# Lecture 1: Structs and Arrays

Arrays:

Have set size

Function example:

void foo(int arr[], int size); //include size of array and the array

//itself

Structs: A C++ struct allows us to store multiple variables in a single entity. It is a way to organize data.

e.g Create a struct to store veterinary patient info: name, species, age, weight, sex

struct AnimalPatient{

string name; \*members

string species;

int age;

double weight;

bool sex;

int arr[5]

};

Different from class because all of the data members are public and there are no functions.

Struct usage:

Once a custom struct is defined, need a declaration in order to use it.

e.g.

int main(){

AnimalPatient p0, p1;

//use access operator to write to individual members

p0.name = “Steve”;

p0.species = “CAT”;

p0.age = 4;

p0.sex = 1;

p0.arr[0] = 17;

};

Array of Structs

AnimalPatient animalArr[3]; //store info for 3 patients

animalArr[0].name = “Bean”;

animallArr[0].species = “CAT”;

animalArr[1].name = “Steve”;

# Lecture 2: Reading in External Data Files

External Data Files:

How to read in external data files?

1. Use the <fstream> library
2. Declare a stream object
   1. ifstream inStream;
3. Connect stream object to the external
4. Read in data until a delimiter is reached (like a comma, for example)
5. Repeat step 4 until desired amount of data has been read into your program’s memory
6. Read until end of file is reached or desired amount of file is read

Method A) assume white-space delimiter

int x;

while (inStream >> x)

cout << x;

Method B) Custom delimeter

string arr[10];

getline(inStream, arr[0], ‘,’);

inStream = stream object

arr[0] = destination

‘,’ = custom delimeter

Command Line Arguments:

We can define main() to take in argument, similarly to other functions.

By doing so, when the compiled program is executed, the user can pass information/parameters to the program.

int main(int argc, char \*argv[])

argc = # of input arguments; at least 1

\*argv[] = array of pointers

first thing in the array is always the executable

# Lecture 3: Pointers

*Quiz released Friday and due Monday*

*Assignment 2 released Saturday due next Sunday*

Pointers: A variable type for storing a memory address.

Syntax:

int\* p; //a pointer is declared. \* specifies what kind of data your

//pointer is pointing too. In this case, it is type int.

int x = 3;

cout << &x << endl; //will print address of variable x

//& - address-of operator. syntax to get address of a variable

p = &x; //pointer storing the address of x

//alternatively

int \*p = &x;

//where to put the \*?

//Each of the following are valid:

int\* p;

int \*p; //preferred

int \* p;

int x, \*p1, y, \*p2; //p1 and p2 are the pointers and x and y are int

//How do we access the “target” value using the pointer

cout << x << endl;

cout << \*p << endl;

//\* = dereference operator – allows you to access what is in that //

//address

//without it, it just prints out the address

\*p = 7; //overwrites what is in the address.

p = 7; //overwrites the address,but cannot be done.

Pass by Value:

void myPBV(int x)

{

x = -7;

}

//a copy of the variable is being passed

//Does not actually change the value in the original function, just in //myPBV.

Pass by Pointer:

e.g:

void myPBP(int \*p)

{

\*p = -7;

}

//an address is being passed

//must call with a pointer type

//by passing a pointer to a function, the function has access to the //original variable.

e.g in main()

int \*myPtr;

myPBP(myPtr);

//alternative

int x;

myPBP(&x);

Pass by Value:

* reference:
  + caller gives access to original variable; changes made in function will persist in the caller’s variable space.
* array
  + passing the address of the 1st element of the array. changes made in function will persist in the caller’s variable space.
* pointer
  + passing the address of the pointer variable

void foo (int x, float &y, int arr[], int \*x)

{

y = 5.4; //causes b to be changed to 5.4

}

int main(){

int a = 4;

float b = 3.1;

int arr[6];

int x;

int \*ptr = &x;

foo(a, b, arr, ptr);

}

# Lecture 4: Automatic vs Dynamic Variables

Dynamically Allocated Memory

Automatic Variables get allocated on a stack.

Dynamically Allocated Variables get allocated at run-time and use heap memory (freestore).

In order to use the heap in C++, we use new and delete keywords.

Have to use a pointer to allocate memory on the heap

e.g.

int \*p1;

p1 = new int; //nameless variable allocation

//enough memory to store one int type variable

delete p1; //free the memory back to the heap.

