Generic Programming

Compiled by
M S Anand

Department of Computer Science

Generic Programming

Concepts and Constraints

C++20 introduced a new feature called **concepts** that gives us a more direct and powerful way of specifying constraints on template parameters. This feature can help prevent errors and make code more readable by limiting what types can be used with a given template.

Constraints

Constraints are the conditions or requirements that need to be satisfied by template arguments when using templates. It allows the user to specify what types or values can be used with a template.

Types of Constraints

Conjunctions

It combines multiple constraints using AND (&) operator. It specifies that all the constraints within the conjunction must be satisfied for the overall constraint to be satisfied.

Disjunctions

It is used to combine multiple constraints with the help of the OR (||) operator. It specifies that at least one of the constraints within the disjunction must be satisfied for the overall constraint to be satisfied.

Atomic constraints

They are individual constraints that define specific requirements on types, expressions, or template arguments.

Generic Programming

Concepts and Constraints

What are concepts?

Concepts are used to specify the requirements of the template arguments. Concepts allow you to define constraints to your template. It is a way using which we can specify some constraints for our template arguments in C++.

This feature can help prevent errors and make code more readable by limiting what types can be used with a given template. Some of the advantages include:

- Enforcing type constraints on template parameters
- Making template error messages more readable and easier to understand
- Improving code documentation by explicitly specifying what types are allowed as parameters
- Preventing unintended template instantiations by restricting what types can be used.

C++20 Concepts

The standard library includes a collection of pre-defined concepts we can use. They're available from the <concepts> header:

```
#include <concepts>
```

For example, the std::integral concept can tell us if a type is an integer.

At their most basic usage, concepts look similar to type traits. We pass the type (and depending on the concept, possibly some additional arguments) and we get a boolean value representing whether or not the type meets the requirements of the concept.

This all happens at compile time.
Sample program: concept1.cpp

Generic Programming
Concepts and Constraints

There are four main ways we can use concepts within our code.

Option 1: Constrained Template Parameters

Where concepts deviate from type traits is that the language has built-in syntactic support to make them easier to use. This most notably applies when we want to use a concept to constrain the types that can be used by a template.

For example, we can replace typename with a concept to constrain a template parameter.

So, if we wanted to ensure our template type was an integer, we no longer need to manually create static assertions. Instead, we can just replace typename with a concept in our template parameter list.

Generic Programming
Concepts and Constraints

So, instead of writing this:

```
template <typename T>
void SomeFunction(T x) {
  // ...
}
```

We can instead write this:

```
template <std::integral T>
void SomeFunction(T x) {
  // ...
}
```

We're now fully documenting our type requirements, and the compiler will generate meaningful error messages if anyone tries to use our template in a way we didn't intend:

Sample program: concept2.cpp

Generic Programming
Concepts and Constraints

Concepts allow us to route function calls to different templates:
Sample code: concept3.cpp

We can also use identical syntax when creating template classes:
Sample code: concept4.cpp

Option 2: The requires Keyword

The addition of concepts also came with a new piece of syntax - the requires keyword. This keyword is used in a few different ways:

Below, we use the **requires** keyword in a function template. It has access to the types our template is using, and will return true if the template can handle those types.

We could rewrite our previous example using requires like this:
Sample code: concept5.cpp

Generic Programming
Concepts and Constraints

The requires method is more verbose than simply replacing typename, but it gives us some more options.

For example, we can use boolean logic.

In the below example, we allow our type to match one of two concepts, using a requires clause and the || operator:

Sample code: concept6.cpp

We're also not restricted to just using concepts within requires statements - we can use other compile-time techniques too. Below, we use the std::is_base_of type trait:

Sample code: concept7.cpp

Note
Most standard library-type traits have equivalent concepts. For example, the std::derived_from concept covers the same use cases as the std::base_of type trait, albeit the order of the two types is inverted.

We can use a requires statement with class templates in the way we might expect:

```
#include <concepts>
template <typename T>
  requires std::integral<T>
class Container {
  T Contents;
};
```

Option 3: Trailing Requires Clause

When creating function templates, we have the option of using a requires statement in a slightly different way. It can be placed between the function signature and the function body, like this:

```
template <typename T>
T Average(T x, T y)
  requires std::integral<T>
{
  return (x + y) / 2;
}
```

Option 4: Abbreviated Function Template

The final way we can use concepts is within abbreviated function templates. These allow us to create function templates by setting one or more parameter types to auto:

```
auto Average(auto x, auto y) {
  return (x + y) / 2;
}
```

Where we want to implement concepts here, we insert the concept before the auto type:

```
auto Average(std::integral auto x,
             std::integral auto y) {
  return (x + y) / 2;
}
```

Combining Concept Approaches

We are free to combine the previous techniques as we see fit. Below, we use a constrained template parameter to specify requirements for the first type, and a requires statement to restrict the second type.

For the template to be eligible for a given function call, all the requirements need to be met.

```
template <std::integral TFirst, typename TSecond>
  requires std::integral<TSecond> ||
  std::floating_point<TSecond>
void Function(TFirst x, TSecond y){}
```

Writing Our Own Concepts

Naturally, we're not restricted to just using the standard library concepts. We can use concepts from any source, or write our own.

A basic concept looks like this:

```
#include <concepts>

template <typename T>
concept Integer = std::integral<T>;
```

We can break it down into four components:

1. A template parameter list (note, this list cannot reference other concepts)
2. The concept keyword
3. The name of the concept
4. An expression that will return true if the concept is met, or false otherwise.

We can then use our concept in any of the usual ways:

Generic Programming
Concepts and Constraints

```
template <Integer T>
T Average(T x, T y) {
  return (x + y) / 2;
}
```

The previous concept is effectively just re-implementing the std::is_integral concept under a different name, but we can of course get more complex. The following concept combines two standard library concepts and will be satisfied if a type matches either:

```
#include <concepts>

template <typename T>
concept Numeric =
    std::integral<T> || std::floating_point<T>;

template <Numeric T>
T Average(T x, T y) {
  return (x + y) / 2;
}
```

Generic Programming
Concepts and Constraints

More complex requirements can be specified by an alternative form of the requires syntax, which looks somewhat like a function. Within the body of this syntax, we can write code that mostly looks like normal function code:

```
#include <concepts>

template <typename T>
concept Averagable =
  requires(T x, T y) {
    (x + y) / 2;
  };
```

But, what we're actually doing here is specifying requirements. In the above example, what we're saying is for a type to satisfy our concept, it must meet two requirements.

Firstly, an object of that type must be addable to another object of that same type.

More specifically, the type T must implement an + operator, where the second operand is also a T.

Secondly, the object returned from that operation must be divisible by 2. That is, it must return a type that implements the / operator, where the second operand is an int, or convertible to an int.

If a type meets those requirements, it is Averagable, otherwise, it is not.

Sample code: concept8.cpp

Prior to concepts, something like this would have been extremely difficult to set up, requiring advanced template metaprogramming. Now, it's fairly straightforward, and with most compilers, the error output is much better than we'd be able to achieve previously.

The output from the compiler might say something like:

It's telling us on line 17 of the cpp file, no function was found that could satisfy that specific call to Average.

It correctly identified our template function as a candidate, but ruled it out because of the type constraints we added.

It tells us exactly which constraint ruled the template out - our Averagable concept, when passed a std::string returned false

And it even tells us exactly why the concept returned false - because the std::string type does not define the / operator we specified as a requirement.

Exploring the standard concepts library

The standard library provides a set of fundamental concepts that can be used to define requirements on the template arguments of function templates, class templates, variable templates, and alias templates.

The standard concepts in C++20 are spread across several headers and namespaces. We will present some of them in this section although not all of them. You can find all of them online at https://en.cppreference.com/.

