CS 138 Feynman

C++ Bootcamp

- Getting arguments from command ~~line~~ line:
  - Code: (# of args, things inputted @ CLI)
    `int main (int argc, char * argv[])`
  - Terminal:
    `>> ./a.out Aaron`
  - Code:
    `argc = 2, argv[2] = {"./a.out", "Aaron"}`
- Strings vs. character strings:
  - `#include <string>` ⇒ string processing
  - Useful functions:
    - `s.length()` # of characters in s
    - `s.at(i)`: character of i^{th} place (`s[i]` ⇒ unchecked)
    - `s.substr(i, n)`: gives n character string of i^{th} position
    - `s.find(str, pos)`: finds index after pos that has str } -1 if nothing found
    - `s.rfind(str, pos)`: " last index before " " " " }
    - `s.insert(.)`, `s.replace(.)`: self-explanatory.
  - Character comparison:
    Recall: `s.at()`, `s[i]` ⇒ return characters
    Use characters for comparison.
  - Concatenation:
    Use strings for concatenation. Even if "M" + "6", will not work: need to explicitly cast into strings.
    ↳ Explicit " " ⇒ char * in C++
- I/O: `#include <iostream>`
  - Error output: `cerr << "Error message" << endl; exit(1);`
  - Grabbing whole line: `getline(cin, storageVar)` - has no '\n'
  - Variable-length input + EOF: 3 methods.
    1. Check if `cin.eof() == true`
    2. Check if `cin.fail() == true`

    Best ⇒ 3. Use cin as boolean: `while (cin >> var)` / `while (getline(...))`
    Requires to go to failure (one step too far)

Edge Case: Order

```
int n;
cin >> n;
if (cin.fail()) { ... }   // exit ✓
```

A

```
string flurble;
cin >> flurble;
if (cin.fail()) { exit(1); }
```

B

1. Input: hello123
   ↳ Works if B → A, but whole thing read as string
2. ~~123~~ Input: 123 hello
   n = 123, hello = "hello".
   ↳ 'h' is not a legal int
3. hello:
   Hits first fail

Redirection:

./a.out < input.txt > output.txt

- File I/O: #include <fstream>
  ① Instantiating fstream objects.

  Input ⇒ ifstream streamName (fileName)
  Output ⇒ ofstream streamName (fileName) ← C char str

  How to make char str:
  1. ~~argv~~ argv
  2. fileName.As Str.c_str(); ⇐ from #include <string>
  3. Use strings, but not string var.
     Ex: // ofstream ("passes.txt") ✓, ofstream (var);

  Can open later:
  ifstream stream; @ stream.open(...)

  ② I/O:
  Just do streamName >> or streamName <<

  ③ Closing:
  streamName.close();

- Default params:

```
int foo ( param1, param2, ..., defaultParam = "..." )
```

define @ end

- vector + arrays:

C-style: int arr [__]; ⇒ Problem: need to pass in size to func, passing in pointer.

#include <vector>:

vector: ⋮ ⇒ Flexible, dynamic

- Instantiation:

vector <data-type> name; // Preset size: name (size)

- API calls:

Access: arr.at(i); ⇒ Could do [], unchecked.

Size/capacity: arr.size(), arr.capacity()

Add: arr.push_back(__)

Remove last: arr.~~pop()~~ pop_back() → Change size

Front + back: arr.front(), arr.back()

Insertion: arr.insert(iterator, elem);

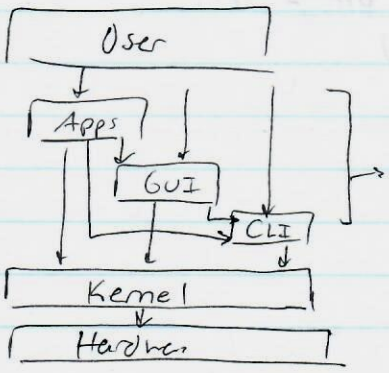
- C-style array things for vectors works!
- Iterator: think of it like pointer! Can do ptr arithmetic.

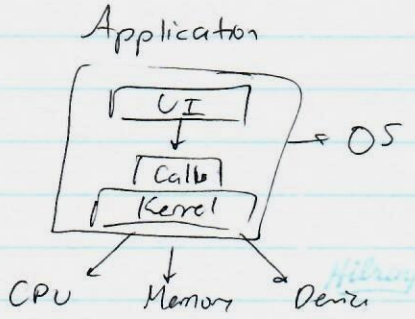
arr.begin(), arr.end(). ⇒ vector <type-pointing>::const_iterator name;

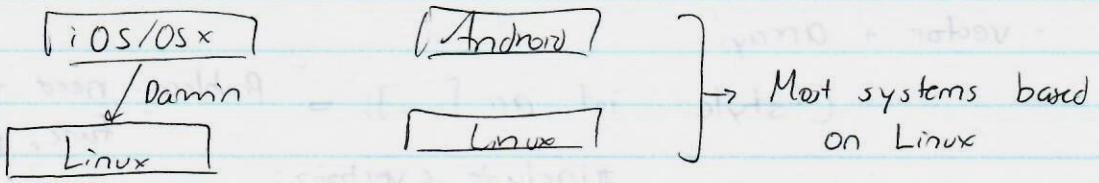
- Loopings:

1. for (int i=0; i < arr.size(); i++)
2. for (vector <data-type>::const_iterator it = arr.begin(); it != arr.end(); it++) ⇒ Access via *it ⇒ like @ ptr. (elem pointing to)
3. for (auto i : arr) {
   cout << i << endl;
   }

UNIX + SHELL







OS History:

Mainframes → Minicomputers → Microcomputers → Mobile
1950 1970 1980 2000s

Overview

AT&T Labs
↓
C + Unix → Reimplemented → BSD (UC Berkely)
Linux (open source)

GNU: completely free unix-like system ← No Unix code in it!
↳ gcc, utilities (ls, chmod, ...)

Linux: OS kernel w/ a distro (utilities to make Linux bdtbl)
↳ Linus Torvalds

How to Use Linux:

Shell: sh, bash, ksh ⇒ Interact w/ kernel

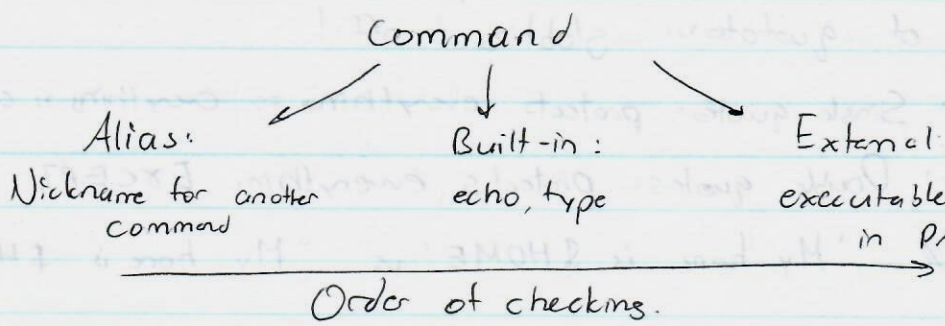
File system:

Has all of your stuff
↳ Access: '~', '$HOME'

Navigation between directories:

"Which dir am I in?"
pwd

"Change directory"
cd ⇒ Back home
cd - ⇒ prev. dir
cd .. ⇒ parent dir.
cd . ⇒ current dir.
cd "..." ⇒ Takes to rel. path



"What type is my command" ⇒ type ___

"Print smth to shell" ⇒ echo ___ ' ' , " " , ___ Don't do this

↳ Print variable: $___ (ex: // echo $HOME)

Globbing: regex. Combine w/ linux commands.

