Dynamic Memory

Memory Models and `new`/`delete`

Use `new` to create an object on the heap and get its address (`new` evaluates to the address of the newly-created object). Use `delete` on a heap object's address to deallocate it and free up memory.

ℹ Using `delete` on a `nullptr` does nothing.

```
struct X {⋯}; class Y {⋯};

new int; // heap-allocates a "junk" int

// Calling default ctor with new:
new Y; // calls default ctor
Y* ptr3 = new Y{}; // ditto
Y* ptr4 = new Y(); // Only works w/ new

// value initialization
int* ptr1 = new int(); // *ptr1 == 0
int* ptr2 = new int{}; // *ptr2 == 0

// direct initialization
int* ptr_d = new int(3);
X* ptr5 = new X{3}; // {} for struct
Y* ptr6 = new Y{3}; // {}, () for class
Y* ptr7 = new Y(3); // invokes Y ctor
```

Object Lifetimes/Creation Revisited:

- A dynamic object's lifetime begins when it's created with `new`, and its lifetime ends (and its dtor is called) only when you use `delete` on its address.
- Static objects (e.g. static variables in classes/functions, global variables) live until a program ends.
- Local objects are destroyed when their scope ends.
- Array elements are constructed left-to-right and are destroyed in the reverse order. Class members are initialized in order of declaration in the class body.

ℹ In this example, every line where `new` is used causes a memory leak since we didn't `delete` any of the heap objects at the end.

ℹ You can delete an object through a pointer-to-`const` (or a `const` pointer).

Dynamic arrays are created with `new` (which yields a pointer to its first element) and deleted by using `delete[]` *on the address of its first element*. (Note that you can't delete individual elements.)

```
int N = 0;
cin >> N; // get num. elements from user
int* arr = new int[N]; // create size N array
arr[0] = 42; // sets arr's first element to 42
delete[] arr; // deletes entire dynamic array
```

ℹ While dynamic arrays are still fixed-size, they're allowed to have their sizes initialized at runtime, unlike non-dynamic arrays (which must have compile-time constant sizes).

❗ Deleting a dynamic array through a pointer with different static/dynamic types causes U.B.

Memory Management Errors

Failing to `delete` a heap object causes a **memory leak**, causing worse performance or possibly a crash at runtime. The following mistakes result in undefined behavior:

1. **Use-after-free**: dereferencing a pointer after using `delete` on it (`delete` doesn't kill the pointer itself).
2. **Bad delete**: Using `delete` on a dynamic array or non-dynamic object, or using `delete[]` on anything other than the address of a dynamic array's first element.
3. **Double free**: Using `delete` on an address twice.

```
int* func(int x) {
  int *y = new int(x);
  y = new int[x]; // Orphaned memory
  return y;
}  // This function leaks memory

int main() {
  int *a = func(5);
  int *b = a;
  delete b; // Wrong delete
  delete[] a; // double delete
  cout << a[2]; // use-after-free
} // We still have a memory leak...
```

RAII (Resource Aquisition Is Initialization)

RAII (or *scope-bound resource management*) is a strategy to prevent memory leaks by wrapping a dynamic object in a class. To create an RAII class:

1. Make the class's ctor allocate a dynamic object
2. Keep track of the object via a `private` pointer.
3. Make the class's dtor deallocate the object

This ties a dynamic object's lifetime to the scope of a non-dynamic class object. Note that an RAII dtor can only clean up resources allocated by the container itself (not its elements).

```
template <typename T>
class UnsortedSet {
private:
  T *arr; // ptr to underlying array
  int cap; // array's size limit
  int N; // current size of array
public:
  UnsortedSet() // ctor allocates arr
    : arr(new T[10]), cap(50), N(0) { }
  ~UnsortedSet() { delete[] arr; }
}; // dtor deallocates arr

// dtor definition OUTSIDE of a class body:
UnsortedSet::~UnsortedSet() {delete[] arr;}
```

Deep Copies and the Big Three

Shallow and Deep Copies

A **deep copy** of a class-type object is a copy whose pointer/reference data members are not shared with the original object's. A **shallow copy**'s data members are "shared" with the original object's. E.g. if you create a shallow copy of an `UnsortedSet`, both objects will have pointers to the *same* dynamic array. The concept only applies to classes that have "owning" pointers/references.

