CSE107
Object Oriented Programming Language

Weeks 6 to 9

Khaled Mahmud Shahriar
Assistant Professor, CSE, BUET
khaledshahriar@cse.buet.ac.bd

Course Content

Source: CSE_BUET_UG_Course_Calendar_2022.pdf

- Philosophy of object oriented programming (OOP); Basic principles of OOP: abstraction, encapsulation, polymorphism, inheritance; Advantages of OOP over structured programming;

- C++: Classes and objects: specifying a class, access specifiers; Functions: inline functions, friend functions; Constructors and destructors; Operator overloading and type conversions; Inheritance: single inheritance, multilevel inheritance, multiple inheritance; Polymorphism: function overloading, virtual functions, pure virtual functions; Templates: class templates, function templates, introduction to the standard template library (STL); Exception Handling

- Java: Nested and Inner classes; Local variable type inference; Strings: String, StringBuffer, StringBuilder; Inheritance: abstract class and anonymous subclasses, object class; Access protection with package; Interface; Exception; Thread: multithreading, Introduction to Java concurrency utilities; Generics and collections; Stream API and lambda expressions; Networking: ServerSocket, Socket

Templates

Templates

- Template is one of C++'s most sophisticated and high-powered features.
- Using templates, it is possible to create generic functions and classes.
- In a generic function or class, the type of data upon which the function or class operates is specified as a parameter.
  - It is possible to use one function or class with several different types of data without having to explicitly recode specific versions for each data type.

Generic Functions

```
int main()
{
   int i = 10, j = 20;
   double x = 10.1, y = 23.3;
   char a = 'x', b = 'z';
   cout << "Original i, j: " << i << ' ' << j << '\n';
   cout << "Original x, y: " << x << ' ' << y << '\n';
   cout << "Original a, b: " << a << ' ' << b << '\n';
   swapargs(i, j); // swap integers
   swapargs(x, y); // swap floats
   swapargs(a, b); // swap chars
   cout << "Swapped i, j: " << i << ' ' << j << '\n';
   cout << "Swapped x, y: " << x << ' ' << y << '\n';
   cout << "Swapped a, b: " << a << ' ' << b << '\n';
   return 0;
}
```

```
// This is a function template
template <class X> void swapargs(X &a, X &b)
{
   X temp;
   temp = a;
   a = b;
   b = temp;
}
```

Generic Functions (2)

- A generic function defines a general set of operations that will be applied to various types of data.
- The type of data that the function will operate upon is passed to it as a parameter.
- Through a generic function, a single general procedure can be applied to a wide range of data.
- Applicable for many algorithms are logically the same no matter what type of data is being operated upon.
  - For example, sorting algorithm
- In essence, a generic function is a function that can automatically overload itself.

Generic Functions (3)

- A generic function is created using the keyword template.
- The normal meaning of the word "template" accurately reflects its use in C++.
- It is used to create a template (or framework) that describes what a function will do, leaving it to the compiler to fill in the details as needed.
- The general form of a template function definition is shown here:

```
template <class Ttype> ret-type func-name (parameter list)
{
   // body of function
}
```

```
template <class Ttype>
ret-type func-name (parameter list)
{
   // body of function
}
```

Functions with Multiple Generic Types

```
#include <iostream>
using namespace std;
template <class type1, class type2>
void myfunc(type1 x, type2 y)
{
   cout << x << ' ' << y << '\n';
}
```

```
int main()
{
   myfunc(10, "I like C++");
   myfunc(98.6, 19L);
   return 0;
}
```

Explicit Overloading of Generic Functions

```
template <class X>
void swapargs(X &a, X &b)
{
   X temp;
   temp = a;
   a = b;
   b = temp;
   cout << "Inside template swapargs.\n";
}
```

```
void swapargs(int &a, int &b)
{
   int temp;
   temp = a;
   a = b;
   b = temp;
   cout << "Inside swapargs int specialization.\n";
}
```

```
int main()
{
   int i = 10, j = 20;
   double x = 10.1, y = 23.3;
   char a = 'x', b = 'z';
   cout << "Original i, j: " << i << ' ' << j << '\n';
   cout << "Original x, y: " << x << ' ' << y << '\n';
   cout << "Original a, b: " << a << ' ' << b << '\n';
   swapargs(i, j); // calls explicitly overloaded swapargs()
   swapargs(x, y); // calls generic swapargs()
   swapargs(a, b); // calls generic swapargs()
   cout << "Swapped i, j: " << i << ' ' << j << '\n';
   cout << "Swapped x, y: " << x << ' ' << y << '\n';
   cout << "Swapped a, b: " << a << ' ' << b << '\n';
   return 0;
}
```

Specialization Syntax for Overloading Generic Functions

```
template <class X>
void swapargs(X &a, X &b)
{
   X temp;
   temp = a;
   a = b;
   b = temp;
   cout << "Inside template swapargs.\n";
}
```

```
// Use new-style specialization syntax.
template <>
void swapargs<int>(int &a, int &b)
{
   int temp;
   temp = a;
   a = b;
   b = temp;
   cout << "Inside swapargs int specialization.\n";
}
```

```
int main()
{
   int i = 10, j = 20;
   double x = 10.1, y = 23.3;
   char a = 'x', b = 'z';
   cout << "Original i, j: " << i << ' ' << j << '\n';
   cout << "Original x, y: " << x << ' ' << y << '\n';
   cout << "Original a, b: " << a << ' ' << b << '\n';
   swapargs(i, j); // calls explicitly overloaded swapargs()
   swapargs(x, y); // calls generic swapargs()
   swapargs(a, b); // calls generic swapargs()
   cout << "Swapped i, j: " << i << ' ' << j << '\n';
   cout << "Swapped x, y: " << x << ' ' << y << '\n';
   cout << "Swapped a, b: " << a << ' ' << b << '\n';
   return 0;
}
```

Overloading Function Templates

```
// First version of f() template.
template <class X>
void f(X a)
{
   cout << "Inside f(X a)\n";
}
```

```
// Second version of f() template.
template <class X, class Y>
void f(X a, Y b)
{
   cout << "Inside f(X a, Y b)\n";
}
```

```
int main()
{
   f(10); // calls f(X)
   f(10, 20); // calls f(X, Y)
   return 0;
}
```

Standard Parameters with Template Functions

```
const int TABWIDTH = 8;
// Display data at specified tab position.
template<class X>
void tabOut(X data, int tab)
{
   for (; tab; tab--)
      for (int i = 0; i < TABWIDTH; i++)
         cout << ' ';
   cout << data << "\n";
}
```

```
int main()
{
   tabOut("This is a test", 0);
   tabOut(100, 1);
   tabOut('X', 2);
   tabOut(10 / 3, 3);
   return 0;
}
```

Generic Function Restrictions

- Generic functions are similar to overloaded functions except that they are more restrictive.
- When functions are overloaded, different actions can be performed within the body of each function.
- But a generic function must perform the same general action for all versions—only the type of data can differ.

Application of Generic Functions – Generic Sort

```
template <class X>
void bubble(
   X *items,  // pointer to array to be sorted
   int count) // number of items in array
{
   register int a, b;
   X t;
   for (a = 1; a < count; a++)
      for (b = count - 1; b >= a; b--)
         if (items[b - 1] > items[b])
         {
            // exchange elements
            t = items[b-1];
            items[b-1] = items[b];
            items[b] = t;
         }
}
```

```
int main()
{
   int iarray[7] = {7, 5, 4, 3, 9, 8, 6};
   double darray[5] = {4.3, 2.5, -0.9, 100.2, 3.0};
   int i;
   cout << "Here is unsorted integer array: ";
   for (i = 0; i < 7; i++)     cout << iarray[i] << ' ';
   cout << endl;
   cout << "Here is unsorted double array: ";
   for (i = 0; i < 5; i++)        cout << darray[i] << ' ';
   cout << endl;
   bubble(iarray, 7);
   bubble(darray, 5);
   cout << "Here is sorted integer array: ";
   for (i = 0; i < 7; i++)    cout << iarray[i] << ' ';
   cout << endl;
   cout << "Here is sorted double array: ";
   for (i = 0; i < 5; i++)        cout << darray[i] << ' ';
   cout << endl;
   return 0;
}
```

Generic Classes

```
const int SIZE = 10;
// Create a generic stack class
template <class StackType> class stack
{
   StackType stck[SIZE]; // holds the stack
   int tos;              // index of top-of-stack
public:
   stack() { tos = 0; }     // initialize stack
   void push(StackType ob); // push object on stack
   StackType pop();         // pop object from stack
};
```

```
template <class StackType>
void stack<StackType>::push(StackType ob)
{
   if (tos == SIZE)    {
      cout << "Stack is full.\n";
      return;
   }
   stck[tos] = ob;
   tos++;
}
```

```
template <class StackType>
StackType stack<StackType>::pop()
{
   if (tos == 0)    {
      cout << "Stack is empty.\n";
      return 0; // return null on empty stack
   }
   tos--;
   return stck[tos];
}
```

Generic Classes (2)

