Inheritance

Class Relationships

- **encapsulates data and behavior to create objects.**

- “is-a” (“**inheritance**”)
    - One class is a **specialization** of another
    - A Car “is-a” Vehicle; a Boat “is-a” Vehicle, etc.
- “has-a” (“**composition**”)
    - A Car “has-a” Wheel (and lots more stuff)
- Classes can also just *collaborate*
    - One class uses another’s methods

Inheritance

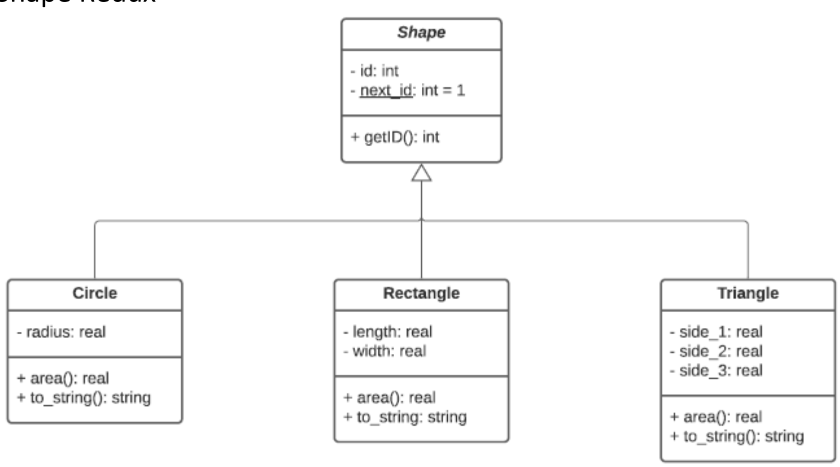
- Implements “is-a” relationships
- Terms:
    - **Base class** (“superclass”), e.g., **Vehicle**
    - **Derived Class** (“subclass”), e.g., **Car**
- Derived Classes “inherit” (or “derive”) from Base Classes
- Inheritance can span many levels
    - Type hierarchies

The Ubiquitous Shape Hierarchy

Benefits of Inheritance

- Models *hierarchical* relationships
    - Concrete Derived objects can **substitute** for base objects
    - “Substitutability Principle”
- Code Sharing
    - All **common** data and functions are **defined once** and **shared** by all derived classes

Shape Redux



Abstract Base Classes

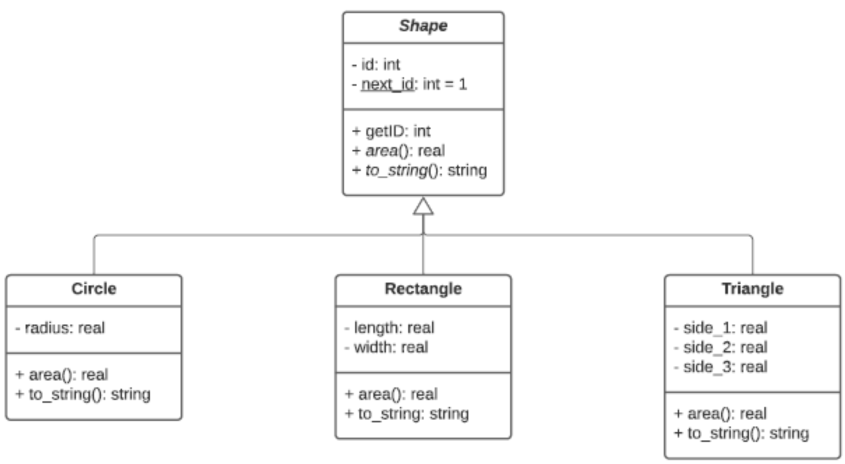
- **Not meant to be instantiated**
- Establish the **interface** for the hierarchy
    - Functions available to “users” of the class

- **Can contain shared data and code**
- **The leaf (bottom) classes are concrete**
    - **No virtual functions within the concrete class**
    - Meant to be *instantiated*
    - Revisit C++ streams hierarchy (slide 4–53)

