Agenda

- Templates
- Shallow Copy and Deep Copy
- Copy Constructor
- STL

Document Link

```
https://en.cppreference.com/w/
```

Template

- If we want to write generic program in C++ then we should use template.
- Using template we can not reduce code size or execution time but we can reduce developers effort.
- It is designed for implementing generic data structure and algorithms
- Types of template:
    1. Function Template
    2. Class Template

1. Function Template

```
//template<typename T>//T : Type Parameter
template<class T> //T : Type Parameter
void swap_number( T &o1, T &o2 )
{
    T temp = o1;
    o1 = o2;
    o2 = temp;
}
int main( void )
{
    int num1 = 10;
    int num2 = 20;
    swap_number<int>( num1, num2 );
    //Here int is type argument
    cout<<"Num1 : "<<num1<<endl;
    cout<<"Num2 : "<<num2<<endl;
    return 0;
}
```

- Type inference : It is ability of compiler to detect type of argument at compile time and passing it as a argument to the function.

```
template<class X, class Y>
void swap_number( X &o1, Y &o2 )
{
    X temp = o1;
    o1 = o2;
    o2 = temp;
}
int main( void )
{
    float num1 = 10.5f;
    double num2 = 20.5;
    swap_number<float, double>(num1,num2 );
    cout<<"Num1 : "<<num1<<endl;
    cout<<"Num2 : "<<num2<<endl;
    return 0;
}
```

- We can pass multiple type arguments to the function.
- Using template argument list, we can pass data type as a argument to the function.
- Using template we can write type safe generic code.

2. Class Template

- In C++, by passing data type as a argument, we can write generic code hence parameterized type is called template.

```
template<class T>
class Array // Parameterized type
{
    private:
    int size;
    T *arr;
    public:
    Array( void ) : size( 0 ), arr( NULL )
    {
    }
    Array( int size )
    {
    this->size = size;
    this->arr = new T[ this->size ];
    }
    void acceptRecord( void ){
    }
    void printRecord( void ){
    }
    ~Array( void ){ }
};
int main( void )
{
    Array<char> a1( 3 );
```

```
    a1.acceptRecord();
    a1.printRecord();
    return 0;
}
```

- Operator overloading
- Conversion Function

Copy Constructor

- Copy constructor is a parametered constructor of the class which take single parameter of same type but using reference.
- Copy constructor gets called in following conditions:

```
class ClassName
{
public:
  //this : Address of dest object
  //other : Reference of src object
  ClassName( const ClassName &other )
  {
    //TODO : Shallow/Deep Copy
  }
};
```

1. If we pass object(of structure/class) as a argument to the function by value then on function parameter, copy constructor gets called.
2. If we return object from function by value then to store the result compiler implicitly create annonymoys object inside memory. On that annonymous object, copy constructor gets called.
3. If we try to initialize object from another object then on destination object, copy constructor gets called.
4. If we throw object then its copy gets created into stack frame. To create copy on stack frame, copy constructor gets called.
5. If we catch object by value then on catching object, copy constructor gets called.

- If we do not define copy constructor inside class then compiler generate copy constructor for the class. It is called, default copy constructor. By default it creates shallow copy.
- Job of constructor is to initialize object. Job of destructor is to release the resources. Job of copy constructor is to initialize newly created object from existing object.
- Note : Creating copy of object is expesive task hence we should avoid object copy operation. To avoid the copy, we should use reference.
- During initialization of object, if there is need to create deep copy then we should define user defined copy constructor inside class.

STL

- We can not divide template code into multiple files.

- Standard Template Library(STL) is a collection of readymade template data structure classes and algorithms.
- The Standard Template Library (STL) is a set of C++ template classes to provide common programming data structures and functions such as lists, stacks, arrays, etc.
- It is a library of container classes, algorithms, and iterators.
- It is a generalized library and so, its components are parameterized.
- Working knowledge of template classes is a prerequisite for working with STL.
- STL has 4 components:

    1. Containers
    2. Algorithms
    3. Function Objects
    4. Iterators

1. Container

- Containers or container classes to store objects and data.
- The C++ container library categorizes containers into four types:

1. Sequence containers
2. Associative containers
3. Unordered associative containers
4. Sequence container adapters

```
STL
│
├── Containers
   ├── Sequence Containers
   │   ├── vector
   │   ├── deque
   │   ├── list
   │   └── forward_list
   ├── Associative Containers
   │   ├── set
   │   ├── multiset
   │   ├── map
   │   └── multimap
   ├── Unordered Associative Containers
   │   ├── unordered_set
   │   ├── unordered_multiset
   │   ├── unordered_map
   │   └── unordered_multimap
   └── Container Adapters
       ├── stack
       ├── queue
       └── priority_queue
```

2. Algorithm

- They act on containers and provide means for various operations for the contents of the containers.
    - Sorting
    - Searching

3. Functions

- The STL includes classes that overload the function call operator.
- Instances of such classes are called function objects or functors.
- Functors allow the working of the associated function to be customized with the help of parameters to be passed.

4. Iterators

- As the name suggests, iterators are used for working upon a sequence of values.
- They are the major feature that allows generality in STL.
- Iterators are used to point at the memory addresses of STL containers.
- They are primarily used in sequences of numbers, characters etc.
- They reduce the complexity and execution time of the program.

Sequence Containers

- Sequence containers are used for data structures that store objects of the same type in a linear manner.
- The STL Sequence Container types are:
- vector: A dynamic array that can grow and shrink in size. It provides fast random access to elements and efficient insertion and deletion at the end.
- deque: A double-ended queue that supports efficient insertion and deletion at both ends. It provides similar functionality to vector but may have better performance for inserting and deleting elements at the beginning.
- list: A doubly-linked list that allows for efficient insertion and deletion of elements at any position. It does not provide random access to elements and has slower traversal compared to vector and deque.
- forward_list: A singly-linked list that provides similar functionality to list but with reduced memory overhead. It allows for efficient insertion and deletion at the beginning and after an element.

Associative Containers

- Associative containers implement sorted data structures that can be quickly searched.
- set : collection of unique keys, sorted by keys
- map : collection of key-value pairs, sorted by keys, keys are unique
- multiset : collection of keys, sorted by keys
- multimap : collection of key-value pairs, sorted by keys

Container adaptors

- Container adaptors provide a different interface for sequential containers.

- stack : adapts a container to provide stack (LIFO data structure)
- queue : adapts a container to provide queue (FIFO data structure)

Vector

- Vectors are the same as dynamic arrays with the ability to resize itself automatically when an element is inserted or deleted, with their storage being handled automatically by the container.
- Vector elements are placed in contiguous storage so that they can be accessed and traversed using iterators.
- In vectors, data is inserted at the end. Inserting at the end takes differential time, as sometimes the array may need to be extended. Removing the last element takes only constant time because no resizing happens.
- When the vector needs to grow beyond its current capacity, it typically doubles its capacity.
- Inserting and erasing at the beginning or in the middle is linear in time.
- the iterator in the vector is a Random Access Iterator that Supports all iterator operations including arithmetic (e.g., +, -), comparison (<, >, etc.), and dereferencing (*, []).