1) Identify whether each of the following is either
    a. (a) memory leak
    b. (b) dangling pointer
    c. (c) unresolved pointer
    d. (d) something else

1a)

```c
int main(int argc, const char * argv[]) {
    int *ptr = (int *)malloc(sizeof(int) * 10);

    *ptr = 100;
    *(ptr + 9) = 10;

    return 0;
}
```

(a) Memory leak – malloc allocation was never freed.

1b)

```c
int main(int argc, const char * argv[]) {
    int *ptr;

    int x = 10;

    int y = 20;

    int sum = x + y;

    *ptr = 2140;

    return 0;
}
```

(c) Unresolved pointer – ptr is never given a value.

1c)

```c
int main(int argc, const char * argv[]) {
    char *ptr = (char *)malloc(sizeof(char));
    char *par = ptr;

    free(ptr);

    *ptr = 'f';

    ptr = (char *)malloc(sizeof(char));

    free(par);

    return 0;
}
```

(b) We have a dangling pointer, (d) double free, and (a) memory leak. Line 12 will set par to ptr (so they point to the same character). Then, we free ptr and attempt to set it to some character 'f' (Error a). Then, we set ptr to another malloc allocation (no problem). Then, we call free on par which has the original value of ptr, that pointer

was already released (Error d – double free). We never free the second call to malloc so there is also a memory leak (Error a). However, in a rare scenario, it is possible that we only have a dangling pointer. This occurs when ptr was given the same address on line 18 as it was given on 11. But, the heap is shared (typically) and therefore almost never going to give you the same address back to back.

2) What is wrong with this code snippet?

```
9
10  struct Node {
11      int data;
12      struct Node *next;
13  };
14
15  void addNode(struct Node **head, int data) {
16      struct Node *newNode = (struct Node *)malloc(sizeof(struct Node));
17      newNode->data = data;
18      newNode->next = *head;
19      *head = newNode;
20  }
21
22  void traverseList(struct Node *head) {
23      struct Node *current = head;
24      while (current != NULL) {
25          printf("%d\n", current->data);
26          current = current->next;
27      }
28  }
29
30  void inOrderTraversal(struct Node *head) {
31      struct Node *current = head;
32      while (current != NULL) {
33          printf("%d\n", current->data);
34          current->next = NULL;
35          current = current->next;
36      }
37  }
38
39  int main(int argc, const char * argv[]) {
40      struct Node *head = NULL;
41      addNode(&head, 1);
42      addNode(&head, 2);
43      addNode(&head, 3);
44
45      inOrderTraversal(head);
46
47      return 0;
48  }
```

Error on line 34. When we have inOrderTraversal, we expect that it is printed or saved to a file. **As such, we should not set next to NULL!** We never expect a traversal function to modify internals. It is called traversal when we go over the list to insert something, but the function name will be indicative of such behavior. This is another reason to use proper naming conventions.

3) What operation do we use to add to a stack?
push – this will add contents to some stack.

4) How do we access the 10th element in a 50-element stack?
Directly – impossible, we can only touch the top element
Indirectly – we pop the first 9 elements, then read the 10th (or pop, depending on what we need)

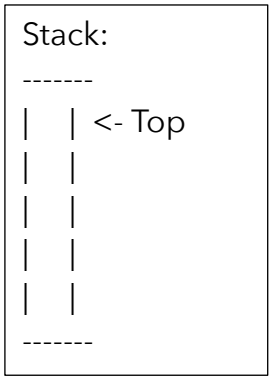
5) Which of the following allows us to copy a list of nodes?

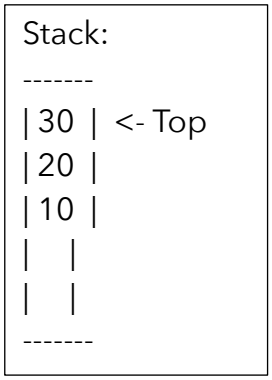
```
15  void copyList(Node ** oldL, Node ** newL) {
16
17  }
18
19  void copyList(Node * oldL, Node * newL) {
20
21  }
22
23  void copyList(Node *& oldL, Node *& newL) {
24
25  }
```

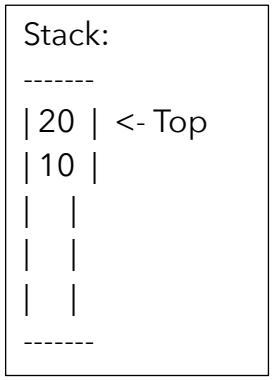
In C, only the 15th line. The 19th will allow us to traverse oldL and newL, but not modify either. Only in C++ is 23 valid. It is a reference to a pointer – if you don't know what that means today, you will know soon enough.

