CS2 - ANB 105 - MWF 14-15

Yubo Su

Contents

1 1/6/14 - C++ introduction

We will be coding mostly in C++ (screw Python), though many lecture examples will be in various other languages. We will talk in high-level, object-oriented languages, and we'll have a lot of practice to destroy all the classic bugs! This will be an awfully biased class in favor of programming though CS is far more! MW lectures will be algorithmic while F recitations (class time) will be about programming! Enroll using Moodle, key "CS2EnrollMe" Moodle questions will be public, so search there first. Email collective TAs at cs2-tas@ugcs.caltech.edu

HW due Tuesdays 5pm through Moodle. Last two weeks = contest. Two 48h extensions free. Now let's start C++ with Head TA Ben Yuan!! :applause:

We are expected to use linux (fistpump). All assignments are about 20 points, of which 15 are "main" points and 5 are "advanced" points. 120 points + no missing work = pass! Let's dive into C++.

C++ is a compiled language! Example of a simple code

```
#include <cstdio>

int main(int argc, char** argv)
{
    printf(``Hello, world!\n'');
    return 0;
}
```

C++ is statically typed (declare variable types; can cast), i.e. `int foo = 42;`

Uninitialized variables have an undefined value until initialized. C++ has increment (++) and decrement (--) operators, inyoface python.

Functions are declared like in Java:

```
<return type> <name>(<argument list)
{
    <foo>
}
```

Variable scope exists. All C++ functions are "pass-by-value", so all passed values are copies of whatever variable is passed (variable $\neq$ pointer).

Global variables okay. Defining constants uses #define, i.e.

```
#define PI 3.14159
```

(probably not allowed b/c exceeds some length). `printf()` is a formatted print like python. If statements are socially acceptable. Note that $(=) \neq (==)$, but curiously `if(a=3)` is syntactically correct! Since `a=3` returns 3 which is not 0 which is not false, the code executes!

Three types of loops, while, do-while, and for, exactly like Java. Example of do-while

```
do
{
    printf(``screw you'');
}while(0);
```

Note semicolon placement.

Arrays are linear sequences of data. Random access, fixed size, elements are contiguous, but no range checking!

Strings are arrays of `char`'s terminated with null-character.

Pointers! Super-powerful, super dangerous. When a variable `i` is declared, its pointer is stored in *address* `&i`. To declare a pointer data type, we first define what the type that the pointer points to then use a star to say it's a pointer. Example

```
int i = 10;
int * j = &i;
printf(j = %p\n``, j);
printf(j points to %d\n``, *j);
```

The star operator is then used both for pointer declaration and pointer "dereferencing" (not to mention multiplication). What dereferencing means is to "resolve" the pointer, i.e. print what it points to.

We mentioned before that functions in C++ are call-by-value, but with pointers we can call-by-reference! Example of how this can be useful

```
void incr(int * i)
{
    (*i) ++;
}
```

which increments the object `i` points to!

2 1/8/14 - C++ at velocity, P.2

Assignments are distributed via Git, go to CS2 website for directions.

No style requirements outside of "be smart." Comments should explain/, not repeat.

`std::cout` is the standard output string, needs to `#include <iostream>`. Automatically converts values to text!

C++ allows to force call-by-reference by requiring argument `int & i` (compare to `int * i`. This functions almost exactly like pointers, but are strictly less powerful.

Pointer arithmetic! We can redirect where pointers point to. For example, examine

```
int i[50];
int * j = &i[0];
j += 5;
```

`j` now points to `i[5]`! Note that we are adding/subtracting multiples of the type size, i.e. here we are adding `5 * sizeof(int)`. Note that array notation is just syntatic sugar for pointers, i.e. `arr[3]` is the same as `*(arr + 3)`; note that the name of the array points to the first element of the array.

We can use pointers to request heap memory (dynamic memory allocation), `new` keyword, since heap $\gg$ stack, so maybe `int * buf = new int[9000]` . We delete by using `delete` for singletons, and `delete[]` for an array.

