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.

ℹ 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**.

Object Storage Durations
- Compiler-Controlled
  - Static
  - Automatic (local)
- Programmer-Controlled
  - Dynamic

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.

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

Atomic (primitive) types: types whose objects 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.

```cpp
// Four different ways to initialize an int to 5
int a = 5; int b(5); int c{5}; int d = {5};
```

```cpp
// Explicitly cast an int 'd' to a double 'e'
double e = static_cast<double>(d);
```

Objects in C++ are **statically-typed**. Although an object may evaluate to a different type in an expression, the type of an object itself cannot change (class objects obey this rule too).

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.

Pointers and Arrays

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

❗ An `int*` pointer variable can *only* point to an `int`; an `int**` pointer variable can *only* point to an `int*`; and so on. (E.g. attempting to make an `int*` pointer point to a `double` will lead to a compile error.)

Dereferencing a pointer: getting the object at an address. Note that the star `*` operator is used both to declare pointers and to dereference them (similarly, the `&` operator is used both to get an object's address and to declare a reference).

```cpp
int x = 3; int y = 4;
int *ptr = &x; // ptr initialized to x's address
cout << *ptr; // dereferences ptr/prints 3
ptr = &y; // no star...assigns y's address to ptr
*ptr = 6; // dereferences ptr/assigns 6 to y
```

ℹ Assigning `ptr = ptr2` *copies* the address stored by `ptr2` to `ptr` (subsequently changing `ptr2` wouldn't change `ptr`).

ℹ A reference to a reference is really another reference for the "original" object.

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

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 nulled; to do so, set it equal to `nullptr` (`0` or `NULL` also work but are bad style).

Common Pointer Bugs/Errors

⚠ Dereferencing a default-initialized 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).

⚠ If a function returns a pointer or reference to one of its local variables (which die when the function returns), using the reference or dereferencing the pointer produces undefined behavior.

Number Swap Function

```cpp
void swap_pointed(int *x, int *y) {
    int tmp = *x;
    *x = *y;
    *y = tmp;
}

int main() {
    int a = 1216, b = 1261;
    swap_pointed(&a, &b);
}
```

Pointers vs References

References and pointers both enable working between stack frames (scopes) and indirection. Some ways they're different:

- References *must* be explicitly initialized (unlike pointers). This is because references are aliases for existing objects.
- Pointers must be dereferenced to access the objects they point at, while references are used "as-is".
- You can change the object that a (non-`const`) pointer points to, while a reference's binding to an object can't be changed.

Arrays and Pointer Arithmetic

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

```cpp
// Valid array declarations
int arr[3] = {1,2}; // {1,2,0}
int zeroArr[3] = {}; // {0,0,0}
```

```cpp
// Valid array declarations
int arr[] = {4,5,6};
int mat[][2] = {1,2,3,4};
```

```cpp
// INVALID array declarations
int junk[4]; // Undefined items
int err[2][] = {5,6,7,8}; // No
```

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.

```cpp
void add_five(int arr[], int size) {
  for (int i = 0; i < size; i++) { arr[i] += 5; }
} // arr[i] += 5 is equiv. to *(arr + i) += 5

int main() {
  int arr[] = { 10, 20, 30 };
  add_five(arr, (sizeof(arr) / sizeof(*arr)));
  cout << arr[1] << endl; // prints 25
} // 1[arr] is equiv. to arr[1], but bad style
```

Passing `arr` by value passes a pointer to `arr[0]` by value. Also, `arr[i]` is shorthand for pointer arithmetic followed by a dereference, i.e., `arr[i] = *(arr + i)`.

ℹ The `sizeof` operator returns the size of an object *in bytes*. In this example, `sizeof(arr)` alone would return 12, not 3.

ℹ `cout << &arr[0]`, `cout << arr`, and `cout << &arr` would all print `0x1008`.

```cpp
int foo = 7;
int& bar = foo; // value of bar is foo (7)
int* ptr = &foo;
int arr[3] = { 4, 5, 9 };
cout << arr << endl; // prints 0x1008 (decay)
cout << (arr + 2) << endl; // prints 0x1010
cout << (&foo + 1) << endl; // prints 0x1004
```

	0x1000	7	foo, bar
	0x1004	0x1000	ptr
arr	0x1008	4	arr[0]
arr	0x100c	5	arr[1]
arr	0x1010	9	arr[2]

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.