Copy constructor: a constructor that takes a reference to an existing object (*not* a value) as an argument and uses it to initialize a new object of the same type. In general, they're called whenever an object is initialized (either via direct or copy initialization) from another object of the same type.

```
class Foo {⋯}; Foo f1;
Foo f2 = f1; // implicitly calls copy ctor
Foo f3(f1); // calls copy ctor
Foo f4 = { f1 }; // ditto
```

```
Foo* f5 = new Foo(f1); // ditto
try { throw f1; } // ditto (throw-by-value)
catch (Foo f6) { } // ditto (catch-by-value)
Foo arr[3] = {f1, f1, f1}; // 3 calls to copy ctor
```

ℹ A class can have multiple copy constructors, but only one destructor (after all, destructors can't take arguments or be const-qualified). Ctors can't be const-qualified either, only their arguments can be.

The compiler-generated copy constructor and assignment `=` operator create member-by-member (shallow) copies, so to avoid memory errors, you must implement custom versions of them for classes that manage dynamic resources via pointers/references.

Ex: Deep Copy Constructor

```
class IntSet { ⋯
  // 1. Initialize the copy's stack members
  IntSet(const IntSet& og)
    : cap(og.cap), N(og.size()) {
  // 2. Create a new, separate dynamic array
    arr = new int[og.size()];
    for (int i = 0; i < og.size(); ++i) {
      arr[i] = og.arr[i];
    } // 3. Copy elements to new array
  }
};
```

Ex: Assignment Operator Overload

```
IntSet& operator=(const IntSet& rhs) {
  if (this == &rhs) { return *this; }
  delete[] this->arr; // this-> optional
  this->cap = rhs.cap;
  this->N = rhs.size();
  this->arr = new int[rhs.size()];
  for (int i = 0; i < rhs.size(); ++i) {
    this->arr[i] = rhs.arr[i];
  }
  return *this;
} // Note: returns a reference to LHS
```

Destructors and Polymorphism (Virtual Destructors)

If you try to delete a derived-class object by calling `delete` on a base-class pointer to the object, it causes undefined behavior unless the base class has a `virtual` destructor defined. The compiler-generated destructor is not `virtual`, so you must define one for classes that make use of inheritance.

⚠ If you call `delete` on an object whose static type and dynamic type are different, the static type must be a base class of the dynamic type (or U.B. will occur even with a `virtual` destructor).

ℹ You can prevent derived class objects from being deleted through base-class pointers by declaring the base class destructor `protected` and non-virtual.

The Rule of Three

The Big Three are a class's destructor, copy constructor, and assignment operator. The **Rule of Three** states that, if a class needs a custom version of one of the Big Three, it usually needs custom versions of the other two (generally, this describes classes that manage dynamic memory).

ℹ Not all classes with heap pointers need the Big Three. Ex: list iterators have pointers to dynamic nodes, but they shouldn't have the Big Three because they're not supposed to delete list nodes when they go out of scope or create new lists when they're copied.

Linked Lists

Linked Lists and Friend Classes

A **linked list** is a container that stores elements non-contiguously in distinct "nodes" that are connected via pointers. A singly-linked list is a list whose nodes are only linked in one direction (ex: `std::forward_list`), and a doubly-linked list's nodes are linked both ways (ex: `std::list`).

A list's `Node`s are defined as a `private` nested struct. Iterators for a linked list are declared as a public nested class, since they are part of the list interface (but their internal `Node*` pointers are still private).

ℹ Iterators typically declare their "parent" container classes as a `friend` class. This is because `private` members are only accessible within the scope of their class that defines them, which means the `private` members of a nested class aren't normally accessible to the "outer" class.

