

- C-Style Struct vs C++ Class Syntax**
- 1 The compiler implicitly creates a default ctor iff there are no user-defined ctors (same for dtors).
 - 2 The order in which members are declared in a class body is always the order they're initialized in.
 - 3 Initialization values from a member init. list take precedence over initializations made at declaration.
 - 4 You can't initialize members within a constructor body—attempting to do so actually performs default-initialization followed by assignment.
 - 5 A delegating ctor must contain a call to the other ctor (and nothing else) in its member init. list.

```
1 class Animal {
2 private: string name;
3 public:
4     Animal(const string& name_in) // 1-argument ctor
5     : name(name_in) {}
6     Animal() : Animal("Blank") {} // Default ctor
7     // Default ctor delegates to other ctor
8
9 class Bird : public Animal {
10 private: bool can_fly;
11 public:
12     Bird(string name_in, bool fly_in)
13     : Animal(name_in), can_fly(fly_in) {}
14     // Derived class ctors must call a base ctor
15
16 class Duck : public Bird {
17 private: int age;
18 public:
19     Duck(string name_in, bool fly_in, int age_in)
20     : Bird(name_in, fly_in), age(age_in) {}
21     // Calling Bird ctor also calls Animal ctor
22
23     // This is how to define a ctor OUTSIDE of body
24     Bird(string name_in, bool fly_in)
25     : Animal(name_in), can_fly(fly_in) {}
26 }
```

```
ADT Function Definition
1 // C-Style Struct
2 void Triangle::scale(Triangle* t, double s) {
3     t->side *= s;
4     // "-->" is necessary here
5 }
6
7 // C++ Class (Inside Body)
8 class Triangle { // this-> is optional
9     void scale(double s) { this->side *= s; }
10    // this-> implicit iff no name conflicts
11 }
12
13 // C++ Class (Outside Body)
14 void Triangle::scale(double s) {...}
```

```
Object Creation/Manipulation
1 // C-Style Struct
2 Triangle t1;
3 Triangle_init(&t1, 3, 4, 5);
4
5 // C++ Class
6 Triangle t1; // Calls default ctor
7 Triangle t2(3,4,5); // calls 3-argument ctor
8 Triangle t3 = Triangle(3,4,5); // ditto
9 Triangle t4{3, 4, 5}; // ditto
10 Triangle t5 = {3, 4, 5}; // ditto
11 Triangle t6 = Triangle{3, 4, 5}; // ditto
12 // The last 3 work for classes and structs
```

```
const Function Definition
1 // C-Style Struct
2 double area(const Triangle* t) {...}
3 // const goes inside argument list
4
5 // C++ Class (Inside Body)
6 class Triangle {
7     double area() const {...}
8     // const comes after signature
9 }
10
11 // C++ Class (Outside Body)
12 double Triangle::area() const {...}
```

Inheritance and Polymorphism

Function Overloading (Ad Hoc Polymorphism) and Operator Overloading

Function Overloading: using one name for functions with *different signatures*. Functions can only be overloaded in the same scope (otherwise the "closer" scope takes priority). Note: `const/non-const` passing only alters the signature if a function has pointer/reference parameters (or implicit `this->` pointers).

Operator Overloading: operators like `+`, `-`, `<<`, etc. must be "overloaded" either as top-level or class member functions to work properly with custom class types.

- 1 An operator must be overloaded as a top level function if the first operand is an atomic type or a class type whose definition we can't access (e.g. `ostream`). Also, the `=`, `()`, `[]` and `->` operators can only be overloaded as member functions (along with overloads that require access to `private` members).

```
[] Overload Example (Member)
1 class IntSet {
2 public: ...
3     bool contains(int v) const {...}
4     bool operator[](int v) const;
5 };
6
7 bool IntSet::operator[](int v) const {
8     return contains(v);
9 }
```

```
<< and == Overload Examples (Top-Level)
1 struct Pixel {...}; // from project 2
2
3 ostream& operator<<(ostream& os, Pixel pixel) {
4     return os << pixel.r << pixel.g << pixel.b;
5 } // os needs to be passed by non-const ref here
6
7 bool operator==(const Pixel& p1, const Pixel& p2) {
8     return p1.r == p2.r && p1.g == p2.g && p1.b == p2.b;
9 } // Don't pass by non-const ref here
```

Inheritance and Derived Classes

All base class members (EXCEPT ctors and dtors) become implicit members of derived classes. So you can call any public base class function on derived class objects or access inherited base class public members via `./->`

- 1 Creating a derived class object *always* calls a base class ctor. If you don't call one explicitly, the base class default ctor is implicitly called. Also, a base class dtor is always called when a derived object dies.

Member name lookup begins in the *static type* of a receiver/object and moves up the inheritance hierarchy (to the base class) if no match is found. It stops when it finds a matching name or reaches the top of the hierarchy.

- 1 Access levels are only checked *after* name lookup ends.
- 2 Member name lookup searches by name. Virtual function resolution at runtime searches by signature.

Constructor/destructor ordering: constructors are called in "top-down" order on derived class objects (i.e., the base-class ctor is called first). Destructors follow *bottom-up* ordering (the derived class's dtor is called first, and the base class's dtor is called last).