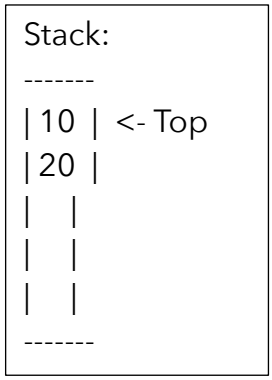
6) Are shallow or deep copies better?
It depends on what you need. There is no real good answer for all situations. Shallow copies allow for easy implementation and are quick. Deep copies require that you go through the entire structure and copy variable by variable. This means manually using malloc to create new objects, strcpy, etc.

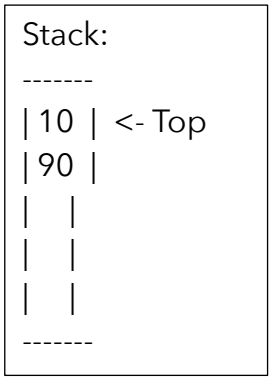
7) Below each stack, declare what operations were performed











Left: Create stack
Left Center: PUSH 10 PUSH 20 PUSH 30
Center: POP
Center Right: POP POP PUSH 20 PUSH 10
Right: POP POP PUSH 90 PUSH 10
You can also use another stack to go from one to another (where new data is not introduced).

8) [Easy] There is nothing wrong with the semantics of the code (it is valid C++ code). Pretend that pop and push are implemented correctly. For simplicity, we use a counter. There is something wrong with the code because it does not return true for a valid set of code. A valid code is considered on with a balanced amount of open (and closing) in addition to a balanced count of { and }. *You may need to add, remove, or modify functions to complete this.*

```
12  static int count = 0;
13
14  void pop() {
15      --count; // Assume we pop here
16  }
17
18  void push() {
19      ++count; // Assume we push here
20  }
21
22  bool isValidCode(string code) {
23      for (auto &c : code) {
24          if (c == '(' || c == '{') {
25              push();
26          } else {
27              pop();
28          }
29      }
30
31      return count == 0;
32  }
```

You need to explicitly check for a closing ')' or '}'. Without checking, you will pop when you should do nothing. You also need to use a stack for () and a stack for {}. Summary: add a stack, check explicitly for the (or { and use separate if statements.

9) Write push, pop, and peek function for some stack

Funcs written by GPT-4o mini

```
void push(Stack* stack, int item) {
    if (isFull(stack)) {
        printf("Stack overflow! Cannot push %d\n", item);
        return;
    }
    stack->items[++stack->top] = item;
    printf("%d pushed to stack\n", item);
}

int pop(Stack* stack) {
    if (isEmpty(stack)) {
        printf("Stack underflow! Cannot pop from empty stack\n");
        return -1; // Return -1 to indicate stack is empty
    }
    return stack->items[stack->top--];
}

int peek(Stack* stack) {
    if (isEmpty(stack)) {
        printf("Stack is empty! Cannot peek\n");
        return -1; // Return -1 to indicate stack is empty
    }
    return stack->items[stack->top];
}
```

10) What is an ADT?

Abstract Data Type – examples are lists, queues, stacks. These are simply abstract concepts which can be made for any type (int, char, etc.).

11)Why do we have a call stack? ~~Is it possible to have custom unwinding?~~

Because it allows us to efficiently track calls using a stack since we work with the top element. Working with other types are more resource heavy and does not make

sense. If we call f(), but then start executing c() because c() is at the bottom of the stack, that makes no sense, therefore, we only need to know of f(). Sorry, the unwinding question was supposed to be removed. But, yes, it is possible to have custom winding meaning that we do not necessarily jump back to the last call frame, but jump to the frame 3 calls ago.