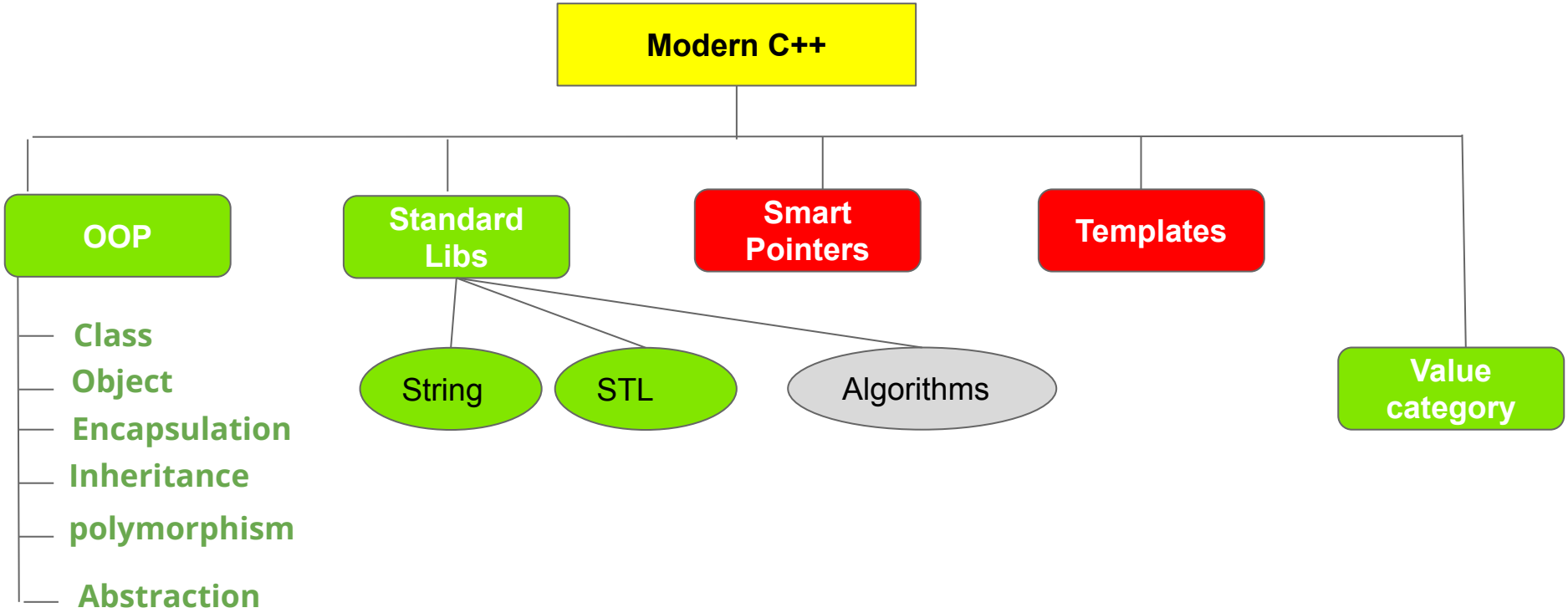
Polymorphism &STL

Moatasem Elsayed

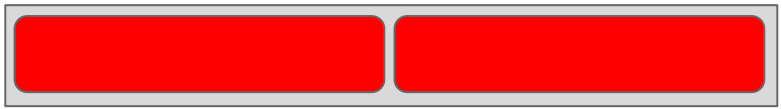
Content

- Inheritance
- Overriding
- Array
- Vector
- Deque
- List
- Fowared_list
- Set /multi/unordered
- map/mult/unordered


Modern C++
OOP
Standard Libs
Smart Pointers
Templates
Value category
Class
Object
Encapsulation
Inheritance
polymorphism
Abstraction
String
STL
Algorithms

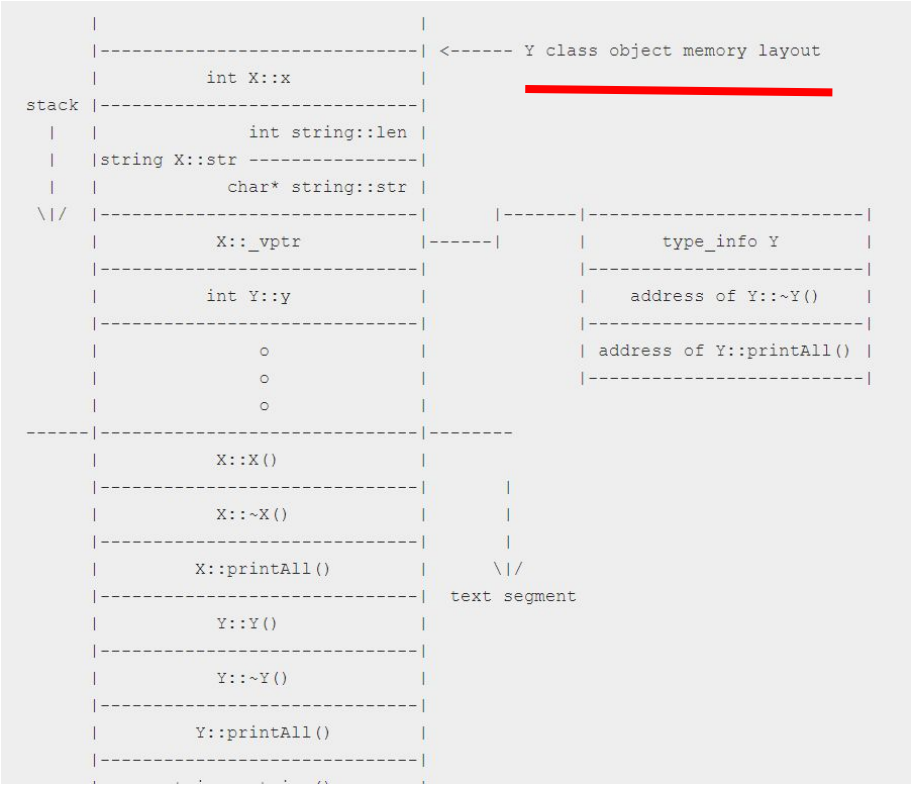
inheritance

The idea of inheritance implements the is a relationship



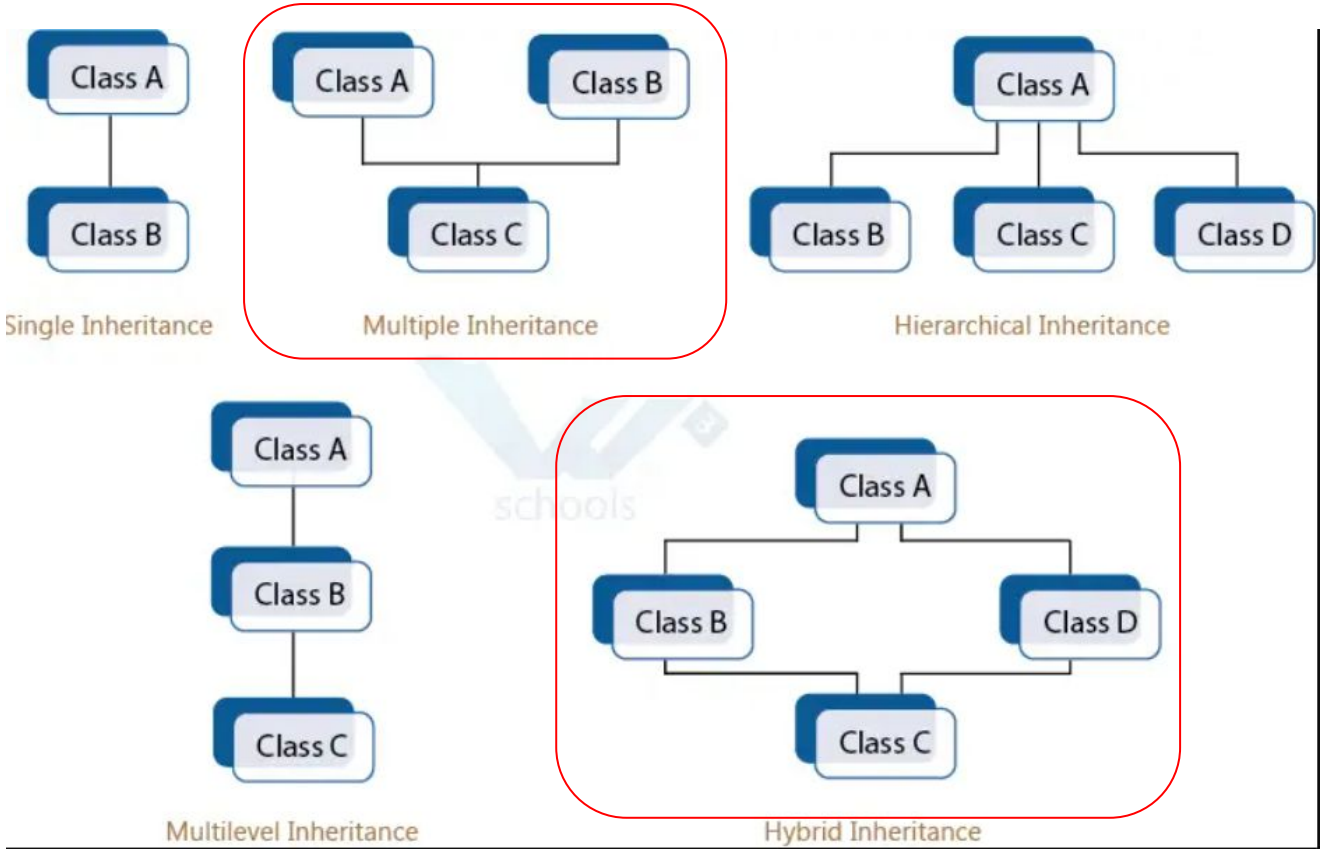
```
// Base class
class Vehicle {
  public:
    string brand = "Ford";
    void honk() {
      cout << "Tuut, tuut! \n" ;
    }
};

// Derived class
class Car: public Vehicle {
  public:
    string model = "Mustang";
};

int main() {
  Car myCar;
  myCar.honk();
  cout << myCar.brand + " " + myCar.model;
  return 0;
}
```

