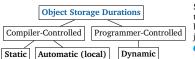
#### **Fundamentals and Machine Model**

#### Machine/Memory Model and the Function Call Stack

**Object:** a piece of data that's stored at a particular location in memory during runtime.

- Variable: a name in source code that is associated with an object at compile time.
- Not all objects are associated with variables; e.g. dynamically-stored objects and string literals are not. • The value stored by a variable's memory object may change, but the association between a variable and an object itself can only change when the variable goes out of scope.



Static objects "live" for essentially a program's runtime. Local objects' lifetimes are tied to scope (e.g. a block of code or pair of curly braces). Dynamic objects are manually created/destroyed.

① Objects declared in a loop body (between the {}) are created/destroyed each time the loop repeats.

Atomic (primitive) types: objects that can't be subdivided into smaller objects; includes int, double, bool, float, char, and all pointer types. Atomic objects are default-initialized to undefined values. The memory allocated to store a function's parameters and local variables during runtime is called a stack frame or activation record. The memory frame for the most-recently called function is added to the "top" of the function call stack and is destroyed when the function returns ("Last In First Out" ordering).

1 Assignments inside of return statements (e.g. return x = y; ) "take effect" before the return. Procedural Abstraction and Program Design

Procedural Abstraction involves using functions to break down a complex procedure into sub-tasks and separate the interface of a procedure (what it does) from implementation (how it works).

Interface examples: declarations in ...h files, valid/invalid inputs, RME statements, signature (function name and parameter types), return type, and ADT representation invariants.

Implementation examples: definitions in .cpp files and code/comments inside function bodies.

A pointer is a type of object that stores another object's memory address as its value.

• An int\* pointer can only point to an int: an int\*\* pointer can only point to an int\*; and so on. (Trying to, for example, make an int\* pointer point to a double will cause a compile error.)

Dereferencing: getting the object at an address. Note that the \* operator is used both to declare pointers and to dereference them (and the & operator is used both to get an object's address and to declare references).

```
1 int x = 3; int y = 4;
2 int* ptrl = &x; int* ptr2 = &y;
3 int** ptrl = &x; int* ptr2 = &y;
4 ptr2 = ptrl; // copies x's address from ptr1 to ptr2
5 ptr1 = &y; // ptr1 now points at y
6 **ptr1.ptr = 6; // now y == 6
7 *ptr2 = 2; // ptr2 still points to x, so now x == 2
```

- 1 Printing a non-char pointer prints the address that it stores.
- 1 A reference to a reference is really another reference for the "original" object.

```
int x = 5; int& a = x;
int* y = &x; // creates pointer to x
int* z = y; // creates another pointer to x
int* &b = z; // creates reference b to pointer z
cout << *b << endl; // Prints 5
cout << y << endl; // prints 0x2710
cout << &(*z) << endl; // equiv. to cout << &x
cout << *(&z) << endl; // equiv. to cout << &x</pre>
                                                                                                                                                                            0x271c 0x2710 z,b
                                                                                                                                                                           0x2714 0x2710 y
                                                                                                                                                                           0x2710 5 x,a
```

Null pointer: a pointer that holds address 0x0 (which no object can be located at) and implicitly converts to false. Any pointer can be nulled by setting it equal to nullptr (or 0, or NULL).

#### Differences Between Pointers and References:

- References are aliases for existing objects, while pointers are distinct objects with distinct values.
- · Pointers must be dereferenced to access the objects they point at, while references are used "as-is".
- You can change what a (non-const ) pointer points to, but you can't change what a reference refers to. Common Pointer/Reference Errors
- A Dereferencing an uninitialized pointer results in undefined behavior, as (like all atomic objects) pointers that aren't explicitly initialized are default-initialized to an undefined value (not nullptr).
- Dereferencing a null pointer also leads to undefined behavior (almost always a program crash).
- An uninitialized reference or a reference-to-non-const that's bound to a "literal" value won't compile. A If a function returns a pointer or reference to one of its local variables (which die when the function
- returns), dereferencing that pointer or using that reference produces undefined behavior.

```
1 int* danglingPtr(int x) { return &x; } // BUGGY 1 int& danglingRef(int x) { return x; } // BUGGY
```