The `struct` allows us to define compound data types compactly. e.g.

```
struct point {
    double x;
    double y;
};//MUST have semicolon

point p;// in C, use 'struct point'
p.x = 3.0;
p.y = 4.0;
```

`structs` can include pointers, notably to other structs (data structures, voila!). Arrow notation allows us to access into a pointer to a struct, i.e. `(*pointer).a` is the same as `pointer->a`.

Linked lists have *root* at its first value and the pointer to access later elements is called the *iterator*.

Classes are extra-fancy structs! Syntax

```
class Polygon
{
    private:
        double width, height;
    public:
        Polygon(){. . .}//constructor
        ~Polygon(){. . .}//destructor
        //functions
}
```

These classes have member functions/visibility! Structs are basically just classes with completely public visibility (in C, much more limited). Visibilities are private (default), public, and protected (only subclasses). Functions fall into four classes, constructors, destructors, accessors, mutators. Header file `*.hpp` tells what variables/functions are in a class definition. Then writing a class implementation in `*.cpp` must use in function names `Class::func()` (note the difference; it tells what `Class` the `func()` belongs to).

Classes are blueprints for objects/instances, not objects themselves. Java! To instantiate, we write `Class constructor(params);`; note no "new" keyword since we are not using dynamic memory allocation. To dynamically allocate, we write `Class * poinhter = new Class(args);`.

To include a class as a data type, we use `#include Class.hpp`. `this` is a pointer to own class. e.g.

```
void Class::SetX(double x)
{
    this->x = x;
}
```

3 1/10/14 - Makeup: C++ at velocity, P.3

Templates are our friend. Like `ArrayList`s in Java, templates can take a generic class type `T` and do something with it. For example

```
template <class T>
T sum(T a, T b)
{
    T result = a + b;
    return result
}
```

Templates can be used for Classes/structs as well! Generally template classes must be completely defined in header file.

4 1/13/14 - Sorting

Suppose we have an array of integers of size `A` of size `A.length` and we want to sort this. Not surprisingly, there are lots of ways to sort! This will be covered below, so we can group sorting algorithms.

There are a few ways to compare efficiency, best/worst case scenario, computational complexity, generalizable to arbitrary data types. Analysis of *computational complexity* is measurement of time to run in terms of number of operations as inputs scale. Usually this is a measure of number of operations for input size n, so $O(n)$ asymptotic behavior worst-case scenario. $\Omega(n)$ is best-case scenario asymptotic. Of course we usually discuss worst-case behavior.

- Bubble sort: iterate through the array in pairs and swap any pairs that are out of order. Repeat until array is sorted. Note that n-th pass takes n-th largest value to the end of the array, so `A.length - 1` number of passes required. We can optimize this slightly by remembering the index of the last swap and only checking up to the last swap (so if the last half is already sorted we only iterate up to the index of the final swap i.e. halfway through the array). $O(n^2)$.
- Insertion sort: insert k-th element within first sorted $(k-1)$ elements. $O(n^2)$.
- Selection sort: find smallest and place at beginning, etc. $O(n^2)$.
- Merge sort: split into smaller sets, sort them, and merge! Recursion is key, complexity is $O(n \log n)$.

Assuming a comparison-based approach, is there a theoretical lower limit to complexity? Given a list of n numbers, there are $n!$ permutations, of which 1 is sorted. We then note that each action of comparing halves the possibilities (note a correspondence; if we take two elements and we are comparing them, they are still one of the $n!$ permutations, so by identifying the order of two terms we invalidate half of the permutations). Thus, the theoretical lower limit is $\log_2 n! \propto n \log n$ (using Stirling approximation). Thus, a comparison-based approach is at best $\Omega(n \log n)$ (since this is the worst-case). Next lecture convex hull!

5 1/15/14 - Convex hull

Given a set S of points in two dimensions, convex hull is smallest convex polygon containing all S. This is closely linked to sorting; consider the convex hull of a 1D list, which is just points in order. Then convex hull is almost 2D sort.

A first algorithm is to check all pairs of points and determine whether it "stabs" the potential convex hull; if it doesn't go through the polygon then it must be an edge. Easy way to check whether it goes through is to take cross products of the vector through the two points and vector through one point and a third point; if signs for all possible choices are the same, then all are on same side of the "stabbing"

line! Clever. This approach is $O(n^3)$ if we think carefully.