The main set of concepts is available in the <concepts> header and the std namespace. Most of these concepts are equivalent to one or more existing type traits. For some of them, their implementation is well-defined; for some, it is unspecified.

They are grouped into four categories: core language concepts, comparison concepts, object concepts, and callable concepts. This set of concepts contains the following (but not only):

Generic Programming
Concepts and Constraints

Concept	Description
`same_as`	Defines the requirement that a type `T` is the same as another type `U`.
`derived_from`	Defines the requirement that a type `D` is derived from another type `B`.
`convertible_to`	Defines the requirement that a type `T` is implicitly convertible to another type `U`.
`common_reference_with`	Defines the requirements that two types, `T` and `U`, have a common reference type.
`common_with`	Defines the requirement that two types, `T` and `U`, have a common type to which both can be convertible.
`integral`	Defines the requirement that a type `T` is an integral type.
`signed_integral`	Defines the requirement that a type `T` is a signed integral type.
`unsigned_integral`	Defines the requirement that a type `T` is an unsigned integral type.
`floating_point`	Defines the requirement that a type `T` is a floating-point type.
`assignable_from`	Defines the requirement that an expression of a type `U` can be assigned to an lvalue expression of a type `T`.

Generic Programming
Concepts and Constraints

Concept	Description
`swappable`	Defines the requirement that two values of the same type `T` can be swapped.
`swappable_with`	Defines the requirement that a value of a type `T` can be swapped with a value of a type `U`.
`destructible`	Defines the requirement that a value of a type `T` can safely be destroyed (without any exception thrown from the destructor).
`constructible_from`	Defines the requirement that an object of a type `T` can be constructed with the given set of argument types.
`default_initializable`	Defines the requirement that an object of a type `T` can be default-constructible (either value initialized `T()`, direct-list-initialized from an empty initializer list `T{}`, or default-initialized, as in `T t;`).
`move_constructible`	Defines the requirement that an object of a type `T` can be constructed with move semantics.
`copy_constructible`	Defines the requirement that an object of a type `T` can be copy constructed and move constructed.
`moveable`	Defines the requirement that an object of a type `T` can be moved and swapped.
`copyable`	Defines the requirement that an object of a type `T` can be copied, moved, and swapped.
`regular`	Defines the requirement that a type `T` satisfies both the `semiregular` and `equality_comparable` concepts.
`semiregular`	Defines the requirement that an object of a type `T` can be copied, moved, swapped, and default constructed.
`equality_comparable`	Defines the requirement that the comparison operator `==` for a type `T` reflects equality, meaning that it yields `true` if and only if two values are equal. Similarly, the `=!` comparison reflects inequality.
`predicate`	Defines the requirement that a callable type `T` is a Boolean predicate.

Generic Programming

Standard Template Library (STL)

The **Standard Template Library or STL** in C++ is a collection of template classes and template functions that provide a generic way of programming. It is a library of container classes, algorithms, and iterators.

It is commonly used for efficiently programming data structures, algorithms, and functions. Some built-in data structures include arrays, vectors, queues, etc.

Components of STL in C++

There are four major components of STL in C++:

1. Containers
2. Iterators
3. Algorithms
4. Function objects or Functors

Generic Programming

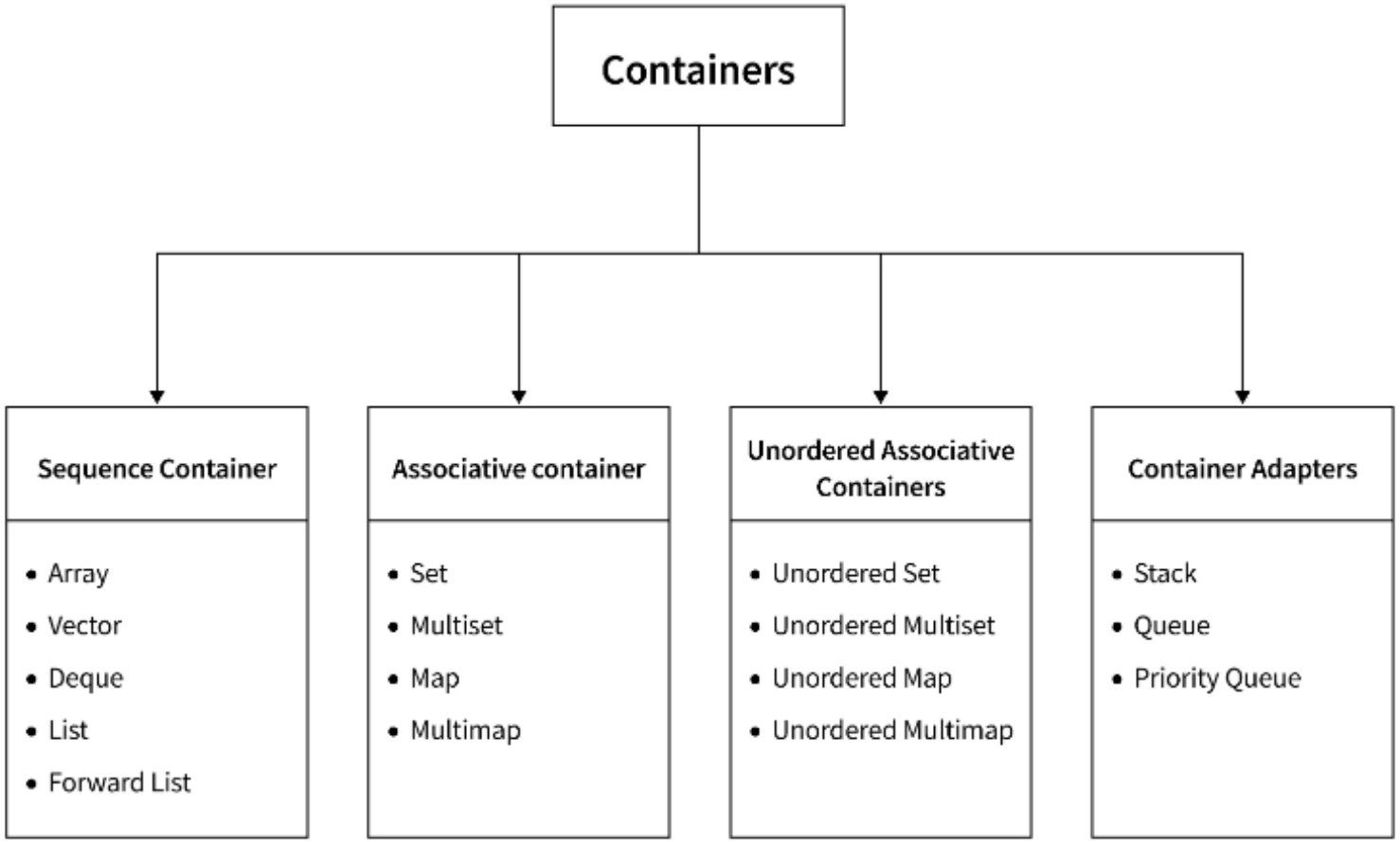
Standard Template Library (STL)

Containers

- A container is a holder object that stores other objects as elements. It is used to implement different data structures.
- They are implemented as class templates (templates use data types as parameters), allowing greater flexibility in the types supported as elements.
- The containers manage storage space for their elements and provide member functions to access them directly or through iterators.
- They replicate the most commonly used data structures like queues (queue), dynamic arrays (vector), stacks (stack), linked lists (list), heaps (priority_queue), trees (set), associative arrays (map), and many others.

