1/6: Introduction

- Start early, develop incrementally
- `scp Desktop/Archive.zip charlesx@cs32.seas.ucla.edu:`
- `ssh charlesx@cs32.seas.ucla.edu`
- `unzip -j -a -d hw5 Archive.zip`
- `cd hw3`
- `ls`
- `g32 -o test *.cpp`

- **#include <iostream>**
- **#include <cstdlib>**
- **#include <cmath>**
- **using namespace std;**
-
- **const double PI = 4 * atan(1.0);**
-
- **class Circle {**
    - **public:**
        - **Circle(double x, double y, double r);**
        - **void scale(double factor);**
        - **void draw() const;**
        - **double radius() const;**
    - **private:**
        - // Class Invariants: m_r > 0
        - **double m_x;**
        - **double m_y;**
        - **double m_r;**
- **};**
-
- **double area(const Circle& x);**
-
- **Circle::Circle(double x, double y, double r)**
- **: m_x(x), m_y(y), m_r(r)) {** // member initialization list
    - **if (r <= 0) {**
        - **cout << “Cannot create circle with radius “ << r << endl;**
        - **exit(1);**
    - **}**
- **}**
-
- **bool Circle::scale(double factor) {**
    - **if (factor <= 0) {**
        - **return false;**
    - **}**

    - **m_r *= factor;**
    - **return true;**
- **}**
-
- **void Circle::draw() const {**
    - **… code to draw the circle ...**
- **}**
-
- **double Circle::radius() const {**
    - **return m_r;**
- **}**
-
- **double area(const Circle& x) {**
    - **return PI * x.radius() * x.radius();**
- **}**
-
- **int main() {**
    - **Circle blah(8, -3, 3.7);**
    - **Circle c(2, -5, 10);**
    - **c.scale(2);**
    - **c.draw();**
    - **cout << area(c) << endl;**
    - **cout << c.radius() << endl;**
    -
    - **double ff;**
    - **cin >> ff;**
    - **if (!c.scale(ff)) {**
        - **… error …**
    - **}**
- **}** // guaranteed to return 0 if no number specified

- Interface (what we can do) vs. Implementation (how do we do it)

<u>1/8: Linking</u>

- myapp.cpp:
    - **#include <iostream>**
    - **using namespace std;**
    - **void f(double x);** // needed to declare
    - **int main() {**
        - **double a = 3;**
        - **f(a);**
        - **cout << a;**
    - **}** // defines main
    - // needs f, cout, **<<**
- stuff.cpp
    - **#include <cmath>**
    - **using namespace std;**
    - **void f(double x) {**
        - **…**
        - **sin(x)**
        - **…**
    - **}** // defines f
    - // needs sin
- Libraries - collections of object files with code for library functions
    - Defines sin, cos, exponential, cout, cin, **>>**, **<<**, etc
    - Needs nothing
- Object files combine with libraries to create an executable file
    - Object file → executable is called the linker
- Nothing can be defined more than once → linker error
- Every need must be satisfied by some definition
- There must be exactly one main routine
- Convention - 2 files per C++ class
- myapp.cpp:
    - **#include “circle.h”**
    -
    - **int main() {**
        - **Circle c(-2, 5, 10);**
        - **c.draw();**
    - **}**
- **circle.h:** // class declaration
    - **class Circle {**
        - **public:**
        - **Circle(,,);**
        - **void draw() const;**
        - **private:**
        - **double m_x;**

    - **};**
    -
    - **double area(Circle x);**
- **circle.cpp:** // implementation
    - **#include “circle.h”**
    - **#include <cmath>**
    - **using namespace std;**
    -
    - **Circle::Circle(,,)**
    - **: _ _ {**
        - **...**
    - **}**
    -
    - **Circle::void draw() const {**
        - **sin(...);**
    - **}**
    -
    - **double area(Circle x) {**
        - **...**
    - **}**
- -o myapp myapp.cpp stuff.cpp (no header files listed)
- Don’t **#include ___.cpp**

1/13: Steps of Construction and Resource Management

- Steps of construction:
    - 1. ------
    - 2. Construct the data members, using the member initialization list; if a member is not listed:
        - If a data member is of a built in type, it is left uninitialized
        - If a data member is of a class type, the default constructor is called for it; error if there is none
            - If you declare no constructor for a class, the compiler writes a default constructor for you
    - 3. Execute the body of the constructor

- **class Circle {**
    - **public:**
    - **Circle(double x, double y, double r);**
    - **…**
    - **private:**
    - **double m_x;**
    - **double m_y;**
    - **double m_r;**
- **};**

- **Circle::Circle(double x, double y, double r) : m_x(x), m_y(y), m_r(r) {**
    - **if (r <= 0) {**
        - **cerr << "...." << endl;**
        - **exit(1);**
    - **}**
- **}**