Layout (ignore virtual for now)




```
class X {
    int      x;
    string str;

public:
    X() {}
    virtual ~X() {}

    virtual void printAll() {}
};

class Y : public X {
    int      y;

public:
    Y() {}
    ~Y() {}

    void printAll() {}
};
```

Types of inheritance



Multilevel

```
class Base
{
public:
    void fun()
    {
        std::cout << "Base" << std::endl;
    }
};
class child1 : public Base
{
public:
    void fun()
    {
        std::cout << "child" << std::endl;
    }
};
class child2 : public child1
{
public:
};

int main()
{
    child2 obj;
    obj.fun();       // child
    obj.Base::fun(); // base
}
```

```
// Base class (parent)
class MyClass {
  public:
    void myFunction() {
      cout << "Some content in parent class." ;
    }
};

// Derived class (child)
class MyChild: public MyClass {
};

// Derived class (grandchild)
class MyGrandChild: public MyChild {
};

int main() {
  MyGrandChild myObj;
  myObj.myFunction();
  return 0;
}
```

Multiple inhe

```
class Base
{
public:
    void fun()
    {
        std::cout << "Base" << std::endl;
    }
};
class Base2
{
public:
    void fun()
    {
        std::cout << "child" << std::endl
    }
};
class child2 : public Base, public Base2
{
public:
};

int main()
{
    child2 obj;
    obj.fun(); //  error: request for mem
}
```

```
class Base
{
public:
    void fun()
    {
        std::cout << "Base" << std::endl;
    }
};
class Base2
{
public:
    void fun()
    {
        std::cout << "Base2" << std::endl;
    }
};
class child2 : public Base, public Base2
{
public:
};

int main()
{
    child2 obj;
    obj.Base::fun();  // Base
    obj.Base2::fun(); // Base2
}
```

Hybrid inheritan

```
class ClassA
{
public:
    int a;
};

class ClassB : public ClassA
{
public:
    int b;
};

class ClassC : public ClassA
{
public:
    int c;
};

class ClassD : public ClassB, public ClassC
{
public:
    int d;
};

int main()
{
    ClassD obj;
    // obj.a = 10;              // Statement 1, Error
    // obj.a = 100;             // Statement 2, Error
    obj.ClassB::a = 10;  // Statement 3
    obj.ClassC::a = 100; // Statement 4

}
```

```
class ClassA
{
public:
    int a;
};

class ClassB : virtual public ClassA
{
public:
    int b;
};

class ClassC : virtual public ClassA
{
public:
    int c;
};

class ClassD : public ClassB, public ClassC
{
public:
    int d;
};

int main()
{
    ClassD obj;

    obj.a = 10;  // Statement 3
    obj.a = 100; // Statement 4
}
```

Call function in base

```cpp
class Base
{
public:
    Base()
    {
        std::cout << "Base" << std::endl;
    }
    Base(int x)
    {
        std::cout << "Base (int x)" << std::endl;
    }
    void fun()
    {
        std::cout << "Hello World" << std::endl;
    }
};

class Derived : public Base
{
public:
    Derived(int x)
    {
        std::cout << "Child" << std::endl;
    }
    void fun()
    {
        // I need to call here fun of base without fall in recursevly concept
        Base::fun();
    }
};
```

Constructors

```
7   class Base
8   {
9   public:
10      Base()
11      {
12          std::cout << "Base" << std::endl;
13      }
14  };
15
16  class Derived : public Base
17  {
18  public:
19      Derived()
20      {
21          std::cout << "Child" << std::endl;
22      }
23  };
24
25  int main()
26  {
27      Derived d; // Base // Child
28  }
```

```
7   class Base
8   {
9   public:
10      Base()
11      {
12          std::cout << "Base" << std::endl;
13      }
14      Base(int x)
15      {
16          std::cout << "Base (int x)" << std::endl;
17      }
18  };
19
20  class Derived : public Base
21  {
22  public:
23      Derived(int x) : Base(x)
24      {
25          std::cout << "Child" << std::endl;
26      }
27  };
28
29  int main()
30  {
31      Derived d(3); // Base(int x) // Child
32  }
```

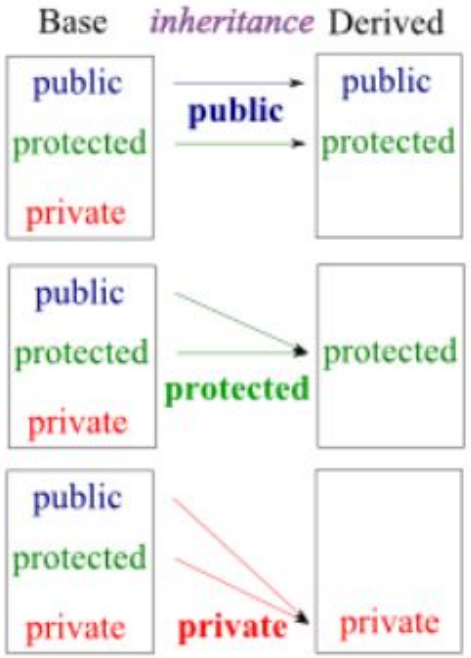
Destructors

```cpp
class Base
{
public:
    Base()
    {
        std::cout << "Base" << std::endl;
    }
    Base(int x)
    {
        std::cout << "Base (int x)" << std::endl;
    }
    ~Base()
    {
        std::cout << "Destructor Base" << std::endl;
    }
};
class Derived : public Base
{
public:
    Derived(int x)
    {
        std::cout << "Child" << std::endl;
    }
    ~Derived()
    {
        std::cout << "Destructor Derived " << std::endl;
    }
};

int main()
{
    Derived d(3); // Base(int x) // Child
    // Destructors //Derived // Base
}
```

```
class Derived : public Base
{
```

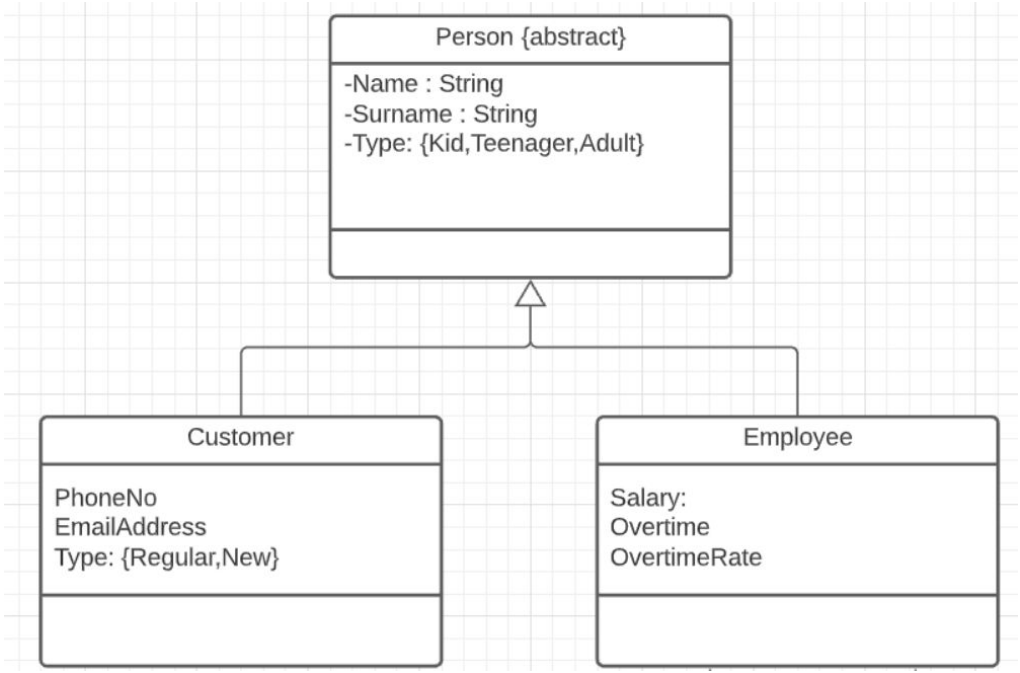
Visibility of Inherited Members

Base class visibility	Derived class visibility		
	Public	**Private**	**Protected**
Private	Not Inherited	Not Inherited	Not Inherited
Protected	Protected	Private	Protected
Public	Public	Private	Protected



In class diagram {inheritance}{Generalization}

```
class Person
{
public:

};
class Customer : public Person
{
public:

};
class Employee : public Person
{
public:

};
```



Function Overriding

The `virtual` keyword creates a **vtable**, enabling

dynamic binding (late binding) of functions.

```cpp
class Shape {
public:
    virtual void display() {
        std::cout << "This is a Shape." << std::endl;
    }
};

class Circle : public Shape {
public:
    void display() override {
        std::cout << "This is a Circle." << std::endl;
    }
};

class Square : public Shape {
public:
    void display() override {
        std::cout << "This is a Square." << std::endl;
    }
};

int main() {
    Shape shape;
    Circle circle;
    Square square;

    Shape* shapePtr;

    shapePtr = &shape;
    shapePtr->display(); // Calls Shape's display()

    shapePtr = &circle;
    shapePtr->display(); // Calls Circle's display()

    shapePtr = &square;
    shapePtr->display(); // Calls Square's display()

    return 0;
}
```

cont..

```cpp
#include <iostream>

class Shape {
public:
    virtual void draw() {
        std::cout << "Drawing a shape." << std::endl;
    }
};

class Circle : public Shape {
public:
    void draw() override {
        std::cout << "Drawing a circle." << std::endl;
    }
};

class Square : public Shape {
public:
    void draw() override {
        std::cout << "Drawing a square." << std::endl;
    }
};

void drawShape(Shape& shape) {
    shape.draw();
}

int main() {
    Circle circle;
    Square square;

    drawShape(&shape: circle); // Draws a circle using polymorphism
    drawShape(&shape: square); // Draws a square using polymorphism

    return 0;
}
```

Interface

```cpp
class Shape {
public:
    virtual void draw() = 0; // Pure virtual function, making Shape an abstract class
};

class Circle : public Shape {
public:
    void draw() override {
        std::cout << "Drawing a circle." << std::endl;
    }
};

class Square : public Shape {
public:
    void draw() override {
        std::cout << "Drawing a square." << std::endl;
    }
};

int main() {
    Circle circle;
    Square square;

    Shape* shapes[] = {[0]=&circle, [1]=&square};

    for (Shape* shape : shapes) {
        shape->draw(); // Using abstraction to draw different shapes
    }

    return 0;
}
```

