**Data Structures (C++)**

# **Introduction**

## **What is a Data Structure?**

* A collection of data values, the relationships among them, and the functions or operations that can be applied to the data.
* A data structure is a mathematical or logical model to store data.
* All containers are also considered Data Structures, not all data structures are containers.
* Data structures are custom classes or patterns that use containers to implement collection of data.

## **Data Structure Differences**

* Performance of operations
* Memory usage
* Difficulty to implement
* Type of data being stored.

## **Big ‘O’**

* runtime cost vs. memory footprint
* O(1) – constant time
* O(log n) – logarithmic time
* O(n) – linear time
* O(n log n) – quasilinear time
* O(n^x) – Polynomial time – n raised to the x operations
* O(x^n) – Exponential time – some number performed n exponential times
* O(n!) – Factorial time

## **Abstract Data type (ADTs) v Concrete Data Types (CDTs)**

**Abstract Data Type** - a data type or interface where only behavior is defined but not implementation...ADTs exist as a logical idea but not having a physical data representation. It's like a class versus an object.

A book is an abstract data type whereas a telephone book would be a concrete data type.

**Concrete Data Type** - a data type with a specialized solution-oriented data type that represents a well-defined single solution domain concept.

The top of the stack is the CPUs current command in process. The previous command that lead the CPU to the current command (top of the stack) was just popped off the stack because it was just executed.

In comparison to how the stack menu works in Visual Studios its displayed opposite the conceptual view of how physical stack diagram would look in popping the previous command off the top of stack once executed.

In Visual Studios, the previous process which was already executed in listed underneath the current process in VS's call list. That previous call or executed command has already been popped off the stack but it is shown in a top-bottom format for reference of what was previously already popped off the stack.

In general terms (aka the 'user level'), Abstract Data Types exist as a blueprint template for how a data type can be used, whereas a concrete data type is an example of or a specific implementation of an abstract data type.

The confusion comes in the level of abstraction or the perspective of which the dialogue is in reference to:

In terms of a user perspective, ADTs v CDTS is as simple as a class and object analogy.

In terms of the language level or machine level, Abstract Data Types (ADTs) from a user perspective could be considered a concrete data types (CDTs) on a language level.

An array or vector are good examples of those:

An Array or Vector are ADTs on the user level because they aren't specific examples of a declared vector or array but a conceptual data structure. From a language perspective, an Array or Vector is a CDT because the implementation is always the same (the way the vector or array is created in memory and how the operations are performed on the stack).

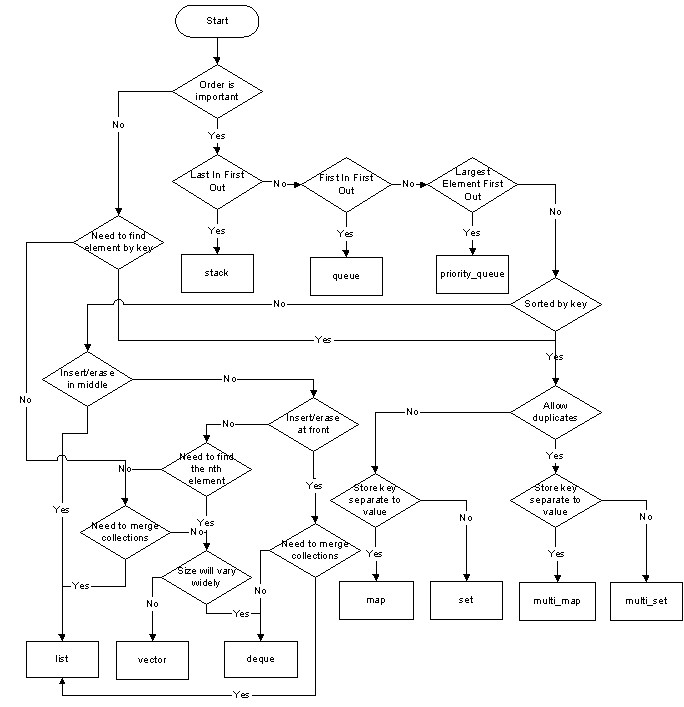
In terms of the language or machine level of abstraction, concrete data types are considered CDTs because the storing and manipulating of elements are always performed in a specific way.

**Iterators**

Something, something Darkside

Iterators are ptrs that are soft copies of a data structure used to iterate over w/o changing data.

## **Choose a Data Structure**



# **STL Data Structures**

## **Primitive Array**

### **Overview**

* Static Array.
* An array is a series of elements of the same type placed in contiguous memory locations.
* Array elements can be individually referenced by using ‘[]’ index operator.
* Indexing of an array starts from 0, first element at 0th index, the second element is at the 1rst index.
* Arrazy size is required at declaration time and its size remains constant throughout the program.

### **Syntax**

// Declare and Initialize

int arr[5];

int arr[] = { 1, 2, 3, 4, 5 };

int arr[5] = { 1, 2, 3, 4, 5 };

// Access and Assign | O(1)

cout << arr[0];

arr[0] = 0;

// Insert and Remove | O(n)

* Assigning a value or adding value to the end of array is O(1)
* Inserting at a specific index while maintaining order is O(n) – require copy

// Search | Unsorted: O(n), Sorted: O(log<n>)

* Depending on algorithm, search whole array or log (whole array)

// Array Size

* Max Size can be found by using: sizeof(arr) / sizeof(array element).
* Current Size can **not** be calculated at runtime, separate variable needed to keep track.

### **Code Example**

int main() // Example with Pointers

{

int arr[] = { 11, 22, 33, 44 };

// Define a pointer

int\* ptr = arr;

// Printing address of the arrary using array name | &arr == arr

cout << "Memory address of arr: " << &arr << endl;

cout << "Memory address of int\* ptr: " << ptr << endl;

// Print elements of an array

cout << "first element: " << \*arr << endl;

cout << "Second element: " << \*(arr + 1) << endl;

cout << "Third element: " << \*(ptr + 2) << endl;

cout << "fourth element: " << arr[3] << endl;

}

// Passing array as a pointer

return\_type function\_name(data\_type\* array\_name ) {// code; return return\_type;}

// Passing array as an unsized array

return\_type function\_name(data\_type array\_name[] ) {// code; return return\_type;}

// Passing array as an sized array

return\_type function\_name(data\_type array\_name[size\_of\_array] ) {// code; }

### **Use Cases**

* When the exact size is known at compile time, upper bound is known and doesn’t waste space.
* Days in the week, months in a year, number of grades of school (k-12).

**Pros:**

* Element access is fast – ptr arithmetic.
* Low overhead
* Direct conversion to pointers

**Cons:**

* Fixed size
* Inflexible, limited optimal uses
* Inserting and deleting elements at the beginning or middle can be inefficient

## **Primitive 2D Array**

### **Overview**

* Static Array. 2 or more dimensions.

### **Syntax**

// Declare and Initialize

char caArray[3][2];

char caArray[3][2] = {'a','b','c','d','e','f'}; // Fills one row at a time

char caArray[3][2] = {{'a', 'b'}, {'c', 'd'}, {'e','f'}};

char caArray[][2] = {{'a', 'b'}, {'c', 'd'}, {'e','f'}}; // Only need column size

int\*\* ippArray = new int\* [3]; // Pointer of pointers

char caArray[i][j];

**i**: Number of rows.

**j**: Number of columns*.*

// Access and Assign

caArray[i][j]; caArray[row][column];

// Insert and Remove

Use nested for loop to search all, iterate all

### **Code Example**

int main() // Example with Pointers

{

int\*\* ippArray = new int\* [3]; // initializes int ptr array to which contain 3 ptr elements, each ptr element points to something on the heap

// initializes each pointer element to allocate an integer array of 2 // // // elements, also on the heap.{ {1,2}, {3,4}, {5,6} }

// Each array element in the array is also considered a pointer.

for (int i = 0; i < 3; i++)

{

ippArray[i] = new int[2];

}

// Populates the 3 outer elements (array of 2 ints)

for (int i = 0; i < 3; i++)

{

for (int j = 0; j < 2; j++)

{

ippArray[i][j] = i + j;

}

}

// Prints the 2 inner elements, per one outer element of the 2d array

for (int i = 0; i < 3; i++)

{

for (int j = 0; j < 2; j++)

{

cout << ippArray[i][j] << " ";

}

cout << endl;

}

for (int i = 0; i < 3; i++)

{

delete[] ippArray[i];

}

delete[] ippArray;

}

### **Use Cases**

* Multidimensional arrays are often used for tasks like representing game boards, tables of data, images, and matrices in mathematical calculations.
* Useful when dealing with nested data structures or organizing data in a grid-like fashion.

**Pros:**

* **Stored in a contiguous space, quick indexing**
* **Allows us to store and process 2D images, videos in three dimensions, and matrix math.**
* **Less memory than other hierarchical data structures like lists.**
* **Less storage management overhead, and quick indexing**

**Cons:**

* **The outer most dimension, the byte size for each outer element needs to be known at compile time. The number of columns per row. How many elements are in each inner array.**
* **Dynamically changing the size of one dimension could be very inefficient**
* **Inflexible, limited optimal uses. Need to know sizes of things ahead of time.**
* **Time complexity increased at the insertion and deletion time.**

## **STL Array**

### **Overview**

* #include <array>
* Container wraps around fixed-size arrays and info of its size are not lost when declared to a pointer.

### **Syntax**

// Declare and Initialize

std::array <char, 3> arr = { 'G','f','G' };

// Access (.front and .back)

std::cout << arr[0] << " " << arr[2]; // Output: G G

std::cout << arr.front() << " " << arr.back(); // Output: G G

// Initialize and Set

arr[0] = ‘G’;

// Insert and Remove | Would need to create a new array and copy to insert/remove while maintaining sorted order, wouldn’t recommend.

arr[1] = ‘F’;

// Swap

std::array <int, 3> arr = { 'G','f','G' }; // ASCII val of 'G' =71

std::array <int, 3> arr1 = { 'M','M','P' }; // ASCII val of 'M' = 77 and 'P' = 80

arr.swap(arr1); // now arr = {M,M,P}

// Empty

bool x = arr.empty(); // false ( not empty)

// At() function

**at( ) function:**This function is used to access the element stored at a specific location, if we try to access the element which is out of bounds of the array size then it throws an exception.

std::cout << arr.at(2) << " " << arr1.at(2);

// Data() function

**data( ):**This function returns the pointer to the first element of the array object. Because elements in the array are stored in contiguous memory locations. This data( ) function return us the base address of the string/char type object.

const char\* str = "GeeksforGeeks";

array<char, 13> arr;

memcpy(arr.data(), str, 13);

std::cout << arr.data() << '\n';

// Size() function

std::array <int, 10> arr;

std::cout << arr.size() << '\n'; // total num of indexes

std::cout << arr.max\_size() << '\n'; // total num of indexes

std::cout << sizeof(arr); // total size of array

### **Code Example**

void main()

{

// std STL::array uses standard template array, need type and size

std::array<int, 10> collection;

int collectionSize = collection.size();

for (int i : collection)

{

// iterate collection

}

for (int i = 0; i < collection.size(); i++)

{

std::cout << collection[i] << std::endl;

}

}

### **Code Example 2**

int main() {

// construction uses aggregate initialization

// double-braces required

array<int, 5> ar1{ {3, 4, 5, 1, 2} };

array<int, 5> ar2 = { 1, 2, 3, 4, 5 };

array<string, 2> ar3 = { {string("a"), "b"} };

cout << "Sizes of arrays are" << endl;

cout << ar1.size() << endl;

cout << ar2.size() << endl;

cout << ar3.size() << endl;

cout << "\nInitial ar1 : ";

for (auto i : ar1)

cout << i << ' ';

// container operations are supported

sort(ar1.begin(), ar1.end());

cout << "\nsorted ar1 : ";

for (auto i : ar1)

cout << i << ' ';

// Filling ar2 with 10

ar2.fill(10);

cout << "\nFilled ar2 : ";

for (auto i : ar2)

cout << i << ' ';

// ranged for loop is supported

cout << "\nar3 : ";

for (auto& s : ar3)

cout << s << ' ';

}

### **Use Cases**

* When the exact size is known at compile time, upper bound is known and doesn’t waste space.
* Days in the week, months in a year, number of grades of school (k-12).

**Pros:**

* **Element access is fast – ptr arithmetic.**
* **Low overhead**
* **Direct conversion to pointers**

**Cons:**

* **Fixed size**
* **Inflexible, limited optimal uses**
* **Inserting and deleting elements at the beginning or middle can be inefficient**

## **Vector**

### **Overview**

* Dynamic arrays with the ability to resize themselves automatically.
* Add/Remove from end(element) → O(1)
* remove(element, index) → O(n)
* insert(element, index) → O(n)
* get(index) → O(1)

Can you index into vectors to add elements **?**

* You need to have pre-allocated memory for a vector to be able to index into a vector.
* The index you pass to operator[] must be in the range, [0, vector.size()).
* Add additional elements with vector.push\_back(value).

### **Syntax**

**// Declare and Initialize**

std::vector<string> strVector;

std::vector<string> strVector(50);

std::vector<string>::iterator it;

**// Iterators**

1. [**begin()**](https://www.geeksforgeeks.org/vectorbegin-vectorend-c-stl/) – Returns an iterator pointing to the first element in the vector
2. [**end()**](https://www.geeksforgeeks.org/vectorbegin-vectorend-c-stl/) – Returns an iterator pointing to the theoretical element that follows the last element in the vector
3. [rbegin()](https://www.geeksforgeeks.org/vector-rbegin-and-rend-function-in-c-stl/) – Returns a reverse iterator pointing to the last element in the vector (reverse beginning). It moves from last to first element
4. [rend()](https://www.geeksforgeeks.org/vector-rbegin-and-rend-function-in-c-stl/) – Returns a reverse iterator pointing to the theoretical element
5. Add a c to the beginning of everything above to make it a constant iterator.

**// Size and Capacity**

1. [**size()**](https://www.geeksforgeeks.org/vectorempty-vectorsize-c-stl/)**–** Returns the number of elements in the vector.
2. [max\_size()](https://www.geeksforgeeks.org/vector-max_size-function-in-c-stl/)**–** Returns the maximum number of elements that the vector can hold.
3. [capacity()](https://www.geeksforgeeks.org/vector-capacity-function-in-c-stl/) – Returns the size of the storage space currently allocated to the vector expressed as number of elements.
4. [**resize(n)**](https://www.geeksforgeeks.org/vector-resize-c-stl/) – Resizes the container so that it contains ‘n’ elements.
5. [empty()](https://www.geeksforgeeks.org/vectorempty-vectorsize-c-stl/) – Returns whether the container is empty.
6. [shrink\_to\_fit()](https://www.geeksforgeeks.org/vector-shrink_to_fit-function-in-c-stl/) – Reduces the capacity of the container to fit its size and destroys all elements beyond the capacity.
7. [reserve()](https://www.geeksforgeeks.org/using-stdvectorreserve-whenever-possible/)– Requests that the vector capacity be at least enough to contain n elements.

