explicit Vector(int sizeHint = 10); // Hint the size of the Vector
    /* ... */
};
```

Now, if we write code like

```
Vector<int> myVector = 10;
```

We'll get a compile-time error since there's no implicit conversion from `int` to `Vector<int>`. However, we can still write

```
Vector<int> myVector(10);
```

Which is what we were trying to accomplish in the first place. Similarly, we eliminate the `myVector = 0` error, and a whole host of other nasty problems.

When designing classes, if you have a single-argument constructor that is not intended as a conversion function, you *must* mark it `explicit` to avoid running into the “implicit conversion” trap. While indeed this is more work for you as an implementer, it will make your code safer and more stable.

### **More to Explore**

If you're interested in writing your own container classes, you might want to consider reading into *generalized copy constructors*. A generalized copy constructor is a templated copy constructor that accepts objects of the same template class whose template types don't match the current object's. For example, given a `Vector` template, you might define a generalized copy constructor so that you could initialize a `Vector<double>` to the contents of a `Vector<int>` or vice-versa. Of course, the syntax for generalized copy constructors can be a bit tricky (it requires the `friend` keyword, which we'll cover next week), but such is the nature of heavily-templated code!

## Practice Problems

These practice problems concern a `RationalNumber` class that encapsulates a rational number (that is, a number expressible as the quotient of two integers). `RationalNumber` is declared as follows:

```
class RationalNumber
{
    public:
        RationalNumber(int num = 0, int denom = 1) :
            numerator(num), denominator(denom) {}

        double getValue() const {
            return double(numerator) / denominator;
        }

        void setNumerator(int value) {
            numerator = value;
        }
        void setDenominator(int value) {
            denominator = value;
        }
    private:
        int numerator, denominator;
};
```

The constructor to `RationalNumber` accepts two parameters that have default values. This means that if you omit one or more of the parameters to `RationalNumber`, they'll be filled in using the defaults. Thus all three of the following lines of code are legal:

```
RationalNumber zero; // Value is 0 / 1 = 0
RationalNumber five(5); // Value is 5 / 1 = 5
RationalNumber piApprox(355, 113); // Value is 355/113 = 3.1415929203...
```

1. Explain why the `RationalNumber` constructor is a conversion constructor.
2. Write a `RealNumber` class that encapsulates a real number (any number on the number line). It should have a conversion constructor that accepts a `double` and a default constructor that sets the value to zero. (*Note: You only need to write one constructor. Use `RationalNumber` as an example*)
3. Write a conversion constructor for `RealNumber` that converts a `RationalNumber` to a `RealNumber`.

General questions:

5. If a constructor has two or more arguments and no default values, can it be a conversion constructor?
6. C++ will apply at most one implicit type conversion at a time. That is, if you define three types `A`, `B`, and `C` such that `A` is implicitly convertible to `B` and `B` is implicitly convertible to `C`, C++ will not automatically convert objects of type `A` to objects of type `C`. Give an reason for why this might be. (*Hint: Add another implicit conversion between these types*)
7. What happens when you mark a copy constructor `explicit`? (*Hint: Will the syntax `MyClass one = two` be legal? How about `MyClass one(two)`?*)
8. Write a conversion constructor converting a C++ `string` to our special `CString` class.