1. **Iterators and the STL:**

STL is basically sets of C++ templated classes that are designed for C++ users to be able to write more efficient and expressive code. It is a library of containers, algorithms, functions, and iterators. It is a part of C++ Standard Library, which contains much more than STL like: internationalization, diagnosis, numeric issues, streams, etc. STL is engineered with uniform design of the interfaces which allows for construction of new components in STL way much easier. There are various kinds of algorithms in the STL. Some are read only while other can manipulate the content inside the container. We can roughly divide the STL algorithms into following categories: Search Algorithms (binary\_search), Sorting Algorithms(sort, heap\_sort), Numeric Algorithms(accumulate, partial\_sum), Non transforming/ Modifying Algorithms (count, equal, mismatch), and Transforming/Modifying algorithms (swap, reverse).

In terms of containers in STL, there are three kinds of containers that is available for us to use. They are: Sequential Containers (vector, deque, arrays, lists, forward\_list), Associative Containers (set, multiset, map, multimap), and Unordered Containers(unordered\_set, unordered\_multiset, unordered\_map, unordered\_multimap).

STL also includes some classes that overload the function call operators. Example of such a class are known as function objects or functors. Functors are basically objects that can be treated as though they are function or function pointer. To create a functor, we create a object that overloads the operator ().

One of the most crucial components of STL are iterators. They are the objects that enable traversal of the containers in some order, for either reading or writing. Iterators are defined as templates and must comply with a very specific set of rules in order to qualify as one of many types of iterators. Iterators work like pointer, but it has higher level of abstraction. For example: the ++ operation in case of vector might mean increment in a unit memory position but it will not be valid in case of a binary tree. That is where iterator can help as every class in STL defines their own ++ within their iterator which can help user of the class to iterate over the next item without having knowledge of internal implementation of the class.

Iterators are divided into four subcategories mainly for performance reasons. Not all iterators are supported within every container. For example: list does not support random access iteration while vector does support random access iteration.

|  |  |
| --- | --- |
| input iterator | Read only, forward moving, only sequential access (++it, it++), only one pass, suitable for input streams like keyboard buffers or read-only files  Supports equality/inequality comparisions like a==b, a!=b  Can be dereferenced as an rvalue e.g. \*a, a->m |
| output iterator | Write only, forward moving, only single pass, suitable for output streams, such as screen text, write only files  Can be dereferenced as lvalue e.g \*a=t, \*a++=t |
| forward iterator | Both read and write, combination of input and output iterator, forward moving, support for multiple passes of container, suitable for linked list  Neither dereferencing nor incrementing affects dereferenceability. E.g { b=a; \*a++; \*b; } |
| bidirectional iterator | Read and write, forward and backward moving, combination of forward iterator and backward traversal, suitable for doubly linked list (list, set)  Can be decremented like: --a, a--, \*a-- |
| random access iterator | Read and write, random access, combination of bidirectional iterator and random access with the help of index, suitable for array or vectors.  Supports arithmetic operators(+,-), inequality comparisons(<,>,<= and >=), compound assignment operations (+=) and dereference operator ([]) |

(All containers require their iterators to comply with the capabilities of some of these types. Some are rather relaxed, and some are more rigorous. )

STL iterators are the bridge that connects the containers to the algorithms. The relationship between iterators, container and algorithm can be described by a sentence, “containers make iterators available, algorithms use them”. For instance: if we look at find() algorithm, it does not need to know anything about what kind of container it is dealing with. All it needs is proper implementation of iterator for the container. As soon as a class has implemented its iterator, find() or other algorithm will use the iterator (++, !=, \* operators) to access or write the content of the container. Find() will basically start from the container’s start() and iterate until it reaches end() in search of the given element. By the end, if the element was found, the position of the iterator as it traverses the entire content of the container using ++ operator specified by container’s iterator will be somewhere in between begin() and end() otherwise it will be at the end(). Here, algorithms like find() do not need to know the inner working of the class, it can simply be implemented as a template whereby we plug in which iterator type we want to use. Algorithm will make use of the iterators to query, manipulate the container. This is how iterators are used to connect Container and Algorithm in STL.

