CS 16 4/30/19

Makefiles

Variables can be declared like the one below and accesses using $(VARIABLE\_NAME)

( e.g. $(CPP) )

CPP=g++ # variable

FILENAME.cpp : FILENAME.o

g++ -c FILENAME.cpp # makes object code (FILENAME.o)

# $@ is shorthand for the rule name

# $^ is shorthand for the right side of the ‘:’ after the rulename

# Here is what we had: g++ -c FILENAME.cpp

# Now using variable CPP the above line becomes

# $(CPP) -c FILENAME.cpp

EXECUTABLE : FILENAME.o

g++ FILENAME.o -o EXECUTABLE # makes executable from .o file

# g++ $^ -o $@ (Is another way of saying the line above)

clean:

rm \*.o EXECUTABLE # removes all compiled code

Memory Storage

Italicized items are not tested on:

*Little Endian – Stores Value 0x00 00 00 2A (ignore whitespace in memory address) as [2A,00,00,00] in memory (Intel and AMD processors)*

*Big Endian - Stores Value 0x00 00 00 2A (ignore whitespace in memory address) as [00,00,00,2A] in memory*

**Arrays**

Initializing array:

int arr[5]; // has 5 elements which are all integers

Initialize a 2-D array:

double myArr[10][2]; // has 10 rows and 2 columns that all have the double type

Initialize and assign values to an array in the same line:

char word[5] = {‘a’,’b’,c’,’a’,’t’};

Sample file:

#include <iostream>

using namespace std;

int main()

{

char word[] = {'a','b','c','a','t'};

for(char elem : word){

cout << elem << endl;

}

return 0;

}

Output:

a

b

c

a

t

Using arrays in a function:

Changing the elements in the array in the function, also changes the elements in the main function (or any other function that calls it)!

Sample file:

#include <iostream>

using namespace std;

int findFirstIndexOfC(char arr[], int sizeOfArr){

for(int i = 0; i < sizeOfArr; ++i){

if(arr[i] == 'c'){ // check if the element in the array is the character c

arr[i]= '~';

return i;

} // when we find the first c in the array we change it to a ~, exit, and return the index where we found the c

}

}

int main()

{

int index;

char word[] = {'a','b','c','a','t'};

cout << "Before we run the function the value of each element in the array words is : " << endl;

for(char elem : word){

cout << elem << endl;

}

index = findFirstIndexOfC(word, 5);

cout << "The first c occurs at index " << index << endl;

cout << "After we run the function the value of each element in the array words is : " << endl;

for(char elem : word){

cout << elem << endl;

}

cout << endl << endl << "Notice that the function call changes the value of the original array which is unlike other data structures." << endl << endl;

return 0;

}

Output:

Before we run the function the value of each element in the array words is :

a

b

c

a

t

The first c occurs at index 2

After we run the function the value of each element in the array words is :

a

b

~

a

t

Notice that the function call changes the value of the original array which is unlike other data structures.

Binary Search:

Only should understand at a high level.

Array is assumed to be sorted.

Start in the middle of the array.

If the value you are looking for is bigger than the middle value of the array then the value you are looking for must be in the right half of the array.

So we only have to search this side.

Else the value you are looking for is smaller than the middle value of the array then the value you are looking for must be in the left half of the array.

So we only have to search this side.

So we keep doing this until we find the value we are looking for.

(E.g. int arr[] = {1,2,3,4,5,6,7,8,9}) (9 elements)

Search for 2.

First choose arr[9/2] (which is index 4 (remember integer division truncates))

Arr[4] is 5.

5 > 2, so search the left half of the array (index 0 to 4) (5 elements).

Now choose arr[5/2] (which is index 2 (same rule as before))

Arr[2] is 3.

2 < 3, so search the right half of this smaller array. (index 2 to 4) ( 3 elements).

Now choose arr[3/2].

Arr[1] is 2.

2 == 2, so we return the index that we found 2 at (index 1).

Call by Reference

Use this to change values of variables that we are using in the main function when we pass it to other functions.

All we have to do is add the ‘&’ symbol when we pass a variable to the function.

Sample file:

#include <iostream>

using namespace std;

void changeVar(int &a){

a = 5;

}

int main()

{

int k = 3;

cout << "Before the function call the value of k is " << k << endl;

changeVar(k);

cout << "After the function call the value of k is " << k << endl;

return 0;

}

Output:

Before the function call the value of k is 3

After the function call the value of k is 5

Pointers

Pointers hold the address of the data type it is declared with.