- A generic class defines all the algorithms used by that class
- The actual type of the data being manipulated are specified as a parameter when objects of that class are created.
- Generic classes are useful when a class uses logic that can be generalized.
  - For example, stack, queue
- The compiler will automatically generate the correct type of object, based upon the type specified when the object is created.
- The general form of a generic class declaration is shown here:

```
template <class Ttype>
class class-name {
            .
            .
            .
};
```

- Once a generic class is created, a specific instance of that class using the following general form: class-name <type> ob;

Generic Classes with Multiple Types

```
template <class Type1, class Type2>
class myclass
{
   Type1 i;
   Type2 j;

public:
   myclass(Type1 a, Type2 b)
   {
      i = a;
      j = b;
   }
   void show() { cout << i << ' ' << j << '\n'; }
};
```

```
int main()
{
   myclass<int, double> ob1(10, 0.23);
   myclass<char, char *> ob2('X', "Templates add power.");
   ob1.show(); // show int, double
   ob2.show(); // show char, char *
   return 0;
}
```

Applications of Generic Classes

```
const int SIZE = 10;
template <class AType>
class atype
{
   AType a[SIZE];

public:
   atype() {}
   AType &operator[](int i);
};
```

```
// Provide range checking for atype.
template <class AType>
AType &atype<AType>::operator[](int i)
{
   if (i < 0 || i > SIZE - 1)
   {
      cout << "\nIndex value of ";
      cout << i << " is out-of-bounds.\n";
      exit(1);
   }
   return a[i];
}
```

```
int main()
{
   atype<int> intob; // integer array
   atype<double> doubleob; // double array
   int i;
   cout << "Integer array: ";
   for (i = 0; i < SIZE; i++) intob[i] = i;
   for (i = 0; i < SIZE; i++)
      cout << intob[i] << "  ";
   cout << '\n';
   cout << "Double array: ";
   for (i = 0; i < SIZE; i++)
      doubleob[i] = (double)i / 3;
   for (i = 0; i < SIZE; i++)
      cout << doubleob[i] << "  ";
   cout << '\n';
   intob[12] = 100; // generates runtime
error
   return 0;
}
```

Non-Type Arguments with Generic Classes

```
template <class AType, int size>
class atype
{
   AType a[size]; // length of array
is passed in size
public:
   atype()
   {
      register int i;
      for (i = 0; i < size; i++)
         a[i] = i;
   }
   AType &operator[](int i);
};
```

```
// Provide range checking for atype.
template <class AType, int size>
AType &atype<AType, size>::operator[](int i)
{
   if (i < 0 || i > size - 1)
   {
      cout << "\nIndex value of ";
      cout << i << " is out-of-bounds.\n";
      exit(1);
   }
   return a[i];
}
```

```
int main()
{
   atype<int, 10> intob;      // integer
array of size 10
   atype<double, 15> doubleob; //
double array of size 15
   int i;
   cout << "Integer array: ";
   for (i = 0; i < 10; i++)
      intob[i] = i;
   for (i = 0; i < 10; i++)
      cout << intob[i] << "  ";
   cout << '\n';
   cout << "Double array: ";
   for (i = 0; i < 15; i++)
      doubleob[i] = (double)i / 3;
   for (i = 0; i < 15; i++)
      cout << doubleob[i] << "  ";
   cout << '\n';
   intob[12] = 100; // generates
runtime error
   return 0;
}
```

Default Arguments in Generic Classes

```
template <class AType = int, int size = 10>
class atype
{
   AType a[size]; // size of array is passed in size
public:
   atype() {}
   AType &operator[](int i);
};
```

```
template <class AType, int size>
AType &atype<AType, size>::operator[](int i){
   if (i < 0 || i > size - 1) {
      cout << "\nIndex value of ";
      cout << i << " is out-of-bounds.\n";
      exit(1);
   }
   return a[i];
}
```

```
int main() {
   atype<int, 100> intarray;  // integer array, size 100
   atype<double> doublearray; // double array, default size
   atype<> defarray;          // default to int array of size 10
   int i;
   cout << "int array: ";
   for (i = 0; i < 100; i++)  intarray[i] = i;
   for (i = 0; i < 100; i++)  cout << intarray[i] << "  ";
   cout << '\n';
   cout << "double array: ";
   for (i = 0; i < 10; i++)  doublearray[i] = (double)i / 3;
   for (i = 0; i < 10; i++)  cout << doublearray[i] << "  ";
   cout << '\n';
   cout << "defarray array: ";
   for (i = 0; i < 10; i++)
      defarray[i] = i;
   for (i = 0; i < 10; i++)
      cout << defarray[i] << "  ";
   cout << '\n';
   return 0;
}
```

Template Class Specialization

```
template <class T>
class myclass
{
   T x;

public:
   myclass(T a)
   {
      cout << "Inside generic myclass\n";
      x = a;
   }
   T getx() { return x; }
};
```

```
// Explicit specialization for int.
template <>
class myclass<int>
{
   int x;

public:
   myclass(int a)
   {
      cout << "Inside myclass<int>
specialization\n";
      x = a * a;
   }
   int getx() { return x; }
};
```

```
int main()
{
   myclass<double> d(10.1);
   cout << "double: " << d.getx() << "\n\n";
   myclass<int> i(5);
   cout << "int: " << i.getx() << "\n";
   return 0;
}
```

Inheritance with Templates

```
template<typename T>
class Box {
protected:
   T value;
};
template<>
class Box<int> {
   protected:
      int value;
public:
   Box(int v){value=v;}
   int get() const { return value;}
};
class IntBox : public Box<int> {
public:
   IntBox(int v):Box<int>(v){}
};
```

```
template<typename T>
class Box {
 protected:
    T value;
};
class IntBox : public Box<int> {
  public:
     IntBox(int v){value=v;}
     int get() const { return value;}
};

int main() {
   IntBox b(10);
   cout<<b.get();
}
```

```
template<typename T>
class Box {
  protected:
    T value;
};
template<typename T>
class LoggedBox : public Box<T> {
public:
   LoggedBox (T v){this->value=v;}
   void log() const {
      cout << this->value << "\n";
   }
};
int main() {
   LoggedBox<int> b(10);
   b.log();
}
```

Typename Keyword

```
template <class X>
 void swapargs(X &a, X &b)
{
   X temp;
   temp = a;
   a = b;
   b = temp;
}
```

```
template <typename X>
void swapargs(X &a, X &b)
{
   X temp;
   temp = a;
   a = b;
   b = temp;
}
```

Standard Template Library (STL)

Introduction to STL

- Developed by Alexander Stepanov and Meng Lee at Hewlett-Packard Lab
  - as proof-of-concept for so-called *generic programming*.
- Released in 1994 and subsequently adopted into the C++98.
- STL provides general-purpose, templatized classes and functions that implement
  - many popular and commonly used algorithms and
  - data structures such as vectors, lists, queues, and stacks.
- It also defines various routines that access them.
- Because the STL is constructed from template classes, the algorithms and data structures can be applied to nearly any type of data.

STL Templates

Container

Templatized data structure. Can be used to hold fundamental-type values or almost any type of objects, e.g., vector<int>, list<string>, deque<Person>.

Iterator

A generalization of pointer. An object that allows traversing through elements of a container, e.g., vector<int>::iterator, list<string>:: iterator.

Algorithms

Used for tasks such as searching, sorting and comparison, e.g., for_each, find, sort.

Function Objects

Objects that act like functions.

Types of Containers

Sequence Containers

linear data structures of elements

array (C++ 11): Not a STL container because it has a fixed size and does not support operations like insert. But STL algorithms can be applied.

vector: dynamically resizable array. Support fast insertion and deletion at back; and direct access to its elements.

deque: double-ended queue. Support fast insertion and deletion at front and back; and direct access to its elements.

forward_list (C++ 11): single-linked list that support forward traversal only. It is simpler than list.

list: double-linked list. Support fast insertion and deletion anywhere in the list; and direct access to its elements.

First Class Containers

Container Adapter Classes

constrained sequence containers

stack: Last-in-first-out (LIFO) queue, adapted from deque (default), or vector, or list. Support operations back, push_back, pop_back.

queue: First-in-first-out (FIFO) queue, adapted from deque (default), or list. Support operations front, back, push_back, pop_front.

priority_queue: highest priority element at front of the queue. Adapted from vector (default) or deque. Supports operations front, push_back, pop_front.

Types of Containers (2)

Associative Containers

nonlinear data structures storing key-value pairs, sorted by keys (ordered)

set: No duplicate element. Support fast lookup.

multiset: Duplicate element allowed. Support fast lookup.

map: One-to-one mapping (associative array) with no duplicate. Support fast key lookup.

multimap: One-to-many mapping, with duplicates allowed. Support fast key lookup.

Unordered Associative Containers

based on hash table, efficient in insertion, removal and search but requires more storage

unordered_set: No duplicate element. Support fast lookup.

unordered_multiset: Duplicate element allowed. Support fast lookup.

unordered_map: One-to-one mapping (associative array) with no duplicate. Support fast key lookup.

Unordered_multimap: One-to-many mapping, with duplicates allowed. Support fast key lookup.