Much faster algorithm is gift wrapping algorithm. We can begin by finding a single point on the convex hull, i.e. min x position. Then we start at this point and determine the "angle" of the other points in the set. The one with an extremum of an angle is then the next point on the CH! etc. Brilliant! This is worst case $\Omega(n^2)$, but most precisely it is $O(nh)$ with h number of points on the CH.

Let's try to do even better. Let's think back to divide and conquer, first in a sorting complex. Recursive sorting? Merge sort! Recursion is also useful for fractals.

6 1/17/14 - `Vector`'s, IO, sorting

Dynamic/static arrays are both fixed size, but static is set at compile time while dynamic is set at runtime; dynamic comes from heap while static comes from stack.

Arrays are pretty limiting though, no easy way to resize, pointers are ugly, but mostly no bounds checking! What we want out of a perfect container includes resizing, bounds checking, and removing/inserting elements at either end easily. C++ gives us this as a `vector`. The way we would use this would be

```
#include<vector>

std::vector<int> nums(20);
for (int i = 0; i < 20; i++)
{nums[i] = i * 2;}
```

We can also use `push_back()` function to insert at the beginning. We can also resize at will! Look into documentation for this because I didn't manage to copy this down.

`Vector` class provides iterator `std::vector<int>::iterator i;` which can be dereferenced to get elements and can be pointer-arithmetic'd to move around in the vector class (since obviously pointer arithmetic doesn't work well for non-arrays. Vector iterator has `begin()` and `end()` functions to give the first and last valid pointers, so bound checking!

Command line arguments is given as argument to `main(int argc, char ** argv)`. Note that `argc` is the number of command line arguments "plus 1" because the program name itself counts as an argument. Then `argv` is the array that contains these arguments.

File IO is also doable! Uses `cstdio` library. To open file can do like `FILE * f = fopen("fish", "rb+"`. Then `fread()` automatically reads from file with pointer `f` (`gread()` would then read from `FILE * g`). C++ also can read files as streams.

Let's go to sorting. Bubble sort. Merge sort. Quick sort.

Convex hull! Gift wrapping. We can also Graham scan, i.e. get two consecutive points on the hull and then keep iterating around and eliminate all "right turns" in any arbitrary path. Since initially though we have to sort by angle the algorithm is $O(n \log n)$.

7 1/22/14 - Graham scan

Now time to discuss Graham scan, after the HW set is in. zzzzz. Merge/recursion can be used to directly compute convex hull, can generalize to 3D. Recursion is powerful! But limited.

Towers of Hanoi! Classic. Rules are explained, zzz. Recursion is best way. Start with stack of n, move stack of $n-1$ to another stack, move bottom to target stack, move $n-1$ on top, gg. Recursion wins.

Matrix multiplication is also suitable for recursion. Consider normal matrix multiplication $C_{ij} = A_{ik}B_{kj}$ which has complexity $O(n^3)$. The Strassen idea of factoring out operations is a very obstruse way of matrix multiplication that only uses seven multiplications instead of eight but has very much addition overhead! Thankfully, multiplication is much slower than addition so this is theoretically faster.

The Strassen multiplication lends itself well to recursion and does reduce complexity under assumption $+ \ll \times$. It lends itself to recursion by taking the "matrix elements" to be matricies themselves! This produces complexit $O(n^{2.81})$. Why 2.81? $T(n) = 7T(n/2) + O(n^2)$ and $\log_2 7 = 2.81$.

Recursion is not a panacea, nt always a good idea. It is sometimes just too slow, fibonacci sequence example. Memoization or non-recursive function is best way to solve this (memoization is to store results of previous computation).

Binary search is conventionally recursive, but can be made iterative! Key is `while(lowel <= upper)` loop, and modifying these indicies.

Common recursion pitfalls include lack of base case, running out of memory, obfuscation.

8 1/24/14 - Data structures

Data structures are the lifeline of computers. Trees! Binary search trees have two child nodes, one smaller and one larger! Very efficient for searching, $O(n \log n) < O(n)$ for a linked list. Obviously balancing trees is EXTREMELY important.

