

# 1 Technique

## Protecting Your Data with Encapsulation

### Save Time By

- ✓ Understanding encapsulation
- ✓ Creating and implementing an encapsulated class
- ✓ Making updates to an encapsulated class

The dictionary defines *encapsulation* as “to encase in or as if in a capsule” and that is exactly the approach that C++ uses. An object is a “capsule” and the information and processing algorithms that it implements are hidden from the user. All that the users see is the functional-level interface that allows them to use the class to do the job they need done. By placing the data within the interface, rather than allowing the user direct access to it, the data is protected from invalid values, wrongful changes, or improper coercion to new data types.



Most time wasted in application development is spent changing code that has been updated by another source. It doesn't really add anything to your program, but it does take time to change things when someone has modified the algorithm being used. If you hide the algorithm from the developer — and provide a consistent interface — you will find that it takes considerably less time to change the application when the base code changes. Since the user only cares about the data and how it is computed, keeping your algorithm private and your interface constant protects the data integrity for the user.

### Creating and Implementing an Encapsulated Class

Listing 1-1 presents the `StringCoding` class, an encapsulated method of encryption. The benefit of encapsulation is, in effect, that it cuts to the chase: The programmer utilizing our `StringCoding` class knows nothing about the algorithm used to encrypt strings — and doesn't really need to know what data was used to encrypt the string in the first place. Okay, but why do it? Well, you have three good reasons to “hide” the implementation of an algorithm from its user:

- ✓ Hiding the implementation stops people from fiddling with the input data to make the algorithm work differently. Such changes may be meant to make the algorithm work correctly, but can easily mess it up; either way, the meddling masks possible bugs from the developers.

- ✓ Hiding the algorithm makes it easy to replace the implementation with a more workable alternative if one is found.
- ✓ Hiding the algorithm makes it more difficult for people to “crack” your code and decrypt your data.

The following list of steps shows you how to create and implement this encapsulated method:

- 1. In the code editor of your choice, create a new file to hold the code for the definition of your source file.**

In this example, the file is named `ch01.cpp`, although you can use whatever you choose.

- 2. Type the code from Listing 1-1 into your file, substituting your own names for the italicized constants, variables, and filenames.**

Or better yet, copy the code from the source file included on this book’s companion Web site.

---

#### LISTING 1-1: THE STRINGCODING CLASS

---

```
#include <stdio.h>
#include <string>

class StringCoding
{
private:
    // The key to use in encrypting the string
    std::string sKey;
public:
    // The constructor, uses a preset key
    StringCoding( void )
    {
        sKey = "ATest";
    }
    // Main constructor, allows the user to specify a key
    StringCoding( const char *strKey )
    {
        if ( strKey )
            sKey = strKey;
        else
            sKey = "ATest";
    }
    // Copy constructor
    StringCoding( const StringCoding& aCopy )
    {
        sKey = aCopy.sKey;
    }

public:
    // Methods
    std::string Encode( const char *strIn );
    std::string Decode( const char *strIn );
private:
    std::string Xor( const char *strIn );
};
```

```
std::string StringCoding::Xor( const char *strIn )
{
    std::string sOut = "";

    int nIndex = 0;
    for ( int i=0; i<(int)strlen(strIn); ++i )
    {
        char c = (strIn[i] ^ sKey[nIndex]);
        sOut += c;
        nIndex ++;
        if ( nIndex == sKey.length() )
            nIndex = 0;
    }

    return sOut;
}

// For XOR encoding, the encode and decode methods are the same.
std::string StringCoding::Encode( const char *strIn )
{
    return Xor( strIn );
}

std::string StringCoding::Decode( const char *strIn )
{
    return Xor( strIn );
}

int main(int argc, char **argv)
{
    if ( argc < 2 )
    {
        printf("Usage: ch1_1 inputstring1 [inputstring2...]\n");
        exit(1);
    }

    StringCoding key("XXX");

    for ( int i=1; i<argc; ++i )
    {
        std::string sEncode = key.Encode( argv[i] );
        printf("Input String : [%s]\n", argv[i] );
        printf("Encoded String: [%s]\n", sEncode.c_str() );
        std::string sDecode = key.Decode( sEncode.c_str() );
        printf("Decoded String: [%s]\n", sDecode.c_str() );
    }

    printf("%d strings encoded\n", argc-1);
    return 0;
}
```