No dotfiles match!

- *: matches w/ 0 or more character.
  - ↳ Ex: // echo a2.* ⇒ Returns all files with a2 prefix
- ?: matches w/ only 1 character.
  - ↳ Ex: // Files: a2.b, a2.aa, a2.cpp
    echo a2.? ⇒ Return file w/ 1 character after period only a2.b

To match w/ certain values:

{...} ⇒ returns in set

↳ Files a2.b, a2.aa, a2.cpp

↳ echo a2.{b, aa} ⇒ a2.b, a2.aa

[...] ⇒ returns 1 character in set.

↳ Ex: // File a2.cpp, a3.cpp, a1.cpp, a4.cpp

Echo a[13].cpp ⇒ a1.cpp, a3.cpp

↳ [!...] ⇒ Matches to character that is not in it

Ex: // echo a[!3].cpp ⇒ a1.cpp, a2.cpp, a4.cpp

Ranges: hyphens:

[0-3]: 0 1 2 3, [!a-zA-Z] ⇒ no alphabets.

Role of quotation: globbing is off!

↳ Single quotes: protects everything ⇒ everything is a string.

↳ Double quotes: protects everything EXCEPT quotes + variables.

Ex: // 'My home is $HOME' vs. "My home is $HOME"

Alias: Nickname

- Set an alias: alias desiredCmd="command to nickname"
- Usage: shortening commands (not for files/directories)
- Duration: shell session
  - ↳ Store in .shellrc file + do source .shellrc to load up aliases in new session.

Other commands

man: short form for manual
↳ man bash man cd

ls: all contents of directory
↳ Options: -l: long listing, -a: dotfiles included
-R: recursive list, -t: sort by time modified
-F: type (*: file, /: dir, @: links) -G: color code

mkdir: create new dir

cp: copies files or dir
↳ cp src_file target_file ← Directory copy: -r

mv: move files
↳ mv src_file target_file/dir

rm: delete file
↳ rm src_file ~~target_file~~

cat: cout for shell
less: "

→ cat: continuous stream, less: pages
↳ Piping

↳ cat file or cat < file

I/O Redirection:

Same thing as redirection in C++

a.out >> , ~~>~~ a.out < input.txt

Difference between '>' and '>>' for output

> : write errors/output ⇒ removes things in file
>> : appending " ⇒ appends to file

Error stream:
↳ Output to file:
1> : stdout
2> : stderr
↳ Ex: // ./a.out < input.txt 1> txtfile.txt 2> error.txt
↳ Output to same file as output (stdout)
↳ Ex: // ./a.out 1> output.txt 2>&1 (error → out)
OR 1>&2 (out → err)

Unix Pipes:

'|': use output of previous to be input of next
↳ cat foo.txt | head -20 | wc -w ⇒ Word count of top 20 lines in foo.txt.

Commands:
uniq: removes adjacent duplicate lines
sort: sorts lines.

Put error in pipeline by using 2>&1 as well.

Hilary

Comparison

cmp: cmp firstFile secondFile
↳ Which character is diff. in which line

diff: diff firstFile secondFile
↳ How to change firstFile → secondFile

Find

find [dir-list] [expr]

Options:
- name "globbing Pattern" | -type fld
dirList is null: searches in directory current
expr is null: -name "*" | -maxdepth N

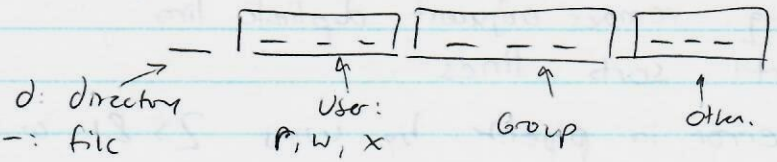
Ex: find -name "t*" ⇒ All files in whole directory (even subdirs) w/ t*

Logical expressions:
- not expr1 ⇒ Not
expr1 -a expr2 ⇒ AND
expr1 -o expr2 ⇒ OR

File Expressions

File
owner = User / Group / Others

Each user has 3 options: read, write, execute.
To see permissions: ls -l:



Change permissions:
chgrp: recursively changes groups
↳ chgrp [-R] group-name file/directory

chmod: permission change.

chmod [-R] mode-list file/directory

Mode list: security-level operator permission

- U: user
- G: group
- O: other
- A: all

- +: add
- -: remove
- =: setting

r, w, x

Ex:

chmod g-r, o-r, g-w, o-w foo

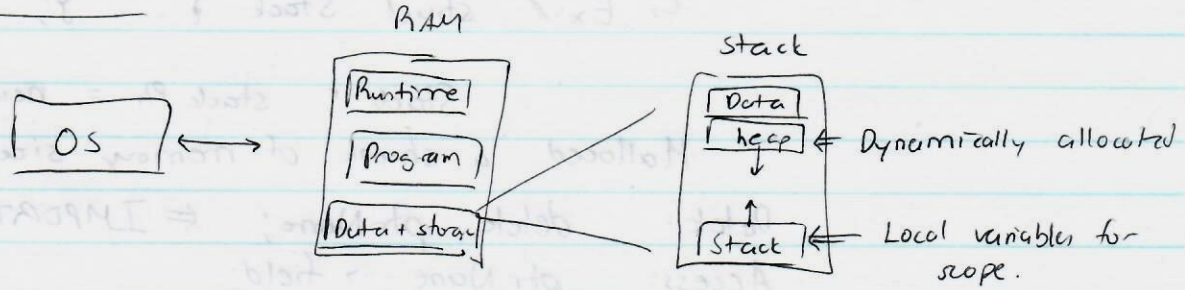
↳ Remove group read write, others read writ for foo

↳ Short form: go-rw

chmod g+rx cs138: Adding read + execute to group

LINKED STRUCTURES

Memory model



Each function/scope (including main) has its own stack.

↳ Done => remove stack (all variables lost unless returned

Compilation

C/C++: program → compiler → native code (hardware)

Java/C#: program → compile → universal VM (runtime env)

Other (Python): program → interpreter (translates + executes as you move along, no middle file)

Milroy

	Portable	Fast
Native code	no	Yes
VM	Yes	Yes
Interpreted Source	Yes	No

Objects

Create objects → state: runtime stack
                ← dynamic: heap

~~Dynamic~~ static instantiation:

ClassName objName;

Problem: disappears when you leave scope

Access: objName. ———

Dynamic instation:

ClassName * ptrName = new ClassName

↳ Ex: // struct Stack { . . . };

:

Stack * stackPtr = new Stack;

Malloced a chunk of memory size of stack.

