Course Review

This document summaries lecture from March 26 to April 21. This document covers the following topic: pointer, 2D-array, recursion function, stack memory and heap memory, detailed comments on linkedlist example(the example provide in Professor’s lecture), basic operations on linkedlist.

This document summarizes lectures from Feb 21 to March 26. Before Feb 21, our lectures covered the following topics: variables, operators, truth table, if statement, switch statement, for loop, while loop, function.

From Feb 21 to March 26, our lecture covers the following topic: struct, class, binary and hexadecimal, fstream, array, pointer, comparison between pass by value, pass by pointer and pass by reference.

# Pointer

The basic idea is pointer is a variable, but the value of this variable is an address. Here we need to understand some basic concept clearly and exactly.

To show the basic concept, we first declare a variable int i = 5

A screenshot of a spreadsheet

Description automatically generated

**Declare a pointer**:

int \* p

**Assign value to a pointer:**

p=&I (here we store 0002 in p)



**The value of a pointer:**



cout << p << endl; //output 0002



**The address of the pointer:**



cout << &p << endl; // output 0009



**Dereferencing a pointer(or pointer dereference):**



cout << \*p << endl; // cout 5



A pointer points to some place means the value of the pointer is the address of the place. In this case, when we say the pointer p points to variable I, we are saying the value of p is the address of i.

# array

* 1. character array

when we talk about character array, we are talking about strings. Strings are stored as a character arrays.

#include<iostream> // #include<sctio>

#include<cstring>

#include<cstdlib>

using namespace std;

void UpperCase(char \*);

int main()

{

// in c++, when we talk about char array, we are essentially talking about string

// string are stored as char array

char mystr[64]; // define a char array, named str

mystr[0] = 'c'; // store character in the char array mystr

mystr[1] = 'a'; // store character in the char array mystr

mystr[2] = 't'; // store character in the char array mystr

mystr[3] = '\0'; // by convention, string terminates with NULL. Here '\0' indicate the end of char array mystr

// print char array mystr

printf("str3: %s\n", mystr);

// to show string is an character array,we define a character pointer points to char array mystr

char \* c = mystr;

printf("c is: %p\n", c); // now c points to char array mystr, so the output is the address of mystr

printf("first element in array is(by dereference): %c \n", \*c); // output c

printf("second element in array is(by dereference): %c \n", \*(c+1)); // output a

printf("third element in array is(by dereference): %c \n", \*(c+1)); // output t

// call UpperCase function to convert "cat" into "CAT"

UpperCase(mystr); //

printf("str3: %s\n", mystr);

return 0;

}

void UpperCase(char\* s) // review line 22 to 26, we define a char pointer c and go through the mystr by dereferring (\*c)

{ // SO HERE the paramter of UpperCase is a character pointer (char \* s)

if (s == NULL) // we ALWAYS need to check the pointer, if the pointer points to NULL ,return

{

return;

}

while(\*s != '\0') // review '\0' indicate the end of the string, \*s means we are dereferring s,\*c, \*(c+1), \*(c+1) in line 23 to line 26

{

if (\*s >= 'a' && \*s <= 'z') // ascII a to z is continuous region 'a' -'a' = 0 'b' -'a' = 1

{

\*s = (\*s - 'a') + 'A';

}

s++;

}

}

* 1. 2-D array

The basic idea of 2-d array is 2-d array is an array of array. See the following example and note the printing shape. The example shows we can initialize an array and print it as any shape we want.

#include<iostream>

using namespace std;

int main()

{

int array2d[2][3];

int count=1;

for (int i=0; i<2; i++){

for (int j=0; j<3; j++){

array2d[i][j]=count;

count++;

}

}

// if we want to see each element, we can print them one by one

cout << "print each element: " << endl;

cout << "array2d[0][0]" << array2d[0][0] << endl;

cout << "array2d[0][1]" << array2d[0][1] << endl;

cout << "array2d[0][2]" << array2d[0][2] << endl;

cout << "array2d[1][0]" << array2d[1][0] << endl;

cout << "array2d[1][1]" << array2d[1][1] << endl;

cout << "array2d[1][2]" << array2d[1][2] << endl;