// note “delete” does not delete the pointer

//allows us to reuse the pointer for other things

Dynamically Allocate Arrays: To allocate an array on the heap

e.g. : length 5 array

int n;

int \*ptr;

cin >> n;

ptr = new int[n]; //allocate

for (int i = 0; i < n; i++)

{

cin >> ptr[i];

}

delete [] ptr;

Array Doubling

Want an algorithm that will take an N length dynamic array and effectively double its length so that it becomes 2N.

1. Original array is pointed to by p
2. Create a new array of length 2N. Use a new pointer for this (call it temp)
3. Now copy all of the contents of the original array (p) to the new array (temp).
4. Deallocate the original memory pointed to by p (delete [] p)
5. Tell p to point to new array (p = temp)

temp = new int[2\*N]

# Lecture 5

* Finish up with dynamic memory allocation
* Array doubling coding example
* C++ classes review
* Multiple file compilation

Array Doubling Recap:

Dynamic Memory allows us to request new memory space at runtime. We can use this fact to expand our arrays at program runtime (as needed).

Consider we have an array (A) of size N. If that array is filled, since we cannot request contiguous memory space (even on the heap), we can grab a chunk of memory that’s 2N somewhere else on the heap. This creates a new array (temp) that is size 2N and array (A) is copied into the new array (temp). Then we deallocate (A) and it returns to the heap. Then, set A equal to temp.

If you don’t deallocate, you get a memory leak ☹

# Lecture 6

* Assignment 2: array doubling function… parameters
* C++ classes review
* Multiple file compilation
* Pointers and structs

*Array Doubling in a Function*

remember:

int foo(int\*a);

//is equivalent to

int foo(int a[]);

int main()

{

int length = 5;

cout << length; //5

arr2x(someDynamicArray, length);

cout << length; //10

//print array contents

}

void arr2x(int \*&inArr, int &N)//integer type that is a pointer to an int type and is a pass by reference

{

//allocate new array

int \*temp = new int[2\*N];

for (int j = 0; j < N; j++)

{

temp[j] = inArr[i];

}

delete []inArr; //deallocate old array

inArr = temp;

N = 2\*N;

}

*Multiple File Compilation*

Instead of keeping everything in one .cpp file, we can stay more organized by separating things into multiple files.

Typically, we end up with at least 3 files:

1. Header file
   1. class definition
      1. only prototypes (declarations)
   2. Extension is .hpp (.h)
2. Implementation file
   1. all member function definitions go here
   2. extension.cpp
   3. #include your header file
3. Driver file
   1. This is where main() goes
   2. purpose to use the class (exercise or test it)
   3. #include header file

Compilation:

Just compilation:

g++ -c Time12.cpp

results in an object file

Time12.o extension

g++ -c Time12Driver.cpp

result: Time12Driver.o

g++ Time12.o Time12Driver.o -o Time12Prog.exe

result: link together the object tiles

All-in-one call to g++:

g++ Time12.cpp Time12Driver.cpp -o Time12Prog.exe

equivalent result

*More on Pointers…:*

Pointer variable is described by the type that it points to :

int \*aptr;

data\_type \* pointer\_name

So far, only used fundamental type (int, double, string)

We can declare pointers to user-defined types.

e.g. struct:

struct Student

{

string name;

int age;

};

int main()

{

Student s0;

Student \*sptr;

sptr = s0;

//Remember, accessing members of the struct, use the dot

//operator:

s0.name = “Pat”;

s0.age = 44;

(\*sptr).name = “Pat Mcgee”;

//or

sptr -> name = “Pat Mcgee”;

}

# Lecture 7

* Pick up with Pointers and Structs
* Intro to Linked Lists

*What if a member of the struct is also a pointer?*

struct Node{

int someData;

Node\* next;

};

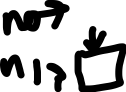
//Not pointing to itself. Rather, pointing to its own type.

int main(){



Node\* n0 = new Node;

Node\* n1 = new Node;



//fill some values

n0 -> someData = 3;

n1 -> someData = 5;

//Now, let’s “link” the two nodes together

n0 -> next = n1;

n0 -> next = nullptr; //nullptr = points to null space

// created a very basic linked list :D

}

*Array Drawbacks*

Have to sort numbers by moving and copying everything to adjust it

*LINKED LISTS*

Alternative method to the array for storing a list of data in memory

Each element occupies its own “Node”

Unlike the elements in an array, the nodes are not sequential in memory space

We can add a new node anywhere in the list by reconnecting pointers

Traversing a LL is done by “hopping” around in the memory space by following pointers

Besides the pointer that is part of each node, we have a separate pointer variable that is called the head.