Declare a pointer:

int \*p; // a pointer named p

int r = 3; // a normal int named r which has value 3

p = &r; // p now stores the address of r

// Now we can change the value of r with \*p.

\*p = 2; // Now r’s value is 2.

Here is how we use pointers similar to the program above:

#include <iostream>

using namespace std;

void changeVar(int \*a){

\*a = 5;

}

int main()

{

int k = 3;

cout << "Before the function call the value of k is " << k << endl;

changeVar(&k);

cout << "After the function call the value of k is " << k << endl;

return 0;

}

Output:

Before the function call the value of k is 3

After the function call the value of k is 5

CS 16 5/2

Pointers

(continues from lecture on 4/30)

Recall : pointers have the value of a memory address. The memory address that the pointer stores can have a value at it. This can be a little confusing. Remember that pointers hold memory addresses, not values at that address. You can access a value at a memory address by dereferencing a pointer.

Refer to code-from-class/05-02/pointers.cpp for the example given in class.

int\* p; // this is an example of a pointer to an int

int y;

p = &y; // this assigns the address of y to p.

// Remember p holds a memory address

\*p = 5; // this is an example of dereferencing a pointer. This accesses the value at the memory // address this changes the value of y

y++; // this changes the value of y from 5 to 6.

Warnings about pointers:

* Make sure you don’t dereference a pointer when you haven’t yet set to an address because you could end up with a segmentation fault (when you try to access memory you don’t have rights to).

Pointer to pointer

* Hold the address of another address that points to byte(s) of data.
* Next page(s) have an example of pointers to pointers

Sample file:

#include <iostream>

using namespace std;

int main()

{

int \*\* A;

int \* B;

int \* C;

int x, y;

x = 3;

cout << "x is " << x << endl;

y = 247;

cout << "y is " << y << endl;

cout << "\n\n";

B = &x;

cout << "We set B = &x" << endl;

cout << "A is at memory address (&A) == " << &A << endl;

cout << "This is the same as the address of x (&x) == " << &x << endl;

cout << "Notice that the derefereced B is the same value as x" <<

" (\*B) == " << \*B << endl;

cout << "\n\n";

C = &y;

cout << "We set C = &y" << endl;

cout << "C is at memory address (&C) == " << &C << endl;

cout << "This is the same as the address of y (&y) == " << &y << endl;

cout << "Notice that the derefereced C is the same value as y" <<

" (\*C) == " << \*C << endl;

cout << "\n\n";

A = &B;

cout << "We set A = &B" << endl;

cout << "A is at memory address (&A) == " << &A << endl;

cout << "Notice that the derefereced A is at the same memory address as B" <<

" (\*A) == " << \*A << endl;

cout << "Now if we dereference A twice we should get the value of x (which is \*B) (\*\*A) == "

<< \*\*A << endl;

cout << "\n\n";

A = &C;

cout << "One last time for clarity\n\nWe set A = &C" << endl;

cout << "A is at memory address (&A) == " << &A << endl;

cout << "Notice that the derefereced A is at the same memory address as C" <<

" (\*A) == " << \*A << endl;

cout << "Now if we dereference A twice we should get the value of y (which is \*C) (\*\*A) == "

<< \*\*A << endl;

return 0;

}

Output:

x is 3

y is 247

We set B = &x

A is at memory address (&A) == 0x7ffee698d9d0

This is the same as the address of x (&x) == 0x7ffee698d9bc

Notice that the derefereced B is the same value as x (\*B) == 3

We set C = &y

C is at memory address (&C) == 0x7ffee698d9c0

This is the same as the address of y (&y) == 0x7ffee698d9b8

Notice that the derefereced C is the same value as y (\*C) == 247

We set A = &B

A is at memory address (&A) == 0x7ffee698d9d0

Notice that the derefereced A is at the same memory address as B (\*A) == 0x7ffee698d9bc

Now if we dereference A twice we should get the value of x (which is \*B) (\*\*A) == 3

One last time for clarity

We set A = &C

A is at memory address (&A) == 0x7ffee698d9d0

Notice that the derefereced A is at the same memory address as C (\*A) == 0x7ffee698d9b8

Now if we dereference A twice we should get the value of y (which is \*C) (\*\*A) == 247

Pointers and Arrays

Refer to pointers\_and\_arrays.cpp for examples.