String

std::string

- dynamic array of `char` (similar to `vector<char>`)
- concatenation with `+` or `+=`
- single character access with `[index]`
- modifiable ("mutable") unlike in e.g., Python or Java
- *regular*: deeply copyable, deeply comparable

```
#include <string>
std::string hw = "Hello";
std::string s = hw;      // copy of hw
hw += " World!";
cout << hw << '\n';      // Hello World!
cout << hw[4] << '\n';   // o
cout << s << '\n';       // Hello
```

```
string s = "I am sorry, Dave.";
indices 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
```

```
s.insert(5, "very ")      ⇒  s = "I am very sorry, Dave."
s.erase(6, 2)             ⇒  s = "I am sry, Dave."
s.replace(12,5,"Frank")   ⇒  s = "I am sorry, Frank"
s.resize(4)               ⇒  s = "I am"
s.resize(20, '?')         ⇒  s = "I am sorry, Dave.???";
```

changes string object

```
s.find("r")     →  7          (first occurrence from begin)
s.rfind("r")    →  8          (first occurrence from end  )
s.find("X")     →  string::npos                 (not found)
s.find('a',5)   →  13         (first occurr. starting at 5)
s.substr(5,6)   →  "sorry,"   (returns new string object)
```

no change to string

Joining

String literals that are only separated by whitespace are joined:

`"first" "second"` ⇒ `"first second"`

```
std::string s =
  "This is one literal"
  "split into several"
  "source code lines!";
```

Raw String Literals

Advantage: special characters can be used without escaping

```
R"(raw "C"-string c:\users\joe)"          char const[]      C++11
R"(raw "std"-string c:\users\moe)"s       std::string       C++14
```

Syntax: `R"DELIMITER(characters…)DELIMITER"`

where `DELIMITER` can be a sequence of 0 to 16 characters except spaces, `(`, `)` and `\`

Task : please use getline to get input from user

std::getline

- read entire lines / chunks of text at once
- target string can be re-used (saving memory)

```
std::string s;
getline(std::cin, s);         // read entire line
getline(std::cin, s, '\t');   // read until next tab
getline(std::cin, s, 'a');    // read until next 'a'
```

STL

Components of STL

STL is divided into three main components:

1. **Containers:** Data structures to store and organize data.
2. **Algorithms:** Functions to perform operations on data stored in containers.
3. **Iterators:** Objects used to traverse and manipulate elements in containers.

Iterators

- **Input Iterators:** Read values from a container.
- **Output Iterators:** Write values to a container.
- **Forward Iterators:** Traverse in one direction.
- **Bidirectional Iterators:** Traverse in both directions.
- **Random Access Iterators:** Supports random access like arrays

Containers

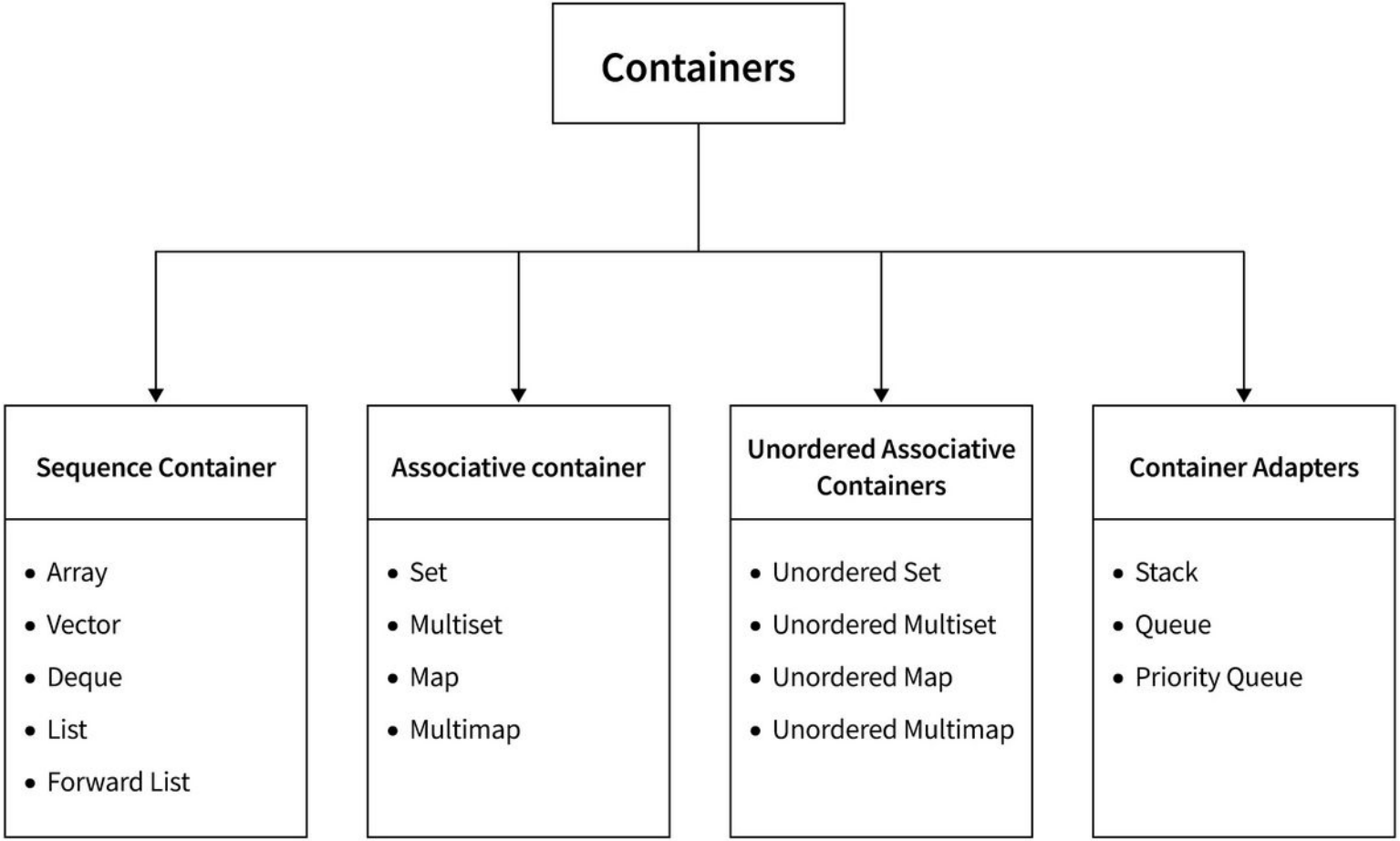
- **Vector:** Dynamic array that can grow or shrink.
- **List:** Doubly-linked list.
- **Deque:** Double-ended queue.
- **Set:** Collection of unique elements.
- **Map:** Key-value pairs, sorted by keys.
- **Stack:** LIFO (Last In, First Out) structure.
- **Queue:** FIFO (First In, First Out) structure.

Algorithms

STL provides a variety of algorithms, including:

- **Sorting:** `sort`, `stable_sort`, `partial_sort`, ...
- **Searching:** `find`, `binary_search`, `lower_bound`, ...
- **Manipulation:** `copy`, `reverse`, `rotate`, `fill`, ...

.


Containers
Sequence Container
Array
Vector
Deque
List
Forward List
Associative container
Set
Multiset
Map
Multimap
Unordered Associative Containers
Unordered Set
Unordered Multiset
Unordered Map
Unordered Multimap
Container Adapters
Stack
Queue
Priority Queue

std::array

array.cpp

```cpp
#include <iostream>
#include <array>

int main() {
    // Declare and initialize an std::array
    std::array<int, 5> numbers = {[0]=10, [1]=20, [2]=30, [3]=40, [4]=50};

    // Accessing elements using [] operator
    std::cout << "Element at index 2: " << numbers[2] << std::endl;

    // Size and maximum size
    std::cout << "Size of array: " << numbers.size() << std::endl;
    std::cout <<  ray: " << numbers.max_size() << std::endl;

    // Front and
    std::cout <<  < numbers.front() << std::endl;
    std::cout << "Last element: " << numbers.back() << std::endl;

    // Iterate through the array using iterators
    std::cout << "Array elements: ";
    for (const auto& num : numbers) {
        std::cout << num << " ";
    }
    std::cout << std::endl;

    // Fill the array with a value
    numbers.fill(0);

    // Check if the array is empty
    std::cout << "Is the array empty? " << (numbers.empty() ? "Yes" : "No") << std::endl;

    return 0;
}
```

string-literal

Type: const char[15]

Size: 15 bytes

Member functions

Iterators

begin	Return iterator to beginning (public member function)
end	Return iterator to end (public member function)
rbegin	Return reverse iterator to reverse beginning (public member function)
rend	Return reverse iterator to reverse end (public member function)
cbegin	Return const_iterator to beginning (public member function)
cend	Return const_iterator to end (public member function)
crbegin	Return const_reverse_iterator to reverse beginning (public member function)
crend	Return const_reverse_iterator to reverse end (public member function)

Capacity

size	Return size (public member function)
max_size	Return maximum size (public member function)
empty	Test whether array is empty (public member function)

Element access

operator[]	Access element (public member function)
at	Access element (public member function)
front	Access first element (public member function)
back	Access last element (public member function)
data	Get pointer to data (public member function)

Modifiers

fill	Fill array with value (public member function)
swap	Swap content (public member function)

std::vector

Vectors are sequence containers representing arrays that can change in size.