C++ 11

Types of Containers (3)

Associative Containers

nonlinear data structures storing key-value pairs, sorted by keys (ordered)

set: No duplicate element. Support fast lookup.

multiset: Duplicate element allowed. Support fast lookup.

map: One-to-one mapping (associative array) with no duplicate. Support fast key lookup.

multimap: One-to-many mapping, with duplicates allowed. Support fast key lookup.

Assoc. Container Adapter Classes

First Iteration, Smaller Memory Consumption

flat_set: No duplicate element. Support fast lookup.

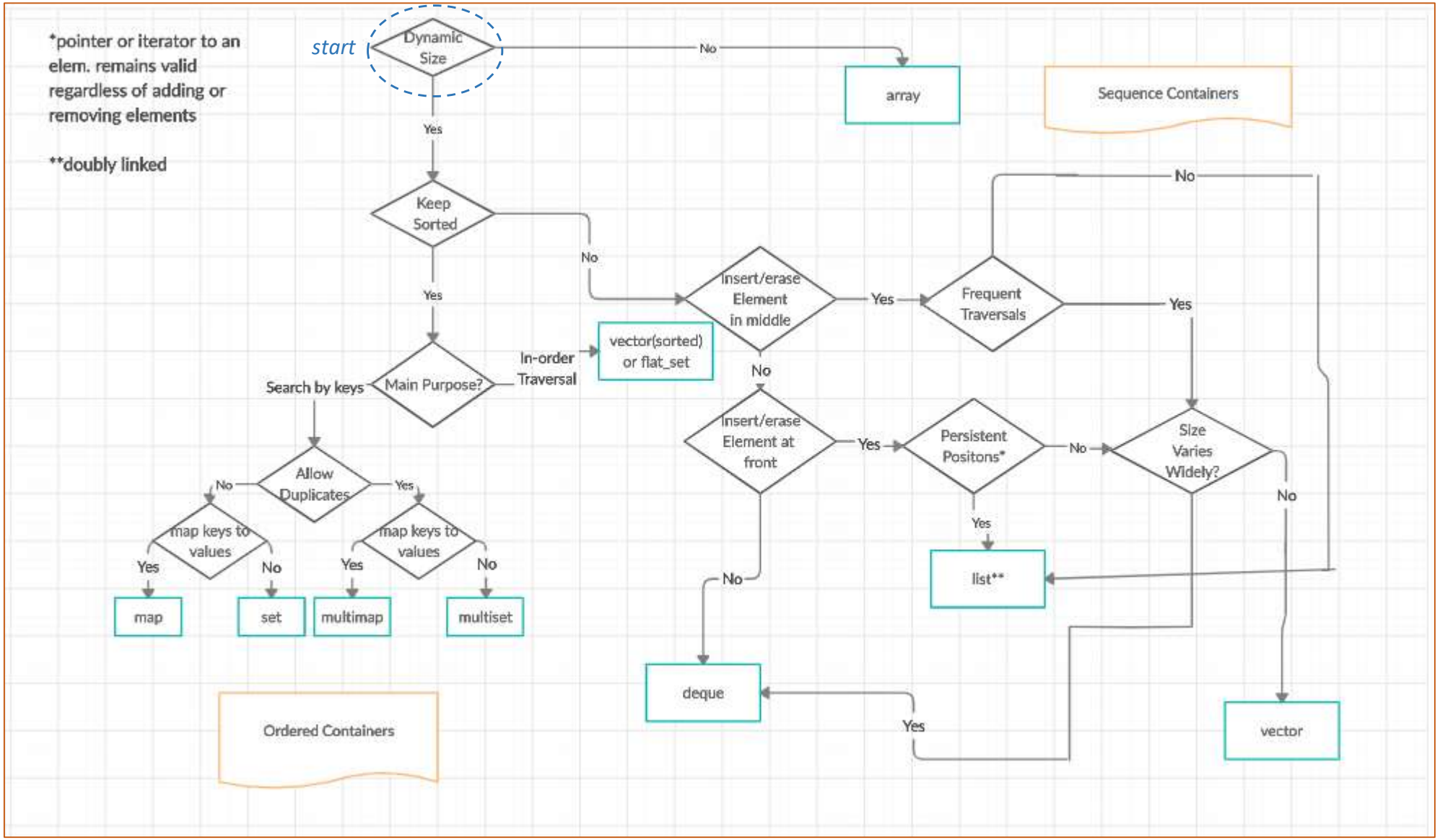
flat_multiset: Duplicate element allowed. Support fast lookup.

flat_map: One-to-one mapping (associative array) with no duplicate. Support fast key lookup.

flat_multimap: One-to-many mapping, with duplicates allowed. Support fast key lookup.

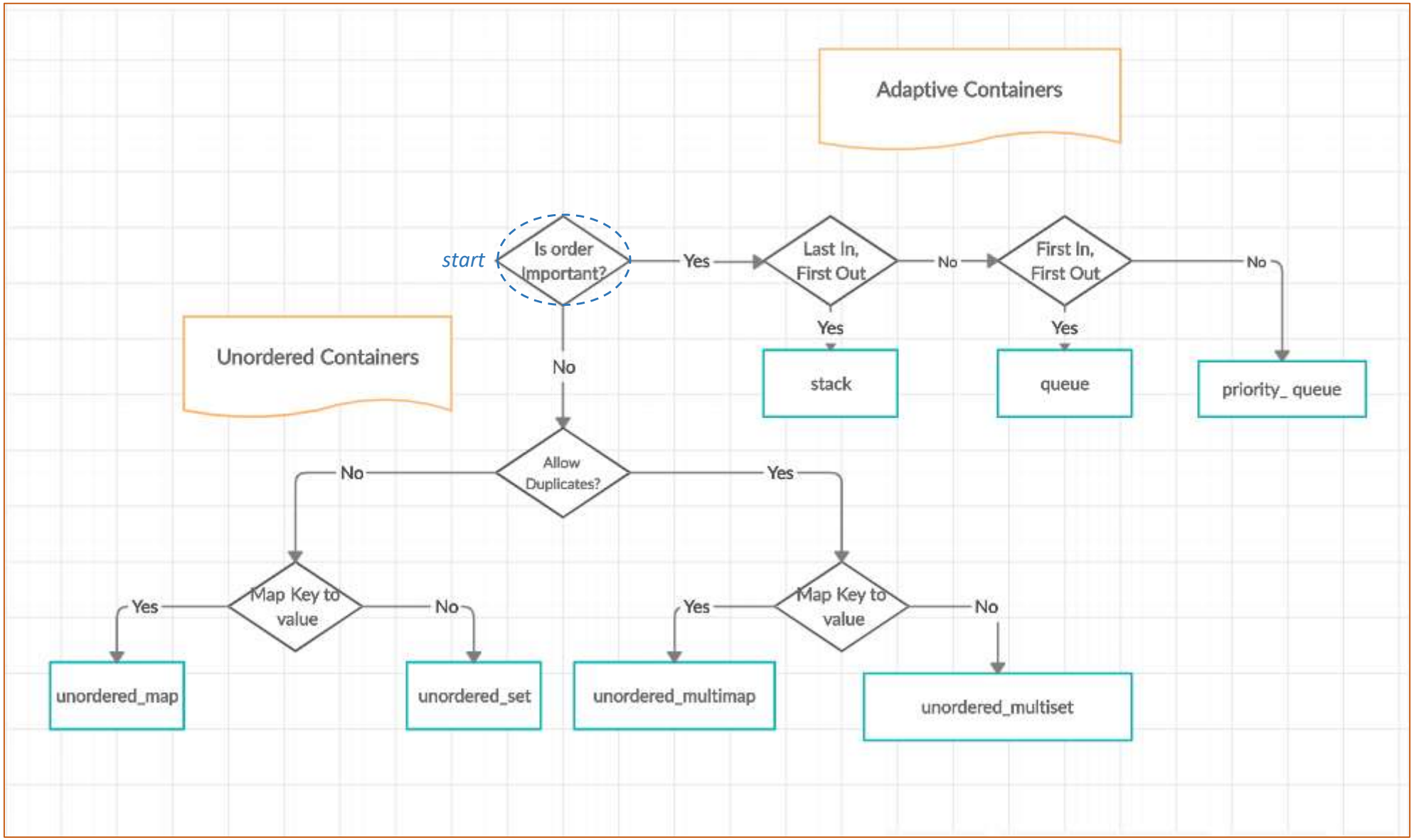
C++ 23

Flowchart of Sequence and Ordered Containers



Ref: https://www.geeksforgeeks.org/containers-cpp-stl/

Flowchart of Adaptive and Unordered Containers



Ref: https://www.geeksforgeeks.org/containers-cpp-stl/

Types of Containers

Full List: https://en.cppreference.com/w/cpp/container

Simple Containers

pair:

- The pair container is a simple associative container consisting of a 2-tuple of data elements or objects, called 'first' and 'second', in that fixed order.
- The STL 'pair' can be assigned, copied and compared.
- The array of objects allocated in a map or hash_map are of type 'pair' by default, where all the 'first' elements act as the unique keys, each associated with their 'second' value objects.

tuple:

- Fixed size collection of heterogeneous values
- Generalization of pair

Other Container Classes

string: C++ String, not a STL container but STL algorithms can be applied

bitset: stores series of bits similar to a fixed-sized vector of bools. Implements bitwise operations and lacks iterators. Not a sequence. Provides random access.

valarray: Another array data type, intended for numerical use (especially to represent vectors and matrices); the C++ standard allows specific optimizations for this intended purpose.

