Introduction to Generic Programming

Introduction

C++ has an extremely powerful generic programming capability based around its template features. Java uses the term Generics to describe its generic programming capability, the ability to define a code blueprint that can be instantiated for any (reference) data type. At a cursory glance, it is tempting to suggest that C++ templates and Java generics are about the same thing in each language. While it is true that C++ templates provide similar capabilities to Java generics, the nature of the template system in C++ enables a much greater capability. To be fair to Java, however, there are some generics features without (at least not yet) equivalent capabilities in C++ templates.

This series starts with the basic C++ template capabilities, showing the typical kinds of uses you've come to expect with generic programming. Once the standard background is understood, the more advanced (whatever advanced means), or interesting, capabilities are presented.

References

- https://www.fluentcpp.com/

Function Templates

The first difference to note between Java generics and C++ templates is that templates may be written at the function level in C++. Java only has methods, therefore, generics only apply at the class level. Because C++ has both functions and methods (classes), templates can be written per function, in addition to classes.

The general form for a template function is:

```
template <typename type>
[return value] functionName([parameters])
{
	// function body
}
```

Where:

- `template` is a required C++ keyword.
- `typename` is a required C++ keyword. The keyword `class` is often used, but there are times where `typename` and `class` are different.
  - https://blogs.msdn.microsoft.com/slippman/2004/08/11/why-c-supports-both-class-and-typename-for-type-parameters/
  - https://stackoverflow.com/questions/2023977/difference-of-keywords-typename-and-class-in-templates
- `type` is a placeholder for the parameterized template data type. There can be more than one parameterized type, each is separated by a comma. For example `template <typename T, typename R>`.
- `[return value]` is the return type for the function.
- `[parameters]` are the function parameters.

Let's start by writing a generic `xtoy` function that returns the result of 'x' to the power of 'y'. This function should take any numeric data type 'x' and raise it to the integer power 'y'. The following code is the C++ template for such a function:

```
template <typename T>
```

```
T xtoy(T x, T y)
{
    T result = 1;
    for (int i = 0; i < y; i++)
    {
        result *= x;
    }

    return result;
}
```

For this function `T` is the parameterized type of the function, used both as the function parameters type and the return value type.

The compiler will correctly match against the following code:

```
std::cout << xtoy(2, 2) << std::endl;
```

But it won't match with this:

```
std::cout << xtoy(2.2, 2) << std::endl;
```

The reason is that there are two different data types in the arguments, a `double` and an `int`. The template function has only a single parameterized data type. It is possible to allow for two different data types in the function parameters:

```
template <typename T, typename R>
T xtoy(T x, R y)
{
    T result = 1;
    for (int i = 0; i < y; i++)
    {
        result *= x;
    }

    return result;
}
```

For this function, we were careful to select `T` as the return type, because we always want the first parameterized type to be the return type. While this works, if the function is called as `xtoy(2, 2.2)`, the return type is an `integer`, which probably isn't what the developer wants. Rather than have the second parameter parameterized, it should be set to require an (positive) `int` type. The final version of the function is:

```
template <typename T>
T xtoy(T x, unsigned int y)
{
    T result = 1;
    for (decltype(y) i = 0; i < y; i++)
    {
        result *= x;
    }

    return result;
}
```

This function has a single parameterized type `T`, as the first parameter and an `unsigned int` as the second. This will correctly match `xtoy(2.2, 2)` and cause a compiler warning on a call like `xtoy(2.2, 2.2)`.

Template Instantiation

The term instantiation or template instantiation is important to understand when it comes to writing template code. When we say a template (function or class) is instantiated when it is bound to a specific set of template parameters. When the

compiler encounters a template, it parses the code and verifies the syntax is correct, but does not generate object code. At this time the compiler can't verify if it makes sense for any particular data type (C++ 20 is going to improve this dramatically). Consider the following template function:

```
template <typename T>
void report(T data)
{
    std::cout << "There are " << data.size() << " elements in the vector" << std::endl;
    for (auto item : data)
    {
        std::cout << "  " << item << std::endl;
    }
}
```

This function assumes the data type has a `.size()` member, along with the necessary iterator (you'll see how to write those soon!) to work in a range-based for loop.