Just like arrays, vectors use contiguous storage locations for their elements, which means that their elements can also be accessed using offsets on regular pointers to its elements, and just as efficiently as in arrays. But unlike arrays, their size can change dynamically, with their storage being handled automatically by the container.

```
#include <vector>

std::vector<int> v {2, 4, 5};
v.push_back(6);
v.pop_back();
v[1] = 3;
v.resize(5, 0);

cout << v[2];
for (int x : v)
  cout << x << ' '
```

2	4	5

2	4	5	6

2	4	5

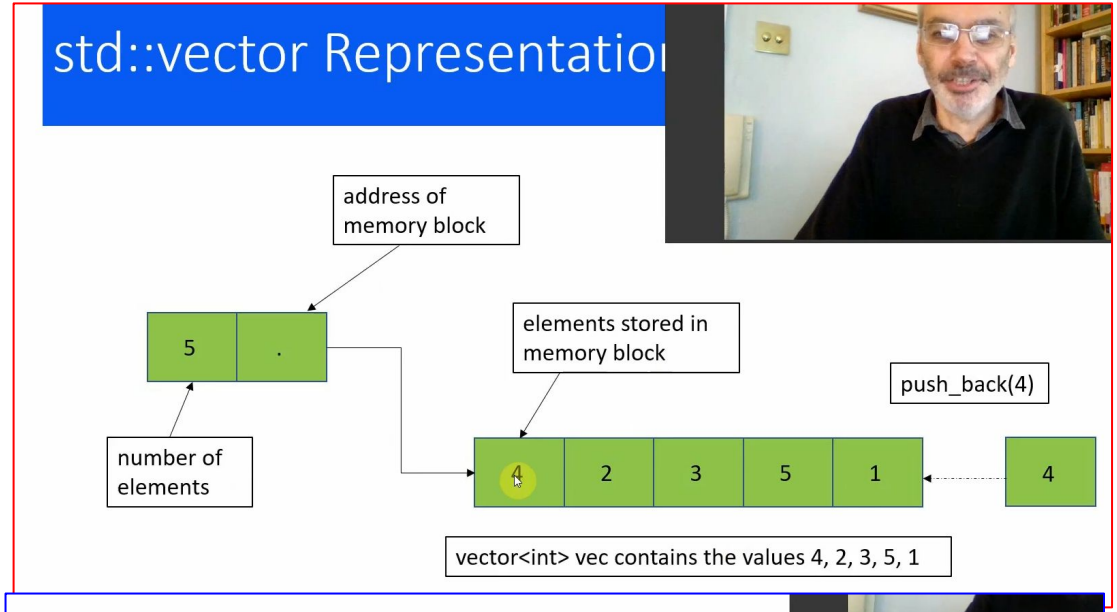
2	3	5

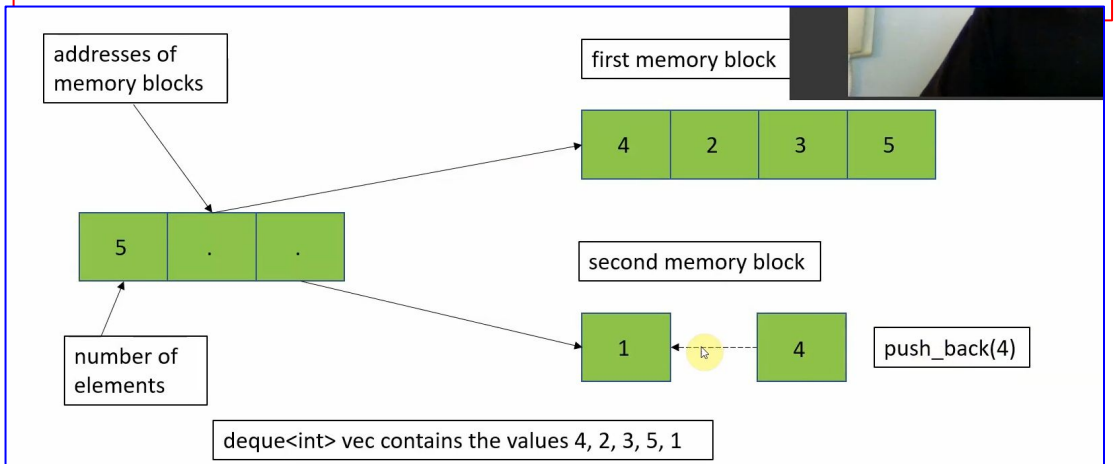
2	3	5	0	0

prints 5

prints 2 3 5 0 0

From Udemy Course





Make New Vector

```
// from list of values: C++11
vector<int> v {0, 1, 2, 3};
// multiple identical values:
vector<int> w (4, 2)
// deep copy:
vector<int> b {v};
```

v: 0 1 2 3

w: 2 2 2 2

b: 0 1 2 3

⚠ Careful!

```
vector<int> v1 {5,2};
```
→ 5 2

```
vector<int> v2 (5,2);
```
→ 2 2 2 2 2

number of elements — default element value

Access Element

```
vector<int> v {2, 4, 6, 8};
// read value at index
cout << v[1];
// assign value at index
v[1] = 3;
```

v: 2 4 6 8

prints 4

v: 2 3 6 8

```
// first element
cout << v.front();
// last element
cout << v.back();
```

prints 2

prints 8

```
vector<int> u {5,7};
vector<int> v {1,2,3};
```

u: 5 7 v: 1 2 3

```
// copy-assign from other
u = v;
```

u: 1 2 3 v: 1 2 3

```
// multiple times same value
v.assign(4, 9);
```

u: 1 2 3 v: 9 9 9 9

Deep copying

`vector` is a so-called *regular* type (it "behaves like `int` ")

- **deep copying**: copying creates a new vector object and copies all contained objects
- **deep assignment**: all contained objects are copied from source to assignment target
- **deep comparison**: comparing two vectors compares the contained objects
- **deep ownership**: destroying a vector destroys all contained objects

Most types in the C++ standard library and ecosystem are *regular*.

```
vector<int> a {1,2,3,4};
vector<int> b = a;  // copy a → b
if (a == b) cout << "equal";
```

```
a: 1 2 3 4
b: 1 2 3 4
equal
```

```
a[0] = 9;
```

```
a: 9 2 3 4
b: 1 2 3 4
```

```
cout << b[0];
if (a != b) cout << "different";
```

```
1
different
```

Iterator

A reverse iterator refers to a position in a vector:

```
vector<int> v {1,2,3,4,5,6,7}
auto i = rbegin(v);
auto e = rend(v);
```

`*i` accesses the element at `i`'s position

```
cout << *i;
cout << *(i+2);
cout << *e;
```

prints 7
prints 5
✗ UNDEFINED BEHAVIOR

```
++i;

cout << *i;
```

prints 6

```
i += 2;

cout << *i;
```

prints 4

```
--i;

cout << *i;
```

prints 5

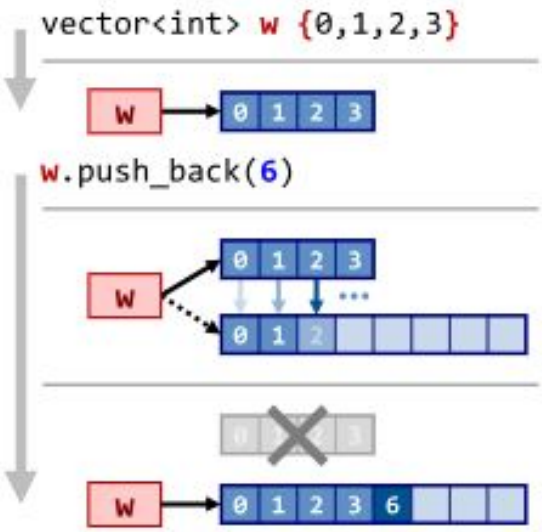
Growth Scheme

Memory blocks, once allocated, can't be resized! (no guarantee that there is space left directly behind previously allocated memory block)

Dynamic array implementations separate the array object from the actual memory block for storing values.

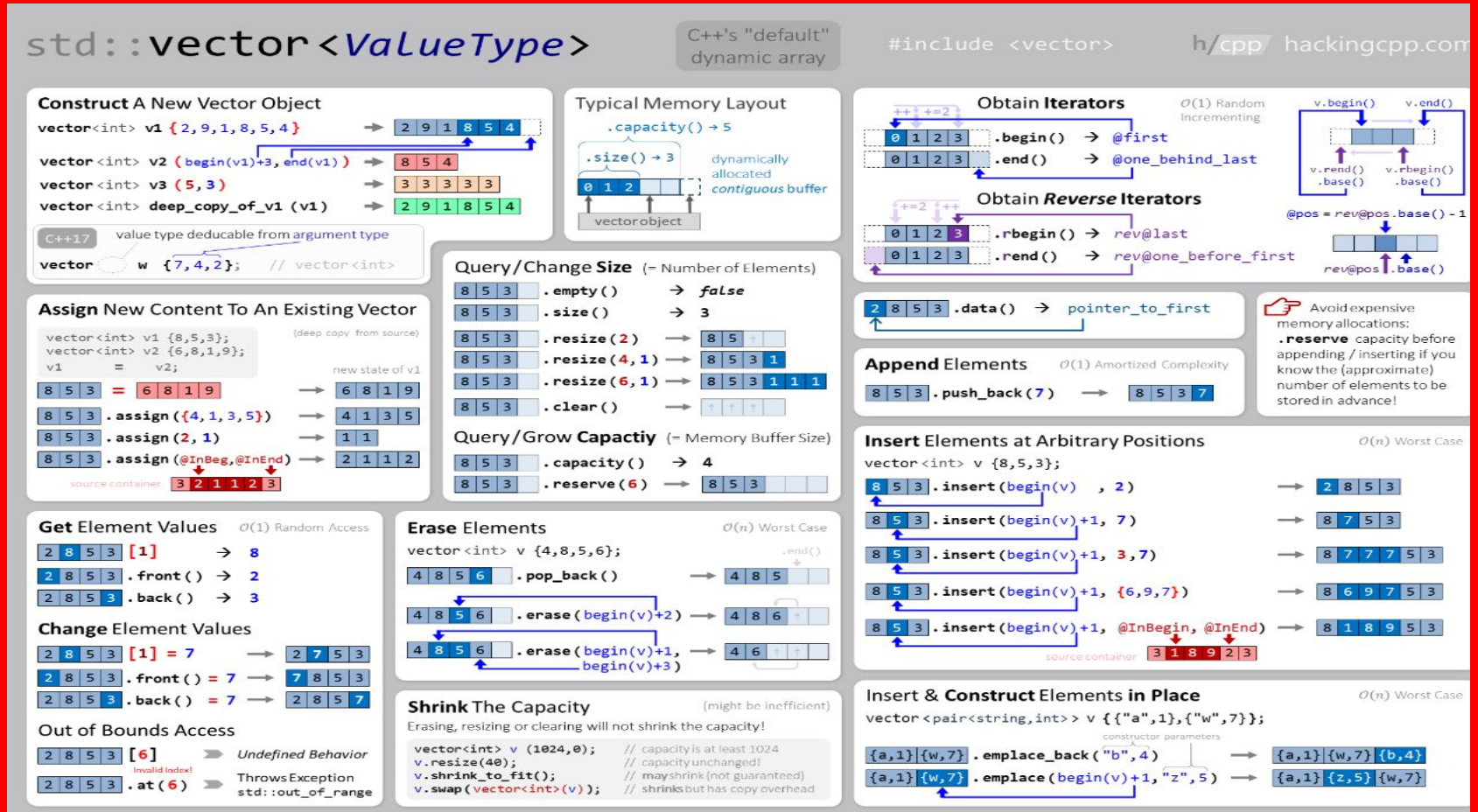
Growth is then done the following way:

- dynamically allocate new, (≈1.1-2×) larger memory block
- copy/move old values to new block
- destroy old, smaller block



W Dynamic Array...

