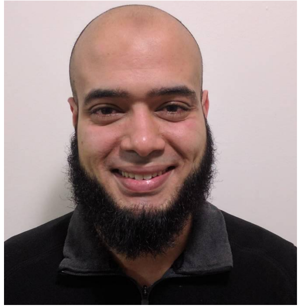
Data Structures

Effective Coding and Debugging for linked list

Mostafa S. Ibrahim
Teaching, Training and Coaching since more than a decade!

Artificial Intelligence & Computer Vision Researcher
PhD from Simon Fraser University - Canada
Bachelor / Msc from Cairo University - Egypt
Ex-(Software Engineer / ICPC World Finalist)



Draining your time

- Linked lists challenges
  - Involves several nodes deletion, insertion and **relinking**!
- Linked lists coding may drain your time
  - Several coding bugs
  - Several run-time errors
  - Hard to visualize
- Thinking time
  - It will also consume time to get the right idea, but this is the ok part
- Take this video very seriously
  - It provides coding and debugging tips to reduce the time waste

Before thinking/coding

- List several test cases and their answers
- Consider lists of different sizes / even-odd size
- Consider the problem and what might be tricky test cases
- This will make you aware of what kind of issues you need to handle
- Later, evaluate your code against all test cases
- Common mistakes
  - To not do that before thinking
  - To test over weak cases
  - We all may miss some good cases, but at least the code should work for several cases

Thinking skills

- KISS: Keep it simple stupid
  - Many problems seems hard, but you can do them. Calm down
  - Many problems can be coded elegantly in a few lines.
    - Sadly many instructors and websites take a longer path
  - Try to make things simple as possible
- Think on papers NOT on PC
  - A very common mistake is to rush to code your idea
  - Then waste hours with run-time errors and buggy outputs
  - Think deeper on papers. **Draw each step**. Use addresses if needed
  - Verify & trace different test cases
  - This will boost your abstract thinking skills, save your time and boost your confidence
- Finally: Compare with my codes. Learn from them!

Coding skills

- Think modular
  - Avoid writing a lengthy function (even with code comments)
  - Every time you notice something can be converted to a helpful method do it
  - Pick very clear names
  - The more you solve, you will notice old functions can be used!
  - Document ur functions: input, output and any conditions
- Tip: Before coding decide the minimum number of needed elements
  - 1 node? 2 nodes? 3 nodes?
  - If no enough nodes, typically no action

