

- 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)
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: Bird(string name_in, bool fly_in)
12       : Animal(name_in), can_fly(fly_in) {}
13 }; // Derived class ctors must call a base ctor
14
15 class Duck : public Bird {
16 private: int age;
17 public:
18   Duck(string name_in, bool fly_in, int age_in)
19     : Bird(name_in, fly_in), age(age_in) {}
20 }; // Calling Bird ctor also calls Animal ctor
21
22 // This is how to define a ctor OUTSIDE of body
23 Bird::Bird(string name_in, bool fly_in)
24   : Animal(name_in), can_fly(fly_in) {}
```

```
ADT Function Definition
1 // C-Style Struct
2 void Triangle::scale(Triangle *t, double s) {
3   t->a *= s; // ">" is necessary here
4 }
5
6 // C++ Class (Inside Body)
7 class Triangle { // "this->" optional
8   void scale(double s) { this->a *= s; }
9   }; // this-> implicit iff no name conflicts
10
11 // C++ Class (Outside Body)
12 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 // Syntax for classes AND structs:
10 Triangle t4(3, 4, 5);
11 Triangle t5 = {3, 4, 5};
12 Triangle t6 = Triangle{3, 4, 5};
13
14 const Function Definition
15
16 // C-Style Struct
17 double area(const Triangle *t) { ... }
18 // const goes inside argument list
19
20 // C++ Class (Inside Body)
21 class Triangle {
22   double area() const { ... }
23   }; // const comes after signature
24
25 // C++ Class (Outside Body)
26 double Triangle::area() const { ... }
```

Inheritance and Polymorphism

Function Overloading (Ad Hoc Polymorphism) and Operator Overloading

Function Overloading: using one name to refer to 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 of functions with pointer/reference parameters (or implicit this-> pointers).

Operator Overloading: operators like +, -, <, etc. must be "overloaded" either as a top-level or class member function to work properly with custom classes (at least 1 operand must be of class-type).

- 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 =, O, [] and -> operators can only be overloaded as member functions (along with overloads that need to access private members).

```
[ ] Overload Example (Member)
1 class IntSet {
2 ... // contains() is also a member function
3 public:
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 class Line { ... }; // start/end are public members
2
3 ostream& operator<<(ostream& os, Line line) {
4   return os << line.start << line.end;
5   // os needs to be passed by non-const ref here
6 }
7
8 bool operator==(const Line la, const Line lb) {
9   return (a.start == b.start && a.end == b.end);
10 } // 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 non-private base class function on a derived class object or access non-private inherited data 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 will be 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 at the first member with a matching *name* or the top of the hierarchy.

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

```
1 class Base {
2 public:
3   void printC() {cout << "B" << endl;}
4   ~Base() {} // custom dtor syntax
5   }; // Base::~Base() outside class body
6
7 class Derived : public Base {
8 public:
9   void printC() {cout << "D" << endl;}
10  void printB() { Base::printC(); }
11 };
12
13 Derived d;
14 d.printC(); // prints "D"
15 d.printB(); // prints "B"
16 d.Base::printC(); // prints "B"
17 Base* b = &d;
18 b->printC(); // prints "B"
```

Modifier	Out-of-scope access	Derived class access
public	✓ Yes	✓ Yes
private	✗ No	✗ No
protected	✗ No	✓ Yes

```
1 class A {
2 public: // Base class
3   AC() {cout << "A_ctor ";}
4   ~AC() {cout << "A_dtor ";}
5 };
6
7 class B : public A {
8 public: // Derived class
9   BC() {cout << "B_ctor ";}
10  ~BC() {cout << "B_dtor ";}
11 };
12
```

```
1 int main() {
2   A obj_a; // Prints "A_ctor "
3   B obj_b; // Prints "A_ctor B_ctor "
4   A* b_ptr = &obj_b; // Doesn't print anything
5   }; // When main returns: "B_dtor A_dtor A_dtor "
```

Subtype Polymorphism and Class Casting

Subtype polymorphism allows a publicly-derived class object to be used in place of a base class object; to do this, a base class reference or pointer to a derived class object must be created.

```
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);
5
```

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 when a receiver's static and dynamic type are different and you want to use the dynamic version of the function.

override keyword: tells the compiler to verify that a function overrides a base-class virtual function with a matching signature (if no override is found, override causes a compile error).

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

Pure Virtual Functions and Abstract Classes

Pure virtual function: a virtual base-class function that has no meaning or implementation (their purpose is to specify the interface of derived classes). To declare one, add = 0; to the end of a function's 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 (i.e., implement) every pure virtual function they inherit.

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

```
1 class Abst { // Abstract class
2 public: virtual void foo() = 0;
3 }; // Note the lack of empty braces
4
5 class Concrete : public Abst {
6 public: void foo() { cout << "foo"; }
7 }; // public/private doesn't matter here
8
9 Concrete c;
10 Abst* c_ptr = &c; Abst& c_ref = c; // ok
11 Abst abst_obj; // COMPILER ERROR
12 c.Abst::foo(); // RUNTIME ERROR (or U.B)
```

- 1 Don't call pure virtual functions or try to instantiate abstract classes.

Containers, Templates and Array-Based 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 container that's designed to operate in a LIFO order.

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

- 1 An efficient way to implement a queue is to create a vector with free space at both ends (a **ring/circular buffer**). To do so, keep track of the data using head (inclusive) and tail (exclusive) variables.