Generic Programming

Standard Template Library (STL)



Generic Programming

Standard Template Library (STL)

There are 4 types of containers of STL in C++:

1. **Sequence Containers** - Array - Vector - Deque - List - Forward List
2. **Associative Containers** - Set - MultiSet - Map - Multimap
3. **Unordered Associative Containers** - Unordered Set - Unordered Multiset - Unordered Map - Unordered Multimap
4. **Container Adapters** - Stack - Queue - Priority Queue

You can also choose the most suitable C++ STL container that matches your needs. Different criteria used for the selection are:

- The functionality offered by the container.
- The efficiency of some members. It is especially true for sequence containers, which offer complex trade-offs between inserting/removing and accessing elements.

Sequence Containers

- Sequence containers implement data structures that can be accessed sequentially via their position.
- They preserve the insertion order of elements.
- These are internally implemented as arrays or linked lists.

The five sequence containers supported by STL are:

Array

Arrays are sequential homogeneous containers of fixed size. The elements are stored in contiguous memory locations.

Syntax:

array<object_type, size> array_name;

Generic Programming

Standard Template Library (STL)

Vector

Vectors are dynamic arrays, allowing the insertion and deletion of data from the end. They can grow or shrink as required. Hence their size is not fixed, unlike arrays.

Syntax:

vector<object_type> vector_name;

Deque

Deque is **double-ended** queue that allows inserting and deleting from both ends. They are more efficient than vectors in case of insertion and deletion. Its size is also dynamic.

Syntax:

deque<object_type> deque_name;

Generic Programming

Standard Template Library (STL)

List

The list is a sequence container that allows insertions and deletions from anywhere. It is a doubly linked list. They allow non-contiguous memory allocation for the elements.

Syntax:

list<object_type> list_name;

Forward List

Forward Lists are introduced from C++ 11. They are implemented as singly linked list in STL in C++. It uses less memory than lists and allows iteration in only a single direction.

Syntax:

forward_list<object_type> forward_list_name;

Associative Containers

- Associative container is an **ordered (sorted) container** that provides a fast lookup of objects based on the keys, unlike a sequence container which uses position.
- A value is stored corresponding to each key.
- They are internally implemented as binary tree data structures. This results in logarithmic time operations -- *O*(log*n*).

Types of associative containers:

Set

The set is used to store unique elements. The data is stored in a particular order (increasing order, by default).

Syntax:

set<object_type> set_name;

Map
The map contains elements in the form of unique key-value pairs. Each key can be associated with only one value. It establishes a **one-to-one** mapping. The key-value pairs are inserted in increasing order of the keys.
Syntax:
map<key_object_type, value_object_type> map_name;

Multiset
Multiset is similar to a set but also allows **duplicate** values.
Syntax:
multiset<object_type> multiset_name;

Multimap
Multimap is similar to a map but allows duplicate key-value pairs to be inserted. Ordering is again done based on keys.
Syntax:
multimap<key_object_type, value_object_type> multimap_name;

Unordered Associative Containers

- Unordered Associative Container is an **unsorted version** of Associative Container.
- It is important to note that **insertion order is not maintained**. Elements are in random order.
- These are internally implemented as a **hash table** data structure. This results, on average, in constant time operations -- *O*(1)

Unordered Set

It is used to store **unique elements**.

Syntax:

unordered_set<object_type> unordered_set_name;

Unordered Map

It is used to store unique key-value pairs.

Syntax:

unordered_map<key_object_type, value_object_type>
unordered_map_name;

Unordered Multiset
It is similar to an unordered set, but the elements need not be unique.
Syntax:
unordered_multiset<object_type> unordered_multiset_name;

Unordered Multimap
It is similar to an unordered map, but the duplicate key-value pairs can be inserted here.
Syntax:
unordered_map<key_object_type, value_object_type>
unordered_map_name;

Container Adapters in C++

- STL in C++ also consists of a special type of container that adapts other containers to give a different interface with restricted functionality. Hence the name Container Adapters.
- The underlying container is encapsulated in such a way that its elements are accessed by the members of the container adaptor independently of the underlying container class used.
- Unlike other containers, values are not directly initialized but with the help of other supported methods.

Stack

A Stack is a container that provides **Last-In-First-Out (LIFO) access**. All the operations occur at the same place called **top** of the stack. It is implemented on top of a deque by default.

Syntax:

stack<data_type> stack_name;

Queue

A queue is a container that provides **First-In First-Out access**. The insertion is done at the **rear** (back) position and deletion at the **front**. It is implemented on top of a deque by default.

Syntax:

queue<data_type> queue_name;

Priority Queue

A priority Queue is similar to a queue, but every element has a priority value. This value decides what element is present at the top, which is, by default, the greatest element in the case of STL in C++. This is similar to the max heap data structure. It is implemented on top of the vector by default.

Syntax:

priority_queue<object_type> priority_queue_name;

Generic Programming

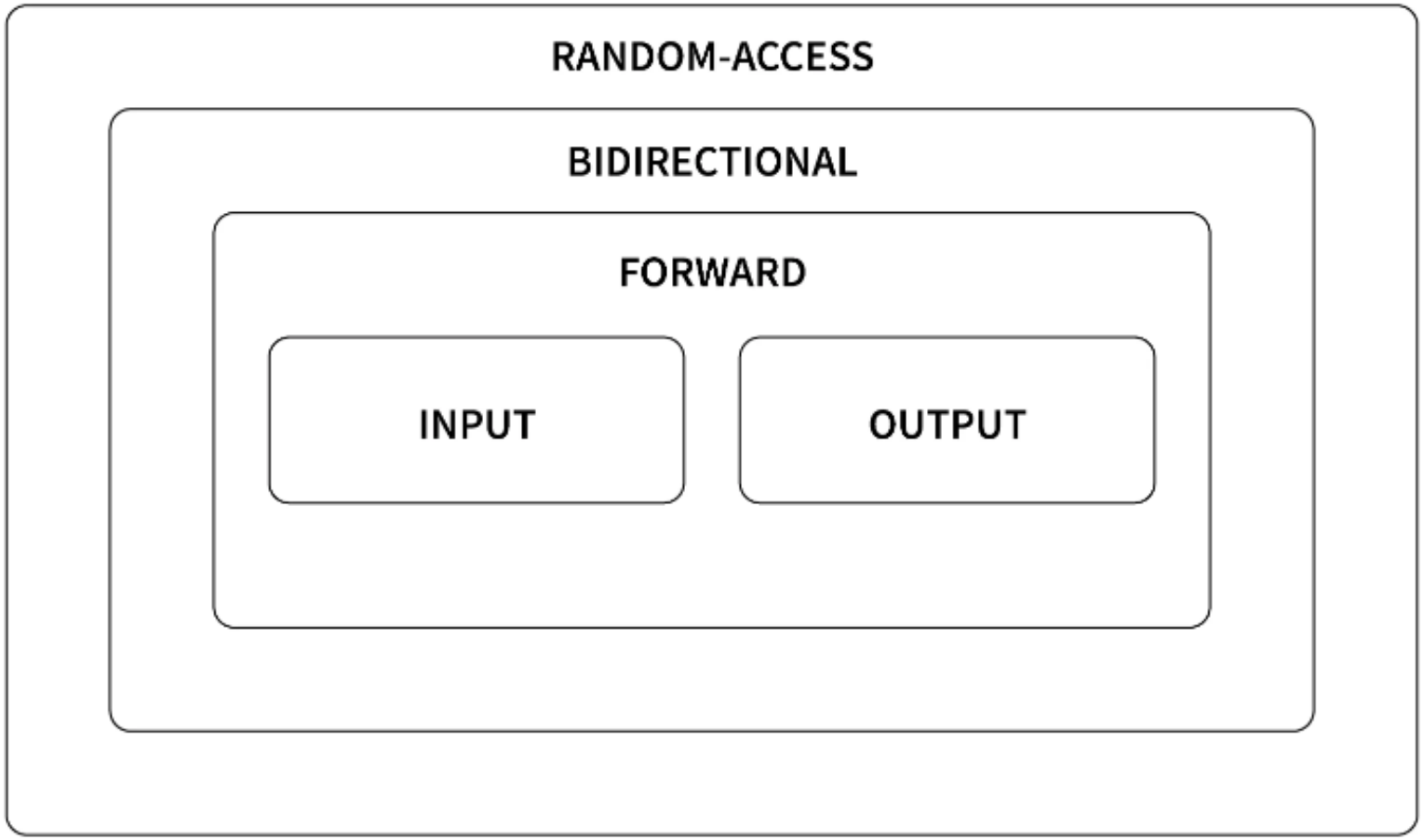
Standard Template Library (STL)