Dilemma
- We want to establish **area** and **to_string** as part of the Shape hierarchy *interface*
    - So, they should be specified in **Shape**
- But derived classes must *implement* them
    - And we want to "force" them to

Abstract Member Functions *What Makes a Class Abstract*
- Part of the **interface**
    - Don't need implementations in the base class
- Should be **overridden** in the derived classes
    - Providing bodies (like we did for concrete shapes)
        - Otherwise, the derived classes would also be *abstract*
- Should be declared **pure virtual**
    - **= 0** after the method *signature*

Finished Shape Hierarchy

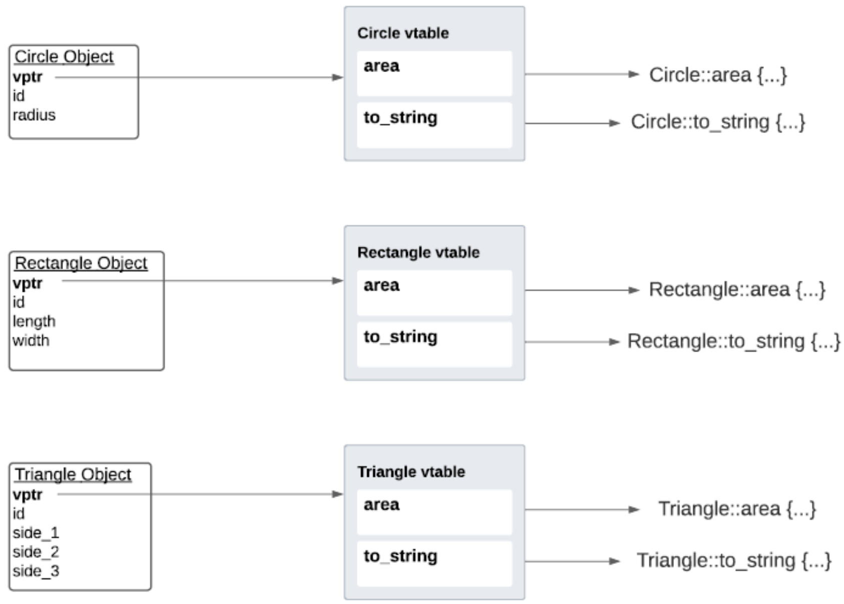


Virtual Functions *Runtime Polymorphism*

***Polymorphism* refers to determining which program behaviour to execute depending on data types.**
- Must be used with **pointers** or **references**
    - Not plain objects
- **Pointers often point to *heap* memory**
    - Obtained by the **new** operator
- Polymorphism requires declaring functions **virtual**

How Virtual Functions Work

- Each **class** with virtual functions has a **vtable**
    - Pointers to its function *implementations*
    - 1 table *per class*
- Each **instance** has a **vptr**
    - Points to the **vtable** for its *class*
    - So, objects have *extra space* due to the **vptr**



Shape Virtual Functions

Protected Access

- In between **public** and **private**
- **Allows *derived* classes access**
    - *Without* being **public**

Overriding vs. Overloading

- *Overloaded* functions have *different signatures* and appear in the same scope (resolved at compilation):
    - void f(int n) {…}
    - void f(const string& s) {…}
    - void f(int n, int m) {…}
- *Overrides* have the *same signature* but appear *in derived classes*(resolved at runtime through **vptrs**)
    - Related by *inheritance*

Initializing Shared Data

- What will the constructors look like?