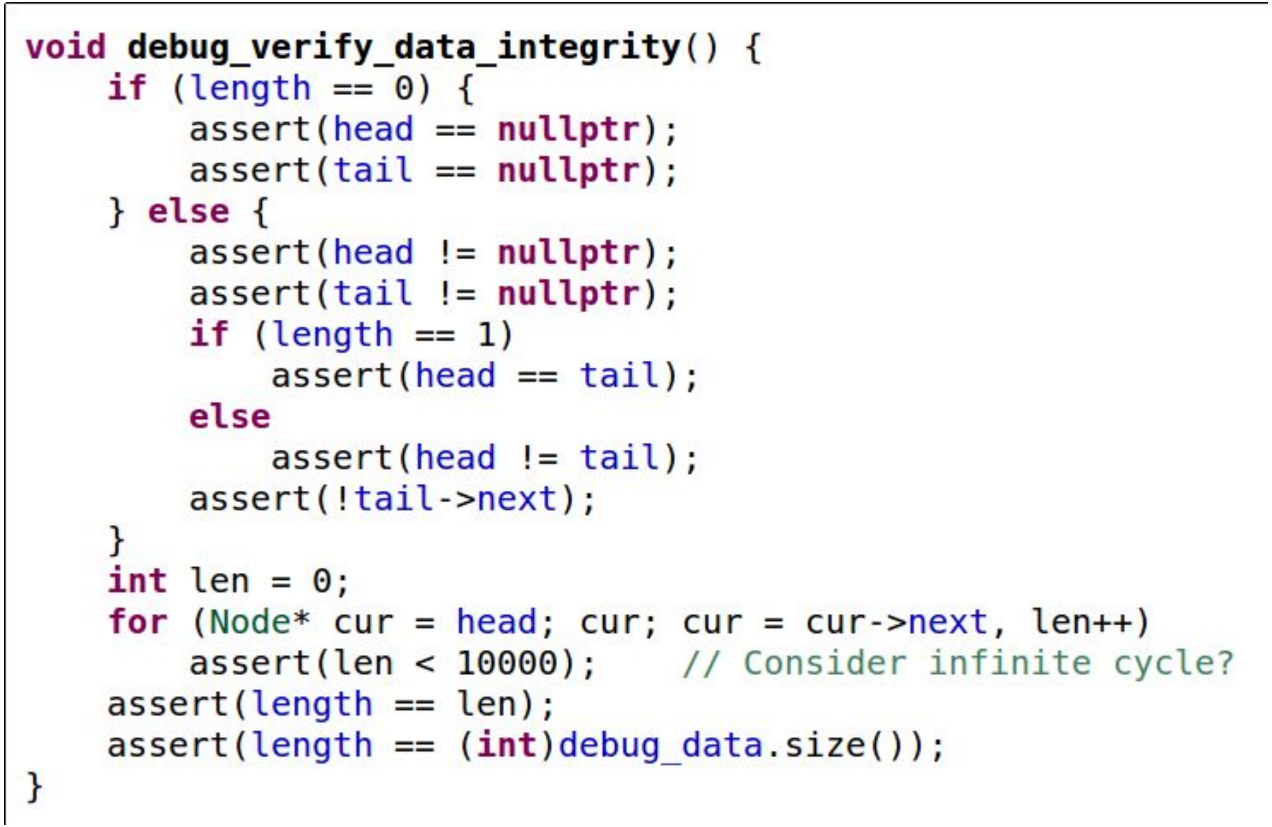
Coding mistakes

- Wrong algorithm
- Wrong order of operations
    - Sometimes you need to take a copy of a node’s next before cancelling its next
- Run time errors
    - node->next will case RTE if node is null
    - node->next->next will case RTE if node->next is null
    - Every Time you have them in your code, make sure
        - EIther logic is correct
        - You are verifying against null first

Data Integrity

- Data integrity is the overall accuracy, completeness, and **consistency** of data
- What is our data?
    - head, tail and length
        - E.g. Head and tail should be null if empty
        - E.g. Length must be really the length of the items
- Write a function that verifies a linked list is correct
- Run it after your main algorithm is done (or intermediate if possible)
    - It will catch too many mistakes!
- Take 10 minutes to code it

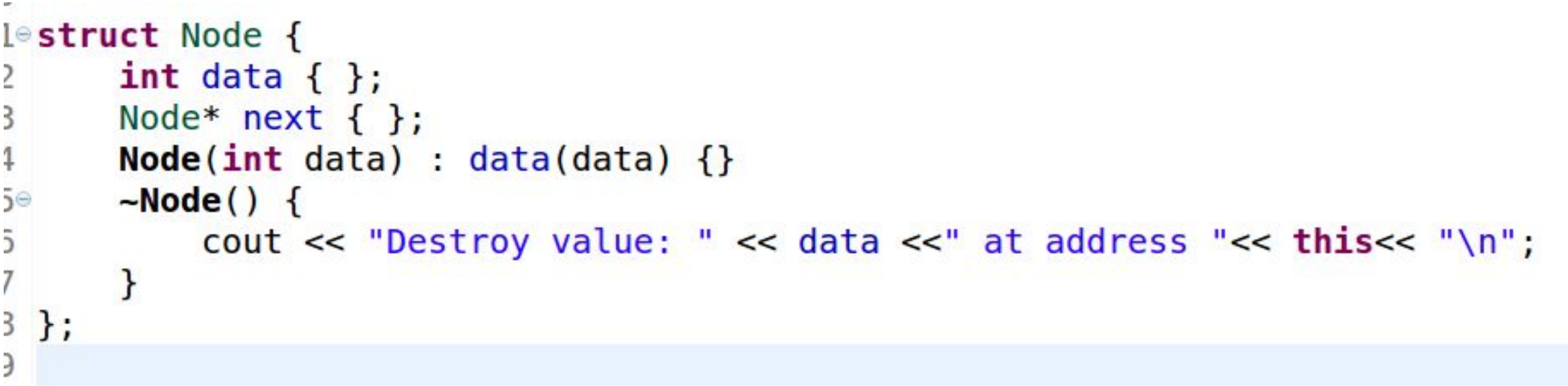
Data Integrity



```
void debug_verify_data_integrity() {
    if (length == 0) {
        assert(head == nullptr);
        assert(tail == nullptr);
    } else {
        assert(head != nullptr);
        assert(tail != nullptr);
        if (length == 1)
            assert(head == tail);
        else
            assert(head != tail);
        assert(!tail->next);
    }
    int len = 0;
    for (Node* cur = head; cur; cur = cur->next, len++)
        assert(len < 10000);    // Consider infinite cycle?
    assert(length == len);
    assert(length == (int)debug_data.size());
}
```