When the compiler first parses the template, it can only verify the syntax looks correct. But it can't yet verify it will work for any or all particular data types. It needs to wait for the template to be instantiated (used) before it can then write a version of the code for that data type and then compile it (as a translation unit). Consider the following code:

```
std::vector<int> primes1 = {1, 2, 3, 5};
report(primes1);

std::array<int, 4> primes2 = {1, 2, 3, 5};
report(primes2);

int primes3[4] = {1, 2, 3, 5};
report(primes3);
```

The first two instantiations of the `report` function, using an `std::vector` and `std::array`, both compile because they both have the necessary members as used in the template function, even though they are different data types with completely different implementations. However, the third instantiation results in a compiler error, because a raw array doesn't have the required members used in the template function.

This purpose of this is to help explain why it is possible to write template code that passes the compiler, but then, at a later time the compiler fails when the template is bound to specific parameters and (attempted to be) instantiated.

Good reference: https://www.experts-exchange.com/articles/1199/Separating-C-template-declaration-and-implementation.html

Non-Type Template Parameters

Usually we think of the template type parameters as placeholders for a data type, one without a value. However, it is possible to define what is called a non-type template parameter, basically a placeholder for a value. The syntax for doing this looks like a regular function parameter. Consider the following template function:

```
template <typename T, unsigned int R>
void repeat(T value)
{
    for (decltype(R) i = 0; i < R; i++)
    {
        std::cout << value << std::endl;
    }
}
```

The template parameter R is a non-type parameter, it is a value (of a specified type). Throughout the template function, the non-type parameter can be used as an `int` value.

Template functions are normally deduced and matched based upon the parameters to the function call. For functions with non-type parameters, the syntax is different. The following are examples of how to call this template function:

```
repeat<std::string, 4>("Hi Mom!");
repeat<double, 6>(3.14159);
```

The most obvious difference is the use of the angle brackets with template parameters specified between them. The first parameter, in this case, is the function argument type (`std::string` and `double`). The second parameter is the non-type parameter, a value, which must be an `unsigned int`. The non-type template parameter is bound to `R` and wherever `R` is used in the template function, the value is used.

Why not have the function simply take a second `unsigned int` parameter, why make it a non-type template parameter? It probably should be a second function parameter, rather than the non-type parameter. I'm showing this here in preparation for something we'll do in the next section when talking about class templates.

Class Templates

Templates can also be defined at the class level. The general form for a class template is:

```
template <typename type>
class ClassName
{
      // class code goes here
};
```

Where `template`, `typename` and `type` all have the same meaning as with function templates.

To declare an object of this class type, it looks like: `ClassName<int> myObj;`

You've seen this before when using the standard library, declaring the type between angle brackets. The template type (or types) are declared between the angle brackets followed by the name of the variable of the type.

Implementation

With non-template class code, the class declaration is placed in a header file (`.hpp`) and the definition is placed in an implementation file (`.cpp`). For template code, on the other hand, all code is placed into the header file (`.hpp`); with a few exceptions that will be noted later. The definition for the various class members may be written inline style, or by first writing the class declaration, followed by writing the class definition, all in the same header file.

--- Trivial Example – Inline Style ---

```
template <typename T>
class MyClass
{
    public:
        MyClass(T data):
            m_data(data)
            {}

        T getData() { return m_data; }

    private:
        T m_data;
};
```

Not much to note here, other than the definition of the class constructor and the `getData` method are provided inline style.