Singly-Linked List Node Deletion

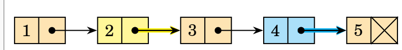
```
template <typename T>
void LinkedList<T>::erase(Node<T>* to_del) {
  if (!head) return; // if empty list
  auto p = head;
  if (head == to_del) {
    head = p->next; delete p; return;
  }
  while (p && p->next != to_del) p = p->next;
  if (!p) return; // if Node not found
  Node* next_node = p->next->next;
  delete p->next;
  p->next = next_node;
} // Assume no tail pointer for simplicity
```

Insert Value at List Index

```
template <typename T>
void LinkedList<T>::insert(int idx, T to_add) {
  Node* ptr = head; int pos = 0;
  while (ptr && pos < idx) {
    ptr = ptr->next; ++pos;
  } // The indexing is O(n) b/c no contiguity
  if (!ptr) return; // if index out-of-bounds
  if (!ptr->next) { // if at end of list
    ptr->next = new Node{to_add, nullptr};
  } // assume the Nodes have the datum first
  Node* old_next = ptr->next;
  ptr->next = new Node{to_add, old_next};
} // Insertion itself is O(1) b/c no shifting
```

Swap List Nodes

```
template <typename T>
void List<T>::swap(Node<T>* a, Node<T>* b) {
  auto p_a = head; auto p_b = head;
  while (p_a->next != a) p_a = p_a->next;
  while (p_b->next != b) p_b = p_b->next;
  auto a_next = a->next;
  auto b_next = b->next;
  auto tmp = a;
  p_a->next = b;
  p_a->next->next = a_next;
  p_b->next = tmp;
  p_b->next->next = b_next;
}
```

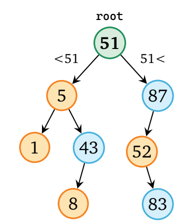


		INSERT			ERASE				FIND
Linking	Given Tail	Front	Back	Random	Front	Back	Current	Random	–
Single	✔ Yes	O(1)	O(1)	O(n)	O(1)	O(n)	O(n)	O(n)	O(n)
Single	✖ No	O(1)	O(n)	O(n)	O(1)	O(n)	O(n)	O(n)	O(n)
Double	✔ Yes	O(1)	O(1)	O(1)	O(1)	O(1)	O(1)	O(n)	O(n)
Double	✖ No	O(1)	O(n)	O(1)	O(1)	O(n)	O(1)	O(n)	O(n)

Binary Search Trees and Maps

Binary Search Trees

A *binary tree* is a tree data structure where each *node* points to at most 2 children. A **binary *search* tree** is a binary tree where, for any given node, each node in its left subtree has a lower value and each node in its right subtree has a greater value (an empty tree is also a BST).



A **leaf** of a BST is a childless node. A BST's **size** is the number of nodes it has, and its **height** is the maximum distance from the root to a leaf.

A BST is **balanced** if the heights of the left and right subtrees at any node differ by at most 1, so the height of a balanced BST is O($\log n$).

Binary Tree Traversal

There are multiple ways to traverse a binary tree:

1. **In-order traversal**: recursively process the *left* subtree, then process the *head* (current) node, then recursively process the *right* subtree.
2. **Pre-order traversal**: process the *head*, then recursively process the *left* subtree, then recursively process the *right* subtree.
3. **Post-order traversal**: recursively process the *left* subtree, then recursively process the *right* subtree, then process the *head*.

In-order Traversal (LNR)

1	5	8	43	51	52	83	87

Pre-order Traversal (NLR)

51	5	1	43	8	87	52	83

Post-order Traversal (LRN)

1	8	43	5	83	52	87	51

```
void pre_order(Node* p) {
  if (!p) return;
  cout << (p->val) << '\n';
  pre_order(p->left);
  pre_order(p->right);
} // preorder traversal
```

```
void in_order(Node* p) {
  if (!p) return;
  in_order(p->left);
  cout << (p->val) << '\n';
  in_order(p->right);
} // inorder traversal
```

```
void post_order(Node* p) {
  if (!p) return;
  post_order(p->left);
  post_order(p->right);
  cout << (p->val) << '\n';
} // postorder traversal
```

Use in-order traversal to visit a BST's nodes in non-decreasing order.
Use pre-order traversal to visit a BST's nodes in their original order of insertion. This is useful for copying a BST into an array (which you can then use to create a copy of the tree).
Use post-order traversal to delete or invert a tree (since you can operate on the leaves first).

BST Operations and Efficiency

BSTs perform best when they're balanced and worst when they're imbalanced (so the "worst-case" BST is a stick, which behaves like a singly-linked list).