Container Functions

All containers provides these functions:

- Default Constructor: to construct an empty container. Other constructors are provided for specific purposes.
- Copy Constructor
- Destructor
- empty(): returns true if the container is empty.
- size(): returns the size (number of elements).
- Assignment Operator (=)
- Comparison Operators (==, !=, <, <=, >, >=).
- swap(): exchanges the contents of two containers.

In addition, the first-class containers support these functions:

- begin, end, cbegin, cend: Returns the begin and end iterator, or the const version.
- rbegin, rend, crbegin, crend: Returns the reverse_iterator.
- clear(): Removes all elements.
- erase(): Removes one element given an iterator, or a range of elements given [begin, end) iterators.
- max_size(): Returns the maximum number of elements that the container can hold.

Container Headers

<vector>
<list>
<deque>
<queue>: queue and priority_queue
<stack>
<map>: map and multimap
<set>: set and multiset

<valarray>
<bitset>
<array> (C++11)
<forward_list> (C++11)
<unordered_map> (C++11)
<unordered_set> (C++11)

In addition to the headers required by the various STL classes, the C++ standard library includes the <utility> and <functional> headers, which provide support for the STL. For example, the template class pair, which can hold a pair of values, is defined in <utility>.

Iterators

- An iterator behaves like a generic pointer, that can be used to
  - reference (point-to) individual element of a generic container; and
  - transverse through elements of a container.
- The purpose of iterator is to make traversing (iterating) of containers independent on the type of the containers (e.g., vector<int>, queue<double>, stack<string>).
- With iterator, it is possible apply generic algorithm (such as searching, sorting and comparison) to the container, independent of types.

Requirements of Iterators

- The dereferencing operator * shall be defined.
  - if *iter* is an iterator, **iter* shall be pointing to an element of the container.
- The assignment operator = shall be defined.
  - if *iter1* and *iter2* are iterators, *iter1* = *iter2* shall assign *iter2* to *iter1*.
- The comparison operators == and != shall be defined.
  - if *iter1* and *iter2* are iterators, we can use *iter1* == *iter2* and *iter1* != *iter2* to compare them for equality.
  - *iter1* == *iter2* is true, then they shall be pointing at the same element, i.e., **iter1* == **iter2*.

Requirements of Iterators (2)

- The increment operator ++ shall be defined.
  - if *iter* is an iterator, ++*iter* and *iter*++ move the iterator to point to the next element.
  - The program shall be able to iterate through all the elements via ++*iter* (or *iter*++) operations.
- For linearly-ordered container, the + (and +=) operator shall be defined.
  - if *iter* is an iterator, *iter*+*n* points to the next *n*-th element in the linear order.
- For iterator that can transverse backward, the decrement operator -- shall be defined.
  - if *iter* is an operator, --*iter* and *iter*-- move the iterator to point to the next element in the reverse order (or previous element in the forward order).

Working with Iterators

- All STL container provides two member functions:
  - begin() and end() that return the iterators pointing to the first element and the pass-the-end element respectively.
- Hence, the following code to transverse all the elements in the container:

```
// Assume that c is a container
iter_type iter;
for (iter = c.begin(); iter != c.end(); ++iter) {
   // Use *iter to reference each element
   ......
}
```

The type of iterator (*iter_type*) depends on the container. In STL, the iterator type can be obtained via `container<T>::iterator`.

Auto and for_each in C++11

- In C++11, the auto keyword is used to derive the type of iterator automatically, as follows:

```
for (auto iter = c.begin(); iter != c.end(); ++iter) {
  // Use *iter to reference each element
  ......
}
```

- In C++11, you can also use the new for-each loop to iterate through all the element of a container:

```
for (auto item : container) {
  // Use item to reference each element
  ......
}
```

The Vector Template Class

- In computing, a vector refers to an array-like structure that holds
  - a set of direct-access elements of the same kinds, instead of mathematical n-component vector.
- Unlike array which is fixed-size, vector is dynamically-sized.
- Supports fast insertion and deletion at back and direct access to its elements.
- `vector` is a class template, declared in the `vector` header.

Vector Basics

```
// Demonstrate a vector.
#include <iostream>
#include <vector>
#include <cctype>
using namespace std;
int main()
{
   vector<char> v(10); // create a vector of length 10
   unsigned int i;
   // display original size of v
   cout << "Size = " << v.size() << endl;
   cout << "Capacity = " << v.capacity() << endl;
   // assign the elements of the vector some values
   for (i = 0; i < 10; i++)
      v[i] = i + 'a';
   // display contents of vector
   cout << "Current Contents:\n";
   for (i = 0; i < v.size(); i++)
      cout << v[i] << " ";
   cout << "\n\n";
```

Header file for vector

Create a vector of type char

Get the size of vector

Get the capacity

Access vector like an array

```
/* put more values onto the end of the vector, it will grow as
needed */
   cout << "Expanding vector\n";
   for (i = 0; i < 5; i++)
      v.push_back(i + 10 + 'a');
   // display current size of v
   cout << "Size now = " << v.size() << endl;
   cout << "Capacity = " << v.capacity() << endl;
   // display contents of vector
   cout << "Current contents:\n";
   for (i = 0; i < v.size(); i++)
      cout << v[i] << " ";
   cout << "\n\n";
   // change contents of vector
   for (i = 0; i < v.size(); i++)
      v[i] = toupper(v[i]);
   cout << "Modified Contents:\n";
   for (i = 0; i < v.size(); i++)
      cout << v[i] << " ";
   cout << endl;
}
```

Insert at the end

What would be the value of capacity here?

Iterator with Vector

```
#include <iostream>
#include <vector>
#include <cctype>
using namespace std;
int main()
{
   vector<char> v(10);
   // create an iterator
   vector<char>::iterator p;
   int i;
   // assign elements in vector a value
   p = v.begin()
   i = 0;
   while (p != v.end())
   {
      *p = i + 'a';
      p++;
      i++;
   }
```

create an iterator

Initialize (like a pointer)

Access through Iterator

Advance to next element

```
// display contents of vector
   cout << "Original contents:\n";
   p = v.begin();
   while (p != v.end())
   {
      cout << *p << " ";
      p++;
   }
   cout << "\n\n";
   // change contents of vector
   p = v.begin();
   while (p != v.end())
   {
      *p = toupper(*p);
      p++;
   }
```

```
// display contents of vector
   cout << "Modified Contents:\n";
   p = v.begin();
   while (p != v.end())
   {
      cout << *p << " ";
      p++;
   }
   cout << endl;
   return 0;
}
```

Auto Type Deduction of Iterators in C++11

```
// Access the elements of a vector through an iterator.
#include <iostream>
#include <vector>
#include <cctype>
using namespace std;
int main() {
   vector<char> v(10); // create a vector of length 10
   int i=0;
   for (auto p = v.begin(); p != v.end(); ++p) {
      *p = i + 'a';
      i++;
   }
   // display contents of vector
   cout << "Original contents:\n";
   for (auto p = v.begin(); p != v.end(); ++p) {
      cout << *p << " ";
   }
   cout << "\n\n";
   return 0;
}
```

Use of auto

```
// Access the elements of a vector through an iterator.
#include <iostream>
#include <vector>
#include <cctype>
using namespace std;
int main() {
   vector<char> v(10); // create a vector of length 10
   int i=0;
   for (auto p = v.begin(); p != v.end(); ++p) {
      *p = i + 'a';
      i++;
   }
   // display contents of vector
   cout << "Original contents:\n";
   for (auto p : v) {
      cout << p << " ";
   }
   cout << "\n\n";
   return 0;
}
```

for_each

Insert and Erase in Vector

```
#include <iostream>
#include <vector>
using namespace std;
int main() {
   vector<char> v(10);
   vector<char> v2;
   char str[] = "<Vector>";
   unsigned int i;
   // initialize v
   for (i = 0; i < 10; i++)
      v[i] = i + 'a';
   // copy characters in str into v2
   for (i = 0; str[i]; i++)
      v2.push_back(str[i]);
   // display original contents of vector
   cout << "Original contents of v:\n";
   for (i = 0; i < v.size(); i++)
      cout << v[i] << " ";
   cout << "\n\n";
```

```
   vector<char>::iterator p = v.begin();
   p += 2; // point to 3rd element
   // insert 10 X's into v
   v.insert(p, 10, 'X');
   // display contents after insertion
   cout << v.size() << endl;
   cout << "Contents after insert:\n";
   for (i = 0; i < v.size(); i++)
      cout << v[i] << " ";
   cout << "\n\n";
   // remove those elements
   p = v.begin();
   p += 2;  // point to 3rd element
   // remove next 10 elements
   v.erase(p, p + 10);
   // display contents after deletion
   cout << v.size() << endl;
   cout << "Contents after erase:\n";
   for (i = 0; i < v.size(); i++)
      cout << v[i] << " ";
   cout << "\n\n";
```