Delete: delete ptrName; ⇐ IMPORTANT.

Access: ptrName -> field;

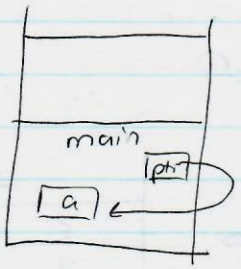
Pointers:

Tips:

1. Make memory model ~~pointer~~ diagrams
   ↳ Understand when objects allocated, ptr pointing to
2. Pay attention to ptr functions:

```
void foo (Stack* s) { . . }

Stack obj;
foo (obj); ⇐ Will not work
foo (&obj); ⇐ Will work
```

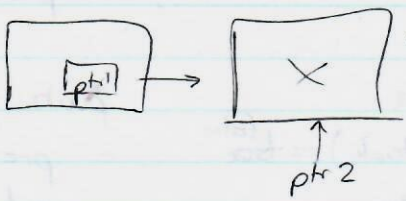


ptr holds address of a.
↳ To access: *ptr

ptr → nullptr ⇒ Highly recommended!
↳ Will contain random garbage otherwise.

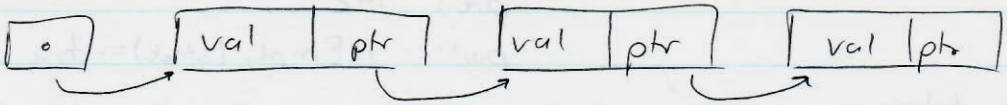
Important concept: set ptr → nullptr $\not\Rightarrow$ the object pointing to it is also null!

- Ex: //



⇐ Deleting it: setting ptr 2 to nullptr has NO EFFECT on ptr1 (garbage value)

Linked Lists



We will implement many data structures (abstract ⇒ ADT) using linked list.

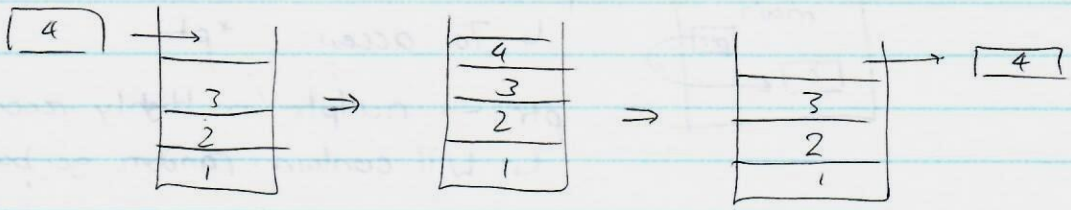
ADT: abstract data type

Data — Well-defined operations.

- Operation:
  1. Signature: params + return type
  2. Pre-condition: assumptions for op to work. Weak (def: true)
  3. Post-condition: describes effect of op. Strong
- Ex:
  - Stack: FIFO / LIFO
  - Array / vector: ordered, random access, add/delete @ end
  - Queue: FIFO
  - Set: unorded, contains elem at most 1
  - Dictionary / map: $(a,b) \overset{+}{=} (a,c)$ in M, then $b=c$

Stacks

LIFO / FILO



ADT:

initStack: (no input) → Stack
pre: true
post: new stack

isEmpty: Stack → boolean
pre: true
post: en == nullptr

pop: stack → stack
pre: isEmpty(Stack) == ~~true~~ false
post: enew-1 ... en

push: stack, val → stack
pre: true
post: enew+1 enew ... en

nuke: stack → stack
pre: true
post: isEmpty(stack) == true

Implementation:

```
struct Node {
    string val;
    Node * next;
};
typedef Node * Stack;

Stack initStack () {
    return nullptr;
}
bool isEmpty (Stack s) {
    return nullptr == s;
}
Stack push (Stack s, string val) {
    Node * newNode = new Node;
    newNode -> val = val;  newNode -> next = s
    return newNode
}
```

```
Stack pop (Stack s) {
    assert (isEmpty(s) == false);
    Node * newNode = s.next;
    delete s;
    return newNode;
}

stack nuke (Stack s) {
    while (!isEmpty(s)) {
        s = pop(s);
    }
    return s
}
```

Adapter:

Q: Why do we use LL for stacks? Why not <vector>?

Problem w/ vector: lots of diff. functionalities

↳ Give client bare-bones: prevent clients from dependence on non-essential features] → Adapter

Also: hide implementation details ⇒ well-defined API.

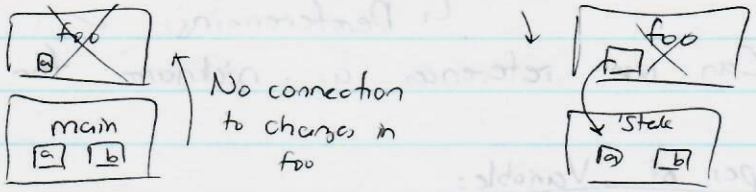
Reference Parameters

Passing into to functions

Pass by - value:
- Copying into
- Info changes do not persist.
  ↳ Ex:

Pass - by ptr:
- Sending in address
- Dereference ⇒ change
- Tricky
- Resist changes

Pass by reference
- Pass-by value but changes persist

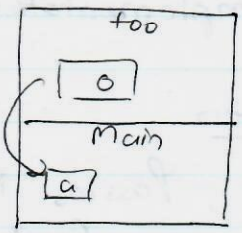


- Function declaration: void foo (int & variable)
  ↳ Variable 1 is an alias for the actual variable
- Calling: foo (a);
- Important note:
  The function gets the same type as you pass in.
  - Ex:

```
void top (Stack & s) {        supposed to be a ptr.
    cout << s-> val << endl;
}
```

```
Stack s;
Stack * ptr = & s;
top (ptr);
top(s);
```

- const: you cannot change reference
  - Func. declaration:
    void foo (const Stack & s) { ... }
  - More space efficient than a value param, preventing bad behav.
  - Ex: //
    void foo (Object * o) {...};
    1. Memory changes persist: NO
       ↳ Dereferencing: ✓. Pointing to something else: X



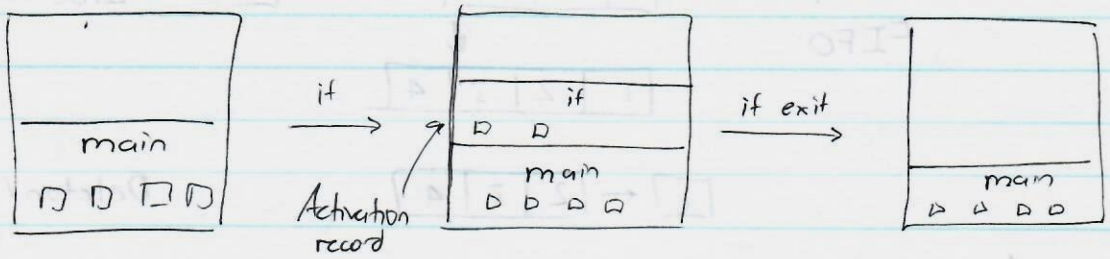
⇒ Creating new ptr on stack!

    void alpha (~~Object~~ const Object * o)
    Giving ptr to a const object.
    Memory changes persist: NO
    ↳ Dereferencing: X (cannot) - Pointing to something: X
    void beta (Object const * o)
    Constant ptr to object O.
    Memory changes persist: No
    ↳ Dereferencing: ✓. Pointing to smth: X (cannot)
- Can use references as nickname for variable

Types of Variables

Global	Local	Member/instance
◦ Declared outside of scope	◦ Defined in fn or scope ◦ Die when scope gone	◦ Part of larger struct/ class ◦ Lifecycle ∈ object lifecycle.

