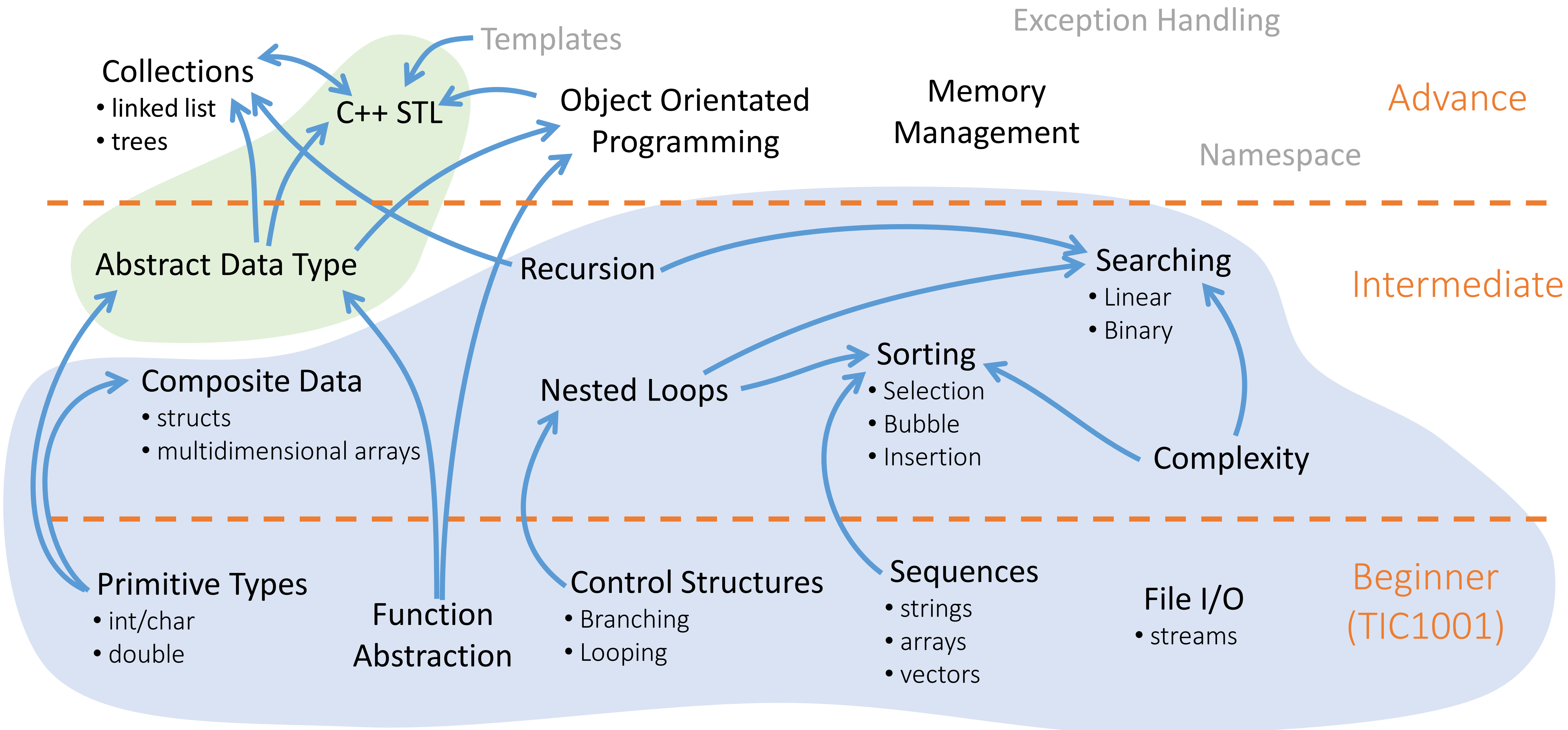
Lecture 6
Abstract Data Types

TIC1002 Introduction to Computing and Programming II

TIC1001/2 Roadmap



Course Schedule (Tentative)

Week	Topic(s)		
7	Midterm Test		
8	Abstract Data Type & C++ STL		
9	Working with Collections	Problem Set 3	
10	Object Oriented Programming		Problem Set 4
11	Inheritance, Polymorphism		
12	Revision		
13	Revision	Practical Exam 2	
Reading			
Exam	Final Exam		

Insofar...