Node Destructor

- When we start deleting nodes (next sessions), it is good to double check what is removed

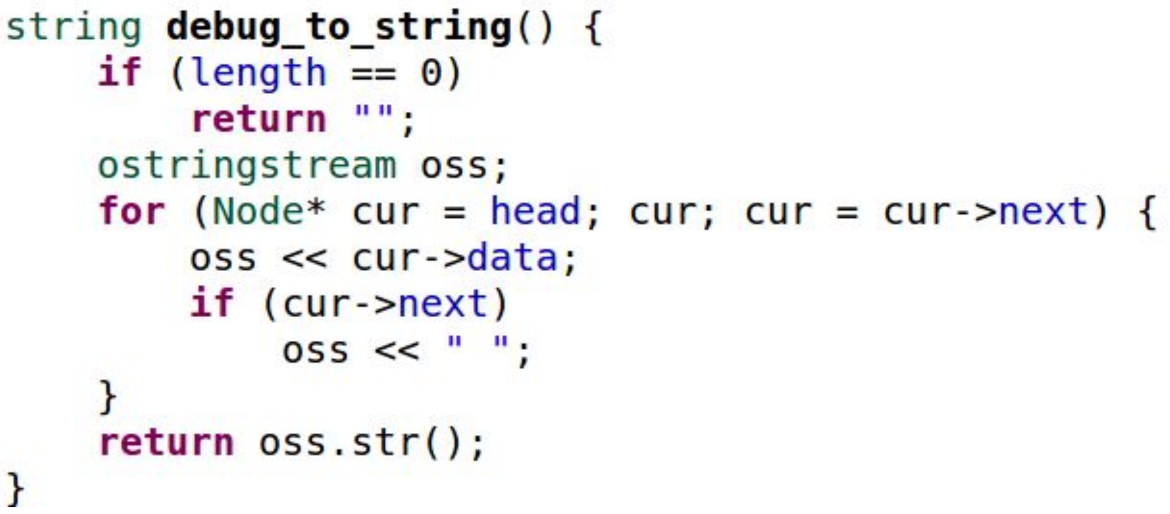


```
struct Node {
    int data { };
    Node* next { };
    Node(int data) : data(data) {}
    ~Node() {
        cout << "Destroy value: " << data <<" at address "<< this<< "\n";
    }
};