- **class StickFigure {**
    - **public:**
    - **StickFigure(double bl, double headDiameter, string nm, double hx, double hy);**
    - **…**
    - **private:**
    - **string m_name;**
    - **Circle m_head;**
    - **double m_bodyLength;**
- **};**

- **StickFigure::StickFigure(double bl, double headDiameter, string nm, double hx, double hy) : m_name(nm), m_head(hx, hy, headDiameter/2), m_bodyLength(bl) {**

// must have m_head in initialization list, since Circle has no default constructor, and will error out if in the body of the constructor, m_name is in this list for performance reasons
  - **if (bl <= 0) {**
    - **cerr << "......" << endl;**
    - **exit(1)**
  - **}**
- **}**

- Common to use member initialization lists to do as much of the initialization as possible

- **void f(String t) {**
  - **String u("Wow");**
  - **…**
- **}**

- **void h() {**
  - **String s = "Hello";**
  - **f(s);** // constructing a String using another String, needs a copy constructor
  - **…**
- **}**

- **int main() {**
  - **String t;**
  - **for (...)**
    - **h();**
  - **…**
- **}**

- **class String {**
  - **public:**
  - **String(const char* value = "");** // makes value "" if there is no argument provided (default parameter value), only works for parameters at the end
  - **String(const String& other);** // copy constructor
  - **String& operator=(const String& rhs);**
  - **~String();**
  - **private:** // Class Invariants: m_text points to a dynamically allocated array of m_len+1 characters
  - m_len >= 0
  - m_text[m_len] == '\0'
  - **char* m_text;**
  - **int m_len;**
- **};**

- **String::String(const char* value) {** // do not repeat the default parameter value(s) here
    - **m_len = strlen(value);**
    - **m_text = new char[m_len + 1];** // must dynamically allocate a new array, otherwise you risk the original array being changed, therefore changing string's m_text's value unintentionally
    - **strcpy(m_text, value);**
- **}**

- **String::String(const String& other) {**
    - **m_len = other.m_len;** // allowed
    - **m_text = new char [m_len+1];**
    - **strcpy(m_text, other.m_text)**
- **}**

- **String& String::operator=(const String& rhs) {** // by convention, this returns the new value of the left hand side, return a reference to the object
    - **delete [] m_text;**
    - **m_len = rhs.m_len;**
    - **m_text = new char[m_len+1];**
    - **strcpy(m_text, rhs.m_text);**
    - **return *this;**
- **}**

- **String::~String() {** // ensure that all dynamically allocated objects are deleted when their local counterparts move out of scope
    - **delete [] m_text;** // make sure you match the form of delete to the form of new used (plain vs. array)
- **}**

1/15: Copy Constructors and Assignment Operators

- If you don't declare a copy constructor for a type, the compiler writes one for you
    - This copy constructor is made by copying all of the members
    - Creates problems with pointers:
        - Changing a value being pointed to inside a function that uses a copy also changes the original object
        - When other functions return, the destructor is called, and the value being pointed at is destructed
- If you write a destructor for a class, you likely have to write a copy constructor and assignment operator too
- Ok for any member functions to access private members of other objects of that class
- The copy constructor cannot be passed by value, as passing by value uses the copy constructor
- Assignment is not the same as the copy constructor

<u>1/17: Discussion 1</u>

- leetcode.com
- Dynamic Memory Allocation
  - Static memory allocation is inflexible
    - Assigns memory when compiling
  - **new** will automatically allocate the sequential memory space for the requested data type and size and return the starting address of the allocated memory space
  - Variables allocated with **new** will remain in the memory until deleted with **delete**
  - Dynamically allocated objects do not get deleted by the default destructor
- Copy Constructors
  - **School::School(const School &aSchool) {**
    - **m_name = aSchool.m_name;**
    - **m_numStudents = aSchool.m_numStudents;**
    - **m_students = new Students[m_numStudents]**
    - **for (int i = 0; i < m_numStudents; i++)**
      - **m_students[i] = aSchool.m_students[i];**
  - **}**
- Assignment Operator
  - **School& School::operator=(const School &aSchool) {**
    - **if (this != &aSchool) {**
      - **m_name = aSchool.m_name;**
      - **m_numStudents = aSchool.m_numStudents;**
      - **delete [] m_students;**
      - **m_students = new m_students[m_numStudents]**
      - **for (int i = 0; i < m_numStudents; i++)**
        - **m_students[i] = aSchool.m_students[i];**
    - **}**
    - **return *this;**
  - **}**

1/22: Linked Lists

- Arrays limited by methods of resizing and inserting items into the middle of the array
- **struct Node {**
    - **int value;**
    - **Node* next;**
- **};**