// if we want to print them as one column, we can write code like this

cout << "print all the elements in one column: " << endl;

for (int i=0; i<2; i++){

for (int j=0; j<3; j++){

cout << array2d[i][j] << " " << endl;

}

}

// if we want to print them as 2-d shape, we can write code like this

cout << "print all the element as 2-d shape: " << endl;

for (int i=0; i<2; i++){

for (int j=0; j<3; j++){

cout << array2d[i][j] << " ";

}

cout << endl;

}

}

The output of the above code is:

A screenshot of a computer

Description automatically generated

2-d array is 2-d array is an array of array. For the above example, we initialize int array2d[2][3], we can image in the memory the elements are stored like this:

A yellow and black text

Description automatically generated

Based on the figure, It become a little easier to understand 2-d array indexing.

#include<iostream>

#include<ctime>

using namespace std;

int main(){

// declare a 2-d array

int array2d[2][3];

int count=1;

// initialize the array

for (int i=0; i<2; i++){

for (int j=0; j<3; j++){

array2d[i][j]=count;

count++;

}

}

for (int i=0; i<2; i++){

for (int j=0; j<3; j++){

cout << array2d[i][j] << " ";

}

cout << endl;

}

// Note from here, 2-d array indexing

int \*p; // declare a pointer. recall that array is a pointer. if we cout << array2d, it will output an address.

// p is an integer pointer here

p = (int \*)array2d; // because p is an integer pointer, we need to cast array2d into integer pointer by using (int \*)

// Now p points to the first address of array2d

// the first address of the array is also the first address of the first row

// the first address of the array is also the address of first element in the array

cout << "address of array2d: " << array2d << endl; // 0x7ff7b183d820 equivalent to "200" in the figure

cout << "the value of p is: " << p << endl;// 0x7ff7b183d820 equivalent to "200" in the figure

cout << "the address of the first element: " << &array2d[0][0] << endl; // 0x7ff7b183d820 equivalent to "200" in the figure

// we can set the pointer to any place we want and do pointer arithmetic

// here set the pointer to second row.

// when we set the pointer to the second row, the pointer points to the first element in the second row

p = (int \*)(array2d+1); // array2d+1 equivalent to 212 in the figure

cout << "dereferencing p: " << \*p << endl; // output 4

cout << "array2d[1][0]: " << array2d[1][0] << endl; // output 4

cout << "address of 'array2d+1' : " << p << endl; // 0x7ff7b70c482c

cout << "address of array2d[1][0]: " << & array2d[1][0] << endl; // 0x7ff7b70c482c

// we also can set the pointer to a specific row using the following code

// here set the pointer to first row.

// when we set the pointer to the first row, the pointer points to the first element in the first row

p = (int \*)array2d[0]; // array2d[0] equivalent to 200 in the figure

cout << "dereferencing p: " << \*p << endl; // output 1

cout << "array2d[0][0]: " << array2d[0][0] << endl; // output 1

cout << "address of 'array2d[0]' : " << p << endl; // 0x7ff7bf262820

cout << "address of array2d[0][0]: " << & array2d[0][0] << endl; // 0x7ff7bf262820

// we can to pointer arithmathic across columns in the current row

// make the pointer points to the second element in the first row

p++;

cout << "dereferencing p: " << \*p << endl; // output 2

cout << "array2d[0][0]: " << array2d[0][1] << endl; // output 2

cout << "address of 'array2d[0][1]' : " << p << endl; // 0x7ff7bb5a0824 equivalent to 204 in the figure

cout << "address of array2d[0][1]: " << & array2d[0][1] << endl; // 0x7ff7bb5a0824 equivalent to 204 in the figure

}

# Recursion

The basic idea in recursion is a function call itself. The classical example is factorial function. Whenever we can use recursion, we can converse recursion into iteration. The following example is factorial function.