--- Trivial Example – Separate Declaration & Definition ---

```
template <typename T>
class MyClass
{
    public:
        MyClass(T data);

        T getData();

    private:
        T m_data;
};

template <typename T>
MyClass<T>::MyClass(T data):
    m_data(data)
{
}

template <typename T>
T MyClass<T>::getData()
{
    return m_data;
}
```

In this example, the class is first declared and then immediately following the constructor and `getData` method are defined. Notice the `template <typename T>` is repeated before each member definition. Also note the special syntax used to identify the name of the class `MyClass<T>` before the scope resolution operator.

While it is tempting to write all template code inline style, it should **not** be done, except for trivial code. The reason is that all member functions are instantiated with the inline style, even if they aren't used. For templates written with separate declaration and definition, only those methods actually used are instantiated. This can lead to faster link times, and smaller executable code.

Example – Stack Template

Let's take a look at a full class template example, a stack class.

```
template <typename T>
class stack
{
  public:
       stack(std::uint16_t size) :
              m_size(size),
              m_top(0)
       {
              m_data = std::make_unique<T[]>(size);
       }

       void push(T value)
       {
              if (isFull())
                     throw new std::exception("Stack is full");

              m_data[m_top++] = value;
       }
```

```
        T pop()
        {
                if (isEmpty())
                        throw new std::exception("Stack is empty");

                return m_data[--m_top];
        }

        bool isFull()
        {
                return m_top == m_size + 1;
        }
        bool isEmpty()
        {
                return m_top == 0;
        }

  private:
        std::uint16_t m_size;
        std::uint16_t m_top;
        std::unique_ptr<T[]> m_data;
};
```

This code should look quite comfortable by this point. There is a simple template type parameter `T`. Rather than dynamically allocating and managing the data with a raw pointer, the code uses a smart pointer (`std::unique_ptr`) for managing the stack memory. This class template can be used as:

```
stack<std::string> stack(10);

stack.push("Clarkston");
stack.push("Newton");
stack.push("Amalga");
stack.push("Smithfield");

while (!stack.isEmpty())
{
        std::cout << stack.pop() << std::endl;
}
```

TODO: Template Specialization

Variadic Templates

References

- https://en.cppreference.com/w/cpp/language/parameter_pack
- http://kevinushey.github.io/blog/2016/01/27/introduction-to-c++-variadic-templates/
- https://eli.thegreenplace.net/2014/variadic-templates-in-c/

Let's say we want to write a template `max` function that returns the max of two numbers. It would look like:

```
template <typename T>
T max(T x, T y)
{
    return (x > y) ? x : y;
}
```

If we want to write a template max function that returns the `max` of three numbers, it might look like (assuming we still have the previous template function):

```
template <typename T>
T max(T x, T y, T z)
{
    return (x > max(y, z) ? x : max(y, z);
}
```

Continuing, what if we want to write a template `max` function for any number of arguments. It is easy to see this gets out of hand quickly. There is a better way...*variadic templates*.

A variadic template is a template function or class that accepts a variable number of template parameters, or specifically, a *template parameter pack* (or *parameter pack*). A template parameter pack is a parameter composed of zero or more template type arguments; it can be a combination of types and non-template types (or even other templates!).

The general form for declaring a template parameter pack for a template is:

```
template <typename... Ts>
[either a function or class declaration]
```

Notice the `...` (three dots) following the `typename` declaration; this indicates the type is a template parameter pack. The `Ts` is a common (but not required) placeholder for the types, with the 's' indicating it is plural; that it may hold zero or more types.