Stacks only operate from the top, "last in first out." Usually include pointer to top of stack, each element includes pointer to next element.

Dequeue is just "first in first out." Usually include pointer to front and rear of queue, each queue item points to next.

Depth-first search searches as far as possible along a branch before backtracking to a node with an unvisited child. Stacks are always DFS while queues are always breadth-first searched. Search "levels" (enqueue each successive element's children while searching).

For the purposes of our HW assignment, mazes are trees!

Advanced topic: quadtrees! Each node can have at most four children, best visualized in 2D with squares. How to add elements to quadtree? When adding a point to an existing quadtree, add to correct place and if already occupied subdivide until everything is in unique subnode..

9 1/27/14 - Data Structures

Data structures!

- Arrays: reserved memory space, access by index. (optimised is to have twice array size of expected).
- Linked List: no direct access, no resizing. Half an hour of how linked lists work
- Doubly-linked list: linke both ways! Circular lists also exist.
- Stack: First in Last out. Implement via linked list
- Queue: First in First out.
- Tree: Set of nodes with a single starting point, nonlinear linked list. Trees are also connected graphs with no cycles.

Terminology for the tree data structure. Nodes have parent and child. Ancestors have depths, root is 0. Same level nodes are "siblings." A node with no children is a leaf. Path is sequence of nodes from any node to any node. Trees with at most n children are n-ary trees, full if exactly n children. Height = depth - 1.

10 1/31/14 - Dypo!!

Homework will be DNA alightment and seam carving, find most matches between DNA strings and find least/most weighted path of an array from top to bottom. Useful classes include

- `std::string` substring gives substring, compare yields 0 if equal, size gives length, +, += concatnate strings.
- `std::unordered_map` stuff I forgot

Dynamic programming is basically just memoized recursion. Trades space for time. Let's try DNA alignment, given two strings of DNA find the optimal way to align them; 2 points for a match, -1 points for a mismatch, -5 points for a gap. Recursively then we want to make function calls to compare shorter strings. The function then call recursively the best of the following three cases

- remove first char from first string
- remove first char from second string
- remove first char from both strings

Then the subcase with the best total score is returned. Base case is then aligning two strings when at least one string is empty. Time complexity is then $O(3^n)$ which is terrible! Let's memoize. Use `std::unordered_map` with keys string1 + "," + string2. Value will be a struct containing int score and something I missed.

Seam carving! Shrinking pixels. Using a *saliency map* we determine how different a pixel is from its neighbors and each iteration remove the least "energetic" i.e. noticable pixels. If we brute force this, that's $O(W^L)$ for picture WxL. Memoizing this then means to build table row by row to calculate shortest path from above pixels. We then just find the lowest cost pixel and backtrace (backtracing is just to take the lowest sum below current).

Saliency map in assignent is stored as 1D array but we have macro to translate to 2D bounds. Knapsack problem is also useful for dypo!

11 2/3/14 - Graphs

Extensions of trees, graphs! Graphs comprise nodes and links (edges) where cycles are allowed. There will often be weighting assigned to edges, and edges sometimes are "directed" or one-dimensional.

Networking! Obviously just graphs between computers. Each machine must have address, i.e. IP address, 32 bits for now, and domain name (DNS servers translate). How do we connect nodes? Before, it used to be connecting two lines manually, circuit switching; no overhead, reliable! Too rigid

though, so itstead of circuit switching we now do *packet switching*, sending packets of data all around with designated receiver. Each packet is sent separately, oftentimes using different path and different orders, but will be reconstituted at receiver! Sender then receives an acknowledgement from receiver that packet was received; delay is called round trip time (RTT).

Virtual circuit switching is possible, where virtual circuits are switched to open up a direct path from sender to receiver. I ignored the rest of the slide. TCP is an example of this, where all computers along the path must consent to being part of the transmission line. Then other protocols like ethernnet, IP, UDP are all the packet switching/connectionless switching type. Networks are graphs.

Two basic ways to encode graphs. First, adjacency matrix, binary bit for each pair of computers (if computers i, j are linked then $A_{ij} = 1$). Second I missed, I am the worst. If we want to find the single-source shortest path then dypo ca be used to traverse!