we have only been dealing with simple data

- Numbers
- Arrays
- Structures

However

Life is complicated

- so is real life data

Example

NUS Registrar has a record of every student

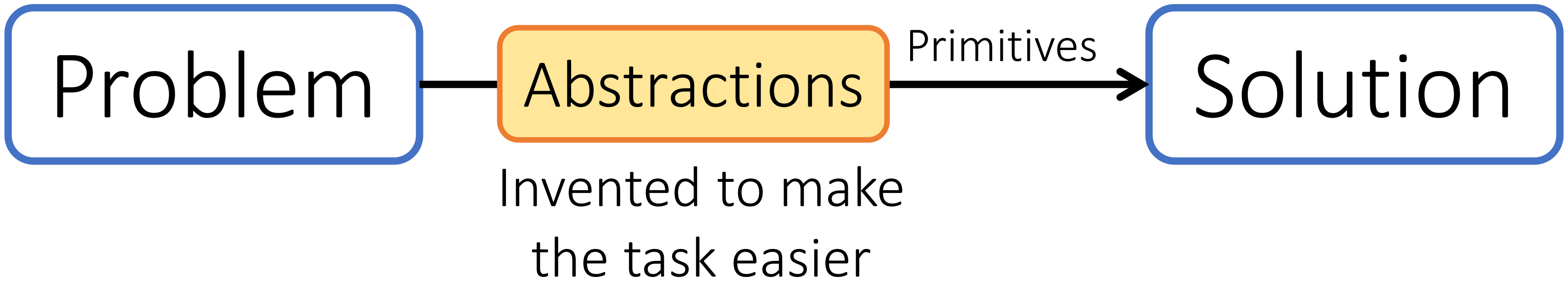
- Personal info, modules taken, grades, etc.
- Record may be a paper folder or electronic document
- Record is a compound data (struct)

However,

- How do we access the fields in the struct?
- Would changing one field affect another?
- We need to know how it is defined

Wouldn’t it be great if we can abstract away the details?

Recall: Function Abstractions



– Only need to know how a function transforms inputs to an output
– Don't need to know how it is implemented

E.g. $\sin(x)$

– Don’t need to know how the function works

Recall: Function Abstractions

- Abstracts away irrelevant details, exposes what is necessary
- Separates usage from implementation
- Captures common programming patterns
- Serves as a building block for more complex functions

Key Idea

Data can be organized and reasoned the same way

Function Abstraction vs Data Abstraction

Function Abstraction

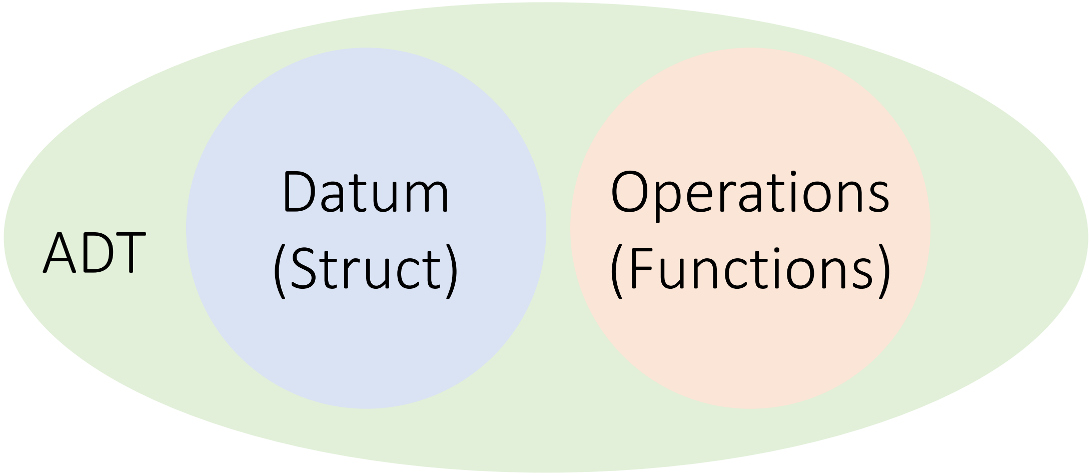
- Abstracts away irrelevant details, exposes what is necessary
- Separates usage from implementation
- Captures common programming patterns
- Serves as a building block for more complex functions

Data Abstraction

- Abstracts away irrelevant details, exposes what is necessary
- Separates usage from implementation
- Captures common programming patterns
- Serves as a building block for other compound data

Abstract Data Type

ADT = Datum + Operations



Specification

- What operations are supported

Implementation

- How data and code is written to meet the specification

Guidelines for ADT

Constructors

- Functions to create a compound data (ADT)