A Be careful with mixing incrementing and dereferencing (and parentheses).

```
int x = 5;
int* ptr = &x;
// Output: 5 and 6
cout << (*ptr)++ << endl;
// ptr == &x

int x = 5;
int* ptr = &x;
// Output: 5 and 5
cout << x << endl;
cout << x << endl;
// ptr == junk (++&x)</pre>
                                                                                                                                                                                       int x = 5;
int* ptr = &x;
// Output: 6 and 6
cout << ++*ptr << endl;
cout << x << endl;
// ptr == &x</pre>
                                                                                                                                                                                                                                                                                    int x = 5;
int* ptr = &x;
// Output: 6 and 6
cout << ++(*ptr) << endl;</pre>
                                                                                                                                                                                                                                                                                     cout << x << endl;
// ptr == &x</pre>
```

#### Arrays and Pointer Arithmetic

Arrays: fixed-size containers that store objects of the same type (and same size) in contiguous memory.

```
int A[3] = {1,2}; // {1,2,0}
int B[3] = {}; // {0,0,0}
int C[] = {1,2}; // size == 2
                                                                 int D[][2] = {1,2,3}; // {{1,2},{3,0}} int E[][3] = {1,2,3}; // {{1,2,3}} int H[2][] = {1,2,3,4}; // Same int I[] = {}; // Same
```

Array decay: using an array in a context where a value is required causes the compiler to convert the array into a pointer to its first element. Array decay is why it's necessary to pass an array's size separately from the array to a function (or to indicate the end of an array with a sentinel character like C-strings do).

A Dereferencing a pointer that goes past the bounds of an array results in undefined behavior. But merely using a pointer that goes just past the end of an array without dereferencing it is well-defined.

```
void reverseArray(int arr[], int size) {
  for (int i = 0; i < (size / 2); ++i) {
    int temp = arr[i];
    arr[i] = arr[(size - i - 1)];
    arr[(size - i - 1)] = temp;
  } // Note: arr[i] == *(arr + i) == i[arr]
} // Therefore, &arr[i] == (arr + i)</pre>
```

Passing an array by value passes a pointer to its first element by value, so functions with array parameters like this one actually have pointer parameters.

1 The number of elements in an array arr is equal to (sizeof(arr) / sizeof(\*arr)).

**Pointer Operations** 

```
cout << &arr[0] and</p>
   cout << &arr would
   also print 0x1000.
```

You can create refer-

ences to array elements.

1 int arr[3] = { 5, 10, 15 };
2 int\* ptr = &arr[2];
3 int\* ptr2 = (arr + 1);
4 cout << arr << endl; // prints 0x1000
5 cout << &ptr2 << endl; // prints 0x1014
6 cout << ptr[-1] << endl; // prints 10</pre>

0x1000 arr[0] 5 0x1004 10 arr[1] 0x1008 15 arr[2] 0x100c 0x1008 ptr 0x1014 0x1004 ptr2

Pointer arithmetic: adding an integer n to a pointer yields a pointer that is n objects forward in memory.

Pointer subtraction: Subtracting two pointers of the same type yields an integer (possibly a negative one) equal to the number of objects between them.

Pointer comparison: comparing pointers of the same type compares the addresses they store.

Using & on an array (without an index) creates a pointer to the entire array, not a pointer to the first element or a pointer to a pointer.

```
1 // Mainly for pointers into the same array
2 int arr[4] = { 5, 10, 15, 20 };
3 int* ptrl = &arr[0], *ptr2 = &arr[3];
4 cout << *arr << endl; // prints 5
5 cout << (ptr2 - ptrl) << endl; // prints 3
6 cout << (ptr1 - ptr2) << endl; // prints -3
7 assert[cptr1 < ptr2]; // true
8 ptrl += 2; // ptrl now points at arr[2]
int arr[4] = { 1, 2, 3, 4 };
int (*arr.ptr)[4] = Sarr; // pointer to entire array
cout << (*arr.ptr)[2] << endt; // prints 3
// ++arr.ptr would increment by the size of 4 ints</pre>
```

