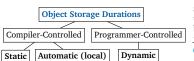
### **Fundamentals and Machine Model**

#### 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.

- 1 Not all objects are associated with variables; e.g. dynamically-stored objects and string literals are not.
- 1 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.

## Pointers, Arrays and Reference

Pointers are a type of object that store other objects' memory addresses as their values.

A pointer can only point to objects of the corresponding type, e.g. an int\* can only point to an int, an int\*\* can only point to an int\*, etc. Also, you can't assign integer values to pointers except 0.

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

- 1 Creating a reference to a reference creates another reference to the original object.
- 1 Creating a pointer to a reference creates a pointer to 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 to z

5 cout << *b << endl; // Prints 5

6 cout << y << endl; // prints 802710

cout << &(*z) << endl; // prints z (0x2710)

5 cout << *k(*z) << endl; // prints z (0x2710)

6 cout << *k(*z) << endl; // prints z (0x2710)
                                                                                                                                                                                                                                 5 x,a
                                                                                                                                                                                           0x2710
                                                                                                                                                                                            0x2714 0x2710 y
                                                                                                                                                                                            0x271c 0x2710 z.b
```

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

#### Differences Between Pointers and References:

- A reference is not a distinct object, it's a name for an existing object. A pointer is a separate object.
- 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.
- ⚠ If a function returns a pointer or reference to one of its local variables (which die when the function returns), dereferencing the pointer or using the 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;
cout << x << endl;
// ptr == &x</pre>
int x = 5;
int* ptr = &x;
// Output: 5 and 5
cout << x << endl;
cout << x << endl;
cout << x << endl;
// ptr == ++&x (junk)</pre>
                                                                                                                                                                                                    int x = 5;
int* ptr = &x;
// Output: 6 and 6
cout << ++*ptr << endl;
cout << x << endl;</pre>
                                                                                                                                                                                                                                                                                                       int x = 5;
int* ptr = &x;
// Output: 6 and 6
cout << ++(*ptr) << endl;
cout << x << endl;
// others or cout</pre>
                                                                                                                                                                                                      // ptr == &x
                                                                                                                                                                                                                                                                                                        // ptr == &x
```

## Arrays and Pointer Arithmetic

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

```
int D[][2] = {1,2,3}; // {{1,2},{3,0}} int E[][3] = {1,2,3}; // {{1,2,3}} int F[3]; // CAUTION: uninitialized! int \underline{I}[2][] = {1,2,3,4}; // Same
int A[3] = {1,2}; // {1,2,0}
int B[3] = {}; // {0,0,0}
int C[] = {1,2}; // size == 2
```

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, so when you pass an array to a function, you need to pass its size separately (or indicate the end of the array with a sentinel value like C-strings do).

A Dereferencing a pointer that goes past the bounds of an array causes undefined behavior. But 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** 

```
f cout << &arr[0] and
   cout << &arr would
   also print 0x1000.
1 You can create pointer-
```

s/refs to array elements.

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

0x1000 arr[0] arr { 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.

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

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

```
Traversal By Pointer: Pattern 1
                                                                                                   Traversal by Pointer: Pattern 2 (C-String Sanitization)
int 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 Cconst int* p = arr; p < end; ++p) {
  if (*p < *min) { min = p; }
  if (*p > *max) { max = p; }
  } // "wall" the pointer across arr
  return (*max - *min);
}
                                                                                                          ++fast; // ++fast every loop
                                                                                                         | *slow = '\0'; // null-terminate when done
|} // NOTE: '\0' is the only char considered "false"
```

The const type qualifier prevents objects from being modified after initialization. Note: const scalars (including const class/struct data members) must be explicitly-initialized to compile.

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.

```
const int &ref a = x; // reference-to-const
int const &ref b = 42; // reference-to-const
const int arra[2] = {1, 2}; // array of consts
int const arr_b[2] = {3, 4}; // array of consts
```

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

### const Conversions and Passing

The compiler only allows const conversions that result in equally strict or stricter protections. For exam ple, you can copy the address stored by a const pointer to a pointer-to-const, but not vice versa. Also, the compiler still enforces const correctness for pointers-to-const that point at non-const objects.

```
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 2
6 int& ref = cref; // ERROR 2
                                                                                                                                                                                                                                                                                                                                                                   int x = 2, y = 5;
const int *x_ptr = &x;
int *y_ptr = &y;
*y_ptr = *x_ptr; // Ok
y_ptr = x_ptr; /* ERROR (even
though x isn't const!) */
```