**// Element Access**

1. [reference operator [g]](https://www.geeksforgeeks.org/vectoroperator-vectoroperator-c-stl/) – Returns a reference to the element at position ‘g’ in the vector
2. [at(g)](https://www.geeksforgeeks.org/vectorat-vectorswap-c-stl/) – Returns a reference to the element at position ‘g’ in the vector
3. [front()](https://www.geeksforgeeks.org/vectorfront-vectorback-c-stl/) – Returns a reference to the first element in the vector
4. [back()](https://www.geeksforgeeks.org/vectorfront-vectorback-c-stl/) – Returns a reference to the last element in the vector
5. [data()](https://www.geeksforgeeks.org/vector-data-function-in-c-stl/) – Returns a direct pointer to the memory array used internally by the vector to store its owned elements.

**// Modify**

1. [assign()](https://www.geeksforgeeks.org/vector-assign-in-c-stl/)– It assigns new value to the vector elements by replacing old ones
2. [**push\_back()**](https://www.geeksforgeeks.org/vectorpush_back-vectorpop_back-c-stl/)**–** It push the elements into a vector from the back
3. [**pop\_back()**](https://www.geeksforgeeks.org/vectorpush_back-vectorpop_back-c-stl/) – It is used to pop or remove elements from a vector from the back.
4. [**insert()**](https://www.geeksforgeeks.org/vector-insert-function-in-c-stl/) – It inserts new elements before the element at the specified position
5. [erase()](https://www.geeksforgeeks.org/vectorclear-vectorerase-c-stl/) – It is used to remove elements from a container from the specified position or range.
6. [swap()](https://www.geeksforgeeks.org/vectorat-vectorswap-c-stl/) – It is used to swap the contents of one vector with another vector of same type.
7. [clear()](https://www.geeksforgeeks.org/vectorclear-vectorerase-c-stl/) – It is used to remove all the elements of the vector container
8. [**emplace()**](https://www.geeksforgeeks.org/vector-emplace-function-in-c-stl/) – It extends the container by inserting new element at position
9. [**emplace\_back()**](https://www.geeksforgeeks.org/vectoremplace_back-c-stl/)**–** It is used to insert a new element into the back of the vector container.

### **Code Example**

#include <vector>

int main()

{

vector<string> strVec;

vector<string>::iterator it;

strVec.push\_back("Ryan"); **// push\_back("");**

strVec.push\_back("James");

strVec.push\_back("Smith");

**// empty()**

if (!strVec.empty())

printVec(strVec); // Ryan James Smith

**// size() and capacity()**

cout << "\n" << "Current Size: " << strVec.size() << endl; // 3

**strVec.reserve(10);** cout << "Reserving capacity for 10 elements \n";

cout << "Current Size: " << strVec.size() << endl; // 3

cout << "Current Capacity: " << **strVec.capacity()** << endl; // 10

strVec.pop\_back(); // **pop\_back()**

printVec(strVec); // Ryan James

strVec.emplace\_back("Smith"); **// emplace\_back("")**

printVec(strVec); // Ryan James Smith

strVec.insert(strVec.begin() + 3, "Tyler"); **// insert(it\*, “str”)**

printVec(strVec); // Ryan James Smith Tyler

**// resize(i)** allows indexing, initializes empty strings, reserve() does not.

strVec.resize(10); // strVec.size() now equals 10.

**strVec[4] = "Robert**"; // **indexing** allowed because of resize()

printVec(strVec); // Ryan James Smith Tyler Robert

**// erase(\*start, \*end) – erases the start index up to end index – 1.**

strVec.erase(strVec.begin() + 3, strVec.begin() + 5); **// erase(it\* start, it\* end)**

printVec(strVec); // Ryan James Smith

system("pause");

}

### **Use Cases**

* Need a container for fast access, not a ton of elements and not inserting/deleting from the middle.
* Use it by default for any container.

**Pros:**

* **Dynamic Size**
* **Element access is fast – ptr arithmetic.**
* **Adding / Removing elements from back is fast.**

**Cons:**

* **Inserting to or removing from a specific spot (middle) of the container.**
* **Searching target element for large data sets.**
* **Adding elements when the capacity is reached requires copy operation.**

## **List**

### **Overview**

* Non-contiguous container of sequenced elements.
* Sequence container, optimized for rapid insert and remove.
* Each element of a linked list is located at a random location in memory.
* In C++, lists are concrete data structure containers part of the stl library.
* In C++, std::list is implemented as a doubly-linked list by default.
* If you want to insert into a LIST by position, keep a separate counter variable.

### **Syntax**

// Declare and Initialize

list<int> num;

list<int> nums = { 1,2,3,4,5 }; // C++11 - initializer list

list<int> nums2 = nums;

// Access and Assign

.front(), .back(), .begin(), .rbegin()

list<int>::iterator it = nums.begin();

\*it = <value>;

// Insert and Remove

nums.insert(it, 100);

nums.erase(it);

nums.remove(4);

### **Code Example**

// STL\_list.cpp : list container as part of the Standard Template Library

// [https://www.youtube.com/watch?v=je7zlYfJf7Q&t=7s](https://www.youtube.com/watch?v=je7zlYfJf7Q&t=7s%20)

// Sequence container, optimized for rapid insert and remove

// Non-contiguous, doesn't support random-access.

// If you want to insert into a LIST by position, keep a separate counter variable

#include <iostream>

#include <string>

#include <list>

#include <vector>

using namespace std;

void printList(list<int>& myList);

int main()

{

list<int> num;

list<int> nums = { 1,2,3,4,5 }; // C++11 - initializer list

list<int> nums2 = nums; // ‘Deep’ Copy

printList(nums);

cout << "Size: " << nums.size() << endl;

// .front(), .back(), // .begin(), .rbegin()

cout << "front: " << nums.front() << ", back: " << nums.back() << endl;

cout << "front: " << \* nums.begin() << ", back: " << \*nums.rbegin() << endl;

// Inserting into a Vector v List.

vector<int> vec = { 1,2,3,4,5 };

vec.insert(vec.begin() + 2, 100);

//nums.insert(nums.begin() + 2, 100); //ERROR! Ptr arithmetic don’t work on lists

// INSERTING in a LIST : need a memory address from an iterator

// Example: Insert based on value of list element,

// find the node that has the value of 3, and insert 100 before that

list<int>::iterator it = nums.begin();

while (\*it != 3 && it != nums.end())

it++; // keep iterating until value is found, hold on to iterator address.

nums.insert(it, 100);

printList(nums);

nums.erase(it); // erase() deletes by iterator address, it still pointing at 3

printList(nums);

nums.remove(4); // remove() deletes by looking for a list element’s value.

printList(nums);

nums.clear();

cout << "Is nums cleared?" << std::boolalpha << nums.empty() << endl;

printList(nums2);

nums2.push\_back(6);

nums2.push\_front(0);

printList(nums2);

nums2.pop\_back();

nums2.pop\_front();

printList(nums2);

system("pause");

}

void printList(list<int>& myList)

{

list<int>::iterator it = myList.begin();

for (it; it != myList.end(); it++)

cout << \*it << " ";

// for (auto& i : myList) // for each loop works as well.

// cout << i << " ";

cout << "\n";

}

### **Use Cases**

* Web browsers: back and forth navigation, doubly linked list.
* When Dynamic memory allocation is needed to manage memory.
* Stacks and Queues are needed for implementation.
* Undo/Redo functionality.
* Hash tables
* Playlists

**Pros:**

* Insertion / Removal at a given position very fast.
* Low overhead – no memory wasted
* Dynamic size

**Cons:**

* Uses more memory than an array (ptr variable)
* Traversal / Search is slow
* Complexity
* Random access not possible.

## **Stack**

### **Overview**

* A linear abstract data structure in which order of operations are Last In Last Out (LIFO).
* Stack of plates.
* Added to the top of the stack (end of data structure a[n], whereas n is last element in the data structure, and also added to the a[n] end of the data structure).
* Implemented in a variety of ways for storing and managing data, therefor it's considered abstract, even though it's still considered a data structure.
* Data Structure that uses LIFO technique - LIFO (Last In First Out) - FILO (First In Last Out)
* Undo / Redo, back / forward on web browser
* empty(), size() - how many currently contain, top(), push(), pop()
* Container adaptors, which are classes that use an
* encapsulated object of a specific container class as its underlying container
* Stack uses an encapsulated object of either vector or deque (by default) or list (sequential container class) as its underlying container,
* providing a specific set of member functions to access its elements.
* template <class Type, class Container = deque<Type> > class stack;
* Element in a stack are inserted at the front of the list and
* a pointer keeps track of the elements underneath the top, aka, the front of the stack

### **Syntax**

// Declare and Initialize

stack<int> numbersStack

// Access

numbersStack.top()

// Insert and Remove

numbersStack.push(1);

numbersStack.emplace(2);

numbersStack.pop();

### **Code Example**

// Stack.cpp | <https://www.youtube.com/watch?v=GBST5uQ_yos&t=16s>

#include <stack>

using namespace std;

void printStack(stack<int> myStack)

{

while (!myStack.empty())

{

// top()

cout << myStack.top() << endl;

myStack.pop();

}

}

int main()

{

stack<int> numbersStack;

// push(), emplace()

numbersStack.push(1);

numbersStack.push(2);

numbersStack.emplace(3);

// size()

cout << "Stack Size: " << numbersStack.size() << endl;

// Empty()

if (numbersStack.empty())

cout << "Stack is Empty!" << endl;

else

cout << "Stack is not Empty" << endl;

printStack(numbersStack);

// Pop - removes first element from top of stack

numbersStack.pop();

cout << "Pop from top of stack" << endl;

printStack(numbersStack);

system("pause");

}

////////////////////////////

// C-Style Stack

struct Element

{

struct Element\* next;

int data;

};

bool push(struct Element\*\* stack, int data)

{

struct Element\* elem = malloc(sizeof(struct Element));

if (!elem) { return false; }

elem->data = data;

elem->next = \*stack; // current stack's address to the new element's next ptr.

\*stack = elem; // the head of the stack is now the new element.

}

### **Use Cases**

* Function Calls
* Recursion
* Undo/Redo functionality
* Syntax parsing
* Depth-First Searches on tree and graph data structures.

**Pros:**

* Simplicity
* LIFO principle, when you need to perform last in first out instructions.
* Efficiency – push/pop are constant time
* Limited memory footprint, no wasted memory

**Cons:**

* Limited Access – access only from the top of the stack (the front of list)
* Data Overflow – more elements are pushed onto the stack than it can hold, data loss.
* No random access
* Limited capacity – capacity unknown or highly variable.

## **Queue**

### **Overview**

* Queues are a type of [container adaptors](https://www.geeksforgeeks.org/containers-cpp-stl/) that operate in a first in first out (FIFO) type of arrangement.
* Elements are inserted at the back (end) and are deleted from the front. Queues use an encapsulated object of [deque](https://www.geeksforgeeks.org/deque-cpp-stl/)or [list](https://www.geeksforgeeks.org/list-cpp-stl/)(sequential container class) as its underlying container
* FIFO (First in First Out) : "first come first serve" | people waiting in line.
* Fair data structure
* When to use a Queue data structure?
* Execute elements of a data structure in exact order they were added,
* but the computer isn't fast enough to keep up. | e.g printer queue. | online orders | movie tickets
* How to access / manipulate element in Queue without using pop? How to access without copying elements in a function.
* Enqueue: Add an element to the back of the queue.
* Dequeue: Remove the element at the front of the queue.
* Peek: Return the element at the front of the queue without removing it.
* Size: Return the number of elements in the queue.
* isEmpty: Check if the queue is empty.

### **Syntax**

// Declare and Initialize

queue<int> numbersQueue;

queue<int> numbersQueue2

// Access

numbersQueue.front();

numbersQueue.back();

// Insert and Remove

numbersQueue.push(1);

numbersQueue.emplace(2);

numbersQueue.pop();

// Swap

numbersQueue.swap(numbersQueue2);

### **Code Example**

// Queue.cpp | <https://www.youtube.com/watch?v=jaK4pn1jXTo>

#include <queue>

void printQueue(std::queue<int> queue)

{

while (!queue.empty())

{

std::cout << queue.front() << " ";

queue.pop();

}

}

int main()

{

std::queue<int> myQueue;

myQueue.push(1);

myQueue.push(2);

myQueue.push(3);

std::cout << "Size is: " << myQueue.size() << std::endl;

std::cout << "First Element is: " << myQueue.front() << std::endl;

std::cout << "Last Element is: " << myQueue.back() << std::endl;

std::cout << "My queue: " << std::endl;

printQueue(myQueue);

system("pause");

}

### **Use Cases**

* Working as a buffer between a slow and a fast device. For example keyboard and CPU, and two devices on network.
* ATM Booth Line
* Ticket Counter Line
* CPU task scheduling
* Waiting time of each customer at call centers.

**Pros:**

* FIFO principles applied.
* It is used in applications in which data is transferred asynchronously between two processes. The queue in C++ can be used for obtaining the synchronization.
* The call center systems use queues to hold the incoming calls and to resolve them one by one.
* The queues can handle the interrupts in a real-time system and are ideal for disk scheduling and CPU scheduling.
* Handles large amounts of data efficiently
* Insertion / Deletion is fast and easy.
* Good for a service that is being used by multiple consumers.
* Play nice with other data structures.

**Cons:**

* The operations such as insertion and deletion of elements from the middle are time consuming.
* In a classical queue, a new element can only be inserted when the existing elements are deleted from the queue.
* Searching an element takes O(N) time.
* Maximum size of a queue must be defined prior in case of array implementation.