Array traversal: arrays can be traversed by index or pointer (differentiated by what gets incremented).

```
Traversal By Pointer: Pattern 1
                                                                                                                                                                                                                                                                                                                                                                         versal by Pointer: Pattern 2 (C-String Sanitiz
                                                                                                                                                                                                                                                                                                                                                                 void sanitize(char username[], char to_remove) {
  char* slow = username, *fast = username;
  while (*slow && *fast) { // while not '\0'
int computeRange(const int arr[], int N) {
             nt computeRange(const int arr[], int N) {
  const int* min = arr; // Need const here
  const int* max = arr;
  const int* end = (arr + N);
  // end is actually "1-past-the-end"
  for (const int* p = arr; p < end; ++p) {
    if (*p * min) { min = p; }
    if (*p * min) { max = p; }
    if (*m * min) { min + min +
                                                                                                                                                                                                                                                                                                                                                                                              if (*fast != to_remove) {
   *slow = *fast;
                                                                                                                                                                                                                                                                                                                                                                                                         ++slow; // ++slow only if we copy
                                                                                                                                                                                                                                                                                                                                                                                                ++fast; // ++fast every loop
                                                                                                                                                                                                                                                                                                                                            9 | }
10 | *slow = '\0'; // null-terminate when done
11 } // NOTE: '\0' is the only char considered "false"
```

#### The const Keyword

The const type qualifier stops objects from being modified after initialization. Note: const scalars must be explicitly-initialized to compile, and const class-type objects must have their data members initialized.

const pointers: pointers that can modify what they point at but cannot be re-pointed to different objects. Pointer-to-const: read-only pointers; pointers that can be re-bound but can't modify what they point at.

( A const pointer must be initialized to compile, but a pointer-to-const doesn't need to be.

Reference-to-const: a read-only alias.

const array: an array of const elements. Note that the placement of const matters for arrays of pointers.

```
Special const Type Syntax
int x = 5;
int * const ptr_a = &x; // const pointer
const int * ptr_b = &x; // pointer-to-const
int const * ptr_c = &x; // pointer-to-const
const int * const ptr_d = &x; // both
int const * const ptr_e = &x; // both
const int &ref_a = x; // reference-to-const
int const &ref_b = 42; // reference-to-const
const int arr_a[2] = {1, 2}; // array of consts
int const arr_b[2] = {3, 4}; // array of consts
```

const int\* A[] = {...}; // pointer-to-const array int\* const B[] = {...}; // const pointer array

#### const Conversions and Passing

The compiler treats every pointer-to- const as if they point to a const object and every reference-toconst as if they're aliased to a const object. It won't allow conversions that could bypass existing const protections (so, e.g., you can assign a const pointer to a pointer-to-const, but the converse is not true).

```
1 int x = 2, y = 5;

2 const int *x_ptr = &x;

3 int *y_ptr = &y;

4 *y_ptr = *x_ptr; // Ok

5 y_ptr = x_ptr; // ERROR (even

6 though x isn't const!) */
                                                                                                                                                        const int x = 3;
   void foo(string& a) {...}
void foo(string& a) {...}
void bar(string b) {...}
void func(const string& c) {...}
const string s = "Hello World";
bar(s); func(s); // both ok
foo(s); foo("Hello"); // ERRORS
                                                                                                                                             1 const int x = 3;
2 int y = x; // 0k
3 const int* cptr = &x; // 0k
4 const int& cref = x; // 0k
5 int* ptr = cptr; // ERROR 1
6 int& ref = cref; // ERROR 2
```

- Pass by pointer/reference: if you need to modify the original object (as opposed to a local copy).
- $\bullet \ Pass \ by \ value: \ if \ an \ object \ is \ small \ (e.g., \ an \ \ int\ ) \ and \ you \ can't/don't \ need \ to \ modify \ the \ original.$
- Pass by pointer/reference-to- const: if you want to pass a large object without modifying it.

#### Strings, Streams and I/O

#### Creating/Using C-Strings and Strings

```
char color[] = "00274C"; // Create 7-element array (including \0) and copy a string literal to it const char* cstr = "abcd"; // Only works for string literals; use .c_str() on string variables cout << cstr << " " << *cstr << " " << &cstr[0] << endl; // prints "abcd a abcd" cout << (cstr + 1) << " " << *(cstr + 1) << " | ' ' < *(cstr + 1) << ' r ' ' < *(cstr + 1) << ' r ' ' < ' | ' ' < ' cstr + 1) </ri>
string xyz = string(cstr); // Explicitly copy cstring to a string (implicit copy would work too)
```

	Length	Copy Value	Index	Concatenate	Compare		
<strin< th=""><th>&gt; str.length();</th><th>str1 = str2;</th><th>str[i];</th><th>str1 += str2;</th><th colspan="3">str1 != str2;</th></strin<>	> str.length();	str1 = str2;	str[i];	str1 += str2;	str1 != str2;		
<cstrin< th=""><th><pre>g&gt; strlen(cstr);</pre></th><th>strcpy(cstr1, cstr2);</th><th>cstr[i];</th><th>strcat(cstr1, cstr2);</th><th>strcmp(cstr1, cstr2);</th></cstrin<>	<pre>g&gt; strlen(cstr);</pre>	strcpy(cstr1, cstr2);	cstr[i];	strcat(cstr1, cstr2);	strcmp(cstr1, cstr2);		