- **Node* head;**

- **Node* ptr = head**
- **while (ptr != nullptr) {**
    - **cout << ptr->value << endl;**
    - **ptr = ptr->next;**
- **}**

- **for (Node* p = head; p != nullptr; p = p->next) {**
    - **cout << p->value << endl;**
- **}**

- Usually preceded by a pointer (head) to the first element of the list
- Square bracket operators not valid
- When following a pointer, make sure it has previously been given a value and make sure that p is not the null pointer

- **Node* p;**
- **for (p = head; p != nullptr && p->value != 42; p = p->next) //** evaluate left to right
    - **;**
- **if (p != nullptr) {**
    - **cout << p->value << endl;**
- **} else {**
    - **cout << "There's no 42" << endl;**
- **}**

- **Node* p;**
- **for (p = head; p != nullptr && p->value != 42; p = p->next)**
    - **;**
- **if (p != nullptr) {**
    - **Node* newGuy = new Node**
    - **newGuy->value = 37;**
    - **newGuy->next = p->next;**
    - **p->next = newGuy;** // order matters, set new node's values first
- **}**

<u>1/24: Linked Lists (cont.) and Aliasing</u>

- Check normal case (middle), empty list, and boundary cases (first and last)

- **struct Node {** // doubly linked list
    - **int value;**
    - **Node* next;**
    - **Node* prev;**
- **}**

- **void transfer(Account& from, Account& to, double amt) {**
    - **if (&from != &to) {**
        - **if (from.balance >= amt) {**
            - **from.debit(amt);**
            - **to.credit(amt);**
            - **if (amt >= 10000) {**
                - **fileForm(...) {**
            - **}**
        - **} else {**
            - **cout << …. << endl;**
        - **}**
    - **}**
- **}**

<u>1/27: Stacks and Queues</u>

- Stack/LIFO:
    - Create an empty stack
    - Push an item onto the stack
    - Pop an item from the stack, undefined on an empty stack
        - The top of the stack keeps the same position
    - Look at the top item in the stack, undefined on an empty stack
    - Is the stack empty?
    - Maybe:
        - How many items are in the stack (C++ has)
        - Look at any item in the stack (C++ doesn't have)

- **#include <stack>**
- **using namespace std;**
- **int main() {**
    - **stack<int> s;**
    - **s.push(10);**
    - **s.push(20);** // adds 20 to top of stack
    - **int n = s.top()** // returns the top item of the stack
    - **s.pop();** // removes the top item of the stack, no return value
    - **if (!s.empty())** // asks if the stack is empty
        - **cout << s.size() << endl;**
- **}**

- Using a stack signifies that you are only using the top element
- Queue/FIFO:
    - Has a front/head and back/tail, 2 active ends
    - Create an empty queue
    - Enqueue an item (put it in the queue), can fail if there is a limited capacity
    - Dequeue an item (remove it from the queue), can fail if the queue is empty
    - Look at the front item in the queue
    - Is the queue empty
    - Maybe:
        - How many items are in the queue
        - Look at any item in the queue

- **#include <queue>**
- **using namespace std**
- **int main() {**
    - **queue<int> q;**
    - **q.push(10);** // enqueues item
    - **q.push(20);**
    - **int n = q.front();** // n is 10

        - **q.pop();**
        - **if(!q.empty())**
            - **cout << q.size() << endl;**
        - **cout << q.back() << endl;**
- **}**

- Queues and stacks have no limited capacity in C++
- String → numerical operations
    - Convert infix to postfix
        - If you hit an operand you append it to the result sequence
        - If you hit an operator and the operator stack is empty, push it onto the stack
            - Else if the top of the stack is an open parenthesis, push the operator
            - Else if the precedence of the current operator is greater than the operator at the top of the stack, push it
            - Else pop the top of the stack, appending it to the result sequence, and check again
        - Open parenthesis: always push
        - Close parenthesis: while top of stack is not an open parenthesis (operator) pop it and append it to the result sequence
            - Pop the open parenthesis
        - At end, pop each operator and append it to the sequence
    - Push operands onto a stack
        - When you hit an operator, you pop the operands and then push the result onto the stack

1/29: Inheritance

- Linked List vs. Array implementation of stacks
    - Harder to expand capacity with the array version
    - The array version takes up less memory
    - List version takes part in a lot of allocations and deallocations
- Linked List vs. Array implementation of queues
    - Use a doubly linked list for list implementation
    - Array:
        - Head and a tail (1 past the last element)
            - # items in queue is tail - head
        - When hit end of array, you can shift everything back to the beginning of the array
            - Downside if there are a lot of items constantly, many shifts
            - Can also wrap around to beginning of array - circular array
                - Head = tail if the list is empty or full
                - Make a size variable
    - Lists are easier when implemented as a doubly linked list
- Circular doubly linked lists are better for when you need to be able to insert/remove anywhere
    - Every node was guaranteed to have a node before and after it
    - Not very beneficial for queue implementation
- Inheritance:

- **class Circle {**
    - **void move(double xnew, double ynew);**
    - **void draw() const;**
    - **double m_x;**
    - **double m_y;**
    - **double m_r;**
- **};**

- **class Rectangle {**
    - **void move(double xnew, double ynew);**
    - **void draw() const;**
    - **double m_x;**
    - **double m_y;**
    - **double m_dx;**
    - **double m_dy**
- **};**

- **Circle ca[100];**
- **Rectangle ra[100];**
- **for (int i = 0; i < …; i++)**

- **ca[i].draw();**
- **for (int i = 0; i < …; i++) {**
  - **ra[i].draw();**

- **void f(Circle& x) {**
  - **x.move(10, 20);**
  - **x.draw();**
- **}**

- **void f(Rectangle& x) {**
  - **x.move(10, 20);**
  - **x.draw();**
- **}**

- This code contains a lot of duplicate code, and adding new classes is tedious and error prone

- **class Shape {** // Shape is a superclass/base class of Circle and Rectangle
  - **void move(double xnew, double ynew);**
  - **void draw() const;**
  - **double m_x;**
  - **double m_y;**
- **};**

- **class Circle : public Shape {** // Circle and Rectangle are a subclass/derived classes of Shape
  - **void draw() const;**
  - **double m_r;**
- **};**

- **class Rectangle : public Shape {**
  - **void draw() const;**
  - **double m_dx;**
  - **double m_dy**
- **};**

- **void Shape::move(double xnew, double ynew) {**
  - **m_x = xnew;**
  - **m_y = ynew;**
- **}**

- **Shape* pic[100];**

- **pic[0] = new Circle;** // if you have a pointer/reference of a derived type, it can be automatically converted to a pointer/reference of the base type
- **pic[1] = new Rectangle;**
- **pic[2] = new Circle;** // heterogeneous collection

- **for (int i = 0; i <...; i++) {**
    - **pic[i]->draw();**
- **}**

- **Circle c;**
- **f(c);** // x has to be another name for a Shape object
- **c.move(30, 40);** // Circle doesn't have a move function, compiler moves up and searches its base class

1/31: Discussion 2

- Linked Lists
  - Minimum:
    - Key component as unit: Node with a value and pointer to the next Node
    - Head pointer → points to the first term
    - Loop-free (except in circular linked list)
  - Regular Operations:
    - Insertion, Search, Removal
  - Pros and cons:
    - Efficient insertion, flexible memory allocation, simple implementation
      - Arrays have to occupy contiguous memory locations
    - High complexity of search
  - Insertion:
    - Steps: Create a new Node, make its next pointer point to the first item, make the head pointer point to the new Node
  - Search:
    - Steps: Find matching Node and return, if no match then return
  - Removal:
    - Remember to set the previous Node's next pointer to point to the Node following the deleted Node
    - Consider what happens if the selected Node is the head or the last Node in the list
  - Suggestions:
    - Draw pictures and carefully trace through your code
    - Check any operations for the beginning of the list, end of the list, empty lists, and one-element lists
- Stack
  - FILO: First in, last out
  - A standard stack implementation:
    - push() and pop()
    - Other methods: top(), count()
  - Applications:
    - Stack memory: function call
    - Check expressions: matching brackets
    - Depth-first graph search
  - How do you implement stacks with linked lists?
    - push(): Insert Node before head
    - pop() : Remove head and return the head value
    - top(): Read head Node
    - count(): Maintain a private int data member to keep track
- Queue
  - FIFO: First in, first out
  - Basic methods:

        - enqueue(), dequeue()
        - front(), back()
        - count()
    - Applications:
        - Data streams
        - Process scheduling (DMV service request)
        - Breadth-first graph search
    - How to implement queue with linked lists?
- Suggestions on stacks and queues
    - Draw pictures and carefully trace the code
    - Infix to postfix conversion is very important
    - Stacks and queues can be applied to trees/graphs
    - You can use the given Standard Template Library to implement stacks and queues

<u>2/3: Inheritance (cont.)</u>

- Every derived object has a base object embedded in it
  - Base object is constructed first
- Calling a member function of a derived object in a function that takes a base object as the parameter will not compile
  - These kind of functions can only access functions that the base object can definitely access
-
- **void Shape::draw() const {**
  - **… draw a cloud centered at (m_x, m_y)** // not the right implementation for derived objects
- **}**

- **void Circle::draw() const {**
  - **.... draw a circle of radius m_r centered at (m_x, m_y)**
- **}**