We can use pointer arithmetic to move through arrays. If we set a pointer with the same type as an array (**int** myArr[]; **int** \*p;) we can set p = arr. This notation (p = arr) means p stores the memory address of the first element of the array in p.

Now \*p == myArr[0]; // is true

We can then use p = p + 1; // so now \*p == myArr[1] is true

05/07

Structs and References

References explained in code:

Sample file:

#include <iostream>

using namespace std;

int **main**()

{

int num = 5;

cout <<"First we do \n\nint num = 5;\nint & myRef = num;\n" << endl << endl;

cout << "Num is the value (num) " << num << " and is at the address (&num) " << &num << endl;

int & myRef = num;

cout << "myRef has the value (myRef) " << myRef << " and is at address (&myRef) " << &myRef << endl;

cout << endl << endl << "Notice that these are the same values and address which is not true if we do something like the following:" <<

" \n\nint a = 5;\nint b = a;\n" << endl;

int a = 5;

cout << "a is the value (a) " << a << " and is at the address (&a) " << &a << endl;

int b = a;

cout << "b has the value (b) " << b << " and is at address (&b) " << &b << endl;

cout << endl << endl << "Notice that these are the same values BUT different addresses." << endl;

return 0;

}

Output:

First we do

int num = 5;

int & myRef = num;

Num is the value (num) 5 and is at the address (&num) 0x7ffee21df9f8

myRef has the value (myRef) 5 and is at address (&myRef) 0x7ffee21df9f8

Notice that these are the same values and address which is not true if we do something like the following:

int a = 5;

int b = a;

a is the value (a) 5 and is at the address (&a) 0x7ffee21df9ec

b has the value (b) 5 and is at address (&b) 0x7ffee21df9e8

Notice that these are the same values BUT different addresses.

Code from class has more examples.

Structs

Basically allow you to declare new data types which are composed of other data types (even other structs).

There are a couple of ways to declare instances of structs.

We could declare two instances of type box (a,b) like the sample file and output below:

Sample File:

#include <iostream>

#include <string>

using namespace std;

struct box{

int width;

int height;

string name;

} a, b;

int **main**()

{

cout << "Now we have two new instances of box!" << endl;

a.width = 2;

a.height = 5;

a.name = "Box1";

b.name = "Box2";

b.width = 1234;

b.height = 123;

cout << "Box a has attributes \n\nname: " << a.name << "\nheight: " << a.height << "\nwidth: " << a.width << endl << endl;

cout << "Box b has attributes \n\nname: " << b.name << "\nheight: " << b.height << "\nwidth: " << b.width << endl << endl;

return 0;

}

Output:

Now we have two new instances of box!

Box a has attributes

name: Box1

height: 5

width: 2

Box b has attributes

name: Box2

height: 123

width: 1234

**OR we can do something like this:**

#include <iostream>

#include <string>

using namespace std;

struct box{

int width;

int height;

string name;

};

int **main**()

{

box a = {2,5,"Box1"}, b;

cout << "Now we have two new instances of box!" << endl;

b.name = "Box2";

b.width = 1234;

b.height = 123;

cout << "Box a has attributes \n\nname: " << a.name << "\nheight: " << a.height << "\nwidth: " << a.width << endl << endl;

cout << "Box b has attributes \n\nname: " << b.name << "\nheight: " << b.height << "\nwidth: " << b.width << endl << endl;

return 0;

}

Which has the same output:

Now we have two new instances of box!

Box a has attributes

name: Box1

height: 5

width: 2

Box b has attributes

name: Box2

height: 123

width: 1234

Example with struct inside a struct:

#include <iostream>

#include <string>

using namespace std;

struct box{

int width;

int height;

string name;

};

struct jackInTheBox{

box jack;

int secondsUntilPop;

};

int **main**()

{

jackInTheBox a = {{2,5,"Box1"},500}, b;

b.jack.name = "Box2";

b.jack.width = 1234;

b.jack.height = 123;

b.secondsUntilPop = 501;

cout << "jackInTheBox a has attributes \n\njack.name: " << a.jack.name << "\njack.height: " << a.jack.height << "\njack.width: " << a.jack.width << endl << endl;

cout << "jackInTheBox b has attributes \n\njack.name: " << b.jack.name << "\njack.height: " << b.jack.height << "\njack.width: " << b.jack.width << endl << endl;

cout << "Additionally jackInTheBox a has " << a.secondsUntilPop << " seconds until pop." << "\n\n";

cout << "And jackInTheBox b has " << b.secondsUntilPop << " seconds until pop." << "\n\n";

return 0;

}

Output:

jackInTheBox a has attributes

jack.name: Box1

jack.height: 5

jack.width: 2

jackInTheBox b has attributes

jack.name: Box2

jack.height: 123

jack.width: 1234

Additionally jackInTheBox a has 500 seconds until pop.