Scopes + Identifiers



Remember that changes propagated if:

1. Return value
2. References / ptr
3. Sub-scope of a bigger scope (if/for in the main)

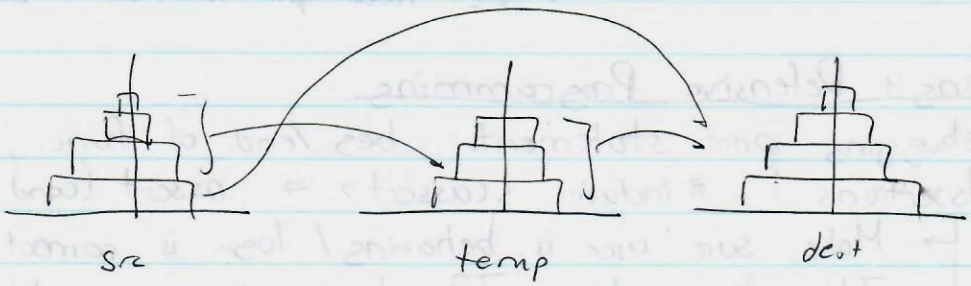
Recursion

Solve a bigger problem by dividing (induction)

3 parts:

1. Base case
2. Reduction operator: data smaller → base case
3. Composition operator: combining answer to smaller subproblems

Ex: // Towers of Hanoi

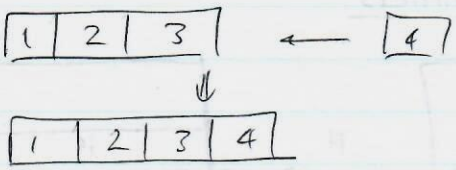


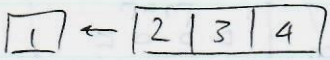

```
void hanoi (int n, int src, int dest, int temp) {
  if (N > 0) {
    hanoi (n-1, src, temp, dest);
    cout << " Move from: " << src << " to " << dest << endl;
    hanoi (n-1, temp, dest, src);
  }
}
```

Hilroy

Queue

FIFO:





Deletion / removal

ADT:

initQueue: no input → queue
- Pre: true
- Post: empty Queue

enter: queue → queue
- Pre: true
- Post: $e_1 \dots e_n \Rightarrow e_1 \dots e_n e_{new}$

leave: queue → queue
- Pre: !empty
- Post: $e_1 \dots e_n \rightarrow e_2 \dots e_{new}$

front: queue → value
- Pre: !empty
- Post: e_1

isEmpty: queue → bool
- Pre: true
- Post: nullptr == queue

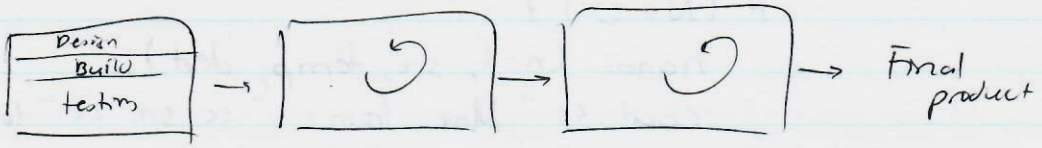
nuke: queue → queue
- Pre: true
- Post: empty == true.

Implementation:
- Vector: space inefficient
- LL: need ptr to front + back

Testing + Defensive Programming

- Debugging print statements: beg./end. of func., decision pts.
- Assertions: #include <cassert> ⇒ assert(cond.) ⇒ fails in cond. == false
  - ↳ Make sure user is behaving / logic is correct
  - ↳ T1: diagnostic, T2: longer-term assertions
- Agile:

Test-driven



4 month 4 month 4 month

Testing:
- Unit: individual components
- Integration: system

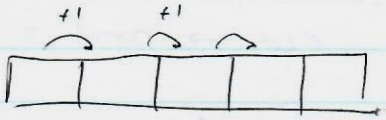
Testing

White
- Try to test all implementation.
- Hit all possible paths

Black ("no code")
- Test against what it supposed to do
- Based on requirements

- Regression: continue to add on test cases + run all test cases when new addition
- Usability: not code-oriented, user interact
- Non-functional requirement: security, robustness, speed.
- Static/dynamic: static ⇒ no exec. testing, dynamic ⇒ execution.

Dynamic Arrays

Old style: 

⇒ One continuous memory chunk.

Dynamic style: `int * A = new int [n]`

`delete [] A;`

To delete, we need the extent (# of memory chunks alloc.)
↳ Not accessible, but can be transferred by ptr.

Sorted Linked List

Sorted	Ordered
- Sorted based on value	- Sorted based off arrival

Implementation: like linked list but insertion based on value
↳ Need a ptr to figure out where to insert.

```
void insert (SortedList & first, string val) {
    Node * newNode = new Node;
    newNode -> val = val;                          // first is ptr to head
    if ( isEmpty (first) || val <= first -> val)   // To insert @ front
        newNode -> next = first,
        first = new Node;
    }else {
      Node * cur = first;
      while (nullptr != cur -> next && val > cur -> next -> val){
          cur = cur -> next;                       // Point to elem before
      }
      newNode -> next = cur -> next;               // Insertion.
      cur -> next = newNode;
    }
void remove (SortedList & first, string val) {
    Node * temp;
    if (first -> val == val) {
      temp = first;
      first = first -> next;
    } else {
        Node * cur = first;
        while (nullptr != cur -> next && cur -> next -> val < val)
            cur = cur -> next;
        )
        temp = cur -> next
        cur -> next = cur -> next -> next          // Skipping over.
    }
    delete temp;                                   // Removing ptr
}
```

Priority Queue

Sorted (based on priority) + ordered (FIFO)

Basically:

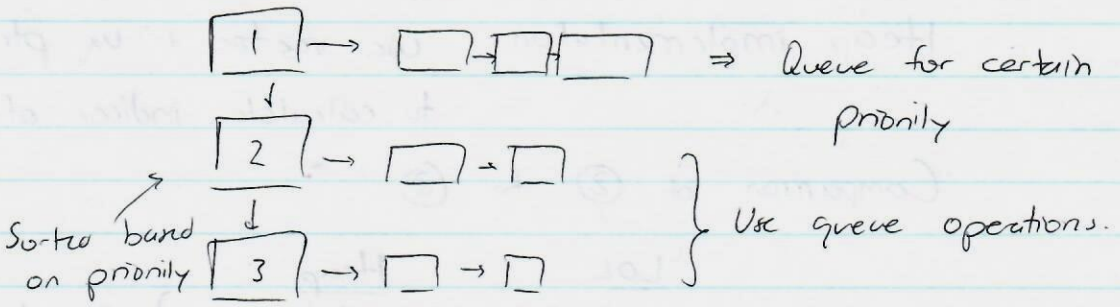
remove ⇒ Look for highest priority + remove oldest element (v. much like Queue)

Implementations:

①: Sorted list based on priority

↳ Complicated for leave operations.

