EXAM 1 REVIEW QUESTIONS CSCI 2270, Fall 2014

Why does the add\_node function

void add\_node(node\*& head\_ptr, const int& payload)

pass in the head ptr by reference? Give me a scenario where this is needed.

If you don’t pass in the head\_ptr by reference, it will be passed in by value. That means a copy of it would be created in the add\_node function, and deleted when add\_node ends. That means you wouldn’t be able to change head\_ptr at all.

For floating point numbers, which of these is most likely to be computationally accurate? Why?

0.375 0.25 0.1 0.3 0.5 0.15

How can I tell if two linked lists are shallow copies of each other?

If two linked lists are shallow copies of each other, the copied list will just be a pointer to the original list. If anything is changed in the copied list, it will also affect the same node(s) in the original list.

With a deep copy, the second list is created completely separate of the first one, so any changes to the second list will not affect the first list.

What’s wrong with this code for adding to an int\_array? How could you fix it?

if (arr.capacity == arr.count)

resize(arr);

arr.data[++arr.count] = payload;

The logic should be if (arr.capacity == (arr.count - 1)) because with arrays, the first element is int\_array[0], not int\_array[1]. This means arr.count would be one number higher than arr.capacity when the int\_array is full.

What will this code do?

for (unsigned int k = arr.count - 1; k >= 0; --k) cout << arr.data[k];

It will print out arr in reverse.

Write me a working function that uses an unsigned int in a loop to print the int\_array backwards.

int array\_backwards(int int\_array[], int size)

{

for (unsigned int i = size - 1; i >= 0; i—)

{ cout << int\_array[i]; }

}

Why do you get a warning when assigning between ints and unsigned ints? Is this a good idea or a bad one?

Signed and unsigned ints have data ranges - signed ints can be positive and negative, whereas unsigned ints must be non-negative. The compiler wants to prevent a signed int (possible negative number) from being assigned to an unsigned int (non-negative number).

When I pass a pointer by value, what happens to the data it points to? When should I pass a pointer by reference?

When a pointer is passed in by value, nothing happens to the data it points to because the function creates a copy of the pointer, and deletes it when the function ends.

A pointer should be passed in by reference when you want to make changes to actual pointer and not just a copy of it.

When should I pass a pointer by constant reference?

A constant reference is an alternative to passing in by value. It does not create a copy of the variable, but lets you make changes to it that only exist within the function, and not outside of it. It’s useful when passing in something large that you don’t want to have to make a copy of.

Why do I have to pass a size along with an array of ints?

Because you only want to access the elements within the bounds of the array. If you do not know the size of the array, you will access information out of the bounds of the array, giving you an error.

cout << "i = " << i << endl; // i = 10

cout << "p = " << p << endl; // p = 0x7fff5fbff88c

cout << "\*p = " << \*p << endl; // \*p = 10

cout << "&i = " << &i << endl; // &i = 0x7fff5fbff88c

cout << "p + 1 = " << p + 1 << endl; // p + 1 = 0x7fff5fbff860

cout << "\*p + 1 = " << \*p + 1 << endl; // \*p + 1 = 11

cout << "&i + 1 = " << &i + 1 << endl; // &i + 1 = 0x7fff5fbff860

Will this function

void lobo(int& num, unsigned int size)

let us change the data for num? Will it let us change the address of num in memory? If so, which of these changes will be permanent after the function runs? Why?

You can change the data for num, you cannot change the address of num in memory. Changes for num will be permanent after the function ends, because num was passed in by reference (no copy was made).

Will this function

void zobo(int num, unsigned int size)

let us change the data for num? Will it let us change the address of num in memory? If so, which of these changes will be permanent after the function runs? Why?

You can change the data for num, you cannot change the address of num in memory. None of the changes that take place in this function will be permanent after the function ends because num was passed in by value (a copy of num was created).

Will this function

void bobo(int\* array, unsigned int size)

let us change the data in an array? Will it let us change the address of the array in memory? If so, which of these changes will be permanent after the function runs? Why?

You cannot change the integer value of array, you can change the memory address of array. None of the changes will be permanent after the function ends because the pointer was passed in by value

Will this function

void flobo(const int\*& array, unsigned int size)

let us change the data in the array? Will it let us change the address of the array? If so, which of these changes will be permanent after the function runs? Why?

It is a reference to a const pointer, i.e. a pointer where you cannot modify the data pointed to. As the reference is used as an argument to a method the method is able to modify the pointer to let it point to something else (still something that cannot be modified).

Will this function

void globo(int\*& array, unsigned int size)

let us change the data in an array? Will it let us change the size of the array? Will it let us change the address? If so, which of these changes will be permanent after the function runs? Why?

What’s the main behavioral difference between variables on the local memory (stack) and variables on the heap?

Variables stored on the stack are destroyed after a function ends. Variables stored on the heap persist after returning the function, and do not disappear until using the delete command.

How do I copy the last 7 elements of a 20-element array to the last 7 elements of a 45-element array using the copy command?

copy (array1 + 7, array1+20, array2 + 38)

How can I tell if a pointer is bad?

You can’t.

Why is this code dangerous? string& get\_a\_string()

{

string answer;

cout << "Tell me a word: " << endl; cin >> answer;

return answer;

}

int main() {

string user\_input = get\_a\_string();

cout << "You entered " << user\_input << endl; }

Answer is destroyed after get\_a\_string ends, but because it was created by reference, when it is called again in main, it will have already been destroyed.

Why is insertion sort O(n2) at the worst? What’s the best case for insertion sort?

What can an array do faster than a list (and why?)

Sort through values. When an array is created, the memory needed for all the elements is sectioned off together.

What can a list do faster than an array (and why?)

Resize itself. Each element contains a pointer to the next element, so if another element needs to be added all that is required is creating that new node, and then setting the pointer to the previously last node to the new node.

What error happens if I write a loop with a condition like this to look for an element in an int\_array arr’s data:

unsigned int k = 0;

while (arr.data[k] < target && k < arr.count)

++k;

What happens in the following code, and is it a problem?

node\*head\_list\_1 = nullptr; add\_node(head\_list\_1, 9); add\_node(head\_list\_1, -78); add\_node(head\_list\_1, 200); node\*head\_list\_2 = list\_head\_1; add\_node(list\_head\_2, 100); print\_list(list\_head\_1); remove\_node(list\_head\_2, -78); print\_list(list\_head\_1);

Because head\_list\_2 is a pointer to list\_head\_1, any changes made to head\_list\_2 will also affect list\_head\_1.

Tell me why it’s better to double the int\_array capacity when we resize. (Hint: plot out how many items you copy as your count increases when your resize doubles the capacity, and how many items you copy when your resize just adds 20 more slots to the capacity. Comparing these plots should help.)

If you increase the int\_array capacity by one each time you resize, you still have to loop through the int\_array and copy each element into a new array, assign the new array’s value to int\_array, and then delete int\_array.

If you double the capacity, after you copy all the elements over, you can just add elements to the new array until the number of elements is double the size of the original int\_array. This way is less memory intensive.

Suppose my algorithm is quadratic, and I triple the size of its input. How much longer will it take to run?

Suppose my algorithm is logarithmic, O(log2n), and I quadruple the size of its input. How much longer will it take to run?