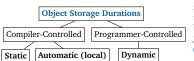
Fundamentals and Machine Model

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. 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 ob-

jects are manually created/destroyed. 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). 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).

```
int x = 3; int y = 4;
int* ptrl = &x; int* ptr2 = &y;
int** ptrl_ptr = &ptrl;
ptr2 = ptrl; // copies x's address from ptrl to ptr2
ptrl = &y; // ptrl now points at y
**ptrl_ptr = 6; // now y == 6
*ptr2 = 2; // ptr2 still points to x, so now x == 2
```

- 1 Printing a non-char pointer prints an address. (char pointers get printed as C-strings.)
- 1 A reference to a reference is really another reference for the "original" object.
- int x = 5;
 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 5x2710
 cout << \$ << endl; // prints 6x2710
 cout << \$ (*z) << endl; // equiv. to cout << &x
 cout << * (&z) << endl; // equiv. to cout << z</pre> 0x271c 0x2710 z,b 0x2714 0x2710 0x2710

Null pointer: a pointer that holds address 0x0 (which no object can be located at) and implicitly con-

verts 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.
- ▲ 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: 6 and 6
cout << ++(*ptr) << endl;</pre>
int x = 5;
int* ptr = &x;
                                                              int x = 5;
int* ptr = &x;
                                                                                                                             int x = 5;
int* ptr = &x;
                                                             // Output: 5 and 5
cout << *ptr++ << endl;
cout << x << endl;
// ptr == junk (++&x)
                                                                                                                            // Output: 6 and 6
cout << ++*ptr << endl;
cout << x << endl;
// ptr == &x
 // Output: 5 and 6 cout << (*ptr)++ << endl;
cout << x << endl;
// ptr == &x
                                                                                                                                                                                           cout << 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 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 F[3]; // CAUTION: uninitialized!
                                                                                                                                                                                                      int G[]; // ERROR: unclear size
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).

▲ 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]; // needed for swapping
 arr[i] = arr[(size - 1 - i)];
}</pre> | arr[ij - arr[(size - 1 - 1)]; | arr[(size - 1 - i)] = temp; |} // Note: arr[i] == *(arr + i) == i[arr] |} // Therefore, &arr[i] == (arr + i)

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)).

cout << &arr[0].</p> cout << arr, and cout << &arr would all print 0x1008.

```
1 int foo = 7;

2 int& bar = foo; // value of bar is foo (7)

3 int* ptr = &foo;

4 int arr[3] = { 4, 5, 9 };

5 cout < arr < endl; // prints 0x1008 (decay)

6 cout << (arr + 2) << endl; // prints 0x1010
                                                                                                                                                                                     foo, bar
                                                                                                                                               0x1004
                                                                                                                                                                                        \mathcal{I}_{\mathsf{ptr}}
                                                                                                                                                                   0x1000
                                                                                                                                              0x1008
                                                                                                                                                                                         arr[0]
                                                                                                                                                                         4
                                                                                                                                              0x100c
                                                                                                                                                                          5
                                                                                                                                                                                         arr[1]
                                                                                                                                                                                         arr[2]
```

Pointer arithmetic: adding an integer in to a pointer vields 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.

Pointer Operations // Mainly for pointers into the same array int arr[4] = { 6, 5, 8, 12 }; int* ptrl = &arr[0], *ptr2 = &arr[3]; cout << *arr << endi; // prints 6 cout << (ptr2 - ptrl) << endi; // prints 3 cout << (ptr1 - ptr2) << endi; // prints -3 assert(ptrl < ptr2); // true ptrl += 2; // ptrl now points at arr[2]

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 int arr[4] = { 1, 2, 3, 4 };
2 int (*arr_ptr)[4] = &arr; // pointer to entire array
3 cout << (*arr_ptr)[2] << endt; // prints 3
4 // ++arr_ptr would increment by the size of 4 ints</pre>
```