Base Class Constructors

- Called automatically
    - Just like member object constructors
- Called **before** the derived type's constructor
    - And **before** member objects' constructors

• Initialize via the constructor **initializer list**
    • Just like with member object initialization

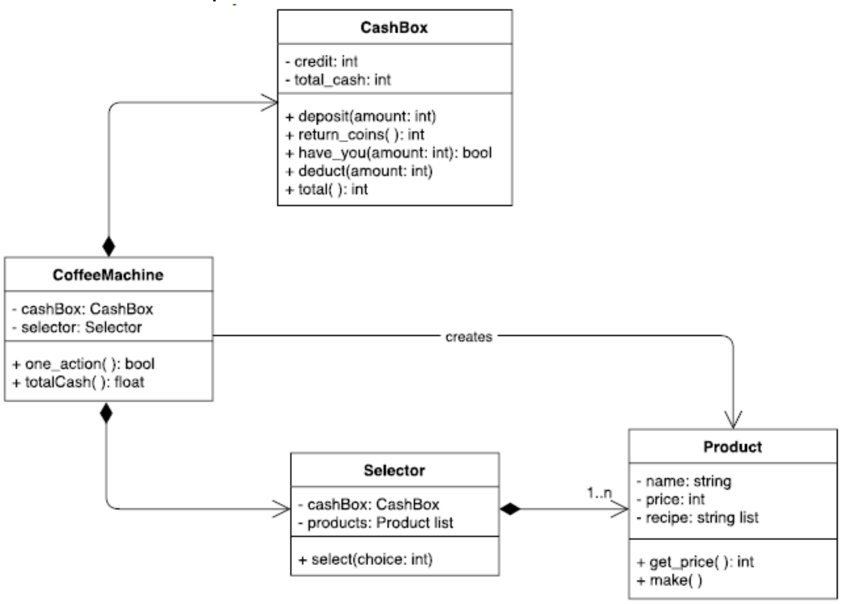
Alert!

• There is a *potential flaw* in everything we have done!
• What if someone has a **Shape* p**, pointing to heap memory?
    • Then we have to call **delete** on it
• But wait! Only the **Shape** destructor will be called!
    • Because it's a **Shape***

Virtual Destructors *They need to be Polymorphic too!*

• Class hierarchies should always declare the destructor **virtual** in the **top** base class
• Otherwise, deleting an object **through a base pointer** will only call the **base destructor**
    • The derived portions of derived objects will **not** be destructed!
    • *Bottom line*: Destruction should also be **polymorphic**!

Has-a Relationships



Summary

• **Is-a** relationships are modeled by **inheritance**
• **Polymorphism comes via virtual functions**
    • Using **pointers** or **references** to objects only
• Abstract classes encapsulate:
    • The hierarchy's **interface**
    • **Shared** data and code
• **Has-a** relationships sometimes use **pointers**

Chapter 13 Templates

Motivation

• Consider the swap function:

void swap(int& n, int& m) {int temp = n;n = m;m = temp;}

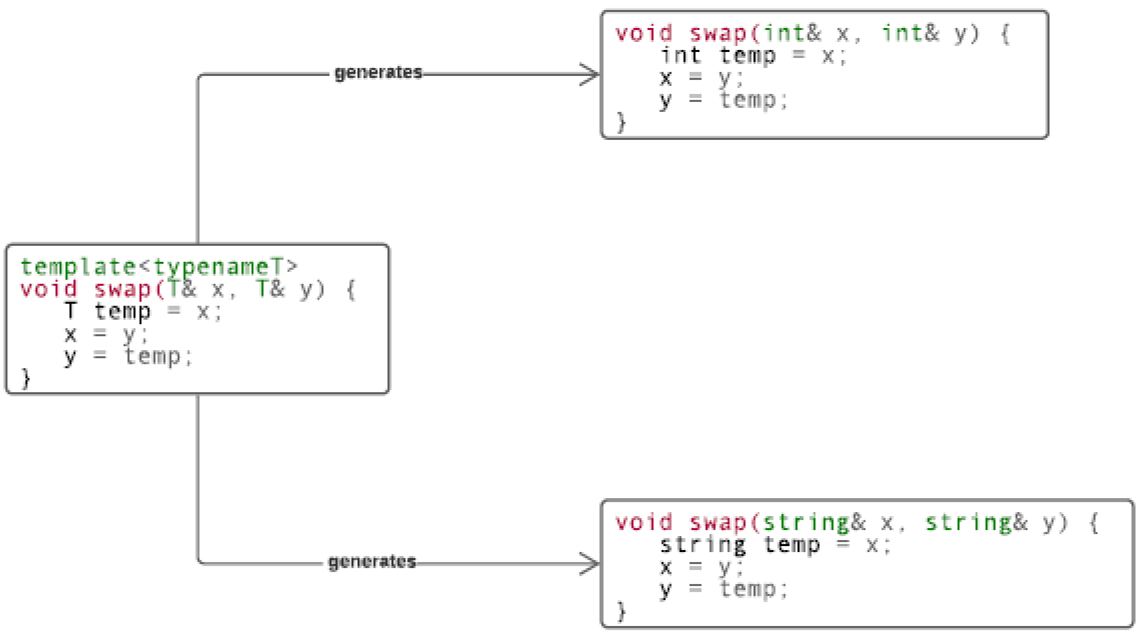
• **The logic is the same for any type**
• So, we make the **type** a **parameter** with a template…