### **Types of Queues**

* **Simple Queue:**Simple queue also known as a linear queue is the most basic version of a queue. Here, insertion of an element i.e. the Enqueue operation takes place at the rear end and removal of an element i.e. the Dequeue operation takes place at the front end.
* **Circular Queue:**This is mainly an efficient array implementation of Simple Queue. In a circular queue, the element of the queue act as a circular ring. The working of a circular queue is similar to the linear queue except for the fact that the last element is connected to the first element. Its advantage is that the memory is utilized in a better way. This is because if there is an empty space i.e. if no element is present at a certain position in the queue, then an element can be easily added at that position.
* **Priority Queue:**This queue is a special type of queue. Its specialty is that it arranges the elements in a queue based on some priority. The priority can be something where the element with the highest value has the priority so it creates a queue with decreasing order of values. The priority can also be such that the element with the lowest value gets the highest priority so in turn it creates a queue with increasing order of values.
* **Dequeue:**Dequeue is also known as Double Ended Queue. As the name suggests double ended, it means that an element can be inserted or removed from both the ends of the queue unlike the other queues in which it can be done only from one end. Because of this property it may not obey the First In First Out property.

## **Priority Queue**

### **Overview**

* A **C++ priority queue** is a type of [container adapter](https://www.geeksforgeeks.org/containers-cpp-stl/), specifically designed such that the first element of the queue is either the greatest or the smallest of all elements in the queue.
* Implemented using arrays, linked lists, heaps, and BSTs.
* Constant time O(n) lookup of largest element
* Logarithmic time O(log\*n) insertion and extraction.
* #include <queue>
* User-provided Compare CAN be supplied to change priority ordering.

#### **User-Provided Compare() function**

**template <class T, class Container<T>, class Compare> class priority\_queue;**

* class T - type of data stored in the container.
* class Container is a vector by default, but can be a vector or an array, <T> being type of variable container is holding.
* Compare = std::less<typename Container::value\_type> - function for ordering priority (can be user-defined).User-provided Compare CAN be supplied to change priority ordering.

#### **Container**

* Type of underlying container to use to store the elements
* Must satisfy requirements of a sequence container: std::vector, std::deque.
* Iterators must satisfy the requirements of Random Access Iterator (can point to any element, bi-directional, in O(1) ).
* Must provide the following functions, front(), push\_back(), pop\_back()

#### **Compare function()**

* Compare type providing a strict weak ordering
* Compare param is defined such that,
* It returns true if it's first argument comes before it's second argument.

**Queue functions**

* size(), =, top(), empty(), push(), pop()
* top() will return highest priority.

#### **Syntax**

// Declare and Initialize

priority\_queue<int> priority\_Q;

priority\_queue<int, vector<int>, std::greater<int>> priority\_Q;

vector<int> myVec = { 8,1,6,4,0,7,2,9 };

// Access

priority\_Q.top();

// Insert and Remove

for (int x : myVec)

priority\_Q.push(x);

priority\_Q.pop();

### **Code Example**

int main()

{

// Greater | Priority\_Queue defaults | Descending Order

priority\_queue<int> priority\_Q;

vector<int> myVec = { 8,1,6,4,0,7,2,9 };

// insert elements into priority queue

for (int x : myVec)

priority\_Q.push(x);

while (!priority\_Q.empty())

{

cout << priority\_Q.top() << endl;

priority\_Q.pop();

}

//Priority Queue | add'l arguments | greater than Compare, Ascending Order

priority\_queue<int, vector<int>, std::greater<int>> priority\_Q;

vector<int> myVec = { 8,1,6,4,0,7,2,9 };

// insert elements into priority queue

for (int x : myVec)

priority\_Q.push(x);

while (!priority\_Q.empty())

{

cout << priority\_Q.top() << endl;

priority\_Q.pop();

}

// Lamba for defining custom Compare function(), useful for custom classes.

auto cmp = [](int a, int b) {

return a < b;

};

priority\_queue<int, vector<int>, decltype(cmp)> priority\_Q(cmp);

vector<int> myVec = { 8,1,6,4,0,7,2,9 };

for (int x : myVec)

priority\_Q.push(x);

while (!priority\_Q.empty())

{

cout << priority\_Q.top() << endl;

priority\_Q.pop();

}

system("pause");

}

### **Use Cases**

* The hospital emergency queue
* Operating Systems task prioritization, resource allocation
* Data compression
* Pathfinding graph algorithms (A\*, Dijkstra)
* Heap sorting
* Video game object rendering
* Network routing
* Database Indexing

**Pros:**

* Element access is fast
* Dynamic sorting based on dynamic prioritization.
* Used in efficient algorithms
* Common to real-time systems that need to prioritize tasks.
* Performance optimization for time to complete average tasks.

**Cons:**

* High complexity
* High memory usage
* Min/Max search may be faster with heaps or BSTs
* Unpredictable, implementation dependent
* Priority conflicts
* Dynamically updating priority can be a large performance hit.

## **Dequeue**

### **Overview**

* Sequence container that supports push/pop from both ends.
* Supports random access d[2] - indexing
* Non-contiguous
* Implemented as a sequence of indivudally allocated fixed-size arrays w/ add'l bookkeeping.
* When size of deque is filled, a new fixed array is created and
* A pointer connects from last element in queue to first element in new fixed array
* Cannot insert a new element by indexing. Indexed element must be < deque.size();
* Generalized version of a queue, allows insert and remove from both ends.
* Doesn’t follow FIFO or LIFO

### **Syntax**

// Declare and Initialize

deque<int> numDeque = { 1,2,3,4,5 };

deque<int> numDeque2;

// Access and Assign

cout << numDeque[2];

numDeque[2] = 4;

// Insert and Remove

numDeque.push\_back(7);

numDeque.push\_front(0);

numDeque.pop\_back();

numDeque.pop\_front();

### **Code Example**

// Deque.cpp | <https://www.youtube.com/watch?v=3U_Eg9WdGr0>

#include <iostream>

#include <deque>

using namespace std;

void printDeque(deque<int>& d)

{

for (deque<int>::iterator it = d.begin(); it != d.end(); it++)

cout << \*it << " ";

cout << endl;

}

int main()

{

deque<int> numDeque = { 1,2,3,4,5 };

cout << "Size: " << numDeque.size() << endl;

cout << "Third element: " << numDeque[2] << endl;

cout << "Front element: " << numDeque.front() << endl;

cout << "Last element: " << numDeque.back() << endl;

printDeque(numDeque);

for (deque<int>::reverse\_iterator it = numDeque.rbegin(); it != numDeque.rend(); it++)

cout << \*it << " ";

cout << endl;

numDeque.push\_back(6);

numDeque.push\_back(7);

numDeque.push\_front(0);

numDeque.push\_front(-1);

printDeque(numDeque);

numDeque.pop\_back();

numDeque.pop\_front();

printDeque(numDeque);

numDeque.clear();

system("pause");

}

### **Use Cases**

* Job Scheduling algorithms
* Stack and queue operations
* Web browser history, recent URLs added to the front, back needs to cleared at some point
* Graph traversal algorithms with BFS, which uses a deque for add/remove of both ends.
* Task management systems
* Caching systems

**Pros:**

* Removal from both front and back of queue.
* Fast add/remove in front or back of queue.
* Dynamic Size
* Versatility, can be implemented as stacks or queues.
* No memory reallocation required
* Cache-friendly

**Cons:**

* Higher memory overhead vs other data structures because of multiple pointers needed.
* Complexity
* Synchronization can be an issue if not carefully used in multi-threaded environments.
* Not platform independents (some platforms may not support deques).
* Not good for sorting and searching, O(n) time complexity.
* Limited functionality.

## **Map**

### **Overview**

* Data Structure that maps a key to a value.
* Self balancing binary search tree, Red-Black tree.
* Use unordered\_map whenever possible over map unless you need an sorted data structure.
* Prints in alphabetic order, keys are ordered.
* To retrieve a copy of data without inserting it, use the at() function.
* Fill out the reference data directly. Saves on copying.
* Insert operator for maps have the ability to mutate objects, via insert.
* Index operator works differently in C++ than in other languages.
* no const version of insertion operator, C++ will always insert things into map.
* Keys have to be unique.
* std::map are ordered(by key) by default
* Duplicate keys are ignored, not overriden with new value.
* map\_name.emplace\_hint(position, key, element), iterator position gives map hint where to start.
* Maps are mainly implemented as binary search trees

### **Syntax**

// Declare and Initialize

int arr[5];

int arr[] = { 1, 2, 3, 4, 5 };

int arr[5] = { 1, 2, 3, 4, 5 };

// Access and Assign

int arr[5];

// Insert and Remove

int arr[5];

### **Code Example**

// STD\_Maps.cpp : Key-Value pair data structure

// <https://www.youtube.com/watch?v=aEgG4pidcKU&t=2s>

//

// Maps order elements by key, in ascending order.

// Keys have to be unique.

void BasicMapFunctions();

void PokeDex();

int main()

{

BasicMapFunctions();

PokeDex();

system("pause");

}

void BasicMapFunctions()

{

// map are ordered(by key) by default

map<string, string> myMap;

// insert - doesn't matter which order you insert them in,

// map will order elements by key in ascending order.

myMap.insert(pair<string, string>("banana", "die Banane")); // 2nd element - "banana"

myMap.insert(pair<string, string>("apple", "der Apfel")); // 1rst element - "apple"

myMap.insert(pair<string, string>("orange", "die Orange")); // 3rd element

myMap.insert(pair<string, string>("strawberry", "die Erdbeere")); // 5th element

// The advantage of emplace is, it does in-place insertion

// and avoids an unnecessary copy of object.

myMap.emplace("raspberry", "die Rasbere"); // 4th element

myMap.emplace("blueberry", "die bluouse"); // either syntax works for emplace.

// print map

for (pair<string, string> element : myMap) // for(auto& pair : myMap)

cout << element.first << " - " << element.second << endl;

// size()

cout << "Size of myMap: " << myMap.size() << "\n\n";

// access and change element of a map.

myMap["strawberry"] = "fruit";

// find if a key exists

if (myMap.find("apple") != myMap.end()) // safer to use .at()

cout << "myMap[""apple""]: " << myMap.at("apple") << endl;

else

cout << "apple is NOT present" << endl;

// erase - remove individual elements by key.

cout << "---- apple key being erased with myMap.erase ----" << endl;

myMap.erase("apple");

for (map<string, string>::iterator it = myMap.begin(); it != myMap.end(); it++)

{

cout << "key: " << it->first << ", value: " << it->second << endl;

}

cout << "\n";

// Print them backwards with reverse iterators

for (auto it = myMap.rbegin(); it != myMap.rend(); it++)

{

cout << it->first << " = " << it->second << endl;

}

cout << "Size of myMap: " << myMap.size() << "\n";

// find if a key exists

if (myMap.count("apple"))

cout << "apple is present" << endl;

else

cout << "apple is NOT present" << endl;

// clears all map elements.

myMap.clear();

}

void PokeDex()

{

// When using a container as a variable type in another container,

// that variable container doesn't need a name. list<string> as map value in this case.

map<string, list<string>> pokeDex;

list<string> pikachuAttacks{ "thunder shock", "tail whip", "quick attack" };

list<string> charmanderAttacks{ "flame thrower", "scary face" };

list<string> chikoritaAttacks{ "razor leaf", "poison powder" };

pokeDex.insert(pair<string, list<string>>("Pikachu", pikachuAttacks));

pokeDex.insert(pair<string, list<string>>("Charmander", charmanderAttacks));

// map\_name.emplace\_hint(position, key, element)

// emplace\_hint speeds up the insertion process by hinting the map where to start the insert.

pokeDex.emplace\_hint(pokeDex.begin(), pair<string, list<string>>("Chikorita", chikoritaAttacks));

for (pair<string, list<string>> pair : pokeDex) // range-based for loop for Map.

{

cout << pair.first << " - ";

for (string attack : pair.second) // range-based for loop for list.

cout << attack << ", ";

cout << "\n";

}

}

### **Use Cases**

* When the exact size is known at compile time, upper bound is known and doesn’t waste space.
* Days in the week, months in a year, number of grades of school (k-12).

**Pros:**

* **Element access is fast – ptr arithmetic.**
* **Low overhead**
* **Direct conversion to pointers**

**Cons:**

* Iterating through a vector is always faster than iterating through a Map.
* **Inflexible, limited optimal uses**
* **Inserting and deleting elements at the beginning or middle can be inefficient**

## **Unordered Map (HashTable)**

### **Overview**

* Unordered Map is a Hash Table
* Unordered Map uses a hash function to hash your key and return an index.
* Hash function returns generated index and finds which bucket your value lies in, and retrieves it.

### **Syntax**

**// Declaration**

std::unordered\_map<std::string, CityRecord> cityMap;

**// Assign / Initialize / Add / Insert**

cityMap["Melbourne"] = CityRecord{"Melbourne", 50000, 2.4, 9.4};

cityMap.emplace(pair("Concord", CityRecord{ "Concord", 30000, 37.3, 47.2 }));

cityMap.insert(pair("Manchester", CityRecord{ "Manchester", 33333, 3.3, 4.4 }));

**// Access**

if (cityMap.find("Boston") != cityMap.end())

CityRecord& bData = cityMap.at("Boston");

**// Iterate (Structured bindings)**

for (auto& [name, city] : cityMap)

std::cout << name << "\n Population: " << city.Population << endl;

**// Remove**

cityMap.erase(cityMap.begin()); // by iterator

cityMap.erase("London"); // by value

### **Code Example**

// ChernoMap.cpp | <https://www.youtube.com/watch?v=KiB0vRi2wlc>

#include <unordered\_map>

struct CityRecord

{

std::string Name;

uint64\_t Population = 0;

double Latitude = 0.0, Longitude = 0.0;

CityRecord() {}

CityRecord(std::string s, uint64\_t i, double lat, double l)

: Name(s), Population(i), Latitude(lat), Longitude(l) {}

// operator defines the unique key inside your actual map

bool operator<(const CityRecord& other) const

{

return Population < other.Population;

}

};

int main()

{

std::unordered\_map<std::string, CityRecord> cityMap;

cityMap["Melbourne"] = CityRecord{"Melbourne", 50000, 2.4, 9.4};

cityMap["Lol-town"] = CityRecord {"Lol-town", 50, 2.4, 9.4};

cityMap["Berlin"] = CityRecord {"Berlin", 250000, 2.4, 9.4};

cityMap["Paris"] = CityRecord {"Paris", 3000000, 2.4, 9.4};

cityMap["London"] = CityRecord {"London", 7723828, 2.4, 9.4};

cityMap.emplace(pair("Conord", CityRecord{ "Concord", 30000, 37.3, 47.2 }));

cityMap.insert(std::make\_pair("Manchester", CityRecord{ "Manchester", 33333, 3.3, 4.4 }));

// Create a blank reference (entry) and auto insert it into map.

CityRecord& bostonData = cityMap["Boston"];

// Fill out the reference data directly. Saves on copying.

bostonData.Name = "Boston";

bostonData.Population = 1200000;

bostonData.Longitude = 178.493;

bostonData.Latitude = 38.382;

const auto& cities = cityMap;

// CityRecord& bostonData = cities["Boston"]; // Error

// Creating a new map entry with a new ref variable requires a non-const map.