Iterators

- An Iterator is a pointer-like object that points to an element inside some container.
- You can use them to iterate over a container which means moving sequentially from one element to another.
- Iterators can also help you manipulate the data stored inside a container and connect algorithms to containers.
- Iterators provided by STL in C++ are analogous to using the cursor while typing some text. You can shift them to any position you want.
- Iterators are classified into five categories depending on the functionalities offered. You can refer to the diagram (next slide) Regarding functionality, the outermost iterator is the most powerful and the innermost the least powerful.

Generic Programming

Standard Template Library (STL)



The common syntax to declare an iterator:
container_name<object_type>::iterator iterator_name;

The two most commonly used iterators are returned by the following member functions of a container:
begin() -- Returns an iterator to the **first element** of a container.
end() -- Returns an iterator **past the last element** of a container.

Iterator Categories
STL in C++ provides the following five iterators:

Input Iterator
- Input iterator is the weakest and simplest of all.
- It has limited functionality and is suitable for single-pass algorithms where you can access the elements in a container sequentially.
- It is a **unidirectional iterator** that only supports increment operation. Thus, it is a **read-only** iterator.

Output Iterator
- It is similar to the input iterator but works exactly the opposite.
- It is used for assignment operations that help you modify the values inside a container.
- It does not allow you to read elements from a container, i.e., it is a **write-only** iterator.

Forward Iterator

- Forward iterator is more powerful than input and output iterators.
- It combines the functionality of both the iterators (input and output).
- It permits you to **access (read)** and **modify (write)** the values of a container.
- It is also a unidirectional iterator, as the name suggests.

Bidirectional Iterator

- Bidirectional iterator has all the features of the forward iterator.
- It can move in both directions, i.e., backward and forward.
- So, you can use both **increment** (++) and **decrement** (--) operations.
- It does not permit the use of all relational operators.

Random Access Iterator

- It is the most powerful iterator.
- You can use it to access any random element inside a container.
- It supports all the functionality of the pointers, like addition and subtraction.
- It also gives you relational operator support.

Generic Programming
Standard Template Library (STL)

Operation Supported by Iterators

Iterator	Access	Read	Write	Iterate	Compare
Input	->	= *it		++	==, !=
Output			*it =	++	
Forward	->	= *it	*it =	++	==, !=
Bidirectional	->	= *it	*it =	++, --	==, !=
Random-Access	->, []	= *it	*it =	++, --, +=, -=, +, -	==, !=, <, >, <=, >=

Generic Programming

Standard Template Library (STL)

Algorithms

- Algorithms in C++ STL define a collection of functions specially designed for use on a sequence of objects.
- These are standalone template functions, unlike member functions of a container.
- There are approximately 60 algorithm functions that help save our time and effort.
- To use algorithms, you must include the <algorithm> header file.
- They also allow you to work with multiple container types simultaneously.

Generic Programming

Standard Template Library (STL)

STL in C++ algorithms can be broadly divided into 5 types:

Non-Manipulative Algorithms

- These are the non-modifying algorithms.
- They neither alter the containers' content nor change their order.
- They utilize forward iterators internally.
- Some Examples: **std::find** -- Find value in a range **std::search** -- Search a range for subsequence **std::count** -- Count occurrences of an element in a given range

Manipulative Algorithms

- These are the modifying algorithms.
- They alter the contents of a container either by changing their values or modifying their order.
- Some Examples: **std::copy** -- Copy a range of elements **std::swap** -- Exchange values of two objects/variables **std::fill** -- Fills a range with a certain value

Generic Programming

Standard Template Library (STL)

Sorting Algorithms

- These are modifying algorithms that are used to sort elements of a container.
- Some Examples: **std::sort** -- Sort elements in a range **std::stable_sort** -- Sort elements in a range and keep the relative order of equivalent elements **std::partial_sort** -- Partially sort elements in a range

Set Algorithms

- These can improve the efficiency of a C++ program.
- These are generic set algorithms that are independent of the set container. They can also be used on any sorted STL container.
- Therefore, they are also called sorted range algorithms.

Some examples: **std::merge** -- Merge the sorted ranges **std::includes** -- Checks whether the sorted range includes another sorted range **std::set_union** -- Computes the set union of two sorted ranges

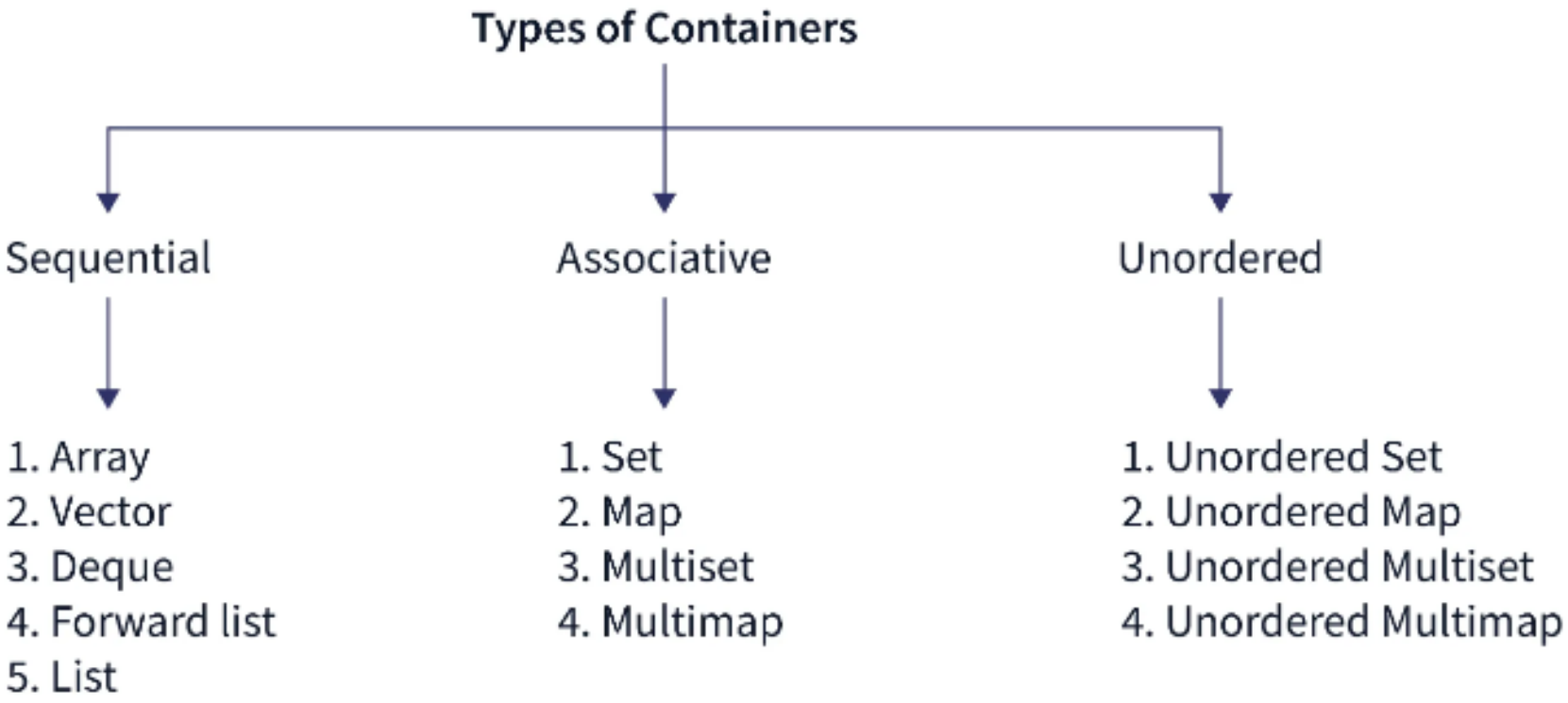
Relational Algorithms

- These are used to operate on numerical data.
- They perform some mathematical operation on all elements of a container.