```
Traversal By Pointer: Pattern 1
```

```
int computeRange(int arr[], int N) {
   int* ptr = arr; int* end = (arr + N);
   // end is actually 1-past-the-end
   int* min = arr; int* max = arr;
   while (ptr < end) {
      if (*ptr < *min) { min = ptr; }
      if (*ptr > *max) { max = ptr; }
    }
}
| ++ptr;
|} // "walk" ptr across arr
| return (*max - *min);
|} // also could've used a for-loop
```

```
Traversal by Pointer: Pattern 2 (C-String Sanitization)
  ++fast_ptr; // ++fast_ptr every loop
  *slow_ptr = '\0'; // null-terminate when done
} // NOTE: '\0' is the only char considered "false"
```

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.

1 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.

```
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
```

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

```
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
```

```
const int x = 3;
int y = x; // 0k
const int* cptr = &x; // 0k
const int& cref = x; // 0k
int* ptr = cptr; // ERROR 1
int& ref = cref; // ERROR 2
                                                                                                                                                    1 int x = 2, y = 5;
2 const int *x_ptr = &x;
3 int *y_ptr = &x;
4 *y_ptr = *x_ptr; // 0k
5 y_ptr = x_ptr; /* ERROR (even 6 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 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 (-) " (-) For this "abcd a abcd" cout << (cstr + 1) << " " << *(cstr < 1) << " << *(cstr + 1) << " | The this "abcd a abcd" cout << (cstr + 1) << | The this "abcd a abcd" cout << (cstr + 1) << | The this "abcd a abcd" cout << (cstr + 1) << | The this "abcd a abcd" cout << (cstr + 1) << | The this "abcd a abcd" cout << (cstr -) | The this "abcd a abcd" cout << (cstr -) | The this "abcd a abcd" cout << (cstr -) | The this "abcd a abcd" cout << (cstr -) | The this "abcd a abcd" cout << (cstr -) | The this "abcd a abcd" cout << (cstr -) | The this "abcd a abcd" cout << (cstr -) | The this "abcd a abcd" cout << (cstr -) | The this "abcd a abcd" cout << (cstr -) | The this "abcd a abcd" cout << (cstr -) | The this "abcd a abcd" cout << (cstr -) | The this "abcd a abcd" cout << (cstr -) | The this "abcd a abcd" cout << (cstr -) | The this "abcd a abcd" cout << (cstr -) | The this "abcd a abcd" cout << (cstr -) | The this "abcd a abcd" cout << (cstr -) | The this "abcd a abcd" cout << (cstr -) | The this "abcd a abcd" cout << (cstr -) | The this "abcd a abcd" cout << (cstr -) | The this "abcd a abcd" cout << (cstr -) | The this "abcd a abcd" cout << (cstr -) | The this "abcd a abcd" cout << (cstr -) | The this "abcd a abcd" cout << (cstr -) | The this "abcd a abcd" cout << (cstr -) | The this "abcd a abcd" cout << (cstr -) | The this "abcd a abcd" cout << (cstr -) | The this "abcd a abcd" cout << (cstr -) | The this "abcd a abcd" cout << (cstr -) | The this "abcd a abcd" cout << (cstr -) | The this "abcd a abcd" cout << (cstr -) | The this "abcd a abcd" cout << (cstr -) | The this "abcd a abcd" cout << (cstr -) | The this "abcd a abcd" cout << (cstr -) | The this "abcd a abcd" cout << (cstr -) | The this "abcd a abcd" cout << (cstr -) | The this "abcd a abcd" cout << (cstr -) | The
```

	Length	Copy Value	Index	Concatenate	Compare
<string></string>	str.length();	str1 = str2;	str[i];	str1 += str2;	str1 != str2;
<cstring></cstring>	strlen(cstr);	strcpy(cstr1, cstr2);	cstr[i];	strcat(cstr1, cstr2);	strcmp(cstr1, cstr2);

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		