A Generic swap

```
template<typename T>

void swap(T& x, T& y) {
T temp = x;
x = y;
y = temp;
}
int main() {
int a = 1, b = 2;
swap(a, b);
cout << a << ", " << b << endl; // 2, 1
string s("one"), t("two");
swap(s, t);
cout << s << ", " << t << endl; // two, one
}
```

Compile-time Code Generation



About Templates

- Can have **multiple** template **parameters**
    - Parameters can be **types** or **integers**
- The compiler generates **separate versions** of code
    - On demand according to usage
- It **infers** function template parameters from the **call**
    - **Class** templates must be instantiated **explicitly**

A Class Template

- **Template Arguments must be specified**
- Example:

- A “safe” Array class template
- Fixed size, no memory overhead
- But checks for access errors

Templates vs. Types

- **A template is *not* a type!**
- It is a “blueprint” for instantiating:
  - 1) *functions*, or
  - 2) *classes*(each such class is a *new type*)
- Therefore, **vector** is not a type, but…
  - **vector**<**int**> is a type
  - **vector**<**string**> is a type, **vector**<**T**> is a generic type…
  - Note the **print_array** function template in *array_t.cpp*

Where to put Template Code

- **Template code should be 100% in a .h file**
  - Including the function bodies!
  - For the same reason **inline** functions are in header files
- Why?
  - Because templates are not code, but *instructions* for *generating* code
  - The compiler needs all the information in order to generate code

Concurrency (not in book)

So What’s the Problem?

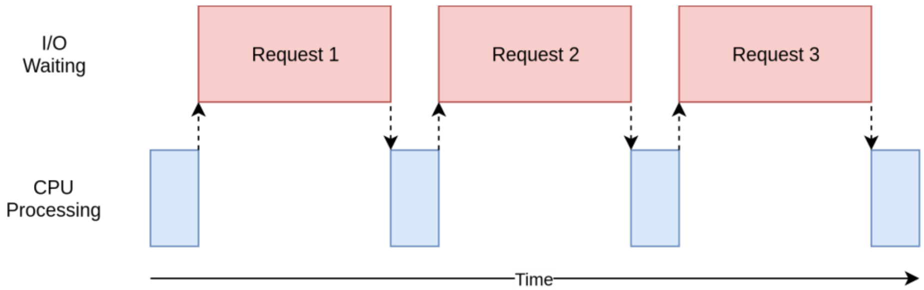
- Central Processing Units are MUCH faster than the input and output devices.
- So for many programs, the CPU is idle much of the time, while it waits for I/O to complete.
- You get a new computer with a faster CPU
- But if the I/O doesn’t also speed up, the new CPU spends a greater proportion of its time just waiting…

Terms

- Concurrency
  - When multiple, independent tasks are logically active at the same time (they may still take turns, though). This is possible on a single-processor, in which case it is called cooperative multi-tasking.
- Parallelism
  - Multiple, independent tasks actually running simultaneously (a special case of Concurrency). This requires multiple processors/cores.