Some Examples: **std::equal** -- Determines if two ranges of elements are equal **std::lexicographical_compare** -- Checks if the range is lexicographically less than another range of elements.

Generic Programming

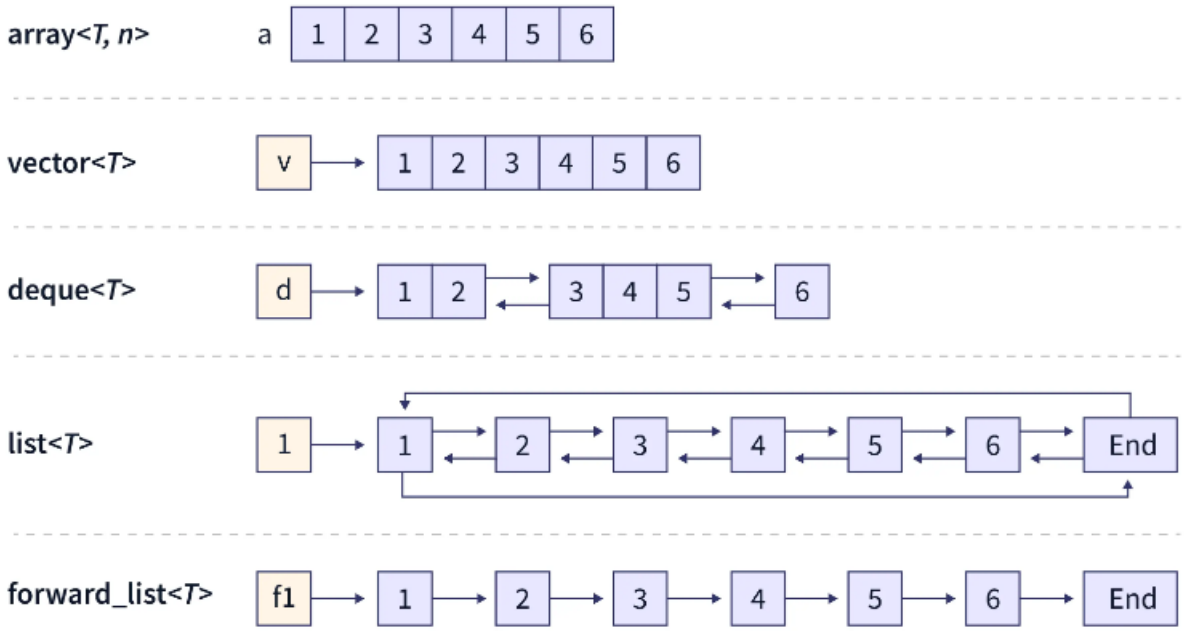
Container types and sequences



Generic Programming

Sequential containers

A Sequential Container in C++ is an ordered collection of the same type of data in which each element is stored in a specific position. The position of an element depends on the place and time of insertion of that element.
Sequential containers are also called sequence containers.
There are five types of sequential containers in the Standard Template Library (STL):



Generic Programming
Sequential containers

Arrays:
Arrays are containers of fixed size. They hold a specific number of elements. These elements are arranged in a strict linear sequence. We can directly use array elements by specifying their index i.e. we do not need to traverse the complete array.

Example: stl_array.cpp

We used the array standard library in the above program to create an array arr. Then, we used different predefined functions of the *array* library. The functions begin() and end() return pointers pointing to the array's first and last elements, respectively. The function size() returns the size of the array.

Generic Programming
Sequential containers

Vectors

Vectors are those array containers that can change their size. Like arrays, vectors use contiguous memory locations to store their elements. However, unlike arrays, the size of a vector can change dynamically, with their storage being handled by the container automatically.

Example: stl_vector.cpp

In the above program, we created a vector vec. We added elements to this vector using the push_back() function. Then, we used the begin() and end() , and the rbegin() and rend() functions to print the vector in left-to-right and right-to-left direction respectively.

Generic Programming
Sequential containers

Deque:
Deque stands for **d**ouble **e**nded **que**ue. A deque is a container whose size can increase or decrease dynamically on both ends. Deque allows us to access its individual elements through random access iterators (generalized pointers that can read, write, and allow random access to containers).

Example: stl_deque.cpp

In the above example, we created a deque dq. We added elements to the back and front of the deque using
the push_back() and push_front() functions, respectively. Then, we used the size() function to get the size of the deque. Finally, we used the pop_front() function to remove the first element of the deque.

Forward_list:
Forward lists are sequence containers that allow insert and erase operations in constant time, anywhere in the sequence. Forward lists are implemented as singly-linked lists. Hence, each element in a forward list can be stored in unrelated memory locations.

Example: stl_flist.cpp

In this example, we created a forward list fr_lst and added five elements to it. Then, we used the push_front() function to add two more elements to the list. Finally, we removed the first element from the list using the pop_front() function.

Generic Programming
Sequential containers

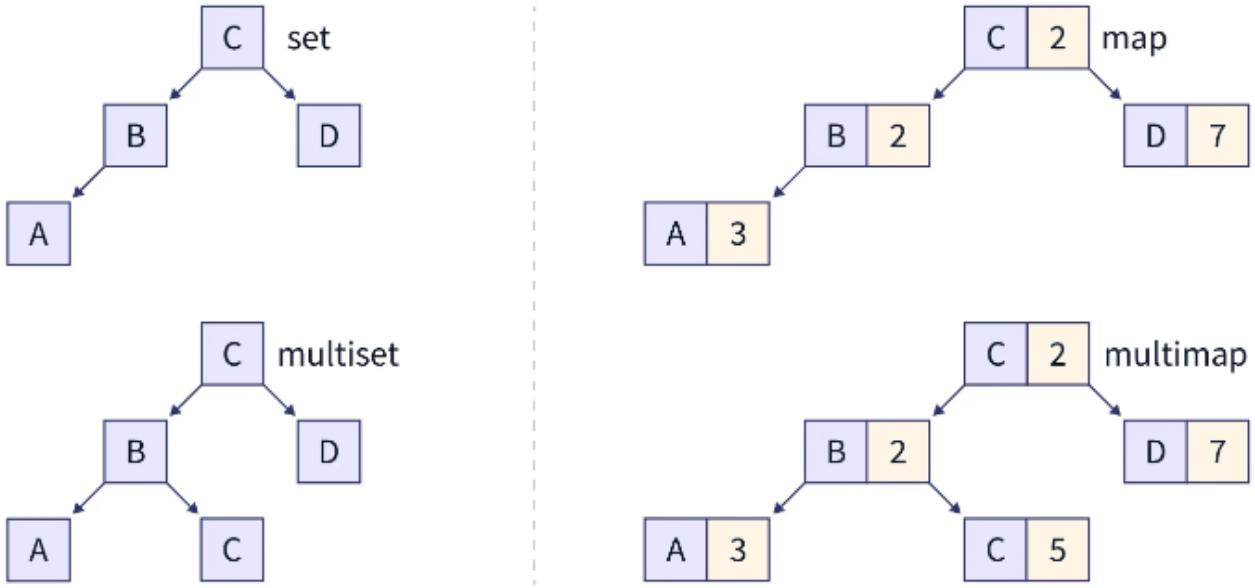
Lists:
Like forward lists, lists are sequence containers that allow insert and erase operations in constant time, anywhere in a sequence. Lists are implemented as doubly-linked lists. Hence, iteration is possible in both directions (front and back).