- Pass by pointer/reference: if you need to modify the original object (as opposed to a local copy).
- 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 Strings (and C-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) <- " " << * (str + 1) <- / prints "bd b 98" ('a' == 97) string str = string(cstr); // Explicitly copy cstring to a string (implicit copy would work too)
```

Length		Assignment		maex		Concatenatio	Comparison			
	s.length();	s.size();	s1 = s2;	str = cstr;	str[i];	s1 += s2;	str += 'a';	str += cstr;	s1 != s2;	s1 < s2;

# Streams and File I/O

stdin Redirection	stdout Redirection	Pipeline	Combined Redirection			
./main.exe < input.txt	./main.exe > output.txt	./output.exe   input.exe	./main.exe < input.in > output.out			

```
Ex 2: Copy One File's Contents to Another
       File I/O Ex 1: Print Lines From File
                                                                                                                                                   minclude <fstream>
void copyFile(string file_in, string file_out) {
   // file streams also accept C-strings as argume
   ifstream inFS(file_in);
   ofstream outFS(file_out);
}
 #include <fstream> // defines if/ofstreams
int main() {
int main() {
   ifstream inFS; // or inFS("file.txt");
   inFS.open("file.txt");
   if (linFS.is_open()) { return 1; }
   string str; // defaults to empty string
   while (getline(inFS, str)) {
        cout << str << end!;
        // could close inFS via inFS.close();
} // inFS also closes when scope ends</pre>
                                                                                                                                                         brstream out-strite_out/,
string input_str;
while (inFS >> input_str) {
    outFS << input_str << endl;
} // could use '\n' instead of endl</pre>
```

1 The stream extraction >> operator treats all leading whitespace the same as a single space.

istringstream: an object that "simulates" input with a string as its source. Note: an istringstream 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.

```
int num = 0;
| while (is >> num) { os << (++num) << " "; }
} // Note: can't pass or return streams by value
istringstream inSS("1 3 5");
ostringstream outSS;
printPlusOne(inSS, outSS);
cout << outSS.str() << endl; // Prints "2 4 6 "
```

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

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

#include <sstream> // defines stringstreams
void printPlusOne(istream& is, ostream& os) {

#### **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;

data and behaviors/functions that act upon it. Not all struct s are ADTs, some are "plain old data".

1 The dot and arrow operators have greater precedence than dereferencing. So, to access a data member through a struct/class pointer, either use -> or use \* inside parentheses. Abstract Data Type: a data type that separates its behavior and implementation. ADTs encompass both

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() : Animal("Blank") { } // Default ctor }; // Default ctor delegates to other ctor
class Bird : public Animal {
private: bool can_fly;
public: Bird(string name_in, bool fly_in)
    : Animal(name_in), can_fly(fly_in) {
}; // Derived class ctors must call a base ctor
 class Duck : public Bird {
private: int age;
public:
Duck(string name_in, bool fly_in, int age_in)
| : Bird(name_in, fly_in), age(age_in) { }
}; // Calling Bird ctor also calls Animal ctor
 // 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
void Triangle_scale(Triangle* t, double s) {
    | t->side *= s;
} // "->" is necessary here
// C++ Class (Inside Body)
class Triangle { // this-> is optional
void scale(double s) { this->side *= s; }
}; // this-> implicit iff no name conflicts
// 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 t43, 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
// C-Style Struct
double area(const Triangle* t) {...}
// const goes inside argument list
// C++ Class (Inside Body)
class Triangle {
  double area() const {...}
}; // const comes after signature
// C++ Class (Outside Body)
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.

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

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

class Base {
public:
 void print() {cout << "B" << endl;}
 ~Base() { } // custom dtor syntax
}; // Base::~Base() outside class body</pre>

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

Derived d;
d.print(); // prints "D"
d.print8(); // 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

class B : public A {
public: // Derived class
| B() {cout << "B ctor ";}
| ~B() {cout << "B dtor ";}</pre>

× No

class Derived : public Base {

#### 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 ./->

A 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.
- 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).

 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.

```
class Bird { }; // Base class
Chicken : nublic Bird { };
1 class Bird { }; // Base class
2 class Chicken : public Bird { };
3 class Duck : public Bird { };
5 class Duck : public Bird { };
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 = &c; // 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 int main() {
2 | A a.object; // Prints "A ctor "
3 | B b.object; // Prints "A ctor B ctor "
4 | A\* ptr = &b.object; // Doesn't print anything
5 } // When main() returns: "B dtor A dtor A

private

class A {
public: // Base class

1 int main() {

A() {cout << "A ctor ";}
~A() {cout << "A dtor ";}

```
// Be careful - validity not checked at runtime:
Chicken* cPtr_a = static_cast<Chicken*>(bird_ptr);
// Bird needs at least 1 virtual function for this:
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 {
... // Can't use virtual outside class body
    virtual void talk() const { cout << "tweet"; }
}</pre>
```

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).