And jackInTheBox b has 501 seconds until pop.

Pointer Arithmetic

int \*ptr, ptr1, ptr2 = // some valid address

**VALID**

ptr + 1 // does the same as the next line

1 + ptr // move to the next piece of memory

ptr – 1 // move to the previous piece of memory

ptr1 – ptr2 // find how many spaces these two ptrs are away from each other

ptr == ptr // if true these pointers at the same address

ptr == 0 // if true this pointer has not been assigned yet

ptr == NULL // this is equivalent to ptr == 0

**INVALID**

ptr + ptr // could be out of accessible memory

1 – ptr

1 == ptr

LLDB and GDB

To use: compile your program with the -g flag ( e.g. g++ -g FILENAME.cpp -o FILENAME ).

Then run with lldb FILENAME. Or gdb FILENAME

Use (run) to start the program (don’t put it in parentheses).

Use (print NAME\_OF\_VARIABLE) to print out values of local variables.

5/9/2019 CS 16 Lecture

Const keyword

Const is a modifier for types. Const is not a type on its own. (e.g. use const like : int const **or** const char **or** double const **or** const float (notice order doesn’t matter)).

Can always convert a non-const to a const.

Pointer arithmetic

Recall from the last lecture that pointer arithmetic is used with contiguous memory (arrays are contiguous memory).

So if we make a pointer point to the zeroth element of an array, we can use pointer arithmetic to move around the array.

Sample file:

#include <iostream>

#include <string>

using namespace std;

int **main**()

{

int \*point;

int arr[] = {123,234,345,456,567,678,789};

point = arr; *// Recall that the name of an array is a pointer to the zeroth element*

cout << "Here is the 0th element of the array using \*(point+0) : " << \*(point + 0) << "\n\n";

cout << "Here is the 3rd element of the array using \*(point+3) : " << \*(point + 3) << "\n\n";

cout << "We can also use pointer arithmetic to loop over an array!\n\n";

for(int \*myP = arr;(myP - arr) < 7; ++myP){ *// (myP - arr) < 7 becasue there are only 7 elements in arr*

*// myP - arr gives you the index you are at*

cout << "Element " << myP - arr << " is " << \*myP << endl;

}

return 0;

}

Output:

Here is the 0th element of the array using \*(point+0) : 123

Here is the 3rd element of the array using \*(point+3) : 456

We can also use pointer arithmetic to loop over an array!

Element 0 is 123

Element 1 is 234

Element 2 is 345

Element 3 is 456

Element 4 is 567

Element 5 is 678

Element 6 is 789

New keyword

We can dynamically allocate memory using the new keyword.

The new keyword puts things on the “heap”. Before today, we have been putting variables on the stack.

Basically, we create variables on the heap to be like global variables .

Delete keyword

We use this when we are done using a variable or array that we originally allocated in the heap.

As a general rule you should have a delete keyword for each new keyword used.

Memory Leak

A memory leak is when you allocate memory on the heap and then lose access to that piece of memory. This happens when you allocate memory and then no longer have a pointer to the piece of memory before you have deleted it.

Dangling Pointer

When you return a pointer that was allocated on the stack from a function.

Sample of dangling pointer: (when we return from makeArrayOfSize5() the array declared in the function is recalled, and not used how you want it to)

#include <iostream>

#include <string>

using namespace std;

int\* **makeArrayOfSize5**(){

int myArr[5] = {1,2,3,4,5};

int \*p = myArr; *// Recall that the name of an array is a pointer to myArr[0]*

return p;

}

int **main**()

{

int \*point = **makeArrayOfSize5**();

for( int i = 0; i < 5; ++i){

cout << "Element "<< i << " = " << \*(point + i) << "\n";

}

cout << endl << endl;

return 0;

}

Output: (we can see from the output that the values of myArr were erased once we were back in main)