// Before looking up data with at() need to use find() to see if it exists.

if (cities.find("Boston") != cities.end())

{

// Use the at() function to retrieve a copy of data without inserting it,

CityRecord& bData = cityMap.at("Boston");

const CityRecord& bDataConst = cities.at("Boston");

}

// Structure binding (Only available in C++17)

for (auto& [name, city] : cityMap)

{

// Not printed in alphabetic order (unordered map)

std::cout << name << "\n Population: " << city.Population << std::endl;

}

}

### **Use Cases**

* Database indexing
* Disk based data structures
* Caching systems
* Cryptographic algorithms: password verification
* Network Routing Tables
* Search Engines

**Pros:**

* Better synchronization
* Constant time searching, insertion, deletion on average.
* Faster searching than search trees.
* Fast data retrieval and manipulation
* Space efficient, no extra memory wasted.
* Easy to use.
* Dynamic resizing.

**Cons:**

* Collisions occur in large data sets.
* Lots of collisions lead to slow programs
* Limited capacity.
* Unsorted data, retrieving a group of elements in a specific order is slow (like elements).

### **Template Specialization**

**// TEMPLATE SPECIALIZATION**

// Template specialization for std::hash struct that is specific to CityRecord value

// if you attempt to use a custom class as a key in a map, you need to create

// a template specialization so that the compiler knows how to hash it.

namespace std

{

template<>

struct hash<CityRecord>

{

// call operator comes with hash struct

size\_t operator()(const CityRecord& key) // key that we will be hashing.

{

// custom hash that uses string Name for hashing

return hash<std::string>()(key.Name); // construct it then call it.

}

};

}

// In main.cpp | possible because of template specialization for hashing CityRecord().

std::map<CityRecord, uint32\_t> foundedMap

foundedMap[CityRecord{ "Melbourne", 500000, 2.4, 9.4 }] = 1850;

### **Misc. Map notes**

* Use unordered map over map unless you need a sorted data structure.
* Use map if you need lookup, otherwise use Vector for iteration.
* References always need to be initialized to a known value, in this case the value is being created and then it's referenced.
* Iterating through a vector is always faster than iterating through a Map.
* Index operator works differently in C++ than in other languages.
* There is no const version of insertion operator
* Insert operator for maps have the ability to mutate objects, via insert
* Thread Safe Rule: If you have a variable shared with 2 or more threads, and at least one of those threads modifies said variable, then you need synchronization. Without you have a data race and UpperBounds

### **Collision**

**Collision Resolution:** Collision resolution in hash can be done by two methods:

* [Open addressing](https://www.geeksforgeeks.org/hashing-set-3-open-addressing/) and
* Closed addressing.

**Closed Addressing:**

Closed addressing collision resolution technique involves chaining. Chaining in the hashing involves both array and linked list. In this method, we generate a probe with the help of the hash function and link the keys to the respective index one after the other in the same index. Hence, resolving the collision.

**Open Addressing:** Open addressing collision resolution technique involves generating a location for storing or searching the data called **probe**.

* **Linear Probing:** If there is a collision at **i** then we use the hash function – **H(k, i ) = [H'(k) + i ] % m**  
  where, **i** is the index, **m** is the size of hash table **H( k, i )** and **H'( k )** are hash functions.
* **Quadratic Probing:** If there is a collision at **i** then we use the hash function –  **H(k, i ) = [H'(k) + c1 \* i + c2 \* i2 ] % m**  
  where, **i** is the index, **m** is the size of hash table **H(k, i )** and **H'( k )** are hash functions, **c1** and **c2** are constants.
* **Double Hashing:** If there is a collision at **i** then we use the hash function – **H(k, i ) = [H1(k, i) + i \* H2(k) ] % m**  
  where, **i** is the index, **m** is the size of hash table **H(k, i )**, **H1( k) = k % m** and **H2(k) = k % m’** are hash functions.

## **MultiMap**

### **Overview**

* Multimap is an associative container of sorted key-value pairs.
* Unlike std::map, std::multimap allows the same key to be added multiple times to the container.
* Nearly identical to a std::map<key, vector< value>>
* Multimap are sorted by the key (either in ascending or descending order).
* multimap<char, int, std::less<>> multiMap; orders keys in ascending order.
* multimap<char, int, std::greater<>> multiMap; orders keys in descending order.
* Cannot use [index] operator or at() function in Multimap because keys are not unique.
* O(log n) time complexity.
* Use const\_iterators for read-only operations.

### **Syntax**

// Declare and Initialize

**multimap<char, int> multiMap;**

// Access

// See Code example: find(), count(), equal\_range(), iterators

// Insert

multiMap.insert(make\_pair('a', 5));

// Remove

multiMap.erase('a');

### **Code Example**

// MultiMap.cpp | <https://www.youtube.com/watch?v=jOJSdEwldwk>

#include <map>

#include <unordered\_map>

using namespace std;

int main()

{

**multimap<char, int> multiMap;**

MultiMap(multiMap);

iterateMultiMap(multiMap);

getPairsWithKey(multiMap, 'a');

countMultiMap(multiMap);

findMultiMap(multiMap);

upperLowerBoundsMultiMap(multiMap);

}

**// Insert(), Empty(), Size(), Erase**

void MultiMap(multimap<char, int>& multimap)

{

**// Insert()**

multimap.insert(make\_pair('a', 5));

multimap.insert(pair<char, int>('a', 1));

multimap.insert(make\_pair('a', 1));

multimap.insert(make\_pair('a', 3));

multimap.insert(make\_pair('b', 3));

multimap.insert(make\_pair('b', 2));

multimap.insert(make\_pair('c', 2));

if(!multimap.empty()) **// Empty(), Size()**

cout << "MultiMap Size: " << multimap.size() << endl;

multimap.erase('a'); **// Erase()**

cout << "MultiMap Size after deleting 'a' key " << multimap.size() << endl;

}

void iterateMultiMap(multimap<char, int>& multimap)

{

// Iterate Multimap | .first .second

for (auto& elm : multimap)

{

cout << elm.first << " -> " << elm.second << endl;

}

// Can use iterators to iterate

std::multimap<char, int>::iterator it;

for (it = multimap.begin(); it != multimap.end(); it++)

{

cout << "Key: " << it->first << ", Value: " << it->second << endl;

}

}

// **EQUAL\_RANGE('a')** - returns a pair<itr1, itr2> for the ‘a’ key.

void getPairsWithKey(std::multimap<char, int>& multimap, char c)

{

// pair<std::multimap<char, int>::iterator, std::multimap<char, int>::iterator>

**range** = multimap.**equal\_range('a')**; // range is a pair of iterators.

auto **range** = multimap.**equal\_range('a')**;

for (auto it = range.first; it != range.second; it++)

{

cout << it->first << " " << it->second << endl;

}

}

**// COUNT()**

void countMultiMap(multimap<char, int>& multimap)

{

// COUNT | # of values for the key 'a' | count('a')

cout << "\n" << "Size of 'a': " << multimap.count('a') << endl;

}

**// FIND()**

void findMultiMap(multimap<char, int>& multimap)

{

// Doesn't guarantee which value

std::multimap<char, int>::iterator it = multimap.find('c');

if (it != multimap.end())

cout << "Key Found!" << endl;

else

cout << "Key NOT Found!" << endl;

}

**// LOWER\_BOUND(), UPPER\_BOUND()**

void upperLowerBoundsMultiMap(multimap<char, int>& multimap)

{

// LOWER\_BOUND | returns first key-value pair with ('a') key added to map.

auto lowerBound = multimap.lower\_bound('a');

cout << lowerBound->first << " " << lowerBound->second << endl;

// UPPER\_BOUND | // last key-value pair + 1 with ('a') key added to map.

auto upperBound = multimap.upper\_bound('a');

cout << upperBound->first << " " << upperBound->second << endl; // ‘b’ 3

}

// UNORDERED MULTIMAP NOTES

// Unordered associative container that can have the same key multiple times.

// Search, insert, remove O(1) - average

// Elements organized into buckets.

// Hashing to insert elements into buckets.

// Same as unordered\_map but can have multiple elements of the container with the same key.

// Why unordered\_multimap?

// Maintain collection of duplicate {key:value} pairs with fast insertion/removal.

// Benefits of multimap with duplicate keys and unordered map for O(1) lookup, insert, remove.

### **Use Cases**

* Word index
* Dictionary
* Bucketing key values.
* Scheduling or Event Management

**Pros:**

* Duplicate keys.
* More convenient syntax and insertion than a map<key, vector< value>>
* Std::map advantages

**Cons:**

* Complexity
* Std::map disadvantages

## **Unordered MultiMap**

### **Overview**

* Hash Table with the ability to have duplicate keys.

### **Syntax**

// Declare and Initialize

unordered\_multimap<char, int> multiMap;

### **Code Examples**

* See Multimap and Unordered Map

### **Use Cases**

* Maintain collection of duplicate {key:value} pairs with fast insertion/removal.

**Pros:**

* Maintain collection of duplicate {key:value} pairs with fast insertion/removal.

**Cons:**

* Complexity

## **Set**

### **Overview**

Sets.cpp | <https://www.youtube.com/watch?v=YuZPHhniZtw>

* Data Structure where each element is unique.
* Each value is a key, which means that we access each value using the value itself
* Value of element identifies it.
* Once element is added to Set, it cannot be modified.
* Elements can only be added or removed.
* Values always remain in sorted order.
* The value can’t be modified once it is stored, immutable
* By default, values of the set are sorted in ascending order.

### **Syntax**

// Declare and Initialize

std::set<int> mySet;

std::set<int> mySet2 = {1,2,3,4,5};

std::set<int, std::greater<int>> myReverseSet;

// Access and Assign

std::set<int>::iterator it = mySet.begin();

std::cout << \*it << std::endl;

// Insert and Remove

mySet.insert(1);

mySet.erase(2)

### **Code Example**

#include <set>

void BasicSet()

{

std::set<int> mySet;

mySet.insert(1);

mySet.insert(4);

mySet.insert(0);

for (std::set<int>::iterator it = mySet.begin();

it != mySet.end(); it++)

{

std::cout << \*it << std::endl;

}

std::cout << "\n";

// Check if empty

if (mySet.empty())

std::cout << "Set is Empty" << std::endl;

else

std::cout << "Set is not Empty" << std::endl;

std::cout << "\n";

// Check sizes

std::cout << "Current Size: " << mySet.size() << std::endl;

std::cout << "Max Size: " << mySet.max\_size() << std::endl;

// Removing elements

mySet.erase(2); // delete a single element

// delete range of elements up to but not including (4).

mySet.erase(mySet.begin(), mySet.find(4));

// Erase entire set

mySet.clear();

}

void ReverseSet()

{

std::set<int, std::greater<int>> myReverseSet;

myReverseSet.insert(40);

myReverseSet.insert(30);

myReverseSet.insert(60);

myReverseSet.insert(20);

myReverseSet.insert(50)

// 50 will only be adde once,compiler checks if value already exists,

// compiler won't attempt to add, if 50 is already in set.

myReverseSet.insert(50);

myReverseSet.insert(10);

myReverseSet.insert(0);

std::set<int, std::greater<int>>::iterator it;

for (it = myReverseSet.begin(); it != myReverseSet.end(); it++)

{

std::cout << \*it << std::endl;

}

std::cout << "\n";

}

### **Use Cases**

* Deduplication
* Music player app to list all the songs in proper order
* Detect cycles in a Linked List
* Database operations such as performing joins
* Auto-sorting an array.

**Pros:**

* store unique values in order to avoid duplications
* Sorted elements, fast lookups.
* Set are dynamic, so there is no error of overflowing of the set.
* Searching operation takes O(logN) time complexity.
* Sets provide fast and efficient operations for checking if an element is present in the set or not.

**Cons:**

* Elements in a set can only be accessed with pointers
* very complex to implement because of its structure and properties
* A set takes O(logN) time complexity for basic operations like insertion and deletion.
* Not suitable for large data sets.
* Sets can only store elements of a specific data type.

## **Unordered Set**

### **Overview**

* UNORDERED\_SET NOTES (#include <unordered\_set>, #include <functional>)
* Data Structures that contain unique set of elements.
* Hash function needed to insert elements into buckets.
* Average constant time for insert, delete, and access.
* Why? Unique collection with fast insertion and removal.
* Unordered\_set template definition: template< class Key, class Hash<Key>, class KeyEqual'<Key>, class Allocator<Key>> class unordered\_set;
* class Hash = std::hash<Key> : default class Hash value | needs #include<functional>
* class KeyEqual' = std::equal\_to<Key> : default class KeyEqual' value
* class Allocator = std::allocator<Key> : default class Allocator value
* Search, removal, and insertion times are constant.
* Standard Set Functions (available in unordered\_set):
* size(), =, clear(), count(), find(), empty(), insert(key), insert(it1, it2),
* insert(initializer\_list), erase(it), erase(it1, it2), erase(key)

Unordered\_Set specific HashTable functions:

bucket\_count() - Maps to range

load\_factor() - on average, each bucket includes how many elements

Different of insert(key) for different Set types:

* Dls
* dskd

### **Syntax**

// Declare and Initialize

int arr[5];

int arr[] = { 1, 2, 3, 4, 5 };

int arr[5] = { 1, 2, 3, 4, 5 };

// Access and Assign

int arr[5];

// Insert and Remove

int arr[5];

### **Code Example**

// Unordered\_Set.cpp | https://www.youtube.com/watch?v=g5RGA50LGCA

// Associative container which contains a set of unique objects of of type, 'Key'.

// Implemented as a Hash Table.