```

ToString for comparisons

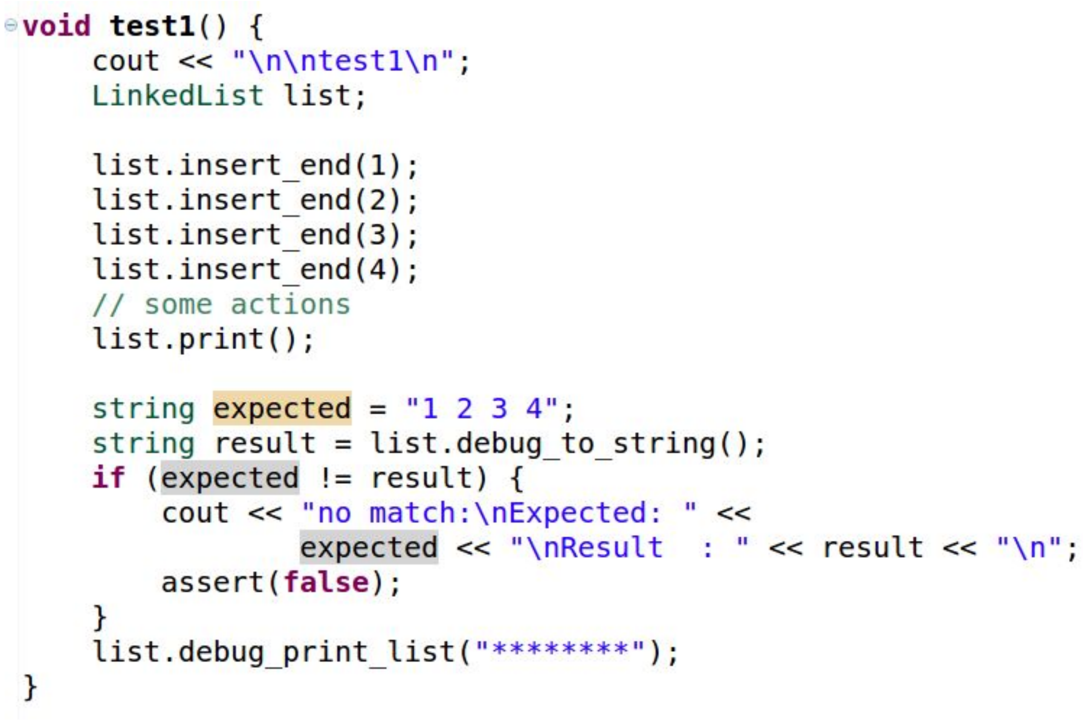
- To easily compare your function result vs expected output, let’s convert the function to string



```
string debug_to_string() {
    if (length == 0)
        return "";
    ostringstream oss;
    for (Node* cur = head; cur; cur = cur->next) {
        oss << cur->data;
        if (cur->next)
            oss << " ";
    }
    return oss.str();
}
```

Testing

- For each test case, develop its list & operations
- Update it
- Compare its content with expected output
- From main, run all your test functions

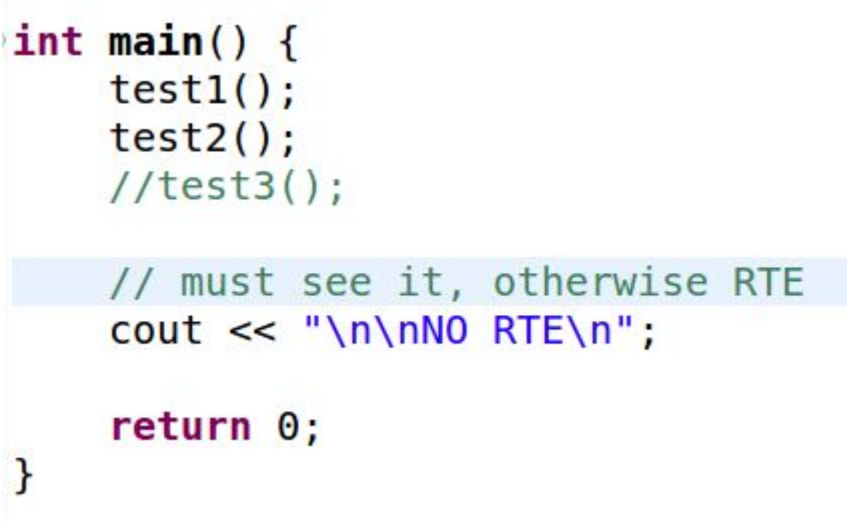


```
void test1() {
    cout << "\n\ntest1\n";
    LinkedList list;

    list.insert_end(1);
    list.insert_end(2);
    list.insert_end(3);
    list.insert_end(4);
    // some actions
    list.print();

    string expected = "1 2 3 4";
    string result = list.debug_to_string();
    if (expected != result) {
        cout << "no match:\nExpected: " <<
                expected << "\nResult  : " << result << "\n";
        assert(false);
    }
    list.debug_print_list("********");
}
```