#include <iostream>

using namespace std;

int factorialRecursion(int);

int factorialMultiple(int);

int main()

{

cout << "The factorialRecursion of 5 is: " << factorialRecursion(5) << endl;

cout << "The factorialMultiple of 5 is: " << factorialMultiple(5) << endl;

return 0;

}

int factorialRecursion(int n)

{

if (n <= 1) // base case, terminating condition

return 1;

else

return n \* factorialRecursion(n-1); // recursive case

// 5 \* factorialRecursion(4)

// 4 \* factorialRecursion(3)

// 3 \* factorialRecursion(2)

// 2 \* factorialRecursion(1)

// 1 reach the base case return 1

// multiply back

// 2\*1

// 3 \* 2 \* 1

// 4 \* 3 \* 2 \* 1

// 5 \* 4 \* 3 \* 2 \* 1

}

int factorialMultiple(int n)

{

int res = 1;

for (int i=1; i <= n; i++)

{

res \*= i;

}

return res;

}

Note in this example, when we run factorialRecursion(5), the program call factorialRecursion() 4 times; At the fourth time that the program call factorialRecursion(), n becomes 1, and the program returns 1; and then 1 will multiply back with all the previous values (5, 4, 3, 2) , and returns the final result 120.

Recursion function keeps calling itself before it reaches its base case; And get the return value from base case; and then executes back to the starting point.

# Stack memory and heap memory

This code is from lecture, I added comments in the code. Beside the stack memory and heap memory, please also note the pointer deference.

/\*

The first part of this code (line to ) gives explanation and comparison between stack memory and heap memory

Stack and Heap are two memory parts in computer memory.

On the stack, memory allocation and deallocation are automatically by Operating System;

On the heap, we need "new" operator to allocate memory on heap and need "delete" operation to relinquish

the heap memory Manually.

Note some basic concepts:

pointer is a variable, its values is an address

when we say "a pointer points to someplace", it means the value stored in the pointer is the address of someplace

\*/

#include<iostream>

#include<cstdlib>

#include<cstring>

using namespace std;

// Here we declare a Self-Referential class. Recall class is a valid aggregation type in C++

// Self-Referential class is a special class, specially for Linkedlist implementation

// Note: To Create a Self-Referential Class,

// we need to declare a data member as a pointer, which points to an object of the same class

class Playa

{

char \* tag;

int sernum;

Playa \* next; // 'next' is pointer. It points to the address of the instance of Playa

// i.e. 'next' stores the address of the instance Playa

public:

Playa(); // initialize instance of the Playa

~Playa(); // disstructor

void TagSet(char \* ); // set the tag for an instance of Playa. Note the parameter is a char \*

char \* TagGet(); // get the tag of an instance of Playa. Note the return type is a char \*, because the 'tag' is a char \*

void NextSet(Playa \* x); // This function sets the value of 'next', i.e. makes 'next' store the address of the next instance of Playa

Playa \* NextGet(); // This function gets the value of 'next', i.e. returns the address of the next instance of Playa

};

Playa::Playa() // construct

{

static int n = 1;

sernum = n++;

next = NULL; // for every instance of Playa, make sure the initial value of its 'next' should be NULL.

tag = new char[128]; // 'new' operator is used for allocate something from the HEAP

// here we use 'new' to find some memory in the heap, and store the address to pointer 'tag'

// 'tag' is a pointer

strcpy (tag, "-Mna\_serei-"); // copy string "-Mna\_serei-" to tag.

// Here note: the parameters in strcpy() should be pointers

// Here 'tag' is a pointer and "-Mna\_serei-" is also a pointer

}

Playa::~Playa() // in disstructor, we need to use 'delete' operation to relinquish the memory we have allocated in the heap

{ // Here the memory we have allocated in the heap refers to "tag = new char[128]" in constructor

cout << "Plays: " << sernum << " bub---bye..." << endl;

delete [] tag; // So here we need to delete the whole tag array

}

void Playa::TagSet(char \* newtag)

{

strcpy(tag, newtag);

}

char \* Playa::TagGet()

{

return tag;

}

void Playa::NextSet (Playa \* x) // Note the parameter in the function.The parameter is a Playa pointer, Playa \*

{

next = x;

}

Playa \* Playa::NextGet() // Note the return type in the function.The return type is a Playa pointer, Playa \*

{

return next;

}

int main()

{

// Here we create an instance of Playa p0. p0 is on the STACK

// the memory allocation and deallocation on stack are both automatical.

