Templates, Exceptions and STL

Introduction to Exception

```
int main()
{
   int a,b,c;
   cout<<"Enter value a=";
   cin>>a;
   cout<<"Enter value b=";
   cin>>b;
   c=a/b;
   cout<<"answer="<<c;
}
```

Output:
Enter value a=5
Enter value b=2
answer=2

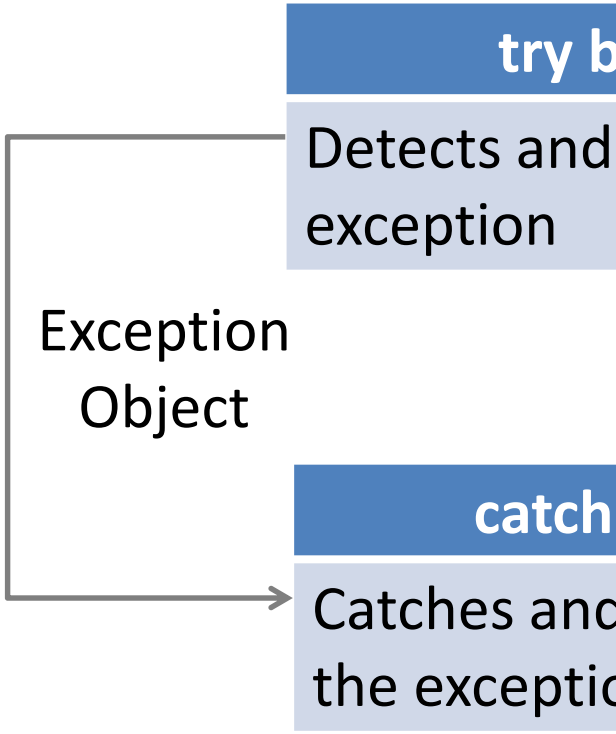
Output:
Enter value a=5
Enter value b=0
Abnormal Termination occur

Introduction to Exception(Cont…)

- Runtime errors are termed as **exception**.
- **Exception handling** is the process to manage the runtime errors by converting the abnormal termination of a program to normal termination of a program.

try, throw and catch

- C++ exception handling mechanism is built upon three keywords **try**, **throw** and **catch**.




```
try
{
   .....
   throw exception; //this block
   detects and throws an exception
}
```

```
catch(type arg)
{
   .....
   ... //exception handling block
}
.....
```

try, throw and catch example

```
int main()
{
   int a,b,c;
   cout<<"Enter two values=";
   cin>>a>>b;
   try
   {
      if(b!=0)
            c=a/b;
            cout<<"answer="<<c;
      else
            throw(b);
   }
   catch(int x)
   {
      cout<<"Exception caught: Divide by zero\n";
   }
}
```

Output:

Enter value a=5

Enter value b=0

Exception caught: Divide by zero

Multiple catch example

```
void test(int x){
   try
   {
      if(x==1)
        throw x;
      else if(x==0)
        throw 'x';
      else if(x==-1)
        throw 5.14;
   }
   catch(int i){
     cout<<"\nCaught an integer";
   }
   catch(char ch){
     cout<<"\nCaught a character";
   }
   catch(double i){
     cout<<"\nCaught a double";
   }
}
```

```
int main()
{
   test(1);
   test(0);
   test(-1);
}
```

Output:

Caught an integer

Caught a character

Caught a double

Catch all Exception

Catch all exception

- In some situations, we may not predict all possible types of exceptions and therefore may not be able to design independent catch handlers to catch them.

```
Syntax:
catch(...)
{
      //statements for processing all exceptions
}
```

Catch all exception example

```
#include<iostream>
using namespace std;
void test(int x)
{
   try
   {
      if(x==0) throw x;
      if(x==-1) throw 'a';
      if(x==1) throw 5.15;
   }
   catch(...)
   {
   cout<<"Caught an exception\n";
   }
}
```

```
int main()
{
   test(-1);
   test(0);
   test(1);
}
```