- **void f(Shape& x) {**
  - **x.move(10, 20);**
  - **x.draw();**
- **}**

- Static binding vs. dynamic binding
  - Connecting the name and the function
  - Static binding - function decided in compile time - uses Shape's draw()
    - C++ default
  - Dynamic binding - function decided in runtime - uses derived type's draw()
- Use **virtual** to make function dynamically bound

- **class Shape {**
  - **void move(double xnew, double ynew);**
  - **virtual void draw() const;**
  - **double m_x;**
  - **double m_y;**
- **}**

- **Shape::move(...)** // calling the base class' function

- **class Rectangle : public Shape {**
  - **void draw() const;**
  - **virtual double diag() const;**
  - **double m_dx;**

- **double m_dy;**
- **}**

- **double Rectangle::diag() const {**
    - **return sqrt(m_x*m_x + m_y*m_y);**
- **}**

- **class Shape {**
    - **void move(double xnew, double ynew);**
    - **virtual void draw() const = 0;** // function introduced as virtual, but not implemented, pure virtual function
    - **double m_x;**
    - **double m_y;**
- **}**

- You cannot create objects that have pure virtual functions
    - You can create pointers to these objects
    - If an object has at least one pure virtual function, it is an abstract class cannot be constructed)
- Abstract classes still have objects, they are just never created by themselves

- **class Shape {**
    - **…**
- **};**

- **class Polygon : public Shape {**
    - **…**
    - **~Polygon()** // necessary for Polygon, not Shape
    - **Node* head;**
    - **…**
- **};**

- **Shape* sp;**
- **sp = new Polygon**
- **delete sp;** // calls the Shape destructor, not Polygon's, must declare Shape to have a virtual destructor (must also be implemented)

- Destruction: 1) Execute the body of the destructor, 2) Destroy the data members, 3) Destroy the base part
- If a class is designed to be a base class, declare and implement a virtual destructor for that class

- **class Shape {**

    - **public:**
    - **Shape(double x, double y)**
    - **…**
    - **private:**
    - **double m_x;**
    - **double m_y;**
- **};**

- **Shape::Shape(double x, double y) : m_x(x), m_y(y) {}**

- **class Circle : public Shape {**
    - **public:**
    - **Circle (double x, double y, double r);**
    - **…**
    - **private:**
    - **double m_r;**
- **};**

- **Circle::Circle (double x, double y, double r) : m_x(x), m_y(y), m_r(r) {}** // doesn't compile, m_x and m_y are private members of Shape, Circle cannot access them

- Construction: 1) Construct the base part, 2) Construct the data members, 3) Execute the body of the constructor

- **Circle::Circle (double x, double y, double r) : Shape(x, y), m_r(r) {}** // fixed

2/5: Recursion

- Base cases: can be solved without making a recursive call
- Recursive cases: make one or more recursive calls for strictly smaller (closer to a base case) instances of the problem
- Verify there's at least one base case and that each recursive call works towards that base case

- **int arr[5];**
- **…**
- **sort(arr, 0, 5);**
- **void sort(int a[], int b, int e) {** // sort a[b] to a[e - 1]
    - **if (e - b > 1) {**
        - **int mid = (b + e) / 2;**
        - **sort(a, b, mid);** // sort left half
        - **sort(a, mid, e);** // sort right half
        - **merge(a, b, mid, e);** // merges sequences b → mid and mid → e that produces a sorted final array
    - **}**
- **}**

- If tracing code, start tracing small, the problem should appear early on
- Divide and conquer - solve the problem by breaking it up (see above)
- The first and the rest - divide collection into first and the rest, then solve and combine
    - Sometimes more convenient to to last and the rest

- **bool has(const int a[], int n, int target) {**
    - **if (n <= 0) return false;**
    - **if (a[0] == target) return true;**
    - **return has(a + 1, n - 1, target)**
- **}**

- When you have a function returning a boolean using recursion above (returning the recursive call) make sure the return is not left off

- **bool solve(start, goal) {** // base cases: at the goal or surrounded by walls
    - **if (start == goal) return true;**
    - // mark this position as visited
    - // for each direction {
        - // if moving one step in that direction is possible and hasn't been visited {
            - // if(solve(pos reached by moving, goal) **return true;**
        - }
    - }
    - **return false;**

2/10: Template Functions and Class Templates

- **int minimum(int a, int b) {**
    - **if (a < b) return a;**
    - **else return b;**
- **}**

- **double minimum(double a, double b) {** // effectively the same code as the int minimum at the C++ language level
    - **if (a < b) return a;**
    - **else return b;**
- **}**

- **int main() {**
    - **int k;**
    - **cin >> k;**
    - **cout << minimum(k, 10) / 2;**
    - **double x;**
    - **…**
    - **double y = 3 * minimum(x * x, x + 10);**
    - **…**
    - **int z = minimum(0, 3 * k - k + 4);**
- **}**