// the memory of p0 will be deallocated automatically when the funtion main() dies.

Playa p0;

// we can see the tag of the P0. Note here p0 is not a pointer. It is an instance of Playa.

printf("Tag of p0: %s\n\n", p0.TagGet());

// declare a poniter named p1. Its type is Playa \*, i.e. it points to an instance of Playa

// Note that p1 is on the STACK

Playa \* p1;

// declare two more pointers p2 and plays. The meaning is the same as p1

Playa \* p2;

Playa \* playas;

// Here Note: 'new' operator finds some memory in the heap, this memory stores an new instance of Playa

p1 = new Playa;

// we can see the tag of the new instance. Here note: p1 is a pointer

printf("Tag of P1 using ->: %s\n", p1->TagGet()); // to dereferring a pointer, we can use pointer->

printf("Tag of P1 using deference: %s\n\n", (\*p1).TagGet());// to dereferring a pointer, we can also use (\*pointer).

p1->TagSet((char \*)"p1p1p1"); // (char \*) cast "p1p1p1" to be a pointer.

printf("Tag of P1 after TagSet() using ->: %s\n", p1->TagGet()); // to dereferring a pointer, we can use pointer->

printf("Tag of P1 after TagSet() using deference: %s\n\n", (\*p1).TagGet());// to dereferring a pointer, we can also use (\*pointer).

// This code shows until now 'p1' and 'playas' points to different address

// we use "playas = p1;" to make 'p1' and 'playas' points to the same address

printf("Address p1 points to : %p address that playas points to : %p\n", p1, playas); // Here the two address are different

playas = p1; // assign the values in p1 (i.e. the address p1 points to) to the values of playas

// i.e. make p1 and playa point to the same address.

printf("Address p1 points to : %p address that playas points to : %p\n\n", p1, playas); // Here two address are the same

// create a new instance p2 and make p1 link to p2

p2 = new Playa(); // here use the constructor to create a new instance

(\*p2).TagSet((char\*)"p2p2p2p2"); // cast char constant to char pointer

p1->NextSet(p2); // recall the implemtation in NextSet() funcion. we take an address as input and assign this address the

// the pointer 'next'

// what we do here is store the address of p2 into variable 'next' of p1

// we can get P2 tag through p1

char \* s1 = p1 -> NextGet()-> TagGet(); // p1-> NextGet() get the value of 'next' of p1, i.e. the address of p2

// because the 'next' of p1 is a pointer, it points to the p2

// we can use ->TagGet() is equivalent to p2 -> TagGet()

printf ("Tag of p2 through p1: %s\n", s1);

char \* s2 = playas -> NextGet()-> TagGet(); // because p1 and playas point to the same place. we also can access the p2 through playas

printf ("Tag of p2 through playas: %s\n\n", s2);

p2 = NULL; // if we declare a pointer, we should make it point to NULL. Don't leave wild pionter around.

delete p1; // relinquish p1 memory backt to os

// here we first manually delete p1

cout << "test" << endl;

return 0; // When the function finishes, p0 will be relinquished automatically

}

# Detailed comments on linkedlist example in lecture

This code is also from Professor’s lecture, I add comments in the code. Besides this example of linkedlist, I provide a more detailed example showing how to add a new node to a linkedlist.

#include<iostream>

#include<cstdlib>

#include<cstring>

using namespace std;

class Playa

{

char \* tag;

int sernum;

Playa \* next; // 'next' is pointer. It points to the address of the instance of Playa

// i.e. 'next' stores the address of the instance Playa

public:

Playa(); // initialize instance of the Playa

~Playa(); // disstructor

void TagSet(char \* ); // set the tag for an instance of Playa. Note the parameter is a char \*

char \* TagGet(); // get the tag of an instance of Playa. Note the return type is a char \*, because the 'tag' is a char \*

void NextSet(Playa \* x); // This function sets the value of 'next', i.e. makes 'next' store the address of the next instance of Playa

Playa \* NextGet(); // This function gets the value of 'next', i.e. returns the address of the next instance of Playa

void Print();

};

Playa::Playa() // construct

{

static int n = 1;

sernum = n++;

next = NULL; // for every instance of Playa, make sure the initial value of its 'next' should be NULL.

tag = new char[128]; // 'new' operator is used for allocate something from the HEAP

// here we use 'new' to find some memory in the heap, and store the address to pointer 'tag'

// 'tag' is a pointer

strcpy (tag, "-Mna\_serei-"); // copy string "-Mna\_serei-" to tag.