- 1 Non-dynamic objects are destroyed in reverse order of their creation.

Subtype Polymorphism and Class Casting

Subtype polymorphism (a form of runtime polymorphism) allows a publicly-derived class object to be used in place of a base class object through a base-class reference or pointer to the derived object.

```
1 class Bird {}; // Base class
2 class Chicken : public Bird {};
3 class Duck : public Bird {};
4 Bird b; Chicken c; Duck d;
5 b = c; // Legal, but "slices" c's data
6 Bird* b_ptr = &c; // Good, no slicing
7 c = b; // ERROR: illegal assignment
8 Chicken* c_ptr = &b; // ERROR: downcast
9 Duck* d_ptr = &c; // ERROR
```

```
C++ allows implicit upcasts (i.e. base pointers/refs to publicly derived objects), but all downcasts must be explicit via static_cast or (less preferably) dynamic_cast.
1 // Be careful - validity not checked at runtime:
2 Chicken* c_ptr_a = static_cast<Chicken*>(bird_ptr);
3 // Bird needs at least 1 virtual function for this:
4 Chicken* c_ptr_b = dynamic_cast<Chicken*>(bird_ptr);
```

Virtual Functions and the override Keyword

Here, the *receiver* of the call to `talk()` on line 13 has a **static type** known at compile time (`Bird`) and a **dynamic type** known at runtime (`Duck`). Member lookup starts in the static class, so `Duck::talk()` won't hide `Bird::talk()`. Instead, `Bird::talk()` is declared as **virtual** to make it dynamically-bound.

Declare a function as **virtual** in the base class to use the "most-derived" version on a receiver whose static and dynamic types are different.

- 1 Placing **override** at the end of a function's signature tells the compiler to verify that it overrides a base-class **virtual** function with a matching signature (a compile error occurs if no override is found).

```
1 class Bird {
2 ... // Can't use virtual outside class body
3     virtual void talk() const { cout << "tweet"; }
4 };
5
6 // Note: ctors can't be virtual, but dtors can
7 class Duck : public Bird {
8     // virtual keyword is implicit here
9     void talk() const override { cout << "Quack"; }
10    // override == an optional "sanity check"
11 }
12
13 Duck duck;
14 Bird* duck_ptr = &duck;
15 duck_ptr->talk(); // prints "Quack"
16 // Scope resolution operator can suppress virtual
17 duck_ptr->Bird::talk(); // prints "tweet"
```

Pure Virtual Functions and Abstract Classes

Pure virtual function: a **virtual** function with no implementation (their purpose is to specify the interface of derived classes). To declare a function as pure virtual, add `= 0;` to the end of its signature.

Abstract class: a class with at least one pure virtual member function. Note that derived classes of an abstract class will also be abstract unless they override (define) every pure virtual function they inherit.

Interface (pure abstract class): a class that contains nothing but pure virtual member functions.

```
1 class Abstract { // Abstract Class
2     public: virtual void foo() = 0;
3 }; // Note the lack of braces after the = 0;
4
5 class Concrete : public Abstract {
6     public: void foo() { cout << "foo"; }
7 };
8
9 Concrete c;
10 Abstract* ptr = &c; Abstract& ref = c; // ok
11 Abstract abstract_object; // COMPILER ERROR
12 c.Abstract::foo(); // RUNTIME ERROR (U.B.)
```

- 1 Don't try to call a pure virtual function or create an abstract class object.

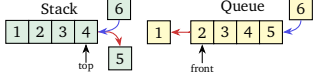
Containers and Data Structures

Container ADTs

static keyword: used to make one copy of a class data member "shared" between all instances of that class. A **static** data member has static storage duration but exists only within the scope of a class.

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

queue: a container data structure that supports first-in/first-out (FIFO) access.



Container	Interface Operations - Optimal Implementations are All O(1)					
Stack	.empty()	.size()	.top()	.push(val)	.pop()	
Queue	.empty()	.size()	.front()	.push(val)	.pop()	

All operations on stacks/queues run in $O(1)$ *amortized* time (if `push` causes a grow operation, it will take $O(n)$ time).

- 1 An efficient way to implement a queue is to create a vector with free space at both ends, i.e., a **circular buffer**.

Useful std::vector Functions and Constructors	Operation	Unsorted Set	Sorted Set*	Stack	Queue	Array	List
.size()	V[1], at(i)	O(n)	O(n)	O(1)	O(1)	O(n)	O(1)
.push_back(val)	search	O(n)	O(logn)	O(n)	O(n)	O(n)	O(n)
.pop_back()	access	O(1)	O(1)	O(n)	O(n)	O(1)	O(n)
.resize(n)							
.clear()							
vector<T> v2(v1.begin(), v1.end())							

* These refer to the array-based sets we saw in class, not STL sets.