Insert elements

Erase from vector

```
   // Insert v2 into v
   v.insert(p, v2.begin(), v2.end());
   cout << "Size after v2's insertion = ";
   cout << v.size() << endl;
   cout << "Contents after insert:\n";
   for (i = 0; i < v.size(); i++)
      cout << v[i] << " ";
   cout << endl;
   return 0;
}
```

Insert from vector

Store Objects in Vector

```
#include <iostream>
#include <vector>
#include <cstdlib>
using namespace std;
class DailyTemp {
   int temp;
public:
   DailyTemp() { temp = 0; }
   DailyTemp(int x) { temp = x; }
   DailyTemp &operator=(int x) {
      temp = x;
      return *this;
   }
   double get_temp() { return temp; }
};
bool operator<(DailyTemp a, DailyTemp b) {
   return a.get_temp() < b.get_temp();
}
bool operator==(DailyTemp a, DailyTemp b) {
   return a.get_temp() == b.get_temp();
}
```

```
int main()
{
   vector<DailyTemp> v;
   unsigned int i;
   for (i = 0; i < 7; i++)
      v.push_back(DailyTemp(60 + rand() % 30));
   cout << "Fahrenheit temperatures:\n";
   for (i = 0; i < v.size(); i++)
      cout << v[i].get_temp() << " ";
   cout << endl;
   // convert from Fahrenheit to Centigrade
   for (i = 0; i < v.size(); i++)
      v[i] = (int)(v[i].get_temp() - 32) * 5 / 9;
   cout << "Centigrade temperatures:\n";
   for (i = 0; i < v.size(); i++)

      cout << v[i].get_temp() << " ";
   return 0;
}
```

Create a vector of objects

Insert objects

Access member function

Functions in Vector Template Class

Constructors

```
vector (const allocator_type & alloc = allocator_type()); // Default Constructor: construct a vector object

vector (size_type n, const value_type & val = value_type(), const allocator_type & alloc = allocator_type()); // Fill Constructor: construct a vector object with n-element filled with val

vector (const vector & v); // Copy Constructor

template <class InputIterator> vector (InputIterator first, InputIterator last, const allocator_type & alloc = allocator_type()); // Range Copy Constructor
```

Size and Capacity

```
size_type size () const;      // Return the size (number of elements)
size_type capacity () const;  // Return the storage allocated (in term of element)
bool empty () const;          // Return true if size is 0
void reserve (size_type n);   // Request for storage to hold n elements
void resize (size_type n, value_type val = value_type());
    // resize to n, remove extra element or fill with val
size_type max_size () const;  // Return the maximum number of element
void shrink_to_fit ();        // (C++11) Request to shrink storage
```

Functions in Vector Template Class

Accessing Elements

```
value_type & operator[] (size_type n);  // [n] operator (without index-bound check)
value_type & at (size_type n);          // Return a reference to n-th element with index-bound check
value_type & front ();    // Return a reference to the first element
value_type & back ();     // Return a reference to the last element
```

Modifying Contents

```
void push_back (const value_type & val); // Append val at the end
void pop_back ();                        // Remove the last element
void clear ();                           // Remove all elements
```

Non-member Friend Functions

```
==, !=, <, >, <=, >=    // Comparison Operators
// E.g.
template <class T, class Alloc>
bool operator== (const vector<T,Alloc> & left, const vector<T, Alloc> & right);   // Compare two vectors. For == and !=, first compare the size, then each element with equal algorithm. Stop at the first mismatch. For <, >, <=, >=, use lexicographical_compare algorithm. Stop at first mismatch.
template <class T, class Alloc>
void swap (vector<T,Alloc> & v1, vector<T,Alloc> v2); // Swap the contents of containers v1 and v2. Both shall has the same type, but can have different sizes.
```

Functions in Vector Template Class

Iterator

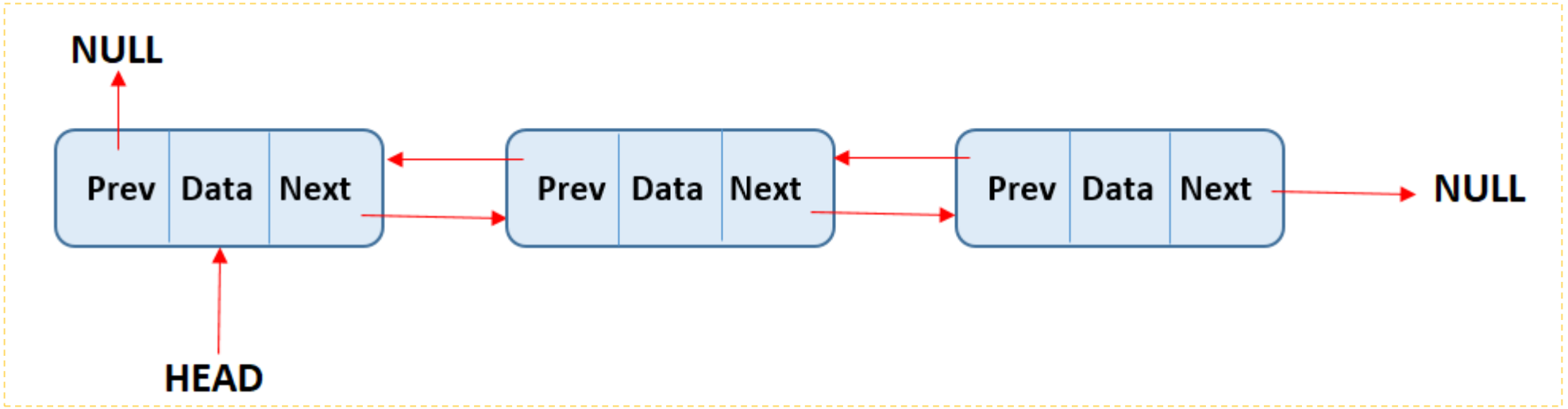
```
iterator begin();  // Return an iterator pointing to the first element
iterator end();    // Return an iterator pointing to the pass-the-end element
reverse_iterator rbegin(); // Return a reverse iterator pointing to the reverse beginning (last element), increasing a reverse iterator to transverse in reverse order
reverse_iterator rend();   // Return a reverse iterator pointing to the reverse past-the-end
```

Iterator-based Operations

```
iterator insert (iterator pos, const value_type & val);  // Single-Element: insert element val before iterator pos
void     insert (iterator pos, size_type n, const value_type & val);  // Fill: insert n copies of val before pos
template <class InputIterator>
void     insert (iterator pos, InputIterator first, InputIterator last)
   // Range-copy: copy the range [first, last) and insert before pos.

iterator erase (iterator pos);  // Single-element: remove element pointed to by iterator pos
iterator erase (iterator first, iterator last);  // Range: remove elements between [first,last)

void assign (size_type n, const value_type & val);  // Fill: clear old contents and assign n copies of val
template <class InputIterator>
void assign (InputIterator first, InputIterator last);  // Range: assign [first, last)
```

The List Template Class

- `list` is a class template, declared in the `list` header.
- The list class supports a bidirectional, linear list.
- Unlike a vector, which supports random access, a list can be accessed sequentially only.
- Since lists are bidirectional, they may be accessed front to back or back to front.



List Basics

```
// List basics.
#include <iostream>
#include <list>
using namespace std;
int main()
{
   list<int> lst; // create an empty list
   int i;
   for (i = 0; i < 10; i++)
      lst.push_back(i);
   cout << "Size = " << lst.size() << endl;
   cout << "Contents: ";
   list<int>::iterator p = lst.begin();
   while (p != lst.end())
   {
      cout << *p << " ";
      p++;
   }
   cout << "\n\n";
```

Create a list of integers

Insert elements

Access with iterator

```
// change contents of list
   p = lst.begin();
   while (p != lst.end())
   {
      *p = *p + 100;
      p++;
   }
   cout << "Contents modified: ";
   p = lst.begin();
   while (p != lst.end())
   {
      cout << *p << " ";
      p++;
   }
   return 0;
}
```

How could you print a list backward?

```cpp
#include <iostream>
#include <list>
using namespace std;
int main()
{
    char string[]="?drawkcab tsil a tnirp uoy dluow woH";
    list<char> lst; // create an empty list
    int i;
    while (string[i])
        lst.push_back(string[i++]);
    cout << "List printed backwards:\n";
                    ?
    return 0;
}
```

Push_Back() and Push_Front()

```
/* Demonstrating the difference between
push_back() and push_front(). */
#include <iostream>
#include <list>
using namespace std;
int main()
{
   list<int> lst1, lst2;
   int i;
   for (i = 0; i < 10; i++)
      lst1.push_back(i);
   for (i = 0; i < 10; i++)
      lst2.push_front(i);
```

```
list<int>::iterator p;
   cout << "Contents of lst1:\n";
   p = lst1.begin();
   while (p != lst1.end())
   {
      cout << *p << " ";
      p++;
   }
   cout << "\n\n";
   cout << "Contents of lst2:\n";
   p = lst2.begin();
   while (p != lst2.end())
   {
      cout << *p << " ";
      p++;
   }
   return 0;
}
```