Please try all methods again



std::vector

```cpp
int main() {
    // std::array
    std::array<int, 5> arrayNumbers = {[0]=10, [1]=20, [2]=30, [3]=40, [4]=50};

    // std::vector
    std::vector<int> vectorNumbers = {[0]=10, [1]=20, [2]=30, [3]=40, [4]=50};

    // Accessing elements using []
    std::cout << "Element at index 2 (std::array): " << arrayNumbers[2] << std::endl;
    std::cout << "Element at index 2 (std::vector): " << vectorNumbers[2] << std::endl;

    // Size
    std::cout << "Size of array: " << arrayNumbers.size() << std::endl;
    std::cout << "Size of vector: " << vectorNumbers.size() << std::endl;

    // Front and back elements
    std::cout << "First element of array: " << arrayNumbers.front() << std::endl;
    std::cout << "Last element of vector: " << vectorNumbers.back() << std::endl;

    // Iterating through the collection
    std::cout << "Array elements: ";
    for (const auto& num: const value_type & : arrayNumbers) {
        std::cout << num << " ";
    }
    std::cout << std::endl;

    std::cout << "Vector elements: ";
    for (const auto& num: int const & : vectorNumbers) {
        std::cout << num << " ";
    }
    std::cout << std::endl;

    // Filling with a value
    arrayNumbers.fill(u: 0);
    std::fill(first: vectorNumbers.begin(), last: vectorNumbers.end(), value: 0);

    // Checking if empty
    std::cout << "Is array empty? " << (arrayNumbers.empty() ? "Yes" : "No") << std::endl;
    std::cout << "Is vector empty? " << (vectorNumbers.empty() ? "Yes" : "No") << std::endl;
```

Iterators:

begin	Return iterator to beginning (public member function)
end	Return iterator to end (public member function)
rbegin	Return reverse iterator to reverse beginning (public member function)
rend	Return reverse iterator to reverse end (public member function)
cbegin	Return const_iterator to beginning (public member function)
cend	Return const_iterator to end (public member function)
crbegin	Return const_reverse_iterator to reverse beginning (public member funct
crend	Return const_reverse_iterator to reverse end (public member function)

Capacity:

size	Return size (public member function)
max_size	Return maximum size (public member function)
resize	Change size (public member function)
capacity	Return size of allocated storage capacity (public member function)
empty	Test whether vector is empty (public member function)
reserve	Request a change in capacity (public member function)
shrink_to_fit	Shrink to fit (public member function)

Element access:

operator[]	Access element (public member function)
at	Access element (public member function)
front	Access first element (public member function)
back	Access last element (public member function)
data	Access data (public member function)

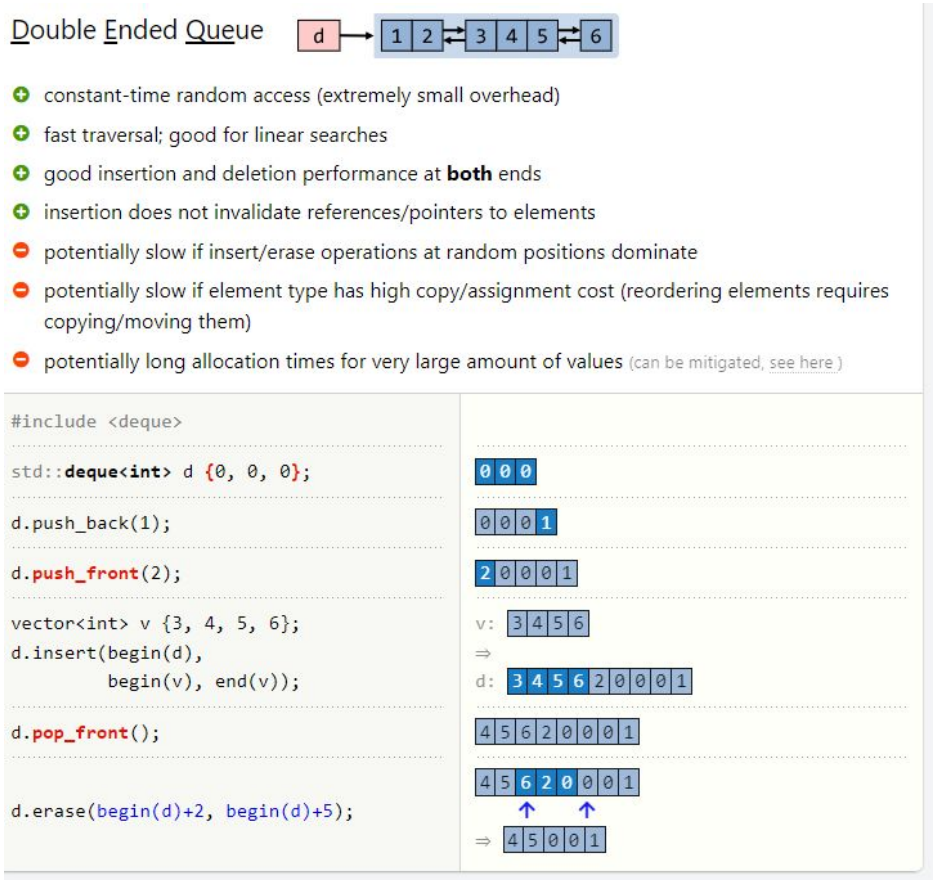
Modifiers:

assign	Assign vector content (public member function)
push_back	Add element at the end (public member function)
pop_back	Delete last element (public member function)
insert	Insert elements (public member function)
erase	Erase elements (public member function)
swap	Swap content (public member function)
clear	Clear content (public member function)
emplace	Construct and insert element (public member function)
emplace_back	Construct and insert element at the end (public member function)

std::deque

A std::deque (short for "double-ended queue") is a container in C++ that provides dynamic array-like functionality with **efficient insertion and deletion operations at both the beginning and the end of the container.**

It can be thought of as a **hybrid between a dynamic array and a linked list**. std::deque is part of the C++ Standard Library and is defined in the <deque> header.



```cpp
#include <iostream>

int main()
{
    std::deque<int> deque;

    // Push elements to the back
    deque.push_back(1);
    deque.push_back(2);
    deque.push_back(3);
    // Push elements to the front
    deque.push_front(0);
    // Print elements using ranged-based for loop
    std::cout << "Elements in the deque: ";
    for (int num : deque)
    {
        std::cout << num << " ";
    }
    std::cout << std::endl;
    // Access elements using indexing
    std::cout << "First element: " << deque[0] << std::endl;
    std::cout << "Last element: " << deque.back() << std::endl;
    // Insert an element at a specific position
    std::deque<int>::iterator insertPos = deque.begin() + 2;
    deque.insert(insertPos, 99);
    // Erase an element at a specific position
    std::deque<int>::iterator erasePos = deque.begin() + 1;
    deque.erase(erasePos);
    // Print elements after insertion and erasure
    std::cout << "Updated elements: ";
    for (int num : deque)
    {
        std::cout << num << " ";
    }
    std::cout << std::endl;
    // Check if the deque is empty
    if (deque.empty())
    {
        std::cout << "The deque is empty." << std::endl;
    }
    else
    {
        std::cout << "The deque is not empty." << std::endl;
        std::cout << "Size of the deque: " << deque.size() << std::endl;
    }

    // Clear all elements from the deque
    deque.clear();
    std::cout << "Cleared the deque. Size: " << deque.size() << std::endl;
```

deque

fx Member functions

(constructor)	Construct deque container (public member function)
(destructor)	Deque destructor (public member function)
operator=	Assign content (public member function)

Iterators:

begin	Return iterator to beginning (public member function)
end	Return iterator to end (public member function)
rbegin	Return reverse iterator to reverse beginning (public member function)
rend	Return reverse iterator to reverse end (public member function)
cbegin	Return const_iterator to beginning (public member function)
cend	Return const_iterator to end (public member function)
crbegin	Return const_reverse_iterator to reverse beginning (public member function)
crend	Return const_reverse_iterator to reverse end (public member function)

Capacity:

size	Return size (public member function)
max_size	Return maximum size (public member function)
resize	Change size (public member function)
empty	Test whether container is empty (public member function)
shrink_to_fit	Shrink to fit (public member function)

Element access:

operator[]	Access element (public member function)
at	Access element (public member function)
front	Access first element (public member function)
back	Access last element (public member function)

Modifiers:

assign	Assign container content (public member function)
push_back	Add element at the end (public member function)
push_front	Insert element at beginning (public member function)
pop_back	Delete last element (public member function)
pop_front	Delete first element (public member function)
insert	Insert elements (public member function)
erase	Erase elements (public member function)
swap	Swap content (public member function)
clear	Clear content (public member function)
emplace	Construct and insert element (public member function)
emplace_front	Construct and insert element at beginning (public member function)

```cpp
std::list<int> myList;
// Push elements to the back
myList.push_back(1);
myList.push_back(2);
myList.push_back(3);
// Push elements to the front
myList.push_front(0);
// Print elements using ranged-based for loop
std::cout << "Elements in the list: ";
for (int num : myList)
{
    std::cout << num << " ";
}
std::cout << std::endl;
// Access elements using iterators
std::list<int>::iterator it = myList.begin();
std::advance(it, 2);
std::cout << "Third element: " << *it << std::endl;
// Insert an element at a specific position
std::list<int>::iterator insertPos = myList.begin();
std::advance(insertPos, 2);
myList.insert(insertPos, 99);
// Erase an element at a specific position
std::list<int>::iterator erasePos = myList.begin();
std::advance(erasePos, 1);
myList.erase(erasePos);
// Print elements after insertion and erasure
std::cout << "Updated elements: ";
for (int num : myList)
{
    std::cout << num << " ";
}
std::cout << std::endl;
// Check if the List is empty
if (myList.empty())
{
    std::cout << "The list is empty." << std::endl;
}
else
{
    std::cout << "The list is not empty." << std::endl;
    std::cout << "Size of the list: " << myList.size() << std::endl;
}

// Clear all elements from the list
myList.clear();
std::cout << "Cleared the list. Size: " << myList.size() << std::endl;
```