Accessors/Getters

- Functions to access individual components of the ADT

Mutators/Setters

- Functions to modify/mutate the components of the ADT

Predicates

- Functions to ask true/false questions on the ADT

Printers

- To display ADT in human-readable form

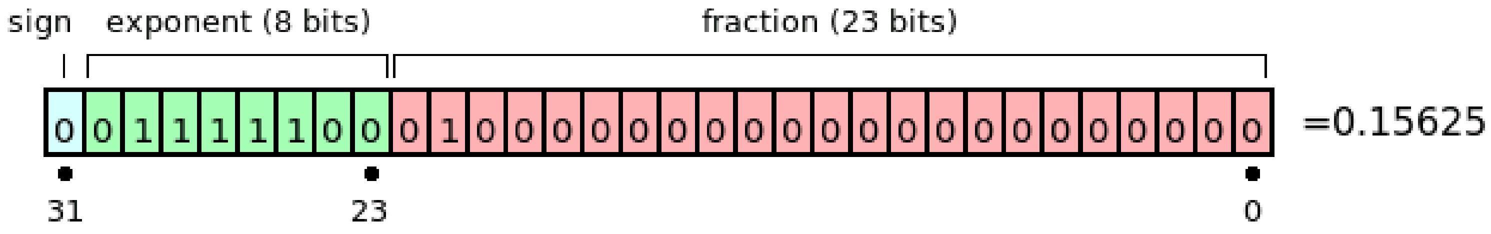
Case Study 1

Primitive Types as ADT

`int, float, double, char, bool`

Internally implemented as bytes

– float is implemented as



– As a user, you do not need to know the implementation to use float types

Case Study 2: Points

Points in a plane (graph/grid)

– Cartesian coordinates: x and y

Wishful thinking: Assume given a Point ADT

– Constructor

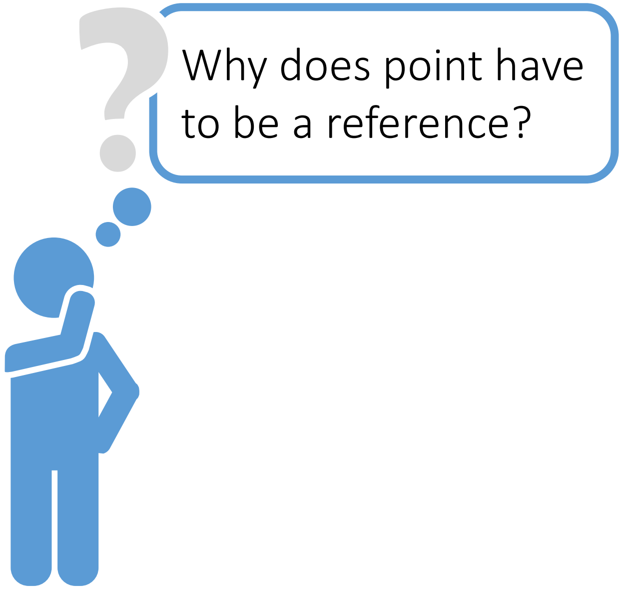
```
void make_point(Point &point, double x, double y);
```

– Accessors:

```
double x_of(const Point &point);
double y_of(const Point &point);
```

– Mutators:

```
void set_x(Point &point, double value);
void set_y(Point &point, double value);
```



Creating new operations

Translation

– Move the point to a new position

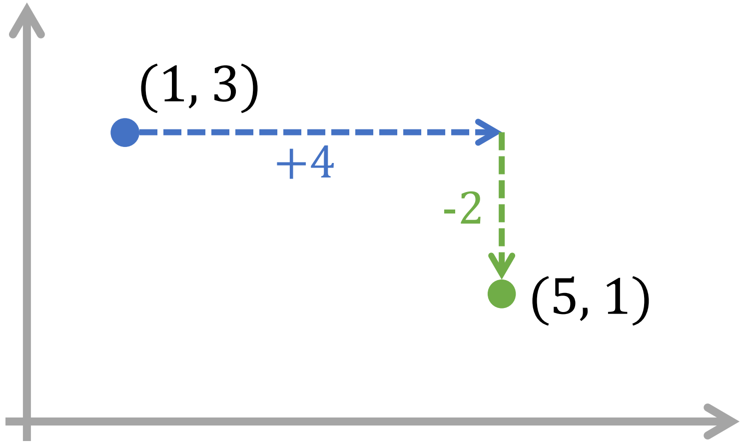
Rotation

– Rotate the point about the origin

Translating a Point

Move x and y by some value

- $(x', y') = (x + \delta_x, y + \delta_y)$

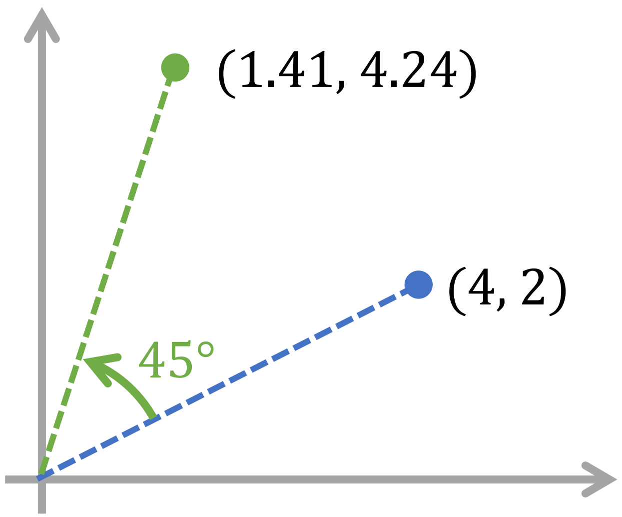



```
void translate_point(Point &p, double dx, double dy) {
    set_x(p, x_of(p) + dx);
    set_y(p, y_of(p) + dy);
}
```

