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:** When it comes to trait classes, they can not only represent additional type information but also constants and other class values that we might need.

We initialized zero value to accT using default constructor for a given parameter type, however we can achieve the default value by specifying the constant value of zero for every AccumulationTraits parameter type.

template <typename T>

class AccumulationTraits;

template <>

class AccumulationTraits<char>

{

public:

typedef int AccT;

static AccT const zero = 0;

};

template <>

class AccumulationTraits<short>

{

public:

typedef int AccT;

static AccT const zero = 0;

};

template <>

class AccumulationTraits<int>

{

public:

typedef long AccT;

static AccT const zero = 0;

};

Here, each of the specialized template has a constant that is evaluated during the compile time and therefore when initializing total in accum() template function, we can simply get the zero variable by using scope resolution operator as so instead of calling default constructor. This is how traits can provide not just extra types but also all the necessary information that accum() might need about the element type.

AccT total = AccumulationTraits<T>::zero;

1. **Parameterized Traits:** Once traits are defined as fixed like in the first example we saw, it cannot be overridden easily. When we really want to override, we will have to use parameterized traits like so.

template <typename T,

typename AT = AccumulationTraits<T> >

class Accum

{

public:

static typename AT::AccT accum(T const \*beg, T const \*end)

{

typename AT::AccT total = AT::zero();

while (beg != end)

{

total += \*beg;

++beg;

}

return total;

}

};

Here, users of our class would never have to provide the second template arguments unless absolutely necessary. The extra optional parameter depends on the first one but user with extra need may even pass the extra parameter to make the template work as they intend.

Traits and Policies work hand in hand in order to enable user to write general code that fits different parameter types. Policies usually define function interfaces. It is a class or class template that defines an interface as a service to other classes. Suppose we wanted to implement the process of accumulation in the above code differently. Let’s say now we want multiplication instead of addition. However, we also don’t want to remove the code that does addition because we will need it at some point. That’s where we will define policies which is capable for doing addition or multiplication.

class SumPolicy

{

public:

template <typename T1, typename T2>

static void accumulate(T1 &total, T2 const &value)

{

total += value;

}

};

class MultPolicy

{

public:

template <typename T1, typename T2>

static void accumulate(T1 &total, T2 const &value)

{

total \*= value;

}

};

Now, after the implementation of different policies, we can include a default policy as our second template optional parameter and we can let user of our class switch to MultPolicy if necessary.

template <typename T,

typename Policy = SumPolicy,

typename Traits = AccumulationTraits<T> >

class Accum

{

public:

typedef typename Traits::AccT AccT;

static AccT accum(T const \*beg, T const \*end)

{

AccT total = Traits::zero();

while (beg != end)

{

Policy::accumulate(total, \*beg);

++beg;

}

return total;

}

};

Policies are used almost everywhere in STL. It is used in the string class, I/O streams, STL containers, iterators, etc. There is even a paradigm in C++ programming called Policy-based design. It has been understood as compile-time variant of strategy pattern and is somewhat related to template metaprogramming.

1. **Cache Friendly Code:**

Caching is the practice of storing data temporarily and retrieving it whenever the data is needed. In doing so, the more expensive I/O call to slower memory is reduced greatly thereby increasing the performance. Since we have a huge gap between the speed of the CPU and the speed of retrieval or write to a Hard Disk, we need Cache to fill in the gap and make computer faster.

Cache friendly code are the codes that follows principle of locality which says that programs tend to reuse data and instructions stored near the instructions that was used recently. So, we can use that principle to code in cache friendly manner. However, it is not straightforward to do so as there are various issues that we need to address in order to write cache friendly code.

1. Spatial Locality: First thing is we want to pack as much as related data into contiguous area in the memory so that all the related data can be retrieved at once. We also need to avoid data structure based on nodes that are connected by pointer as they are usually not stored in contiguous memory block.
2. Time: We also want to finish all possible operation on certain data structure at once rather than doing the operation one at a time after performing other operation in the middle. This way we can make sure the data we are operating on is always on the cache and we don’t get any miss that can hinder the performance. It is also better idea not to be in situation where we create lots of temporary variable for later use. It is better to do operate on them whenever we have hold of them.
3. Line Sharing: In order to write efficient and hardware aware code, we must try to avoid line sharing as much as possible. Simplest cache is usually direct mapped which means certain address in memory can only take specific spot in the cache. If we have many memory items that occupies the same spot in the cache, we are in big trouble. To prevent this, modern cache are set asscociative. A 4 way set associative means a memory can be stored in 4 different cache spots. When lots of memory that we are accessing happens to occupy those 4 cache spots, we are in trouble again.
4. Unpredictable branches: Modern compilers usually tries to reorder code or cache we might need by looking what’s ahead. However, if we have lots of unpredictable branches that can lead to any part of code, it becomes hard for compiler to prefetch any data. One example would be the use of goto statement. Goto can make a program have unpredictable behavior as it can lead to any part of code.
5. Using appropriate containers: We cannot use any containers for solving our task. Even though any container might solve the problem, we need to be able to realize which one would be most efficient for our usage. For example: if we want to read and write from a container frequently in random way, we must use something like vector. As a matter of fact, we might solve the problem using even list, but the performance would be hurt badly since it doesn’t allow random access and writes. Also, we need to be aware of how a particular container stores its content. For instance: vector stores its content in contiguous block while list doesn’t. We might therefore incur more cache miss while using list than vector.
6. Frequent use cases need to be optimized: Any program will often call some sets of functions most of the times. Those functions should be designed in a way that they possess good locality.
7. Nested Loop: When we write the nested loop, we need to try to minimize cache misses in the innermost loop since that is the content that tends to happen consecutively and consequently if they do not incur any cache miss, such code will be much efficient. Also, when we want to access 2-D array, usually we have to write nested loop. When accessing the element, we always try to access the rows first. This is because by default a row is stored contiguously one after another. Consequently we will have less cache miss this way.