#include <iostream>

#include <string>

#include <vector>

#include <unordered\_set>

#include <functional>

using namespace std;

// for custom classes, need to create your own comparator.

class Student {

public:

int id;

string name;

bool operator==(const Student& s) const {

return (this->id == s.id && this->name == s.name);

}

void print\_student() const {

cout << "id: " << id << ", name: " << name << endl;

}

};

class StudentHashFunction {

public:

size\_t operator()(const Student& s) const {

return (hash<int>{}(s.id) + hash<string>{}(s.name) );

}

};

int main()

{

// ### HASHING ###

size\_t h1 = hash<string>{}("Hello");

size\_t h2 = hash<string>{}("World");

cout << h1 << ", " << h2 << endl;

cout << hash<int>{}(100) << endl;

cout << hash<float>{}(100.5) << endl;

unordered\_set<int> us = { 5, 10, 4, 20, 5, 5, 15 };

for (int x : us)

cout << x << " "; // prints out in random order

cout << endl;

cout << "size: " << us.size() << endl;

cout << "count of 5: " << us.count(5) << endl;

cout << "Key 5 erased? " << us.erase(5) << endl;

cout << boolalpha << "found 16 = " << (us.find(16) != us.end()) << endl;

//// Load\_factor() & Num\_Buckets()

cout << "number of buckets: " << us.bucket\_count() << endl;

cout << "load factor: " << us.load\_factor() << endl;

unordered\_set<Student, StudentHashFunction> uss = { {50, "Simon"}, {20, "Thomas"} };

for (auto& st : uss)

st.print\_student();

system("pause");

}

### **Use Cases**

* When the exact size is known at compile time, upper bound is known and doesn’t waste space.
* Days in the week, months in a year, number of grades of school (k-12).

**Pros:**

* Element access is fast – ptr arithmetic.
* Low overhead
* Direct conversion to pointers

**Cons:**

* Fixed size
* Inflexible, limited optimal uses
* Inserting and deleting elements at the beginning or middle can be inefficient

## **MultiSet**

### **Overview**

* Associative container that contains a sorted set of non-unique objects of type, 'Key'.
* Multiple elements consisting of the same value can exist more than once.
* User-Provided Compare() can be supplied to change ordering (sorting).Search, removal, and insertion are logarithmic time. O(log\*n)
* Implemented as Red-Black trees
* Definiton template: template<class Key, class Compare<Key>, class Allocator<Key>> class multiset.
* size(), =, clear(), count(), find(), empty(), insert(key), insert(it1, it2), insert(initializer\_list),
* erase(it), erase(it1, it2), erase(key), upper\_bound(key), lower\_bound(key)
* insert(key) - inserts key (if already present, inserts at upper\_bound and RETURNS iterator to inserted element)
* insert(it1, it2) - inserts element in range [it1, it2) of a vector or other linearly ordered container.
* insert(initializer\_list) - insert( {1,2,3} ).
* erase(key) - removes all element with key provided.
* erase(iterator pos) - removes element at position pos
* erase(iterator pos1, iterator pos2) - removes all element in range [pos1, pos2) - pos2 is up to but not including
* bool contains(key) - C++v20 - returns true or false if key exists in multiset.

**Syntax**

// Declare and Initialize

multiset<int> s = { 10, 20, 5, 10, 15, 20, 4 };

// Access and Assign

int arr[5];

// Insert and Remove

int arr[5];

### **Code Example**

// Multi\_Set.cpp | <https://www.youtube.com/watch?v=iJCnqHrkPq8>

#include <iostream>

#include <string>

#include <vector>

#include <set>

using namespace std;

void STL\_Set\_example();

void STL\_Multi\_Set\_example();

class Student

{

public:

int id;

string name;

void print\_Student() const

{

cout << "[ name = " << name << ", id = " << id << "]" << endl;

}

// user-defined comparator

bool operator < (const Student& other) const

{

return (this->id < other.id);

}

};

int main()

{

STL\_Set\_example();

STL\_Multi\_Set\_example();

system("pause");

}

void STL\_Set\_example()

{

set<int> s = { 10, 20, 5, 10, 15, 20, 4 };

cout << "Size: " << s.size() << endl;

s.insert(100);

s.insert(10);

cout << "Size: " << s.size() << endl;

for (auto& element : s)

cout << element << " ";

cout << endl;

//set<int>::iterator it = s.erase(s.find(10));

//cout << \*it << endl;

int num\_erased = s.erase(10);

cout << "number erased: " << num\_erased << endl;

auto ub = s.upper\_bound(15);

auto lb = s.lower\_bound(15);

cout << "upper bound of 15: " << \*ub << endl;

cout << "lower bound of 15: " << \*lb << endl;

s.insert({ -10, -30, -20 });

for (auto& element : s)

cout << element << " ";

cout << endl;

vector<int> v = { 10, 20, 15, 5, 4 };

s.insert(v.begin(), v.end());

for (auto& element : s)

cout << element << " ";

cout << endl;

set<Student> stdSet = { {50, "Simon"}, {20, "Thomas"} };

for (auto& st : stdSet)

st.print\_Student();

}

void STL\_Multi\_Set\_example()

{

multiset<int> s = { 10, 20, 5, 10, 15, 20, 4 };

cout << "Size: " << s.size() << endl;

s.insert(100);

s.insert(10);

cout << "Size: " << s.size() << endl;

for (auto& element : s)

cout << element << " ";

cout << endl;

//set<int>::iterator it = s.erase(s.find(10));

//cout << \*it << endl;

int num\_erased = s.erase(10);

cout << "number erased: " << num\_erased << endl;

auto ub = s.upper\_bound(15);

auto lb = s.lower\_bound(15);

cout << "upper bound of 15: " << \*ub << endl;

cout << "lower bound of 15: " << \*lb << endl;

s.insert({ -10, -30, -20 });

for (auto& element : s)

cout << element << " ";

cout << endl;

vector<int> v = { 10, 20, 15, 5, 4 };

s.insert(v.begin(), v.end());

for (auto& element : s)

cout << element << " ";

cout << endl;

multiset<Student> stdSet = { {50, "Simon"}, {20, "Thomas"}, {50, "Simon\_jr"} };

for (auto& st : stdSet)

st.print\_Student();

}

### **Use Cases**

* When the exact size is known at compile time, upper bound is known and doesn’t waste space.
* Days in the week, months in a year, number of grades of school (k-12).

**Pros:**

* Element access is fast – ptr arithmetic.
* Low overhead
* Direct conversion to pointers

**Cons:**

* Fixed sizesds
* Sds

## **Unordered MultiSet**

### **Overview**

// Unordered\_set template definition:

// template< class Key, class Hash<Key>, class KeyEqual'<Key>, class Allocator<Key>> class unordered\_set;

//

// class Hash = std::hash<Key> : default class Hash value | needs #include<functional>

// class KeyEqual' = std::equal\_to<Key> : default class KeyEqual' value

// class Allocator = std::allocator<Key> : default class Allocator value

//

// Standard Set Functions(available in unordered\_set) :

// size(), =, clear(), count(), find(), equal\_range(), empty(), insert(key), insert(it1, it2),

// insert(initializer\_list), erase(it), erase(it1, it2), erase(key)

//

// Unordered\_Set specific HashTable functions :

// bucket\_count() - Maps to range

// load\_factor() - on average, each bucket includes how many elements

//

// long chains of one bucket detoriates unordered sets to a linked list, slow

// equal\_range() - returns 2 iterators, it1 = set.begin(), it2 = set.end(). pair of iterators.

### **Syntax**

// Declare and Initialize

int arr[5];

int arr[] = { 1, 2, 3, 4, 5 };

int arr[5] = { 1, 2, 3, 4, 5 };

// Access and Assign

int arr[5];

// Insert and Remove

int arr[5];

### **Code Example**

// Unordered\_multiset.cpp | https://www.youtube.com/watch?v=AdCryKn9T9o

// Associative container that contains a set of non-unique objects of type 'key'.

// Search, removal, and insertion are constant time on average

// Implemented as a HASH TABLE.

#include <iostream>

#include <string>

#include <vector>

#include <unordered\_set>

#include <functional>

using namespace std;

// for custom classes, need to create your own comparator.

class Student {

public:

int id;

string name;

bool operator==(const Student& s) const {

return (this->id == s.id && this->name == s.name);

}

void print\_student() const {

cout << "id: " << id << ", name: " << name << endl;

}

};

class StudentHashFunction {

public:

size\_t operator()(const Student& s) const {

return (hash<int>{}(s.id) + hash<string>{}(s.name));

}

};

int main()

{

unordered\_multiset<int> us = { 5, 10, 4, 20, 5, 5, 15 };

for (int x : us)

cout << x << " "; // prints out in random order

cout << endl;

auto its = us.equal\_range(5); // equal\_range

for (auto it = its.first; it != its.second; it++)

{

cout << \*it << " ";

}

cout << endl;

cout << "size: " << us.size() << endl;

cout << "count of 5: " << us.count(5) << endl;

cout << "Key 5 erased? " << us.erase(5) << endl;

cout << boolalpha << "found 16 = " << (us.find(16) != us.end()) << endl;

//// Load\_factor() & Num\_Buckets()

cout << "number of buckets: " << us.bucket\_count() << endl;

cout << "load factor: " << us.load\_factor() << endl;

unordered\_set<Student, StudentHashFunction> uss = { {50, "Simon"}, {20, "Thomas"} };

for (auto& st : uss)

st.print\_student();

system("pause");

}

### **Use Cases**

* When the exact size is known at compile time, upper bound is known and doesn’t waste space.
* Days in the week, months in a year, number of grades of school (k-12).

**Pros:**

* Element access is fast – ptr arithmetic.
* Low overhead
* Direct conversion to pointers

**Cons:**

* Fixed sizesds
* sds

# **Classic Data Structures**

## **Array Class** (custom)

### **Overview**

* Hash Table with the ability to have duplicate keys.

### **Syntax**

// Declare and Initialize

unordered\_multimap<char, int> multiMap;

### **Code Examples**

// customArray.cpp : https://www.youtube.com/watch?v=TzB5ZeKQIHM

// Stack Allocated custom array class - not dynamic

// STD::array class uses a template.

#include <iostream>

#include <string>

using namespace std;

//template<size\_t S> | cArray<5> data;

template<typename T, size\_t S>

class cArray

{

public:

// no storage for the size, no additional storage optimization

// adding constexpr means this function can be evaluated at compile time.

// Generally better to use size\_t instead of ints for Array types

constexpr int Size() const { return S; }

// Simple index operator overload:

// T operator[](int index) { return m\_Data[index]; }

// Passes by value - makes a copy - inefficient

// Also, because it makes a copy, and returns by value, you can't use assignment:

// - data[i] = 2 won't compile; data[i] isn't modifiable lvalue

// - returning a brand new copy, nothing to assign to, no storage.

// implement the index operator to return specific array elements.

// by returning by reference, we can assign into that index.

T& operator[](int index) { return m\_Data[index]; } //read-only reference.

// adding this 2nd overloaded operator function,

// we can use this custom array class as a const array, allowing class

// to read data and not write it. First operator func doesn't

// return a const value, thus using first operator function won't read from

// const array using this customArray class.

const T& operator[](int index) const { return m\_Data[index]; }

// this allows you to use memset to set all elements to a value.

T\* Data() { return m\_Data; }

const T\* Data() const { return m\_Data; }

private:

T m\_Data[S];

};

int main()

{

cArray<int, 5> data{};

memset(data.Data(), 0, data.Size() \* sizeof(int));

data[0] = 3;

data[1] = 6;

cout << "Custom Array Size: " << data.Size() << endl;

// Size needs to have the constexpr keyword added to return

// to have 'static\_assert' be able to be evaluated at compile time.

static\_assert(data.Size() < 10, "Size is too large!");

// since Size() is a constexpr, it can be evaluated at compile time.

cArray<string, data.Size()> newArray;

for (int i = 0; i < data.Size(); i++)

{

cout << data[i] << endl;

}

system("pause");

}

### **Use Cases**

* Maintain collection of duplicate {key:value} pairs with fast insertion/removal.

**Pros:**

* Maintain collection of duplicate {key:value} pairs with fast insertion/removal.

**Cons:**

* Complexity

## **Graph**

### **Overview**

// WeightedGraph Notes:

// Math Graph terms:

// Graph G is an ordered pair of a set Vertices V and Edges E. G = (V, E). Mathematical

// Ordered Pair: (a, b) != (b,a) if a doesn't equal b. Pair of mathematical objects. Order DOES matters.

// Unordered Pair: {a, b} = {b, a}. Order DOESN't matter.

// Directed edge is an ordered pair (origin, dest) one way.

// 2 Directed edges are bidirectional connection between ordered pairs. (origin, dest) and (dest, origin)

// Undirected edge is also bidirectional but is represented as {origin, dest} because it's an unordered pair.

// Social network is an Undirected graph. Friendship is two ways and order doesn't matter.

// Web pages can be represented as a directed graph. The relationship isn't mutual. Co-links between pages not required.

// Web crawling (search engines) use graph traversal for search queries.

// Weighted Graph Notes:

// All connections cannot be treated as equals.

// Intercity road network is an example of a weighted graph (undirected for highways).

// Associate weight or cost for each edge.

// Unweighted graph is a type of weighted graph where all edges are weighted as 1 (the same).

// QUESTIONS

// When to use a graph? When to use a standard graph vs a weighted graph?

// Do weighted maps always use unordered\_maps?

// Any tree is also a graph? Yes. Graph is a parent of a tree.

// Any graph also a tree? Nope. tree is child, Graph is parent.

// When you index into a map, is that the same thing as creating a <pair> object and inserting it into the map?

// How to represents an edge? A pair of two endpoints.

// Hashmap vs. an unordered\_map?