```
File I/O Ex 1: Print Lines From File
 #include <fstream> // defines if/ofstreams
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 << endl; }
   // could close inFS via inFS.close(); }
} // inFS also closes when scope ends</pre>
  int main() {
```

```
Ex 2: Copy One File's Contents to Another
#include <fstream>
void copyFile(string file_in, string file_out) {
   // fstreams can use C-strings as file names too
   ifstream inFS(file_in);
     ofstream outFS(file_out);
   string input_str;
white (inFS >> input_str) {
    outFS << input_str << endl;
} // could use '\n' instead of endl</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 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
 istringstream inSS("1 3 5"):
istringstream unitS;
printPlusOne(inSS, outSS);
cout << outSS.str() << endl; // Prints "2 4 6 "</pre>
```

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 the start of a C-strings—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
if (string(argv[i]) == "add") {
   int sum = 0;
   for (int i = 2; i < argc; ++i) {
        sum += stoi(argv[i]);
   }
}</pre>
```



ADTs, Structs and Classes

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; 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". Avoid accessing the member data of an ADT directly (even in tests) because it breaks the interface.

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.

Array traversal: arrays can be traversed by index or by pointer.

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() : Ani 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
void Triangle_scale(Triangle *t, double s) {
   t->a *= s; // "->" is necessary here
}
// C++ Class (Inside Body)
class Triangle { // "this->" optional
    void scale(double s) { this->a *= 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
 // (++ Class
Triangle t1; // Calls default ctor
Triangle t2(3,4,5); // calls 3-argument ctor
Triangle t3 = Triangle(3,4,5); // ditto
// Syntax for classes AMD structs:
Triangle t4{3, 4, 5};
Triangle t5 = {3, 4, 5};
Triangle t6 = Triangle{3, 4, 5};
```

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)
class IntSet {
    ... // contains() is also a member function
   bool operator[](int v) const:
3;
bool IntSet::operator[](int v) const {
    return contains(v);
```

```
<< and == Overload Examples (Top-Lev
1 class Line { /* start/end are public */ ... };
 3
  ostream& operator<<(ostream& os, Line line) {
4  | return os << line.start << line.end;</pre>
   } // 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::~Base() outside class body</pre>

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

Access Modifier Out-of-scope access Derived class access

class Derived : public Base {

13 belived d,
14 d.print(); // prints "D"
15 d.printB(); // prints "B"
16 d.Base::print(); // prints "B"
17 Base* b = &d;

18 b->print(); // prints "B"

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 . / ->

A 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.

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

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 derived dtor is called first, and the base dtor last).

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

```
× No

✓ Yes

                                                                     class B : public A {
public: // Derived class
| B() {cout << "B_ctor";}
| ~B() {cout << "B_dtor";}</pre>
class A {
nublic: // Base class
   A() {cout << "A_ctor";}
~A() {cout << "A_dtor";}
```

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

Derived d:

public

private

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.

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).

```
class Bird { }; // Base class
class Chicken : public Bird { };
class Duck : public Bird { };
Bird b; Chicken c; Duck d;
b = c; // Legal, but "slices" c's data
Bird* b_ptr = &c; // Good, no slicing
c = b; // ERROR (illegal assignment)
Chicken* c.ptr = &b; // ERROR (downcast)
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.

```
// 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.

```
};
// Note: ctors can't be virtual, but dtors can
// override always goes at end of signature
Duck duck;
Bird* duck_ptr = &duck;
duck_ptr->talk(); // prints "Quack"
// Scope resolution operator can suppress virtual
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.

```
class Abst { // Abstract Class
| public: virtual void foo() = 0;
}; // Note the lack of empty braces
class Concrete : public Abst {
    public: void foo() { cout << "foo"; }
};</pre>
Concrete c;
Abst* cptr = &c; Abst& cref = c; // ok
Abst abst_object; // COMPILE ERROR
c.Abst::foo(); // RUNTIME ERROR (or UB)
```