// Cache-friendly version - processes pixels which are adjacent in memory

void addOneToEveryPixel(int \*\*image, int numOfRows, int numOfCols)

{

for (int i = 0; i < numOfRows; i++)

{

for (int j = 0; j < numOfCols; j++)

{

image[i][j] = image[i][j] + 1;

}

}

}

// Cache-unfriendly version - jumps around in memory for no good reason

void addOneToEveryPixel(int \*\*image, int numOfRows, int numOfCols)

{

for (int j = 0; j < numOfCols; j++)

{

for (int i = 0; i < numOfRows; i++)

{

image[i][j] = image[i][j] + 1;

}

}

}

1. **Std::string**

Explain in detail the problem it addresses and the various approaches to solving it.

Strings are one of the most important thing in programming. Without string, it would be almost impossible to think about writing any small or large scale application. In Facebook, <string> is the most included file and it accounts for whooping 18% of all the CPU time spent in std. Therefore, if we can somehow make strings better, it can significantly help in the performance of our overall application.

Basically, every string is the size of 8 byte which has the pointer to the heap containing the actual string. The actual strings are prefixed with size, capacity, reference count and the string data terminating at null terminator.

The most popular string is an empty string. It is not necessary to ask for heap space while creating new empty. In fact, the optimization would be the presence of a single global empty string which is 25 bytes (8+8+8 +1). And all the empty strings in our program will point to that global string. This can be done not only to empty string but also to every single string. Since a string cannot change, we store reference Count and have a global version of that string be available for any number of referencing purposes. When the reference count reaches zero, we can then release the memory on the heap for that string. This way we have to malloc only once, that is, when we first initialize a new string that does not yet have a global copy.

Old GCC string implementation also followed copy on write semantics. It basically means when we call copy constructor of the string, we simply copy the pointer and increment the reference counter for that string. We will only perform memcopy when we want to modify the string and thereby creating a new string. However, GCC does another trick. It stores reference Counter as refCount-1 which means when our program is loaded into zero initialized memory, our empty string does not require any additional processing before it comes in valid state.

However, there was still room for improvement in the implementation of the string. That’s why facebook came up with their own fbstring implementation which included small string optimization. The basic idea of small string optimization is we want to avoid as many mallocs as possible so that we don’t have to access random part of memory every time only to access the same string. fbstring stores the whole small string in the stack itself which will increase spatial locality (doesn’t have to access heap). Normal string however still has pointer pointing to a heap space that contains the actual data but the size and capacity was put in stack rather than the heap. And for larger string (>=255 bytes), a reference Count was stuck before the data in the heap so that we don’t ask for new memory space when copying these long strings. Fbstring was found to be faster than the old GCC despite its assembly code being much longer and containing branches because they had to check whether the string was small, normal or large which GCC didn’t have to do. This was all because of the memory layout of the program. Fbstring didn’t have to access heap at all for small strings while GCC had to go to heap and probably to different page in memory possibly incurring cache misses in order to access each and every string data and its prefixes. When Facebook switched over to fbstring from gcc’s string, the Facebook application had 1% performance boost.

So how exactly does SSO (Small String Optimization) improve the old GCC strings? First, we don’t have as many mallocs which tends to slow down a program. We can simply store small strings in the stack instead of having to do malloc. This is especially demonstrated by below code. The first string will be simply stored in the stack buffer while the string2 will be stored in the heap.

#include <iostream>

#include <string>

using namespace std;

void \*operator new(size\_t size)

{

cout << "Allocating from heap " << size << " bytes \n";

return malloc(size);

}

int main()

{

string string1("I m in Stack");

string string2("I m allocated in Heap");

}

Output:



Therefore, instead of generalizing string to a particular type which always stores its data on the heap, we can make our code much more efficient and cache friendly by dividing strings into different sub types where smaller string are stored in the stack itself while the longer ones are stored in the heap. Overall, this provides performance improvement over the regular string implementation.