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: 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).

ℹ 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 References

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).

```cpp
int x = 3; int y = 4;
int* ptr1 = &x; int* ptr2 = &y;
int** ptr1_ptr = &ptr1;
ptr2 = ptr1; // copies x's address from ptr1 to ptr2
ptr1 = &y; // ptr1 now points at y
**ptr1_ptr = 6; // now y == 6
*ptr2 = 2; // ptr2 still points to x, so now x == 2
```

ℹ Creating a reference to a reference creates another reference to the original object.

ℹ Creating a pointer to a reference creates a pointer to the original object.

```cpp
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
cout << *b << endl; // Prints 5
cout << y << endl; // prints 0x2710
cout << &(*z) << endl; // prints z (0x2710)
cout << *(&z) << endl; // prints z (0x2710)
```

0x2710	5	x,a
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

⚠ 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.

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

```cpp
int& danglingRef(int x) { return x; } // BUGGY
```

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

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

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

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

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

Arrays and Pointer Arithmetic

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

```cpp
int A[3] = {1,2}; // {1,2,0}
int B[3] = {}; // {0,0,0}
int C[] = {1,2}; // size == 2
```

```cpp
int D[][2] = {1,2,3}; // {{1,2},{3,0}}
int E[][3] = {1,2,3}; // {{1,2,3}}
int F[3]; // CAUTION: uninitialized!
```

```cpp
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, 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).

⚠ 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.

```cpp
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)
```

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.

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

ℹ `cout << &arr[0]` and `cout << &arr` would also print `0x1000`.

ℹ You can create pointers/refs to array elements.

```cpp
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
cout << &ptr2 << endl; // prints 0x1014
cout << ptr[-1] << endl; // prints 10
```

arr	0x1000	5	arr[0]
	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.

Pointer Operations

```cpp
// Mainly for pointers into the same array
int arr[4] = { 5, 10, 15, 20 };
int* ptr1 = &arr[0], *ptr2 = &arr[3];
cout << *arr << endl; // prints 5
cout << (ptr2 - ptr1) << endl; // prints 3
cout << (ptr1 - ptr2) << endl; // prints -3
assert(ptr1 < ptr2); // true
ptr1 += 2; // ptr1 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.

```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
```

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

Traversal By Pointer: Pattern 1

```cpp
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 (const int* p = arr; p < end; ++p) {
    if (*p < *min) { min = p; }
    if (*p > *max) { max = p; }
  } // "walk" the pointer across arr
  return (*max - *min);
}
```

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

```cpp
void sanitize(char username[], char to_remove) {
  char* slow = username, *fast = username;
  while (*slow && *fast) { // while not '\0'
    if (*fast != to_remove) {
      *slow = *fast;
      ++slow; // ++slow only if we copy
    }
    ++fast; // ++fast every loop
  }
  *slow = '\0'; // null-terminate when done
} // NOTE: '\0' is the only char considered "false"
```

The const Keyword

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.

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 = 42; // reference-to-const
const int arr_a[2] = {1, 2}; // array of consts
int const arr_b[2] = {3, 4}; // array of consts
```

```cpp
const int* A[] = {…}; // pointer-to-const array
```

```cpp
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 example, 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.

```cpp
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
```

```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;
*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)

```cpp
char color[] = "00274C"; // Create 7-element array (including \0) and copy a string literal to it
const char* cstr = "abcd"; // Only works for string literals; use .c_str() on string variables
cout << cstr << " " << *cstr << " " << &cstr[0] << endl; // prints "abcd a abcd"
cout << (cstr + 1) << " " << *(cstr + 1) << " " << (*cstr + 1); // prints "bcd b 98" ('a' == 97)
string str = string(cstr); // Explicitly copy cstring to a string (implicit copy would work too)
```

Length		Assignment		Index	Concatenation			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`

File I/O Ex 1: Print Lines From File

```cpp
#include <fstream> // defines if/ofstreams
int main() {
  ifstream inFS; // or inFS("file.txt");
  inFS.open("file.txt");
  if (!inFS.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
```

Ex 2: Copy One File's Contents to Another

```cpp
#include <fstream>
void copyFile(string file_in, string file_out) {
  // file streams also accept C-strings as arguments
  ifstream inFS(file_in);
  ofstream outFS(file_out);
  string input_str;
  while (inFS >> input_str) {
    outFS << input_str << endl;
  } // could use '\n' instead of endl
}
```

ℹ 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.