12 2/5/14 - Parallel processing!

Recall that shortest path in graph is very useful, Google maps! Roads can be weighted by a lot of factors. Our algorithm is thanks to *Dijkstra*!!! NEVER forget this name.

Two ways to graph search, DFS and BFS. MapReduce is the parallel version of BFS, implemented in Hadoop.

All-pair shortest paths can be computed dypo, Dijkstra is fastest for positive weights, Floyd-Warshall otherwise. Minimum spanning trees, shortest length tree that connects all nodes in a graph, called Prim's algorithm.

Our discussion of networking has been very condensed, haven't discussed ports or sockets.

Today we discuss concurrency, designing and understandi systems that use parallel processing. Programs only look concurrent because alternating between tasks, interrupts! Multiple "threads." How does a single core emulate multiple processors/threads? OS (kernel! say Brian and Yubio) takes care of this. Processes are time-shared by the kernel which executes a context switch at appropriate times (maximize efficiency/fairness/utility).

Global code is dangerous when multitasking! Critical code. Mutual exclusion (mutex) during code's critical section; no code can execute while code is inside critical section. But at the same time, not everybody can say in critical section, so no deadlock; bounded waiting.

13 Big respite...

Mutexes and Semaphores are the basic data types for parallel processing; mutex means only one function can execute, semaphore means up to some limit.

We then covered numerics, how numbers are represented and how we can do even bigger numbers using scientific notation (at loss of precision). This is super brief-d; check lecture slides for more info.

Networking was next. A network is a collection of interconnected hosts (computers). Each host has network interface(s) with network address. Ports tell host which data packet goes to which application. Contemporary computers have ports $[0, 65535]$ where $[0, 1023]$ are system ports. Sockets translate from software to hardware; TCP/UDP. TCP guarantees in-order delivery, super reliable but slower; bad for low-latency apps. UDP is like the Wild West comparatively, much faster but no checking and packetizing.

For hw, we will use a `CS2::Net::Socket` class to take care of many underlying operations. Using an instance of this class `sock`, we can call `Connect(std::string hostname, uint16_t port)` with the port just an integer to connect to a host; if return value is 0 then we're okay, else connection failed, so check in code! Then `sock.Send(&data)` will send some data, and `sock.Recv(int size, bool blockAll)` will receive into its return type. Both are blocking until first server response, `Recv()` can also block until either error or all data is received by toggling `bool blockAll`.

To avoid having to wait forever for blocking, we can `poll()` among other things; refer to lecture slides. Endian-ness is whether most significant bits are put first or last.

In our assignment we will have a GUI. The idea is that the GUI will listen for signals and call a callback function that can do useful things. String tokenization is useful, refer to lecture slides. Rest is BS.

14 2/23/14 - Makeup: FFT

We begin FFT by discussing polynomials $p(x) = a_k x^k$. Adding two polynomials must be $O(n)$ to add each coefficient; evaluating them as follows is optimal

$$p(x) = a_0 + x(a_1 + x(a_2 + x(\dots))) \tag{1}$$

Polynomial multiplication seems to be $O(n^2)$, but we can be clever. Notice that *evaluating* the product is easy, but it's hard to express it explicitly. We want to map the coefficients to something that preserves this ease. Consider $p(x)$ as a list of its coefficients

a_k. We can then map to a different parameterization $\hat{a}_k$ such that $\hat{a}_k = p(w^k)$ where w is the nth root of unity, n the degree of the polynomial. This is the DFT.

Before we discuss why this is useful (`-.-`) we will discuss the FFT. Assume that n is a power of 2 (we can pad with zeroes if not). Let's define two auxiliary functions

$$p_{even}(x) = a_{2i}x^i \tag{2}$$

$$p_{odd}(x) = a_{2i+1}x^i \tag{3}$$

Then $p(x) = p_{even}(x^2) + xp_{odd}(x^2)$.

The cool thing about DFT is that the components are orthogonal, so multiplying the coefficients is $O(n)$ and inverse DFT, so yields multiplied coefficients faster than $O(n^2)$. This is basically it, find 30 more applications.