You might be surprised to find it isn't possible to access the individual types in a template parameter pack. You might expect it to have something like a `.size()` method to indicate how many items are in it and then some way to iterate over those types, but there isn't. *Hold this thought for a bit, we'll backtrack in just a bit and find a way to accomplish this.*

Using variadic templates, we might start by writing a `max` function like:

```
template <typename... Ts>
T max(Ts… ts)
{
    hmmm, what to do here?
}
```

A couple of issues with this code. The first is there is no type `T` to return. The second, because we don't have access to the individual types in `Ts`, we can't make any comparisons. The solution to these problems is to think recursively, as in, compile-time recursion/expansion. Consider the following code:

```
template <typename T>
T max(T x)
{
    return x;
}

template <typename T, typename... Ts>
T max(T x, Ts... ts)
{
    return (x > max(ts...)) ? x : max(ts...);
}
```

Rather than a single `max` function, there are two. The first one is the recursive base case, when there is only a single parameter. The second `max` function is when there are two or more parameters. The first parameter to the second function is a single type; it picks off the first parameter into this time, the remaining parameters are in the `Ts` parameter pack. When there are two or more parameters, `Ts` has 1 or more parameters in its pack. Then, a (compile time) recursive call to `max` is made. Which `max` function gets called depends on how many parameters are left in the parameter pack. If there are 0, then the first `max` function is called, otherwise the second is called. Pretty slick!

This approach is called pack expansion.

sizeof… operator

In the previous section, it was stated there isn't a member method like `.size()` on a parameter pack. But, there is an operator that allows the number of types in the parameter pack to be determined, the `sizeof…` operator. Consider the following code:

```
template <typename... Ts>
unsigned int howManyTypes(Ts... ts)
{
    return sizeof...(Ts);
}
```

This function returns the number of types (number of type arguments) used when it is called. The following code:

```
std::cout << "How many types: " << howManyTypes(1, 2.0, "3", true, false) << std::endl;
std::cout << "How many types: " << howManyTypes() << std::endl;
```

Produces the following output:

```
How many types: 5
How many types: 0
```

Iterators

Note to self: Remember you have written an `array2` type that allows the developer to specify the start/end indexes for arrays. The code for that is found in Code Samples/iterators.

References

- https://en.cppreference.com/w/cpp/iterator

Notice: *With the coming C++ 20 standard, a new approach to iterators is being introduced based on C++ concepts. It isn't a fundamental change in the requirements for iterators, but the implementation is based on C++ concepts. This section details writing iterators pre C++ 20.*

Consider the following code:

```
std::array<int, 5> primes = {2, 3, 5, 7, 11};
for (std::size_t i = 0; i < primes.size(); i++)
{
	std::cout << primes[i] << std::endl;
}
```

This is a good old for loop, or what I like to call a counted for loop. A loop like this is also called iteration, or again, counted iteration. Where the definition of *iteration* is to repeat a process.

The following loop is also iteration:

```
for (auto i = primes.begin(); i != primes.end(); ++i)
{
	std::cout << *i << std::endl;
}
```

This may be a little different than you've seen before, but it is looping over the items in the array. The `.begin` method returns what is called an iterator. The iterator has something of a complex type, therefore the `auto` declaration. In this case (in Visual Studio), the iterator type is `std::array<int, 5Ui64>::iterator`, where:

- `std` is the standard library namespace.
- `array<int, 5Ui64>` is the type of the array, built from the template parameters `<int, 5>`.
- `iterator` is a nested class inside of the `array<int, 5Ui64>` class. A closer look at the requirements for this class is detailed later.

The `.end` method returns a value that is *after the last element* of the container.

You'll notice the iterator is incremented using the prefix `++` operator. It can be incremented using the postfix `++` operator, but you'll soon see why you don't want to to that with non-primitive types, if you don't already know why. Access to the value represented by the iterator is through the dereference operator `*`.

Finally, the next loop is also iteration:

```
for (auto i : primes)
{
	std::cout << i << std::endl;
}
```

This is a range-based for loop, you've seen this already. The reason the `std::array` type can be used in the range-based for loop is because it provides an iterator. In fact, any data type in C++ that provides an iterator, that meets certain specifications, can be used in a range-based for loop.

Iterator Types

It is fairly technical what it means to be an iterator, I'll try to be as technical as necessary, but not to the extreme.