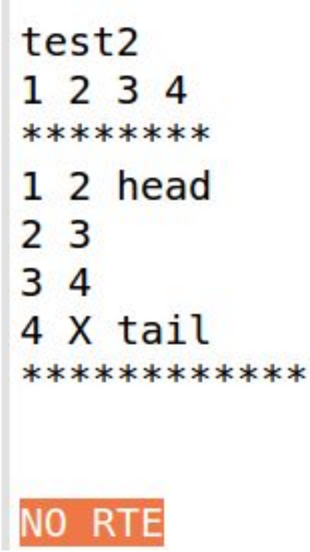
Testing: Observe RTE



```
int main() {
    test1();
    test2();
    //test3();

    // must see it, otherwise RTE
    cout << "\n\nNO RTE\n";

    return 0;
}
```

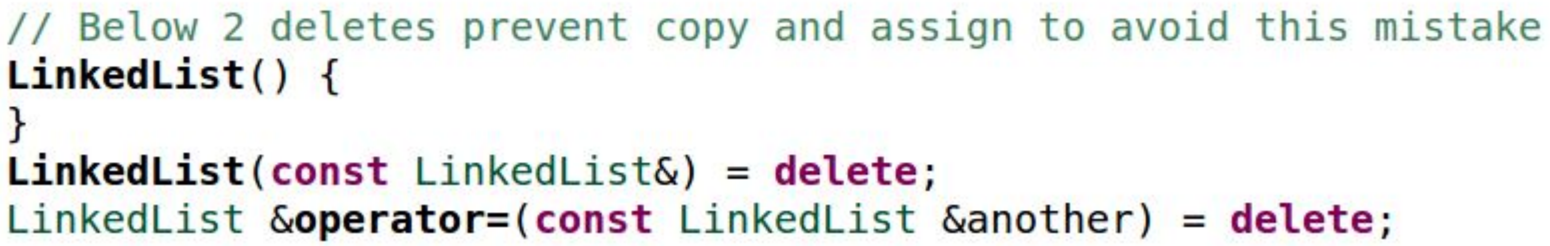


```
test2
1 2 3 4
********
1 2 head
2 3
3 4
4 X tail
************


NO RTE
```

Prevent crashes

- Most of the code doesn’t handle pointers copy correctly
  - It is not needed
- But students forget and do copy
- Below lines of code will prevent it
  - Prevent return Linked List from function
  - Or pass it without &
  - Ot assign it



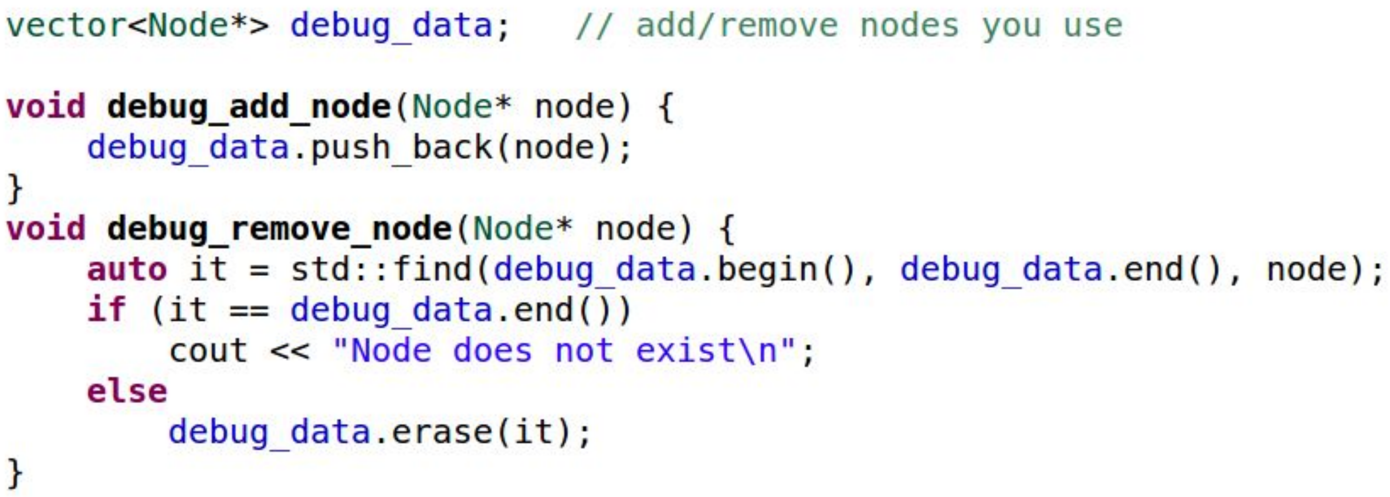
```
// Below 2 deletes prevent copy and assign to avoid this mistake
LinkedList() {
}
LinkedList(const LinkedList&) = delete;
LinkedList &operator=(const LinkedList &another) = delete;
```