List DL

begin	Return iterator to beginning (public member function)
end	Return iterator to end (public member function)
rbegin	Return reverse iterator to reverse beginning (public member function
rend	Return reverse iterator to reverse end (public member function)
cbegin	Return const_iterator to beginning (public member function)
cend	Return const_iterator to end (public member function)
crbegin	Return const_reverse_iterator to reverse beginning (public membe
crend	Return const_reverse_iterator to reverse end (public member functio

Capacity:

empty	Test whether container is empty (public member function)
size	Return size (public member function)
max_size	Return maximum size (public member function)

Element access:

front	Access first element (public member function)
back	Access last element (public member function)

Modifiers:

assign	Assign new content to container (public member function)
emplace_front	Construct and insert element at beginning (public member function)
push_front	Insert element at beginning (public member function)
pop_front	Delete first element (public member function)
emplace_back	Construct and insert element at the end (public member function)
push_back	Add element at the end (public member function)
pop_back	Delete last element (public member function)
emplace	Construct and insert element (public member function)

Forward List(single linked list)

```cpp
std::forward_list<int> forwardList;
// Push elements to the front
forwardList.push_front(3);
forwardList.push_front(2);
forwardList.push_front(1);
// Print elements using ranged-based for loop
std::cout << "Elements in the forward_list: ";
for (int num : forwardList)
{
    std::cout << num << " ";
}
std::cout << std::endl;
// Insert an element after a specific position
std::forward_list<int>::iterator insertPos = forwardList.begin();
std::advance(insertPos, 1);
forwardList.insert_after(insertPos, 99);
// Erase an element after a specific position
std::forward_list<int>::iterator erasePos = forwardList.begin();
std::advance(erasePos, 2);
forwardList.erase_after(erasePos);
// Print elements after insertion and erasure
std::cout << "Updated elements: ";
for (int num : forwardList)
{
    std::cout << num << " ";
}
std::cout << std::endl;
// Check if the forward_list is empty
if (forwardList.empty())
{
    std::cout << "The forward_list is empty." << std::endl;
}
else
{
    std::cout << "The forward_list is not empty." << std::endl;
    std::cout << "Size of the forward_list: " << std::distance(forwardList.begin(), forwardList.end()) << std::endl;
}

// Clear all elements from the forward_list
forwardList.clear();
std::cout << "Cleared the forward_list. Size: " << std::distance(forwardList.begin(), forwardList.end())
          << std::endl;
```

end	Return iterator to end (public member function)
cbefore_begin	Return const_iterator to before beginning (public member functio
cbegin	Return const_iterator to beginning (public member function)
cend	Return const_iterator to end (public member function)

Capacity

empty	Test whether array is empty (public member function)
max_size	Return maximum size (public member function)

Element access

front	Access first element (public member function)

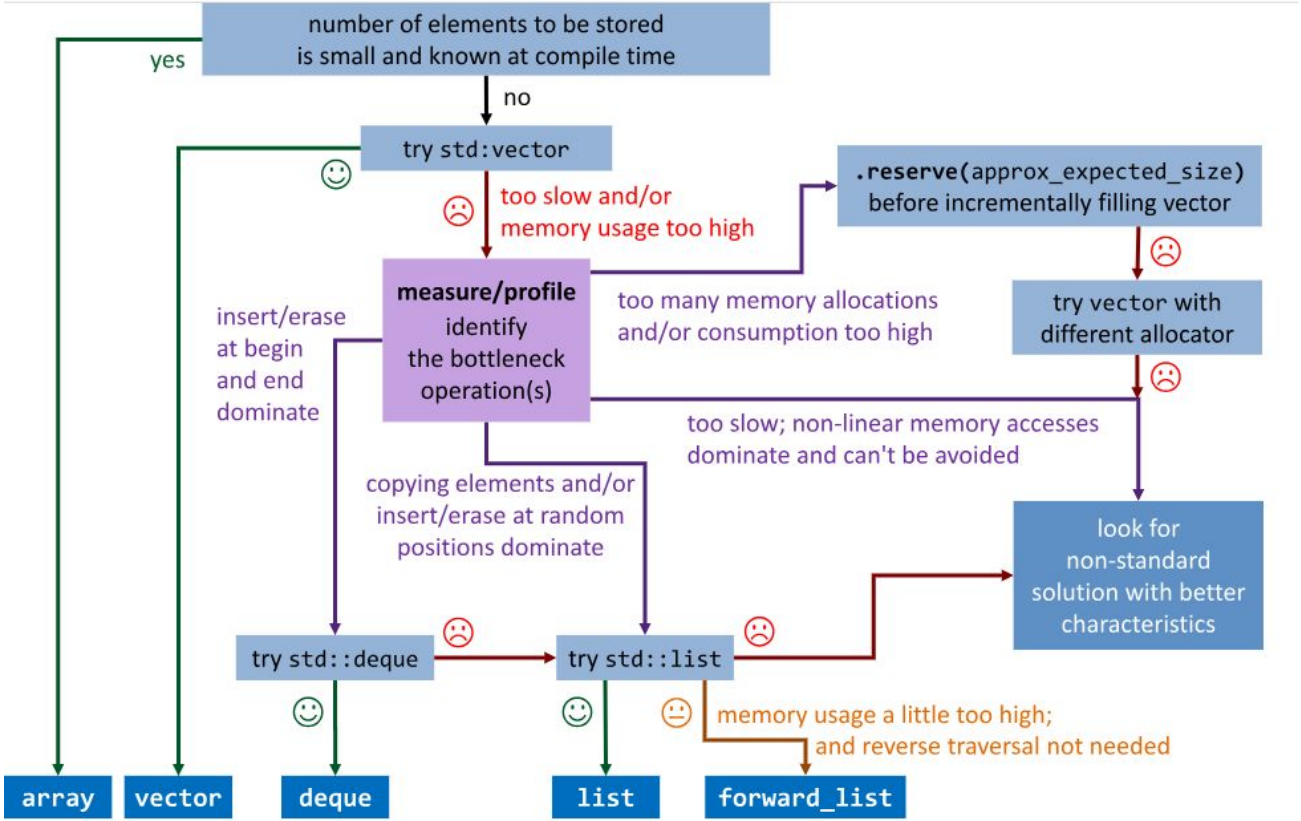
Modifiers

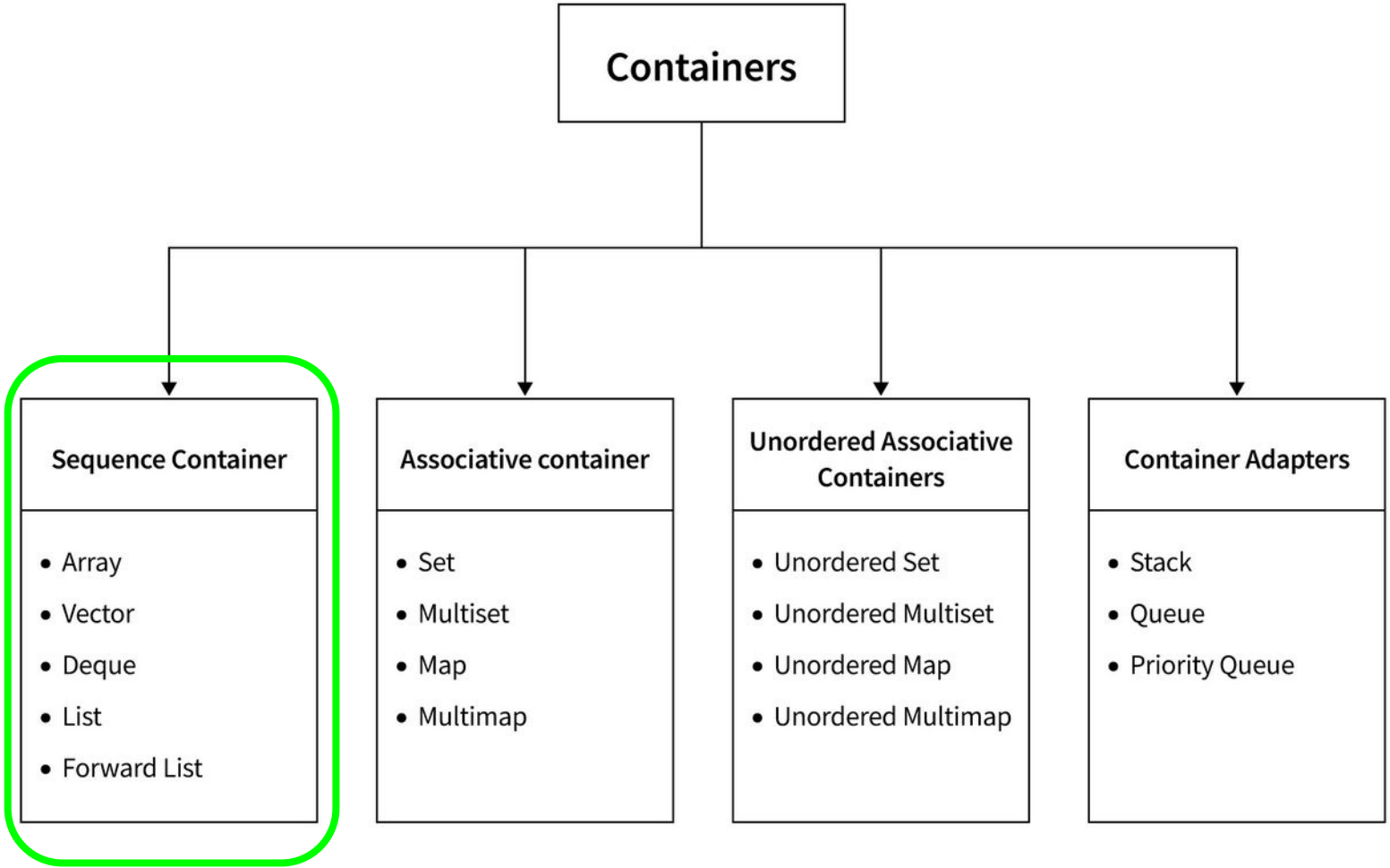
assign	Assign content (public member function)
emplace_front	Construct and insert element at beginning (public member functi
push_front	Insert element at beginning (public member function)
pop_front	Delete first element (public member function)
emplace_after	Construct and insert element (public member function)
insert_after	Insert elements (public member function)
erase_after	Erase elements (public member function)
swap	Swap content (public member function)
resize	Change size (public member function)
clear	Clear content (public member function)

Operations

splice_after	Transfer elements from another forward_list (public member fun
remove	Remove elements with specific value (public member function)

Which Sequence Container Should I Use?