Example: stl_list.cpp

In the above example, we created a list lst and added elements using the push_back() function. Then, we printed the first and the last elements of the list using the front() and back() functions. Finally, we used the reverse() function to reverse the list.

Generic Programming
Associative containers

Associative containers are those containers in which the location of an element depends on the element's value. The order of insertion of elements is not considered in determining the position of the element. The elements of an associative container are accessed via keys, also called **search keys**.
There are four types of associative containers in the STL:



Generic Programming
Associative containers

Set:
A set is a container in which each element (or key) must be unique. Once an element is inserted into a set, it can't be modified. However, the element can be removed from the set. The elements in a set are stored in a sorted way so that the storage and retrieval of the elements are quick.

For example: stl_set.cpp

In this example, we tried to insert the number 50 two times in the set. However, since each element in a set must be unique, 50 was added only once to the set. The erase() function was then used to remove an element from the set.

Multiset:
Multisets are containers similar to sets but they allow duplicate elements (keys). Once stored, the value of an element can't be modified in a multiset, but the element can be removed from the container.

Example: stl_multiset.cpp

In the above program, we were able to add element 50 twice in the multiset because multisets allow the duplication of elements. We also used the *erase()* function to remove all the elements in the multiset that was smaller than the number 40. Had 40 not been in the set, the erase function would have deleted the whole multiset.

Generic Programming
Associative containers

Map:
A map is a container that stores data in key-value pairs. The keys in a map container must be unique, and there must be only one value associated with a particular key. The data type of the key and its associated value may be different. The elements in a map are stored with respect to the key. We can add or remove elements from any position in a map container.

For example: stl_map.cpp

In the above example, we created a map and added elements to it using the insert() function. Then, we used the erase() function to remove the element with key = 4.

Generic Programming
Associative containers

Multimap:
Multimap containers are similar to map containers, but multimap containers allow duplicate keys to be stored in a container. Hence, the relationship between the key-value pairs is one to many. In multimaps, the datatype of the key can be different from its associated value as well.

For example: stl_multimap.cpp

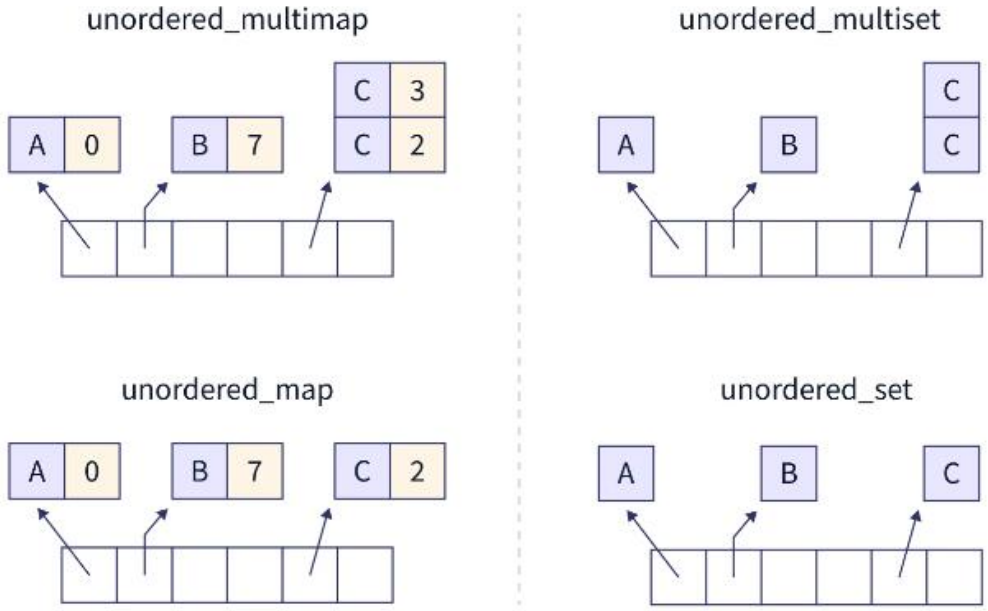
In this example, we created a multimap. Because multimap supports duplicate key values, we were able to insert the same key twice in the multimap. We also used the function erase() to remove all the elements with the key values less than 30.

Generic Programming
Associative containers

Unordered (Associative) Containers

Unordered containers or unordered associative containers are those containers in which the position of elements does not matter. The elements in unordered containers are not stored according to the order of their insertion or their values. If *n* elements are stored in an unordered container, the order of their positions will be undefined, and it might even change over time. The elements of these containers can be accessed using hash.

There are four types of unordered containers in C++ STL:



Generic Programming
Associative containers

Unordered_set:
Unordered sets are containers used to store unique elements in no particular order. The value of an element in an unordered set is it's key itself. The elements of an unordered set can not be modified, however, they can be inserted and removed.

For example: stl_unordered_set.cpp

In the above example, we inserted elements to the unordered set using the insert() function. We used the find() function to check whether the elements 's' and 'o' were present in the unordered set or not. From this example, we can observe that the order in which the elements were printed is different from the order of insertion of the elements. It means that the elements in unordered sets are not stored in a particular sequence.

Unordered_multiset:
Unordered multisets are similar to unordered sets, the only difference being, unordered multisets allow duplicate elements. Like in unordered sets, the elements of an unordered multiset can not be modified. They can only be inserted or removed.

For example: stl_unordered_mset.cpp

In this program, we used the *size()* function to get the size of the unordered multiset. Then, we used the *count()* function to find out how many times the element 5 appeared in the multiset. Finally, we used the *clear()* function to remove all the elements from the multiset.

Generic Programming
Associative containers

Unordered_map:
Unordered maps are those containers that store elements in key-value pairs in no particular order. In unordered maps, the key is used to uniquely identify the element. In unordered maps, each key must be unique. The data type of the key and the mapped value can be different.

For example: stl_unordered_map.cpp

In this example, we created an unordered map *uno_map* and added elements to it using square brackets (**[ ]**). Because unordered maps can only store unique keys, we were able to add the key *Mangoes* only once in the unordered map. When we try to re-insert the key in the unordered map it updates the new value for the same key.

Unordered_multimap:
Unordered multimaps are similar to unordered maps, the only difference being, in unordered multimaps, different elements can have the same key. The data type of the key and the mapped value(s) can be different.

For example stl_unordered_mmap.cpp

In the above example, we were able to add the key "*Apples*" twice in the unordered multimap because multimaps allow duplicate keys. We used the *find()* function to check whether the key "*Oranges*" was present in the multimap or not. Finally, we used the *clear()* function to remove all the elements having the key *Apples* from the unordered multimap.

Sequence vs Associative (complexity wise)

In the case of sequence containers:

1. Simple insertion of elements takes O(1) (constant time).
2. Insertion of elements in the middle is slower compared to insertion in associative containers.

In associative containers, most complexities are in logarithmic terms:

1. Inserting, removing, and searching for an element takes O(log n) time.
2. Incrementing or decrementing an iterator takes O(1) amortized time.
3. Insertion of an element in the middle is faster compared to insertion in sequence containers.

Generic Programming

STL Algorithms

STL algorithms are extremely useful because they reduce or eliminate the need to 'reinvent the wheel' when implementing common algorithmic functionality.