// Here note: the parameters in strcpy() should be pointers

// Here 'tag' is a pointer and "-Mna\_serei-" is also a pointer

}

Playa::~Playa() // in disstructor, we need to use 'delete' operation to relinquish the memory we have allocated in the heap

{ // Here the memory we have allocated in the heap refers to "tag = new char[128]" in constructor

cout << "Plays: " << sernum << " bub---bye..." << endl;

delete [] tag; // So here we need to delete the whole tag array

}

void Playa::TagSet(char \* newtag)

{

strcpy(tag, newtag);

}

char \* Playa::TagGet()

{

return tag;

}

void Playa::NextSet (Playa \* x) // Note the parameter in the function.The parameter is a Playa pointer, Playa \*

{

next = x;

}

Playa \* Playa::NextGet() // Note the return type in the function.The return type is a Playa pointer, Playa \*

{

return next;

}

// Pay special attention to the Print() function. Here is a recursion

// This Print() function, printf () the tag first and then reach to recursion part

// in this case, printf("%s ->", tag) prints playa 22 first, then evaluate if (next != NULL), it is true now,

// so it goes to next -> Print(), the 'next' points to the playa 21, so it print playa 21

// and so on

// until the playa 1. the 'next' of playa 1 is NULL, the Print() function ends at playa 1

void Playa::Print(){

printf("%s ->", tag); // print the tag first

if (next != NULL) // traverse the linkedlist from the last element to the first element

{

next -> Print();

}

}

int main() {

// every thing is a pointer here

Playa\* p;

char strbuf[64];

Playa\* playas = NULL; // 'playas' denotes a list, We can think 'playas' as the head of the linkedlist.

// initialize it to NULL.

for (int i = 0; i < 23; i ++){

sprintf(strbuf, "%s.%d","Playa", i); // sprintf store strings into a buffer, here the buffer named 'strbuf'

// the following step is a typical process to create a linkedlist. There are four step to create a linkedlist

p = new Playa; // intialize the pointer, make it to a new instance in the heap

p->TagSet(strbuf); // set values to the new instance

p->NextSet(playas); // make the pointer variable 'next' of the intance points to the head of the linkedlist

playas = p; // update the head of the linkedlist to the new instance.

}

playas ->Print();

cout << endl;

cout << "test" << endl;

int n;

for ( n = 0,p = playas; p!=NULL; p = p->NextGet()) //

{n++;}

cout << "There are " << n << "playas in the list" << endl;

return 0;

}

# Operations on linkedlist

1. The first example

The first example is the most straight forward example, I don’t have any function or loop in the first example. It shows the basic operation step by step.

#include<iostream>

using namespace std;

class Node{

private:

int data;

Node \* next;

public: // when we define a linkedlist, we always need these methods.

// They are constructor, destructor, set value methods and get value methods

Node();

//~Node(); if the constructor allocate memory in heap, we need destructor.

int getData();

Node \* getNext();

void setData(int x);

void setNext(Node \* nextnode);

void printlist();

};

Node::Node(){ // whenever we create a new node, it wil have this default value

data = 999;

next = NULL;

}

int Node::getData(){

return data;

}

Node \* Node::getNext(){

return next;

}

void Node::setData(int x){

data = x;

}

void Node::setNext(Node \* nextnode){

next = nextnode;

}

void Node::printlist(){

cout << data << " ";

if (next != NULL)

{

next -> printlist();