Pipeline

#### Streams and File I/O stdin Redirection ./main.exe < input.txt

File I/O Ex 1: Print Lines From File	Ex 2: Copy One File's Contents to Another
<pre>1 #include <fstream> // defines if/ofstreams 2 int main() { 3    ifstream inFS; // or inFS("file.txt"); 4    inFS.open("file.txt"); 5    if (linFS.is.open()) { return 1; } 6    string str; // defaults to empty string 7    while (getline(inFS, str)) { 8      cout &lt; str &lt; endl; 9    } // could close inFS via inFS.close();</fstream></pre>	<pre>1 #include <fstream> 2 void copyFile(string file_in, string file_out) { 3</fstream></pre>

1 The insertion << and extraction >> operators "stop" at any white space (spaces, line breaks, etc). istringstream : an object that "simulates" input with a string as its source. Note: an istringstream

10 } // inFS also closes when scope ends

stdout Redirection

an ifstream and cin can all be passed to a function with a std::istream& parameter. ostringstream: an object that captures output and

stores it in a string. Note: an ostringstream, an ofstream and cout can all be passed to a function with a std::ostream& parameter.

```
#include <sstream> // defines stringstreams
void printPlusOne(istream& is, ostream& os) {
  int num = 0;
  while (is >> num) { os << (++num) << " "; }
} // Note: can't pass or return streams by value</pre>
      istringstream inSS("1 3 5");
      ostringstream outSS;
9 printPlusOne(inSS, outSS);
10 cout << outSS.str() << endl; // Prints "2 4 6 "</pre>
```

Combined Redirection

#### **Command-Line Arguments**

argc: an int parameter of main() representing the number of a command's arguments.

argv: an array of the arguments passed to a program. (Technically, argv is an array of pointers to Cstrings—so argv is passed to main() as a pointer to an array of pointers to C-strings).

```
#include <string> // defines stoi()/stod()
int main(int argc, char* argv[]) { // char** argv also OK
int (string(argv[1]) = "add") {
int sum = 0;
int sum = 
                                                             for (int i = 2; i < argc; ++i) {
| sum += stoi(argv[i]);
| } | cout << "Sum: " << sum << ", argc: " << argc << endl; | } // pay attention to where the "actual" arguments start } // Also remember to use stoi()/string() when needed
```

```
hugokim@ubuntu:~$ ./main.exe add 7 2
Sum: 9, argc: 4
hugokim@ubuntu:~$ ./main.exe add 1 2 3
Sum: 6, argc: 5
hugokim@ubuntu:~$ .
```

## **ADTs, Structs and Classes**

# C-Style Structs and ADTs

A struct is a class-type object composed of member subobjects (heterogeneous data). They're passed by value by default, and they support assignment and initialization via the = operator. A struct or class object can also be declared as const, which prevents it and all of its data members from being modified.

- 1 You cannot call non-const member functions on a const instance of a class or struct. Also, you can't call non-const member functions from within a const member function.
- Arrow -> operator: shorthand for a dereference followed by member access. (\*ptr).x == ptr->x;
- 1 Without parentheses, the dot and arrow operators have greater precedence than dereferencing. Abstract Data Type: a data type that separates its behavior and implementation. ADTs encompass both data and behaviors/functions that act upon it. Not all struct s are ADTs, some are "plain old data".

#### 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 struct's default to public access/inheritance.

#### Constructors

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

```
Constructor Definition Example
    class Animal {
        private: string name:
  public:
    Animal(const string& name_in) // 1-argument ctor
    | : name(name_in) { }
    Animal(): 
class Duck : public Bird {
private: int age;
// This is how to define a ctor OUTSIDE of body Bird::Bird(string name_in, bool fly_in) | : Animal(name_in), can_fly(fly_in) { }
```

