

Fundamentals and Machine Model

Machine/Memory Model and the Function Call Stack

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

Variable: a *name* in source code that is associated with an object at compile time.

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

❗ The value stored by a variable's memory object may change, but the association between a variable and an object itself can only change when the variable goes out of **scope**.



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

❗ 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.

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

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

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, Arrays and References

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 (and the `&` operator is used both to get an object's address and to declare a reference).

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

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

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

```
1 int x = 5;
2 int* y = &x; // creates pointer to x
3 int* z = y; // creates another pointer to x
4 int &a = x; // creates reference to x
5 int* &b = z; // creates reference to z
6 cout << &b << endl; // Prints 5
7 cout << &y << endl; // Prints 0x2714
8 cout << y << endl; // prints 0x2710
9 cout << &(x) << endl; // equiv. to cout << &x
10 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 be 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), dereferencing that pointer or using that reference produces undefined behavior. Functions should only return pointers or references to objects whose lifetimes extend beyond the function call.

```
1 int* returnPtr(int x) // BUG
2 { return &x; }

1 int& returnByRef(int x) // BUG
2 { return x; }

1 int returnRef(int x) // BUG
2 { int& y = x; return y; }
```

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.

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

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

1 // INVALID array declarations
2 int junk[4]; // Undefined items
3 int err[2][1] = {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.

```
1 void addFive(int arr[], int size) {
2     for (int i = 0; i < size; i++) { arr[i] += 5; }
3 } // arr[i] += 5 is equiv. to *(arr + i) += 5
4
5 int main() {
6     int arr[] = { 10, 20, 30 };
7     addFive(arr, (sizeof(arr) / sizeof(*arr)));
8     cout << arr[1] << endl; // prints 25
9 } // i[arr] is equiv. to arr[i], 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)`.

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

```
1 cout << &arr[0],
2 cout << arr, and
3 cout << &arr
4 would all print
5 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
7 cout << (&foo + 1) << endl; // prints 0x1004
```

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

Pointer Operations

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

```
1 int arr[4] = { 1, 2, 3, 4 };
2 int (*arr_ptr)[4] = &arr; // pointer to entire array
3 cout << (*arr_ptr)[2] << endl; // prints 3
4 // *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	Traversal by Pointer: Pattern 2 (C-String Sanitization)
<pre>1 int const SIZE = 3; 2 int arr[SIZE] = {-1, 7, 2}; 3 int *ptr = arr; 4 int *end = arr + SIZE; 5 // int* end is just past the end of arr 6 while (ptr < end) { 7 cout << *ptr << endl; 8 ++ptr; // "Walk" ptr across arr 9 } // Alternative to while loop below 1 for (; ptr < end; ++ptr) { ... }</pre>	<pre>1 void sanitize_username(Account *acc, char to_remove) { 2 char *ptr_a = acc->username, *ptr_b = acc->username; 3 while (ptr_a && *ptr_b) { // while not '\0' 4 if (*ptr_b != to_remove) { 5 *ptr_a = *ptr_b; 6 ++ptr_a; // ++ptr_a only when a char gets copied 7 } 8 ++ptr_b; // ++ptr_b every time the loop executes 9 } 10 *ptr_a = '\0'; // null-terminate string when done 11 }</pre>

The const Keyword

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

const pointers: pointers that can modify what they point at but cannot be re-pointed to different objects.

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

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

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

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

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).

Special const Type Syntax
<pre>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 = x; // reference-to-const const int arr_a[2] = {1, 2}; // array of const int const arr_b[2] = {3, 4}; // array of const</pre>

- 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

❗ Output streams treat pointers to char arrays differently from pointers to other types of arrays.