C++ supports six types of iterators: *InputIterator*, *OutputIterator*, *ForwardIterator*, *BidirectionalIterator*, *RandomAccessIterator* and *ContiguousIterator*. Each of these iterator types does pretty much what they sound like, although ContiguousIterator doesn't convey a strong meaning; refer to this link for a technical description: https://en.cppreference.com/w/cpp/named_req/ContiguousIterator.

One thing to know, C++ doesn't yet (but does with C++ 20) have a way to enforce or validate an iterator class has meets all the requirements it should. There is no inheritance hierarchy from which iterators derive and then provide implementations for abstract classes or methods. Instead, an iterator is defined by the methods it implements. If it provides all the functionality required of one of the iterator categories, it **is** that kind of iterator. It is possible to write an iterator class that doesn't meet all the language requirements, but still works in all the code situations you may need. However, in order to guarantee your iterator works in all the contexts it should based on it being one of the six types, it must implement all the requirements as specified by the language standard. For all the details, I highly recommend the following web page: https://en.cppreference.com/w/cpp/iterator.

All iterators must satisfy the requirements of an Iterator. These requirements are (refer to the previous web link for the full details):

- It must be copy constructable.
- It must be copy assignable.
- It must be destructuctable.
- l-values of the iterator must be swappable.
- It must have member typedefs (or `using` post C++ 11) for
  - `iterator_category`
  - `value_type`
  - `difference_type`
  - `reference`
  - `pointer`
- It must be dereferenceable.
- It must be incrementable.

Implementing an Iterator

Probably the best way to understand how to implement an iterator is to demonstrate an example on a custom container. At the same time, I want to show the combination of several parts of generic programming all together to create an interesting demonstration. For a good demonstration, we'll write our own `usu::array` type that works just like the `std::array` type. A few requirements for the type:

- Can't conflict with the `std::array` type. Therefore, it must be placed into its own namespace; we'll use `usu` as the namespace.
- Is a template type. The first template parameter is the type of the array elements, the second template parameter is a non-type that specifies the size of the container.
- Provides a default constructor.

- Provides a constructor that accepts an `std::initializer_list`.
- Provide array-like element access, overloading the `[]` operator.
- Exposes a `.size` method that returns the capacity of the container.
- Exposes `.begin` and `.end` methods that return a ForwardIterator.

Here is what the declaration for the `usu::array` type, without the iterator class, looks like:

```
namespace usu
{
    template <typename T, unsigned int N>
    class array
    {
      public:
        array() = default;
        array(const std::initializer_list<T>& list);

        reference operator[](unsigned int index);
        size_type size() { return N; }

        iterator begin() { return iterator(m_data); }
        iterator end() { return iterator(N, m_data); }

      private:
        T m_data[N];
    };
}
```

The template parameters are `T` and `N`, where `T` is a template type and `N` is a template non-type, an `unsigned int`. This allows for a declaration like: `usu::array<std::string, 4> myArray;` You'll notice the use of `reference` and `size_type` for two return types, those are explained in the section below on iterator type aliases.

We explicitly accept the default constructor the compiler writes and additionally declare a constructor that accepts an `std::initializer_list<T>`. The definition for this constructor is:

```
template <typename T, unsigned int N>
array<T, N>::array(const std::initializer_list<T>& list)
{
	if (list.size() > N)
	{
		throw new std::exception("Initializer list contains too many elements");
	}
	unsigned int pos = 0;
	for (auto i = list.begin(); i != list.end(); i++, pos++)
	{
		m_data[pos] = *i;
	}
}
```

The definition for the `[]` operator is:

```
template <typename T, unsigned int N>
typename array<T, N>::reference array<T, N>::operator[](unsigned int index)
{
	if (index < 0 || index >= N)
	{
		throw new std::exception("Index out of bounds");
	}
	return m_data[index];
}
```

Nothing remarkable about the code comprising the constructor and the `[]` operator.