Output:

Caught an exception

Caught an exception

Caught an exception

Re-Throwing exception

- An exception is thrown from the catch block is known as the **re-throwing** exception.
- It can be simply invoked by **throw** without arguments.
- Rethrown exception will be caught by **newly defined catch statement**.

```
void divide(double x, double y){
  try
  {
   if(y==0)
     throw y;
   else
     cout<<"Division="<<x/y;
  }
  catch(double)
  {
   cout<<"Exception inside
function\n";
   throw;
  }
}
```

```
int main()
{
  try
  {
   divide(10.5,2.0);
   divide(20.0,0.0);
  }
  catch(double)
  {
   cout<<"Exception inside
main function";
  }
}
```

Output:

Division=5.25

Exception inside function

Exception inside main function

Exceptions thrown from functions

```
#include <iostream>
using namespace std;
void test(int x)
{
cout<<"Inside function:"<<x<<endl;
if(x) throw x;
}
int main()
{
cout<<"Start"<<endl;
try
{
test(0);
test(1);
test(2);
}
catch(int x)
{
cout<<"Caught an int exception:"<< x<<endl;
}

}
```

User defined Exception

- There maybe situations where you want to generate some **user specific exceptions** which are not pre-defined in C++.
- In such cases C++ provided the mechanism to create our own exceptions by inheriting the exception class in C++.

User defined Exception

```
#include <iostream>
#include <exception>
class myexception: public exception
{
  virtual const char* what() const throw()
  {
    return "My exception happened";
  }
} myex;
int main (){
  try
  {
    throw myex;
  }
  catch (exception& e)
  {
    cout << e.what() << '\n';
  }
}
```

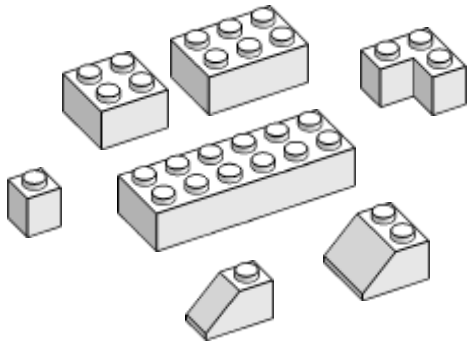
User defined Exception(Cont…)

```
double myfunction (char arg) throw (int);
```

- This declares a function called **myfunction**, which takes one argument of type **char** and returns a value of type **double**.
- If this function throws an exception of some type other than int, the function calls **std::unexpected** instead of looking for a handler or calling **std::terminate.**
- If this throw specifier is left empty with no type, this means that **std::unexpected** is called for any exception.
- Functions with no throw specifier (regular functions) never call **std::unexpected**, but follow the normal path of looking for their exception handler.

```
int myfunction (int param) throw(); //all exceptions call unexpected
int myfunction (int param);         //normal exception handling
```

Template



Need of Templates

```
int add(int x, int y)
{
  return x+y;
}
```

```
float add(float x, float y)
{
   return x+y;
}
```