②: List of list approach.



ADT:

void first_pq (const PQ & pq, string & val, int & priority)

Pre: !isEmpty (PQ)

Post: Return (value, priority) pair of first element

void leave_pq (PQ & pq):

Pre: !isEmpty (PQ)

Post: perform leave Q on first queue in PQ

↳ Remove first element / remove whole list.

void enter_pq (const PQ & pq, string val, int p)

Pre: true

Post: 1. Find if priority exists

↳ If not, create new list for priority

2. Insert val into list.

Complexity: O(k) (k = # of lists)

③: Heap (most common)

Heap property: val of parent >= all children.

Insertion in heap:

1. Insert value into next open slot
2. Fix heap property:
   ↳ Compare to parent + swap if child > parent + continue

$O(\log n)$

Deletion in heap:

1. Find value using heap property + delete
2. Fix heap property

$O(\log n)$

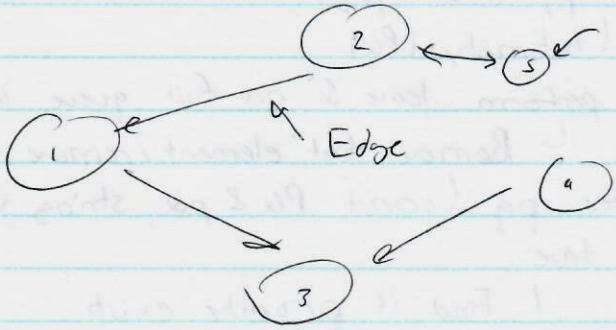
Heap implementation: use vector + use ptr math to calculate indices of children.

Comparison of ② to ③:

	LOL	Heap	
enter	$O(k)$	$O(\log n)$	If $k \ll n$, then LOL. Else, heap.
leave	$O(1)$	$O(\log n)$	
first	$O(1)$	$O(1)$	

GRAPHS, TREES, HASH TABLES

Graph



Math

$G = \{ V, E \}$

$V = \{ v_1, v_2, v_3, v_4, v_5 \}$

$E = \{ (v_2, v_1), (v_5, v_2), (v_2, v_5) (v_1, v_3), (v_4, v_3) \}$

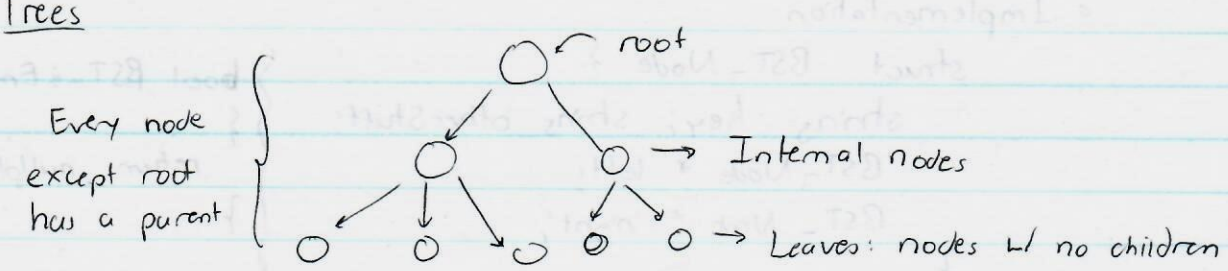
Directed graph: $(v_1, v_2) \neq (v_2, v_1)$

Undirected graph: direction of connection doesn't matter.

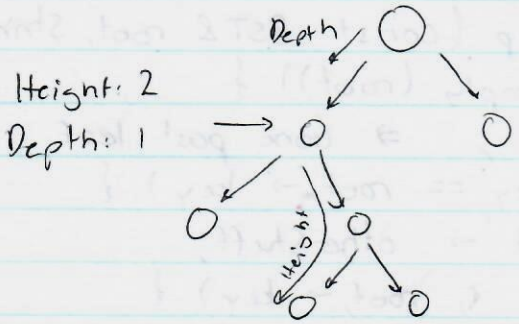
Ex: // Travelling salesman: given a graph, how can I travel the shortest possible route by visiting all nodes + returning to original.

- NP-hard! Believed to have a non-polynomial run time

Trees



Height vs. depth:



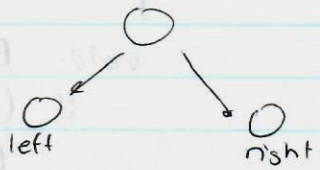
Height of node:
longest path to leaf

Depth of node:
length of root to ~~te~~ node

Binary tree: tree where each node has a max of 2 children.

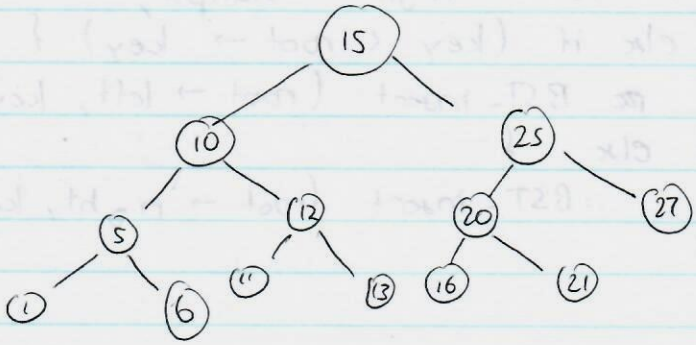
```
struct Node {
    string val;
    Node * left;
    Node * right;
}
```

Implementation:



Binary search tree: comparing keys.

- Property: keys of all left children < parent key < keys of all right children

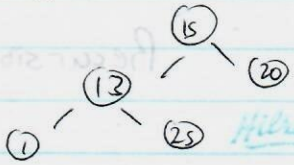


Not the same as

left → key < cur → key < right → key

↳ Only on one node.

- Ex:



Many variations in 1 dataset

- Implementation:

```
struct BST_Node {
    string key; string otherStuff;
    BST_Node * left;
    BST_Node * right;
};
typedef BST_Node * BST;
```

```
bool BST_isEmpty (BST & root)
{
    return nullptr == root;
}
```

```
string BST_lookup (const BST & root, string key) {
    if (BST_isEmpty (root)) {
        return "";          ⇒ Gone past leaf
    } else if (key == root -> key) {
        return root -> otherStuff;
    } else if (key < root -> key) {
        return BST_lookup (root -> left, key);
    } else if (key > root -> key) {
        return BST_lookup (root -> right, key);
    }
}
```

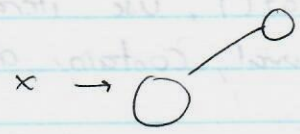
V. similar to binary search also

```
void BST_insert (BST & root, string key) {
    if (BST_isEmpty (root)) {      ⇒ We are at correct place!
        BST newNode = new BST_node;
        root -> key = key;
        root -> left = nullptr;
        root -> right = nullptr;
    } else if (key < root -> key) {
        BST_insert (root -> left, key);
    } else {
        BST_insert (root -> right, key);
    }
}
```

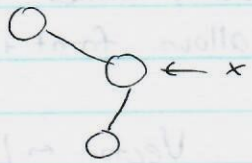
Oldest node is at root

Recursion is key to BST

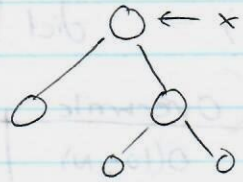
- BST removal 0 node, 1 child, 2 children are cases.