```cpp
#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");
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 C-strings—so `argv` is passed to `main()` as a pointer to an array of pointers to C-strings).

```cpp
#include <string> // defines 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.

❗ 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;`

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

C++ Classes

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

Constructors

- ℹ The compiler implicitly creates a default ctor iff there are no user-defined ctors (same for dtors).
- ℹ The order in which members are declared in a class body is always the order they're initialized in.
- ℹ Initialization values from a member init. list take precedence over initializations made at declaration.
- ⚠ 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

```cpp
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
```

```cpp
// 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

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

```cpp
// 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

```cpp
// C-Style Struct
Triangle t1;
Triangle_init(&t1, 3, 4, 5);
```

```cpp
// C++ Class
Triangle t1; // Calls default ctor
Triangle t2(3,4,5); // calls 3-argument ctor
Triangle t3 = Triangle(3,4,5); // ditto
Triangle t4{3, 4, 5}; // ditto
Triangle t5 = {3, 4, 5}; // ditto
Triangle t6 = Triangle{3, 4, 5}; // ditto
// The last 3 work for classes and structs
```

const Function Definition

```cpp
// C-Style Struct
double area(const Triangle* t) {···}
// const goes inside argument list
```

```cpp
// 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)

```cpp
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)

```cpp
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

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
```

Inheritance and Derived Classes

All base class members (EXCEPT ctors and dtors) become implicit members of derived classes. So you can call any public base class function on derived class objects or access inherited base class public members via `.`/`->`.

- ⚠ Creating a derived class object *always* calls a base class ctor. If you don't call one explicitly, the base class default ctor is implicitly called. Also, a base class dtor is always called when a derived object dies.

Member name lookup begins in the *static type* of a receiver/object and moves up the inheritance hierarchy (to the base class) if no match is found. It stops when it finds a matching name or reaches the top of the hierarchy.

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

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

class Derived : public Base {
public:
  void print() {cout << "D" << endl;}
  void printB() { Base::print(); }
};

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

Access Modifier	Out-of-scope access	Derived class access
`public`	✔ Yes	✔ Yes
`private`	✖ No	✖ No
`protected`	✖ No	✔ Yes

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).

- ℹ Non-dynamic objects are destroyed in reverse order of their creation.

```cpp
class A {
public: // Base class
  A() {cout << "A ctor ";}
  ~A() {cout << "A dtor ";}
};
```

```cpp
class B : public A {
public: // Derived class
  B() {cout << "B ctor ";}
  ~B() {cout << "B dtor ";}
};
```

```cpp
int main() {
  A a_object; // Prints "A ctor "
  B b_object; // Prints "A ctor B ctor "
  A* ptr = &b_object; // Doesn't print anything
} // When main() returns: "B dtor A dtor A dtor "
```

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.

```cpp
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`.

```cpp
// 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.

