

# INTRO TO PA02

## RULE OF THREE

## RECURSION

## GDB

---

Problem Solving with Computers-II

# C++

```
#include <iostream>
using namespace std;

int main(){
    cout<<"Hola Facebook!";
    return 0;
}
```

GitHub



# Announcements

- PA01 due tomorrow (1/29)- you may submit until this date for a 5% deduction.
- Lab02 due Thursday (1/31)
- Midterm next week (Monday)(02/04) - All topics covered so far.
- PA02: checkpoint due next week (02/06), final deadline (02/15)

How did PA01 go?

A. Done, it was okay

B. Done but I found it difficult

C. Not done, I need more guidance

D. So hard, I gave up!

# Review PA02: Card matching game involving linked lists

Expected files: Makefile, main.cpp, cards.cpp, cards.h, testcards.cpp

Correct output after running `make && ./game alice_cards.txt bob_cards.txt`:

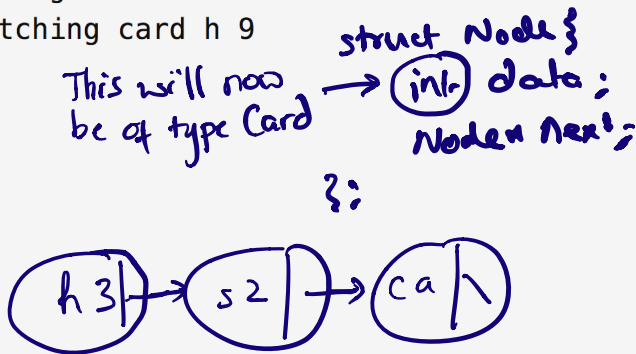
```
Alice picked matching card c 3
Bob picked matching card s a
Alice picked matching card h 9
```

Alice's cards:

```
h 3
s 2
c a
```

Bob's cards:

```
c 2
s a
d j
```



Contents of `alice_cards.txt`:

```
h 3
s 2
c a
c 3
h 9
s a
```

*Alice's hand*

Contents of `bob_cards.txt`:

```
c 2
s a
d j
h 9
c 3
```

*Bob's hand*

Note: 0=10, a=ace, k=king, q=queen, j=jack

# Review PA02: Checkpoint: Design and test!

Expected files: Makefile, main.cpp, cards.cpp, cards.h, gameplan.cpp, testcards.cpp

Correct output after running `make && ./game alice_cards.txt bob_cards.txt`:

Alice's cards:

```
h 3
s 2
c a
c 3
h 9
s a
```

Bob's cards:

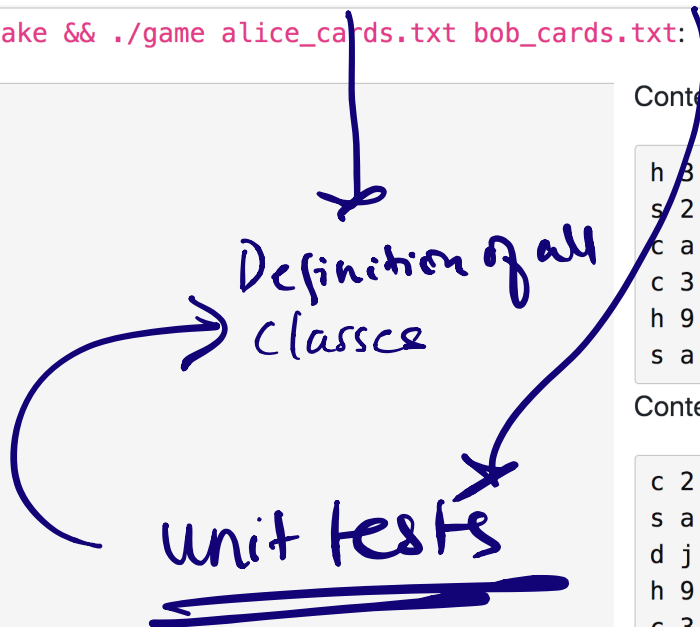
```
c 2
s a
d j
h 9
c 3
```

Contents of `alice_cards.txt`:

```
h 3
s 2
c a
c 3
h 9
s a
```

Contents of `bob_cards.txt`:

```
c 2
s a
d j
h 9
c 3
```



testcards.cpp

Unit test all your class methods.

```
void test_Player () {
```

```
    test_Player.insert();
```

```
    test_Player.search();
```

```
}
```