#### 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.

nothing but pure virtual member functions.

# class Abstract { // Abstract Class public: virtual void foo() = 0; }; // Note the lack of braces after the = 0; class Concrete : public Abstract { public: void foo() { cout << "foo"; } };</pre> Concrete c; Abstract\* ptr = &c; Abstract& ref = c; // ok Abstract abstract\_object; // COMPILE ERROR c.Abstract::foo(); // RUNTIME ERROR (U.B.)

Interface (pure abstract class): a class that contains 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

queu

suppo

ue: a container data structure that	Stack	.empty()	.size()	.top()	.push(val)	.pop()			
orts first-in/first-out (FIFO) access.	Queue	.empty()	.size()	.front()	.push(val)	.pop()			
Stack 6 Queue 6		The interface operations of stacks/queues run in $O(1)$ <i>amortized</i> time ( push is $O(n)$ if it causes a grow operation).							
2 3 4 5	1 An effic	1 An efficient way to implement a queue is to create a vec-							

1 2 3 4 5 front tor with free space at both ends, i.e., a circular buffer Useful std::vector Functions and Construct .size() v[i]/.at(i) .push\_back(val) .pop\_back() .
.front() .back() .erase(iterator) .empty() vector<T> v2(v1.begin(), v1.end()) vector<T> v2(v1)

s	Operation	Unsorted Set*	Sorted Set*	Stack	Queue	Array	List		
.resize(n)	insert/remove	O(n)	O(n)	0(1)	O(1)	O(n)	O(1)		
	search	O(n)	$O(\log n)$	-	-	O(n)	O(n)		
.clear()	access	O(1)	O(1)	-	-	O(1)	O(n)		
v2(v1)	2(v1) * These refer to the array-based sets we saw in class, not STL sets.								

Container Interface Operations - Optimal Implementations are All O(1)

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

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
std::array<int, 4> arr = {1,2,3,4};
std::array arr2{1, 2}; // Only way to omit size
std::list<int> doubly_linked = {1,2,3};
std::vector<int> v(3, -1); // {-1,-1,-1}
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);
Class | Ordering | Sorting | Resizable | Contiguous | Duplicate Items | Modifying Items | Index[1 | Insert/Erase | Search
```

std::array	Sequential	Unsorted	× No	✓ Yes	✓ Allowed	✓ Allowed	O(1)	-		O(n)
std::vector	Sequential	Unsorted	✓ Yes	✓ Yes  ✓ Allowed		✓ Allowed	O(1)	O(n) < end	O(1) 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(lo	gn)	$O(\log n)$
std::map	Associative	Asc. Key	✓ Yes	× No		x Keys/✔ Vals	$O(\log n)$	O(lo	gn)	$O(\log n)$
Iterator	array <t,n>::iterator</t,n>		vector <t>::</t>	iterator	list <t>::iterator</t>	map <key,t>::iterator</key,t>		set <key>:</key>	iterator	

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.

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

# 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.

```
Function Template Syntax
            Class Template Syntax
                                                                                       // Note: "class" also works in place of "typename"
template <typename T> // "T" is also an arbitrary name
T maxValue(const T &valB) T &valB |
return (valB > valA) ? valB : valA;
} // This function returns the greater of valA and valB
// Syntax to call it: maxValue<int/double/etc>(...);
   template <typename T>
class UnsortedSet {
    public:
       ablic:
void insert(T my_val);
bool contains(T my_val) const;
int size() const;
       T elts[ELTS_CAPACITY];
int elts_size;
                                                                                          Templated Class Member Function Syntax
                                                                                   1 template <typename T> // Necessary if outside class body 2 void UnsortedSet<T>::insert(T my_val) \{\cdots\}
11 }; // Syntax: UnsortedSet<type> s;
```

▲ 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:

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

Inserting/Erasing: inserting into or erasing Searching: searching an unsorted sequence for a value 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).

takes O(n) time. A sorted sequence can optimally be searched in  $O(\log n)$  time with binary search.

```
int SortedIntSet::Search(int v, int L, int R) const {
  while (L < R) {
    int middle = L + (R - L) / 2;
    if (v == elts[middle]) {      return middle; }
    else if (v < elts[middle]) {      R = middle; }
    else if (v < elts[middle]) {      R = sarch space each loop
      return -1; // if we didn't find v
    } // This is O(log n), but it requires sorted input</pre>
```