Templates

Swapping Values

A common operation is swapping the values of two variables:

```
void swap(int& value1, int& value2) {
    int temp = value1;
    value1 = value2;
    value2 = temp;
}
```

With this function defined, we can easily swap two ints:

```
int n1 = 42;
int n2 = 10;
swap(n1, n2);
```

Swapping Values

Unfortunately, we'd need another function to swap two strings:

```
void swap(string& value1, string& value2) {
    string temp = value1;
    value1 = value2;
    value2 = temp;
}
```

Now we can also swap two strings:

```
string s1 = "Emperor Penguin";
string s2 = "King Penguin";
swap(s1, s2);
```

Swapping Values

We'd need yet another for chars:

```
void swap(char& value1, char& value2) {
    char temp = value1;
    value1 = value2;
    value2 = temp;
}
```

Now we can also swap two chars:

```
char c1 = 'X';
char c2 = 'O';
swap(c1, c2);
```

Sensing a pattern here?

We need an entire function for each data type,
but the logic itself is exactly the same for each one!

Repeating Ourselves Sucks...

You might think to use a typedef:

```
typedef ________ val;
void swap(val& value1, val& value2) {
    val temp = value1;
    value1 = value2;
    value2 = temp;
}
```

This allow us to adapt the function to a new type with one change...

- however, the swap function work for a single data type (whatever the typedef is)
- it must be recompiled to work with a new type

But there's a better way!

No more gross repetition!

Templates

We could define a template for the swap function:

```
template <typename Item>
void swap(Item& value1, Item& value2) {
    Item temp = value1;
    value1 = value2;
    value2 = temp;
}
```

The compiler creates a version for each data type we use it with

- this will work for ANY data type...
- as long as it has a copy constructor and an assignment operator

Templates

Using templated functions:

```
int n1 = 10, n2 = 20;
string s1 = "a", s2 = "b";
double d1 = 3.14, d2 = 1.01;

swap(n1, n2); // compiler creates a swap for ints
swap(s1, s2); // compiler creates a swap for strings
swap(d1, d2); // compiler creates a swap for doubles
```

Templates

Another example:

```
template <typename Item>
void minimum(const Item& v1, const Item& v2) {
    return (v1 < v2) ? v1 : v2;
}
```

This method will work with any data type...

- as long as it supports comparison via operator <

Templates

To make a template function, add a template declaration before it:

```
template <typename Item>
```

About this statement:

- this lets the compiler know that this is a *pattern* for creating a function, using Item as some unspecified type
- the compiler will create the actual function and code when it encounters a call to this function with a specific type
- this declaration must go before *both* the prototype and the implementation of a function
- the template parameter (Item) must appear in the parameter list of the function, or the compiler will complain about failed unification errors
- Item is not a required name, but it is common to see. You can use any valid identifier you like, but the convention is to have its first letter as uppercase

Templates

Take a look at this templated function:

```
template <typename Item>
Item array_max(const Item array[], size_t size) {
    // find and return the largest value
}
```

C++ will try to match argument types *exactly*...

```
int my_array[] = { 1, 2, 3, 2, 1 };
size_t size = 5;

// this will work (size is a size_t variable)
cout << array_max(my_array, size) << endl;
```

Templates

Take a look at this templated function:

```
template <typename Item>
Item array_max(const Item array[], size_t size) {
    // find and return the largest value
}
```

C++ will try to match argument types *exactly*...

```
int my_array[] = { 1, 2, 3, 2, 1 };
const size_t size = 5;

// this will NOT work (a const size_t variable O_o)
cout << array_max(my_array, size) << endl;
```

Templates

Take a look at this templated function:

```
template <typename Item>
Item array_max(const Item array[], size_t size) {
    // find and return the largest value
}
```

C++ will try to match argument types *exactly*...

```
int my_array[] = { 1, 2, 3, 2, 1 };



// this will NOT work (5 is an integer)
cout << array_max(my_array, 5) << endl;
```

Templates

An example with two template parameters:

```
template <typename Item, typename SizeType>
Item array_max(const Item array[], SizeType size) {
    Item max = array[0];

    for (SizeType i = 1; i < size; i++) {
        if (array[i] > max) max = array[i];
    }

    return max;
}
```

Templates

An example with two template parameters:

```
template <typename Item, typename SizeType>
Item array_max(const Item array[], SizeType size) {
    // find and return the largest value
}
```

Now this function will work with any data type for size

- this includes size_t, const size_t, integers, whatever!