Make it = nullptr



Make parent point to child (change root ptr)



⇒ 1. Find min value in right subtree (keep going left)
2. Replace data w/ ~~left~~ min node
3. Delete min node

- Time complexity:

Print: $O(n)$ Insert/lookup: $O(\log N)$ or $O(h)$ ⇒ h is height
↳ If tree is not balanced, $h \to n$

Deletion: $O(h)$ (h is either $\log_2 N$ or N)

- General formulas (binary):

Height	# nodes at level	Cumulative # of nodes
0	1	1
1	2	3
2	4	7
3	8	15
k	2^k	$2^{k+1} - 1$
$\log_2 M$	M	$2M - 1$
$\log_2 N$	$\frac{N+1}{2}$	N

- Sequence ADT: similar array
  - Common operations: insertion (at specified position), append (@ end), at (element @ index), remove

	insert	append	at	remove
Array	$O(n)$	$O(1)$	$O(1)$	$O(N)$
vector	$O(n)$	$O(1)$	$O(1)$	$O(N)$
LL	$O(n)$	$O(N)$	$O(N)$	$O(N)$

- Alternatives to vector: list, deque.
  - List: doubly-linked list, not at(), use iterators for remove + insert
  - Deque: allows front + back removal, contains at()

Vector ↔ List ↔ Deque Choose what's best.

- Dictionary ADT: map/associative array
  (unsorted/sorted)

(key, value) ⇒ dict[key] = value

		add	overwrite	lookup	remove
Sorted	vector	O(N)	O(log N)	O(log N)	O(N)
Unsorted	vector	O(1)	O(N)	O(N)	O(N)
Sorted	LL	O(N)	O(N)	O(N)	O(N)
Unsorted	LL	O(1)	O(N)	O(N)	O(N)
Balanced	BST	O(log N)	O(log N)	O(log N)	O(log N)

Can use binary search

overwrite = lookup if insertion/data manip. not included

  - C++: map

```
#include <map>
:
map <~~string~~ key_dataType, value_dataType> m;
m[key_val] = val;
```

Mathematically:

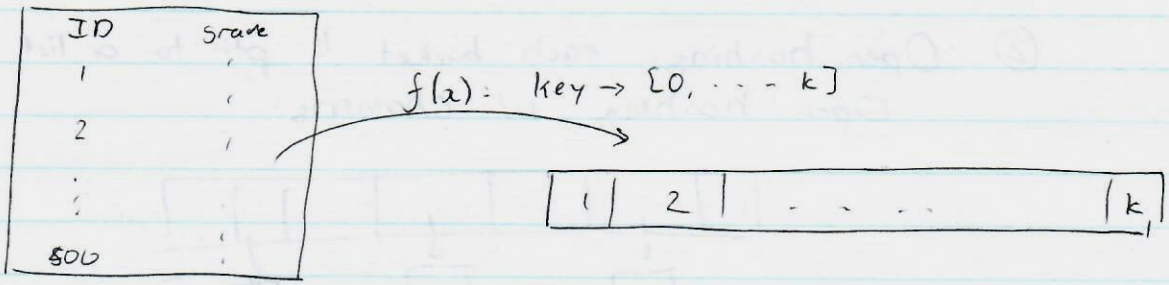
① : ∀x, y, z (keys):
  ① Anti-reflexive: $x \not< x$
  ② Anti-symmetric: $x < y \Rightarrow y \not< x$
  ③ Transitive: $x < y \wedge y < z \Rightarrow x < z$

② : Keys must be able to be compared!
  ↳ <, > symbol is defined for key data type.

Uses red-black trees ⇒ self-balancing B.T.
Implemented via sorted key on arrival
  ↳ unordered_map needed

Hashing

O(1) ⇒ adding, removing, lookup
↳ Unordered.



We can use hash function to find / lookup places in vector w/ O(1) ⇒ index!

Problem: collsions are frequent

①: Closed hashing: find next open bucket

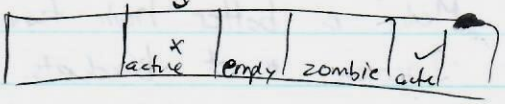
Linear probing:

Insert/init

1. Create vector of struct, k elements
   struct . . . {
   key, value, flag = {zombie, active, empty}
   }
   Zombie: element that was removed
   Active: element w/ smth. in it
   Empty: self-explanatory.
2. Init vector to all empty.
3. Use hash function to find key in vector. If empty, we insert
4. If not: find next empty/zombie slot.

Removal: zombie init when removed.

Lookup: f(x)



1. Is it active?
   ↳ Skip if not
2. Else, check key
   ↳ Keep moving if not.

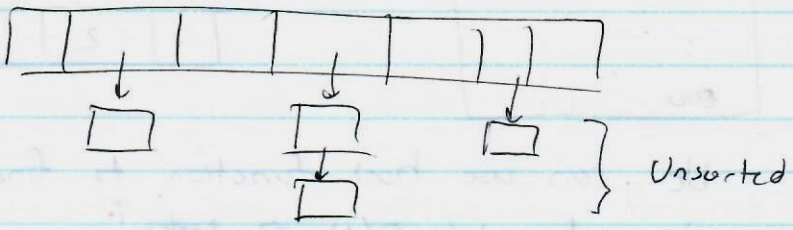
Problem w/ closed hashing:

1. Not dynamic + wasteful if mostly empty
2. Slightly slower

Works well if dealing w/ small problem.

② Open hashing: each bucket is ptr to a list

Open hashing w/ chaining:



Analysis:

Best: perfect hash function ⇒ O(1)

Worst: only maps to 1 place ⇒ O(N)

Hash table ← vector hash table ✓
← hash function
← collision strategy ✓

What makes a good hash fn?

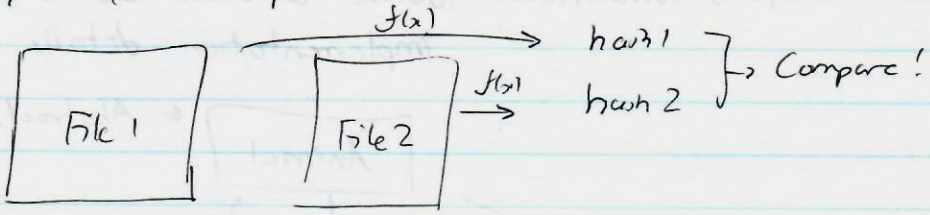
1. Deterministic: same index → for same id / input
2. Good spread
3. Cheap to compute
4. Supports variable range

Note:

1. Do not need unique keys ⇒ will have collisions.
2. Terrible for ranges / sorts
   ↳ key mapping is not related to key similarity
3. How to improve hash table performance.
   1. Make a better hash function
   2. Increase # of buckets.