 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

	Container	Interface Operations - Optimal Implementations are All O(1)								
	Stack	empty	size	top (next	to pop)	push_back	pop_back			
ĺ	Queue	empty	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). To do so, keep track of the data using head (inclusive) and tail (exclusive) variables

		, 1				,					
	Useful std::vector Functions					Unsorted Set	Sorted Set	Stack	Queue	Array	List
-46	.size() v[i]/.at(i) .push_back(val) .pop_back() .resize(n)				insert/remove	O(n)	O(n)	O(1)*	O(1)*	O(n)	O(1)
.51ze()	V[1]/.at(1)	.pusn_back(val)	.pop_back() .resize(n)	search	O(n)	$O(\log n)$	O(n)	O(n)	O(n)	O(n)	
.front(.back()	.erase(iterator)	.empty()	.clear()	access	O(1)	O(1)	O(n)	O(n)	O(1)	O(n)

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

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 All of the standard containers listed above (except std::array s) are default-constructed to empty containers. They each also have <code>.begin()/.end()</code> public member functions and support for <code>operator=.</code>

```
C++ Standard Library Containers
                                                                                                                                                                  Useful std::map function
std::array<int, 4> arr = {-4, 0, 3, 6};
std::array arr2[1,2]; // Only way to omit size
std::setsint> unique = {3,2,2,1}; // [1,2,3}
// Note: can't index into a std::list
std::list-cint> values = {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 pair of <Key, T value> into a map
// Returns <iterator, false> if Key in map already
pair<iterator, bool> insert(const Pair_type &val);
// Maps store <key, value> pairs
std::map<string, int> EECS = { "Bill", 183} };
EECS["Emily"] = 203; // Multiple ways to add
EECS.insert(pair<string, int>"1.3maes", 280));
cout << EECS["James"] << endl; // prints 280</pre>
                                                                                                                                        // 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	Resizable	O(1) Indexing	Contiguous	Duplicate Items	Insertion and Removal	Search
std::array	Sequential	Unsorted	× No	✓ Yes	✓ Yes	✓ Yes	O(n) front/O(1) back	O(n)
std::vector	Sequential	Unsorted	✓ Yes	✓ Yes	✓ Yes	✓ Yes	O(n) front/O(1) back	O(n)
std::list	Sequential	Unsorted	✓ Yes	× No	× No	✓ Yes	O(1)	O(n)
std::map	Associative	Ascending key	✓ Yes	× No	× No	✗ Not keys	$O(\log n)$	$O(\log n)$
std::set	Associative	Ascending	✓ Yes	× No	× No	× No	$O(\log n)$	$O(\log n)$

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.

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

```
Class Template Syntax
                                                                                                                              Function Template Syntax
                                                                                                template <typename T> // "T" = an arbitrary parameter name
// Note: "class" also works in place of "typename"
T maxValue(const 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:
    void insert(T my_val);
    bool contains(T my_val) const;
    int size() const;
    orivate:
          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;
```

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.

• 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.

Iterator Type	Random Access Iterators	Bidirectional Iterators	Forward Iterators		
Interface Operations	-> [] ++ == != < *it +n	++ == != *it	*it ++ == !=		
Data Structures	std::vector, std::array	std::list, std::map, std::set	std::forward_list		

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</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;</pre>
```

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 $O(n^2)$, $O(n^3)$, etc). Big- Θ and Big- Ω represent average and lower bounds, respectively.

```
 \boxed{ O(1) \quad O(\log n) \quad O(\sqrt{n}) \quad O(n) \quad O(n\log n), \quad 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}) }
```

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: $O(n^2 + n + \log n) = O(n^2)$.

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

Non-nested loops: sum the complexities of each operation inside the loop body and 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 $O(n^2)$.

Partitioning/repeated division: procedures that divide the "remaining steps" by a constant number each time they execute (e.g. binary search/searching a sorted set) are usually $O(\log n)$.