C++ Standard Library Containers

std::array: Containers with *compile-time constant* sizes that store elements in contiguous memory.

```
int x = 5;
array<int, x> arr;
// invalid constant

const int y = 10;
array<int, y> arr;
// okay

void foo(const int z) {
    array<int, z> arr;
    // invalid constant
}

int a = 5; const int b = a;
array<int, b> arr;
// still not a valid constant
```

std::vector: resizable containers that store elements at the front and free space at the back.

std::list: a container whose elements are linked via pointers in non-contiguous memory. Lists trade random access for constant-time insertion/deletion at arbitrary positions.

std::map: an associative array that maps unique keys to values (keys act like indexes for a map).

std::set: an associative sorted container that only stores unique keys.

- 1 Using comparison operators like `<` with the containers listed above will lexicographically compare their elements (`std::map` s are compared using their keys).

```
C++ Standard Library Containers
1 std::array<int, 4> arr = {1,2,3,4};
2 std::array arr2{1, 2}; // Only way to omit size
3 std::list<int> doubly_linked = {1,2,3};
4 std::vector<int> v{3, -1}; // {-1,-1,-1}
5 std::set<int> nums = {3,2,2,1}; // {1,2,3}
6
7 std::map<string, int> EECS = { {"Bill", 183} };
8 cout << EECS["Bill"] << endl; // prints 183
9 // Two ways to insert into a map:
10 EECS["Emily"] = 203;
11 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);
```

Class	Ordering	Sorting	Resizable	Contiguous	Duplicate Items	Modifying Items	Index()	Insert/Erase	Search
std::array	Sequential	Sorted	✗ No	✓ Yes	✓ Allowed	✓ Allowed	O(1)	-	O(n)
std::vector	Sequential	Unsorted	✓ Yes	✓ Yes	✓ Allowed	✓ Allowed	O(1)	O(n) <end	O(n)
std::list	Sequential	Unsorted	✓ Yes	✗ No	✓ Allowed	✓ Allowed	-	O(1)	O(n)
std::set	Associative	Asc. key	✓ Yes	✗ No	✗ Not Allowed	✗ Not Allowed	-	O(logn)	O(logn)
std::map	Associative	Asc. Key	✓ Yes	✗ No	✗ Keys/✗Vals	✗ Keys/✗Vals	O(logn)	O(logn)	O(logn)

Iterator	array<T,N>::iterator	vector<T>::iterator	list<T>::iterator	map<Key,T>::iterator	set<Key>::iterator
Operations	<code>[] ++ -- == < *[] +n</code>	<code>[] ++ -- == < *[] +n</code>	<code>[] ++ -- == < *[] +n</code>	<code>[] ++ -- == < *[] +n</code>	<code>[] ++ -- == < *[] +n</code>

Most STL containers allow you to use **range-based loops** to iterate over their elements. For a `std::map`, you'll need to access the `.first` and `.second` elements of the loop variable to get keys/values.

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

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

Templates (Parametric Polymorphism)

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

```
Class Template Syntax
1 template <typename T>
2 class UnsortedSet {
3 public:
4     void insert(T my_val);
5     bool contains(T my_val) const;
6     int size() const;
7 private:
8     T elts[ELTS_CAPACITY];
9     int elts_size;
10 ...
11 // Syntax: UnsortedSet<type> s;
```

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

```
Templated Class Member Function Syntax
1 template <typename T> // Necessary if outside class body
2 void UnsortedSet<T>::insert(T my_val) {...}
```

- 1 You can't pass `auto` or a `const`-qualified type to an STL container's template (the C++ standard doesn't allow containers of `const` elements).

Time Complexity

Determining Asymptotic/Big-O Complexity

General rules for finding the time complexity of an algorithm:

- 1 Multiply the complexities of nested procedures, e.g. two nested $O(n)$ loops are $O(n^2)$.
- 2 For sequential procedures, the most complex operation determines the overall runtime of the algorithm (e.g. an $O(\log n)$ step followed by an $O(n)$ step makes for an $O(n)$ algorithm).

$O(1)$	$O(\log n)$	$O(\sqrt{n})$	$O(n)$	$O(n \log n)$, $O(\log(n!))$	$O(n^2)$	$O(n^3)$	$O(2^n)$	$O(3^n)$	$O(n!)$	$O(n^n)$	$O(2^{2^n})$
--------	-------------	---------------	--------	-------------------------------	----------	----------	----------	----------	---------	----------	--------------

Inserting/Erasing: inserting into or erasing from the middle of a contiguous container is $O(n)$, since you'll need to shift the other elements. If contiguity is not required, then the act of insertion/deletion is $O(1)$.

- 1 Deleting an arbitrary node from a list is $O(1)$, but deleting a specific node is $O(n)$, because finding the node is $O(n)$.

Indexing: If you store objects of the same type contiguously, random access is $O(1)$.

Searching: searching an *unsorted* sequence for a value takes $O(n)$ time. A *sorted* sequence can optimally be searched in $O(\log n)$ time with **binary search**.

```
1 int SortedIntSet::Search(int v, int L, int R) const {
2     while (L < R) {
3         int middle = L + (R - L) / 2;
4         if (v == elts[middle]) { return middle; }
5         else if (v < elts[middle]) { R = middle; }
6         else { L = (middle + 1); }
7     } // eliminates half the search space each loop
8     return -1; // if we didn't find v
9 } // This is O(log n), but it requires sorted input
```