Applications for hashing:

1. Fat, lossy comparison.



↳ Errors, plagiarism

2. Dictionary
3. Password:

Create a hash of pssw → cryptographically secure.
↳ RSA

Cryptographic hash:

1. Easy to compute
2. Given hash → reengineer pssw impossible
3. Cannot modify ~~message~~ pssw w/out changing hash
4. Collisions are rare.

OBJECT-ORIENTED PROGRAMMING

1. Procedural: procedures (functions) + variables + structures.
   ↳ Instances of struct w/ separate states

⇒ 2. Object-based: classes + instances

```
Class A {
    methods  →  functions, overloaded
    fields   →  variables
}
```

Overloaded function: same function name, dif. params.

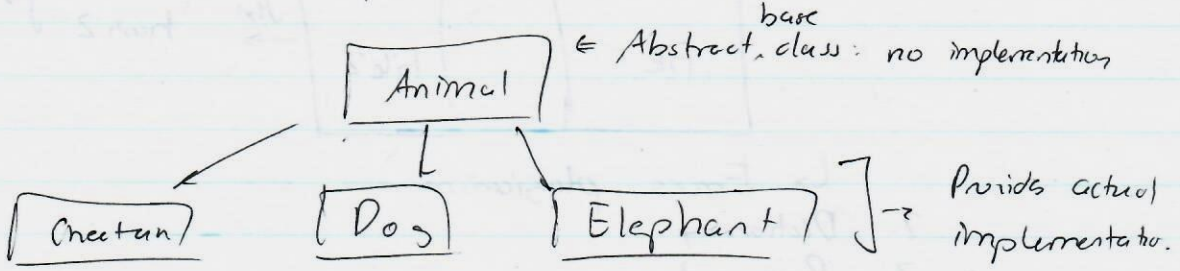
⇒ 3. Object-oriented: extending classes via inheritance

Overrided function: child class has dif. implementation of function.

Hilary

Features of OOP:

1. Abstractness: define a class as a template ⇒ no implementation details



2. Polymorphism: treating classes of ~~sim~~ similar descent in similar manner.

Declaration + Definition of Classes

Declaration:

```
class Balloon {
public:                          ← access type
    Balloon ();                  ← constructor
    Balloon (string colour);
    virtual ~Balloon ();         ← dtor
    void speak () const;
private:
    string colour                ← field
};
```

Methods

Definition:

```
type
className :: func_Name (...){
    ...
};

Balloon::Balloon (string colour) {
    this → colour = colour;
}
```

this → colour: colour in this → colour is part of class, not param.

Only use this → ... is param = class variable

.h → declarations ⟷ .cc → implementations

⟷: #include "...h"

Notes:

- Const after function: promise that no change happening

  ↳ `className :: funcType funcName () const { ... }`

- Difference between private + public:

```
class A {
    public:
        foo()      ] → Accessed by user
    private:
        alpha()    ] → Cannot be accessed by user.
                       Only functions within class instance
                       can access it.
```

A.foo() ✓ A.alpha() ✗ ⇒ private

↳ Work around: public function w/ access to private.

- Heap allocation: A.foo ⇒ A→foo();

Constructors

```
int main () {
    className obj{}; ⇒ Creates object
    obj{}; ⇒ Constructor: initializes + sets up object.
```

Class:

```
class A {
  public:
    ~~Bella~~ A(); ⇒ default constructor
    A(...); ⇒ parameterized constructor
```

If no default ctor ⇒ compile creates one for you
+ fields initialized by compiler

Note: Use { } for ctor in int main!

```
int main () {
    Balloon obj(); → function that returns Balloon (prototype)
    Balloon obj2{}; → Ctor
```

}

Initializers

```
Balloon :: Balloon (string colour) {
    this -> colour = colour;
}
```

Variable name is seen:
1. Local variable
2. Parameter ← Reason for this →
3. Field of class
4. Field on an ancestor
5. Global

```
Balloon :: Balloon (string colour) : colour {____} {};
```
↳ Initialize.

Template: ctor : fieldName { param / default } - .. { };
↳ Multiple inits:
: field1 { . . } : field2 { . . . } : field3 . . .

Order of initialization ← order of declaration

```
class Balloon {
  private:
    string colour, name;
  public:
    Balloon ();
};
```

```
Balloon :: Balloon() :
  name { "Bob" } : colour {
  "red" } { }
```
↳ colour initialized before name!

Can call constructor in a constructor! ↗

Es: //
Ballon :: Ballon :
→ Ballon(color){
constructor!

Design:

Principle of Least Astonishment (POLA): API to a class should perform as expected, minimal suprise.

Minimize amount of overloaded functions that do different implementations.

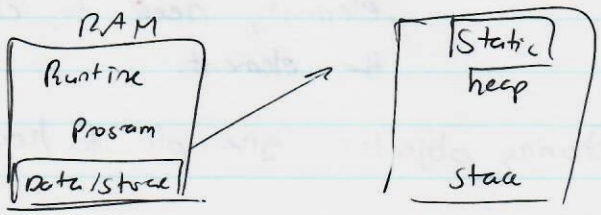
OOP

Member variables
- Instance variables (per obj)
- class/static variables (per class)

Member methods
- Instance method: (operates on obj)
- Class/static method
  - ↳ Cannot call instance methods
  - ↳ Can call static methods.

Static

Only creating variable/function once per program.



Scope: class defined in

static varType varName; (same thing for function)

Copy Constructors

Copying function that takes const ref. to another obj + copies for new obj.

Declaration:

```
C:: C (const C &obj);
```

Default copy ⇒ element-wise copy.

Recognize that this is ALSO a copy constructor:

```
className obj1 = obj2;
```

Note: be careful if copying objects w/ ptr to another object ⇒ copy ptr but not create a new object!

DESTRUCTORS

```
class Balloon {
public:
  ~Balloon();
  :
```

Destructor responsibility:

① : Remove all heap allocated elements (you have to code)

② : Cleans up object.

Destructors called if object exits scope / ~~delete~~ delete heap allocated-objected.

Note of caution: if multi objects point to 1 heap-allocated element, need to consider who is deleting the element.