Sort a List

```
// Sort a list.
#include <iostream>
#include <list>
#include <cstdlib>
using namespace std;
int main()
{
   list<int> lst;
   int i;
   // create a list of random integers
   for (i = 0; i < 10; i++)
      lst.push_back(rand());
   cout << "Original contents:\n";
   list<int>::iterator p = lst.begin();
   while (p != lst.end())  {
      cout << *p << " ";
      p++;
   }
   cout << endl << endl;
```

```
// sort the list
   lst.sort();  ← Sort list
   cout << "Sorted contents:\n";
   p = lst.begin();
   while (p != lst.end())
   {
      cout << *p << " ";
      p++;
   }
   return 0;
}
```

Merge Two Lists

```
// Merge two lists.
#include <iostream>
#include <list>
using namespace std;
int main() {
   list<int> lst1, lst2;
   int i;
   for (i = 0; i < 10; i += 2) lst1.push_back(i);
   for (i = 1; i < 11; i += 2) lst2.push_back(i);
   cout << "Contents of lst1:\n";
   list<int>::iterator p = lst1.begin();
   while (p != lst1.end())  {
      cout << *p << " ";
      p++;
   }
   cout << endl  << endl;
   cout << "Contents of lst2:\n";
   p = lst2.begin();
   while (p != lst2.end()) {
      cout << *p << " ";
      p++;
   }
   cout << endl << endl;
   // now, merge the two lists
   lst1.merge(lst2);  ← Merge lists
   if (lst2.empty()) cout << "lst2 is now empty\n";
   cout << "Contents of lst1 after merge:\n";
   p = lst1.begin();
   while (p != lst1.end()) {
      cout << *p << " ";
      p++;
   }
   return 0;
}
```

Store Objects in List

```
class myclass {
   int a, b, sum;
public:
   myclass() { a = b = 0; }
   myclass(int i, int j)  {
      a = i; b = j; sum = a + b;
   }
   int getsum() { return sum; }
   friend bool operator<(const myclass &o1, const myclass &o2) {
       return o1.sum < o2.sum;
   }
   friend bool operator>(const myclass &o1, const myclass &o2) {
       return o1.sum > o2.sum;
   }
   friend bool operator==(const myclass &o1, const myclass &o2) {
      return o1.sum == o2.sum;
   }
   friend bool operator!=(const myclass &o1, const myclass &o2) {
      return o1.sum != o2.sum;
   }
};
```

Mandatory

Mandatory

```
int main() {
   int i;
   list<myclass> lst1; // create first list
   for (i = 0; i < 10; i++)   lst1.push_back(myclass(i, i));
   cout << "First list: ";
   list<myclass>::iterator p = lst1.begin();
   while (p != lst1.end())  { cout << p->getsum() << " "; p++; }
   cout << endl;
   list<myclass> lst2; // create a second list
   for (i = 0; i < 10; i++) lst2.push_back(myclass(i * 2, i * 3));
   cout << "Second list: ";
   p = lst2.begin();
   while (p != lst2.end()) { cout << p->getsum() << " ";   p++; }
   cout << endl;
   lst1.merge(lst2);  // now, merget lst1 and lst2
   cout << "Merged list: ";
   p = lst1.begin();
   while (p != lst1.end()) { cout << p->getsum() << " "; p++; }
   return 0;
}
```

The Stack Template Class

- A container adaptor that permits LIFO access
- Uses vector, deque (by default) or list as underlying container
- Defined in header stack

```
template <class Type, class Container = deque<Type> >
class stack;
```

- Supported functions are
  - empty()
  - size()
  - top()
  - push(), emplace(C++11), push_range (C++23)
  - pop()
  - swap (C++11)

```
#include <iostream>
#include <stack>
using namespace std;
int main() {
   stack<int> stack;
   stack.push(21);
   stack.push(22);
   stack.push(24);
   stack.push(25);
   int num = 0;
   stack.push(num);
   stack.pop();
   stack.pop();
   stack.pop();

   while (!stack.empty())    {
      cout << stack.top() << " ";
      stack.pop();
   }
}
```

The Queue Template Class

- A container adaptor that permits FIFO access
- Uses deque (by default) or list as underlying container
- Defined in header queue

```
template <class Type, class Container = deque<Type> >
class queue;
```

- Supported functions are
  - front()
  - back()
  - empty()
  - size()
  - push(), emplace(C++11), push_range (C++23)
  - pop()
  - swap()

```
#include <iostream>
#include <queue>
using namespace std;
// Print the queue
void print_queue(queue<int> q){
   queue<int> temp = q;
   while (!temp.empty()) {
      cout << temp.front()<<“ "<<temp.back()<<'\n';
      temp.pop();
   }
   cout << '\n';
}
int main(){
   queue<int> q1;
   q1.push(1);
   q1.push(2);
   q1.push(3);
   cout << "The first queue is : ";
   print_queue(q1);
   return 0;
}
```

The Priority Queue Template Class

- A container adaptor that provides constant time lookup of the largest (by default) or smallest element
- Uses vector (by default) as underlying container
- Defined in header queue

```
template< class T, class Container = std::vector<T>,
   class Compare = std::less<typename Container::value_type>
> class priority_queue;
```

- Supported functions are
  - empty()
  - size()
  - top()
  - push(), emplace(C++11)
  - pop()

```
#include <iostream>
#include <queue>
using namespace std;
int main(){
   priority_queue<int> q;
   q.push(1);
   q.push(2);
   q.push(3);
   cout << "The priority queue is : ";
   while (!q.empty()) {
      cout << q.top() <<" ";
      q.pop();
   }
   cout << '\n';
   return 0;
}
```

The Pair Template Class

teach yourself: tuple class

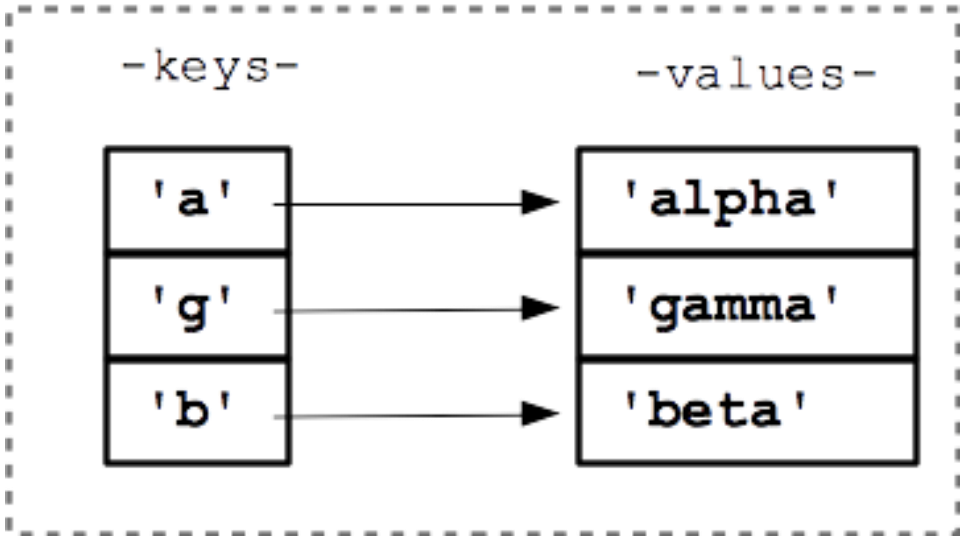
```cpp
#include <iostream>
#include <utility>
using namespace std;
int main(){
   pair<int, char> PAIR1;
   pair<string, double> PAIR2("GeeksForGeeks", 1.23);
   pair<string, double> PAIR3;
   PAIR1.first = 100;
   PAIR1.second = 'G';
   PAIR3 = make_pair("GeeksForGeeks is Best", 4.56);
   cout << PAIR1.first << " ";
   cout << PAIR1.second << endl;
   cout << PAIR2.first << " ";
   cout << PAIR2.second << endl;
   cout << PAIR3.first << " ";
   cout << PAIR3.second << endl;

   return 0;
}
```

Ref: https://en.cppreference.com/w/cpp/utility/pair

```cpp
#include <iostream>
#include <utility>
using namespace std;
int main(){
   pair<char, int> pair1 = make_pair('A', 1);
   pair<char, int> pair2 = make_pair('B', 2);

   cout << "Before swapping:\n ";
   cout << "Contents of pair1 = " << pair1.first << " "
      << pair1.second;
   cout << "Contents of pair2 = " << pair2.first << " "
      << pair2.second;
   pair1.swap(pair2);

   cout << "\nAfter swapping:\n ";
   cout << "Contents of pair1 = " << pair1.first << " "
      << pair1.second;
   cout << "Contents of pair2 = " << pair2.first << " "
      << pair2.second;

   return 0;
}
```

The Map Template Class

- The map class supports an associative container in which unique keys are mapped with values.
- In essence, a key is simply a name that is given to a value.
- Once a value has been stored, it can be retrieved by using its key.
- Thus, in its most general sense, a map is a list of key/value pairs.
- A map can hold only unique keys. Duplicate keys are not allowed.
- To create a map that allows nonunique keys, use multimap.