Element 0 = 20053757

Element 1 = 1

Element 2 = -1511921808

Element 3 = 32767

Element 4 = 20053757

// End output

^^^^^^^^^^^^^^^^^^^

The above is not what we wanted (1,2,3,4,5)

New keyword example

Sample file: (the above file done the right way)

#include <iostream>

#include <string>

using namespace std;

int\* **makeArrayOfSize5**(){

int\* myArr = new int[5];

*// \*myArr = {1,2,3,4,5}; won't work*

for(int i = 0; i < 5; i++){

myArr[i] = i + 1;

}

int \*p = myArr; *// Recall that the name of an array is a pointer to myArr[0]*

return p;

}

int **main**()

{

int \*point = **makeArrayOfSize5**();

for( int i = 0; i < 5; ++i){

cout << "Element "<< i << " = " << \*(point + i) << "\n";

}

cout << endl << endl;

delete [] point; *// delete [] (which means the whole array) point (which is the array name)*

return 0;

}

Output:

Element 0 = 1

Element 1 = 2

Element 2 = 3

Element 3 = 4

Element 4 = 5

// This is what we wanted!

5/14 CS 16 Lecture 12

Infinite Inclusion

When you recursively include the same header file recursively.

Fix this by using #ifndef and #define at the top of the file and #endif at the end of the file.

See the code\_from\_class to see an example. (Remove the #ifndef, #define, and #endif to see the error).

Linked Lists – Not on the midterm

More on dynamic memory management (Using the heap)

Basically a review

Beginning of class, reviewed memory leaks and dangling pointers.

Linked List is a solution to the issue of arrays. Arrays are an issue because if you want to insert a value in the middle of an array you have to move all the elements after where you need to insert to the right.

Linked Lists are not contiguous in memory. Easy to add elements anywhere in the array.

Example of insertions into a linked list:

Sample File:

#include <iostream>

#include <string>

using namespace std;

struct LLNode{

int data;

LLNode\* nextNode;

}; *// a linked list*

int **main**()

{

*// Create a linked list: (123,456,567,789) then change it to (115,123,234,456,567,737,789,999)*

LLNode \*head = new LLNode; *// This keeps track of the start of the list*

LLNode \*zero = new LLNode;

LLNode\* one = new LLNode;

LLNode \* two = new LLNode;

LLNode \*three = new LLNode;

LLNode\* tail = new LLNode;

*// First set the data for all the elements*

zero->data = 123;

one->data = 456;

(\*two).data = 567; *// Notice we can use the -> or (\*). to access*

three->data = 789;

*// Now lets connect the linked list*

head->nextNode = zero;

(\*zero).nextNode = one;

one->nextNode = two;

two->nextNode = three;

three->nextNode = NULL; *// This one is the end of the list so we set it to NULL*

tail-> nextNode = three;

cout << "Printing the new linked list: \n\n";

for(LLNode\* i = head->nextNode; i != NULL; i = i->nextNode){

if(i == head->nextNode) cout << "Linked list: ";

cout << i->data << "->";

if(i->nextNode == NULL) cout << 'X';

}

cout << "\n\n";

*//Now lets modify the linked list to be (115,123,234,456,567,737,789,999)*

LLNode\* aNode0 = new LLNode; *// This will later store 115*

LLNode\* aNode1 = new LLNode; *// This will later store 234*

LLNode\* aNode2 = new LLNode; *// This will later store 737*

LLNode\* aNode3 = new LLNode; *// This will later store 999*

*// We could set the nextNode first, but we're doing it this way to better follow the logic*

aNode0->data = 115; *// Now this stores 115*

aNode1->data = 234; *// Now this stores 234*

aNode2->data = 737; *// Now this stores 737*

aNode3->data = 999; *// Now this stores 999*

*// Now set the head to 115 instead*

head->nextNode = aNode0;

aNode0->nextNode = zero; *// Set the node we are inserting's next node to the node after head*

cout << "\nPrint after inserting 115\n\n" ;

for(LLNode\* i = head->nextNode; i != NULL; i = i->nextNode){

if(head->nextNode == i) cout << "Linked list: ";

cout << i->data << " ";

if(i->nextNode == NULL) {

cout << "\n\n";

}

}

*// Now put 234 in between 123 (zero) and 456 (one)*

aNode1->nextNode = zero->nextNode; *// Node after 234 is 456 now*

zero->nextNode = aNode1; *// Node after 123 is 234 now*

cout << "\nPrint after inserting 234\n\n" ;

for(LLNode\* i = head->nextNode; i != NULL; i = i->nextNode){

if(head->nextNode == i) cout << "Linked list: ";

cout << i->data << " ";

if(i->nextNode == NULL) {

cout << "\n\n";