Rotating a Point

Rotate around the origin (0, 0) by θ

– $(x', y') = (x\cos\theta - y\sin\theta,\ y\cos\theta + x\sin\theta)$




```
#include <math.h>
void rotate_point(Point &p, double angle) {
    double c = cos(angle), s = sin(angle),
           x = x_of(p)    , y = y_of(p);
    set_x(p, x*c - y*s);
    set_y(p, y*c + x*s);
}
```

Displaying Points

Output to a stream

```
void display_point(ostream &out, const Point &p) {
    out << "(" << x_of(p) << "," << y_of(p) << ")" << endl;
}
```

Using Points ADT

```
int main() {
  Point p;
  make_point(p, 1, 3);
  display_point(cout, p);

  translate_point(p, 4, -2);
  display_point(cout, p);

  rotate_point(p, M_PI/4);
  display_point(cout, p);
  return 0;
}
```

Recall our assumption

the existence of

`make_point, x_of, y_of, set_x, set_y`

From which we defined our new operations

`translate_point, rotate_point`

Power of Abstraction (A^x)

We still do not know how a Point is implemented

– But we can still make use of them

But… what’s the point?

Why do we need constructors, accessors, mutators and predicates?

1. Hide (abstract away) complicated logic
2. Hide implementation

Why do we want the hide implementation?

– Suppose we do this

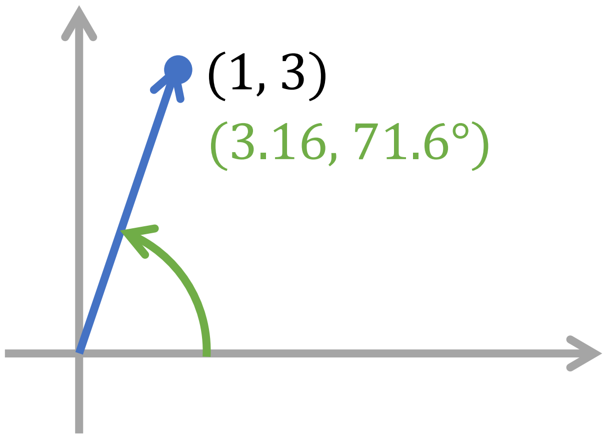
```
void translate_point(Point &p, double dx, double dy) {
    p.x += dx;
    p.y += dy;
}
```

Shouldn’t this still work?

New Implementation

Why should Points be represented as (x, y) coordinates?

- Can also use polar representation
- A point is represented as
  1. Distance from origin
  2. Angle from origin




```
struct Point { double distance, angle; };

void make_point(Point &p, double x, double y) {
    p.distance = hypot(x, y);
    p.angle = atan(y/x);
}
```

What happens to our function?

```
void translate_point(Point &p, double dx, double dy) {
    p.x += dx;
    p.y += dy;
}
```

Compile error!

- Point doesn’t even have x and y fields
- Contents no longer represent the same

This is known as breaking abstraction

- When you make certain assumptions about the organization of the ADT
- Your code is liable to break if implementation changes

Why change implementation?

Suppose new accessors are given

```
double dist_of(const Point &p);
double angle_of(const Point &p);
```

Along with new mutators

```
void set_dist(Point &p, double value);
void set_angle(Point &p, double angle);
```

– We do not have to care how Point is implemented

```
void rotate_point(Point p, double angle) {
    set_angle(p, angle+angle_of(p));
}
```

Common ADTs

And C++ Standard Template Library

STL Vector

http://en.cppreference.com/w/cpp/container/vector

Header file

```
#include <vector>
```

Defined as template class

```
vector<int> int_vector;
```

Stores contiguous elements (a sequence) like an array

Advantages

- Fast insertion and removal at end of vector
- Dynamic sizing
- Automatic memory management
- The simplest STL container class, and in many cases the most efficient

STL Vector: Common Methods

Constructor	
`vector<T> v`	Construct a vector v to store elements of type T
Accessors	
`at(n)` or `[n]`	returns an element at position n
`front()`	returns a reference to the first element
`back()`	returns a reference to the last element
`size()`	returns the number of items
Predicates	
`empty()`	returns true if the vector has no elements
Mutators	
`clear()`	removes all elements
`pop_back()`	removes the last element
`push_back(e)`	add element e to the end

Template?

