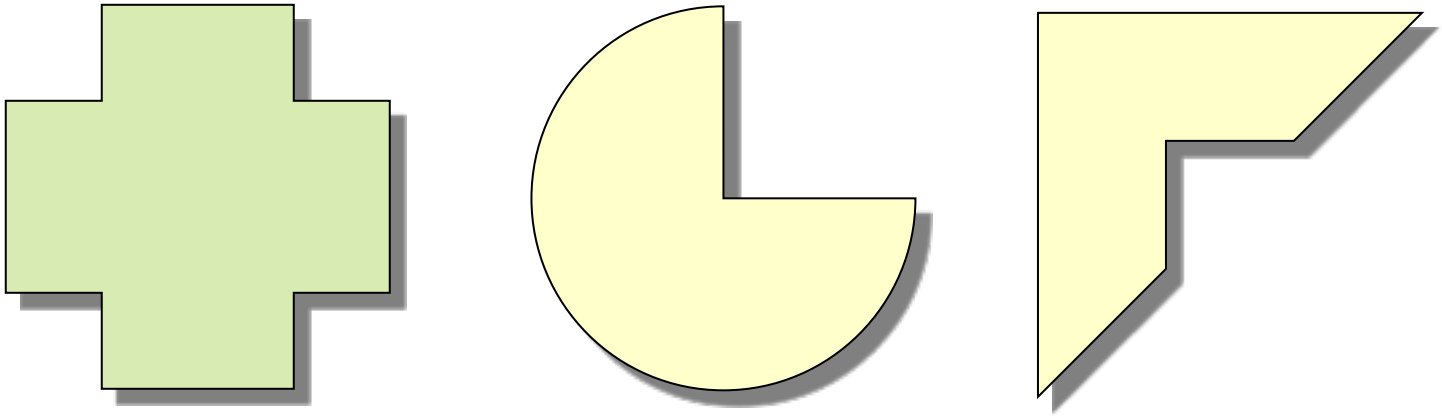
Chapter 12
Templates



Today's Objectives
(12.2 – 12.4)

- Understand the motivation and benefits of **templates** (12.2)
- **Template functions** (12.2)
- **Template classes** (12.4)

Overload functions

```
int Min(int val1, int val2)
{     if ( val1 < val2 )
            return val1;
      else
            return val2;
}
double Min(double val1, double val2)
{     if ( val1 < val2 )
            return val1;
      else
            return val2;
}
char Min(char val1, char val2)
{     if ( val1 < val2 )
            return val1;
      else
            return val2;
}
```

The only difference between these functions are the argument types

The C++ compiler will pick the correct version of **Min()** based on its argument

```
int main()
{
    cout << Min( 10, 5 ) << endl;
    cout << Min( 2.5, 13.2 ) << endl;
    cout << Min( 'A', 'B' ) << endl;
}
```

We can define **generic functions** using templates!

The **template** prefix specifies a generic parameter type **T**

```
template< typename T >
T Min2( T val1, T val2 )
{
    if ( val1 < val2 )
        return val1;
    else
        return val2;
}
```

The compiler builds Min2() functions based on the argument types

```
int main()
{
   cout << Min2( 10, 5 ) << endl;
   cout << Min2( 2.5, 13.2 ) << endl;
   cout << Min2( 'A', 'B' ) << endl;
}
```

Try it out

- Create a new project, and copy Min2<T> into a source file
- Test it with the three statements given

- Add your own template version of Max2<T>

Remember how to **swap** values?

```
void Swap(int & v1, int & v2)
{
    int temp = v1;
    v1 = v2;
    v2 = temp;
}
```

Exercise:
Creating a swap template function

- Create a template function called

```
template< typename T >
void Swap( T & v1, T & v2 )
```

Remember the template rules for functions

1. Add the prefix code:

```
template <typename T>
```

2. Replace every occurrence with the data type in the function with the parameter type, e.g.

```
void Swap(T &first, T &second) {
        T temp = first;
        …
}
```

Generic Sort Function

- See how an array sort function is converted to a **generic template sort** in **section 12.3** of your textbook.

```
template<typename T>
void sort(T list[ ],  int listSize)
```

Exercise 1

- Consider your homework assignment EX04_02
- Modify the function **DoubleCapacity()** and make it a **template function**.
- Test your new template function.

Generic Template Classes!

```
template <typename T>
class Storage
{
private:
    T data;
public:
    T get_store() {
        return data;
    }
    void set_store(const T &item) {
        data = item;
    }
};
```

Declaring objects of a template class

```
Storage <int>  intStore; // T is int
intStore.set_store(4);
cout << intStore.get_store() << endl;


Storage<string> strStore; // T is string
strStore.set_store("eddie");
cout << strStore.get_store() << endl;


Storage<char>  charStore; // T is char
```

Remember the template rules for classes

1. Add the **prefix** code:

```
template <typename T>
```

2. Replace every occurrence of the data type that needs to be **generic** in the class with the **parameter type**, e.g.

```
T data;
```

3. Specify the **actual type** when declaring object of the template class type

```
Storage<int> intStore;
```

Exercise 2

- Try out the template **Storage** class.
- Test it using the code given for ints and strings
- Complete the test for chars
- Try it for your MyInteger class (EX03_05)

Summary

- Templates provide the ability to use multiple types in a given function or class. (Much like overloading functions – except in one definition)
- The compiler will use the templates to construct functions (or classes) that take the types of arguments used in the program