#### C-Style Struct vs C++ Class Syntax

```
ADT Function Definition
// C-Style Struct
 // C-style Struct
// Coid Triangle_scale(Triangle* t, double s) {
   t->a *= s; // "->" is necessary here
}
// C++ Class (Outside Body)
void Triangle::scale(double s) {...}
```

```
Object Creation/Manipulation
 // C-Style Struct
Triangle t1;
Triangle_init(&t1, 3, 4, 5);
  // C++ Class
// C++ Class
Triangle t1; // Calls default ctor
Triangle t2(3,4,5); // calls 3-argument ctor
Triangle t3 = Triangle(3,4,5); // ditto
Triangle t4(3,4,5); // ditto
Triangle t5 = {3,4,5}; // ditto
Triangle t6 = Triangle{3,4,5}; // ditto
// 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
1 // C++ Class (Inside Body) 2 class Triangle { 3   double area() const {} 4 }; // const comes after signature 5
6 // C++ Class (Outside Body) 7 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 a top-level or class member function to work properly with custom classes (at least 1 operand must be of class-type).

• 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
     bool operator[](int v) const;
  bool IntSet::operator[](int v) const {
    return contains(v);
```

# << and == Overload Examples (Top-Level) 1 class Line { /\* start/end are public \*/ ... }; ostream& operator<<(ostream& os, Line line) { | return os << line.start << line.end; } // os needs to be passed by non-const ref here bool operator==(const Line &a, const Line &b) { return (a.start == b.start && a.end == b.end); } // Don't pass by non-const ref here

void print() {cout << "B" << endl;}
-Base() { } // custom dtor syntax

~Base() { } // custom dtor syntax
}; // Base::~Base() outside class body

void print() {cout << "D" << endl;}
void printB() { Base::print(); }</pre>

Derived d;
d.print(); // prints "D"
d.printB(); // prints "B"
d.Base::print(); // prints "B"
Base\* ptr = &d; // Base == static type
ptr->print(); // prints "B"

Access Modifier Out-of-scope access Derived class access

× No

✓ Yes

class Derived : public Base {

class Base {
public:

#### 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 . / ->

▲ 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 at the first member with a matching name or the top of the hierarchy.

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

# Constructor/Destructor Ordering

Destructors: special functions that are invoked when a class object's lifetime ends. Constructors are called in top-down order for derived classes (the base ctor is called first) while destructors are bottom-up (the derive dtor is called first, and the base dtor last).

 Non-dynamic objects are destroyed in the opposite order that they were created in.

or	<pre>class A { public: // Base class</pre>	<pre>class B : public A { public: // Derived class B() {cout &lt;&lt; "B_ctor";} -B() {cout &lt;&lt; "B_dtor";} };</pre>
), ed ne	1 int main() { 2   A obj_a; // Prints "A_c 3   B obj_b; // Prints "A_c 4   A* b_ptr = &obj_b; // D 5 } // When main() returns:	tor B_ctor " oesn't print anything

#### 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.

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.

```
// Be careful - validity not checked at runtime:
Chicken* cPtr_a = static_cast<Chicken **(bird_ptr);
// Bird needs at least 1 virtual function for this:</pre>
Chicken* cPtr_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.

```
class Bird {
public: // Can't use virtual outside class body
   virtual void talk() const { cout << "tweet"; }</pre>
};
// Note: ctors can't be virtual, but dtors can
class Duck : public Bird {
public: // virtual keyword is implicit here
Duck duck:
```

```
void talk() const override { cout << "Quack"; }
}; // override == an optional "sanity check"</pre>
Duck ouck,
Bird* duck.ptr = &duck;
duck.ptr->talk(); // prints "Quack"
// Scope resolution operator can suppress virtual
duck.ptr->Bird::talk(); // prints "tweet"
```

OPPLICATION PROPERTY OF THE base-class virtual function with a matching signature (a compile error occurs if no override is found).

#### 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 (implement) every pure virtual function they inherit.

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