A vector is a container

- But what does it contain?
- Needs to be a specified type

Template allows us to declare what type it should contain

```
vector<int> int_vector;  // vector of int
vector<Points> pts_vector;   // vector of Points
```

- Type is specified in the < > brackets

Example: Vector of Points

```
vector<Point> points;
Point p;
make_point(p, 0, 0);
for (int i = 0; i < 5; i++) {
    translate_point(p, i, i);
    points.push_back(p);
}

for (int i = 0; i < points.size(); i++) {
    display_point(cout, points[i]);
    cout << endl;
}
```

Point p is reused by translating and then pushed into the vector points. How come the points in the vector did not change?

Stack ADT

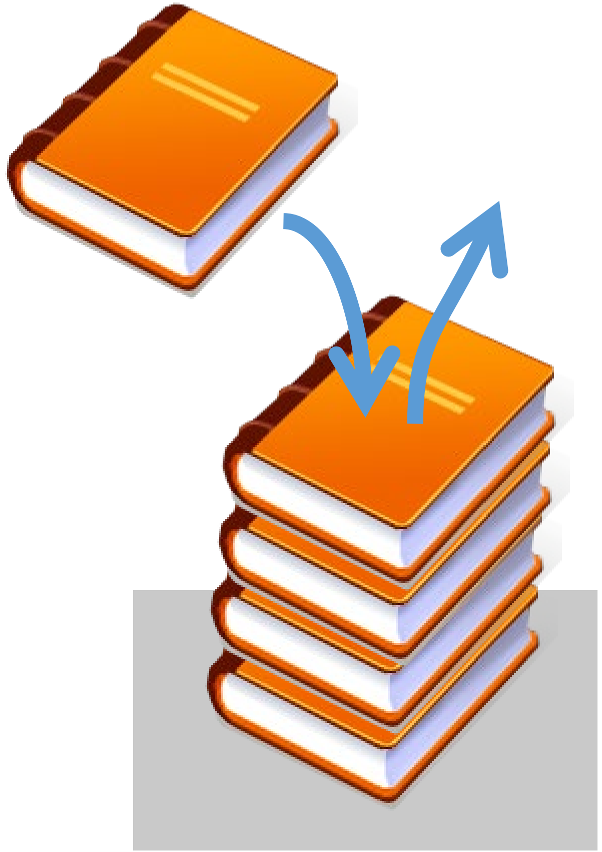
What is a Stack?

A specialized list

– Allows access only from one position

Example: Stack of books

– Can only add and remove from the top
– Can only see the top book



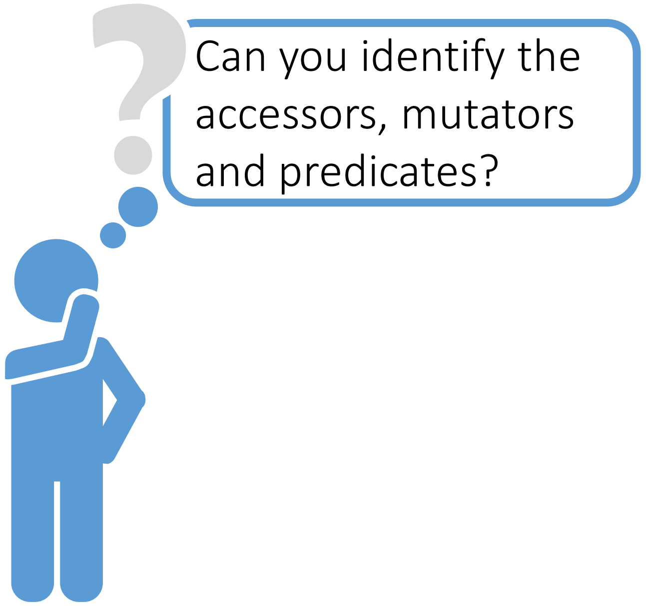
LIFO access of data

– Last-in, first-out order

Stack Operations

List of supported operations

- push: insert element into top of stack
- pop: remove and return top element of stack
- peek: return top element of stack
- empty: returns true if stack is empty, false otherwise



Example: Using a stack

```
Stack<int> s;
push(s, 1);
push(s, 2);
push(s, 3);

cout << (empty(s) ? "true" : "false") << endl;
cout << peek(s) << endl;

cout << pop(s) << endl;
cout << pop(s) << endl;
cout << pop(s) << endl;
cout << (empty(s) ? "true" : "false") << endl;
```

Our Homemade Stack ADT

Using a vector internally

```
template <typename T>
struct Stack {
    vector<T> v;
};

template <typename T>
void push(Stack<T> &s, T value) {
    s.v.push_back(value);
}
```

Our Homemade Stack ADT


```
template <typename T>
T pop(Stack<T> &s) {
    T value = s.v.back();
    s.v.pop_back();
    return value;
}

template <typename T>
T peek(const Stack<T> s) { return s.v.back(); }

template <typename T>
bool empty(const Stack<T> s) { return s.v.empty(); }
```

Performance of our Stack

Simple the performance of vector

- push_back: O(1) since it just adds the element at the back
- pop_back: O(1) since it just removes the element at the back
- empty: O(1) since vector maintains a count of contents

O(1) for everything! It's very good!