Example: copy over elements from an array into a linked list:

struct Node {



int key;

Node \*next;

};

int main(){

int arr[] = {15, 12, 14};

int N = 3;

Node \*head, \*previous, \*temp;

head = new Node;

head -> key = arr[0];

head -> next = nullptr;

previous = head;

for (int I = 1; I < N; i++)

{

temp = new Node;

temp -> key = arr[i];

temp -> next = nullptr;

previous -> next = temp;

previous = temp;

}

//How to search a LL

}

# Lecture 9

struct Node{

string key;

int data;

…

node \*next;

};

int main(){

//statically allocate a node: not typical for Linked Lists

Node N; //let’s not do this :/

//Let’s dynamically allocate a node

Node \*ptr = new Node; //allocating one node’s worth of memory

//Populate the node

ptr.key = “fancy”; //NOPE, ptr does not have members

//Dereference the pointer

(\*ptr).key = “fancy”;

// equivalent result, different syntax

ptr->key = “fancy”;

//Let’s create another node

Node \*ptr2 = new Node;

//Link the two nodes;

ptr->next = ptr2;

//How to set second node (pointed to by ptr2) to point to null?

ptr2->next = nullptr;

//that’s fine, but what if I wiped out ptr2

ptr2 = nullptr;

ptr->next->next = null;

}

Singly Linked List Generic ADT:

private:

head – *ptr to first element in list. Null means empty list.*

public:

initialize() – set header to null (constructor)

nodePtr = search(value)- *find a value and return ptr to its node*

insertNode(previousPtr, newname) – *given a ptr to a node and insert a new node right*

*after it.*

displayList() – *traverse and list the values*

# Lecture 10

delete(Node \*ptr)

//function should never be called nullptr

2 cases to consider:

1. Node to be deleted is the head node
   1. establish the new head
   2. deallocate the old head
2. Node to be deleted is anywhere else in the list
   1. traverse list, stop at node that is previous to the one that needs to be deleted
   2. reconnect previous node to the next node
   3. deallocate the node to be deleted

*The Destructor*

Just like there is a constructor that gets called automatically when an object is instantiated, the destructor gets called when the object goes out of scope

No need to define destructors when not working with dynamic memory.

The syntax in definition uses ~<class name>

*Problem Statement:*

Add a method to the SLL class that will sort the entire list using the bubble sort algorithm. Use an auxiliary swap method.

Two Approaches:

1. Move nodes around the LL ( re-arrange the pointers )
   1. greater code complexity
   2. possibly more efficient
2. Copy data from one node to another
   1. simpler code
   2. possibly less efficient

# Lecture 11

**EXAM FRIDAY, OCTOBER 4, 5-7PM**

*Complexity*

How do we predict the performance of an algorithm? So many variables depending on the environment, etc.,

We use something called the big-O notations to describe the theoretical upper bound of an algorithm as N reaches infinity (time increases and N increases over time).

We can say time complexity for the given algorithm is O(N).

The best possible complexity an algo can have is O(1) – constant running time. Not dependent on input size. Rarely achievable.

*Stack*

Last in First Out: LIFO

Structure that only allows for a specific order which operations on its data can be performed. -> limited access data structure.

**push:** means to add

**pop:** means to remove

usage:

call stack during program execution

* currently active subroutine is at the top of the stack

# Lecture 12

**Stacks**

* Last in First Out data structure
* A limited access data structure
  + can only add to the top (push)
  + can only remove from the top (pop)
  + depends on implementation, can have a hard limit on max size
* Usage Examples:
  + during program execution, keeping track of active subroutines
  + work processor “undo”