Visualizing linked list

- In many cases, our links will be separated (-> next)
- We have hard time in printing
- A good way is to track the nodes and print their info
- Use a seperate vector to track current nodes
- Print the queue nodes itself not the linked list nodes

Visualizing linked list

- You don’t need to understand the details
  - Use these 2 functions to add/remove a node

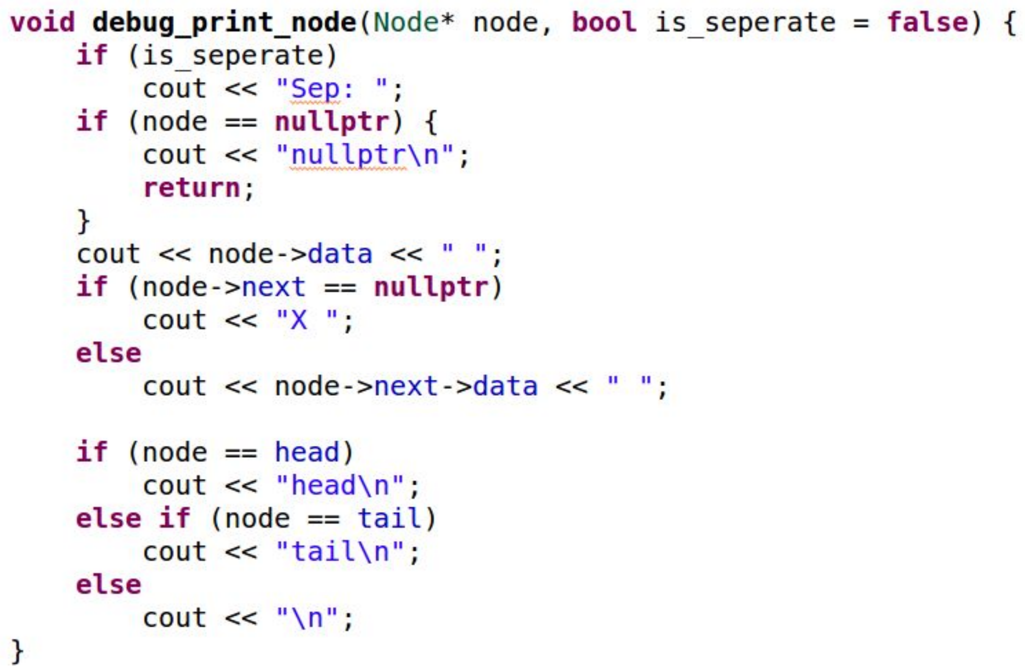


```
vector<Node*> debug_data;   // add/remove nodes you use

void debug_add_node(Node* node) {
    debug_data.push_back(node);
}
void debug_remove_node(Node* node) {
    auto it = std::find(debug_data.begin(), debug_data.end(), node);
    if (it == debug_data.end())
        cout << "Node does not exist\n";
    else
        debug_data.erase(it);
}
```