You can specify as many template arguments as you want

- each must appear in the argument list at least once, though

Templated Classes

Template Classes

A templated class can easily store different underlying data types

- this is especially useful for container classes, like our bag class
- the STL makes extensive use of template classes for its data structures

For example, we might want:

- a bag of ints
- a bag of doubles
- a bag of strings
- or a bag of Penguins...

A typedef helps, but still only allows one data type at a time

- a templated class can allow for as many data types at once as we need!

Template Classes

The bag class as a template:

```
template <typename Item>
class bag {
    public:
        typedef Item value_type;

    private:
        Item* data; // an array of items
};
```

Template Classes

Inside the class declaration, the compiler knows about the Item type

- outside of the class declaration, we need to make a number of modifications to let C++ know about the dependency on the Item template argument

Some general rules:

- the `template <typename Item>` prefix is placed before each function prototype and implementation that uses the templated class
- each use of the class name should be changed to refer to the templated class name:

Template Classes

Non-templated version:

```
// constructor implementation

bag::bag() {
    // create an empty bag
}
```

Template Classes

Templated version:

```
// constructor implementation
template <typename Item>
bag<Item>::bag() {
    // create an empty bag
}
```

Notice:

- the implementation is preceded by: `template <typename Item>`
- the class prefix is now `bag<Item>::`, rather than just `bag::`

Template Classes

Non-templated version:

```
// copy constructor implementation

bag::bag(const bag& source) {
    // copy a bag
}
```

Template Classes

Templated version:

```
// copy constructor implementation
template <typename Item>
bag<Item>::bag(const bag<Item>& source) {
    // copy a bag
}
```

Notice:

- the implementation is preceded by: `template <typename Item>`
- the class prefix is now `bag<Item>::`, rather than just `bag::`
- the argument type is now `bag<Item>`, rather than just `bag`

Template Classes

Non-templated version:

```
// insert method implementation

void bag::insert(const value_type& entry) {
    // insert an item
}
```

Template Classes

Templated version:

```
// insert method implementation
template <typename Item>
void bag<Item>::insert(const Item& entry) {
    // insert an item
}
```

Notice:

- the implementation is preceded by: `template <typename Item>`
- the class prefix is now `bag<Item>::`, rather than just `bag::`
- the argument type is now `Item`, rather than `value_type`

Template Classes

Non-templated version:

```
// global operator + implementation

bag        operator +(const bag& b1,
                      const bag& b2)
{
    // add two bags together
}
```

Template Classes

Templated version:

```
// global operator + implementation
template <typename Item>
bag<Item> operator +(const bag<Item>& b1,
                     const bag<Item>& b2)
{
    // add two bags together
}
```

Notice:

- the implementation is preceded by: `template <typename Item>`
- the argument type is now `bag<Item>`, rather than just `bag`

Template Classes

The compiler creates new versions of methods for each data type

- to do this, it must have <u>direct access</u> to the implementations to be able to create new version of the functions for each new data type
- normally, we just #include the header files, which gives us the prototypes, but not the implementations themselves

Template implementations must be provided <u>in the header file</u>

- `#include` the implementation file at the bottom of the class header file
- remove the `#include` for the .h file from the implementation file
- because the implementation file gets `#included` into the header file, make sure not to have any `using namespace` directives, as these will be forced upon anyone who uses your class header file

Another alternative is to `#include` the .cpp file when using the class