Single Thread of Execution

- The CPU could work on something else while it is waiting

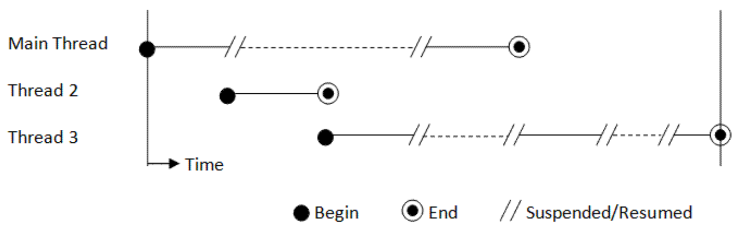


Threads
• A piece of code (usually a function) that can runs concurrently with other threads.

Multiple tasks:
- They can share the same CPU, taking turns while waiting for I/O (concurrency)
- They can run at the same time on separate CPUs

Programmatically, it's the same mechanism
- We let the operating system or hardware to do the actual assigning of tasks.

Taking turns



Threads in C++
- Create a thread object
- You pass it the name of the function for that task.
- The constructor launches a system thread immediately
- The main program (or whatever code created the thread) continues to run
    - Now you have two separate threads running
- You may create other threads the same way.

Join
- Eventually you have to wait for a thread to finish
- The calling thread (the one that created the thread object) waits for its child thread to finish.
- Call .join() to wait for the child thread to finish.

FAQs
How do I run a particular function as a separate thread?
- Create a thread object, with the name of a function as an argument to the constructor

What if the function takes arguments itself?
- Just make those arguments to the thread constructor: it will pass them to the function automagically

Can I have multiple threads that use the same function?

- Yes! They will be separate executions of the same function.

What about the local variables in said function?

- Each thread has its own copy of the LOCAL variables.

FAQ, #2

When does a thread end?

- When the function finishes
- But you still have to wait for it (join)

How many threads can I have at once?

- Lots!
- But each thread has some overhead with it…

Race Conditions

- My dessert wax program didn't work! **The output is all scrambled!**
- **cout is a shared resource**
- **The threads interrupted each other**
    - There are many machine level instructions being run for the output operation
    - The interruption can happen anywhere, anytime
- **We need to protect the output operation from being interrupted**
    - We put the code to protect in a critical section
    - Only one thread at a time should pass through that section

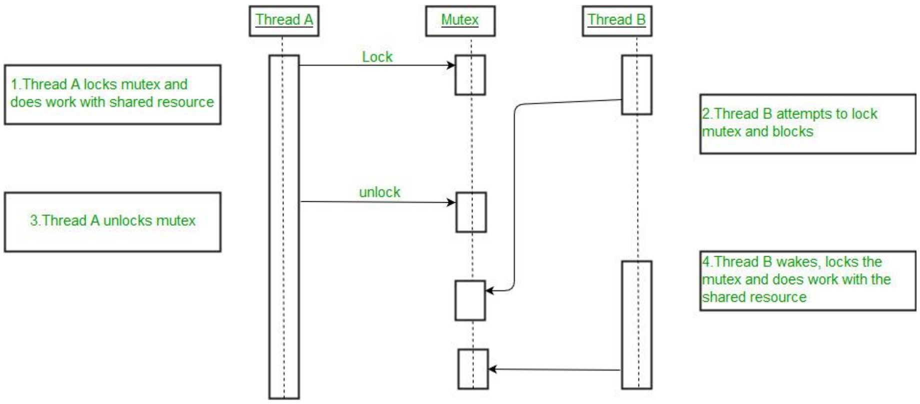
Shared resources

- Input and output streams
- Data in memory that is shared!
    - Because the CPU is faster than accessing memory
    - (We are only interested in data that might change)
- When one thread is using a resource, we must block the other threads from using it.