}

}

int main(){

// create a pointer

Node \* n1; // Here note: we create a pointer

// we will manipulate instance of class Node through this pointer

// we will make this pointer points to a newly created instance of class Node,

// and then manipulate the instance (get value, set value... and other operation) through dereferring this pointer

n1 = new Node; // make this pointer points to a newly created instance of class Node,

cout << "the address n1 points to: " << n1 << endl;

cout << "n1 data: " << n1->getData() << endl; // output 999; get value of 'data' of the instance through dereferring pointer n1

cout << "n1 next: " << n1->getNext() << endl; // output 0; get value of 'next' of the instance through dereferring pointer n1

n1->setData(1);

cout << "n1 data: " << n1->getData() << endl; // output 1; get value of 'data' of the instance through dereferring pointer n1

cout << "n1 next: " << n1->getNext() << endl; // output 0; get value of 'next' of the instance through dereferring pointer n1

// create another pointer n2

Node \* n2;

n2 = new Node;

cout << "the address n2 points to: " << n2 << endl;

cout << "n2 data: " << n2->getData() << endl;

cout << "n2 next: " << n2->getNext() << endl;

n2->setData(2);

cout << "n2 data: " << n2->getData() << endl;

cout << "n2 next: " << n2->getNext() << endl;

// again create another pointer n3

Node \* n3;

n3 = new Node();

cout << "the address n3 points to: " << n3 << endl;

cout << "n3 data: " << n3->getData() << endl;

cout << "n3 next: " << n3->getNext() << endl;

n3->setData(3);

cout << "n3 data: " << n3->getData() << endl;

cout << "n3 next: " << n3->getNext() << endl;

// link these 3 Node instance one by one.

// Note we manipulate the Node instance through dereferring the pointer, which points to them

n1->setNext(n2); // here the list is 1 2 3

n2->setNext(n3); // here the list is 1 2 3

n1->printlist(); // here the list is 1 2 3

// Now we add a new node to the head of the list, make the list to 0 1 2 3

// create a new node

Node \* n0;

n0 = new Node();

n0->setData(0);

cout << "n3 data: " << n0->getData() << endl;

cout << "n3 next: " << n0->getNext() << endl;

// link it the head of the linkedlist

n0->setNext(n1);

// print the linkedlist

n0->printlist(); // output 0 1 2 3

// Now we add a new node to the end of the list, make the list to 0 1 2 3 4

// create a new node

Node \* n4;

n4 = new Node();

n4->setData(4);

cout << "n4 data: " << n4->getData() << endl;

cout << "n4 next: " << n4->getNext() << endl;

// link it the head of the linkedlist

n3->setNext(n4);

// print the linkedlist

n0->printlist(); // 0 1 2 3 4

// Now we add a new node after node 2

// create a new node

Node \* middle;

middle = new Node();

middle->setData(6666);

cout << "middle data: " << middle->getData() << endl;

cout << "middle next: " << middle->getNext() << endl;

// link it between node 2 and node 3

// \*\* important: we first make node middle link to node 3

// \*\* important: then second make node 2 link to node middle

// we can't do it in the opposite order

middle->setNext(n3);

n2-> setNext(middle);

// print the linkedlist

n0->printlist(); // 0 1 2 666 3 4

}

1. The second example

The second example shows how to add a node to the front of the a linkedlist and to the end of the linkedlist

#include<iostream>

using namespace std;

class Node{

private:

int data;

Node \* next;

public: // when we define a linkedlist, we always need these methods.

// They are constructor, destructor, set value methods and get value methods

Node();

//~Node(); if the constructor allocate memory in heap, we need destructor.

int getData();

Node \* getNext();

void setData(int x);

void setNext(Node \* nextnode);

void printlist();

};

void addToTail(Node \* , int ); // we put the head node of the list to function and traverse the list

// traverse the list until the end

// and put the new node at the end of the list

// so here Node \* indicate the head node of the list

Node::Node(){ // whenever we create a new node, it wil have this default value

data = 999;

next = NULL;

}

int Node::getData(){

return data;

}

Node \* Node::getNext(){

return next;

}

void Node::setData(int x){

data = x;

}

void Node::setNext(Node \* nextnode){

next = nextnode;