- **template<typename T>** // declare T to be used in template, T is now a placeholder for a type name
- **T minimum(T a, T b) {** // both functions fit into this template
    - **if (a < b) return a;**
    - **else return b;**
- **}** // this function can replace both the int and double declarations above

- **"**Instantiate the template"
    - You have to match some template
    - The instantiated template compiles
    - The instantiated template has to do what you want
- You cannot mix types in the example above (passing an int and a double doesn't work)

- **template<typename T1, typename T2>**
- **bool isEqualTo(T1 a, T2 b) {** // this function can take 2 different types as parameters
    - **return a == b;**
- **}**

- **template<typename T1, typename T2>**
- **??? minimum (T1 a, T2 b) {** // no way to give a correct return type

- **if (a < b) return a;**
- **else return b;**
- **}**

- Conversions:
    - Boolean compared to int/double → boolean converted to int/double
    - Int compared to double → int converted to double

- **Chicken c1, c2;**
- **minimum(c1, c2);** // how do you compare 2 chickens? fits template but probably won't compile

- **char ca1[100];**
- **char ca2[100];**
- **char* ca3 = minimum(ca1, ca2);** // minimum would compare the addresses of the pointers, not the values they point to, will compile, wouldn't do what you want

- Given a function vs. a template for a function, the language will choose the function
- Passing by const reference is common

- **template<typename T>**
- **T sum(const T a[], int n) {**
    - **T total = 0;** // won't compile with string call, no conversion between string and int, solution: *builtInType*() gives the "default" value, change to **T total = T();**
    - **for (int k = 0; k < n; k++) {**
        - **total += a[k];**
    - **}**
    - **return total;**
- **}**

- **int main() {**
    - **double da[100];**
    - **…**
    - **cout << sum(da, 100);**
    - **…**
    - **string sa[10] = {"This ", "is ", "a ", "test."};**
    - **string s = sum(sa, 4)**
- **}**

2/12: STL

- **<vector>**
  - **push_back(n)** // adds an index to the vector that contains n
  - **pop_back()** // deletes the last index of the vector
  - **.size(), .front(), .back()** // same as queues
  - **[]** // access elements, undefined for subscripts out of range
  - **.at()** // access elements, throws an exception for subscripts out of range
  - **vector<double> vd(10)** // 10 doubles, each is 0.0
  - **vector<string> vs(10, “Hello”)** // 10 strings, each is “Hello”
  - **vector<int> vx(a, a+5)** // vx.size() is 5, v[0] is 10, vx[1] is 20, … , vx[4] is 50
  - **vector<int>::iterator q = vi.insert(p, 40);** // inserts 40 before p, returns iterator to new element
    - vector iterators support using +=/-= i
  - **p = vi.erase(q);** // erases the value at q, returns iterator to new element in that spot
- **<list>**
  - **list<int> li;**
  - **li.push_back(20);**
  - **li.push_back(30);**
  - **li.push_front(10);**
  - **cout << li.size();** // writes 3
  - **cout << li.front();** // writes 10
  - **cout <<li.back();** // writes 30
  - **li.push_front(40);**
  - **li.pop_front();**
  - **for (list<int>::iterator p = li.begin(); p != li.end(); p++)**
    - **cout << *p << endl;** // writes 10 20 30
  - **list<double> ld(10);** // ld.size() is 10, each element is 0.0
  - **vector<string> ls(10, “Hello”);** // ls.size() is 10, each element is “Hello”
  - **vector<string> vs(ls.begin(), ls.end());** // vs.size() is 10, each element is “Hello”
  - **list<int>::iterator p = li.end();**
  - **p--;**
  - **p--;**
  - **p += 2**; // won’t compile
  - **list<int>::iterator q = li.insert(p, 40);** // inserts 40 before p, returns iterator to new element
  - **list<int>::iterator q = li.erase(p);** // erases element at p and using p is now undefined, returns iterator to element after
- **<set>**
  - **set<string> s;**
  - **s.insert(“Fred”);**
  - **s.insert(“Ethel”);**
  - **s.insert(“Fred”);**

    - **cout << s.size();** // 2
    - **s.insert("Desi");**
    - **s.erase("Ethel");**
    - **for(set<string>::iterator ptr = s.begin(); p!= s.end(); p++) {**
        - **cout << *p << endl;** // prints using < (Desi is first)
    - **}**

- **template<typename T>**
- **T* find(T* b, T* e, const T& target) {**
    - **for (; b != e; b++) {**
        - **if (*b == target) break;**
    - **}**
    - **return b;**
- **}**