# RULE OF THREE

If a class defines one (or more) of the following it should probably explicitly define all three:

1. Destructor
2. Copy constructor
3. Copy assignment

The questions we ask are:

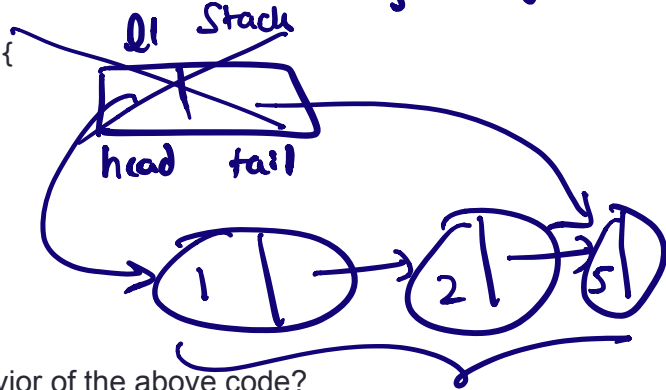
1. What is the behavior of these defaults (taking linked lists as our running example)?
2. Is the default behavior the outcome we desire ?
3. If not, how should we overload these operators?

# Behavior of default

Assume that your implementation of LinkedList uses the default destructor, copy constructor, copy assignment

Using default

```
void test_defaults(){  
    LinkedList l1;  
    l1.append(1);  
    l1.append(2);  
    l1.append(5);  
    l1.print();  
}
```



What is the expected behavior of the above code?

- A. Compiler error
- B. Memory leak
- C. Code is correct, output: 1 2 5
- D. None of the above



# Behavior of default copy constructor

Assume that your implementation of LinkedList uses the overloaded destructor,  
default: copy constructor, copy assignment

l1 : 1 -> 2 -> 5 -> null

```
void test_default_copy_constructor(LinkedList& l1){
```

```
    // Use the copy constructor to create a
```

```
    // copy of l1
```

```
    LinkedList l2(l1);
```

```
    or  
    LinkedList l2 = l1;
```

```
    or  
    LinkedList* l2 = new
```

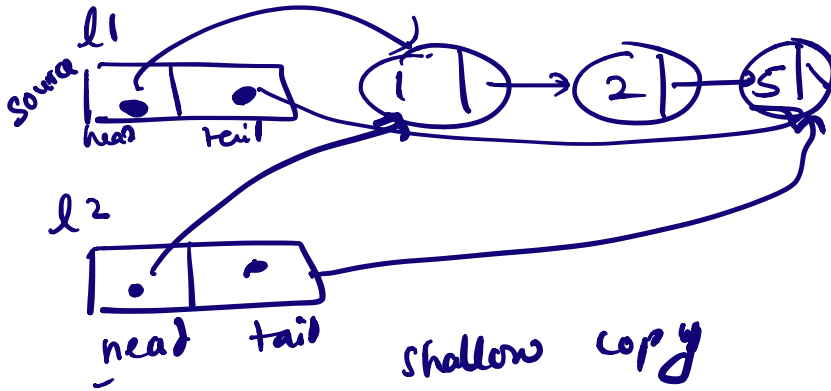
```
    LinkedList(l1);
```

```
}
```

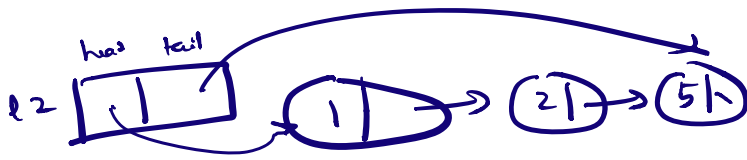
\* What is the default behavior?

\* Is the default behavior the outcome we desire ?