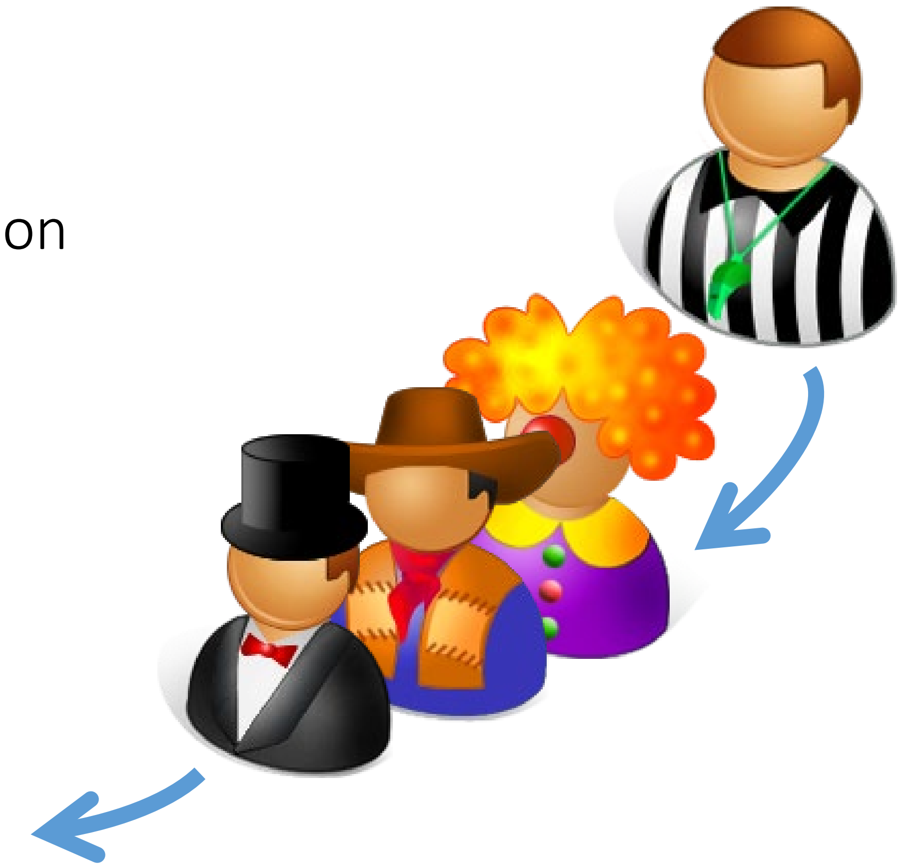
Queue

What is a Queue?

What is a Queue?

A specialized list

- Allow access only from specific position
- Add to the tail
- Remove from the head



FIFO access of data

- First-in, First-out

Queue Operations

List of supported operations

- enqueue: inserts element into tail of queue
- dequeue: removes element from head of queue and returns it
- head: returns the element at head of queue
- empty: returns true if queue is empty, false otherwise

Example: Using a queue

```
Queue<int> q;
enqueue(q, 1);
enqueue(q, 2);
enqueue(q, 3);

cout << (empty(q) ? "true" : "false") << endl;
cout << head(q) << endl;

cout << dequeue(q) << endl;
cout << dequeue(q) << endl;
cout << dequeue(q) << endl;
cout << (empty(a) ? "true" : "false") << endl;
```

Our Homemade Queue ADT

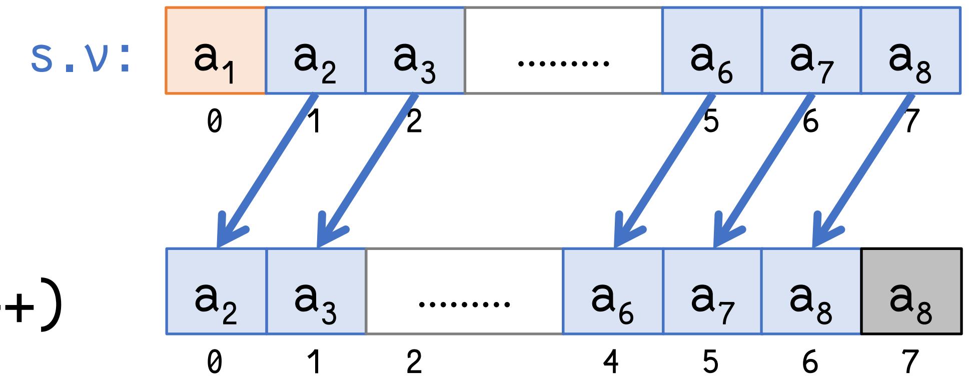
Using a vector internally

```
template <typename T>
struct Queue {
    vector<T> v;
};

template <typename T>
void enqueue(Queue<T> &q, T value) {
    q.v.push_back(value);
}
```

