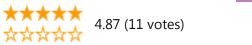
## **Adapt Cast**

Francis Xavier Pulikotil, 30 Oct 2015 MIT



Rate: vote 1vote 2vote 3vote 4vote 5

Enable passing an argument by reference, to a function which expects an argument of a different type.

Download adaptcast header - 1 KB

#### Introduction

Consider the following simple piece of code:

Hide Copy Code

```
void multiplyByTwo(double &value)
{
    value *= 2;
}

void test()
{
    double value = 10;
    multiplyByTwo(value);
    // value is now 20
}
```

The code above is so simple, it doesn't warrant any explanation. Now what if our test function didn't have adouble value to pass to the multiplyByTwo function. If test had an int, we might have implemented it like so:

Hide Copy Code

```
void test()
{
   int value = 10;

   double temp = static_cast<double>(value);
   multiplyByTwo(temp);
   value = static_cast<int>(temp);

   // value is now 20
}
```

If we had another function which took multiple parameters and we needed to pass *N* arguments whose types differed from those which the function accepted, then we would have *N* temporary additional variables. This causes a lot of noise in the code, and distracts the reader from the real business logic of the code.

## adapt\_cast

"An adapter helps two incompatible interfaces to work together" - Wikipedia

adapt\_cast enables passing an argument by reference, to a function which expects an argument of a different type. You can grab the *adaptcast.h* implementation from the archive <u>attached to this tip</u> and namespace it according to your project needs.

Let's see adapt\_cast applied to our example above:

Hide Copy Code

```
void test()
{
   int value = 10;
   multiplyByTwo(adapt_cast<double&>(value));

// value is now 20
}
```

In this fashion, if there were multiple arguments incompatible with the function signature, then the use ofadapt\_cast would greatly reduce the noise in the code, making the business logic clearer.

### **Output-only Arguments**

If an argument was output-only, i.e., its initial value is not used by the function, then adapt\_cast\_out should be used instead. The following piece of code illustrates this:

Hide Copy Code

```
void getMagicNumber(float &value)
{
    value = 7.2f;
}

void test()
{
    string str;
    getMagicNumber(adapt_cast_out<float&>(str));

// str is now "7.2"
}
```

adapt\_cast\_out also has a nice side effect that the code is now documented as well, indicating that the argument is for output purposes only.

# **Input-only Arguments**

adapt\_cast was purposefully not designed to handle input-only arguments (i.e., arguments whose inital values only are used by the function, but nothing is returned via the references). The reasoning is that, if an argument is input-only, then the function interface will probably take it by *value*, or by *const reference*. In these cases, the user can manually do the static\_cast (which adapt\_cast does by default), or use a conversion function directly.

### **Explicitly Specifying Converters**

adapt\_cast will, by default, internally use static\_cast to convert between *source* and *target* arguments. The source argument being the object to be passed into the function whose type is incompatible with the function signature. And the *target* argument being a temporary which is compatible with the function signature, which acts as a substitute for the *source* argument.

Illustration of source and target arguments:

Hide Copy Code

```
void test()
{
    int value = 10; // source argument

    multiplyByTwo(adapt_cast<double&>(value));
/*
    The above line hypothetically translates to this:

    double target = static_cast<double>(value);
    multiplyByTwo(target);
    value = static_cast<int>(target);

*/
    // value is now 20
}
```

adapt\_cast also allows custom conversion functions to be provided. The syntax for using adapt\_cast with
custom conversion functions is:

Hide Copy Code

```
adapt_cast<TargetType &>(sourceObject, ConvertSourceToTargetFunction, ConvertTargetToSourceFunction)
```

Where the signature of a conversion function is:

Hide Copy Code

```
OutputType func(const InputType &)
```

The following example illustrates usage of custom conversion functions with adapt\_cast:

```
Hide Copy Code
```

```
double toDouble(const int &value)
{
```

```
return static_cast<double>(value);
}
int toInt(const double &value)
{
    return static_cast<int>(value);
}

void test()
{
    int value = 10;

    multiplyByTwo(adapt_cast<double&>(value, toDouble, toInt));

// value is now 20
}
```

# Closing

There is much potential for improvement; if you make changes to the code, improve it, or have some better ideas, I would love to know. I can be reached by email at francisxavierjp [at] gmail [dot] com. Comments and suggestions are always welcome!

#### License

This article, along with any associated source code and files, is licensed under The MIT License

#### About the Author



#### **Francis Xavier Pulikotil**



Besides loving spending time with family, Francis Xavier likes to watch sci-fi/fantasy/action/drama movies, listen to music, and play video-games. After being exposed to a few video-games, he developed an interest in computer programming. He currently holds a Bachelor's degree in Computer Applications.