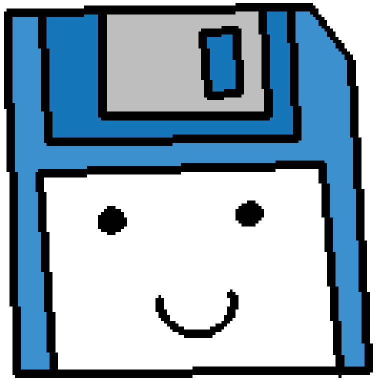
Templates

Written by Rachel J. Morris, last updated 2016-04-16

One size fits all*



** may not fit all, not legally responsible for lack of fit*

Overloading for same functionality

Currently, we can utilize function overloading so that we can have multiple functions with the same name and different parameter list.

```
float Sum( float arr[], int size );
int Sum( int arr[], int size );
Fraction Sum( Fraction arr[], int size );
```

This might be useful if we want to be able to use the same functionality for different data-types.

Overloading for same functionality

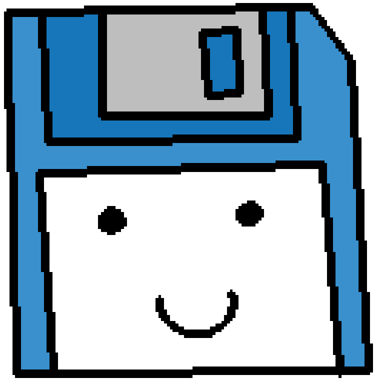
But in some cases, this means duplicating some code...

```
float Sum( float arr[], int size )
{
    float sum = 0;
    for ( int i = 0; i < size; i++ ) { sum += arr[i]; }
    return sum;
}
```

```
int Sum( int arr[], int size )
{
    int sum = 0;
    for ( int i = 0; i < size; i++ ) { sum += arr[i]; }
    return sum;
}
```

So can we write the function once, and use it with any data-type?

Function Templates



Function Templates

By utilizing templates...

* note for this example, all data-types passed in must have `operator+` overloaded to work.

```
template <typename T>
T Sum( T arr[], int size )
{
    T sum = 0;
    for ( int i = 0; i < size; i++ ) { sum += arr[i]; }
    return sum;
}
```

… we can make a function that will handle any* data-type

```
int numbers[] = { 4, 3, 6, 2, 1 };
int isum = Sum( numbers, 5 );
cout << "Int Sum: " << isum << endl;

float prices[] = { 2.99, 3.19, 5.29 };
float fsum = Sum( prices, 3 );
cout << "Float Sum: " << fsum << endl;
```

Function Templates

To create a function that uses a template to be a “placeholder” for a variable's data-type, we need to prefix the function with the **template prefix:**

`Template <typename T>`

```
template <typename T>
void OutputResult( T arr[], int size, T sum )
{
    for ( int i = 0; i < size; i++ )
    {
        if ( i != 0 )
            cout << " + ";

        cout << arr[i];
    }
    cout << " = " << sum << endl;
}
```

You can put anything you want in place of “T”, but T is the standard in much of the code that you will see.

Function Templates

Second, you will use "T" in place of the data-type that would change.

Note that you can still have hard-coded data-types in the parameter list and as local variables in the function.

Template

Not a template

Template

```
template <typename T>
void OutputResult( T arr[], int size, T sum )
{
    for ( int i = 0; i < size; i++ )
    {
        if ( i != 0 )
            cout << " + ";

        cout << arr[i];
    }
    cout << " = " << sum << endl;
}
```

Function Templates

As long as the variable passed into the template has all the needed functionality that is used within the function, it should be O.K.

```
template <typename T>
void OutputResult( T arr[], int size, T sum )
{
    for ( int i = 0; i < size; i++ )
    {
        if ( i != 0 )
            cout << " + ";

        cout << arr[i];
    }
    cout << " = " << sum << endl;
}
```

For this function, any data-types that T represents must have `operator<<` overloaded.

Function Templates

```
template <typename T>
T Sum( T arr[], int size );

template <typename T>
void OutputResult( T arr[], int size, T sum );

int main()
{
    int numbers[] = { 4, 3, 6, 2, 1 };
    int isum = Sum( numbers, 5 );
    OutputResult( numbers, 5, isum );

    float prices[] = { 2.99, 3.19, 5.29 };
    float fsum = Sum( prices, 3 );
    OutputResult( prices, 3, fsum );

    return 0;
}
```

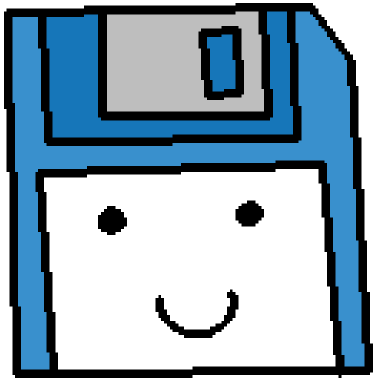
Then, calling the functions is simple – it looks like any other function call.

Function Templates

```
template <typename T1, typename T2>
void Display( T1 itemA, T2 itemB )
{
    cout << itemA << "\t\t" << itemB << endl;
}

int main()
{
    int num = 50;
    string str = "hello";
    float price = 9.99;

    Display( num, str );
    Display( str, num );
    Display( num, price );

    return 0;
}
```

You can also have multiple type parameters as well, if you require different data-types that are not hard-coded.