While some of the algorithms are designed to modify the individual values within *ranges* (certain STL containers or arrays of values), they do not modify the containers themselves - i.e. there is no reallocation or size modification to the containers.

Generic Programming
STL Algorithms

Algorithm Categories

To use these algorithms it is necessary to include the <algorithm> header file. For the numeric algorithms, it is necessary to include the <numeric> header file. As with containers and iterators, algorithms are categorised according to their behaviour and application:

Nonmodifying algorithms

Nonmodifying algorithms do not change the value of any element, nor do they modify the order in which the elements appear. They can be called for all of the standard STL container objects, as they make use of the simpler forward iterators.

Modifying algorithms

Modifying algorithms are designed to alter the value of elements within a container. This can either be done on the container directly or via a copy into another container range. Some algorithms classed as 'modifying' algorithms can change the order of elements (in particular copy_backward()), but most do not.

Removal algorithms

Removal algorithms are, by definition, modifying algorithms, but they are designed to remove elements in a container or when copying into another container.

Generic Programming

STL Algorithms

Mutating algorithms

Once again, mutating algorithms are modifying algorithms, but they are designed specifically to modify the order of elements (e.g. a random shuffle or rotation).

Sorting algorithms

Sorting algorithms are modifying algorithms specifically designed for efficient sorting of elements in a container (or into a range container).

Sorted range algorithms

Sorted range algorithms are special sorting algorithms designed to function on a container which is already sorted according to a particular sorting criterion. This allows for greater efficiency.

Numeric algorithms

Numeric algorithms are designed to work on numerical data in some fashion. The principal algorithm in this category is accumulate(), which allows mathematical operators to be applied to all elements in a container.

Nonmodifying Algorithms

for_each() - This is possibly the most important algorithm in this section, as it allows any unary function (i.e. a function of one argument) to be applied to each element in a range/container. Note that this function can actually also be modifying (hence why it is included below). It is often better to use a more specific algorithm, if one exists, than to use this, as specialist implementations will be more efficient. stl_for_each.cpp

count() - This returns the number of elements in a range or container. stl_count.cpp

count_if() - This counts how many elements in a range or container much a particular criterion. stl_count_if.cpp

min_element() - Returns the element that has the smallest value, making use of the < relation to perform comparison. It can accept a custom binary function to perform the comparison instead. stl_min_max.cpp

Generic Programming
STL Algorithms

max_element() - Returns the element that has the largest value, making use of the > relation to perform comparison. It can accept a custom binary function to perform the comparison instead.
find() - Finds the first element in a range or container that equals a passed value. stl_find.cpp
find_if() - Finds the first element in a range or container that matches a particular criterion, rather than a passed value. stl_find_if.cpp
search_n() - This is like find AND find_if except that it looks for the first occurances of such a value OR the first occurances where a relational predicate is met. stl_search_n.cpp

search() - This searches for the first occurance of a subrange within a range/container and can do so either by first/last value of the subrange or via a predicate matching all the values of the desired first/last subrange. stl_search.cpp

Generic Programming

STL Algorithms

find_end() - Similar to search, except that it finds the last occurance of such a subrange. stl_find_end.cpp

find_first_of() - Finds the first element in a subrange of a range or container. Can make use of a binary predicate function, otherwise uses direct value. stl_find_first_of.cpp

adjacent_find() - Returns the location (an iterator) to the first matching consecutive pair of values in a range/container. Can also match via a binary predicate. stl_adjacent_find.cpp

equal() - Compares two ranges to see if they are equal. stl_equal.cpp

mismatch() - Compares two ranges and returns a pair of iterators containing the points at which the ranges differ. stl_mismatch.cpp

lexicographical_compare() - The lexicographical comparison is used to sort elements in a manner similar to how words are ordered in a dictionary. It can either use operator< or make use of a binary predicate function to perform the comparison. stl_lexicograph.cpp

Generic Programming

STL Algorithms

Modifying Algorithms

for_each() - This is the same as the for_each we discussed above, but has been included in the Modifying section to reinforce that it can be used this way too! stl_for_each.cpp

copy() - Copies a range/container of elements into another range. stl_copy.cpp

copy_backward() - Copy a range/container of elements into another range, starting from the last element and working backwards. stl_copy_backward.cpp

transform() - Transform is quite a flexible algorithm. It works in two ways. A unary operation can be applied to the source range, on a per element basis, which ouputs the results in the destination range. A binary operation can be applied to both elements in the source and destination range, subsequently overwriting elements in the destination range. stl_transform.cpp

merge() - Merge is intended to take two sorted ranges and combine them to produce a merged sorted range. However, it is possible to utilize unsorted ranges as arguments but this then leads to an unsorted merge! For this reason, it has been included in the modifying category, rather than the category for sorted ranges. stl_merge.cpp

Generic Programming

STL Algorithms

swap_ranges() - This swaps the elements of two ranges. stl_swap_ranges.cpp

fill() - This replaces each element in a range with a specific value. stl_fill.cpp

fill_n() - Similar to fill, but replaces the first elements in a range with a specific value. stl_fill_n.cpp

generate() - This replaces each element in a range with the result of an operation of a generator function. stl_generate.cpp

generate_n() - Similar to generate, but replaces the first elements in a range with the result of an operation of a generator function. stl_generate_n.cpp

replace() - This replaces elements matching a specific value with another specific value. stl_replace.cpp

replace_if() - This replaces elements matching a specific criterion with another specific value. stl_replace_if.cpp

replace_copy() - Similar to replace, except that the result is copied into another range. stl_replace_copy.cpp

replace_copy_if() - Similar to replace_if, except that the result is copied into another range. stl_replace_copy_if.cpp

Removal Algorithms

remove() - This removes elements from a range that match a specific value. stl_remove.cpp

remove_if() - This removes elements from a range that match a specific criterion, as determined via a unary predicate. stl_remove_if.cpp

remove_copy() - Similar to remove, except that elements are copied into another range. stl_remove_copy.cpp

remove_copy_if() - Similar to remove_if, except that elements are copied into another range. stl_remove_copy_if.cpp

unique() - This is quite a useful algorithm. It removes adjacent duplicate elements, i.e. consecutive elements with specific values. stl_unique.cpp

unique_copy() - Similar to unique, except that it copies the elements into another range. stl_unique_copy.cpp

Generic Programming
STL Algorithms

Mutating Algorithms

reverse() - This simply reverses the order of the elements in a range or container. stl_reverse.cpp

reverse_copy() - Similar to reverse, except that the results of the reversal are copied into another range. stl_reverse_copy.cpp

rotate() - By choosing a 'middle' element in a range, this algorithm will cyclically rotate the elements such that the middle element becomes the first. stl_rotate.cpp

rotate_copy() - Similar to rotate, except that the result is copied into another range. stl_rotate_copy.cpp

next_permutation() - This rearranges the elements in a range to produce the next lexicographically higher permutation, using operator<. It is also possible to use a binary predicate comparison function instead of operator<. stl_next_permutation.cpp

prev_permutation() - Similar to next_permutation, except that it rearranges to produce the next lexicographically lower permutation.

stl_prev_permutation.cpp

random_shuffle() - Rearranges the list of elements in a range in a random fashion. The source of randomness can be supplied as a random number generator argument.

stl_random_shuffle.cpp

partition() - Rearranges a range/container such that the elements matching a predicate are at the front. Does NOT guarantee relative ordering from the original range.

stl_partition.cpp

stable_partition() - Similar to partition, except that it does guarantee relative ordering from the original range.

stl_stable_partition.cpp

Generic Programming
STL Algorithms

Sorting Algorithms

sort() - Sorts the elements into ascending order, using operator< or another supplied comparison function. stl_sort.cpp

stable_sort() - This is similar to sort. It is used when you need to ensure that elements remain in the same order when they are "tied" for the same position. This often comes up when dealing with priorities of tasks. Note also that the performance guarantee is different.