}

void addToTail(Node \* n, int data){

Node \* newNode;

newNode = new Node(); // initialize the new node

newNode->setData(data); // set its data

Node \* temp = n; // create a temp node, it now points to Node n,

// because when we call the function, we will put head as actual parameter of 'n'

// so here now we actually make temp points to head node of the list

if (n == NULL) { // if list is empty, newNode becomes the head

n = newNode;

}

while (temp->getNext() != NULL) { // traverse to the end of the list

temp = temp->getNext();

}

temp->setNext(newNode); // add newNode to the end of the list

}

void addToLocation(Node \*){

}

void InsertAfter(Playa \* Item, Playa \* NewPlaya)

{

if (Item == NULL || NewPlaya == NULL)

{ return; }

NewPlaya -> NextSet(Item->NextGet());

Item->NextSet(NewPlaya);

}

void Node::printlist(){

cout << data << " ";

if (next != NULL)

{

next -> printlist();

}

}

int main(){

Node \* head = NULL; // we need a pointer to store the address of the first element, we generally name it "head".

// it is the identifier of the linkedlist

// when we know the address of the first element, we can access the other element,

// because the elements are linked one by one

head = NULL;

Node \* n;

// in our lecture, we add a new node to a linkedlist by the following code

for (int i=1; i<=3; i++){

n = new Node();

n -> setData(i);

n -> setNext(head);

head = n;

}

head->printlist(); // 3 2 1

addToTail(head, 0);

addToTail(head, -1);

head->printlist(); // 3 2 1 0 -1

}

1. The third example

The third example is example code of lab12. This code shows basic operations of linkedlist.

#include <iostream>

using namespace std;

class Node {

private:

double data;

Node\* next;

public:

Node(double value);

Node\* getNext();

void setNext(Node\* node);

double getValue();

void setValue(double value);

void incValue();

int size();

double largest();

//double smallest();

double average();

void print();

void clear();

};

Node::Node(double value)

{

data = value;

next = NULL;

}

Node\* Node::getNext()

{

return next;

}

void Node::setNext(Node\* node)

{

next = node;

}

double Node::getValue()

{

return data;

}

void Node::setValue(double value)

{

data = value;

}

void Node::incValue()

{

Node\* current = next;

while (current != NULL) {

current->setValue(current->getValue() + 1);

current = current->getNext();

}

}

int Node::size()

{

int count = 1;

Node\* current = next;

while (current != NULL) {

count++;

current = current->getNext();

}

return count;

}

double Node::largest()

{

double largestValue = data;

Node\* current = next;

while (current != NULL) {

if (current->getValue() > largestValue) {

largestValue = current->getValue();

}

current = current->getNext();

}

return largestValue;

}

double Node::average()

{

double sum = 0;

int count = 1;

Node\* current = next;

while (current != NULL) {

sum += current->getValue();

count++;

current = current->getNext();

}

return sum / count;

}

void Node::print()

{

Node \* current = this;

while (current != NULL) {

cout << current->getValue() << endl;

current = current->getNext();

}

cout << endl;

}

void Node::clear()

{

while (next != NULL) {

Node\* temp = next;

next = next->getNext();

delete temp;

}

}

int main() {

Node\* head = new Node(5.5);

Node\* secondNode = new Node(10.3);

Node\* thirdNode = new Node(15.7);

// // Linking the nodes to create a linked list

head->setNext(secondNode);

secondNode->setNext(thirdNode);

// Testing methods

cout << "Linked list values: ";

head->print();

cout << "Size of the list: " << head->size() << endl;

cout << "Largest value in the list: " << head->largest() << endl;

cout << "Average value in the list: " << head->average() << endl;

cout << "Incrementing the first value: " << endl;

head->incValue();

cout << "Linked list values after increment: ";

head->print();

head->clear(); // Clearing the list to release memory

return 0;

}

The key points in linkedlist operation are:

1. To identify a linkedlist, we create a pointer, this pointer points to the address of first elements of the linkedlist. i.e. this pointer store the address of the first element of the linkedlist. That is why when we traverse a list, we always from the first element.
2. When we reach the node through dereferencing the pointer.
3. The most important thing is to clearly know where the pointer points to.