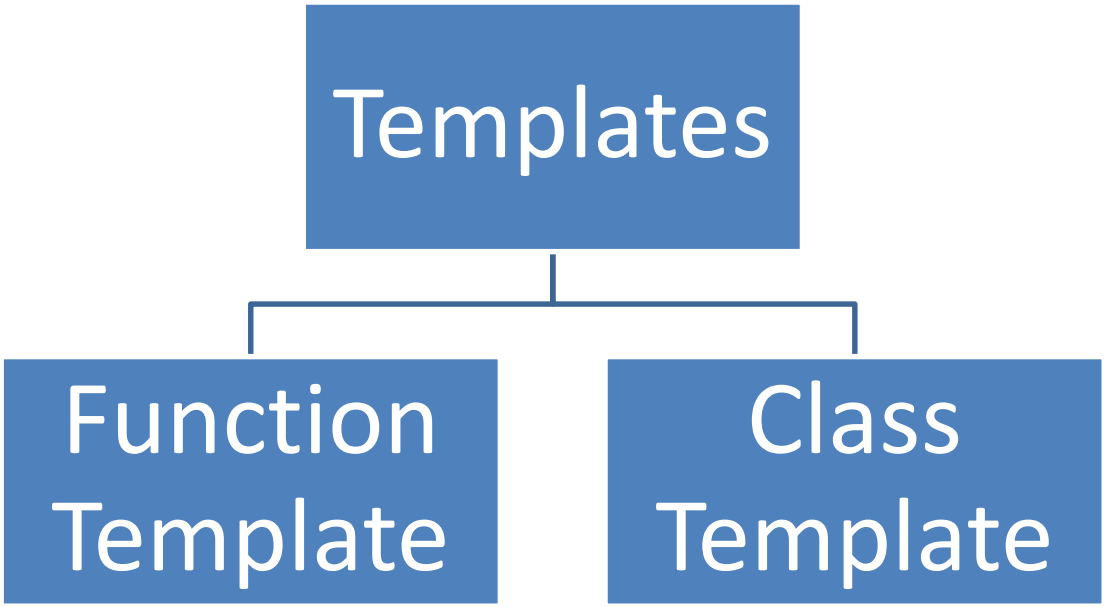
```
char add(char x,char y)
{
      return x+y;
}
```

```
double add(double x, double y)
{
  return x+y;
}
```

We need a single function that will work for int, float, double etc…

Templates

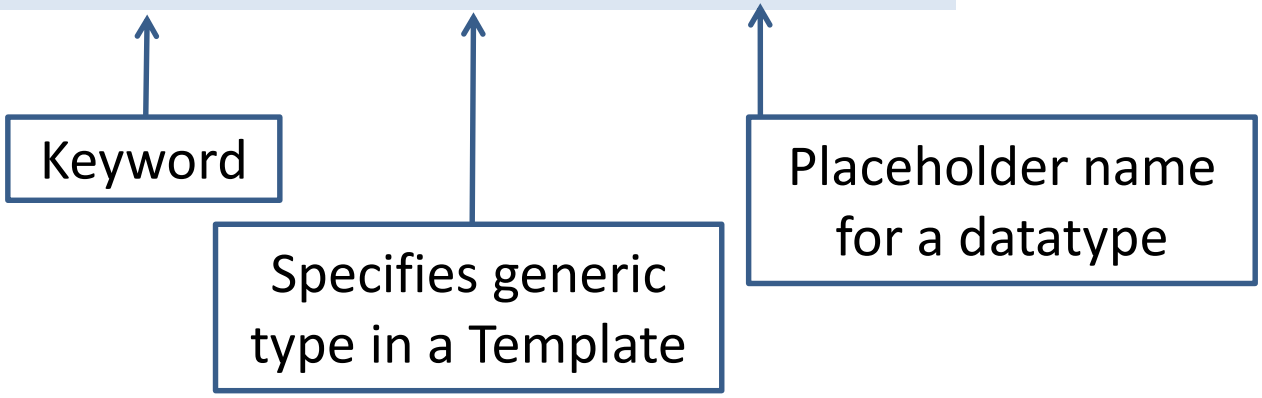
- Templates concept enables us to define **generic classes** and **functions**.
- This allows a function or class to work on many different data types without being rewritten for each one.



Function Template

Syntax:

```
template<class Ttype>
```




```
template<typename Ttype>
```

Templates

- C++ **templates** are a powerful mechanism for code reuse, as they enable the programmer to write code that behaves the same for any data type.
- By **template** we can define generic classes and functions.
- In simple terms, you can create a single function or a class to work with different data types using **templates**.
- It can be considered as a kind of macro. When an object of a specific type is defined for actual use, the template definition for that class is substituted with the required data type.