// ADJACENCY GRAPH EXAMPLE

// (0) Node --- (1) Node

// \ /

// (2) Node

// |

// |

// (3) Node

// ARRAY of Lists of Size v

// 0 -> 1,2 : List L1

// 1 -> 2,0 : List L2

// 2 -> 3, 0, 1 : List L3

// 3 -> 2 : List L4

### **Syntax**

// Declare and Initialize

unordered\_multimap<char, int> multiMap;

### **Code Examples**

**// AdjList Graph** : [https://www.youtube.com/watch?v=dhgKr8942rs](https://www.youtube.com/watch?v=dhgKr8942rs%20)

**// Weighted Graph** : <https://www.youtube.com/watch?v=drpdVQq5-mk>

#include <iostream>

#include <list>

#include <unordered\_map>

using namespace std;

**// ADJACENCY LIST GRAPH CLASS**

class AdjList\_Graph

{

public:

AdjList\_Graph(int size)

{

this->m\_Size = size;

listArray = new list<int>[size];

**// list<int>[size]; list<int>(size);**

}

**// ASSUMES bidirectional edge**

void addEdge(int x, int y)

{

listArray[x].push\_back(y);

listArray[y].push\_back(x);

}

void printAdjList()

{

for (int i = 0; i < m\_Size; i++)

{

cout << "Vertex " << i << "->";

for (int neighbor : listArray[i])

{

cout << neighbor << ",";

}

cout << endl;

}

for (int i = 0; i < m\_Size; i++)

{

cout << "Vertex " << i << "->";

list<int>::iterator it = (listArray[i]).begin();

for(; it != listArray[i].end(); it++)

cout << \*it << ", ";

}

}

private:

int m\_Size;

**// array of lists of size m\_Size | size not known at compile time.**

**// Each array index contains a list of nodes that are adjacent to that array's index number.**

**// listArray[0] = {1, 2}; for example. See below. Node 0 has Node's 1 and 2 adjacent,**

**// therefor, they are the list elements associated with the array's 0 index.**

list<int>\* listArray = nullptr;

};

**// WEIGHTED GRAPH CLASS**

class WeightedGraph

{

public:

// Pass in what type of edge as a bool, unidirectional or bidirectional.

void addEdge(string x, string y, bool bidirectional, int weight)

{

**// Map of x, the value of the map at string x.**

**// The list associated with the string x.**

**// That list can perform the operation directly with .push\_back**

**// inserting a new pair into that specific list.**

myWeightedGraph[x].push\_back(make\_pair(y, weight));

**// true = bidirectional, false = unidirectional**

if (bidirectional) // if bidirectional, x->y and y->x

myWeightedGraph[y].push\_back(make\_pair(x, weight));

}

void printWeightedGraph()

{

for (auto& mapIndex : myWeightedGraph)

{

string city = mapIndex.first;

list<pair<string, int>> neighbors = mapIndex.second;

for (auto nbr : neighbors)

cout << city << "->" << nbr.first << ", Distance: " << nbr.second << endl;

}

private:

**// Adj list of an unordered\_map, Hash Table**

unordered\_map<string, list<pair<string, int>>> myWeightedGraph;

};

int main()

{

**// Adjacency List Graph | Array of 4 Lists or Buckets**

AdjList\_Graph g(4);

g.addEdge(0, 1);

g.addEdge(0, 2);

g.addEdge(2, 3);

g.addEdge(1, 2);

//g.printAdjList();

**///////////////////////////////////////////////////////////////**

**// unordered\_map<string, list<pair<string, int>>>**

WeightedGraph g2;

**// Need to pass what type of edge it is, unidirectional or bidirectional**

**// true = bidirectional, false = unidirectional**

g2.addEdge("A", "B", true, 20);

g2.addEdge("B", "D", true, 30);

g2.addEdge("A", "C", true, 10);

g2.addEdge("C", "D", true, 40);

g2.addEdge("A", "D", false, 50);

g2.printWeightedGraph();

system("pause");

}

### **Use Cases**

* Maintain collection of duplicate {key:value} pairs with fast insertion/removal.

**Pros:**

* Maintain collection of duplicate {key:value} pairs with fast insertion/removal.

**Cons:**

* Complexity

## **Hash Table**

### **Overview**

* Hash Table with the ability to have duplicate keys.

A concrete data structure that store items with identifiers (key-value pairs). The identifier (or key) is hashed using a hashing algorithm, hence, the key-value pair doesn't require a literal key/identifier declaration. The object itself is the key. Map interface is abstract data type.

* **Hashing** is a blanket term for taking an input and then producing an output to identify it with.
* **Hashtable** is a data structure that uses a hash to store data which is produced from a provided key, or from the object itself. This key-value pair determines where in the table the object is stored.
* **Hashmaps** are similar to **hashtables**, but only one example of each object is stored in it (hence no key needs to be provided, the object itself is the key).

### **Syntax**

// Declare and Initialize

unordered\_multimap<char, int> multiMap;

### **Code Examples**

// HashTable.cpp | https://www.youtube.com/watch?v=2\_3fR-k-LzI&std::list=PLm0fFUL4gEt9QZ5gGl63ty4qSwrmzVvH\_&index=13

//

#include <iostream>

#include <list>

#include <string>

class HashTable // Hashtable phonebook | Array of Lists[10]

{

private:

static const int hashGroups = 10; // # of hashed indexes, buckets, or lists.

std::list <std::pair<int, std::string>> table[hashGroups]; // each list has 0 or more pairs

public:

int hashFunction(int key);

bool isEmpty() const;

void insertItem(int key, std::string value);

void removeItem(int key);

std::string searchTable(int key);

void printTable();

};

void CheckEmpty(HashTable hash)

{

if (hash.isEmpty())

std::cout << "Is Empty." << std::endl;

else

std::cout << "Not Empty" << std::endl;

}

int main()

{

HashTable ht;

CheckEmpty(ht);

ht.insertItem(905, "Jim");

ht.insertItem(201, "Tom");

ht.insertItem(332, "Bob");

ht.insertItem(124, "Sally");

ht.insertItem(107, "Sandy");

ht.insertItem(929, "Barb");

ht.insertItem(928, "Rob");

ht.insertItem(928, "Rick");

ht.printTable();

ht.removeItem(332);

ht.removeItem(100);

CheckEmpty(ht);

system("pause");

}

bool HashTable::isEmpty() const

{

int sum = 0;

for (int i = 0; i < hashGroups; i++)

{

sum += table[i].size(); // Check the size of each list, bucket

}

if (!sum) // if sum is still 0. return true for empty.

return true;

return false;

}

// Since only 10 buckets, we want to return a value between 0-9.

int HashTable::hashFunction(int key)

{

return key % hashGroups; // Key: 905, in return, this function will spit out 5 (5th index).

}

void HashTable::insertItem(int key, std::string value)

{

int hashValue = hashFunction(key); // which list index this key-value pair needs to go into.

std::list<std::pair<int, std::string>>& cell = table[hashValue]; // table at a given index

// auto& cell = table[hashValue];

bool keyExists = false; // Why are we iterating, why not just push? Need to know if key exists.

for (auto bItr = begin(cell); bItr != end(cell); bItr++)

{

if (bItr->first == key)

{

keyExists = true;

bItr->second = value;

std::cout << "Warning: Key exists. Value replaced." << std::endl;

break;

}

}

if (!keyExists)

{

cell.emplace\_back(key, value);

}

return;

}

void HashTable::removeItem(int key)

{

int hashValue = hashFunction(key);

auto& cell = table[hashValue];

bool keyExists = false;

for (auto bItr = begin(cell); bItr != end(cell); bItr++)

{

if (bItr->first == key)

{

keyExists = true;

bItr = cell.erase(bItr); // reuse old iterator after removing key, value pair.

std::cout << "Warning: Item Removed." << std::endl;

break;

}

}

if (!keyExists)

{

std::cout << "Item Not Found" << std::endl;

}

return;

}

std::string HashTable::searchTable(int key)

{

return std::string();

}

void HashTable::printTable()

{

for (int i = 0; i < hashGroups; i++)

{

if (table[i].size() == 0) continue;

auto bItr = table[i].begin();

for (; bItr != table[i].end(); bItr++)

{

std::cout << "Key: " << bItr->first << " Value: " << bItr->second << std::endl;

}

}

return;

}

// Hash Table is a 'key' => 'value' lookup.

// Hash Tables require a HASH FUNCTION

// O(1). Getting and Setting.

// ## Example ## - Store people objects in an array

// key ("std::string") => value[Person object]

// How do we go from a std::string to an index in an array? Hash function

// A function that passes in a std::string and returns some integer.

// Remaps an integer into an index of that array | key -> to hashcode -> array index.

// Array that stores data might be much smaller than all potential hash codes. Kind of like a 2D array

// After re-mapping to array indexes, each index will contain the original "name" std::string key and object value.

// So each index in the remapped array will contain at least one map element {"name", Person()}.

// Two std::strings could have same hash code, infinite # of std::strings, finite # of hash codes.

// Two std::strings with two different hash codes could wind up with same index. COLLISION.

// When COLLISION happens, use CHAINING, store them in a LinkedList.

// Chaining v Open Addressing : 2 ways to deal with COLLISION.

/\*

class Hastable

{

linkedstd::list[] data;

boolean put(std::string key, Person value)

{

int hashcode = getHashCode(key)

int index = convertToIndex(hashcode)

linkedstd::list std::list = data[index]

std::list.insert(key, value);

}

}

\*/

// ### QUESTIONS ###

// HashTable v HashMap v std::map v std::unordered\_map v std::set? When, why, how?

// Do you use HashTable and std::unordered\_maps together in C++?

// Do you use HashTables with a LL, typically?

// How are hashtables different in C++ v other languages?

// What is reflection? How is it used?

// When you don't use linked std::lists as the values in hash table, would it be a single map or multiple maps?

// When you do use a linked std::list as the values in hash table, how does the stl map work there?

// When to use static and when not to use static?

// pairs v maps?

### **Use Cases**

* Maintain collection of duplicate {key:value} pairs with fast insertion/removal.

**Pros:**

* Maintain collection of duplicate {key:value} pairs with fast insertion/removal.

**Cons:**

* Complexity

## **Iterator**

### **Overview**

* Hash Table with the ability to have duplicate keys.

### **Syntax**

// Declare and Initialize

unordered\_multimap<char, int> multiMap;

### **Code Examples**

// CustomIterator.cpp // https://www.youtube.com/watch?v=F9eDv-YIOQ0

// An Iterator is a 'gang of four' behavioral design pattern.

// Iterators are objects used for iterating over data structures.

//

// Create Iterator class, define begin and end in custom Vector class, start constructor, adding ValueType to Vector and Vector Iterator class.

// look up original vector class, look at Valuetype, use using to define Valuetype specifying Vector,

// and create PointerType and RefType based off that value type,

// finish constructor with PointerType, define a member variable of PointerType, fill out rest of itr functions.

// Prefix, postfix, == operator, != operator, index operator, dereference operator, -> operator functions.

// Technically don't need to make a custom iterator class for a vector because you can index into a vector,

// but this iterator concept translates to any data structure.

#include <iostream>

#include <string>

#include <vector>

using namespace std;

// ##### VERTEX STRUCT #####

struct Vertex

{

float x = 0.0f, y = 0.0f, z = 0.0f;

Vertex() {}

Vertex(float scalar) : x(scalar), y(scalar), z(scalar) {}

Vertex(float x, float y, float z) : x(x), y(y), z(z) {}

//Vertex(Vertex&& other) {};

Vertex(const Vertex& other) : x(other.x), y(other.y), z(other.z)

{

cout << "Copy\n";

}

Vertex (Vertex&& other) noexcept : x(other.x), y(other.y), z(other.z)

{

cout << "Move\n";

}

~Vertex()

{

cout << "Destroy\n";

}

Vertex& operator=(const Vertex& other)

{

cout << "Copy\n";

x = other.x;

y = other.y;

z = other.z;

return \*this;

}

Vertex& operator=(Vertex&& other) noexcept

{

cout << "Move\n";

x = other.x;

y = other.y;

z = other.z;

return \*this;

}

};

// ##### CUSTOM ITERATOR #####

template<typename Vector>

class VectorIterator

{

public:

// ValueType isn't it's own thing in Iterator class, it's an alias, or a

// reference to the actual Valuetype variable in the custom Vector DS.

// You use the same variable name, Valuetype in iterator and actual Vector class to reference

// the template primitive variable type that is passed in, T, during construction.

using ValueType = typename Vector::ValueType;

using PointerType = ValueType\*;

using ReferenceType = ValueType&;

public:

// needs to take in some kind of ptr.

VectorIterator(PointerType ptr) : m\_Ptr(ptr) {}

VectorIterator& operator++()

{

m\_Ptr++;

return \*this; // vector iterator reference.

}

VectorIterator operator++(int)

{

VectorIterator iterator = \*this;

++(\*this); // calls the operator++() above

return iterator;

}

VectorIterator& operator--()

{

m\_Ptr--;

return \*this; // vector iterator reference.

}

VectorIterator operator--(int)

{

VectorIterator iterator = \*this;

--(\*this); // calls the operator--() directly above

return iterator;

}

ReferenceType operator[](int index)

{

return \*(m\_Ptr + index);

}

PointerType operator->()

{

return m\_Ptr;

}

ReferenceType operator\*()

{

return \*m\_Ptr;

}

bool operator==(const VectorIterator& other) const

{

return m\_Ptr == other.m\_Ptr;

}

bool operator!=(const VectorIterator& other) const

{

return !(\*this == other);

}

private:

// PointerType was defined as the ValueType\* above

PointerType m\_Ptr;

};

// ##### CUSTOM VECTOR CLASS #####

template<typename T>

class Vector

{

public:

// What is a ValueType? Variable name substitution?

// ValueType is a variable that represents the type of primitive variable, T, passed in when creating a Vector data structure.

// Using is an alias that introduces a name that is defined elsewhere into the declarative region where this using-declaration appears.

// Brings a specific member from the namespace into the current scope.

using ValueType = T;

// aliasing class name to smaller name, Iterator, and adding an iterator to the vector class for use.

// don't have to use name VectorIterator everywhere throughout out code.

using Iterator = VectorIterator<Vector<ValueType>>; // Iterator is the type we created in this class.

// VectorIterator<Vector<ValueType>> Iterator; // why not?

public:

Vector()

{ // allocate memory to store 2 elements

ReAlloc(2);

}

~Vector()

{

// delete[] m\_Data; Problem

}

void PushBack(const T& value)

{

if (m\_Size >= m\_Capacity)

{

ReAlloc(m\_Capacity + m\_Capacity / 2);

}

m\_Data[m\_Size] = value;

m\_Size++;

}

void PushBack(T&& value) // What does && mean? this is an lvalue

{

if (m\_Size >= m\_Capacity)

{

ReAlloc(m\_Capacity + m\_Capacity / 2);

}

m\_Data[m\_Size] = move(value); // move is actually a cast, casts an rvalue reference to (value)

m\_Size++;

}

template<typename... Args>

T& EmplaceBack(Args&&... args)

{

if (m\_Size >= m\_Capacity)

{

ReAlloc(m\_Capacity + m\_Capacity / 2);

}

new(&m\_Data[m\_Size]) T(forward<Args>(args)...);

// m\_Data[m\_Size] = T(forward<Args>(args)...);

return m\_Data[m\_Size++];

}

void PopBack()

{

if (m\_Size > 0)

{

m\_Size--;

m\_Data[m\_Size].~T();

}

}

void Clear()

{

for (size\_t i = 0; i < m\_Size; i++)

m\_Data[i].~T();

m\_Size = 0;

}

const T& operator[](size\_t index) const

{

return m\_Data[index];

}

T& operator[](size\_t index)

{

return m\_Data[index]; // add asserts in full version, checking size

}

size\_t Size() const { return m\_Size; }

Iterator begin()

{

// Need to take in a vector iterator over this current Vector class, copy the std library for std::vector.

return Iterator(m\_Data);

}

Iterator end()

{

return Iterator(m\_Data + m\_Size);

}

private:

void ReAlloc(size\_t newCapacity) // Don't use smart pointers when you are this low level

{

// 1. allocate a new block of memory.

// 2. copy all existing elements into new block of memory (try to move them).