Useful std::vector Functions					Operation					Interface Operations - Optimal Implementations are All O(1)				
size()	v[i]	.push_back(val)	.pop_back()	.resize(n)	insert/remove	Unsorted Set	Sorted Set	Stack	Queue	Array	List			
.front()	.back()	.at(i)		.clear()	search	O(n)	O(logn)	O(1) ²	O(1) ²	O(n)	O(1)	push_back	pop_back	
					access	O(1)	O(1)	O(n)	O(n)	O(n)	O(n)	push_back	pop_front	

C++ Standard Library Containers

std::array: Fixed-size containers that store elements in contiguous memory.

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 (i.e., in non-contiguous memory).

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.

```
C++ Standard Library Containers
1 std::array<int, 4> arr = {-4, 0, 3, 6};
2 std::vector<int> vec = {4, 7, 2};
3 // Note: can't index into a std::list
4 std::list<int> values = {1, 2, 3};
5
6 std::map<string, int> EECs; // <key, val>
7 EECs["Bill"] = 183; EECs["Emily"] = 203;
8 EECs["James"] = 280; // This also works
9 cout << EECs["James"] << endl; // prints 280
10
11 std::set<int> unique = {3, 2, 2, 1}; // {1, 2, 3}
```

```
Useful std::map functions
// Returns an iterator to the value at the key k
// Returns .end() iterator if key isn't in the map
iterator find(const Key_type& key_val) const;
// Inserts a pair of <key, T value> into a map
// Returns .iterator, false if key exists already
pair<iterator, bool> insert(const Pair_T &val);
// Returns a reference to a key's associated value
// If it doesn't exist, enters it into the map
Value_type& operator[](const Key_type& key_val);
```

Container	Ordering	Default Sort	Sizing	Default-Construction	Allocation	Value Type	Duplicate Values
std::array	Sequential	Unsorted	Fixed-size	Undefined values	Contiguous	T	Yes
std::vector	Sequential	Unsorted	Dynamic	Empty container	Contiguous	T	Yes
std::list	Sequential	Unsorted	Dynamic	Empty container	Not Contiguous	T	Yes
std::map	Associative	Ascending (by key)	Dynamic	Empty container	Not Contiguous	pair<const Key, T>	No repeat keys
std::set	Associative	Ascending	Dynamic	Empty container	Not Contiguous	Key	No

Templates (Parametric Polymorphism)

Templates: flexible models for producing code that take a data type as a parameter to create an object or call a function that works with that type. They help reduce code duplication in container ADT interfaces.

- 1 A template can accept an invalid type argument during instantiation (leading to a runtime error).

```
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 template <typename T> // "T" = an arbitrary parameter name
2 // Note: "class" also works in place of "typename".
3 T maxVal(const T &valA, const T &valB) {
4   return (valB > valA) ? valB : valA; // " ? " == conditional
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) { ... }
```

Iterators, Traversal by Iterator and Range-Based Loops

Iterators: objects that have the same *interface* as pointers; they provide a general interface for traversing different types of container ADTs. To implement iterators for an ADT, define them as a nested class within the container's class and overload the * (dereference), ++, ==, != operators.

- 1 std::begin() returns an iterator to the start of an STL container. std::end() returns an iterator that's 1 past the end of an STL container (the iterator returned by std::end() should not be dereferenced).
- 1 Using an *invalidated* iterator (e.g. an iterator pointing at a deleted element) causes undefined behavior.

Container	std::array	std::vector	std::list	std::map	std::set
Iterator Supported	Random Access	Random Access	Bidirectional	Bidirectional	Bidirectional
Allowed Operations	[], [], ++, --, ==, !=, <, >, +int	[], [], ++, --, ==, !=, <, >, +int	++, --, ==, !=, *it, +int	++, --, ==, !=, *it	++, --, ==, !=, *it

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

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

```
Range-Based For Loop (Works on Any Sequence Traversable by Iterator)
1 vector<int> v = {1, 2, 3, 4};
2 for (int <type> <variable> : <sequence>) { ... }
3 // int item = v[i]; // 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 }
```

Time Complexity

We define **runtime complexity** in terms of *number of steps*, not literal runtime. Big-O notation represents an *upper-bound* of the magnitude of a function's growth rate with respect to input size (thus, all O(n) functions are also O(n²), O(n³), etc). Big-Θ and Big-Ω represent average and lower bounds, respectively.

O(1)	O(logn)	O(√n)	O(n)	O(n logn)	O(log(n!))	O(n ²)	O(n ³)	O(2 ⁿ)	O(3 ⁿ)	O(n!)	O(n ⁿ)	O(2 ^{2ⁿ})
------	---------	-------	------	-----------	------------	--------------------	--------------------	--------------------	--------------------	-------	--------------------	--------------------------------

- 1 Functions in the same complexity class should have growth rates that differ by a constant factor as n → ∞. So O(2ⁿ) and O(8ⁿ) are NOT in the same complexity class, but O(log₂ n) and O(log₃ n) are.

Determining Asymptotic/Big-O Complexity

Constant coefficients: if they're not part of an exponent, ignore them. Ex: O(3n) = O(0.5n) = O(n).

Addition (sequential procedures): the highest-complexity term dominates. Ex: O(n² + n + logn) = O(n²).

Multiplication: multiply the individual terms' complexities. Ex: O(n × logn) = O(n logn).

Non-nested loops: *sum* the complexities of each operation *inside* the loop body, and then *multiply* that by the number of times the loop executes. Ex: a loop that runs from 0 to n with a O(logn) body is O(n logn).

Nested loops: start at the innermost loop and work outwards (the individual complexities of the loops should multiply). Ex: two nested O(n) loops will do n work n times, so they're O(n²).

Partitioning/repeated division: procedures that divide the "remaining steps" by a constant number each time they execute (e.g. binary search, for-loops that double the loop counter) are usually O(logn).