templates

part 2

refresher on templates

a template is like a “blueprint” for generating a function, class, or variable, allowing for the use of generic types

template parameters can be types or nontypes (values)

non-variadic templates have a fixed number of parameters

template functions/classes are not **instantiated** until used with specific template parameters, at which point the compiler creates the specific function/class

class templates
(from C++ Primer)

class template example

`Blob` class holds a collection of elements

when we copy a `Blob` object, we want the original and copy to refer to the same underlying elements, that is the class must allocate resources with a lifetime independent of the original object

```
Blob<string> b1; // empty Blob
{ // new scope
    Blob<string> b2 = {"a", "an", "the"};
    b1 = b2; // b1 and b2 share the same elements
} // b2 is destroyed, but the elements in b2 must not be destroyed
  // b1 points to the elements originally created in b2
```

how would the behavior of this code differ if we used `std::vector` instead?

class template example

```
template <typename T> class Blob {
public:
    typedef T value_type;
    typedef typename std::vector<T>::size_type size_type;
    // constructors
    Blob();
    Blob(std::initializer_list<T> il);
    // number of elements in the Blob
    size_type size() const { return data->size(); }
    bool empty() const { return data->empty(); }
    // add and remove elements
    void push_back(const T &t) {data->push_back(t);}
    void push_back(T &&t) { data->push_back(std::move(t)); }
    void pop_back();
    // element access
    T& back();
    T& operator[](size_type i);
private:
    std::shared_ptr<std::vector<T>> data;
    // throws msg if data[i] isn't valid
    void check(size_type i, const std::string &msg) const;
};
```

instantiating a class template

unlike with a function template, the compiler cannot deduce template parameter types for a class template, they must be supplied explicitly in the angle brackets:

```
Blob<int> ia;                 // empty Blob
Blob<int> ia2 = {0,1,2,3,4};  // Blob with five elements
Blob<string> names;           // Blob that holds strings
```

a type-specific version of `Blob` is instantiated from each definition:

```
template <> class Blob<int> {
    typedef typename std::vector<int>::size_type size_type;
    Blob();
    Blob(std::initializer_list<int> il);
    // ...
   int& operator[](size_type i);
private:
    std::shared_ptr<std::vector<int>> data;
    void check(size_type i, const std::string &msg) const;
};
```

references to template type in template scope

when reading template class code, remember that the name of the class template is not the name of a type

a class template is used to instantiate a type, and an instantiated type always includes template arguments

```
std::shared_ptr<std::vector<T>> data;
```

here `data` is the instantiation of `shared_ptr` that points to the instantiation of `vector` that holds objects of type `T`

member functions of class templates

member functions can be defined either inside or outside the class body

if defined outside, we must include the template parameters of the class template and say to which class the member belongs:

```
template <typename T>
void Blob<T>::check(size_type i, const std::string &msg) const
{
    if (i >= data->size())
        throw std::out_of_range(msg);
}
```

member functions of class templates

```
template <typename T>
T& Blob<T>::back()
{
    check(0, "back on empty Blob");
    return data->back();
}

template <typename T>
T& Blob<T>::operator[](size_type i)
{
    // if i is too big, check will throw, preventing access to a nonexistent element
    check(i, "subscript out of range");
    return (*data)[i];
}

template <typename T>
void Blob<T>::pop_back()
{
    check(0, "pop_back on empty Blob");
    data->pop_back();
}
```

exercise: defining `Blob` constructors

```
template <typename T> class Blob {
public:
    // constructors
    Blob();
    Blob(std::initializer_list<T> il);
    // ...
private:
    std::shared_ptr<std::vector<T>> data;
    /// ...
};
```

what does the default constructor do here?

write the parameterized constructor `Blob(std::initializer_list<T> il)`

(hint: see https://en.cppreference.com/w/cpp/memory/shared_ptr/make_shared)

exercise: defining `Blob` constructors

```
template <typename T> class Blob {
public:
    // constructors
    Blob();
    Blob(std::initializer_list<T> il);
    // ...
private:
    std::shared_ptr<std::vector<T>> data;
    /// ...
};
```

what does the default constructor do here?

write the parameterized constructor `Blob(std::initializer_list<T> il)`

(hint: see https://en.cppreference.com/w/cpp/memory/shared_ptr/make_shared)

```
template <typename T>
```

exercise: defining `Blob` constructors

```
template <typename T> class Blob {
public:
    // constructors
    Blob();
    Blob(std::initializer_list<T> il);
    // ...
private:
    std::shared_ptr<std::vector<T>> data;
    /// ...
};
```

what does the default constructor do here?

write the parameterized constructor `Blob(std::initializer_list<T> il)`

(hint: see https://en.cppreference.com/w/cpp/memory/shared_ptr/make_shared)

```
template <typename T>
Blob<T>::Blob(std::initializer_list<T> il)
```

exercise: defining `Blob` constructors

```
template <typename T> class Blob {
public:
    // constructors
    Blob();
    Blob(std::initializer_list<T> il);
    // ...
private:
    std::shared_ptr<std::vector<T>> data;
    /// ...
};
```

what does the default constructor do here?

write the parameterized constructor `Blob(std::initializer_list<T> il)`

(hint: see https://en.cppreference.com/w/cpp/memory/shared_ptr/make_shared)

```
template <typename T>
Blob<T>::Blob(std::initializer_list<T> il):
    data(std::make_shared<std::vector<T>>(il)) { }
```

instantiation of class-template member functions

by default, a member function of a class template is instantiated only if the program uses that member function

```
// instantiates Blob<int> and the initializer_list<int> constructor
Blob<int> squares = {0,1,2,3,4,5,6,7,8,9};
// instantiates Blob<int>::size() const
for (size_t i = 0; i != squares.size(); ++i)
    squares[i] = i*i; // instantiates Blob<int>::operator[](size_t)
```

if a member function isn’t used it isn’t instantiated

thus we could instantiate a class with a type that may not meet the requirements for some of the template’s operations

member templates of class templates

can define a member template of a class template, in which case both have their own, independent template parameters

```
template <typename T> class Blob {
    template <typename It> Blob(It b, It e); // constructor taking two iterators denoting range to copy
    // ...
};
```

to define outside the class template body, we must provide template parameter list for the class template and the function template

```
template <typename T>   // type parameter for the class
template <typename It>  // type parameter for the constructor
    Blob<T>::Blob(It b, It e):
        data(std::make_shared<std::vector<T>>(b, e)) { }
```