Printing a single node info

- This function print the node info
  - Its data
  - Is it head or tail
  - Its value and next value
- The function is called by another debug function
  - But u can call anytime



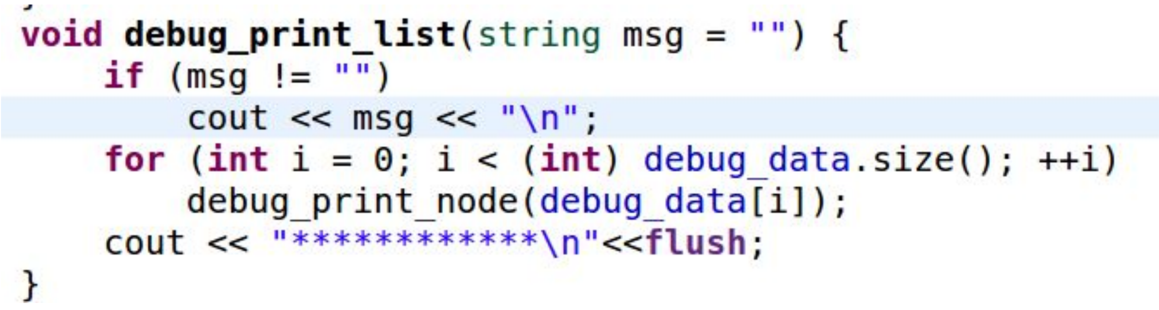
```cpp
void debug_print_node(Node* node, bool is_seperate = false) {
    if (is_seperate)
        cout << "Sep: ";
    if (node == nullptr) {
        cout << "nullptr\n";
        return;
    }
    cout << node->data << " ";
    if (node->next == nullptr)
        cout << "X ";
    else
        cout << node->next->data << " ";

    if (node == head)
        cout << "head\n";
    else if (node == tail)
        cout << "tail\n";
    else
        cout << "\n";
}
```

Printing all nodes

- Example:
- 1 2 head
- 2 3
- 3 4
- 4 X tail

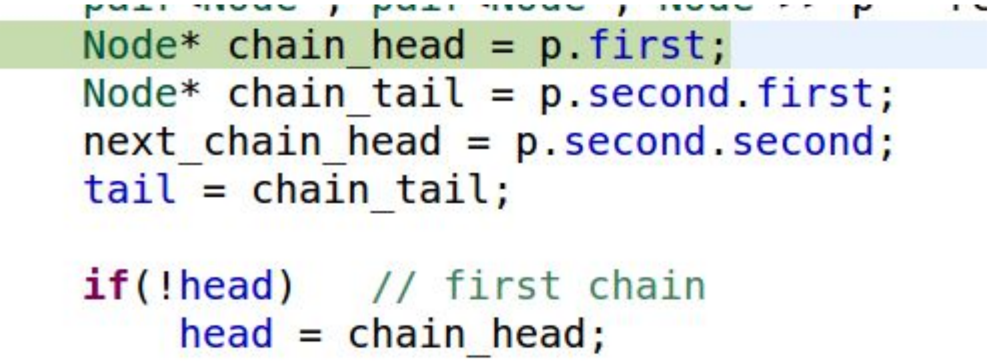
- Example:
- 3 4
- 4 X tail
- 1 2 head
- 2 3



```
void debug_print_list(string msg = "") {
    if (msg != "")
        cout << msg << "\n";
    for (int i = 0; i < (int) debug_data.size(); ++i)
        debug_print_node(debug_data[i]);
    cout << "************\n"<<flush;
}
```

Debugger

- Your IDE’s debugger will help you to discover also several mistakes!
- Make sure to be comfortable in using
- Prepare test case. Draw its steps.
- Run debugger and confirm the match!



```
Node* chain_head = p.first;
Node* chain_tail = p.second.first;
next_chain_head = p.second.second;
tail = chain_tail;

if(!head)   // first chain
    head = chain_head;
```

Code template

- Base ur algorithms on my code
- Utilize the debugging facilities in it

"Acquire knowledge and impart it to the people."

"Seek knowledge from the Cradle to the Grave."