→ Sharing objects: give ptr to heap-allocated element

ACCESS + MUTATORS

Methods → Access: reports value back (const)

Methods → Mutators: changes value in class

PUBLIC, PRIVATE, PROTECTED

```
class A {
private:
  :
protected:
  :
public:
  :
};
```

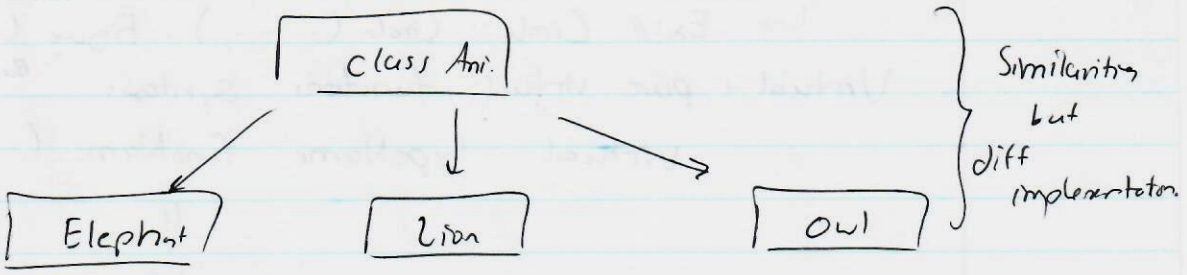
private: → Only class itself can access

Visible to descendants!

protected: → The class + any descendants can access

public: → Anyone can access

INHERITANCE



Solve problem of similar class but diff. implementation.

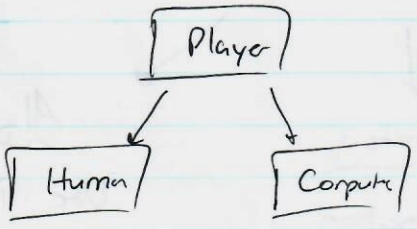
①: Define an abstract class (abstract base class, ABC):
Methods / fields that all children must have.
Never create an instance of this class.

②: Define concrete classes that inherit ABC:
Overriding / provide implementation details
↳ Virtual / pure virtual: methods that had no implementation details in parent

Polymorphism: treat similar objects the same

A ptr to a base class: can point to an instance of the base class / descendants

Ex:




```
vector <Player *> players;
players.push_back(new Human("..."));
                  (new Computer("..."));
for (auto player : players) {
    player.playTurn(); ← Poly.!
```

Ctor + dtor: not inherited but automatically invoked
↳ Ex: // Circle::Circle(...) : Figure(...), ... {}
Base class

Virtual + pure virtual functions syntax:

```
virtual typeName funcName(...);
                                = 0;  ⇐ Pure!
```

↳ If its virtual ⇒ provide default implementation.
↳ Specify override in implementation.

```
class Circle {
public:
    virtual void draw() const override;
```

Don't need in defining function.

How do you actually do inheritance:

```
class A: accessType parentClass {...};
```

Access type: public, private + protected
↳ For ∀ inherited stuff, in which access should it go under?

Virtual functions can go in private + get redefined!
↳ No implementation details ⇒ can be overriden!

C++ TEMPLATE LIBRARY

Template Library

Containers:
Takes element type + creates container of type

Iterators:
Navigation of containers

Algorithms
Use iterators for operation.

ITERATORS

Problem: container is opaque + hard to understand but we need to iterate through it

Solution: iterators

Container provides iterator ⇒ operations.

```
container <type> :: iterator i;
i = containerVarName.begin();
i++
End = containerVarName.end();
```

} Think of iterator as ptr

Different containers have different iterator rules

- Vector: forward, backward + random access it.
- List: forward, backward it
- set: forward it

Auto:

for (vector<string> :: iterator it = ...)

instead

→ for (auto i = v.begin() : . .)

↳ auto tells compiler that i is same type as v.begin() return. Still a ptr

→ for (auto i : containerVarName)

↳ i is a <u>copy</u> of a particular element

Reference:

for (auto &i : containerVarName)

Types of iterators:

const iterators: promise that no changes happening
↳ cend(); cbegin();

bidirectional: supports ++, --

Random access: bidirectional w/ O(ACT)
↳ If vi is a random access iterator,
vi[3] gets 3rd element after vi
in O(ACT)

ALGORITHMS

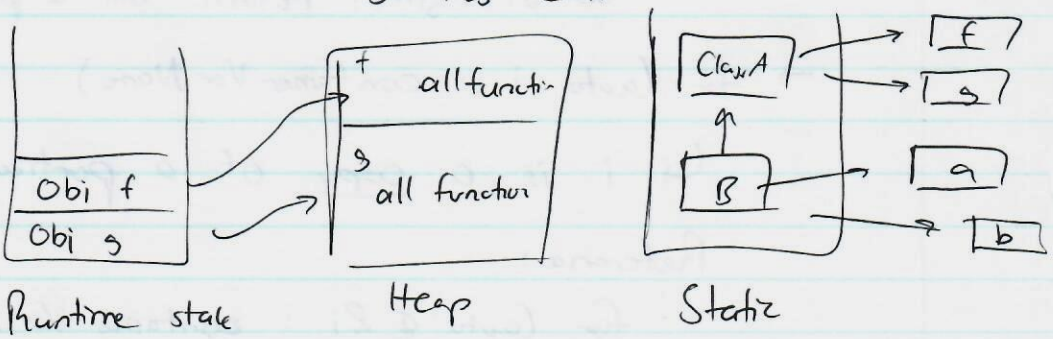
Take iterators of container (begin + end) ⇒ operation
↳ f(iter$_1$, iter$_2$, arg$_1$, arg$_2$ ...)

#include <algorithm>

- find
- count
- for_each
- remove
- replace
- sort
- unique
- min/max_element
- nth_element
- random_shuffle
- next_permutation

VTABLE

Problem: When calling a polymorphic function, which function is getting call.

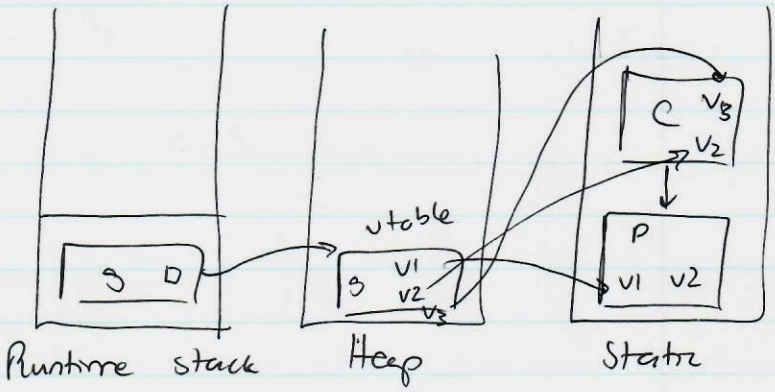


```
class P {
  public :
    virtual void v1();
    virtual void v2();
};
P:: void v1() {
  cout << " v1 " << endl;
  v2();
}
P:: void v2() {
  cout << " v2 " << endl;
}
```

```
class C : public P {
  public:
    virtual void v3();
    virtual void v2();
};
C:: void v3() {
  cout << " v3 " << endl;
}
C:: void v2() {
  cout << " v2 new " << endl;
}
```

```
int main() {
  P *p = new C;
  p -> v1();
}
```

DIAGRAM:



1. p → v1()
   ↳ Check vtable + looks where v1 is defined → v1 is defind in P class

2. v1 calls v2:
   ↳ Check vtable + look where v2 is point
   ↳ cout << C::v2() << endl;

To create vtable: look @ current class. If not defined, move up