Pointer Operations

```cpp
// Mainly for pointers into the same array
double arr[4] = { 2.5, 5.0, 8.0, 7.0 };
double* ptr1 = &arr[0], *ptr2 = &arr[3];
cout << *arr << endl; // prints 2.5
cout << (ptr2 - ptr1) << endl; // prints 3
cout << (ptr1 - ptr2) << endl; // prints -3
(ptr1 > ptr2); // equates to false (0)
ptr1 += 2; // ptr1 now points at arr[2]
```

Using the `&` operator on an array produces a pointer to the entire array, not a pointer to the first element or a pointer to a pointer (`&` does not require a value, so it doesn't cause decay).

```cpp
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
```

Traversal By Pointer: arrays can be traversed by pointer (mostly used with C-strings and iterators).

Traversal By Pointer: Pattern 1

```cpp
int const SIZE = 3;
int arr[SIZE] = {-1, 7, 2};
int *ptr = arr;
int *end = arr + SIZE;
// int* end is just past the end of arr
while (ptr < end) {
  cout << *ptr << endl;
  ++ptr; // "Walk" ptr across arr
} // Alternative to while loop below
```

```cpp
for (; ptr < end; ++ptr) { ... }
```

Traversal by Pointer: Pattern 2 (C-String Sanitization)

```cpp
void sanitize_username(Account *acc, char to_remove) {
  char *ptr_a = acc->username, *ptr_b = acc->username;
  while (*ptr_a && *ptr_b) { // while not '\0'
    if (*ptr_b != to_remove) {
      *ptr_a = *ptr_b;
      ++ptr_a; // ++ptr_a only when a char gets copied
    }
    ++ptr_b; // ++ptr_b every time the loop executes
  }
  *ptr_a = '\0'; // null-terminate string when done
}
```

The const Keyword

The `const` type qualifier prevents objects from being modified after initialization (attempting to do so causes a compile error). `const` scalars 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 positioning of `const` matters for arrays of pointers.

Special const Type Syntax

```cpp
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
```

```cpp
const int &ref_a = x; // reference-to-const
int const &ref_b = x; // reference-to-const
const int arr_a[2] = {1, 2}; // array of consts
int const arr_b[2] = {3, 4}; // array of consts
```

const Conversions and Passing

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

```cpp
int foo(int* a) { ... }
int bar(int b) { ... }
int func(const int* c) { ... }
const int x = 3;
bar(x); func(&x); // both ok
foo(&x); // ERROR
```

```cpp
const int x = 3;
int y = x; // Ok
const int* cptr = &x; // Ok
const int& cref = x; // Ok
int* ptr = cptr; // ERROR 1
int& ref = cref; // ERROR 2
```

```cpp
int x = 2, y = 5;
const int *x_ptr = &x;
int *y_ptr = &y;
int *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

Using C-Strings and Strings

```cpp
const char* msg = "Welcome!"; // Only works for string literals; use .c_str() on string variables
char color[] = "00274C"; // Create 7-element array (including \0) and copy a string literal to it
// Note: '\0' is the only char that evaluates to false (useful for traversal-by-pointer loops).
cout << cstr << " " << *cstr << " " << &cstr[0] << endl; // prints "abcd a abcd"
cout << (cstr + 1) << " " << *(cstr + 1) << " " << (*cstr + 1); // prints "bcd b 98"
string xyz = string(cstr); // Explicitly copy cstring to a string (implicit copy would work too)
```

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

Streams and File I/O

Input redirection	Output 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 Example: Print Lines From File

```cpp
#include <fstream> // defines (if/of)stream objects
int main() {
  ifstream inFS;
  inFS.open("file.txt"); // valid
  if (!inFS.is_open()) { return 1; }
  string my_string; // initialized to empty string
  while (getline(inFS, my_string)) {
    cout << my_string << endl;
  } // could close inFS manually via inFS.close();
} // inFS also closes when scope ends/main returns
```

Ex: Copy One File's Contents to Another

```cpp
#include <fstream>
int main() {
  ifstream inFS("input.txt"); // Also valid
  ofstream outFS("output.txt");
  string my_string;
// newline and space both "delimit" words
  while (inFS >> my_string) {
    outFS << my_string << '\n';
  } // '\n' is the newline char
}
```

istringstream: a stream that "simulates" input from a hardcoded string.

ostringstream: a stream that captures output and stores it in a string (use `.str()` to get the string).