```cpp
1  class Bird {
2  ... // Can't use virtual outside 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 keyword is implicit here
8    void talk() const override { cout << "Quack"; }
9  }; // override == an optional "sanity check"
10
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"
```

- ℹ 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.

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

```cpp
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"; }
};

Concrete c;
Abstract* ptr = &c; Abstract& ref = c; // ok
Abstract abstract_object; // COMPILE ERROR
c.Abstract::foo(); // RUNTIME ERROR (U.B.)
```

- ❗ Don't try to call a pure virtual function or create an abstract class object.

Containers and Data Structures

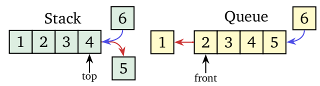
Container ADTs

`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.

queue: a container data structure that supports first-in/first-out (FIFO) access.

Container	Interface Operations - Optimal Implementations are All O(1)				
Stack	`.empty()`	`.size()`	`.top()`	`.push(val)`	`.pop()`
Queue	`.empty()`	`.size()`	`.front()`	`.push(val)`	`.pop()`



All operations on stacks/queues run in $O(1)$ *amortized* time (if `push` causes a grow operation, it will take $O(n)$ time).

- ℹ An efficient way to implement a queue is to create a vector with free space at both ends, i.e., a **circular buffer**.

Useful `std::vector` Functions and Constructors				
`.size()`	`v[i]/.at(i)`	`.push_back(val)`	`.pop_back()`	`.resize(n)`
`.front()`	`.back()`	`.erase(iterator)`	`.empty()`	`.clear()`
`vector<T> v2(v1.begin(), v1.end())`			`vector<T> v2(v1)`	

Operation	Unsorted Set*	Sorted Set*	Stack	Queue	Array	List
insert/remove	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$	$O(1)$
search	$O(n)$	$O(\log n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$
access	$O(1)$	$O(1)$	$O(n)$	$O(n)$	$O(1)$	$O(n)$

* These refer to the array-based sets we saw in class, *not* STL sets.

C++ Standard Library Containers

`std::array`: Containers with *compile-time constant* sizes that store elements in contiguous memory.

```cpp
int x = 5;
array<int, x> arr;
// invalid constant
```

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

```cpp
void foo(const int z) {
  array<int, z> arr;
} // invalid constant
```

```cpp
int a = 5; const int b = a;
array<int, b> arr;
// still not a valid constant
```

`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.

- ℹ 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

```cpp
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

```cpp
// 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[]	Insert/Erase	Search
<array>	Sequential	Unsorted	✖ No	✔ Yes	✔ Allowed	✔ Allowed	$O(1)$	–	$O(n)$
<vector>	Sequential	Unsorted	✔ Yes	✔ Yes	✔ Allowed	✔ Allowed	$O(1)$	$O(n)$ < end, $O(1)$ end	$O(n)$
<list>	Sequential	Unsorted	✔ Yes	✖ No	✔ Allowed	✔ Allowed	–	$O(1)$	$O(n)$
<set>	Associative	Asc. key	✔ Yes	✖ No	✖ Not Allowed	✖ Not Allowed	–	$O(\log n)$	$O(\log n)$
<map>	Associative	Asc. Key	✔ Yes	✖ No	✖ Keys/✔ Vals	✖ Keys/✔ Vals	$O(\log n)$	$O(\log n)$	$O(\log n)$

Iterator	`array<T,N>::iterator`	`vector<T>::iterator`	`list<T>::iterator`	`map<Key,T>::iterator`	`set<Key>::iterator`
Operations	`[] ++ -- == < *it +n`	`[] ++ -- == < *it +n`	`++ -- == *it`	`++ -- == *it`	`++ -- == *it`

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.

```cpp
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
```

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

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.

Class Template Syntax

```cpp
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

```cpp
// Note: "class" also works in place of "typename"
template <typename T> // "T" is also an arbitrary name
T maxValue(const T &valA, 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>(...);
```

Templated Class Member Function Syntax

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

- ⚠ 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).

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

Inserting/Erasing: inserting into or erasing 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 $O(1)$.

- ℹ 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 $O(1)$.

Searching: searching an *unsorted* sequence for a value takes $O(n)$ time. A *sorted* sequence can optimally be searched in $O(\log n)$ time with **binary search**.

```cpp
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 { L = (middle + 1); }
  } // eliminates half the search space each loop
  return -1; // if we didn't find v
} // This is O(log n), but it requires sorted input
```