Class Templates



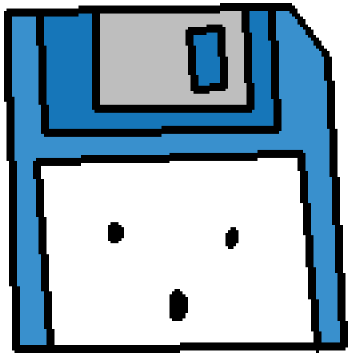

Good news, everyone!
Templates can also be used with classes!

Class Templates

Templates can also be used with classes in order to build more complex structures.

For example: A class that contains a dynamic array, and handles the memory management and resizing functionality internally.

By making it a template, we only need one "Dynamic Array" object, for any data-type!



Sound familiar?

```
vector<string>      names;
list<float>         prices;
map<int, string>    idToStudents;
```

Class Templates

```
template <typename T>
class Array
{
    public:
    void Set( int index, T value )
    {
        if ( index < 0 || index >= 10 )
            return ;
        data[ index ] = value;
    }

    T Get( int index )
    {
        if ( index < 0 || index >= 10 )
            return NULL;
        return data[ index ];
    }

    void DisplayAll()
    {
        for ( int i = 0; i < 10; i++ )
        {
            cout << data[ i ] << ", ";
        }
        cout << endl << endl;
    }

    private:
    T data[10];
};
```

Similar to with a function template, we start with the template prefix.

Class Templates

```
template <typename T>
class Array
{
    public:
    void Set( int index, T value )
    {
        if ( index < 0 || index >= 10 )
            return ;
        data[ index ] = value;
    }

    T Get( int index )
    {
        if ( index < 0 || index >= 10 )
            return NULL;
        return data[ index ];
    }

    void DisplayAll()
    {
        for ( int i = 0; i < 10; i++ )
        {
            cout << data[ i ] << ", ";
        }
        cout << endl << endl;
    }

    private:
    T data[10];
};
```

Then, anywhere we use this “type” within the class, we use T.

And we write the rest of the functionality normally.

Class Templates

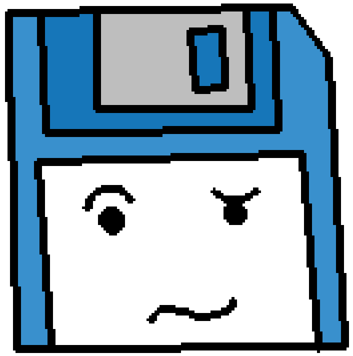
```
template <typename T>
class Array
{
    public:
    void Set( int index, T value )
    {
        if ( index < 0 || index >= 10 )
            return ;
        data[ index ] = value;
    }

    T Get( int index )
    {
        if ( index < 0 || index >= 10 )
            return NULL;
        return data[ index ];
    }

    void DisplayAll()
    {
        for ( int i = 0; i < 10; i++ )
        {
            cout << data[ i ] << ", ";
        }
        cout << endl << endl;
    }

    private:
    T data[10];
};
```

When we declare a variable using this class, we must specify the type within < and > signs.

```
Array<int>      numbers;
numbers.Set( 0, 10 );
numbers.Set( 1, 13 );
numbers.DisplayAll();

Array<string>   words;
words.Set( 0, "Cat" );
words.Set( 1, "Puppy" );
words.Set( 4, "Crocodile" );
words.DisplayAll();
```

Then we simply call the functions and pass in the data-type we need.

But keep in mind...



Definitions in the class...

```
template <typename T>
class Array
{
    public:
    void Set( int index, T value )
    {
```

```
    T Get( int index )
    {
```

```
    void DisplayAll()
    {
        for ( int i = 0; i < 10; i++ )
        {
            cout << data[ i ] << ", ";
        }
        cout << endl << endl;
    }

    private:
    T data[10];
};
```

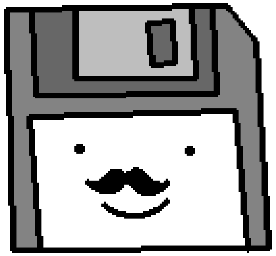
When you view classes using templates written by other people, you might notice that the function **definitions** tend to be stored within the class **declaration**, all in one .h or .hpp file.

Different compilers may handle templates differently, but this is the safest thing to do – include the entire class and its function definitions all in one header file.

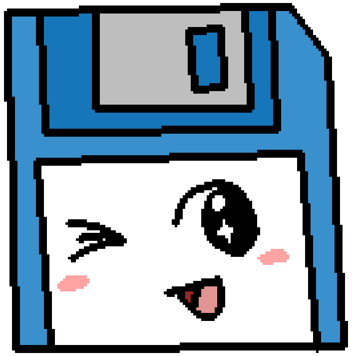
Definitions in the class...

```
template <typename T>
class Array
{
    public:
    void Set( int index, T value )
    {
    ---------------------------------
    T Get( int index )
    {
    ---------------------------------
    void DisplayAll()
    {
        for ( int i = 0; i < 10; i++ )
        {
            cout << data[ i ] << ", ";
        }
        cout << endl << endl;
    }

    private:
    T data[10];
};
```

There are ways to work around this, but for the time being implementing function definitions within the class declaration is generally good enough.



Practice!



Practice

Let's implement a Dynamic Array wrapper class and use templates!

Then you won't have to implement a class that takes care of memory allocation and resizing anymore, you'll already have a generic class for it! ;)

(Though really, I tend to just use std::vectors anyway)