	BST		Array		Sorted Array	
	Avg.	Worst	Avg.	Worst	Avg.	Worst
INSERT	O($\log n$)	O(n)	O(n)	O(n)	O(n)	O(n)
ERASE	O($\log n$)	O(n)	O(n)	O(n)	O(n)	O(n)
FIND	O($\log n$)	O(n)	O(n)	O(n)	O($\log n$)	O($\log n$)
MAX/MIN	O($\log n$)	O(n)	O(n)	O(n)	O(1)	O(1)

ℹ Constructing a BST by repeatedly inserting the maximum or minimum value of a set creates a stick.

ℹ `std::map` and `std::set` are implemented using *self-balancing* binary search trees, which is how they maintain $\Theta(\log n)$ insertion, removal, and searching even in the worst case.

```
Node* tree_min(Node* root) {
  if (!root) return nullptr;
  while (root->left) { root = root->left; }
  // go right to find max
  return root;
} // mix left/right for searching
```

```
int tree_height(TreeNode* root) {
  if (!root) return 0;
  if (!root->left && !root->right) return 1;
  return max(tree_height(root->left) + 1,
             tree_height(root->right) + 1);
} // the +1 "keeps count" of the recursions
```

The height of a balanced BST is proportional to $\log_2 n$, but *finding* its height is $\Theta(n)$. This is because the lengths of the paths from the root to each leaf can vary by 1, so in the worst case (when the longest path is the last one you search), you'd need to search every node to find the height.

BST Node Insertion

```
template <typename T>
void bst_insert(Node<T>*& root, T val) {
  if (!root) { root = new Node(val); }
  else if (val < root->datum) {
    bst_insert(root->left, val);
  }
  else { bst_insert(root->right, val); }
}
```

BST Search

```
template <typename T>
Node<T>* tree_find(Node<T>* root, T val) {
  while (!root && root->datum != val) {
    if (root->datum > val)  { root = root->left; }
    else { root = root->right; }
  }
  return root;
}
```

Recursion

Properties and Types of Recursion

Recursive functions are functions defined in terms of themselves (i.e., that call themselves). They're defined in terms of **base cases**, problems small enough to solve without recursing, and **recursive cases**, which the function breaks down and passes to itself.

There are three types of recursive functions:

1. **Linear recursive**: functions that make at most one call to themselves per invocation.
2. **Tail recursive**: a linear recursive function where the recursive call (if made) is the last instruction.
3. **Tree recursive**: a function that can call itself multiple times from the same stack frame/invocation.

ℹ The number of stack frames for a linear recursive function should monotonically decrease after the base case is reached (this is not necessarily true for a tree recursive function).

ℹ If you see multiple recursive calls separated by `if-else` branches, treat them as one recursive call when classifying the recursion type (because only one branch can execute per invocation).

Linear Recursion (Non-Tail)

```
Node* reverseList(Node* head) {
  if (!head || !head->next) return head;
  Node* new_head = reverseList(head->next);
  head->next->next = head;
  head->next = nullptr;
  return new_head;
} // Reverses a linked list in O(n) time
```

Tail Recursion

```
int func(int x, int y) {
  if (x <= 0 || y < 0 ) return 1;
  else if (x > y) return func(x - y, y);
  else return func(x, y - x);
}
```

Tree Recursion

```
bool SameTree(TreeNode* p,TreeNode* q) {
  if (!p || !q) {return (!p == !q);}
  if (p->val != q->val) {return false;}
  return SameTree(p->L, q->L)
         && SameTree(p->R, q->R);
} // check if two trees are identical
void func(int n) { // makes (2^n) - 1 calls
  for (int i = 0; i < n; ++i) func(i);
} // Yes, this is actually tree-recursive
```

⚠ A recursive call being on the last line or in the return statement does NOT itself guarantee that the function is tail recursive—the recursive call must be the last *instruction* executed by the function.

```
int factorial(int n) {
  return (n <= 1) ? 1 : n * factorial(n - 1);
} /* This is NOT tail-recursive because the last instruction the function executes is
    multiplying the recursive result by n, not the recursive call itself. */
```

Memory Usage of Recursive Functions

Non-tail linear recursive functions allocate an additional stack frame with each recursive call. However, the compiler can optimize a tail recursive function to reuse the same stack frame, which means that tail-recursive functions can be optimized to use a constant number of stack frames.