```
1 const char* cstr = "abcd"; // Only works for string literals; use .c_str() on string variables
2 char color[] = "00274C"; // Create 7-element array (including '\0') and copy a string literal to it
3 // Note: '\0' is the only char that evaluates to false (useful for traversal-by-pointer loops).
4 cout << cstr << " " << *cstr << " " << &cstr[0] << endl; // prints "abcd abcd"
5 cout << (cstr + 1) << " " << *(cstr + 1) << " " << *(cstr + 1); // prints "bcd b 98" ('a' = 97)
6 string xyz = string(cstr); // Explicitly copy cstring to a string (implicit copy would work too)
```

	Length	Copy Value	Index	Concatenate	Compare
<string>	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

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	Ex 2: Copy One File's Contents to Another
<pre>1 #include <fstream> // defines (if/of)stream objects 2 int main() { 3 ifstream inFS; 4 inFS.open("file.txt"); // valid 5 if (!inFS.is_open()) { return 1; } 6 string my_string; // initialized to empty string 7 while (getline(inFS, my_string)) { 8 cout << my_string << endl; 9 } // could close inFS manually via inFS.close(); 10 // inFS also closes when scope ends/main returns</pre>	<pre>1 #include <fstream> 2 int main() { 3 ifstream inFS("input.txt"); // Also valid 4 ofstream outFS("output.txt"); 5 string my_string; 6 while (inFS >> my_string) { 7 outFS << my_string << '\n'; 8 // newline and space both 'delimit' strings 9 } 10 }</pre>

istream: an object that "simulates" input with a string as its source. Note: an `istream`, an `ifstream` and `cin` can all be passed to a function with a `std::istream&` parameter.

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

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 that each point to the start of a C-string—so `argv` is passed to `main` as a pointer to an array of pointers to C-strings).

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

Terminal

```
hugokin@ubuntu:~$ ./main.exe add 7 2
Sum: 9, arg: 4, argv[1]: add
hugokin@ubuntu:~$ ./main.exe add 1 2 3
Sum: 6, arg: 5, argv[1]: add
hugokin@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 can't 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 pointer dereferencing 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 structs 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 structs default to public access/inheritance.

Constructors

- 1 The compiler implicitly creates a default ctor if there are no user-defined ctors.
- 2 The order in which members are declared in a class is *always* the order they're initialized in.
- 3 Initialization values from a member init. list over-write initializations made during declarations.
- 4 Data members that aren't included in a ctor's member initializer list or initialized at declaration get default-initialized/constructed.
- 5 A delegating ctor must contain a call to the other ctor (and nothing else) in its member init. list.

```
1 class Animal {
2 private: string name;
3 public:
4   Animal(string name,in) // Non-default ctor
5   : name(name.in) { // Member init. list
6   Animal() // Default ctor (no arguments)
7   : Animal("Blank") {} // ctor delegation
8 }; // Note the semicolon here!
9
10 class Bird : public Animal {
11 private: bool has_wings;
12 public: Bird(string name, bool wings.in)
13   : Animal(name), has_wings(wings.in) {}
14 }; // Derived class ctors must call base ctor
15
16 class Duck : public Bird {
17 private: string color;
18 public: Duck(string name, bool wings, string rgb)
19   : Bird(name, wings), color(rgb) {}
20 }; // Calling Bird ctor also calls Animal ctor
21
22 // This is how to define a ctor OUTSIDE of body
23 Duck: Duck(string name, bool wings, string rgb)
24 : Bird(name, wings), color(rgb) {}
```

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.

```
1 class Book {
2 public:
3   Book(double price.in)
4   : price(price.in) {}
5 // Note: no default Book ctor
6 private:
7   double price;
8 };
9
10 class Person {
11 public:
12   Person(string& n, double p)
13   : name(n), favBook(p) {}
14 private:
15   string name;
16   Book favBook;
17 };
18
19 const Book b1 = Book(1.5);
20 const Person p1 = Person("John", 1.5);
```

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).

- 1 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).

```
1 // overload example (member)
2 class IntSet {
3 ... // contains() is also a member function
4 public:
5   bool operator==(int v) const;
6 };
7
8 // overload example (top-level)
9 class Line {...}; // start/end are public members
10
11 ostream& operator<<(ostream& os, Line line) {
12   return os << line.start << line.end;
13 } // os needs to be passed by non-const ref here
14
15 bool operator==(const Line &a, const Line &b) {
16   return (a.start == b.start && a.end == b.end);
17 } // 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 non-private base class function on a derived class object or access non-private inherited data members via ./->

- 1 Creating a derived class object *always* calls a base class ctor. If you don't call one explicitly, the compiler will insert an implicit call to the base default ctor (causing a compile error if one doesn't exist). Also, 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.

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