\* How do we change it?



Default copy constructor will do this



Over loaded copy constructor should create copies of all the nodes  
 Use the append functions of linked list but remember to set head and tail to null

# Behavior of default copy assignment

Assume that your implementation of LinkedList uses the overloaded destructor, copy constructor, default copy assignment

l1 : 1 -> 2 -> 5 -> null

```
void test_default_1(LinkedList& l1){
```

```
    LinkedList l2;
```

```
    l2 = l1;
```

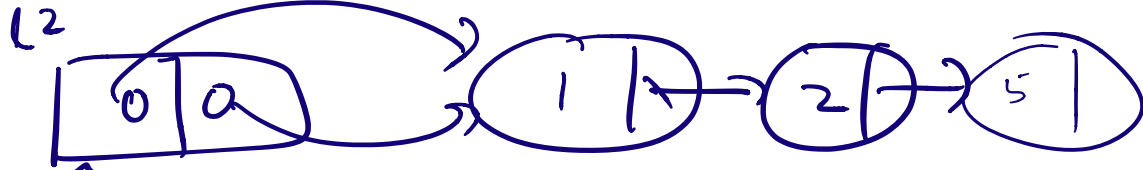
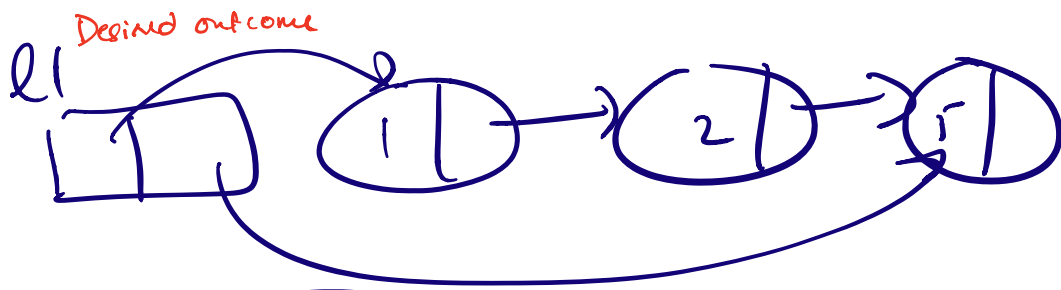
```
}
```

*l2.operator=(l1);*

\* What is the default behavior?

*l1's member variables are copied over to l2*

*shallow copy*



this → linkedlist l2;  
 l2.append(100)

$l2 = l1$  ; Note: l2 may have existing nodes that need to be deleted before l1's nodes are appended to l2

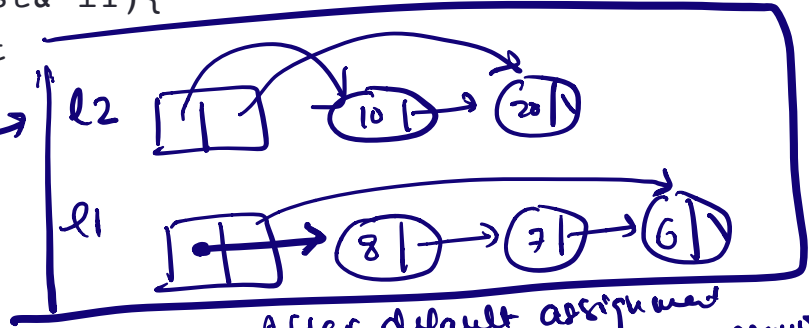
# Behavior of default copy assignment

Assume that your implementation of LinkedList uses the overloaded destructor, default: copy constructor, copy assignment

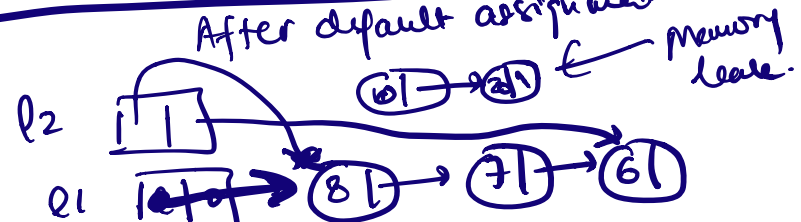
l1 : 1 -> 2 -> 5 -> null

```
void test_default_2(LinkedList& l1){  
    // Use the copy assignment  
    LinkedList l2;  
    l2.append(10);  
    l2.append(20);  
    l2 = l1;  
}
```

Before default assignment



After default assignment



\* What is the default behavior?