1. **Templates:**

A template is a class or function that we can parameterize with our own set of types. For example: we wrote our Safe Array class as a template. Therefore, we could instantiate Safe Array of any type as needed, even Safe Array of Safe Array. We use templates to represent concepts that can apply to various types and we can generate function or class of that specific type when we want to by specifying arguments. Template Specialization is the process of generating a special class or function from the general templated class by using the provided template arguments. This specialization can be either explicit (full) or partial.

**Explicit Specialization** happens when we explicitly write the body of class of certain type. For instance: If we have an existing templated Heap class, it might not work the same for type ‘const char \*’. There might be cases in which we might even want different methods for a special type. This is where explicit specialization comes into picture. Here the template-parameter list is empty. However, we append the template argument for which we are specializing to the template name.

**Partial Specialization** is useful when we want to specialize group of certain types. Instead of explicitly specializing for each type, we can partially specialize for the group of types that might have the same implementation. For instance: instead of now specializing Heap for double \* and other pointer type, we can simply write a partial specialization for pointer types. The syntax of partial specialization is similar to that of complete specialization, but the template parameter is not empty. Partial specialization therefore allows separate implementation of class for certain subset of types that needs to be implemented differently from the primary template.

One of the application of templates and their specialization is in Metaprogramming. Metaprogramming refers to a particular type of programming techniques that helps us ‘program a program’. In other words, when we write a metaprogram, system will be able to generate new code by itself to implement different functionality.

#include <iostream>

#include <cstdlib>

#include <cassert>

using namespace std;

// primary template to compute 3 to the Nth

template <int N>

class Factorial

{

public:

static int const result = N \* Factorial<N - 1>::result;

};

//Complete specialization base case

template <>

class Factorial<0>

{

public:

static int const result = 1;

};

int main()

{

Factorial<4> a;

cout << a.result << endl;

}

* 1. *Meta-program to calculate Factorial*

As we can clearly see, templates are heavily used on the above code where we calculate the factorial for any number using recursion and template specialization. However, there is much more to the code than just that; the code calculates the factorial during compile time. It might sound strange at first, but this is one of the power of using meta programming. Basic idea of how above code runs is as follows. The first template implements the general recursive rule. The template Factorial<> needs to compute the value of its static variable result which has recursive call to Factorial<N-1>: N\* Factorial<N-1>. Now, another factorial template is instantiated and the process goes on until the base specialization template is called whose result value is 1. This is how, the value of result is calculated during the instantiation process of various template classes.

#include <iostream>

#include <cstdlib>

#include <cassert>

using namespace std;

// primary template to compute 3 to the Nth

template <int N, int LO = 1, int HI = N>

class Sqrt

{

public:

static int const mid = (LO + HI + 1) / 2;

static int const result = (N < mid \* mid) ? Sqrt<N, LO, mid - 1>::result : Sqrt<N, mid, HI>::result;

};

//Base Case using partial specialization

template <int N, int M>

class Sqrt<N, M, M>

{

public:

static int const result = M;

};

int main()

{

Sqrt<82> a;

cout << a.result << endl;

}

* 1. *MetaProgram to Calculate Square Root*

Like the example above 2.2, calculating square root using meta programming is somewhat similar in which there is regular primary template and there is base specialization template. However, base specialization in this case is partial as it contains template as parameter. It is also different in a way that it uses method of bisection to get to the answer. However, concept of recursion is still the same like in the previous example and all meta programming examples. The first template is invoked with the template parameter value of 82 or N and two other optional parameter which are automatically set to 1 and 82(N). Now the same template will be called recursively until 2nd parameter’s value becomes equal to 3rd parameter’s value during which the value of either of 2nd or 3rd is put into original calculation for result. This happens all during compile time during the template instantiation from the primary and partial template.