Function Template

- Suppose you write a function printData:

```
void printData(int value){
      cout<<"The value is "<<value;
}
```

- Now if you want to print double values or string values, then you have to overload the function:

```
void printData(float value){
      cout<<"The value is "<<value;
}
void printData(char *value) {
      cout<<"The value is "<<*value;
}
```

- To perform same operation with different data type, we have to write same code multiple time.

Function Template (Cont…)

- C++ provides templates to reduce this type of duplication of code.

```
template<typename T>
void printData(T value){
      cout<<"The value is "<<value;
}
```

- We can now use `printData` for any data type. Here **`T`** is a template parameter that identifies a type.
- Then, anywhere in the function where **`T`** appears, it is replaced with whatever type the function is instantiated.

```
int i=3;
float d=4.75;
char *s="hello";
printData(i); // T is int
printData(d); // T is float
printData(s); // T is string
```

```
#include <iostream>
using namespace std;
template <typename T>
T Large(T n1, T n2)
{
 return (n1 > n2) ? n1 : n2;
}
int main(){
   int i1, i2; float f1, f2; char c1, c2;
   cout << "Enter two integers:\n";
   cin >> i1 >> i2;
   cout << Large(i1, i2) <<" is larger." << endl;
   cout << "\nEnter two floating-point numbers:\n";
   cin >> f1 >> f2;
   cout << Large(f1, f2) <<" is larger." << endl;
   cout << "\nEnter two characters:\n";
   cin >> c1 >> c2;
   cout << Large(c1, c2) << " has larger ASCII value.";
}
```

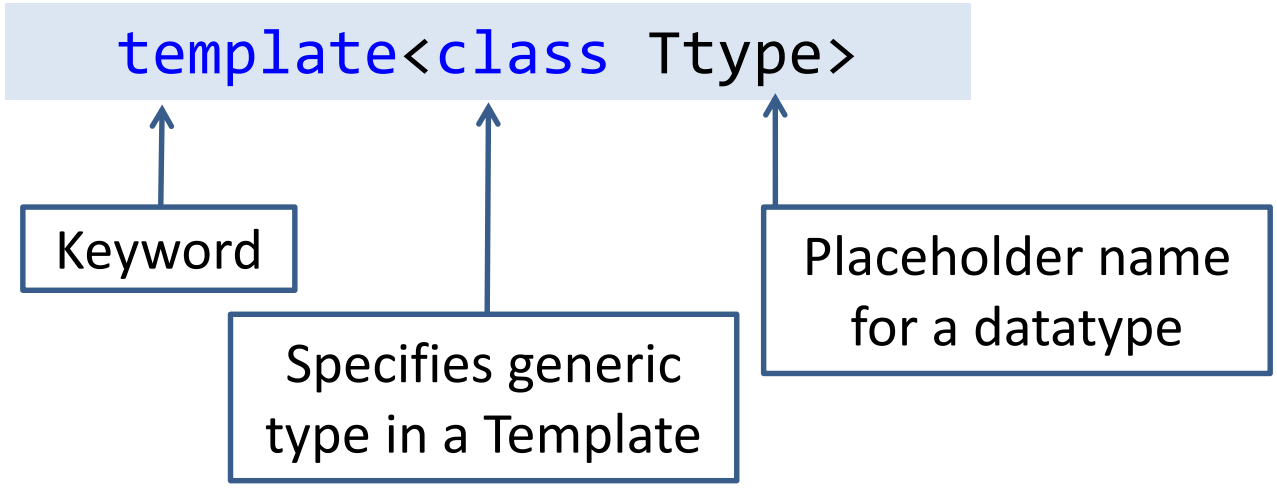
- T is a **template** argument that accepts different data types
- **typename** is a keyword
- You can also use keyword **class** instead of **typename**

Class Template

- Sometimes, you need a class implementation that is same for all classes, only the data types used are different.
- Normally, you would need to create a different class for each data type OR create different member variables and functions within a single class.