⚠ The total number of recursive calls a function makes to itself does not indicate its space complexity, since not all stack frames have to exist at the same time. Instead, look at the number of return statements until the base case. For example, a recursive algorithm to traverse a balanced BST of height h uses a maximum of $O(h)$ space—not $O(2^h)$ space—at any given time.

Structural Recursion

Structural recursion is when an abstract data type is defined in terms of itself. The `Node` structs of a linked list are one example: each `Node` contains a pointer to another `Node`. Recursive structures have base cases and recursive cases just like recursive functions do: for a linked list, the "base case" is an empty list, and the recursive cases are non-empty lists.

Recursion vs Iteration (and Converting Between Them)

Any recursive procedure can be converted to an iterative procedure, although this might require you to use your own stack to emulate the function call stack.

Iterators and Functors

Iterators are objects that emulate the interface of a pointer (so iterators are a superset of pointers). They allow a program to traverse and work with different containers in a uniform manner. Iterators are specific to their container, so they're usually defined as nested classes within a container class.

ℹ All iterators are dereferenceable with `*` and incrementable with prefix `++` (most iterators also have `!=`/`==` overloaded). Also, all iterators must be copy-constructible and copy-assignable.

ℹ All STL containers with iterators support `.begin()` and `.end()`, which return an iterator to the start of a container and a "past-the-end" iterator respectively (dereferencing a `.end()` iterator causes undefined behavior).

Traversal by iterator: a more general form of traversing a container data type by pointer.

```
vector<int> v(3, -1); // initializes v to {-1,-1,-1}
for (vector<int>::iterator it = v.begin(); it != v.end(); ++it)
  { cout << *it << endl; } // ::const_iterator if const vector
```

Range-Based For Loop (Works on Any Sequence Traversible by Iterator)

```
vector<int> v = { 1, 2, 3, 4 };
// for (<type> <variable> : <sequence>) {...}
for (int item : v) { // works with arrays too
  cout << item << endl;
} // could also declare item as const or a ref
```

```
// Compiler translation of range-based loop
for (auto it = v.begin(); it != v.end(); ++it) {
  int item = *it;
  cout << item << endl;
} // auto keyword makes compiler deduce type
```

Function Pointers

Functors (function objects): *first-class* entities that provide the same interface as a function. They can be created from classes that overload the function-call operator, i.e., `operator()`.

ℹ First-class entities can *store state* (store and access information), be created at runtime, and be a function's parameter or return type.

ℹ `operator()` can only be overloaded from within a class body. It and `operator[]` are also the only operators that can be overloaded as `static` member functions.

Binary Search with Recursion

```
template <typename Iter_T, typename T>
int binarySearch(Iter_T left, Iter_T right, const T& val) {
  int size = (right - left);
  if (size <= 0) return -1;
  Iter_T mid = left + (size / 2);
  if (*mid == val) return (mid - left);
  if (*mid > val) return binarySearch(left, mid, val);
  return binarySearch(mid + 1, right, val);
} // eliminates half the search space each loop
```

	`std::sort(it1,it2)`	`it2 - it1`	`it[n]`	`it += n`	`it1 < it2`	`--it`	`++it`	`it1 == it2`	`*it`
Random Access	✔ Yes	✔ Yes	✔ Yes	✔ Yes	✔ Yes	✔ Yes	✔ Yes	✔ Yes	✔ Yes
Bidirectional	✖ No	✖ No	✖ No	✖ No	✖ No	✔ Yes	✔ Yes	✔ Yes	✔ Yes
Forward	✖ No	✖ No	✖ No	✖ No	✖ No	✖ No	✔ Yes	✔ Yes	✔ Yes

Iterator	`array<T,N>::iterator`	`vector<T>::iterator`	`list<T>::iterator`	`map<Key,T>::iterator`	`set<Key>::iterator`
Iterator Class	Random Access	Random Access	Bidirectional	Bidirectional	Bidirectional
Container Resizing	N/A	All iterators invalidated	Dynamic	Empty container	Linked
Insertion	Sequential	Iterators at/above point of insertion are invalididated	Unaffected	Unaffected	Unaffected
Erasure	Sequential	Iterators at/above point of erasure are invalididated	Only iterator at removed element is invalididated	Only iterator at removed element is invalididated	Only iterator at removed element is invalididated

Friend Classes/Functions

If a class `A` declares class `B` as a `friend`, that will make `A`'s private/protected data members accessible to `B` (but `B`'s private members won't become accessible to `A` that). You can also declare a non-member function as a `friend` to give it access to a class's private/protected member variables.