Map Basics

```
// A simple map demonstration.
#include <iostream>
#include <map>
using namespace std;
int main()
{
   map<char, int> m;
   int i;
   // put pairs into map
   for (i = 0; i < 26; i++)
   {
      m.insert(pair<char, int>('A' + i, 65 + i));
   }
```

```
char ch;
   cout << "Enter key: ";
   cin >> ch;
   map<char, int>::iterator p;
   // find value given key
   p = m.find(ch);
   if (p != m.end())
      cout << "Its ASCII value is  " << p->second;
   else
      cout << "Key not in map.\n";
   return 0;
}
```

Application of Maps

```
// Use a map to create a phone directory.
#include <iostream>
#include <map>
#include <cstring>
using namespace std;
class name {
   char str[40];
public:
   name() { strcpy(str, ""); }
   name(char *s) { strcpy(str, s); }
   char *get() { return str; }
};
// Must define less than relative to name objects.
bool operator<(name a, name b) {
   return strcmp(a.get(), b.get()) < 0;
}
class phoneNum {
   char str[80];
public:
   phoneNum() { strcmp(str, ""); }
   phoneNum(char *s) { strcpy(str, s); }
   char *get() { return str; }
};
```

```
int main(){
   map<name, phoneNum> directory;
   // put names and numbers into map
   directory.insert(pair<name, phoneNum>(name("Tom"), phoneNum("555-4533")));
   directory.insert(pair<name, phoneNum>(name("Chris"), phoneNum("555-
9678")));
   directory.insert(pair<name, phoneNum>(name("John"), phoneNum("555-
8195")));
   directory.insert(pair<name, phoneNum>(name("Rachel"), phoneNum("555-
0809")));
   // given a name, find number
   char str[80];
   cout << "Enter name: ";
   cin >> str;
   map<name, phoneNum>::iterator p;
   p = directory.find(name(str));
   if (p != directory.end())
      cout << "Phone number: " << p->second.get();
   else
      cout << "Name not in directory.\n";
   return 0;
}
```

Algorithms in STL

- The STL defines a large number of algorithms for a variety of purposes (e.g. searching, sorting, counting, manipulating) that operate on ranges of elements.
- All of the algorithms are non-member template functions. This means that they can be applied to any type of container.
- The algorithms operate on elements of STL container only indirectly through the iterator. It accepts a pair of iterators, denoted as first and last, that mark the range of operation as [first,last) (including first, but excluding last). For example,

```
sort(aVector.begin(), aVector.end());  // Sort the entire vector
sort(aVector.begin(), aVector.begin + aVector.size()/2);  // Sort first half
```

- Some examples of algorithms are
  - for_each, find, find_if, count, count_if, remove_copy, replace_copy, unique, shuffle, transform, etc.

See https://en.cppreference.com/w/cpp/algorithm for the complete list

Algorithms: for_each

```
//Testing for_each algorithms (TestForEach.cpp)
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;




int main() {
  vector<int> v;
  v.push_back(11);
  v.push_back(3);
  v.push_back(4);
  v.push_back(22);
```

```
 // Invoke the given function (print, square)
  // for each element in the range
  for_each(v.begin(), v.end(), print);
  for_each(v.begin() + 1, v.begin() + 3, square);
  for_each(v.begin(), v.end(), print);
  return 0;
}
```

Algorithms: count and count_if

```
// Demonstrate count().
#include <iostream>
#include <vector>
#include <cstdlib>
#include <algorithm>
using namespace std;
int main() {
   vector<bool> v;
   unsigned int i;
   for (i = 0; i < 10; i++) {
      if (rand() % 2) v.push_back(true);
      else v.push_back(false);
   }
   cout << "Sequence:\n";
   for (i = 0; i < v.size(); i++)
      cout << boolalpha << v[i] << " ";
   cout << endl;
   i = count(v.begin(), v.end(), true);
   cout << i << " elements are true.\n";
   return 0;
}
```

```
// Demonstrate count_if().
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
/* This is a unary predicate that determines if number
is divisible by 3. */
bool dividesBy3(int i){
   if ((i % 3) == 0) return true;
   return false;
}
int main() {
   vector<int> v;
   int i;
   for (i = 1; i < 20; i++)  v.push_back(i);
   cout << "Sequence:\n";
   for (i = 0; i < v.size(); i++) cout << v[i] << " ";
   cout << endl;
   i = count_if(v.begin(), v.end(), dividesBy3);
   cout << i << " numbers are divisible by 3.\n";
   return 0;
}
```

Algorithms: remove_copy, replace_copy

```
#include <vector>
#include <algorithm>
using namespace std;
int main()
{
   char str[] = "The STL is power programming.";
   vector<char> v, v2(30);
   unsigned int i;
   for (i = 0; str[i]; i++) v.push_back(str[i]);
   // **** demonstrate remove_copy ****
   cout << "Input sequence:\n";
   for (i = 0; i < v.size(); i++)  cout << v[i];
   cout << endl;
// remove all spaces
   remove_copy(v.begin(), v.end(), v2.begin(), ' ');
   cout << "Result after removing spaces:\n";
   for (i = 0; i < v2.size(); i++)  cout << v2[i];
   cout << endl << endl;
```

```
// **** now, demonstrate replace_copy ****
   cout << "Input sequence:\n";
   for (i = 0; i < v.size(); i++)  cout << v[i];
   cout << endl;
   // replace spaces with colons
   replace_copy(v.begin(), v.end(), v2.begin(), ' ', ':');
   cout << "Result after replacing spaces with colons:\n";
   for (i = 0; i < v2.size(); i++)
      cout << v2[i];
   cout << endl
        << endl;
   return 0;
}
```

Algorithms: reverse and transform

```
// Demonstrate reverse.
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
int main() {
   vector<int> v;
   unsigned int i;
   for (i = 0; i < 10; i++)  v.push_back(i);
   cout << "Initial: ";
   for (i = 0; i < v.size(); i++)  cout << v[i] << " ";
   cout << endl;
   reverse(v.begin(), v.end());
   cout << "Reversed: ";
   for (i = 0; i < v.size(); i++)  cout << v[i] << " ";
   return 0;
}
```

```
// An example of the transform algorithm.
#include <iostream>
#include <list>
#include <algorithm>
using namespace std;
// A simple transformation function.
double reciprocal(double i) { return 1.0 / i; // return reciprocal}
int main() {
   list<double> vals;
   int i;
   // put values into list
   for (i = 1; i < 10; i++) vals.push_back((double)i);
   // transform vals
   p = transform(vals.begin(), vals.end(), vals.begin(), reciprocal);
   cout << "Transformed contents of vals:\n";
   p = vals.begin();
   while (p != vals.end()) {
      cout << *p << " ";
      p++;
   }
   return 0;
}
```

Function ref: https://en.cppreference.com/w/cpp/algorithm/transform

Function Objects in STL

- Function Objects or Functors are objects that can be treated as though they are a function or function pointer.
- Function objects are simply classes that define operator( ).
- The STL provides many built-in function objects, such as
  - less, minus, negate, etc.
- It is also possible to define custom function objects.
- Functors that returns only true or false are called predicates

Function Objects - Unary

```cpp
#include <iostream>
#include <algorithm>
#include <vector>
#include <functional> // Required header

int main() {
    std::vector<int> numbers = {5, 2, 8, 1, 9, 4};
    // std::sort normally sorts in ascending order (using std::less by default)
    std::sort(numbers.begin(), numbers.end(),
                                        std::greater<int>());

//Print the sorted vector
    std::cout << "Sorted in descending order: ";
    for (int num : numbers) {
        std::cout << num << " ";
    }
    std::cout << std::endl;
```

```cpp
 // Directly use the function object in a variable or expression
    std::greater<int> is_greater;
    if (is_greater(10, 5)) {
        std::cout << "10 is greater than 5" << std::endl;
    }
    return 0;
}
```

Function ref: https://en.cppreference.com/w/cpp/utility/functional/greater.html

Function Objects – Unary (2)

```
// Use a unary function object.
#include <iostream>
#include <list>
#include <functional>
#include <algorithm>
using namespace std;
int main() {
   list<double> vals;
   int i;
   // put values into list
   for (i = 1; i < 10; i++)  vals.push_back((double)i);
   cout << "Original contents of vals:\n";
   list<double>::iterator p = vals.begin();
   while (p != vals.end())    {
      cout << *p << " ";
      p++;
   }
   cout << endl;
```

```
// use the negate function object
   p = transform(vals.begin(), vals.end(), vals.begin(),
                 negate<double>()); // call function object
   cout << "Negated contents of vals:\n";
   p = vals.begin();
   while (p != vals.end())
   {
      cout << *p << " ";
      p++;
   }
   return 0;
}
```