Class Template

Syntax:



Object of template class

The object of template class are created as follows

```
class name <data type> object name;
```

```
template<class Ttype>
class sample
{
   Ttype a,b;
   public:
      void getdata()
      {
        cin>>a>>b;
      }
      void sum();
};
```

```
int main()
{
  sample <int>s1;
  sample <float>s2;
  s1.getdata();
  s1.sum();
  s2.getdata();
  s2.sum();
}
```

Class Template Example

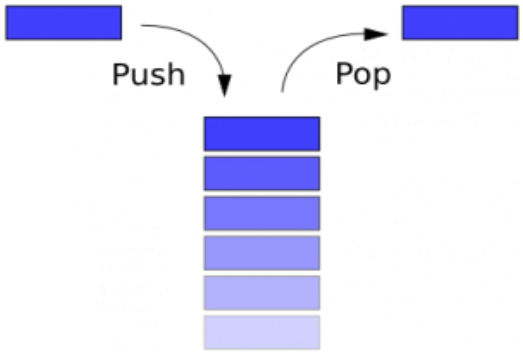
```
template<class T1, class T2>
class Sample
{
   T1 a; T2 b;
   public:
   Sample(T1 x,T2 y){
      a=x;
      b=y;
   }
   void disp(){
      cout<<"\na="<<a<<"\tb="<<b;
   }
};
int main(){
 Sample <int,float> S1(12,23.3);
 Sample <char,int> S2('N',12);
 S1.disp();
 S2.disp();
}
```

- To create a class template object, define the data type inside a **< >** at the time of object creation.
- className<int> classObj;
  className<float> classObj;

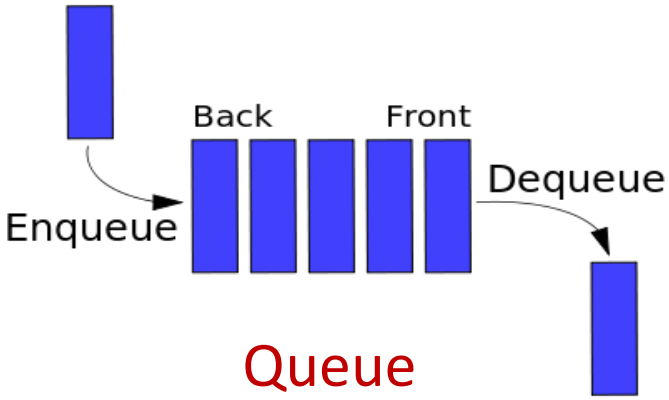
Programs

1. Write a function template for finding the minimum value contained in an array.
2. Create a generic class stack using template and implement common Push and Pop operations for different data types.
3. Write program to swap Number using Function Template.