A `friend` function (or operator) needs to be declared as a `friend` inside of the class it's befriending, and it has to be declared before the friend declaration.

```
friend class F;
friend F; // access specifiers don't affect friend declarations
{
```

ℹ Friendships are neither inheritable nor transitive (a friend of a friend is not a friend).

ℹ If you declare an unqualified function or class as a `friend` inside of a local class, name lookup doesn't go beyond the innermost scope outside of the class you're declaring the friendship in.

ℹ A friend class can access virtual (and only virtual) functions of its friend's derived classes.

Error Handling and Exceptions

C++ **exceptions** separate error detection from error handling and provide a separate control flow for automatic error handling. They do this by transferring control of the program to **handler** functions.

To catch an exception, enclose a portion of potentially buggy code in a `try`-`catch` **block**. If the code inside of a `try` block `throw`s an **exception** (an object that signifies an error) when it's run, execution of the program will pause and all code will be skipped until the error is handled by a `catch` block (an *object* is "thrown" and is "caught" by a `catch` block.

ℹ An unhandled exception causes a program crash (i.e., termination of the program) at runtime.

ℹ A `try` block can have multiple `catch` blocks (e.g. with different parameters) associated with it.

Don't throw exception objects by reference.

C++ Stuff and Impostor Syndrome

A **static member function** does *not* have access to the `this` pointer and also can't access non-`static` data members (reminder: `static` members are "shared" between all instances of a class).

Templates and function overloading are forms of **compile-time polymorphism**, while subtype polymorphism is a form of **runtime polymorphism**.

The compiler automatically defines: (insert chart)

Container ADTs

stack: a container that's designed to operate in a LIFO order.

queue: a container designed to operate in a first-in/first-out (FIFO) order.

Container	Interface Operations - Optimal Implementations are All O(1)					
Stack	`empty`	`size`	`top (next to pop)`		`push_back`	`pop_back`
Queue	`empty`	`size`	`front (next)`	`back (last)`	`push_back`	`pop_front`

ℹ An efficient way to implement a queue is to create a vector with free space at both ends (a **ring/circular buffer**). This requires keeping track of the data's "head" (inclusive) and "tail" (exclusive).

C++ Standard Library Containers

Class	Ordering	Sorting	Iterator	Default Contents	Resizable	Contiguous	Duplicate Items	Mutable Items
`std::array`	✔ YesSequential	Unsorted	Rand. Access	Junk Data	✖ No	✔ Yes	✔ Allowed	✔ Yes
`std::vector`	✔ YesSequential	Unsorted	Rand. Access	Empty	✔ Yes	✔ Yes	✔ Allowed	✔ Yes
`std::list`	✔ YesSequential	Unsorted	Bidirectional	Empty	✔ Yes	✖ No	✔ Allowed	✔ Yes
`std::set`	✖ NoAssociative	Asc. key	Bidirectional	Empty	✔ Yes	✖ No	✖ Not Allowed	✖ No
`std::map`	✖ NoAssociative	Asc. Key	Bidirectional	Empty	✔ Yes	✖ No	✖ Keys/✔ Vals	✖ Keys/✔ Vals

	`std::array`		`std::vector`		`std::list`		`std::set`		`std::map`	
Function	Header(s)	Time	Header(s)	Time	Header(s)	Time	Header(s)	Time	Header(s)	Time
`.insert()`	O(1)	O(1)	O(n)	O(1)	O(n)	O(n)	O(n)	O(n)		
`operator[]`	O(1)	O(1)	O(n)	O(1)	O(n)	O(n)	O(n)	O(n)		
`.push_back()`	O(1)	O(1)	O(1)	O(1)	O(1)	O(1)	O(n)	O(n)		
`.push_front()`	O(1)	O(1)	O(1)	O(1)	O(1)	O(1)	O(n)	O(n)		
`.clear()`	O(1)	O(n)	O(1)	O(1)	O(n)	O(1)	O(n)	O(n)		