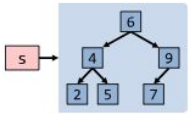

Containers
Sequence Container
Array
Vector
Deque
List
Forward List
Associative container
Set
Multiset
Map
Multimap
Unordered Associative Containers
Unordered Set
Unordered Multiset
Unordered Map
Unordered Multimap
Container Adapters
Stack
Queue
Priority Queue

Set

Sets

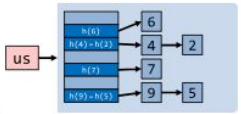
Ordered Sets

```
#include <set>
```



Hash Sets

```
#include <unordered_set>
```



set<*Key*> / unordered_set<*Key*>

unique orderable / hashable keys

```
std::set<int> s {9,2,8};
```
{2, 8, 9}

```
s.insert(7);
s.erase(8);
```
{2, 7, 8, 9}
{2, 7, 9}

```
if (s.find(7) != end(s)) {…}
if (s.contains(7)) {…}  C++20
```
true (7 found)
true (7 found)

```
// find returns an iterator:
auto i = s.find(7);
if (i != end(s)) i = s.erase(i);
```
{2, 7, 9}
  ↑i
{2,9}
  ↑i (after)

multiset<*Key*> / unordered_*multiset*<*Key*>

multiple equivalent keys possible

```
std::multiset<int> s;
```
{}

```
s.insert(8);
s.insert(7);
s.insert(2);
s.insert(7);
s.erase(7);
```
{8}
{7, 8}
{2, 7, 8}
{2, 7, 7, 8}
{2, 8}

Set (sorted)(binary search)

```cpp
{
    std::set<int> mySet;
    // Insert elements
    mySet.insert(3);
    mySet.insert(1);
    mySet.insert(4);
    mySet.insert(2);
    // Print elements using ranged-based for loop
    std::cout << "Elements in the set: ";
    for (int num : mySet)
    {
        std::cout << num << " ";
    }
    std::cout << std::endl;
    // Find an element
    auto findIt = mySet.find(2);
    if (findIt != mySet.end())
    {
        std::cout << "Found: " << *findIt << std::endl;
    }
    // Check if an element exists
    bool exists = mySet.count(3) > 0;
    std::cout << "Element 3 exists: " << (exists ? "Yes" : "No") << std::endl;

    std::multiset<int> myMultiset;
    // Insert elements
    myMultiset.insert(3);
    myMultiset.insert(1);
    myMultiset.insert(4);
    myMultiset.insert(2);
    myMultiset.insert(2); // Inserting duplicate element
    // Print elements using ranged-based for loop
    std::cout << "Elements in the multiset: ";
    for (int num : myMultiset)
    {
        std::cout << num << " ";
    }
    std::cout << std::endl;
    // Count occurrences of an element
    int count = myMultiset.count(2);
    std::cout << "Count of element 2: " << count << std::endl;
    return 0;
}
```

Iterators:

begin	Return iterator to beginning (public member function)
end	Return iterator to end (public member function)
rbegin	Return reverse iterator to reverse beginning (public member function)
rend	Return reverse iterator to reverse end (public member function)
cbegin	Return const_iterator to beginning (public member function)
cend	Return const_iterator to end (public member function)
crbegin	Return const_reverse_iterator to reverse beginning (public member function)
crend	Return const_reverse_iterator to reverse end (public member function)

Capacity:

empty	Test whether container is empty (public member function)
size	Return container size (public member function)
max_size	Return maximum size (public member function)

Modifiers:

insert	Insert element (public member function)
erase	Erase elements (public member function)
swap	Swap content (public member function)
clear	Clear content (public member function)
emplace	Construct and insert element (public member function)
emplace_hint	Construct and insert element with hint (public member function)

Observers:

key_comp	Return comparison object (public member function)
value_comp	Return comparison object (public member function)

Operations:

find	Get iterator to element (public member function)
count	Count elements with a specific value (public member function)

std::set<KeyType, Compare>

(unique keys)

#include <set>

h/cpp hackingcpp.com

default: std::less<KeyType>

std::multiset<KeyType, Compare>

(multiple equivalent keys)

- keys are ordered according to their values
- keys are compared / matched based on *equivalence*: a and b are equivalent if neither is ordered before the other, e.g., if `not (a < b) and not (b < a)`
- default ordering comparator is `std::less`
- sets are usually implemented as a balanced binary tree (e.g., as red-black-tree)

Construct A New Set Object

```
set<int> s0 {}
set<int> s1 {2,1,8,5,4}
set<int> s2 (begin(s1)+2, end(s1))
set<int> deep_copy_of_s1 (s1)
```

C++17 key type deducable from argument type

```
set s3 {7,2,4}; // set<int>
```

Assign New Content To An Existing Set

(deep copy from source)

```
set<int> s1 {1,3,5,7};
set<int> s2 {4,6,8};
s1 = s2;
```

new state of s1

Key Lookup

O(log n)

```
.contains(2) → false
.contains(5) → true
```

C++20

```
.count(2) → 0
.count(5) → 1
```

can be > 1 only for std::multiset

```
.find(2) → @end (= no match)
.find(5) → @match
.lower_bound(3) → @first_not_smaller
.upper_bound(3) → @first_greater
.equal_range(3) → {@Lbound, @Ubound}
```

Query Size (= Number of Keys)

```
.empty() → false
.size() → 3
```

Erase All Keys

```
.clear()
```

Insert A Single Key

O(log n)

```
.insert(4) → {@inserted, true}
.insert(5) → {@blocking, false}
```

(not inserted)

potential performance benefit by hinting at probable insert position

```
.insert(@hint, 7) → @inserted_or_block
```

Insert Multiple Keys

O(#inserted · log n)

```
.insert({1,6,8})
.insert(@in0, @in1)
```

source container

Insert & Construct A Key in Place

O(log n)

```
set<pair<int,int>> s {{1,3},{5,6}};
.emplace(4,7) → {@inserted, true}
```

(if inserted)

pair constructor arguments

potential performance benefit by hinting at probable insert position

```
.emplace_hint(@hint, 4, 7) → @inserted
```

Erase One Key or A Range of Keys

O(log n)

```
.erase(2) → 0
.erase(5) → 1
```

#erased

O(1) amortized

```
.erase(@pos) → @after_erased
.erase(@beg, @end) → @after_erased
```

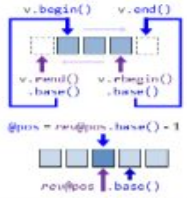
O(log n + #erased)

Obtain Iterators

```
.begin() → @first
.end() → @one_behind_last
```

Obtain *Reverse* Iterators

```
.rbegin() → rev@last
.rend() → rev@one_before_first
```



Extract Nodes

C++17

Allows efficient key modification and transfer of keys between different set objects.

```
.extract(5) → 5 O(log n)
.extract(@pos) → 5 O(1)
```

Merge Two Sets

C++17

```
set<int> s1 {1,3,5,7};
set<int> s2 {2,5,8};
.merge( )
```

$O(n_1 \cdot \log(n_1 + n_2))$

Insert Nodes

C++17

members of the return type

```
.insert(5) → {.position|.inserted|.node}
.insert(1) → {.position|.inserted|.node}
```

(empty node)

```
.insert( ) → {.position|.inserted|.node}
.insert(@hint, 5) → @inserted
.insert(@hint, 1) → @blocking
```

Modify Key

Direct modification not allowed! Instead: extract key, modify its value and re-insert.

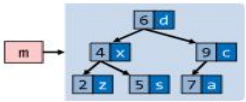
```
set<int> s {1,5,7};
auto node = s.extract(5);
if (node) {  // if key existed
  node.value() = 8;
  s.insert(std::move(node));
}
```

std::map

Key→Value Maps

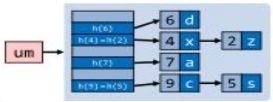
Ordered Key→Value Maps

`#include <map>`



Hashed Key→Value Maps

`#include <unordered_map>`



Maps store `std::pair<Key const,Value>`

map<Key,Value> / unordered_map<Key,Value>

unique orderable / hashable keys

```
std::map<int,std::string> m;
```
{ }

```
m.insert({2, "B"});
m.emplace(1, "A");
```
{2:B}
{1:A,2:B}

```
m[2] = "Y";  // modify
m[3] = "C";  // insert!
```
{1:A,2:Y}
{1:A,2:Y,3:C}

```
auto i = m.find(2);
if (i != end(m))
  cout << i->first
       << i->second;
```
→ iterator
if found
2 (key)
Y (value)

```
if (m.contains(2)) {…}  C++20
```
true (2 found)

```
m.erase(2);
auto j = m.find(3);
if (j != end(m)) j = m.erase(j);
```
{1:A,3:C}
↑j
{1:A }
↑j (after)

```
// C++17 features:
m.insert_or_assign(4,"D");
m.insert_or_assign(1,"X");
m.try_emplace(4,"Z");
m.try_emplace(5,"E");
```
{1:A}
{1:A,4:D}
{1:X,4:D}
{1:X,4:D}
{1:X,4:D,5:E}

```
std::multimap<int,std::string> m;
```
{ }

```
m.emplace(1, "A");
m.insert({2, "B"});
m.emplace(1, "C");
m.erase(1);
```
{1:A}
{1:A,2:B}
{1:A,1:C,2:B}
{ 2:B}

Cont ..