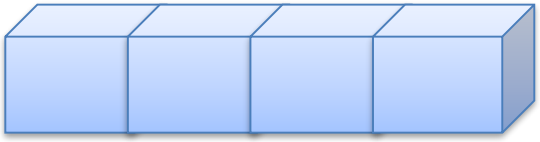
STL – Standard Template Library



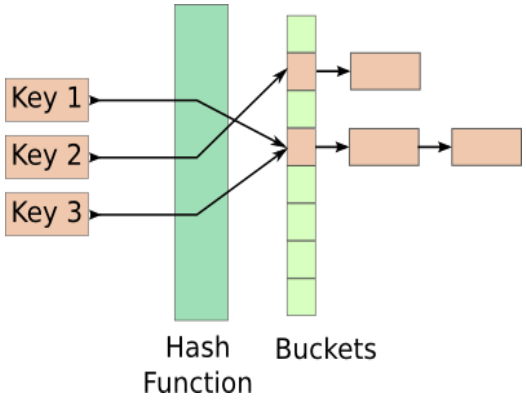
Stack



Queue



Vector



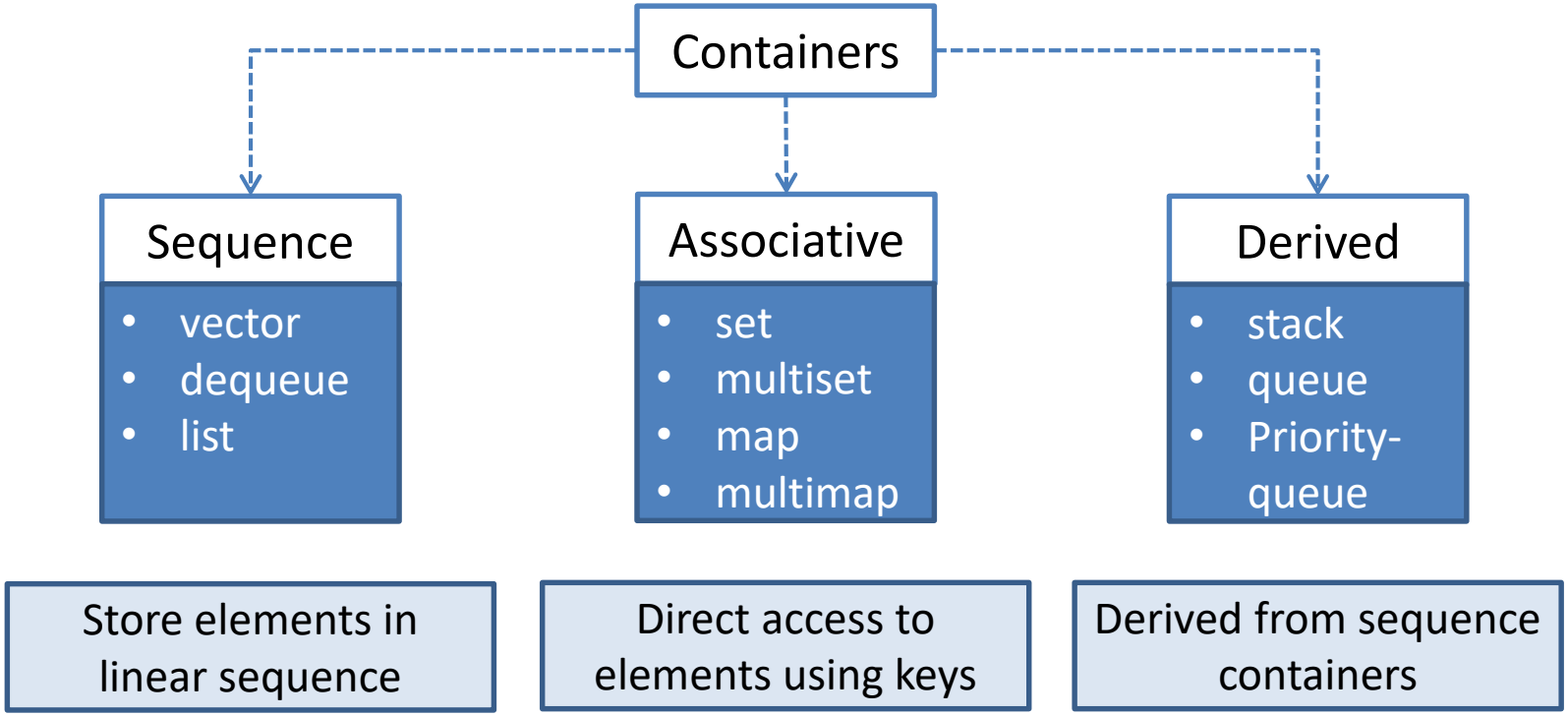
map

STL- Standard Template Library

- The C++ **STL** (Standard Template Library) is a powerful set of C++ template classes to provides general-purpose templatized classes and functions that implement many popular and commonly used algorithms and data structures like vectors, lists, queues, and stacks.
- There are three core components of STL as follows:
  1. Containers (an object to store data)
  2. Algorithms (procedure to process data)
  3. Iterators (pointer object to point elements in container)

STL- Containers

- A **container** is an object the actually stores data.
- The STL containers can be implemented by **class templates** to hold different data types.