}

}

*// Now put 737 in between 567 (two) and 789 (three)*

aNode2->nextNode = two->nextNode; *// Node after 737 is 789 now*

two->nextNode = aNode2; *// Node after 567 is 737 now*

cout << "\nPrint after inserting 737\n\n" ;

for(LLNode\* i = head->nextNode; i != NULL; i = i->nextNode){

if(head->nextNode == i) cout << "Linked list: ";

cout << i->data << " ";

if(i->nextNode == NULL) {

cout << "\n\n";

}

}

*// Now put 999 after 789 (three)*

three->nextNode = aNode3; *// Node after 789 is 999 now*

aNode3->nextNode = NULL; *// 999's next node indicates the end of the linked list*

cout << "\nPrint after inserting 999\n\n" ;

for(LLNode\* i = head->nextNode; i != NULL; i = i->nextNode){

if(head->nextNode == i) cout << "Linked list: ";

cout << i->data << " ";

if(i->nextNode == NULL) {

cout << "\n\n";

}

}

return 0;

}

Output:

Printing the new linked list:

Linked list: 123->456->567->789->X

Print after inserting 115

Linked list: 115 123 456 567 789

Print after inserting 234

Linked list: 115 123 234 456 567 789

Print after inserting 737

Linked list: 115 123 234 456 567 737 789

Print after inserting 999

Linked list: 115 123 234 456 567 737 789 999

CS 16 5/21

**Recursion**

* Fractals are examples of recursion.
* The last recursive call is called the base case.
* Otherwise, we are in the recursive case which is solving a smaller subproblem of the larger problem.

Sample File:

#include <iostream>

#include <string>

using namespace **std**;

struct LLNode{

int data;

LLNode\* nextNode;

}; *// a mode of a linked list*

struct LL{

LLNode\* head;

LLNode \*tail;

}; *// a linked list*

int **sumOfLinkedList**(LLNode\* *l*){

if(l->nextNode == NULL)

{

return l->data;

}

else{

*// RECURSIVE CASE*

return l->data + **sumOfLinkedList**(l->nextNode);

}

}

int **main**()

{

*// Create a linked list: (123,456,567,789) then change it to (115,123,234,456,567,737,789,999)*

LLNode \*head = new LLNode; *// This keeps track of the start of the list*

LLNode \*zero = new LLNode;

LLNode\* one = new LLNode;

LLNode \* two = new LLNode;

LLNode \*three = new LLNode;

LLNode\* tail = new LLNode;

*// First set the data for all the elements*

zero->data = 123;

one->data = 456;

(\*two).data = 567; *// Notice we can use the -> or (\*). to access*

three->data = 789;

*// Now lets connect the linked list*

head->nextNode = zero;

(\*zero).nextNode = one;

one->nextNode = two;

two->nextNode = three;

three->nextNode = NULL; *// This one is the end of the list so we set it to NULL*

tail-> nextNode = three;

*//Now lets modify the linked list to be (115,123,234,456,567,737,789,999)*

LLNode\* aNode0 = new LLNode; *// This will later store 115*

LLNode\* aNode1 = new LLNode; *// This will later store 234*

LLNode\* aNode2 = new LLNode; *// This will later store 737*

LLNode\* aNode3 = new LLNode; *// This will later store 999*

*// We could set the nextNode first, but we're doing it this way to better follow the logic*

aNode0->data = 115; *// Now this stores 115*

aNode1->data = 234; *// Now this stores 234*

aNode2->data = 737; *// Now this stores 737*

aNode3->data = 999; *// Now this stores 999*

*// Now set the head to 115 instead*

head->nextNode = aNode0;

aNode0->nextNode = zero; *// Set the node we are inserting's next node to the node after head*

*// Now put 234 in between 123 (zero) and 456 (one)*

aNode1->nextNode = zero->nextNode; *// Node after 234 is 456 now*

zero->nextNode = aNode1; *// Node after 123 is 234 now*

*// Now put 737 in between 567 (two) and 789 (three)*

aNode2->nextNode = two->nextNode; *// Node after 737 is 789 now*

two->nextNode = aNode2; *// Node after 567 is 737 now*

*// Now put 999 after 789 (three)*

three->nextNode = aNode3; *// Node after 789 is 999 now*

aNode3->nextNode = NULL; *// 999's next node indicates the end of the linked list*