Function ref: https://en.cppreference.com/w/cpp/utility/functional/negate

Function Objects - Binary

```
// Use a binary function object.
#include <iostream>
#include <list>
#include <functional>
#include <algorithm>
using namespace std;
int main()
{
   list<double> vals;
   list<double> divisors;
   int i;
   // put values into list
   for (i = 10; i < 100; i += 10)  vals.push_back((double)i);
   for (i = 1; i < 10; i++)  divisors.push_back(3.0);
   cout << "Original contents of vals:\n";
   list<double>::iterator p = vals.begin();
   while (p != vals.end())    {
      cout << *p << " ";
      p++;
   }
   cout << endl;
```

```
// transform vals
   p = transform(vals.begin(), vals.end(), divisors.begin(),
                 vals.begin(), divides<double>()); // call
                                                   function object

   cout << "Divided contents of vals:\n";
   p = vals.begin();
   while (p != vals.end())
   {
      cout << *p << " ";
      p++;
   }
   return 0;
}
```

Functor Ref: https://en.cppreference.com/w/cpp/utility/functional/divides

User Defined Function Objects

```
// Create a reciprocal function object.
#include <iostream>
#include <list>
#include <functional>
#include <algorithm>
using namespace std;
// A simple function object.
template <class result_type, class argument_type>
class reciprocal {
public:
   result_type operator()(argument_type i)  {
      return (result_type)1.0 / i; // return reciprocal
   }
};
```

```
int main() {
   list<double> vals;
   int i;
   // put values into list
   for (i = 1; i < 10; i++) vals.push_back((double)i);
   cout << "Original contents of vals:\n";
   list<double>::iterator p = vals.begin();
   while (p != vals.end())  {
      cout << *p << " ";
      p++;
   }
   cout << endl;
   // use reciprocal function object
   p = transform(vals.begin(), vals.end(), vals.begin(),
                 reciprocal<double, double>());
   cout << "Transformed contents of vals:\n";
   p = vals.begin();
   while (p != vals.end())  {
      cout << *p << " ";
      p++;
   }
   return 0;
}
```

Test Question

```
int main() {
    // List 1 with some negative values
    std::list<int> list1 = {5, -3, 7, -8, -2, 10};
    // List 2 to provide replacement values for negative elements in list1
    std::list<int> list2 = {100, 200, 300, 400, 500, 600};
    // Output modified list1
    std::cout << "Modified list1: ";
    for (int value : list1) std::cout << value << “, ";
    std::cout << std::endl;
    return 0;
}
```

Task: Replace all the negative elements of list1 with the corresponding elements of list 2.

Expected Output

```
Modified list1: 5, 200, 7, 400, 500, 10
```

```
template <typename X>
class ReplaceNegative {
public:
    X operator()(X value1, X value2) const {
        return (value1 < 0) ? value2 : value1;
    }
};
```

Strings in C++

- C++ does not support a built-in string type like int, float etc.
- There are two ways of handling strings.
  1. Using the traditional, null terminated character array; sometimes referred to as a C string.

```
char str[] = "Hello world";
```

  2. Using a class object of type string

```
string str("Hello World");
```

Why the String Class

- Null-terminated strings cannot be manipulated by any of the standard C++ operators.
- Nor can they take part in normal C++ expressions.

```
char s1[80], s2[80], s3[80];
s1 = "Alpha"; // can't do
s2 = "Beta"; // can't do
s3 = s1 + s2; // error, not allowed
```

- Using standard string class, it becomes possible to manage strings in the same way as any other type of data: through the use of operators.

```
string s1, s2, s3;
s1 = "Alpha";
s2 = "Beta";
s3 = s1 + s2;
cout << s1 << " " << s2 << " " << s3;
return 0;
```

Why the String Class (2)

- It is very easy to overrun the end of an array that holds a null-terminated string due to programming error.
  - For example, consider the standard string copy function strcpy( ). This function contains no provision for checking the boundary of the target array.
  - If the source array contains more characters than the target array can hold, then a program error or system crash is possible (likely).
- The standard string class prevents such errors.

The Strings Class

- The string class is not part of the STL, but has implemented many STL features.
- string can be treated as a STL container of char.
  - It defines member functions begin(), end(), rbegin(), rend() which returns an iterator for forward and reverse transversal.
  - Most of the algorithms (such as transform(), sort()) are applicable to string, operating on individual characters.
- string is declared in the header <string>

Definition of The Strings Class

For complete list check the link below
https://en.cppreference.com/w/cpp/string/basic_string

- The string class is a specialization of a more general template class called basic_string.

```
template<
    class CharT,
    class Traits = std::char_traits<CharT>,
    class Allocator = std::allocator<CharT>
> class basic_string;
```

- Several *typedefs* for common character types are provided in header <string> such as

Type	Definition
std::string	std::basic_string<char>
std::wstring	std::basic_string<wchar_t>
std::u8string (C++20)	std::basic_string<char8_t>
std::u16string (C++11)	std::basic_string<char16_t>
std::u32string (C++11)	std::basic_string<char32_t>

Basic String Usage

```
#include <iostream>
#include <string>
using namespace std;
int main(){
    string str1("Alpha");
    string str2("Beta");
    string str3("Omega");
    string str4;
    // assign a string
    str4 = str1;
    cout << str1 << "\n"
         << str3 << "\n";
    // concatenate two strings
    str4 = str1 + str2;
    cout << str4 << "\n";
    // concatenate a string with a C-string
    str4 = str1 + " to " + str3;
    cout << str4 << "\n";
```

```
    // compare strings
    if(str3 > str1) cout << "str3 > str1\n";
    if(str3 == str1+str2)
    cout << "str3 == str1+str2\n";
    /* A string object can also be assigned a normal
string. */
    str1 = "This is a null-terminated string.\n";
    cout << str1;
    // create a string object using another string
object
    string str5(str1);
    cout << str5;
    // input a string
    cout << "Enter a string: ";
    cin >> str5;
    cout << str5;
    return 0;
}
```

Some String Member Functions

```
// Demonstrate insert(), erase(), and replace().
#include <iostream>
#include <string>
using namespace std;
int main(){
   string str1("String handling C++ style.");
   string str2("STL Power");
   cout << "Initial strings:\n";
   cout << "str1: " << str1 << endl;
   cout << "str2: " << str2 << "\n\n";
   // demonstrate insert()
    cout << "Insert str2 into str1:\n";
    str1.insert(6, str2);
    cout << str1 << "\n\n";
   // demonstrate erase()
   cout << "Remove 9 characters from str1:\n";
   str1.erase(6, 9);
   cout << str1 << "\n\n";
   // demonstrate replace
   cout << "Replace 8 characters in str1 with str2:\n";
   str1.replace(7, 8, str2);
   cout << str1 << endl;
   return 0;
}
```

```
#include <iostream>
#include <string>
using namespace std;
int main()
{
   int i;
   string s1 ="Quick of Mind, Strong of Body, Pure of Heart";
   string s2;
   i = s1.find("Quick");
   if (i != string::npos)
      cout << "Match found at " << i << endl;
   // find last "of"
   i = s1.rfind("of");
   if (i != string::npos)
   {
      cout << "Match found at " << i << endl;
      cout << "Remaining string is:\n";
      s2.assign(s1, i, s1.size());
      cout << s2;
   }
   return 0;
}
```

String as Containers

```
// Strings as containers.
#include <iostream>
#include <string>
#include <algorithm>
using namespace std;
int main()
{
   string str1("Strings handling is easy in C++");
   string::iterator p;
   unsigned int i;
   // use size()
   for (i = 0; i < str1.size(); i++)
      cout << str1[i];
   cout << endl;
   // use iterator
   p = str1.begin();
   while (p != str1.end())
      cout << *p++;
   cout << endl;
```

```
   // use the count() algorithm
   i = count(str1.begin(), str1.end(), 'i');
   cout << "There are " << i << " i's in str1\n";
   // use transform() to upper case the string
   transform(str1.begin(), str1.end(), str1.begin(), ::toupper);
   p = str1.begin();
   while (p != str1.end())
      cout << *p++;
   cout << endl;
   return 0;
}
```

String in Other Containers

```
// Use a map of strings to create a phone directory.
#include <iostream>
#include <map>
#include <string>
using namespace std;
int main() {
   map<string, string> directory;
   directory.insert(pair<string, string>("Tom", "555-4533"));
   directory.insert(pair<string, string>("Chris", "555-9678"));
   directory.insert(pair<string, string>("John", "555-8195"));
   directory.insert(pair<string, string>("Rachel", "555-0809"));
   string s;
   cout << "Enter name: ";
   cin >> s;
   map<string, string>::iterator p;
   p = directory.find(s);
   if (p != directory.end())
      cout << "Phone number: " << p->second;
   else  cout << "Name not in directory\n";
   return 0;
}
```

Test Question

```
int main() {
    std::vector<std::string> strings = {
        "Sheikh Hasina National Institute of Burn and Plastic Surgery",
        "Sheikh Hasina National Youth Center",
        "Gangachara Sheikh Hasina Bridge",
    };
    // Output modified strings
    for (auto& str : strings) std::cout << str << std::endl;
    return 0;
}
```

Remove the name ‘Sheikh Hasina’ from all the strings and then sort the strings in ascending order

```
void removeText(std::string& str) {
    size_t pos = 0;
    if ((pos = str.find("Sheikh Hasina")) != std::string::npos) {
        str.erase(pos, 14);
    }
}
```

Expected Output

```
Gangachara Bridge
National Institute of Burn and Plastic Surgery
National Youth Center
```

Thank You