#include <iostream>

#include <cstdlib>

#include <cassert>

#include <string>

using namespace std;

template <int DIM, typename T>

class DotProduct

{

public:

static T result(T \*a, T \*b)

{

return \*a \* \*b + DotProduct<DIM - 1, T>::result(a + 1, b + 1);

}

};

// partial specialization as end criteria

template <typename T>

class DotProduct<1, T>

{

public:

static T result(T \*a, T \*b)

{

return \*a \* \*b;

}

};

// convenience function

template <int DIM, typename T>

inline T dot\_product(T \*a, T \*b)

{

return DotProduct<DIM, T>::result(a, b);

}

int main()

{

int a[3] = {1, 2, 3};

int b[3] = {5, 6, 7};

cout << "dot\_product<3>(a,b) = " << dot\_product<3>(a, b)

<< '\n';

}

* 1. *Meta-program to calculate Dot product*

We have already seen example of how we can translate the concept of recursion in meta programming. However, the above example 2.4 goes a step beyond. It translates the concept of loop in terms of recursion to get the loop working in metaprogramming. It is achieved by setting the number of times we want to run the loop as the template parameter. Now the recursion is done as many times as the template parameter during which each of the array passed as parameter to static function is accessed and their product is returned. Result of products are accumulated in the first template as it continues to create rest of the template recursively with N-1 as template parameter until the base partial specialization is reached.

1. **Smart Pointers: (Skipped)**
2. **Traits and Policies:**

When we try to write a general primary template class, we want to make it easy to use and specialize by introducing as many parameters as possible. However, more parameters there is, harder it becomes for users of the class to call the class. We also know that not all parameters are equally significant. Some tend to have default values while some seems to be derived from the other main parameter. Policy and Traits are tools that facilitates the management of the extra parameters.

Bjarne Stroustrup gave his insight to trait class, “Think of a trait as a small object whose main purpose is to carry information used by another object or algorithm to determine policy or implementation details”. Traits classes provide additional information about a type, typically by defining typedefs or constants inside trait. Some of the purposes of traits are explained with code example below:

1. Fixed Traits: There might be problem when template is instantiated for one type but it turns out depending on the what the instantiated type is, the other sub parameter required to solve the problem may vary. In such situation, we can bind one type to use certain other type in order to be able to perform optimally and even for getting to the right answer.

#include <iostream>

#include <cstdlib>

#include <cassert>

using namespace std;

template <typename T>

class AccumulationTraits;

template <>

class AccumulationTraits<char>

{

public:

typedef int AccT;

};

template <>

class AccumulationTraits<short>

{

public:

typedef int AccT;

};

template <>

class AccumulationTraits<int>

{

public:

typedef long AccT;

};

template <>

class AccumulationTraits<unsigned int>

{

public:

typedef unsigned long AccT;

};

template <>

class AccumulationTraits<float>

{

public:

typedef double AccT;

};

template <typename T>

inline typename AccumulationTraits<T>::AccT accum(T const \*beg, T const \*end)

{

// return type is traits of the element type

typedef typename AccumulationTraits<T>::AccT AccT;

AccT total = AccT();

while (beg != end)

{

total += \*beg;

++beg;

}

return total;

}

int main()

{

// create array of 5 integer values

int num[] = {1, 2, 3, 4, 5};

// print average value

cout

<< "the average value of the integer values is "

<< accum(&num[0], &num[5]) / 5

<< "\n";

// create array of character values

char name[] = "templates";

int length = sizeof(name) - 1;

// (try to) print average character value

cout << "the average value of the characters in " << name << "\" is " << accum(&name[0], &name[length]) / length << "\n";

}

In the above program, if we were to simply call T() to initialize total in accum() function, we would end up getting wrong answer when trying to find the average of characters in a word. However, by implementing AccumulationTraits class we were able to hold a trait for each of the parameter type, and consequently we were able to specify what accumulatorType (accT) each parameter type is supposed to hold. We had accumulatorType of integral even for character parameter type where the regular template would have failed. This is how trait can represent additional type information related to a given main type.

1. Value Traits:
2. Parameterized Traits: Once a traits are defined, it cannot be overridden easily. When we really want to override, we