	`std::array`		`std::vector`		`std::list`			
Function	Header(s)	Complexity	Header(s)	Complexity	Header(s)	Complexity	Header(s)	Complexity
`.erase()`	O(1)	O(n)	O(n)	O(1)	O(n)	O(n)	O(n)	O(n)
`.pop_back()`	O(1)	O(n)	O(1)	O(1)	O(n)	O(1)	O(n)	O(n)
`.pop_front()`	O(1)	O(n)	O(1)	O(1)	O(n)	O(1)	O(n)	O(n)
`.clear()`	O(1)	O(n)	O(1)	O(1)	O(n)	O(1)	O(n)	O(n)

	`std::array`		`std::vector`		`std::list`			
Function	Header(s)	Complexity	Header(s)	Complexity	Header(s)	Complexity	Header(s)	Complexity
`operator[]`	O(1)	O(1)	O(n)	O(1)	O(n)	O(n)	O(n)	O(n)
`.find()`	O(1)	O(n)	O(1)	O(1)	O(n)	O(1)	O(n)	O(n)
`.empty()`	O(1)	O(n)	O(1)	O(1)	O(n)	O(1)	O(n)	O(n)
`.clear()`	O(1)	O(n)	O(1)	O(1)	O(n)	O(1)	O(n)	O(n)
`.size()`	O(1)	O(n)	O(1)	O(1)	O(n)	O(1)	O(n)	O(n)

Default Element Initialization:

C++ Standard Library Containers

```
std::array<int, 4> arr = {-4,0,3,6};
std::array arr2{1, 2}; // Only way to omit
    size
std::list<int> doubly_linked = {1,2,3};
std::forward_list<int> singly_linked = {4,5,6
    };
std::set<int> nums = {3,2,2,1}; // {1,2,3}
std::map<string, int> EECS = { {"Bill", 183}
    };
cout << EECS["Bill"] << endl; // prints 183
// Two ways to insert into a map:
EECS["Emily"] = 203;
EECS.insert(pair<string, int>("James", 280));
```

Useful `std::map` functions

```
// Returns iterator to the pair with Key == k
// Returns .end() iterator if no such pair
    exists
iterator find(const Key_type& k) const;
// Inserts a <key, value> std::pair into a map
// Returns <iterator, false> if Key already in
    use
pair<iterator,bool> insert(const Pair_type&
    pair);
/* Finds or enters a value for a given key,
   then
returns a reference to the associated value */
Value_type& operator[](const Key_type& key);
```

Templates (Parametric Polymorphism)

Templates: special functions that take a data type as a parameter at compile time and instantiate an object or function compatible with that type. They reduce code duplication in container interfaces.

Class Template Syntax

```
template <typename T>
class UnsortedSet {
public:
  void insert(T my_val);
  bool contains(T my_val) const;
  int size() const;
private:
  T elts[ELTS_CAPACITY];
  int elts_size;
  ...
}; // Syntax: UnsortedSet<type> s;
```

Function Template Syntax

```
// Note: "class" also works in place of "typename"
template <typename T> // "T" is also an arbitrary name
T maxValue(const T &valA, const T &valB) {
  return (valB > valA) ? valB : valA;
} // This function returns the greater of valA and valB
// Syntax to call it: maxValue<int/double/etc>(...);
```

Templated Class Member Function Syntax

```
template <typename T> // Necessary if outside class body
void UnsortedSet<T>::insert(T my_val) {···}
```

Iterators, Traversal by Iterator and Range-Based Loops

	`std::sort(it1,it2)`	`it2 - it1`	`it[n]`	`it += n`	`<, <=`	`--`	`++`	`==, !=`	`*it`
Random Access Iterators	✔ Yes	✔ Yes	✔ Yes	✔ Yes	✔ Yes	✔ Yes	✔ Yes	✔ Yes	✔ Yes
Bidirectional Iterators	✖ No	✖ No	✖ No	✖ No	✖ No	✔ Yes	✔ Yes	✔ Yes	✔ Yes
Forward Iterators	✖ No	✖ No	✖ No	✖ No	✖ No	✖ No	✔ Yes	✔ Yes	✔ Yes