STL Algorithms

- It is a procedure that is **used to process data** contained in containers.
- It includes algorithms that are used for initializing, searching, copying, sorting and merging.

- Mutating Sequence Algorithms

  like copy(), remove(), replace(), fill(), swap(), etc.,
- Non Modifying sequence Algorithms

  like find(), count(),search(), mismatch(), and equal()
- Numerical Algorithms

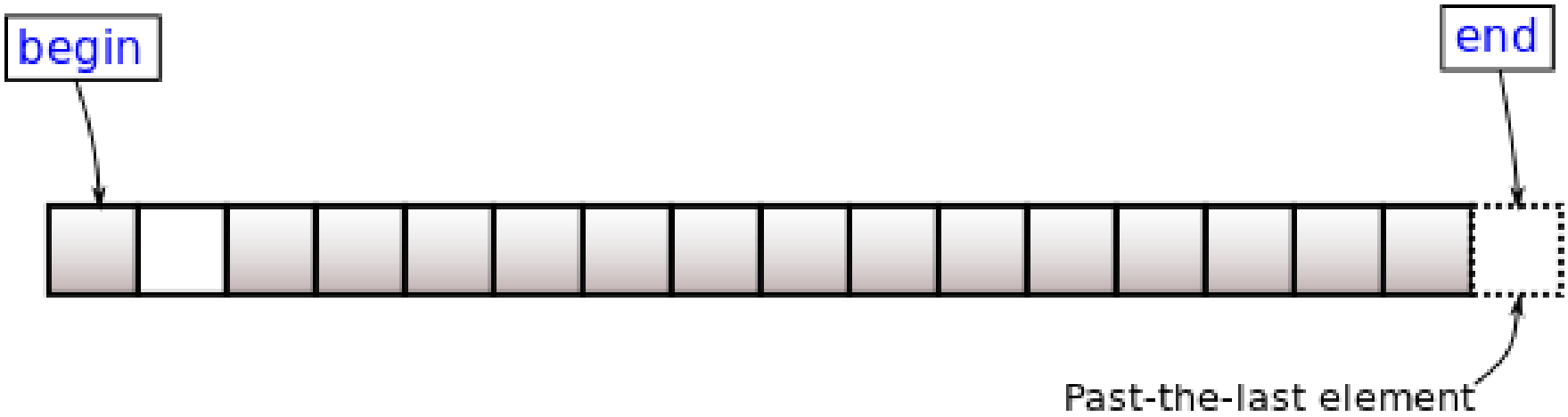
  accumulate(), partial_sum(), inner_product(), and adjacent_difference()

STL- Algorithms

- STL provide number of algorithms that can be used on any container, irrespective of their type. Algorithms library contains built in functions that performs complex algorithms on the data structures.
- For example: one can reverse a range with reverse() function, sort a range with sort() function, search in a range with binary_search() and so on.
- Algorithm library provides abstraction, i.e you don't necessarily need to know how the algorithm works.

STL- Iterations

- Iterators behave like **pointers**.
- Iterators are used to **access container elements**.
- They are used to traverse from one element to another.



STL components

- STL provides numerous **containers** and **algorithms** which are very useful in completive programming , for example you can very easily define a linked list in a single statement by using list container of container library in STL , saving your time and effort.
- STL is a generic library , i.e a same **container** or **algorithm** can be operated on any data types , you don’t have to define the same algorithm for different type of elements.
- For example , sort algorithm will sort the elements in the given range irrespective of their data type , we don’t have to implement different sort algorithm for different datatypes.

Thank You