# Behavior of default copy assignment

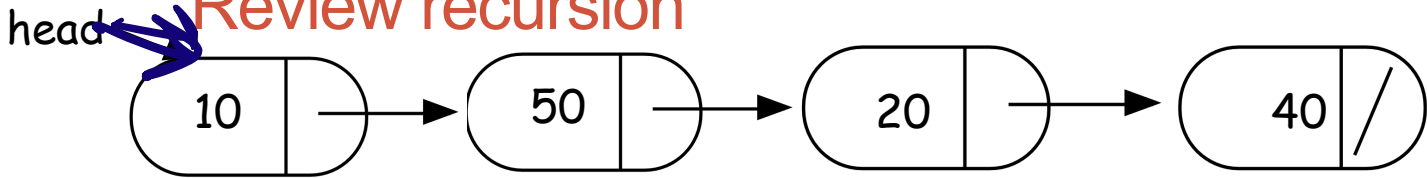
Assume that your implementation of LinkedList uses the overloaded destructor, copy constructor, default copy assignment

l1 : 1 -> 2 -> 5 -> null

```
void test_default_assignment(LinkedList& l1){  
    // Use the copy assignment  
    LinkedList l2;  
    l2.append(10);  
    l2.append(20);  
    l2 = l1;  
    l1 = l1;  
}
```

\* What is the default behavior?

## Review recursion



```
int IntList::search(int value){
```

```
    //Search for a value in a linked list
```

```
    //using recursion
```

```
}
```

# Helper functions

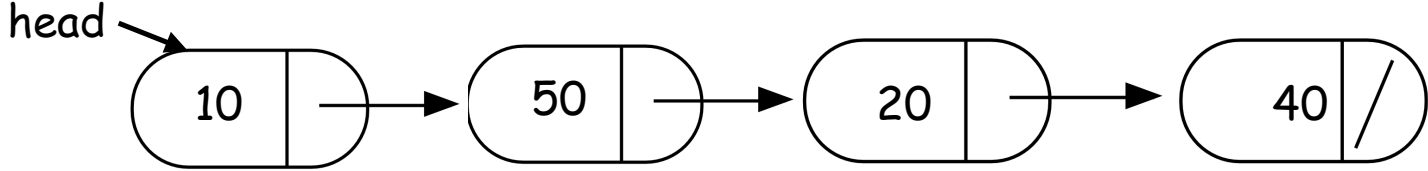
- Sometimes your functions takes an input that is not easy to recurse on
- In that case define a new function with appropriate parameters: This is your helper function
- Call the helper function to perform the recursion
- Usually the helper function is private

For example

```
bool IntList::search(int value){  
  
    return searchHelper(head, value);  
    //helper function that performs the recursion.  
  
}
```



# Review recursion



```
int IntList::searchHelper(Node* h, int value) {
```

```
    if(!h ) return false;
    if ( h->value == value)
        return true;
```

```
    searchHelper(h->next, value);
}
```

