

# CS010C

Lab3

# Lab2 demo

- Test your rule of 3
- Btw: Answer
  - Someclass A = B;
  - **copy constructor**
- Conclusion
  - `IntList list4 = list1;`
  - `IntList list3(list2);`
  - Syntactically equivalent
  - Has nothing to do with the Big 3

## Lab2 manually graded part

- Rule of Three: If implement 1, implement **the other 2** as well!
  - Destructor
    - `~IntList();`
    - Delete all data nodes
    - Delete dummyHead and dummyTail
  - Copy constructor
    - `IntList(const IntList &other);`
    - Initialize an empty list
    - Copy each element
  - Copy assignment operator
    - `IntList& operator=(const IntList &other);`
    - Clear current list
    - Copy elements from "other" list
  - Guess: `IntList list4 = list1; ???`
- To get **full credit**, you should include the other two as well
  - `IntList(const IntList &other); IntList& operator=(const IntList &other);`
  - Declaration & Definition & Implementation

# Main.cpp

- Just copy main.cpp
- Explanation
  - Implement `printListDetails(const IntList &list)`

```
void printListDetails(const IntList &list) {  
    IntNode *current = list.dummyHead->next;  
    while (current != list.dummyTail) {  
        cout << "Value: " << current->data << ", Address: " << current << endl;  
        current = current->next;  
    }  
}
```

- Modify `main()`
  - Use copy constructor
  - Use assignment operator

```
// Test copy constructor  
IntList copyList(list);  
cout << "Copy list (using copy constructor): " << copyList << endl;  
cout << "Copy list details (using copy constructor):" << endl;  
printListDetails(copyList);  
  
// Test copy constructor  
cout << "list: " << list << endl;  
cout << "list details:" << endl;  
printListDetails(list);  
  
// Test copy assignment operator  
IntList assignedList;  
assignedList = list;  
cout << "Assigned list (using operator=): " << assignedList << endl;  
cout << "Assigned list details (using operator=):" << endl;  
printListDetails(assignedList);
```

# Check 3 cases

- Checkbox

~	()	=
---	----	---

- Destructor
- Copy constructor
- Copy assignment operator

```
Deleting node with value: 20 at address: 0x228df10
Deleting node with value: 10 at address: 0x228def0
Near the end of main, before deleting
free(): double free detected in tcache 2
run.sh: line 1: 186468 Aborted (core dumped) ./a.out
```

```
list1 : 0x7fffaf6af110
Dummy Head | address: 0x9f2eb0
value: 20 | address: 0x9f2f10
value: 10 | address: 0x9f2ef0
Dummy Tail | address: 0x9f2ed0
```

```
Testing = operator
list2 : 0x7fffaf6af100
Dummy Head | address: 0x9f2eb0
value: 20 | address: 0x9f2f10
value: 10 | address: 0x9f2ef0
Dummy Tail | address: 0x9f2ed0
```

```
Testing copy constructor
list3 : 0x7fffaf6af0f0
Dummy Head | address: 0x9f2eb0
value: 20 | address: 0x9f2f10
value: 10 | address: 0x9f2ef0
Dummy Tail | address: 0x9f2ed0
```

```
Testing = at initialization
list4 : 0x7fffaf6af0e0
Dummy Head | address: 0x9f2eb0
value: 20 | address: 0x9f2f10
value: 10 | address: 0x9f2ef0
Dummy Tail | address: 0x9f2ed0
```

```
list5 : 0x9f3380
Dummy Head | address: 0x9f2eb0
value: 20 | address: 0x9f2f10
value: 10 | address: 0x9f2ef0
Dummy Tail | address: 0x9f2ed0
```

```
list1 : 0x7ffe27df0500
Dummy Head | address: 0x1b1aeb0
value: 20 | address: 0x1b1af10
value: 10 | address: 0x1b1aef0
Dummy Tail | address: 0x1b1aed0
```

```
Testing = operator
list2 : 0x7ffe27df04f0
Dummy Head | address: 0x1b1b340
value: 20 | address: 0x1b1b3a0
value: 10 | address: 0x1b1b380
Dummy Tail | address: 0x1b1b360
```


```
Testing copy constructor
list3 : 0x7ffe27df04e0
Dummy Head | address: 0x1b1b3c0
value: 20 | address: 0x1b1b420
value: 10 | address: 0x1b1b400
Dummy Tail | address: 0x1b1b3e0
```

```
Testing = at initialization
list4 : 0x7ffe27df04d0
Dummy Head | address: 0x1b1b440
value: 20 | address: 0x1b1b4a0
value: 10 | address: 0x1b1b480
Dummy Tail | address: 0x1b1b460
```

```
list5 : 0x1b1b4c0
Dummy Head | address: 0x1b1b4e0
value: 20 | address: 0x1b1b540
value: 10 | address: 0x1b1b520
Dummy Tail | address: 0x1b1b500
```

Lab3

# Description

- Arguments
  - $4 = 3 + 1$   the executable file itself
  - Dictionary
    - In each run, **only 1 dictionary** file is provided
- $k$ -characters dictionary
  - Insertion:  $k \rightarrow k + 1$
  - Removal:  $k \rightarrow k - 1$
  - Replacement:  $k \rightarrow k$
  - **Conclusion: only consider "replacement"**

```
✓ int main(int argc, char* argv[
✓   if (argc != 4) {
      cerr << "Usage error, expected 4 arguments\n";
      exit(1);
    }

    string dict_file = argv[1];
    string s2 = argv[2];
    string s3 = argv[3];
```

distance is used to measure the similarity between two strings. The "distance" refers to the amount of work, or cost of the operations needed to transform one sequence of characters into the other. Common operations can include: ~~replacement~~, ~~insertion~~ or ~~removal~~ of a character, insertion of one or more blank spaces.

# Hints

```
WordLadder(const string& listFile);  
void outputLadder(const string& start, const string& end);
```

- **class WordLadder**

- Only 1 member variable

```
list<string> dict;
```

- **WordLadder(const string& listFile);**

- Constructor: set up some member variables
  - Just initialize `dict` will be enough in this case
    1. File input
    2. Add words into the list: `push_back`

- **void outputLadder(const string& start, const string& end);**

- Stack
  - Consisting of word(s)
  - the path from the “starting” word to “current” word
- Queue
  - Consisting of the stack(s) above
  - (the paths to) **all words reachable** from the “starting” word

## Algorithm: Find Word Ladder

```
Create a stack of strings.  
Push the start word on this stack.  
Create a queue of stacks.  
Enqueue this stack.
```

# Too early at this stage

10.9 Lab X: Graph DFS

10.10 Extra Credit

10.11 SUM24 Lab 6: Graph and **BFS**

- BFS & DFS are in section 10
- Just simply use the algorithm provided on zyBook
- Hint: eliminate words that are already present
  - Where?
  - How?

## Algorithm: Find Word Ladder

```
Create a stack of strings.
Push the start word on this stack.
Create a queue of stacks.
Enqueue this stack.

while the queue is not empty
  for each word in the dictionary
    if a word is exactly 1 letter different than the top string of the front stack
      then
        if this word is the end word
          then
            word ladder found, it is the front stack plus the end word
        else
          Make a copy of the front stack.
          Push the found word onto the copy.
          Enqueue the copy.
      Dequeue front stack.
  end while loop
```

HINTS