---

3. Save the source code as a file in the code-editor application and then close the code editor.
4. Compile your completed source code, using your favorite compiler on your favorite operating system.
5. Run your new application on your favorite operating system.

If you have done everything properly, you should see the output shown here in the console window of your operating system:

```
$ ./ch1_1.exe "hello"
Input String : [hello]
Encoded String: [0=447]
Decoded String: [hello]
1 strings encoded
```

Note that our input and decoded strings are the same — and that the encoded string is completely indecipherable (as a good encrypted string should be). And any programmer using the object will never see the algorithm in question!

## Making Updates to an Encapsulated Class

One of the benefits of encapsulation is that it makes updating your hidden data simple and convenient. With encapsulation, you can easily replace the underlying encryption algorithm in Listing 1-1 with an alternative if one is found to work better. In our original algorithm, we did an “exclusive logical or” to convert a character to another character. In the following example, suppose that we want to use a different method for encrypting strings. For simplicity, suppose that this new algorithm encrypts strings simply by changing each character in the input string to the next letter position in the alphabet: An *a* becomes a *b*, a *c* becomes a *d*, and so on. Obviously, our decryption algorithm would have to do the exact opposite, subtracting one letter position from the input string to return a valid output string. We could then modify the `Encode` method in Listing 1-1 to reflect this change. The following steps show how:

### 1. Reopen the source file in your code editor.

In this example, we called the source file `ch01.cpp`.

### 2. Modify the code as shown in Listing 1-2.

#### LISTING 1-2: UPDATING THE STRINGCODING CLASS

```
std::string StringCoding::Encode( const char *strIn )
{
    std::string sOut = "";
    for ( int i=0; i<(int)strlen(strIn); ++i )
    {
        char c = strIn[i];
        c ++;

        sOut += c;
    }
    return sOut;
}

std::string StringCoding::Decode( const char *strIn )
{
    std::string sOut = "";
```

```
for ( int i=0; i<(int)strlen(strIn); ++i )
{
    char c = strIn[i];
    c --;

    sOut += c;
}
return sOut;
}
```

- 3. Save the source code as a file in the code editor and then close the code editor.**
- 4. Compile the application, using your favorite compiler on your favorite operating system.**
- 5. Run the application on your favorite operating system.**

You might think that this approach would have an impact on the developers who were using our class. In fact, we can make these changes in our class (check out the resulting program on this book's companion Web site as `ch1_1a.cpp`) and leave the remainder of the application alone. The developers don't have to worry about it. When we compile and run this application, we get the following output:

```
$ ./ch1_1a.exe "hello"
Input String: [hello]
Encoded String: [ifmmp]
Decoded String: [hello]
1 strings encoded
```

As you can see, the algorithm changed, yet the encoding and decoding still worked and the application code didn't change at all. This, then, is the real power of encapsulation: It's a black box. The end

users have no need to know how something works in order to use it; they simply need to know what it does and how to make it do its thing.

Encapsulation also solves two other big problems in the programming world:

- By putting all the code to implement specific functionality in one place, you know exactly where to go when a bug crops up in that functionality. Rather than having to chase the same code in a hundred scattered places, you have it in one place.
- You can change how your data is internally stored without affecting the program external to that class. For example, imagine that in the first version of the code just given, we chose to use an integer value rather than the string key. The outside application would never know, or care.



If you really want to “hide” your implementation from the user — yet still give the end user a chance to customize your code — implement your own types for the values to be passed in. Doing so requires your users to use your specific data types, rather than more generic ones.