stl_stable_sort.cpp

partial_sort() - Similar to sort, except that it only sorts the first elements and terminates after they're sorted.

stl_partial_sort.cpp

partial_sort_copy() - Similar to partial_sort except that it copies the results into a new range. stl_partial_sort_copy.cpp

nth_element() - This allows you to ensure that an element at position is in the correct position, were the rest of the list to be sorted. It only guarantees that the elements preceeding are less than in value (in the sense of operator<) and that proceeding elements are greater than in value.

stl_nth_element.cpp

Generic Programming
STL Algorithms

partition() - See above for partition.
stable_partition() - See above for stable_partition.

make_heap() - Rearranges the elements in a range such that they form a heap, i.e. allowing fast retrieval of elements of the highest value and fast insertion of new elements. stl_make_heap.cpp

push_heap() - This adds an element to a heap.
stl_push_heap.cpp
pop_heap() - This removes an element from a heap.
stl_pop_heap.cpp
sort_heap() - This sorts the elements in a heap, with the caveat that the range is no longer a heap subsequent to the function call.
stl_sort_heap.cpp

Generic Programming
STL Algorithms

Sorted Range Algorithms

binary_search() - Searches the range for any matches of a specific value.
includes() - Determines whether each element in one range is also an element in another range. stl_binary_search.cpp

lower_bound() - Searches for the first element in a range which does not compare less than a specific value. Can also use a custom comparison function. stl_lower_bound.cpp

upper_bound() - Searches for the first element in a range which does not compare greater than a specific value. Can also use a custom comparison function. stl_upper_bound.cpp

equal_range() - Finds a subrange within a range which contains values equal to a specific value. stl_equal_range.cpp

merge() - See above for merge.

Generic Programming

STL Algorithms

set_union() - Creates a set union of two sorted ranges. Thus the destination range will include elements from either or both of the source ranges. Since this is a set operation, duplicates are eliminated.
stl_set_union.cpp

set_intersection() - Creates a set intersection of two sorted ranges. Thus the destination range will include elements that exist only in both of the source ranges. stl_set_intersection.cpp

set_difference() - Creates a set difference of two sorted ranges. Thus the destation range will include elements from the first range that are not in the second range. stl_set_difference.cpp

set_symmetric_difference() - This is the symmetric version of set_difference. It creates a set symmetric difference, which is formed by the elements that are found in one of the sets, but not in the other.
stl_set_symmetric_difference.cpp

inplace_merge() - This combines two sorted ranges into a destination range, which is also sorted. It is also stable as it will preserve ordering for subranges. stl_inplace_merge.cpp

Generic Programming
STL Algorithms

Numeric Algorithms

accumulate() - This is an extremely useful algorithm. It allows all elements of a container to be combined by a particular operation, such as via a sum, product etc. stl_accumulate.cpp

inner_product() - This combines all elements of two ranges by summing the multiples of each 'component'. It is possible to override both the 'sum' and 'multiply' binary operations. stl_inner_product.cpp

adjacent_difference() - Assigns every element the value of the difference between the current value and the prior value, except in the first instance where the current value is simply used.
stl_adjacent_difference.cpp

partial_sum() - Assigns the result of the partial sum of the current element and its predecessors, i.e.

$$y_i = \sum_{k=1}^{i} x_i,$$

for y the result and x the source. stl_partial_sum.cpp

Generic Programming

Functors

Functor, a short name for a **Function object**, is an instance of a certain class but can be called and executed like a normal function.

This can be achieved by overloading the function call operator (), and once overloaded, this allows us to call the objects of that specific class as if they were simply a function; meaning functors are objects in reality, but due to the overloading of the function call operator (), we can make them behave like ordinary functions, and can directly call them using the normal function call syntax, by passing an argument and later can receive the returned value after the successful execution of the call as well.

Generic Programming

Functors

Need of Functors Over Normal Functions
There are various reasons justifying why Functors should be preferred over normal Functions in their respective scenarios. For example

Separation of Logic - While solving a problem using the functor-based approach, the appropriate logic can be separated from the other logical computations happening in the code, allowing us to use the functors in multiple situations as well.

Parameterisation - Parameterising a functor is easier and more convenient than traditional functions.

Statefulness - This is probably the most important part of why we should prefer functors over normal functions. Normal functions are like **free functions**, and they cannot retain states between function calls, but talking about functors, they can have the states, meaning the data we are working with will be remembered and will be carried between subsequent calls to a function or class methods there.

Performance - Functors are most commonly used in STL implementations, and the concept of templates usually allows better optimization because more details are defined at compile time. As a result, passing functors or function objects in the program instead of ordinary functions often result in better performance.

Types of C++ Functors

There are generally **three types** of functors

Generator Functors - Functors can be called without arguments.

Unary Functors - Functors can be called with one argument.

Binary Functors - Functors that can be called with two arguments.

How to Create a C++ Functor?
We can easily create functors in C++. For this purpose, we have to create a class first. Then we have to overload the function call operator (), and after that, we can create an instance or an object of that specific class and call that a function.

Generic Programming

Functors

Example1: FunctorEx1.cpp

In the above example, we have seen how we can work with C++ functors, but one thing should be remembered that functors are not limited to printing string messages onto the console. We can do much more with C++ functors.

Example2: FunctorEx2.cpp

we are calculating the square of a number using C++ functors.

Example to Demonstrate State Retention of C++ Functors

Example3: FunctorEx3.cpp

C++ Functor with Return Type and Parameter

Example4: FunctorEx4.cpp

Generic Programming

Functors

C++ Functor with a Member and a Variable

Example5: FunctorEx5.cpp

Generic Programming

Functors in STL

The C++ language has a wide collection of predefined useful function objects or simply functors, defined inside the **functional** header file.

Example: stl_functor1.cpp

Generic Programming
Functors in STL

Predefined functors in STL

Arithmetic Functors

Functors	Description
plus	takes two parameters and returns their sum
minus	takes two parameters and returns their difference
multiplies	takes two parameters and returns their product
divides	takes two parameters and returns their division
modulus	takes two parameters and returns the remainder after division
negate	takes a single parameter and returns the negated value

Generic Programming

Functors in STL

Relational Functors

Functors	Description
equal_to	takes two parameters and returns true if both are equal
not_equal_to	takes two parameters and returns true if they are unequal
greater	takes two parameters and returns true if the first parameter is greater than the second parameter
greater_equal	takes two parameters and returns true if the first parameter is either greater than or equal to the second one
less	takes two parameters and returns true if the first parameter is less than the second parameter
less_equal	takes two parameters and returns true if the first parameter is either less than or is equal to the second one

Generic Programming

Functors in STL

Logical Functors

Functors	Description
logical_and	returns the result of the Logical AND operation of two booleans
logical_or	returns the result of the Logical OR operation of two booleans
logical_not	returns the result of the Logical NOT operation of a boolean

Generic Programming

Functors in STL

Bitwise Functors

Functors	Description
bit_and	performs the Bitwise AND operation on two parameters and returns the result
bit_or	performs the Bitwise OR operation on two parameters and returns the result
bit_xor	performs the Bitwise XOR operation on two parameters and returns the result

Generic Programming

STL functors

More Sample programs

Sample2: stl_functor2.cpp

Sample3: stl_functor3.cpp

Sample4: stl_functor4.cpp

Sample5: stl_functor5.cpp

Sample6: stl_functor6.cpp

Sample7: stl_functor7.cpp

Sample8: stl_functor8.cpp

THANK YOU

M S Anand
Department of Computer Science Engineering

anandms@pes.edu