```
1 class Abstr { // Abstract Class
2 | public: virtual void foo() = 0;
3 }; // Note the lack of empty braces
  class Concrete : public Abstr {
    public: void foo() { cout << "foo"; }
};</pre>
  Concrete bar;
Abstr* ptr = &bar; Abstr& ref = bar; // ok
Abstr abstract_object; // COMPILE ERROR
bar.Abstr::foo(); // RUNTIME ERROR/U.B.
```

Don't call pure virtual functions or try to create abstract class objects.

#### **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 container that's designed to operate in a LIFO order.

queue: a container designed to operate

in a first-in/first-out (FIFO) order.

			orenness or	itations are i	re All O(1)		
Stack en	mpty	size	top (next	to pop)	push_back	pop_back	
Queue en	mpty	size	front (next)	back (last)	push_back	pop_front	

1 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).

Useful std::vector Functions and Constructors					ľſ	Operation	Unsorted Set*	Sorted Set*	Stack	Queue	Array	List
.size()	v[i]/.at(i)	.push_back(val)	.pop_back()	.resize(n)	1 [	insert/remove	O(n)	O(n)	O(1)	O(1)		O(1)
.front()	.back()	.erase(iterator)	.emptv()	,		search	O(n)	$O(\log n)$	O(n)	O(n)	O(n)	O(n)
			1 1 7 10			access	O(1)	O(1)	O(n)	O(n)	O(1)	O(n)
vector <t> v2(v1.begin(), v1.end()) vector<t> v2(v1)</t></t>				*	These refer	to the array-ba	sed sets we s	aw in c	lass, not	STL se	ts.	

#### C++ Standard Library Containers

std::array: Containers with compile-time constant sizes that store elements in contiguous memory. std::vector: resizable containers that store elements at the front and free space at the back.

▲ Element insertions/deletions will invalidate std::vector iterators at and after the modified index. Also, any operation that changes the maximum capacity of a vector invalidates all existing iterators.

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. 1 Removing an element from a std::list, std::set, or std::map only invalidates iterators to the re-

moved element. Also, inserting an element doesn't ever invalidate iterators for those containers. C++ Standard Library Containers Useful std::map functions

```
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}
                                                                                                                                                                                       // 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);
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));
                                                                                                                                                                                      /* Finds or enters a value for a given key, then
returns a reference to the associated value */
Value_type& operator[](const Key_type& key);
```

Container	Ordering	Sorting	Resizable	Contiguous	Duplicate Items Iterator		Access[]	Insertion/Removal		Search
std::array	Sequential	Unsorted	× No	✓ Yes	✓ Yes	Rand. Access	O(1)	$O(n) < end^*$	O(1) end*	O(n)
std::vector	Sequential	Unsorted	✓ Yes	✓ Yes	✓ Yes	Rand. Access	O(1)	O(n) < end	O(1) end	O(n)
std::list	Sequential	Unsorted	✓ Yes	× No	✓ Yes	Bidirectional	N/A	O(1) any	where	O(n)
std::set	Associative	Ascending	✓ Yes	× No	× No	Bidirectional	N/A	O(log	(n)	$O(\log n)$
std::map	Associative	Asc. Key	✓ Yes	× No	★ Keys/  ✓ Vals	Bidirectional	$O(\log n)$	$O(\log n)$		$O(\log n)$

# 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 help reduce code duplication in container interfaces.

```
Class Template Syntax
                                                                                                 // Note: "class" also works in place of "typename"
template <typename T> // "I" is also an arbitrary name
T maxValue(const T &vala, const T &vala) {
  return (valb > valA) ? valB : valA;
      template <typename T>
class UnsortedSet {
      ctass UnsortedSet {
public:
    void insert(T my_val);
    bool contains(T my_val) const;
    int size() const;
private:
                                                                                                 // This function returns the greater of valA and valB
// Syntax to call it: maxValue<int/double/etc>(...);
          T elts[ELTS_CAPACITY];
int elts_size;
11 }; // Syntax: UnsortedSet<type> s;
```

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), ++, ==, and != 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).

🛕 Using an invalidated iterator (e.g. an iterator pointing at a deleted element) causes undefined behavior.

ı	Iterator Type	Random Access Iterators	Bidirectional Iterators	Forward Iterators			
ı	Interface Operations	-> [] ++ == != < *it +n	++ == != *it	*it ++ == !=			
- 1							

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 < end(); // ::const_iterator if const vector</pre>
```

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

# **Time Complexity**

# Determining Asymptotic/Big-O Complexity

General rules for finding the time complexity of an algorithm:

Multiply the complexities of nested procedures, e.g. two nested O(n) loops are  $O(n^2)$ .

For "independent" 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 is an O(n) algorithm).

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

Searching: searching an unsorted sequence for a value takes O(n) time. A sorted sequence can be searched in  $O(\log n)$  time by performing binary search (which repeatedly halves the range left to search). Insertion/Removal: inserting into or deleting 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 insertion/deletion is O(1). Note that, if you need to preserve a sorting order, finding the "target" index adds an additional  $O(\log n)$  step. Indexing: If you store elements of the same size in contiguous memory, accessing an arbitrary index is O(1), as only a single pointer arithmetic operation is needed.