```cpp
string input = "abc";
istringstream inSS(input);
```

```cpp
ostringstream outSS; // (i/o)stringstream are defined in <sstream>
Mat_print(mat, outSS); // Capture output
```

ℹ `ifstream`, `istringstream`, and `cin` can all be passed to a function with an `istream&` parameter. Likewise, `ofstream`, `ostringstream`, and `cout` can all be passed to a function with an `ostream&` parameter.

Command-Line Arguments

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

`argv`: functionally, an array of the arguments. Technically, `argv` is passed to `main` as a pointer to an array of pointers to C-strings. So `argv[0]` is a pointer to a C-string that represents the name of the program.

```cpp
#include <iostream>
#include <string> // includes stoi()/stod()
int main(int argc, char* argv[]) { // char** argv also OK
  if (string(argv[1]) == "add") {
    int sum = 0;
    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
```

Terminal

```
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.

ℹ `const` class-type objects must have their data members initialized (or a runtime error will occur).

⚠ A `const` instance of a `class` or `struct` cannot call non-`const` member functions.

Arrow -> operator: shorthand for pointer dereferencing followed by member access. `(*ptr).x == ptr->x;`

ℹ 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".

ℹ Accessing the member data of an ADT directly is said to break the interface and should be avoided. Unit tests should also respect the interface (because they should test behavior, not implementation).

C++ Classes

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

Constructors

- ℹ The compiler implicitly creates a default ctor iff there are no user-defined ctors.
- ℹ The order in which members are declared in a class is *always* the order they're initialized in.
- ℹ Initialization values from a member init. list overwrite initializations made during declarations.
- ⚠ Data members that aren't included in a ctor's member initializer list or initialized at declaration get default-initialized/constructed.
- ❗ 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(string name_in)  // Non-default ctor
    : name(name_in) {  }  // Member init. list
  Animal()  // Default ctor (no arguments)
    : Animal("Blank") {  } // ctor delegation
}; // Note the semicolon here!

class Bird : public Animal {
private: bool has_wings;
public: Bird(string name, bool wings_in)
        : Animal(name), has_wings(wings_in) {  }
}; // Derived class ctors must call base ctor

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

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

C-Style Struct vs Class Syntax

ADT Function Definition

```
// C-Struct
void Triangle_scale(Triangle *t, double s) {
  t->a *= s; // "->" is necessary here
}
```

```
// C++ Class (Inside Body)
class Triangle {
  void scale(double s) {
    this->a *= s; // "this->" optional here
  } // this-> implicit iff no name conflicts
};
// C++ Class (Outside Body)
void Triangle::scale(double s) { ... }
```

Object Creation/Manipulation

```
// C-Struct
Triangle t1, t2;
Triangle_init(&t1, 3, 4, 5);
t2 = {6, 8, 10}; // Assignment
Triangle t3{3, 4, 5}; // Initialization
printSides({3, 4, 5}); // Ok if pass-by-val
```

```
// C++ Class
Triangle t0; // Calls default ctor
Triangle t1(3,4,5); // calls 3-argument ctor
Triangle t2 = Triangle(3,4,5); // ditto
Triangle t3{3, 4, 5}; // ditto
Triangle t4 = {3, 4, 5}; // ditto
Triangle t5 = Triangle{3, 4, 5}; // ditto
```

const function definition

```
// C-Struct
double Triangle_perimeter(const Triangle *t)
  { ... } // const goes inside argument list
```

```
// C++ Class
class Triangle { // const "this->"
  double perimeter() const { ... }
}; // const comes after signature
```

Nested Classes and Constructors

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

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

```
class Book {
public:
  Book(double price_in)
    : price(price_in) { }
// Note: no default Book ctor
private:
  double price;
};
```

```
class Person {
public:
  Person(string& n, double p)
    : name(n), favBook(p) { }
private:
  string name;
  Book favBook;
};
```

Inheritance and Polymorphism

Function Overloading (Ad Hoc Polymorphism) and Operator Overloading

Function Overloading: using one name to refer to functions with *different signatures*. Functions can only be overloaded in the same scope (otherwise the "closer" scope takes priority). Note: `const`/non-`const` passing only alters the signature of functions with pointer/reference parameters (or implicit `this->` pointers).