Mutexes

- Stands for "mutual exclusion"
- Has lock and unlock operations
    - Only one thread at a time can "hold the lock"
    - Effectively "synchronizes" blocks of code (critical sections)
- The mutex must exist outside the scope of the threads' functions
    - Could be global, or at class scope (usually static)

Lock/unlock sequence



Locks and Exception Safety
• What happens if an exception occurs inside a critical section? **unlock will not be called!**

RAII wrapper for mutexes:
- **• lock_guard**

• Solution: Resource Acquisition is Initialization (RAII)
- • Of course! There has been a bit of a movement to try to rename this concept as **Scope-Bound Resource Management**
- • 'Resource' doesn't just mean memory - it could be file handles, network sockets, database handles, GDI objects... In short, things that we have a finite supply of and so we need to be able to control their usage. The 'Scope-bound' aspect means that the lifetime of the object is bound to the scope of a variable, so when the variable goes out of scope then the destructor will release the resource. A very useful property of this is that it makes for greater exception-safety.

Sharing Multiple Resources
• Each resource requires a mutex
• Easy to deadlock
- • aka "deadly embrace"
- **• order of acquiring/releasing the lock matters!**

Locking Multiple Mutexes Effectively
• Obtaining multiple locks simultaneously in more complicated situations is tricky
• "Try-and-back-out" procedure
- • If you obtain lock1, and lock2 is busy, you must release lock1
- • And try again!

std::lock does this for you automatically!
- • Use the adopt_lock option for RAII…

https://cplusplus.com/reference/mutex/adopt_lock

Concurrency and Parallelism
- • Parallel computing requires multiple CPUs

Chapter 14 Generic Algorithms

- **Function** templates that have *type parameters*
- Commonly needed behavior for handling data
    - They use **begin** and **end** *iterators* to traverse data
    - Work for *any* sequence, including *arrays*
- **binary_search**, **copy**, **find**, **sort**, **transform**
- Almost 100 algorithms in the standard C++ library!
    - **Minimizes** the number of **loops** you need to write

    Iterators

Algorithms use Iterators

- Act like pointers to elements in a sequence
    - **begin** "points" at *first* element (position **0**)
    - **end** "points" 1 *past* the *last* element (position **n**)
- Different containers have different types of iterators
    - But they *all* work like *pointers*
        - *****iter, iter**->**member, **++**iter
    - You *rarely* have to do ptr *arithmetic* with algorithms!

std::find

- Returns *iterator* to first occurrence of search key
    - Or the **end** iterator if not found
- **find_if** uses a *predicate* function for matching
    - We pass a *function* as a *parameter*!
- Look at the implementation for **find_if** in zyBook

std::sort

- Sorts a *contiguous* sequence *in place*
- Can give it *a comparison function* to alter the *ordering*
    - e.g., ascending vs. descending order
    - Default to *ascending*(using the **<**operator)
    - The comparator returns **true** if the 2 elements are in order

back_inserter

- Turns *writing* to a **vector** into *appending*
    - To guarantee *space* for the incoming data
    - Used to add to an *existing* vector

Reading Items from a Stream *Directly into a vector*

- Use **istream_iterator**
    - **begin** iterator: istream_iterator<*type*>(*ifstream*);
    - **end** iterator: istream_iterator<*type*>();

Function Objects *aka "functors"*

- *Objects* that can be called as *functions*
    - By overloading operator()(<args…>)
- 1 function object class can take the place of *many functions*
    - By storing different data in different objects

Lambda Expressions

- A shorthand for creating anonymous *function objects*
    - On-the-fly in executable code (often as *arguments*)

- Without previously defining them via classes!
- Syntax:
  - [](<parm-list>) {<function body>}

std::string is a Container
- A sequence of characters
  - Containers provide iterators
  - So that they can be used with the standard algorithms!