The interesting and tricky part of the `usu::array` class is the declaration and definition of the *nested* iterator class. Looking at the `.begin` and `.end` methods, you can see they create and return instances of the iterator.

In order to satisfy the requirements of a *ForwardIterator*, we'll need the following in the (nested) iterator class:

- Default, (optionally) Overloaded, Copy, and Move constructors.
- Copy and Move assignment operators.
- Prefix and postfix increment operators.
- Dereference (`*`) operator.
- Equality (`==`) and inequality (`!=`) relational operators.
- Current position of the iterator in the array.
- A pointer to the array data from the parent class.

Iterator Type Aliases

The container classes in the standard library define aliases to various types used within the iterator itself. If you look at the code for `std::array` you'll see a series of `using` statements declaring the aliases. Some developers do this and others don't, as with everything, especially with C++, there is no universal accepted approach. For the example iterator in these notes, we'll follow the standard library convention.

This aliases are declared in the `public:` section of the `usu::array` class, along with the `iterator` class, like this…

```
template <typename T, unsigned int N>
class array
{
  public:
       using value_type = T;
       using size_type = std::size_t;
       using pointer = T*;
       using reference = T&;

       class iterator
       {
         public:
            using iterator_category = std::forward_iterator_tag;
```

This iterator defines aliases for `value_type`, `size_type`, `pointer`, and `reference`. Also notice inside the `iterator` class itself defines the `iterator_category` alias; this is because the nature of the iterator needs to be part of the iterator. Depending on other source you look for writing iterators, you'll see different patterns defining or not defining these aliases.

Iterator Position & Data Pointer

The best place to start is by defining two members of the `iterator` class that hold the current position of the iterator, along with a pointer to the instance of the nesting container (class). Because `usu::array` is a simple contiguous sequence of elements (an array), the position of the iterator is indicated by an index into the array.

```
private:
   size_type m_pos;
   pointer m_data;
```

Default & Overloaded Constructors

An iterator is only required to have a default constructor, to satisfy the *DefaultConstructable* requirement. It is typically useful, however, to be able to construct an iterator from a pointer to the container and also a starting position and pointer to the container. For the `usu::array` class, the following default and overloaded constructors are defined:

```
iterator() :
	iterator(nullptr)  // DefaultConstructable
{
}
iterator(pointer ptr) :
	m_data(ptr),
	m_pos(0)
{
}
iterator(size_type pos, pointer ptr) :
	m_pos(pos),
	m_data(ptr)
{
}
```

Copy & Move Constructors

In order to satisfy the *CopyConstructable* and *MoveConstructable* requirements, these two constructors are necessary.

Declaration

```
iterator(const iterator& obj); // CopyConstructable
iterator(iterator&& obj) noexcept;      // CopyConstructable, MoveConstructable
```

Definition

```
template <typename T, unsigned int N>
array<T, N>::iterator::iterator(const iterator& obj)
{
	this->m_pos = obj.m_pos;
	this->m_data = obj.m_data;
}

template <typename T, unsigned int N>
array<T, N>::iterator::iterator(iterator&& obj) noexcept
{
	this->m_pos = obj.m_pos;
	this->m_data = obj.m_data;

	obj.m_pos = 0;
	obj.m_data = nullptr;
}
```

Copy & Move Assignment Operators

In order to satisfy the *CopyAssignable* and *MoveAssignable* requirements, the assignment operator needs to be overloaded for these two operations:

Declaration

```
iterator& operator=(const iterator& rhs); // CopyAssignable
iterator& operator=(iterator&& rhs);      // CopyAssignable, MoveAssignable
```