*Missing return*

What is the output of  
`cout<<list.searchHelper(head, 50);`

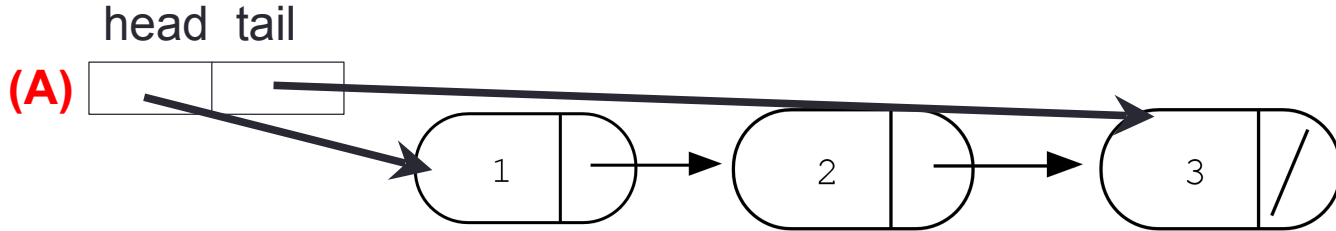
- A. Segmentation fault
  - B. Program runs forever
  - C. Prints true or 1 to screen
  - D. Prints nothing to screen
  - E. None of the above
- undefined behavior must probably a null value is printed*

# Concept Question

```
LinkedList::~~LinkedList(){  
    delete head;  
}
```

```
class Node {  
    public:  
        int info;  
        Node *next;  
};
```

Which of the following objects are deleted when the destructor of Linked-list is called?



**(B): only the first node**

**(C): A and B**

**(D): All the nodes of the linked list**

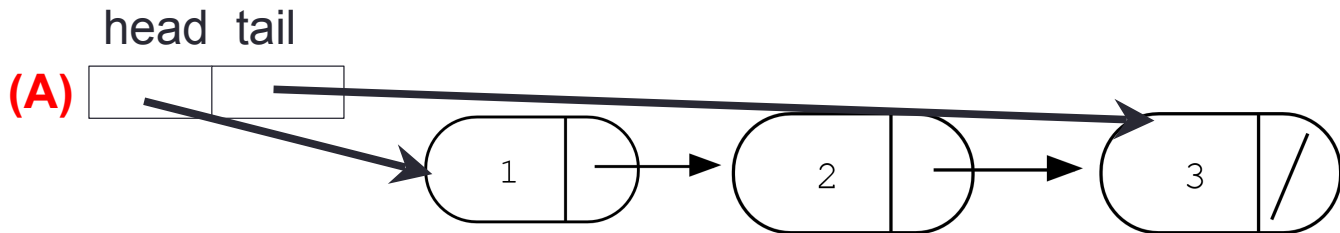
**(E): A and D**

# Concept question

```
LinkedList::~~LinkedList(){  
    delete head;  
}
```

```
Node::~~Node(){  
    delete next;  
}
```

Which of the following objects are deleted when the destructor of Linked-list is called?



**(B): All the nodes in the linked-list**

**(C): A and B**

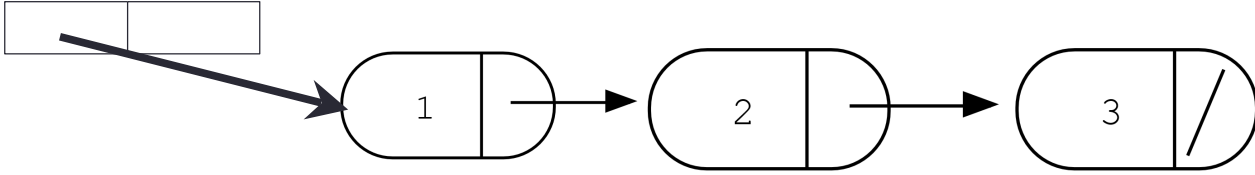
**(D): Program crashes with a segmentation fault**

**(E): None of the above**

```
LinkedList::~~LinkedList(){
    delete head;
}
```

```
Node::~~Node(){
    delete next;
}
```

head tail



# GDB: GNU Debugger

- To use gdb, compile with the -g flag
  - Setting breakpoints (b)
  - Running programs that take arguments within gdb (r arguments)
  - Continue execution until breakpoint is reached (c)
  - Stepping into functions with step (s)
  - Stepping over functions with next (n)
  - Re-running a program (r)
  - Examining local variables (info locals)
  - Printing the value of variables with print (p)
  - Quitting gdb (q)
  - Debugging segfaults with backtrace (bt)
- \* Refer to the gdb cheat sheet: <http://darkdust.net/files/GDB%20Cheat%20Sheet.pdf>

# Next time

- Complexity and running time analysis