// 3. Delete old block of memory

T\* newBlock = new T[newCapacity];

// checks to see if we are shrinking Vector

if (newCapacity < m\_Size) // change the size of the vector if size is smaller

m\_Size = newCapacity; // \*Note: Not typically implemented in the ReAlloc function.

for (size\_t i = 0; i < m\_Size; i++) // if check above accounts for shrinking vector.

{

newBlock[i] = move(m\_Data[i]);

}

delete[] m\_Data;

m\_Data = newBlock;

m\_Capacity = newCapacity;

}

private:

T\* m\_Data = nullptr;

size\_t m\_Size = 0; // num of elements currently in Vector

size\_t m\_Capacity = 0; // num of elements total that could be stored, total memory allocated

};

// PRINT FUNCTIONS EXTERNAL TO CUSTOM VECTOR CLASS

template<typename T>

void PrintVector(const Vector<T>& vector)

{

for (size\_t i = 0; i < vector.Size(); i++)

{

cout << vector[i] << endl;

}

cout << "---------------------------\n";

}

template<>

void PrintVector(const Vector<Vertex>& vector)

{

for (size\_t i = 0; i < vector.Size(); i++)

{

cout << vector[i].x << ", " << vector[i].y << ", " << vector[i].z << endl;

}

cout << "---------------------------\n";

}

int main()

{

Vector<int> values;

values.EmplaceBack(1);

values.EmplaceBack(2);

values.EmplaceBack(3);

values.EmplaceBack(4);

values.EmplaceBack(5);

// Not using iterators - index operator defined in our custom "Vector" class.

cout << "Std index based operators:\n";

for (int i = 0; i < values.Size(); i++)

{

cout << values[i] << endl;

}

// Range-Based

cout << "\nRange-based for loop:\n";

for (int value : values) // begin() and end() added to Vector class

{

cout << value << endl;

}

cout << "\nIterator Implementation:\n";

for (Vector<int>::Iterator it = values.begin(); it != values.end(); it++)

cout << \*it << endl;

cout << "\n";

system("pause");

}

### **Use Cases**

* Maintain collection of duplicate {key:value} pairs with fast insertion/removal.

**Pros:**

* Maintain collection of duplicate {key:value} pairs with fast insertion/removal.

**Cons:**

* Complexity

## **List** (custom)

### **Overview**



### **Syntax**

// Declare and Initialize

### **Code Examples**

// LinkedList.cpp | https://www.youtube.com/watch?v=RNMIDj62o\_o&t=571s

//

// Linked List is a data structure containing a non-contigous collection of elements

// Linked List elements contain a value and a ptr variable to the next element in the list.

// Last element in the linked list has a null ptr.

//

// Linked Lists are better than arrays: No fixed size, dynamic data structure.

// Linked Lists are worse than arrays: No random access of elements, non-contigous, slow access.

// Linked Lists are worse than arrays: Uses more memory. Each element also has a ptr.

//

// Linked Lists are used when you are frequently adding and removing elements.

// Linked Lists are NOT used when you are frequently accessing elements.

// Advantage to SLL: adding and removing nodes is faster (ability to add before and after a node).

// Disadvantage to Singly Linked Lists (SLL): takes more memory.

//

// Link List nodes use pointers because Linked Lists are dynamic,

// thus need pointers to dynamically adjust size.

#include<list>

// Node represents one element of a Linked List.

class Node

{

public:

int value;

Node\* next;

};

// Doubly Linked-List Node, same as LL, but can move backward and forward, take up more memory.

class Double\_Node

{

public:

int value;

Node\* next;

Node\* previous;

};

Node\* createNode();

void addNodeFront(Node\*\* head, int val); // How do ptr of ptrs work?

void addNodeBack(Node\*\* head, int val);

void addNodeMiddle(Node\* previous, int val);

void printList(Node\* n);

void basic\_LL\_Example();

void creatingAndAddingNodes\_Example();

#include <iostream>

#include <string>

#include <list>

int main()

{

basic\_LL\_Example();

creatingAndAddingNodes\_Example();

// User-defined LL (above) vs an STL List

std::list<int> myList;

std::list<int> myList2{ 1,2,3,4,5 };

system("pause");

}

Node\* createNode()

{

Node\* head = nullptr;

head = new Node;

Node\* second = new Node();

Node\* third = new Node();

(\*head).value = 1; // head->value = 1;

(\*second).value = 2; // second->value = 2;

(\*third).value = 3; // third->value = 3;

(\*head).next = second; // head->next = second;

(\*second).next = third; // second->next = third;

(\*third).next = nullptr; // third->next = nullptr;

return head;

}

// ADDNODE: Pass Address of the head node variable, which is also a pointer,

// so, you need a pointer to a pointer, or a double pointer in order to modify the LL outside of main scope.

// Passing in head without a reference address, will only modify local pointer's pointee, which doesn't save after it leaves

// function scope

void addNodeFront(Node\*\* head, int val)

{

// Prepare New Node()

Node\* newNode = new Node();

(\*newNode).value = val;

// Put New Node in front of current Node | put new node in front of current head first. passing in current head's address to newNode next.

// De-reference head, which is passed in as a double pointer, so function dereferences a double ptr(\*\*) leaves just a ptr. (head pointer)

newNode->next = \*head; // newNode's ptr is pointing to current head of Linked List.

// Move head of the list to point to the newNode | second, redefine where the head of LL resides, which is new Node's address.

\*head = newNode;

}

void addNodeBack(Node\*\* head, int val)

{

// 1. Prepare a new Node

Node\* n = new Node();

(\*n).value = val;

(\*n).next = nullptr;

// 2. If Linked list is empty, the new Node\* n will be the head node

if (\*head == nullptr)

{

\*head = n;

return;

}

// 3. Find the last node (How do I create an iterator for a LL in this case Node\*\*?)

Node\* last = \*head; // allows you to navigate list without touching the head of LL.

while (last->next != nullptr)

{

last = last->next;

}

// for (std::list<Node\*>::iterator it = (\*head).begin();

// 4. Insert new Node after the last node (at the end of the list)

last->next = n;

}

void addNodeMiddle(Node\* previous, int val)

{

// 1. Check if previous node is NULL

if (previous == nullptr)

{

std::cout << "Previous cannot be NULL\n";

return;

}

// 2. Prepare new Node()

Node\* newNode = new Node();

newNode->value = val;

// cannot assign ptr at this point.

// 3. Insert newNode after previous

newNode->next = previous->next;

previous->next = newNode;

}

void printList(Node\* n)

{

while (n != nullptr)

{

std::cout << n->value << std::endl;

n = n->next;

}

}

void basic\_LL\_Example()

{

// First node of Linked List.

Node\* head = new Node();

Node\* second = new Node();

Node\* third = new Node();

head->value = 1;

head->next = second;

second->value = 2;

second->next = third;

third->value = 3;

third->next = NULL;

printList(head);

}

void creatingAndAddingNodes\_Example()

{

// First node of a Linked List.

Node\* head = createNode();

addNodeFront(&head, -1);

addNodeFront(&head, -2);

addNodeBack(&head, 4);

addNodeBack(&head, 5);

addNodeMiddle(head, -1);

addNodeMiddle(head->next->next, -3);

printList(head);

}

// Linked List Notes:

// Each element (or Node) of a linked list contains 2 parts: value of element, ptr to next element.

// LLs CAN NOT represent an element value of linked list with a built-in data type.

// LINKED List element values HAVE to be a user-defined data type (struct or class).

// Is there a purpose to pass a pointer by reference?

// You would want to pass a pointer by reference if you have a need to modify the pointer rather than the object that the pointer is pointing to.

// This is similar to why double pointers are used; using a reference to a pointer is slightly safer than using pointers.

// Deleting a ptr and null'n it out in local scope would only affect local pointer, not the one out of scope.

// Linked Lists v Maps? When you would use a LL over a Map?

### **Use Cases**

* .

**Pros:**

* Maintain collection of duplicate {key:value} pairs with fast insertion/removal.

**Cons:**

* Complexity

## **Stack**

### **Overview**

* Hash Table with the ability to have duplicate keys.

### **Syntax**

// Declare and Initialize

### **Code Examples**

// Stack.cpp | https://www.youtube.com/watch?v=yIBkQzL77aY&t=314s

// User-Defined Stack: LIFO

// Stacks are typically defined as a list,

// where the newest element is added to the front of the list

// and when an element is popped, it's removed from the front of the list as well.

#include <iostream>

#include <string>

using namespace std;

struct Node {

string data;

struct Node\* next;

};

class stack {

public:

stack(){}

~stack()

{

while (!isEmpty())

{

pop();

}

}

void push(string d)

{

Node\* tmp = new Node{ d, m\_top };

m\_top = tmp;

}

// pops top Node off of the stack

string pop()

{

if (!isEmpty())

{

Node\* tmp = m\_top;

m\_top = m\_top->next;

delete tmp;

return m\_top->data;

}

return "Empty";

}

void printStack(Node\* t)

{

while (t != NULL)

{

cout << t->data << ", ";

t = t->next;

}

cout << endl;

}

Node\* top()

{

return m\_top;

}

bool isEmpty() {

return (m\_top == NULL);

}

private:

Node\* m\_top = NULL;

};

int main()

{

// Does this need to be a pointer?

stack\* myStack = new stack();

myStack->push("CheeseBurger");

myStack->push("Pizza");

myStack->push("Large Coffee");

myStack->pop();

myStack->printStack(myStack->top());

delete myStack;

system("pause");

}

// Do you need to have a separate Stack class to implement user-defined Stack?

### **Use Cases**

* .

**Pros:**

* Maintain collection of duplicate {key:value} pairs with fast insertion/removal.

**Cons:**

* Complexity

## **Tree**

### **Binary Search Tree**

#### Overview

* Hash Table with the ability to have duplicate keys.

#### Syntax

// Declare and Initialize

unordered\_multimap<char, int> multiMap;

#### Code Examples

// customArray.cpp : https://www.youtube.com/watch?v=TzB5ZeKQIHM

// Stack Allocated custom array class - not dynamic

// STD::array class uses a template.

#include <iostream>

#include <string>

using namespace std;

//template<size\_t S> | cArray<5> data;

template<typename T, size\_t S>

class cArray

{

public:

// no storage for the size, no additional storage optimization

// adding constexpr means this function can be evaluated at compile time.

// Generally better to use size\_t instead of ints for Array types

constexpr int Size() const { return S; }

// Simple index operator overload:

// T operator[](int index) { return m\_Data[index]; }

// Passes by value - makes a copy - inefficient

// Also, because it makes a copy, and returns by value, you can't use assignment:

// - data[i] = 2 won't compile; data[i] isn't modifiable lvalue

// - returning a brand new copy, nothing to assign to, no storage.

// implement the index operator to return specific array elements.

// by returning by reference, we can assign into that index.

T& operator[](int index) { return m\_Data[index]; } //read-only reference.

// adding this 2nd overloaded operator function,

// we can use this custom array class as a const array, allowing class

// to read data and not write it. First operator func doesn't

// return a const value, thus using first operator function won't read from

// const array using this customArray class.

const T& operator[](int index) const { return m\_Data[index]; }

// this allows you to use memset to set all elements to a value.

T\* Data() { return m\_Data; }

const T\* Data() const { return m\_Data; }

private:

T m\_Data[S];

};

int main()

{

cArray<int, 5> data{};

memset(data.Data(), 0, data.Size() \* sizeof(int));

data[0] = 3;

data[1] = 6;

cout << "Custom Array Size: " << data.Size() << endl;

// Size needs to have the constexpr keyword added to return

// to have 'static\_assert' be able to be evaluated at compile time.

static\_assert(data.Size() < 10, "Size is too large!");

// since Size() is a constexpr, it can be evaluated at compile time.

cArray<string, data.Size()> newArray;

for (int i = 0; i < data.Size(); i++)

{

cout << data[i] << endl;

}

system("pause");

}

#### Use Cases

* Maintain collection of duplicate {key:value} pairs with fast insertion/removal.

**Pros:**

* Maintain collection of duplicate {key:value} pairs with fast insertion/removal.

**Cons:**

* Complexity

### **Red-Black Tree**

#### Overview

* Hash Table with the ability to have duplicate keys.

#### Syntax

// Declare and Initialize

unordered\_multimap<char, int> multiMap;

#### Code Examples

// customArray.cpp : https://www.youtube.com/watch?v=TzB5ZeKQIHM

// Stack Allocated custom array class - not dynamic

// STD::array class uses a template.

#include <iostream>

#include <string>

using namespace std;

//template<size\_t S> | cArray<5> data;

template<typename T, size\_t S>

class cArray

{

public:

// no storage for the size, no additional storage optimization

// adding constexpr means this function can be evaluated at compile time.

// Generally better to use size\_t instead of ints for Array types

constexpr int Size() const { return S; }

// Simple index operator overload:

// T operator[](int index) { return m\_Data[index]; }

// Passes by value - makes a copy - inefficient

// Also, because it makes a copy, and returns by value, you can't use assignment:

// - data[i] = 2 won't compile; data[i] isn't modifiable lvalue

// - returning a brand new copy, nothing to assign to, no storage.

// implement the index operator to return specific array elements.

// by returning by reference, we can assign into that index.

T& operator[](int index) { return m\_Data[index]; } //read-only reference.

// adding this 2nd overloaded operator function,

// we can use this custom array class as a const array, allowing class

// to read data and not write it. First operator func doesn't

// return a const value, thus using first operator function won't read from

// const array using this customArray class.

const T& operator[](int index) const { return m\_Data[index]; }

// this allows you to use memset to set all elements to a value.

T\* Data() { return m\_Data; }

const T\* Data() const { return m\_Data; }

private:

T m\_Data[S];

};

int main()

{

cArray<int, 5> data{};

memset(data.Data(), 0, data.Size() \* sizeof(int));

data[0] = 3;

data[1] = 6;

cout << "Custom Array Size: " << data.Size() << endl;

// Size needs to have the constexpr keyword added to return

// to have 'static\_assert' be able to be evaluated at compile time.

static\_assert(data.Size() < 10, "Size is too large!");

// since Size() is a constexpr, it can be evaluated at compile time.

cArray<string, data.Size()> newArray;

for (int i = 0; i < data.Size(); i++)

{

cout << data[i] << endl;

}

system("pause");

}

#### Use Cases

#### Maintain collection of duplicate {key:value} pairs with fast insertion/removal.

#### Pros:

#### Maintain collection of duplicate {key:value} pairs with fast insertion/removal.

#### Cons:

### **AVL Tree**

#### Overview

* Hash Table with the ability to have duplicate keys.