Definition

```
template <typename T, unsigned int N>
typename array<T, N>::iterator& array<T, N>::iterator::operator=(const iterator& rhs)
{
	this->m_pos = rhs.m_pos;
	this->m_data = rhs.m_data;
```

```
        return *this;
}

template <typename T, unsigned int N>
typename array<T, N>::iterator& array<T, N>::iterator::operator=(iterator&& rhs)
{
        if (this != &rhs)
        {
                std::swap(this->m_pos, rhs.m_pos);
                std::swap(this->m_data, rhs.m_data);
        }

        return *this;
}
```

Prefix & Postfix Increment Operators

In order to satisfy the *Incrementable* requirement, the iterator must overload both the prefix and postfix operators:

Declaration

```
iterator operator++();    // incrementable e.g., ++r
iterator operator++(int); // incrementable e.g., r++
```

Definition

```
template <typename T, unsigned int N>
typename array<T, N>::iterator array<T, N>::iterator::operator++()
{
        m_pos++;
        return *this;
}

template <typename T, unsigned int N>
typename array<T, N>::iterator array<T, N>::iterator::operator++(int)
{
        iterator i = *this;
        m_pos++;
        return i;
}
```

Pay particular attention to the postfix increment operator. The semantics are “use the value, then increment.” The only way to do that through a function call is to make a copy of the iterator, increment the original, and return the copy. Therefore, when using the postfix increment operator on objects (but not primitives), it incurs a performance penalty; making all those copies. Unless there is a compelling reason otherwise, always use the prefix increment operator on objects.

Dereference Operator

In order to satisfy the *Dereferenceable* requirement, the dereference operator must be provided:

```
reference operator*()     // Derefrenceable
{
        return m_data[m_pos];
}
```

Equality & Inequality Relational Operators

Finally, iterators need to be able to be compared to each other for equality and inequality. Therefore, those relational operators need to be provided:

```
bool operator==(const iterator& rhs) { return m_pos == rhs.m_pos; }
bool operator!=(const iterator& rhs) { return m_pos != rhs.m_pos; }
```

With all of the above done, we now have an `usu::array` class that works (nearly) just like `std::array`, including the ability to be used in a range-based for loop.

```
template <typename T>
void report(T data)
{
    std::cout << "There are " << data.size() << " elements in the container" << std::endl;
    for (auto item : data)
    {
        std::cout << "  " << item << std::endl;
    }
}
```

--- main() ---

```
usu::array<int, 5> a1 = {0, 1, 2, 3, 4};       // std::initializer_list
usu::array<double, 10> a2;
usu::array<std::string, 15> a3;

std::cout << a1.size() << std::endl;
std::cout << a2.size() << std::endl;
std::cout << a3.size() << std::endl;

for (unsigned int i = 0; i < a1.size(); i++)   // operator[]
{
      std::cout << a1[i] << std::endl;
}

for (auto i = a1.begin(); i != a1.end(); i++)  // manual iterator
{
      std::cout << *i << std::endl;
}

for (auto element : a1)    // automatic iterator use
{
      std::cout << element << std::endl;
}

report(a1);   // Use the array in the templated 'report' function
```

The Curiously Recurring Template Pattern (CRTP)

In February 1995, James Coplien published a paper with the title, “Curiously Recurring Template Pattern” in which he describes a C++ pattern he had seen several times, one that intrigued him, leading him to illustrate and describe the pattern. This pattern has both been called an invention and a discovery, with James giving it the name, “Curiously Recurring Template Pattern.” It is worth noting he said the pattern had first been shown to him 5 years before the publication of his paper; in other words, the pattern has been around a long time! Independently, around the same time, this same pattern was “discovered” by Jan Falkin, a developer at Microsoft. This discovery resulted the Microsoft Active Template Library and Windows Template Library being designed around this feature of C++ templates.

References

- https://en.wikipedia.org/wiki/Curiously_recurring_template_pattern
- https://www.fluentcpp.com/2017/05/12/curiously-recurring-template-pattern/
- https://pixorblog.wordpress.com/2019/08/14/curiously-recurring-template-pattern-crtp-in-depth/

What is the CRTP?