Our Homemade Stack ADT

```
template <typename T>
T dequeue(Stack<T> &s) {
    T value = s.v.front();
    for (int i = 0; i < s.v.size()-1; i++)
        s.v[i] = s.v.[i+1];
    s.v.pop_back();
    return value;
}
```




```
template <typename T>
T peek(const Stack<T> s) { return s.v.front(); }
```

Performance of our Queue

enqueue

- (synonymous with push_back): O(1) since it just adds the element at the back

dequeue

- Need to copy every element to the front, then delete the element at the back
- O(n), where n is the size of the queue

Oh no, this can't be good

	Stack	Queue
Inserting	$O(1)$	$O(1)$
Removing	$O(1)$	$O(n)$

Let’s optimize

Since stack is so good, let’s use stacks to model queue

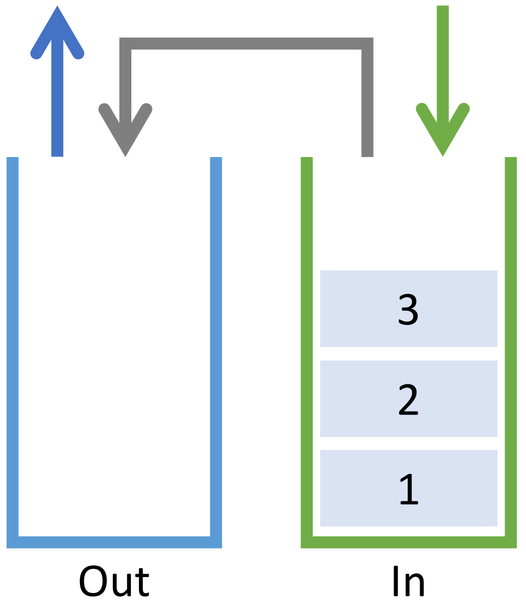
– Use two stacks: One for insertion, one for removal

Enqueued items are pushed to “In” stack

– O(1) constant time

Dequeued items are popped from “Out” stack

– O(1) constant time



Let's optimize

Since stack is so good, let's use stacks to model queue

- Use two stacks: One for insertion, one for removal

Enqueued items are pushed to "In" stack

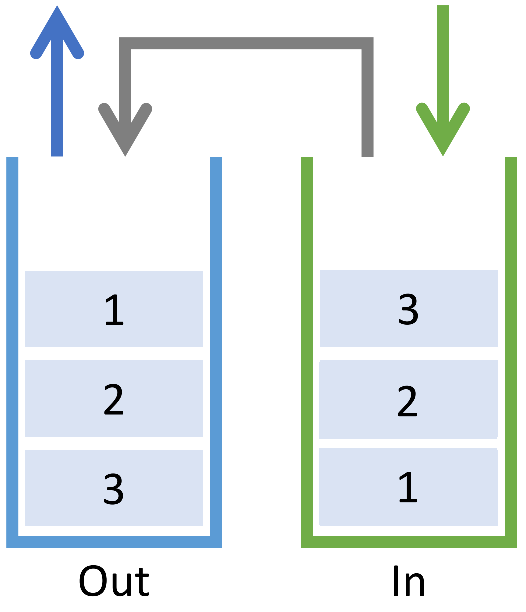
- O(1) constant time

Dequeued items are popped from "Out" stack

- O(1) constant time

When "Out" stack is empty

- pop all contents from "In" stack to "Out"
- Items will be nicely reversed and in correct order
- O(n) operation, but only once in a while



Implementation

```
template <typename T>
struct Queue {
    Stack<T> in, out;
};

template <typename T>
void enqueue(Queue<T> q, T value) {
    q.in.push(value);
}
```

Implementation

```
template <typename T>
T dequeue(Queue<T> q) {
    if (empty(q.out))
        // begin the transfer
        while (!empty(q.in))
            push(q.out, (pop(q.in));
    return pop(q.out);
}

template <typename T>
bool empty(Queue<T> q) { return empty(q.in) and empty(q.out); }
```

Moral of the Story

Performance

	Stack	Queue
Inserting	$O(1)$	$O(1)$
Removing	$O(1)$	Mostly $O(1)$, Sometimes $O(n)$