LL\* list = new LL;

list->head = head;

list->tail = tail;

cout << "Printing the linked list: \n\n";

for(LLNode\* i = head->nextNode; i != NULL; i = i->nextNode){

if(i == head->nextNode) cout << "Linked list: ";

cout << i->data << "->";

if(i->nextNode == NULL) cout << 'X';

}

cout << "\n\n";

*// Sum should be 115+123+234+456+567+737+789+999 = 4020*

cout << "Sum of the linked list is " << **sumOfLinkedList**(list->head) << "\n\n";

return 0;

}

Output:

Printing the linked list:

Linked list: 115->123->234->456->567->737->789->999->X

Sum of the linked list is 4020

* Notice that the only recursive part of this is the function sumOfLinkedList(). This takes the first node in the linked list as the parameter and propagates through the linked list from there.
* Alternatively you could use this function to sum a contiguous subset which concludes with the tail of the linked list.

**Helper Functions**

* Sometimes recursion can be tricky
* This makes us want more abstraction
* We don’t have to think the whole problem through at once, so we do it in parts with helper functions

Sample file of helper function:

#include <iostream>

#include <string>

using namespace **std**;

*// STRUCT AND FUNCTION DECLARATIONS (definitions are below main)*

struct LLNode{

int data;

LLNode\* nextNode;

}; *// a mode of a linked list*

struct LL{

LLNode\* head;

LLNode \*tail;

}; *// a linked list*

int **sumOfLinkedList**(LLNode\* *p*);

int **sumMyList**(LL\* *l*);

void **printLL**(LL\**l*);

int **main**()

{

*// Create a linked list: (123,456,567,789) then change it to (115,123,234,456,567,737,789,999)*

*// YOU SHOULD IGNORE THE FOLLOWING UNTIL THE FUNCTION CALL: printLL(list);*

LL\* list = new LL;

LLNode \*zero = new LLNode;

LLNode\* one = new LLNode;

LLNode \* two = new LLNode;

LLNode \*three = new LLNode;

LLNode\* tail = new LLNode;

LLNode\* aNode0 = new LLNode;

LLNode\* aNode1 = new LLNode;

LLNode\* aNode2 = new LLNode;

LLNode\* aNode3 = new LLNode;

aNode0->data = 115;

aNode1->data = 234;

aNode2->data = 737;

aNode3->data = 999;

zero->data = 123;

one->data = 456;

(\*two).data = 567;

three->data = 789;

list->head = zero;

(\*zero).nextNode = one;

one->nextNode = two;

two->nextNode = three;

three->nextNode = NULL;

list->tail = three;

list->head = aNode0;

list->head->nextNode = zero;

aNode1->nextNode = zero->nextNode;

zero->nextNode = aNode1;

aNode2->nextNode = two->nextNode;

two->nextNode = aNode2;

list->tail->nextNode = aNode3;

list->tail = aNode3;

aNode3->nextNode = NULL;

*// RESUME READING HERE*

**printLL**(list);

*// Sum should be 115+123+234+456+567+737+789+999 = 4020*

*// cout << "Sum of the linked list is " << sumOfLinkedList(list->head) << "\n\n";*

*// USE THE HELPER FUNCTION*

cout << "Sum of the linked list is " << **sumMyList**(list) << "\n\n";

return 0;

}

int **sumOfLinkedList**(LLNode\* *p*){

if(p->nextNode == NULL)

{

return p->data;

}

else{

*// RECURSIVE CASE*

return p->data + **sumOfLinkedList**(p->nextNode);

}

}

int **sumMyList**(LL\* *l*){

return **sumOfLinkedList**(l->head);

}

void **printLL**(LL\* *l*){

cout << "\n\nPrinting the linked list: \n\n";

for(LLNode\* i = l->head; i != NULL; i = i->nextNode){

if(i == l->head) cout << "Linked list: ";

cout << i->data << "->";

if(i->nextNode == NULL) cout << 'X';

}

cout << "\n\n";

}

Output:

Printing the linked list:

Linked list: 115->123->234->456->567->737->789->999->X

Sum of the linked list is 4020

* Notice that this code uses helper function sumMyList(LL\* l);
* We also use encapsulation by defining a function printLL