asem_course > map.cpp

```cpp
#include <iostream>
#include <map>
#include <string>

int main()
{
    std::map<int, std::string> myMap;

    // Insert key-value pairs
    myMap[1] = "One";
    myMap[2] = "Two";
    myMap[3] = "Three";

    // Print elements using ranged-based for loop
    std::cout << "Elements in the map:" << std::endl;
    for (const auto& pair : myMap)
    {
        std::cout << pair.first << ": " << pair.second << std::endl;
    }

    // Find an element by key
    int keyToFind = 2;
    std::map<int, std::string>::iterator findIt = myMap.find(keyToFind);
    if (findIt != myMap.end())
    {
        std::cout << "Value for key " << keyToFind << ": " << findIt->second << std::endl;
    }
    else
    {
        std::cout << "Key not found." << std::endl;
    }

    // Check if a key exists
    bool keyExists = myMap.count(4) > 0;
    std::cout << "Key 4 exists: " << (keyExists ? "Yes" : "No") << std::endl;

    return 0;
}
```

Construct A New Map Object

```
map<int,string> m0 { }
map<int,string> m1 {{4,"Z"},{2,"A"},{7,"Y"}}
map<int,string> m2 ( begin(m1)+1, end(m1) )
map<int,string> deep_copy_of_m1(m1)
```

C++17 key and mapped types deducable from arguments

```
map m3 {{2,3.14},{5,6.0}};   // map<int,double>
```

Insert A Single Key-Value Pair

$\mathcal{O}(\log n)$

```
.insert({2,"W"}) → {@inserted,true}
.insert({3,"X"}) → {@blocking,false}
```

potential performance benefit by hinting at probable insert position

```
.insert(@hint,{2,"W"}) → @ins/block
```

Obtain Iterators

```
.begin() → @first
.end()   → @one_behind_last
```

Obtain *Reverse* Iterators

```
.rbegin() → rev@last
.rend()   → rev@one_before_1st
```

v.begin() v.end() v.rend().base() v.rbegin().base()

@pos = rev@pos.base() - 1

rev@pos.base()

- key-value pairs are ordered by key
- key matching is *equivalence*-based: 2 keys a and b are equivalent if not (a < b) and not (b < a)
- default key comparator is std::less
- maps are usually implemented as a balanced binary tree (e.g., as red-black-tree)

map object

Assign New Content To An Existing Map

```
map<int,string> m1 {{2,"X"}};
map<int,string> m2 {{1,"A"},{4,"G"}};
m1 = m2;
```

(deep copy from source)

new state of m1

Insert Multiple Key-Value Pairs

$\mathcal{O}\left(\#\text{inserted} \cdot \log n\right)$

```
.insert({{5,"K"},{3,"Y"},{1,"G"}})
.insert(@inB, @inE)
```

source container

Access / Modify Value

$\mathcal{O}(\log n)$

```
map<int,string> m {{1,"F"},{3,"A"}};
[3] → "A"
[3] = "X"
```

Attention: $[k]$ inserts new pair if key k is not present!

```
[2] = "W"
[2] → ""
```

Extract Nodes

Allows efficient transfer of key-value pairs.

C++17

```
.extract(2) →
.extract(@pos) →
```

$\mathcal{O}(\log n)$ $\mathcal{O}(1)$

(Re-)Insert Nodes

members of the return type

```
.insert(5 N) → {.position|.inserted|.node}
.insert(3 Z) → {.position|.inserted|.node}
.insert( ) → {.position|.inserted|.node}
.insert(@hint,5 X) → @inserted
.insert(@hint,1 G) → @blocking
```

(empty node) (empty)

Lookup Using Keys as Input

$\mathcal{O}(\log n)$

```
map<int,string> m {{3,"A"},{5,"X"},{1,"F"}};
.contains(2) → false
.contains(5) → true
```

C++20

```
.count(2) → 0
.count(5) → 1
```

can be > 1 only for std::multimap

```
.find(2) → @end   (= no match)
.find(5) → @match
.lower_bound(3) → @1st_not_smaller
.upper_bound(3) → @1st_greater
.equal_range(3) → {@Lower,@Upper}
.at(3) → "A"
.at(2) → Throws Exception std::out_of_range
```

Insert & Construct Key-Value Pair

$\mathcal{O}(\log n)$

```
.emplace(2,"W") → {@inserted,true}
```

potential performance benefit by hinting at probable insert position

```
.emplace_hint(@hint,2,"W") → @inserted
.try_emplace(2,"W") → {@inserted,true}
```

C++17

advantage: does not move from rvalue input parameters if not inserted

Insert or Assign Value

$\mathcal{O}(\log n)$ C++17

```
.insert_or_assign(3,"X") → {@as,false}
.insert_or_assign(5,"R") → {@ins,true}
.insert_or_assign(@hint,3,"W") → @as
.insert_or_assign(@hint,2,"G") → @ins
```

Erase Key-Value-Pair(s)

```
.erase(2) → 0
.erase(3) → 1
```

$\mathcal{O}(\log n)$ #erased

$\mathcal{O}(1)$ amortized

```
.erase(@pos) → @after_erased
.erase(@b,@e) → @after_erased
```

$\mathcal{O}(\log n + \#\text{deleted})$

Merge Two Maps

$\mathcal{O}(n_2 \cdot \log(n_1 + n_2))$ C++17

```
map<int,string> m1 {{1,"F"},{3,"S"},{5,"T"}};
map<int,string> m2 {{2,"A"},{5,"X"}};
.merge(2 A 5 X)
```

Modify Key

Direct *key* modification **not** allowed!

Instead:
- extract
- modify
- re-insert

```
map<int,string> m {{1,"F"},{3,"A"}};
auto node = m.extract(3);
if (node) {  // if key existed
  node.key() = 8;
  m.insert(move(node));
}
```

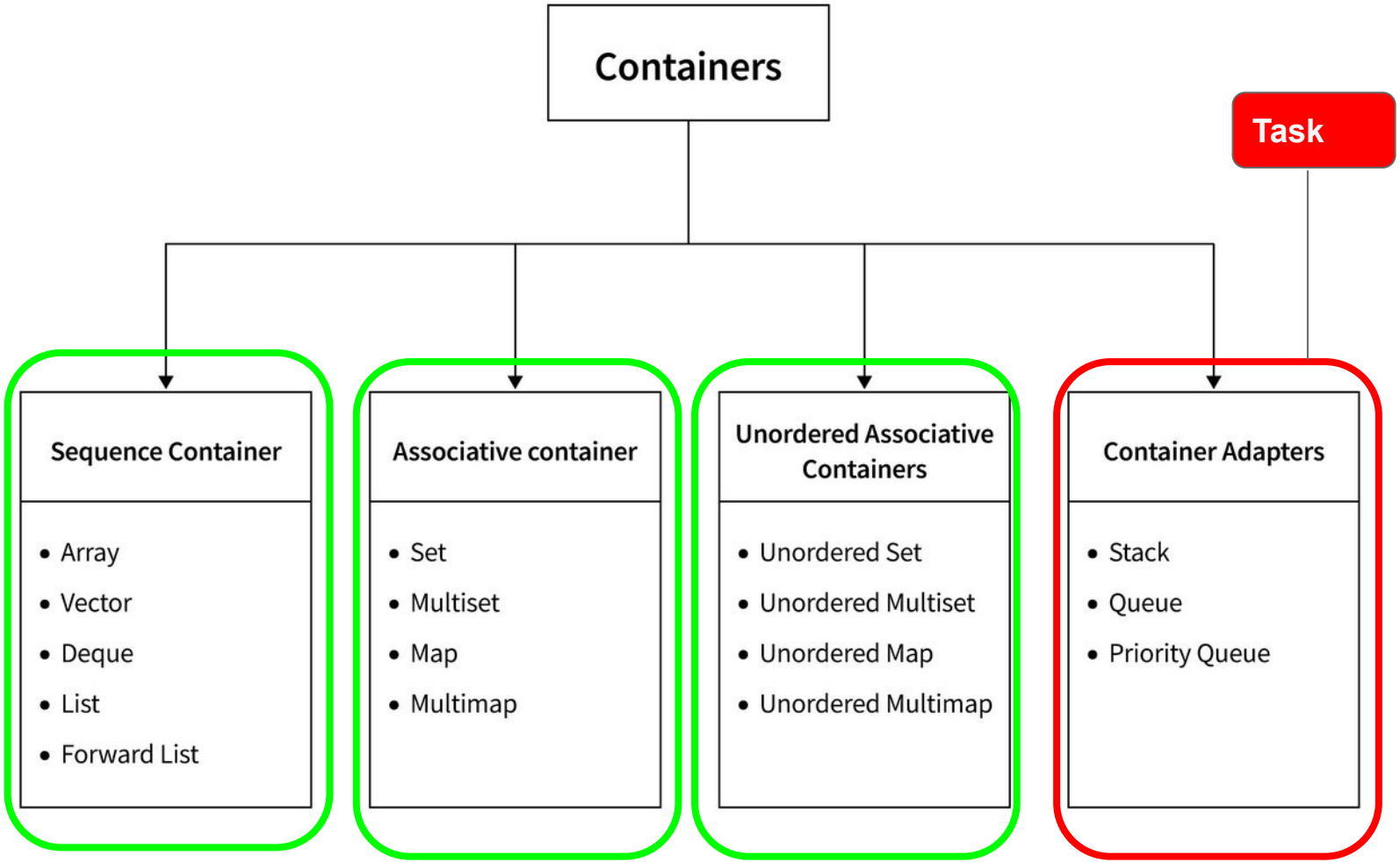
Query Size (= number of key-value pairs)

```
.empty() → false
.size()  → 2
```

Erase All

```
.clear()
```

v1.0


Containers
Task
Sequence Container
Array
Vector
Deque
List
Forward List
Associative container
Set
Multiset
Map
Multimap
Unordered Associative Containers
Unordered Set
Unordered Multiset
Unordered Map
Unordered Multimap
Container Adapters
Stack
Queue
Priority Queue

Tasks

1- Test all functions of STL
https://cplusplus.com/reference/
https://hackingcpp.com/cpp/

2- Interface and Multiple Inheritance:
-Create an interface class Drawable with a pure virtual function draw().
-Derive classes like Circle, Rectangle, and Triangle from Shape and implement the Drawable interface.
-Create objects of these derived classes and call the draw() function through a pointer to the Drawable interface

3-Uart Debugger
https://github.com/Moatasem-Elsayed/Uart_debuger

References

https://cplusplus.com/reference/
https://hackingcpp.com/cpp/