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

- ❗ 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
public:
  bool operator[](int v) const;
};

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

<< and == overload example (top-level)

```
class Line {...}; // start/end are public members

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
```

Inheritance and Derived Classes

All base class members (EXCEPT constructors and destructors) become implicit members of derived classes. So you can call non-private base functions on derived class objects or access non-private inherited data members via (`.`)/`->`

- ⚠ Creating a derived class object *always* calls a base class ctor (if it's not explicitly called, the compiler attempts to implicitly call the base's default ctor). A base class dtor is always called when a derived object dies.

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

```
class Base {
public:
  void print() { cout << "B" << endl; }
};

class Derived : public Base {
public: // Hiding Base::Print()
  void print() { cout << "D" << endl; }
  void print_b() { Base::print(); }
};

Derived d;
d.print(); // prints "D"
d.print_b(); // prints "B"
d.Base::print(); // prints "B"
Base* b = &d; // another way to un-hide
b->print(); // prints "B"
```

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

Indirect access of inherited privates

```
class Base {
private:
  int x = 5;
public:
  int* x_ptr = &x;
  int get_x() const { return x; };
};
class Derived : public Base { };
```

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

Summary: how access modifiers affect *direct* access

Modifier	Accessible to derived classes	Accessible out of scope
`public`	✔ Yes	✔ Yes
`private`	✖ No	✖ No
`protected`	✔ Yes	✖ No

Destructors: special functions that are invoked when a class object's lifetime ends (e.g. when you `delete` a dynamic object or when a local object goes out of scope). Syntax: `Triangle::~Triangle() {}`

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

Subtype Polymorphism and Class Casting

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

```
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);
```

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

Virtual Functions and the `override` Keyword

Here, the *receiver* of the call to `talk()` on line 13 has a **static type** known at compile time (`Bird`) and a **dynamic type** known at runtime (`Duck`). Member lookup starts in the static class, so `Duck::talk` won't hide `Bird::talk`. Instead, `Bird::talk` is declared as `virtual` to make it dynamically-bound.

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

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

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

Pure Virtual Functions and Abstract Classes

Pure virtual function: a `virtual` base-class function that has no meaning or implementation; they simply make up part of the interface for derived classes. To declare one, add `= 0;` to the end of a function's signature.

Abstract class: a class with a pure virtual member function. Note that derived classes of an abstract class will also be abstract unless they override (define) every pure virtual function they inherit.

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

```
class Abst { // Abstract Class
public: virtual void foo() = 0;
}; // Note the lack of empty braces

class Concrete : public Abst {
public: void foo() { cout << "foo"; }
}; // public/private doesn't matter here

Concrete c;
Abst* c_ptr = &c; Abst& c_ref = c; // ok
Abst abst_object; // COMPILE ERROR
c.Abst::foo(); // RUNTIME ERROR (or U.B)
```

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

Container ADTs and Templates (Array-Based Data Structures)

Container ADTs

`static` keyword: used to make one copy of a class data member "shared" between all instances of that class. A `static` data member has static storage duration but exists only within the scope of a class.

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

stack: a container that's designed to operate in a LIFO order.
queue: a container designed to operate in a first-in/first-out (FIFO) order.