- **int main() {**
    - **int a[5] = {10, 40, 50, 20, 30};**
    - **int k;**
    - **cin >> k;**
    - **int* p = find(a, a + 5, k);**
    - **if (p == a + 5)**
    - **… not found …**
    - **else**
    - **… found, p points to the first element with that value ...**
    - **string sa[4] = {"Lucy", "Ricky", "Fred", "Ethel"};**
    - **string* sp = find(sa, sa + 4, "Fred")** // won't compile, "Fred" is a pointer to a char[]
- }

- **template<typename Iter, typename T>**
- **Iter find(Iter b, Iter e, const T& target) {**
    - **for (; b != e; b++) {**
        - **if (*b == target) {**
            - **break;**
        - **}**
    - **}**
    - **return b;**
- **}**

- **int main() {**
    - **int a[5] = {10, 40, 50, 20, 30};**
    - **int k;**
    - **cin >> k;**

  - **int* p = find(a, a+5, k);**
  - **if (p == a+5)**
  - **… not found …**
  - **else**
  - **… found, p points to the first element with that value ...**
  - **list<string> ls;**
  - **list<string>::iterator q = find(ls.begin(), ls.end(), “Fred”);**
  - **if (q == ls.end()) **
  - **… not found …**
  - **vector<int> vi;**
  - **vector<int>::iterator r = find(vi.begin(), vi.begin() + 5, 42);**
  - **if (r == vi.begin() + 5)**
  - **… not found ...**
- }

- **template<typename Iter, typename Func>**
- **Iter find_if(Iter b, Iter e, Func f) {**
  - **for (; b != e; b++) {**
    - **if (f(*b)) break;**
  - **}**
  - **return b;**
- **}**

- **bool isNegative(int k) {**
  - **return k < 0;**
- **}**

- **bool isEmpty(string s) {**
  - **return s.empty();**
- **}**

- **int main() {**
  - **vector<int> vi;**
  - **…**
  - **vector<int>::iterator p = find_if(vi.begin(), vi.end(), isNegative);** // function name by itself acts as a pointer to the function
  - **if (p == vi.end())**
  - **… not found …**
  - **list<string> ls;**
  - **…**
  - **list<string>::iterator q = find_if(ls.begin(), ls.end(), isEmpty)**
- **}**

2/24: Big-O

- **for (int i = 0; i < N; i++) {** // O(N)
    - **c[i] = a[i] * b[i];** // O(1)
- **}**

- Work from the inside out

- **for (int i = 0; i < N; i++) {** // O(N^2)
    - **a[i] *= 2;** // O(1)
    - **for (int j = 0; j < N; j++) {** // O(N)
        - **d[i][j] = a[i] * c[j];** // O(1)
    - **}**
- **}**

- **for (int i = 0; i < N; i++0 {** // O(N)
    - **a[i] *= 2;** // O(1)
    - **for (int j = 0; j < 100; j++) {** // O(1) → still a constant, not reliant on N
        - **d[i][j] = a[i] * c[j];** // O(1)
    - **}**
- **}**

- **for (int i = 0; i < N; i++) {** // O(N^2)
    - **a[i] *= 2;** // O(1)
    - **for (int j = 0; j < i; j++) {** // O(i) → proportional to 0 + 1 + 2 + … + (N-1) = (N-1)N/2
        - **d[i][j] = a[i] * c[j];** // O(1)
    - **}**
- **}**

- **for (int i = 0; i < N; i++) {** // O(N^2)
    - **if (std::find(a, a+N, 10*i != a+N) {** // takes time proportional to O(N)
        - **count++;**
    - **}**
- **}**

- **for (int i = 0; i < N; i++) {** // O(N^2 log N)
    - **a[i] *= 2;**
    - **for (int j = 0; j < N; j++) {** // O(N log N)
        - **d[i][j] = f(a, N);** // suppose f(a, N) is O(log N)
    - **}**
- **}**

- **for (int i = 0; i < N; i++) {** // O(RC log C)

    - **a[i] *= 2;**
    - **for (int j = 0; j < N; j++) {** // O(C log c)
        - **d[i][j] = f(C);** // suppose f(C) is O(log C)
    - **}**
- **}**

- Selection Sort: O(N^2)
- Bubble Sort:
    - Worst Case: O(N^2)
    - Average Case: O(N^2)
- Insertion Sort: O(N^2) → best of the N^2 sorts
- Shell Sort: O(N^1.47)
- Merge Sort: O(N log N)

<u>3/2: Hash Tables</u>

- Use a formula to set up a data structure that can contain a lot of data and is inexpensive to search/insert
- Always expect collisions
    - If load factor (# of items/# of buckets) > 1, there is guaranteed to be a collision
- Using a prime number of buckets allows there to be no common factor → more equal distribution