Implementation matters

– Increase efficiency and/or performance

Other Common ADTs

So far we have been dealing with sequence type ADTs

- Array
- Vector
- Stack
- Queue

Let's examine some non-sequence type ADTs

- Map
- Set

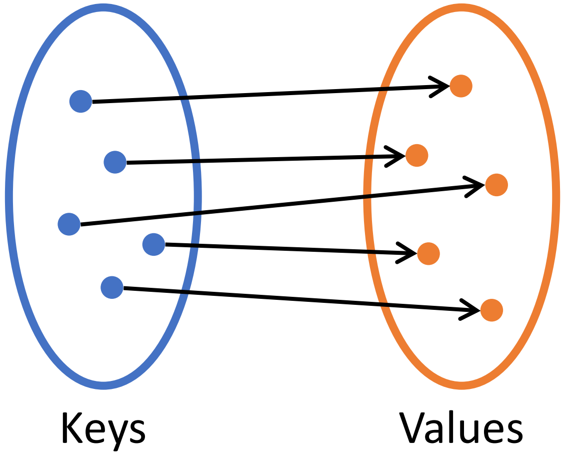
Map/Dictionary

Not a diagram, but a mapping

- Each element is a (key → value) pair

Kind of like a table

- Keys are unique
- Accessing each key is O(1)



Keys	Values

STL Map

http://en.cppreference.com/w/cpp/container/map

Header file

```
#include <map>
```

Defined as template class

```
map<int, string> int_map;
```

Stores sorted key-value pairs

- An associative container with unique keys mapping to values.

Advantages

- Fast insertion and removal (stored as binary tree)
- Dynamic sizing
- Automatic memory management

STL Map: Common Methods

`map<K, T> m`	Construct a map m to store elements of type K associated with type T
`size()`	returns the number of items
`empty()`	returns true if the map has no elements
`clear()`	removes all elements
`at(key)` or `[key]`	returns value T associated with key Key
`[key] = v`	inserts or updates key with value v
`erase(key)`	removes the element of key, and returns the number of elements removed
`count(key)`	Count the number of elements of key

Using STL Map

Header

```
#include <map>
using namespace std;
```

Create a new map

```
map<string, int> students; // creates a new map
```

Insert new entries

```
students["Ben"] = 90;       // this inserts a new entry
students["Cathy"] = 67;     // insert another entry
```

Using STL Map

Modifying entries

```
students["Ben"] = 98;  // Replaces old value if it exists
```

Number of elements

```
cout << students.size();  // returns 2
```

Accessing elements

```
cout << students["Ben"];  // returns 98
```

Erasing elements

```
students.erase("Ben")
```

Using STL Map

Total number of elements

```
cout << students.size()
```

Finding if key exists

```
cout << students.count("Ben");    // returns 0
cout << students.count("Cathy");  // returns 1
```

How to obtain all the keys (and hence values) of a map?

- Cannot iterate using index like a array/vector
- Need to use STL Iterator, or fancy new `for` loop

What's in a Map?

But first… what is exactly stored in a map?

– It is a `std::pair` ADT

```
struct pair {
    T1 first;
    T2 second;
};
```

Iterating through a Map

Fancy for-loop construct (C++11 onwards)

```
for (pair<string, int> element : students) {
    string name = element.first;  // access the KEY
    int marks = element.second;   // access the VALUE
    cout << name << " obtained " << marks << endl;
}
```

Creating a vector of keys

```
vector<string> keys;
for (pair<string, int> element : students) {
    keys.push_back(element.first);
}
```

Set

- A set contains elements, like a vector, except
  - There is no ordering (hence no index)
  - No duplicate elements are allowed
- Basically like a mathematical set

STL Set

http://en.cppreference.com/w/cpp/container/set

Header file

```
#include <set>
```

Defined as template class

```
set<int> int_set;
```

Stores a sorted set of elements

- i.e. OO implementation of set

Advantages

- Fast insertion and removal (stored as binary tree)
- Dynamic sizing
- Automatic memory management

STL Set: Common Methods

set<T> s	Construct a set s to store elements of type T
size()	returns the number of items
empty()	returns true if the set has no elements
clear()	removes all elements
insert(e)	adds element e to the set
erase(e)	removes element e from the set
count(e)	returns the number of elements e in the set (either 0 or 1)

Using STL Set (similar to Map)

Header

```
#include <set>
using namespace std;
```

Create a new set

```
set<int> matric; // creates a new set of integers
```

Insert new entries

```
matric.insert(1);
matric.insert(2);
matric.insert(3);
matric.insert(2);  // what happens when inserting duplicate?
```

Using STL Set

Total number of elements

```
cout << matric.size();  // returns 3 (Duplicates are not added)
```

Erasing elements

```
matric.erase(1);
matric.erase(2);
```

Finding element in set

```
cout << matric.count(3);  // returns 1
cout << matric.count(2);  // returns 0
```

Obtaining all the elements in set?

Iterating through a Set

Fancy for-loop construct

```
for (int element : matric) {
    cout << element << endl;
}
```

Copy set into a vector

```
vector<int> matric_v;
for (int element : matric) {
    matric_v.push_back(element);
}
```

Summary

Abstract Data Type

- Make use of data abstraction to hide implementation
- Make use of function abstraction to access the data

Why?

- Decouple implementation from usage

Common ADT

- Vector, Stack, Queue, Map, Set
- Template type allows ADT to contain arbitrary type of data
- Available in C++ STL. Why reinvent the wheel?