```
1 Indirect access of inherited privates
2 class Base {
3 private:
4   int x = 5;
5 public:
6   int* x_ptr = &x;
7   int get_x() const { return x; };
8 };
9
10 class Derived : public Base { };
11
12 Derived d; // cannot directly access x
13 cout << *(d.x_ptr) << endl; // prints 5
14 cout << d.get_x() << endl; // prints 5
15
16 Summary: how access modifiers affect direct access
17
18 Modifier | Accessible to derived classes | Accessible out of scope
19 ----- | ----- | -----
20 public   | Yes | Yes
21 private | No  | No
22 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() {}
1 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.

```
1 class Bird { }; // Base class
2 class Chicken : public Bird { };
3 class Duck : public Bird { };
4 Bird b; Chicken c; Duck d;
5 b = c; // Legal, but "slices" c's data
6 Bird* b_ptr = &c; // Good, no slicing
7 c = b; // ERROR (illegal assignment)
8 Chicken* c_ptr = &b; // ERROR (downcast)
9 Duck* d_ptr = &c; // ERROR
10
11 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.
12
13 1 // Be careful - validity not checked at runtime:
14 2 Chicken* c_ptr_a = static_cast<Chicken*>(bird_ptr);
15 3 // Bird needs at least 1 virtual function for this:
16 4 Chicken* c_ptr_b = dynamic_cast<Chicken*>(bird_ptr);
17
18 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 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.

```
1 class Abst { // Abstract Class
2 public: virtual void foo() = 0;
3 }; // Note the lack of empty braces
4
5 class Concrete : public Abst {
6 public: void foo() { cout << "foo"; }
7 }; // public/private doesn't matter here
8
9 Concrete c;
10 Abst* c_ptr = &c; Abst& c_ref = c; // ok
11 Abst abst_object; // COMPILER ERROR
12 c.Abst::foo(); // RUNTIME ERROR (or U.B)
```

- 1 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.

		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

queue: a container designed to operate in a first-in/first-out (FIFO) order.

- 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.

		Useful <vector> functions					
.size()	v[i]	.push_back(val)	.pop_back()	.resize(n)			
.front()	.back()	.at(i)	.empty()	.clear()			

```
1 SortedIntSet::remove() implementation
2 void remove(int v) {
3   if (i == indexOf(v), // indexOf() is a member
4     if (i == -1) return;
5     for (i = 0; i < elts_size - 1; ++i) {
6       elts[i] = elts[i + 1];
7     }
8   --elts_size; // elts_size == cardinality
9 }
10
11 SortedIntSet::insert() implementation
12 void insert(int v) {
13   if (indexOf(v) != -1) return;
14   int i = elts_size;
15   for (i = 0; i < elts_size - 1; ++i) {
16     if (elts[i] > v) break;
17   }
18   elts[i] = v;
19   ++elts_size;
20 }
```

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.

```
1 Class Template Syntax
2 template <typename T>
3 class UnsortedSet {
4 public:
5   void insert(T my_val);
6   bool contains(T my_val) const;
7   int size() const;
8 private:
9   T elts[ELTS_CAPACITY];
10   int elts_size;
11 }; // Syntax: UnsortedSet<type> s;
12
13 Function Template Syntax
14 template <typename T> // "T" = an arbitrary parameter name
15 // Note: "class" also works in place of "typename".
16 T maxVal(const T &aVal, const T &bVal) {
17   return aVal > bVal ? aVal : bVal; // Note: "?" = ternary
18 } // This function returns the greater of aVal and bVal
19
20 Templated Class Member Function Syntax
21 template <typename T> // Necessary if outside class body
22 void UnsortedSet<T>::insert(T my_val) {...}
```

- 1 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.

- 1 std::begin() returns an iterator to the start of an STL container. std::end() returns an iterator that's 1 past the end of an STL container (the iterator returned by std::end() should not be dereferenced).

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
```

```
1 Range-Based For Loop (Works on Any Sequence Traversable by Iterator)
2 vector<int> v = { 1, 2, 3, 4 };
3 // for (<type> <variable> : <sequence>) {...}
4 for (int item : v) { // works with arrays too
5   cout << item << endl;
6 } // could also declare item as const or a ref
7
8 // Compiler translation of range-based for loop
9 for (auto it = v.begin(); it != v.end(); ++it) {
10   int item = *it;
11   cout << item << endl;
12 }
```

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.

O(1)	O(log n)	O(√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^n^n)
------	----------	-------	------	------------	------------	--------	--------	--------	-------	--------	----------

- 1 Functions in the same complexity class should have growth rates that differ by a constant factor as n → ∞. 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 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 × 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. Ex: a loop that runs from 0 to n with an 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, for-loops that double the loop counter) are usually O(log n).

log(xy) = log(x) + log(y)	log(x/y) = log(x) - log(y)	log(x^n) = n log(x)	log_b(x) = log_c(x) / log_c(b) = 1 / log_b(b)
---------------------------	----------------------------	---------------------	---