#### Syntax

// Declare and Initialize

unordered\_multimap<char, int> multiMap;

#### Code Examples

// customArray.cpp : https://www.youtube.com/watch?v=TzB5ZeKQIHM

// Stack Allocated custom array class - not dynamic

// STD::array class uses a template.

#include <iostream>

#include <string>

using namespace std;

//template<size\_t S> | cArray<5> data;

template<typename T, size\_t S>

class cArray

{

public:

// no storage for the size, no additional storage optimization

// adding constexpr means this function can be evaluated at compile time.

// Generally better to use size\_t instead of ints for Array types

constexpr int Size() const { return S; }

// Simple index operator overload:

// T operator[](int index) { return m\_Data[index]; }

// Passes by value - makes a copy - inefficient

// Also, because it makes a copy, and returns by value, you can't use assignment:

// - data[i] = 2 won't compile; data[i] isn't modifiable lvalue

// - returning a brand new copy, nothing to assign to, no storage.

// implement the index operator to return specific array elements.

// by returning by reference, we can assign into that index.

T& operator[](int index) { return m\_Data[index]; } //read-only reference.

// adding this 2nd overloaded operator function,

// we can use this custom array class as a const array, allowing class

// to read data and not write it. First operator func doesn't

// return a const value, thus using first operator function won't read from

// const array using this customArray class.

const T& operator[](int index) const { return m\_Data[index]; }

// this allows you to use memset to set all elements to a value.

T\* Data() { return m\_Data; }

const T\* Data() const { return m\_Data; }

private:

T m\_Data[S];

};

int main()

{

cArray<int, 5> data{};

memset(data.Data(), 0, data.Size() \* sizeof(int));

data[0] = 3;

data[1] = 6;

cout << "Custom Array Size: " << data.Size() << endl;

// Size needs to have the constexpr keyword added to return

// to have 'static\_assert' be able to be evaluated at compile time.

static\_assert(data.Size() < 10, "Size is too large!");

// since Size() is a constexpr, it can be evaluated at compile time.

cArray<string, data.Size()> newArray;

for (int i = 0; i < data.Size(); i++)

{

cout << data[i] << endl;

}

system("pause");

}

#### Use Cases

* Maintain collection of duplicate {key:value} pairs with fast insertion/removal.

**Pros:**

* Maintain collection of duplicate {key:value} pairs with fast insertion/removal.

**Cons:**

* Complexity

### **Heap**

#### Overview

* Hash Table with the ability to have duplicate keys.

#### Syntax

// Declare and Initialize

unordered\_multimap<char, int> multiMap;

#### Code Examples

// customArray.cpp : https://www.youtube.com/watch?v=TzB5ZeKQIHM

// Stack Allocated custom array class - not dynamic

// STD::array class uses a template.

#include <iostream>

#include <string>

using namespace std;

//template<size\_t S> | cArray<5> data;

template<typename T, size\_t S>

class cArray

{

public:

// no storage for the size, no additional storage optimization

// adding constexpr means this function can be evaluated at compile time.

// Generally better to use size\_t instead of ints for Array types

constexpr int Size() const { return S; }

// Simple index operator overload:

// T operator[](int index) { return m\_Data[index]; }

// Passes by value - makes a copy - inefficient

// Also, because it makes a copy, and returns by value, you can't use assignment:

// - data[i] = 2 won't compile; data[i] isn't modifiable lvalue

// - returning a brand new copy, nothing to assign to, no storage.

// implement the index operator to return specific array elements.

// by returning by reference, we can assign into that index.

T& operator[](int index) { return m\_Data[index]; } //read-only reference.

// adding this 2nd overloaded operator function,

// we can use this custom array class as a const array, allowing class

// to read data and not write it. First operator func doesn't

// return a const value, thus using first operator function won't read from

// const array using this customArray class.

const T& operator[](int index) const { return m\_Data[index]; }

// this allows you to use memset to set all elements to a value.

T\* Data() { return m\_Data; }

const T\* Data() const { return m\_Data; }

private:

T m\_Data[S];

};

int main()

{

cArray<int, 5> data{};

memset(data.Data(), 0, data.Size() \* sizeof(int));

data[0] = 3;

data[1] = 6;

cout << "Custom Array Size: " << data.Size() << endl;

// Size needs to have the constexpr keyword added to return

// to have 'static\_assert' be able to be evaluated at compile time.

static\_assert(data.Size() < 10, "Size is too large!");

// since Size() is a constexpr, it can be evaluated at compile time.

cArray<string, data.Size()> newArray;

for (int i = 0; i < data.Size(); i++)

{

cout << data[i] << endl;

}

system("pause");

}

#### Use Cases

#### Maintain collection of duplicate {key:value} pairs with fast insertion/removal.

#### Pros:

#### Maintain collection of duplicate {key:value} pairs with fast insertion/removal.

#### Cons:

#### Complexity

## **Trie**

### **Overview**

* Hash Table with the ability to have duplicate keys.

### **Syntax**

// Declare and Initialize

### **Code Examples**

### **Use Cases**

* .

**Pros:**

* Maintain collection of duplicate {key:value} pairs with fast insertion/removal.

**Cons:**

* Complexity

## **Vector**

### **Overview**

* Hash Table with the ability to have duplicate keys.

### **Syntax**

// Declare and Initialize

unordered\_multimap<char, int> multiMap;

### **Code Examples**

template<typename T>

class Vector

{

public:

Vector()

{ // allocate memory to store 2 elements

ReAlloc(2);

}

~Vector()

{

// delete[] m\_Data; Problem

}

void PushBack(const T& value)

{

if (m\_Size >= m\_Capacity)

{

ReAlloc(m\_Capacity + m\_Capacity / 2);

}

// m\_Data is a ptr/array for this class, don't need to define index operator

// to be able to index into class's interal array, already built into array's functionality

// You need to define index operator for entire vector class to index this class,

// not it's internal array. This works before defining the index operator.

m\_Data[m\_Size] = value;

m\_Size++;

}

// What does && mean? this is an lvalue

void PushBack(T&& value)

{

if (m\_Size >= m\_Capacity)

{

ReAlloc(m\_Capacity + m\_Capacity / 2);

}

// move is actually a cast, casts an rvalue reference to (value)

m\_Data[m\_Size] = std::move(value);

m\_Size++;

}

// Variatic Template

// Variable number of arguments

// Constructing something in place, need to return a T&, not easy to get object back

template<typename... Args>

T& EmplaceBack(Args&&... args)

{

if (m\_Size >= m\_Capacity)

{

ReAlloc(m\_Capacity + m\_Capacity / 2);

}

// constructing an object inside the m\_Data array, placement new

// grab the memory address of the object, and forward the arguments into the constructor

// new provides the actual memory to be constructed.

new(&m\_Data[m\_Size]) T(std::forward<Args>(args)...);

// m\_Data[m\_Size] = T(std::forward<Args>(args)...); // constructing then copying.

return m\_Data[m\_Size++];

}

void PopBack()

{

if (m\_Size > 0)

{

m\_Size--;

// manually calling the destructor | delete function also calls the destructor.

m\_Data[m\_Size].~T();

}

}

void Clear()

{

for(size\_t i = 0; i < m\_Size; i++)

m\_Data[i].~T();

m\_Size = 0;

}

const T& operator[](size\_t index) const

{

return m\_Data[index]; // add assert for debug mode.

}

T& operator[](size\_t index)

{

return m\_Data[index]; // add asserts in full version, checking size

}

size\_t Size() const { return m\_Size; }

private:

// Don't use smart pointers when you are this low level

void ReAlloc(size\_t newCapacity)

{

// 1. allocate a new block of memory.

// 2. copy all existing elements into new block of memory (try to move them).

// 3. Delete old block of memory

T\* newBlock = new T[newCapacity];

// checks to see if we are shrinking Vector, change if size is smaller.

// if we are downsizing, we only copy up to the new capacitySize.

if(newCapacity < m\_Size)

m\_Size = newCapacity;

for (size\_t i = 0; i < m\_Size; i++)

{

// Can't use memcpy because we need to hit the copy constructor for all the elements,

// Because they are complex objects being copied, the copy constructor for those objects are required.

newBlock[i] = std::move(m\_Data[i]);

}

delete [] m\_Data;

m\_Data = newBlock;

m\_Capacity = newCapacity;

}

private:

T\* m\_Data = nullptr;

size\_t m\_Size = 0; // num of elements currently in Vector

size\_t m\_Capacity = 0; // num of elements total that could be stored, total memory allocated

};

struct Vertex

{

float x = 0.0f, y = 0.0f, z = 0.0f;

Vertex() {}

Vertex(float scalar) : x(scalar), y(scalar), z(scalar) {}

Vertex(float x, float y, float z) : x(x), y(y), z(z) {}

//Vertex(Vertex&& other) {};

// COPY CONSTRUCTOR

Vertex(const Vertex& other) : x(other.x), y(other.y), z(other.z)

{

std::cout << "Copy\n";

}

// MOVE - doesn't use COPY CONSTRUCTOR

Vertex(Vertex&& other) noexcept : x(other.x), y(other.y), z(other.z)

{

std::cout << "Move\n";

}

~Vertex()

{

std::cout << "Destroy\n";

}

Vertex& operator=(const Vertex& other)

{

std::cout << "Copy\n";

x = other.x;

y = other.y;

z = other.z;

return \*this;

}

Vertex& operator=(Vertex&& other) noexcept

{

std::cout << "Move\n";

x = other.x;

y = other.y;

z = other.z;

return \*this;

}

};

template<typename T>

void PrintVector(const Vector<T>& vector)

{

for (size\_t i = 0; i < vector.Size(); i++)

{

std::cout << vector[i] << std::endl;

}

std::cout << "---------------------------\n";

}

template<> // template specialization for the Vertex class

void PrintVector(const Vector<Vertex>& vector)

{

for (size\_t i = 0; i < vector.Size(); i++)

{

std::cout << vector[i].x << ", " << vector[i].y << ", " << vector[i].z << std::endl;

}

std::cout << "---------------------------\n";

}

int main()

{

/\*Vector<std::string> vector;

vector.PushBack("Cherno");

vector.PushBack("C++");

vector.PushBack("Vector");

PrintVector(vector); \*/

Vector<Vertex> vecVertex;

vecVertex.PushBack(Vertex(1.0f));

vecVertex.PushBack(Vertex(2, 3, 4));

vecVertex.PushBack(Vertex());

//vecVertex.PushBack(Vertex());

//vecVertex.PopBack();

PrintVector(vecVertex);

/\*

Vector<Vertex> vecVertex2;

vecVertex2.EmplaceBack(Vertex(1.0f));

vecVertex2.EmplaceBack(Vertex(2, 3, 4));

vecVertex2.EmplaceBack(Vertex());

PrintVector(vecVertex2);\*/

system("pause");

}

### **VECTOR Optimization**

// VectorOptimization.cpp : KNOW YOUR ENVIRONMENT

// // https://youtu.be/PocJ5jXv8No?si=Kce9Co3YTV\_1ydOi

// Copy operation occurs: the compiler creates a new block of memory with the old data and a new element

// then the old location in memory where the original vector was is deleted.

// THE Goal is to reduce the number of times we resize our vector.

// In this environment (know your environment) we are strictly dealing with objects, not ptrs.

// Know when copies happen and why they happen?

#include <vector>

#include <iostream>

#include <stdlib.h>

struct Vertex

{

float x, y, z;

Vertex(float x, float y, float z)

: x(x), y(y), z(z) {}

// Add a copy constructor to Vertex to find out how many copies are being created.

Vertex(const Vertex& vertex)

: x(vertex.x), y(vertex.y), z(vertex.z)

{

std::cout << "Copied!" << std::endl;

}

};

std::ostream& operator<<(std::ostream& stream, const Vertex& vertex) // adds index access to Vertex struct

{

stream << vertex.x << ", " << vertex.y << ", " << vertex.z;

return stream;

}

int main()

{

// vector and Vertex are NOT the same, Vertex struct created up top.

//std::vector<Vertex> vertices; // not storing Vertex pointers, just storing vertexe objects.

//vertices.reserve(3); // different from resizing

//vertices.push\_back({1, 2, 3}); // no constructor but member initializer list works.

//vertices.push\_back({4, 5, 6}); // 3 copies on second Vertex added

//vertices.push\_back(Vertex(7, 8, 9)); // 6 copies with third Vertex added (if no reserve call).

std::vector<Vertex> vertices;

vertices.reserve(3);

vertices.emplace\_back(1, 2, 3);

vertices.emplace\_back(4, 5, 6);

// Each time we constuct an object, we are creating the object on the current stack frame in Main()

// After that we need to put that object into the vector. Copy from main function into the vector.

// WHAT IF we could construct that Vertex directly into the actual vector, vertices?

// Each time the vector adds additional objects, it needs to be resized, which adds a 2nd copy.

// So, for the 2nd element, copy gets called moving from main to vector, then another when resizing.

// Resizing each element adds n copies to the previous amount (1...3...6...10....15).. n-1 resizes + main copy

// Ranged based or foreach loop, add the '&' ampersand to avoid copying vertices.

for (Vertex& v : vertices)

std::cout << v << std::endl;

// vertices.erase(1); ERROR! Can't index directly in vector to erase, need to use iterator.

vertices.erase(vertices.begin() + 1);

// Empty or clear the vertices

vertices.clear();

system("pause");

}

void PassVectorByReference(const std::vector<Vertex>& vertices)

{

// '&' after vector<datatype> allows the function to pass vector by reference

}

### **Use Cases**

* Maintain collection of duplicate {key:value} pairs with fast insertion/removal.

**Pros:**

* Maintain collection of duplicate {key:value} pairs with fast insertion/removal.

**Cons:**

* Complexity

# **Data structures comparisons**

Arrays v Vectors:

* Dks
* D

Std::list v std::map:



# **General Questions**

Hash tables vs Binary Search Trees?  What’s better for an address book in a contacts list?

* Hash tables are great at storing and retrieving data quick O(1)
* BST can insert and retrieve at O (log n), but it also maintains its data in sorted order
* For an address book a BST would be better because it’s already sorted.
* For a contact list you would want to sort it, which would cost additional memory and resources for a hash table, and the slight performance difference between a hash table over a BST for lookup/insert wouldn’t be noticeable on a small list.

General breakdown of how to analyze a programming problem?

How to choose which data structure?

Difference between alloca and malloc?

When would you pass a pointer as an array as a parameter versus creating a local dynamic array and returning that array?

How much the computer allocated versus how many elements?

\* Know the difference of insert(key) for different Set types?