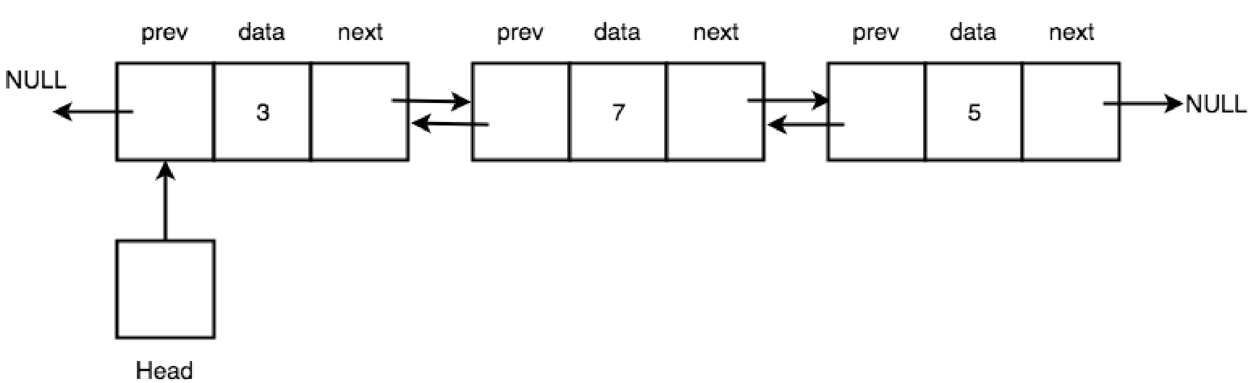
Lambda Capture
- When a function object accesses *non-local* data
  - Needs to *record* it to make it available later
- Can capture by **value** or **reference**

Chapter 15Containers

Standard C++ Containers
- Sequences
  - **vector**, **list**, deque, **array**
- Specialized Containers (Container Adaptors)
  - stack, queue, priority_queue
- Ordered Containers
  - set, **map**, **multiset**, multimap
- Unordered containers
  - unordered_set, unordered_map

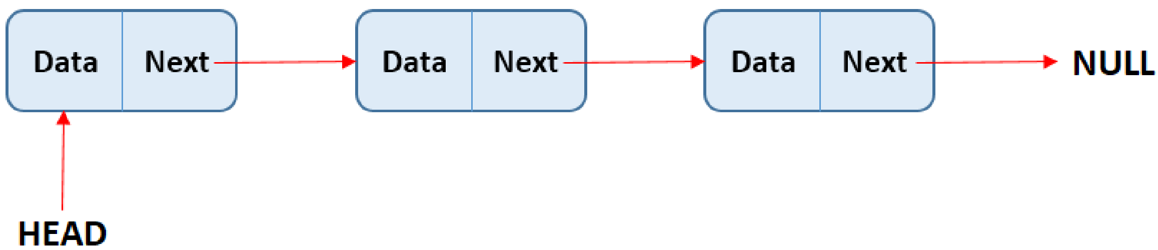
std::array
- A safe array
- Checks array bounds
- Can be passed by reference or value

std::list
  - A doubly-linked list
    - Can traverse forward (**++**)
    - or backwards (**--**)



  - There is also **std::forward_list**
    - A singly-linked list (less overhead)

- Can only traverse *forward*



Pairs and Tuples

- A convenient way to process *multiple values*
  - As loop *indexes*(using **structured bindings**)
  - As *return* values
  - Access items with **.first** and **.second**
- Tuples can hold more than 2 items
  - A pair is a tuple of size 2
  - Access with **get<n>(tup)**

std::set

- Allows no duplicates
  - Repeated entries are ignored
- Ordered by “less-than”
  - operator<

std::multiset

- Aka “bag”
  - Can hold duplicates
- Useful for checking for subsets

std::map

- Stores <key, value> pairs
  - You search them by key
    - Key and value can be separate types
  - An ordered container (by **operator<**)
- Can use the key as an array index!
  - mymap["greeting"] = "hello";
- Key methods:
  - at, emplace, erase, find, insert, insert_or_assign, []