*Stack ADT:*

private:

top – keeps track of top element

maxSize – limit on total size of stack (optional – depends on implementation)

count – current number of elements in stack

public:

initialize() – constructor

bool = isFull() – check if stack is full

bool = isEmpty() – check if stack is empty

value = peek() – show top item

push(item)- add a new item to the top

pop()- discard item from the top

disp() – traverse entire stack

//Stack SLL implementation

bool isEmpty()

if top == nullptr

return true

else

return false

Node \*Peek()

if isEmpty()

return nullptr;

else

return top

void push(*type* item) //O(1) does not depend on list size

*allocate a new node*

if isEmpty()

make the new node top

set new node->next = nullptr

else

make new node point to old top

make new node the top

void pop()

if !isEmpty()

temp = top;

move top to point to top->next

deallocate temp

else

“stack underflow” (stack is empty)

void display

*same as SLL display*

*Stack- array implementation*

private:

int top; //index of next available element

int count; //current number of elements

string s[MAX\_SIZE]; //the stack array

public:

bool isFull(){

return top == MAX\_SIZE;

}

bool isEmpty(){

return top == 0

}

string peek(){

if (!isEmpty)

return s[top-1];

else

“empty stack”

}

void push(string newItem){

if (!isFull()){

s[top] = newItem;

top++;

}

**Queue**

Another limited access data structure.

* now we “enqueue” at the “tail” of the queue
* “dequeue” form the head of the queue
* First in First Out (FIFO) (first in, first out)
* example usage:
  + call center
  + printer
  + read/write commands in storage firmware

*Queue ADR*

private:

head – first item in the Q (next thing to be processed)

tail – last item in the Q

queueSize – optional depending on implementation. Number of

elements currently in the Q

public:

initialize()

bool = isEmpty()

bool = isFull()

enque(item)

item = dequeue

Implementations:

* Linked list
* array
  + linear:
    - head is always element 0
    - tail holds the index of next available element

*Circular Array Queue Implementation*

Allowing for both tail and head to shift when enqueing and dequeing

head indexes the beginning of the Q

Tail indexes the end of the Q

empty Q => both head and tail point to arr[0]

# Lecture 13

Circular Q implementation

const int MAXSIZE = 4

private:

int head, tail, qSize;

*type* qArray[MAXSIZE];

public:

constructor:

head = tail = 0; //indexing

qSize = 0; //current number of elements in Q

enQ(item)

if(not full)//qSize ?= MAXSIZE

qArray[tail] = item;

qSize++;

if (tail is at the last array element)

tail = 0

else

tail++;

else

“q already full”

*type* deQ(){

type out =””;

if (Q not empty)

out = qArray[head];

qSize—

if (head is at last array element)

make head point to first array element

head = 0;

else

head++;

else

“Q is empty returning empty string

reutnr out;

}

**Trees**

hierarchical data structure (not linear)

Each element is called a node which are connected by edges.

Parent-child relationships.

Cannot have cycles and disconnected parts

*Binary Tree*

* In a binary tree, each node has exactly 2 children.
* One or both of the children can be null, so we can typically say that a node has 0, 1, 2 children.
* *parent, left child, right child*
* root is the topmost node
  + root’s parent is null
* Self-similarity
  + tree composed of smaller sub trees
  + each sub tree has properties of a regular tree
    - each child can be considered the root of a subtree.
  + computationally significant:
    - can use recursion to apply the same algorithm to smaller and smaller subtrees.

*Binary Search Tree*

* Smaller to the left and greater to the right

# Lecture 14

**Binary Search Tree**

* A BST is a special case of a general tree
* Every node can have 0-2 children
* Root is the topmost node
* Nodes w/ no children are called leaves
* From any node
  + left subtree has smaller key values
  + right subtree has greater or equal key values

*Searching a BST*

1. set root to crawler
2. if (crawler.key==searchKey)
   1. done
3. if (search.key < crawlerKey)
   1. “visit the left child”
   2. crawler = crawler.leftChild
4. else
   1. “visit the right child”
   2. crawler = crawler.rightChild
5. repeat 2-4 until searchKey is found

* *height* = number of “hops” from root to leaf
  + tells us the worst case number of operations needed to find **any** node in the tree.
* N = 2^(h+1) – 1 – way to calculate how many nodes are in a BST for a given height
* h = log\_2(N+1) – 1 – calculate the head / worst case num of operations to find any node in the tree
* in big-O notation, we drop the constants:
  + O(logN)
* what if N = 1,000,000
  + log(1,000,000) = 13.51