- Hash function
    - **unsigned int hash(string s) {** // numbers greater than the max size for unsigned ints are just represented by their last 32? digits
        - **unsigned int h = 216613261U;** // U makes the compiler treat it as an unsigned integer constant
        - **for (int i = 0; i != s.size(); i++) {**
            - **h += s[i];**
            - **h *= 16777619;**
        - **}**
        - **return h;**
    - **}**

- Hash table with N items: lookup/insert/erase for fixed # of buckets: O(N) with low constant of proportionality
- Hash table with N items + rehashing: lookup/insert/erase: O(1) on average

- **unordered_set**
- **unordered_multiset** // allows duplicates
- **unordered_map**
- **unordered_multimap** // allows duplicate keys

- **#include <unordered_map>**
- **using namespace std;**
- **unordered_map<string, double> ious;**
- **string name;**
- **double amt;**
- **while (cin >> name >> amt) {**
    - **ious[name] += amt;**
- **}**
- **if (ious.find(“ricky”) == ious.end()) {**
    - **cout << “Ricky doesn’t owe me anything” << endl;**
- **}**

- **#include <functional>**
- **#include <string>**

- **string s;**
- **cin >> s;**
- **unsigned int h = std::hash<std::string>()(s);**
- **unsigned int h2 = std::hash<double>()(3.75);**
- **unsigned int h3 = std::hash<string*>()(&s);**

3/4: Binary Trees

- Made up of nodes connected by edges, stemming from a root node, and paths that connect one node to another
    - There is a unique path from the root node to any other node in the tree
    - Nodes follow a parent/child relationship
    - A leaf node has no children, interior nodes do
    - The depth of a node is determined by how many edges away from the root node a node is
    - Any program with a tree that goes down multiple paths is most likely best implemented with recursion

- **struct Node {**
    - **string data;**
    - **vector<Node*> children;**
    - **Node* parent;** // not always necessary, returning to parent may or may not be relevant
- **};**
- **Node* root;**

- **int count(const Node* p) {** // base cases: empty tree and leaf nodes
    - **if (p == nullptr) {**
        - **return 0;**
    - **}**
    - **int total = 1;** // count self
    - **for (int k = 0; k != p->children.size(); k++) {**
        - **total += count (p->children[k]);**
    - **}**
    - **return total;**
- **}**
- **cout << count(root) << endl;**

- **void printTree(const Node* p, int depth) {**
    - **if (p == nullptr) {**
        - **return;**
    - **}**
    - **cout << string(2 * depth, ‘ ‘) << p->data << endl;**
    - **for (int i = 0; i < p->children.size(); i++) {**
        - **printTree(p->children[i], depth + 1);**
    - **}**
- **}**

- Preorder traversal: process the node before you process the children
- Postorder traversal: process the node after you process the children

- A binary tree is either:
    - Empty
    - Or is a node with 2 binary subtrees (a left subtree and a right subtree)

- **struct Node {**
    - **string data;**
    - **Node* left;**
    - **Node* right;**
- **};**

- A binary search tree (BST) is either:
    - Empty
    - A node with a left BST and a right BST such that the value at every node on the left subtree is < the value at this node, and every node in the right subtree is > the value at this node
        - Assuming no duplicates

- **set<string> visitors;**
- **…**
- **string name;**
- **…**
- **visitors.insert(name)** // inserts if name is not already in the set
- **…**
- **if (visitors.find(“Ricky”) == visitors.end();**
    - **cout << “not found” << endl;**
- **for (set<string>::iterator p = visitors.begin(); p!= visitors.end(); p++) {**
    - **cout << *p << endl;**
- **}**
- **…**
- **set<string>::iterator p = visitors.find(“Fred”);**
- **if (p != visitors.end()) {**
    - **cout << “Bye, Fred” << endl;**
    - **visitors.erase(p)** // harmless to pass erase the end iterator
- **}**
- **…**
- **visitors.remove(“Lucy”);**

- In-order traversal:
    - **void printTree(const Node* p) {**
        - **if (p != nullptr) {**
            - **printTree(p->left);**
            - **cout << p->data << endl;**
            - **printTree(p->right);**

        - }
    - }

- Insertion/deletion/search - O(log N) in a relatively balanced BST
- Deletion:
    - Leaf node - delete the node
    - Node with one child - delete the node, move the child up
    - Node with two children - delete the node and take either the greatest value in the left subtree or the least value in the right subtree to replace it
- AVL Tree:
    - A BST where for every node, the height of the left subtree and the height of the right subtree differ by at most 1
    - Guarantees log N behavior
- 2-3 Tree:
    - Every node has 2 or 3 children
        - Nodes with 3 children contain 2 values, the 3rd child is between these 2 values
    - All leaf nodes have the same depth
- Red-black tree:
    - Used by STL for set, map, multiset, multimap
    - log N data structures