

C++ Classes

In C++, the only real difference between classes and structs are that classes have private member access and private inheritance by default while structs default to public access/inheritance.

Constructors

- 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 is *always* the order they're initialized in.
- 3 Initialization values from a member init. list over-write initializations made during declarations.
- 4 Data members that aren't included in a ctor's member initializer list or initialized at declaration get default-initialized/constructed.
- 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(string name,in) // Non-default ctor
5   : name(name.in) { } // Member init. list
6   Animal() // Default ctor (no arguments)
7   : Animal("B!tAnk") { } // ctor delegation
8 }; // Note the semicolon here!

10 class Bird : public Animal {
11 private: bool has_wings;
12 public: Bird(string name, bool wings.in)
13   : Animal(name), has_wings(wings.in) { }
14 }; // Derived class ctors must call base ctor

16 class Duck : public Bird {
17 private: string color;
18 public: Duck(string name, bool wings, string rgb)
19   : Bird(name, wings), color(rgb) { }
20 }; // Calling Bird ctor also calls Animal ctor

// This is how to define a ctor OUTSIDE of body
2 Duck: Duck(string name, bool wings, string rgb)
3   : Bird(name, wings), color(rgb) { }
```

Nested Classes and Constructors

To initialize a nested class object, initialize it with a valid argument for the nested class's ctor.

Nested class objects in a const class object are also const.

```
1 class Book {
2 public:
3   Book(double price.in)
4   : price(price.in) { }
5 // Note: no default Book ctor
6 private:
7   double price;
8 };

1 class Person {
2 public:
3   Person(string& n, double p)
4   : name(n), favBook(p) { }
5 private:
6   string name;
7   Book favBook;
8 };
```

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 ~, (), [] 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 };

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   bool operator==(const Line &a, const Line &b) {
8     return (a.start == b.start && a.end == b.end);
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 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 (causing a compile error if it doesn't exist or isn't accessible). Also, a base class dtor is always called when a derived object dies.

Member name lookup via (.)-> starts in the "first" class scope; if no match is found, the base class scope (if one exists) is searched. Lookup stops at the first match; member access levels are checked after name lookup finishes.

1 Attempting to overload functions inherited from a base class will "hide" them, not overload them.

```
1 class Base {
2 private:
3   int x = 5;
4 public:
5   int* x_ptr = &x;
6   int get_x() const { return x; };
7 };
8 class Derived : public Base { };

1 Derived d; // cannot directly access x
2 cout << *(d.x_ptr) << endl; // prints 5
3 cout << d.get_x() << endl; // prints 5

Summary: how access modifiers affect direct access
+-----+-----+-----+
| Modifier | Accessible to derived classes | Accessible out of scope |
+-----+-----+-----+
| public   | ✓ Yes                           | ✓ Yes                     |
| private  | ✗ No                            | ✗ No                      |
| protected | ✓ Yes                          | ✗ No                      |
```

Destructors: special functions that are invoked when a class object's lifetime ends (e.g. when you delete a dynamic object or when a local object goes out of scope). They look like ctors with ~ before their name.

1 For derived class objects, constructors follow *top-down* behavior (i.e., the base class ctor is called first), while destructors are *bottom-up* (the derived class dtor is called first, and the base dtor is called last).

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

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);
```

1 An invalid dynamic_cast evaluates to nullptr, and an invalid static_cast will cause a runtime error.

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 the function overrides a base-class virtual function with a matching signature (if no override is found, override causes a compile error).

Pure Virtual Functions and Abstract Classes

Pure virtual function: a virtual base-class function that has no meaning or implementation; they simply make up part of the interface for 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 (define) every pure virtual function they inherit.

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

```
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"
```

```
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_object; // COMPILER ERROR
12 c.Abst::foo(); // RUNTIME ERROR (or U.B.)
```

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

Container ADTs and Templates (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.

Vectors: resizable array-based container ADTs that store elements at the front and free space at the back.

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.

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

```
SortedIntSet::remove() Implementation
1 void remove(int v) {
2   int i = indexOf(v); // indexOf() is a member
3   if (i == -1) { return; }
4   for (; i < elts_size - 1; ++i) {
5     elts[i] = elts[i + 1];
6   }
7   --elts_size; // elts_size == cardinality
8 }

SortedIntSet::insert() Implementation
1 void insert(int v) {
2   if (indexOf(v) != -1) { return; }
3   int i = elts_size;
4   for (; (i > 0) && (elts[i-1] > v); i--)
5     elts[i] = elts[i - 1];
6   elts[i] = v;
7   ++elts_size;
8 }
```

Templates (Parametric Polymorphism)

Templates: flexible models for producing code that take a data type as a parameter to create an object 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 valA > valA ? valA : valB; // Note: "?" = ternary
5 // This function returns the greater of valA and valB
6 // Syntax to call it: maxValues= 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 an iterator for a particular 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.

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 (<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 }
```

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(n!)	O(n ⁿ)	O(2 ^{n²})
------	---------	-------	------	-----------------------	--------------------	--------------------	--------------------	-------	--------------------	--------------------------------

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 + log n) = O(n²).

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

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(log n) body is O(n log n). **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(log n).

log(xy) = log(x) + log(y)	log($\frac{x}{y}$) = log(x) - log(y)	log(x ⁿ) = n log(x)	log _b (x) = $\frac{\log_2(x)}{\log_2(b)}$ = $\frac{1}{\log_2(b)}$
---------------------------	--	---------------------------------	--