It is a pattern, or idiom, in which a class `X` derives from another template class `Y`, using the class `X` itself as a template argument. Let’s take a look at that in code.

We start with a template-based class, where `T` is a template parameter...

```
template <typename T>
class Base
{
       ...class declaration/implementation...
};
```

Using this template, a concrete class is derived, instantiating the template using the derived class as the template argument…

```
class Derived : public Base<Derived>
{
       ...class declaration...
};
```

That is a little weird, but so what?

So this: Think about what it means that `Derived` is an argument to the `Base` template class. When `Base` is instantiated, the type `T` is `Derived`; anything defined as part of class `Derived` can be invoked from within class `Base`. In other words, `Base` can be written in terms of the implementation of `Derived`, even before `Derived` has been defined!

One way the CRTP is used is as simulated virtual functions, but without the overhead involved by needing a v-table and the run-time indirection. Another way of thinking about this is the base class uses “interface” that must be implemented by the derived class. The code below demonstrates this by having the base class act as an *interface* for derived classes, but calling into the derived classes implementation of the interface.

```
template <typename T>
class VehicleInterface
{
  public:
    void moveForward(double howFar)
    {
        T* impl = static_cast<T*>(this);
        impl->moveForwardImpl(howFar);
```

```
    }

    void turnRight(double angle)
    { // same type cast as above, just shortened into a single line
        static_cast<T*>(this)->turnRightImpl(angle);
    }

    void turnLeft(double angle)
    {
        static_cast<T*>(this)->turnLeftImpl(angle);
    }
};
```

Notice the definition of the public methods. They all first obtain a pointer to the instance itself, but of type `T`, not of type `VehicleInterface`, and then invoke an "`Impl`" method that is not yet defined. `VehicleInterface` doesn't declare/define any "`Impl`" methods, instead, it is expecting type passed into parameter `T` to provide those implementations. Therefore, the this pointer can't be used directly to call the "`Impl`" methods. Instead, a pointer to the template parameter `T` is needed, which in reality IS `this` (because of inheritance) in order to call the "`Impl`" methods.

With the `VehicleInterface` defined, it is now possible to create classes that derive from this class, but use themselves as the `T` argument. The following are two examples of using CRTP to implement this interface.

```
class Car : public VehicleInterface<Car>
{
  public:
    void moveForwardImpl(double howFar)
    {
        std::cout << "Car moves forward: " << howFar << std::endl;
    }

    void turnRightImpl(double angle)
    {
        std::cout << "Car turned right: " << angle << std::endl;
    }

    void turnLeftImpl(double angle)
    {
        std::cout << "Car turned left: " << angle << std::endl;
    }
};

class Motorcycle : public VehicleInterface<Motorcycle>
{
  public:
    void moveForwardImpl(double howFar)
    {
        std::cout << "Motorcycle moves forward: " << howFar << std::endl;
    }

    void turnRightImpl(double angle)
    {
        std::cout << "Motorcycle turned right: " << angle << std::endl;
    }

    void turnLeftImpl(double angle)
    {
        std::cout << "Motorcycle turned left: " << angle << std::endl;
    }
};
```

With these two different concrete classes defined, the next step is to use them. While `Car` and `Motorcycle` both derive from `VehicleInterface`, they provide a different type to the template parameter. Because of this, there is no common base class that can be used in the same way a there is a base class when using the C++ language defined polymorphism

through virtual methods. Therefore, in order to write code that can use both of these class, template code is necessary, as shown next.

```
template <typename T>
void driveVehicle(std::string title, T& vehicle)
{
    std::cout << title << std::endl;

    vehicle.moveForward(2.2);
    vehicle.turnLeft(4.4);
    vehicle.turnRight(6.6);
}

Car c;
Motorcycle m;

driveVehicle("--- Driving a Car ---", c);
driveVehicle("--- Driving a Motorcycle ---", m);
```