Stack and Queue Interfaces

Container	Operations - optimal implementations are all O(1)					
Stack	`empty`	`size`	`back/top (next)`		`push_back`	`pop_back`
Queue	`empty`	`size`	`front (next)`	`back (last)`	`push_back`	`pop_front`

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

Useful `<vector>` functions

`.size()`	`v[i]`	`.push_back(val)`	`.pop_back()`	`.resize(n)`
`.front()`	`.back()`	`.at(i)`	`.empty()`	`.clear()`

Operation	Unsorted Set	Sorted Set	Stack	Queue	Array
`insert, remove`	$O(n)$	$O(n)$	$O(1)^*$	$O(1)^*$	$O(n)$
`contains`	$O(n)$	$O(\log n)$	$O(n)$	$O(n)$	$O(n)$
`access`	$O(1)$	$O(1)$	$O(n)$	$O(n)$	$O(1)$

`SortedIntSet::remove()` implementation

```
void remove(int v) {
  int i = indexOf(v); // indexOf() is a member
  if (i == -1) { return; }
  for (; i < elts_size - 1; ++i) {
    elts[i] = elts[i + 1];
  }
  --elts_size; // elts_size == cardinality
}
```

`SortedIntSet::insert()` implementation

```
void insert(int v) {
  if (indexOf(v) != -1) { return; }
  int i = elts_size;
  for (; (i > 0) && (elts[(i-1)] > v); i--)
    { elts[i] = elts[(i - 1)]; }
  elts[i] = v;
  ++elts_size;
}
```

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.

Class Template Syntax

```
template <typename T>
class UnsortedSet {
public:
  void insert(T my_val);
  bool contains(T my_val) const;
  int size() const;
private:
  T elts[ELTS_CAPACITY];
  int elts_size;
  ...
}; // Syntax: UnsortedSet<type> s;
```

Function Template Syntax

```
template <typename T> // "T" = an arbitrary parameter name
// Note: "class" also works in place of "typename".
T maxValue(const T &valA, const T &valB) {
  return valB > valA ? valB : valA; // Note: "?" = ternary
} // This function returns the greater of valA and valB
// Syntax to call it: maxValue<int/double/etc>(...);
```

Templated Class Member Function Syntax

```
template <typename T> // Necessary if outside class body
void UnsortedSet<T>::insert(T my_val) {...}
```

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

Iterators, Traversal By Iterator, and Range-Based Loops

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

- ℹ `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).

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

```
vector<int> v = { 1, 2, 3, 4 };
for (vector<int>::iterator it = v.begin(); it != v.end(); ++it)
  { cout << *it << endl; } // ::const_iterator if const vector
```

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
```

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

Time Complexity

We define **runtime complexity** in terms of *number of steps*, not literal runtime. Big-O runtime is an *upper-bound* of the magnitude of a function's runtime with respect to input size (this means that any $O(n)$ function is also $O(n^2)$, $O(n^3)$, etc). Big-Θ and Big-Ω represent average and lower bounds, respectively.

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

- ℹ Functions in the same complexity class should have growth rates that differ by a constant factor as $n \to \infty$. So $O(2^n)$ and $O(8^n)$ are NOT in the same complexity class, but $O(\log_2 n)$ and $O(\log_3 n)$ are.

Determining Big-O Time Complexity

Constant coefficients: drop them unless they're part of an exponent. n, $2n$, and $3n$ are all $O(n)$.

Addition (sequential procedures): highest-complexity term dominates. E.g. $2n + \log n = O(n)$.

Multiplication: find the product of the individual terms' complexities. E.g. $O(n) \times O(\log n) = O(n \log n)$.

Non-nested loops: *sum* the complexities of each operation *inside* the loop body, and then *multiply* that by the number of times the loop executes. E.g. a loop that runs from `0` to `n` and has a $O(1)$ body is $O(n)$.

Nested loops: start at the innermost loop and work outwards (the individual complexities of the loops should multiply). E.g. two nested $O(n)$ loops are $O(n^2)$.

Partitioning/Repeated Division: procedures that divide the "remaining steps" each time they execute (e.g. binary search, for-loops that double the loop variable after each iteration, etc) are usually $O(\log n)$.

$\log(xy) = \log(x) + \log(y)$	$\log\left(\frac{x}{y}\right) = \log(x) - \log(y)$	$\log(x^n) = n \log(x)$	$\log_b(x) = \frac{\log_c(x)}{\log_c(b)} = \frac{1}{\log_x(b)}$