CSE 333 Section 7

Memory Diagram Review &
Smart Pointers

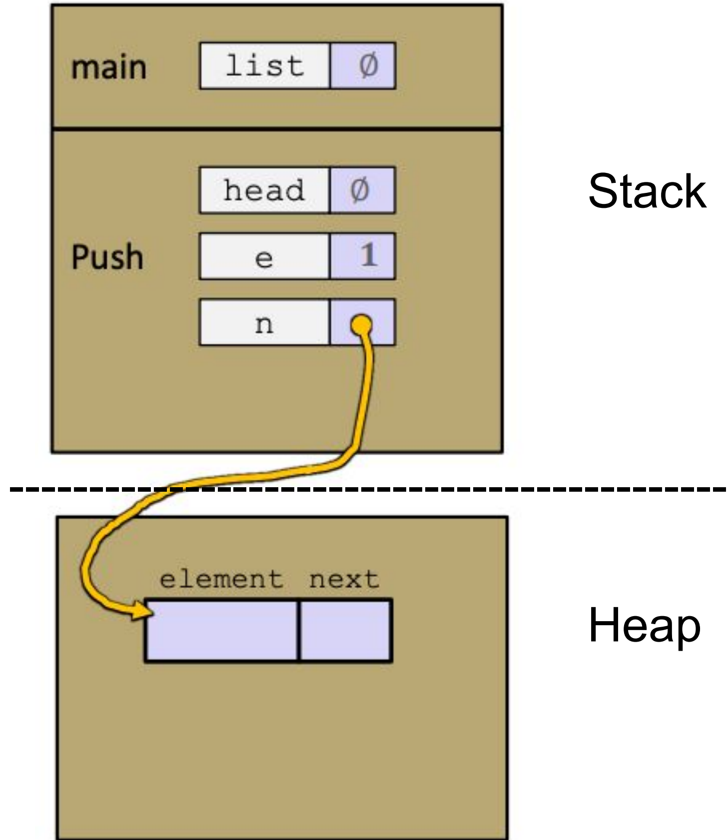
Logistics

- HW3:
    - Due **11/19 (Tuesday) @ 10 PM**
- Exercise 13:
    - Due **11/08 (Tomorrow!) @ 10 AM**

Memory Diagram Review

Memory Diagrams / Box-and-Arrow Diagrams

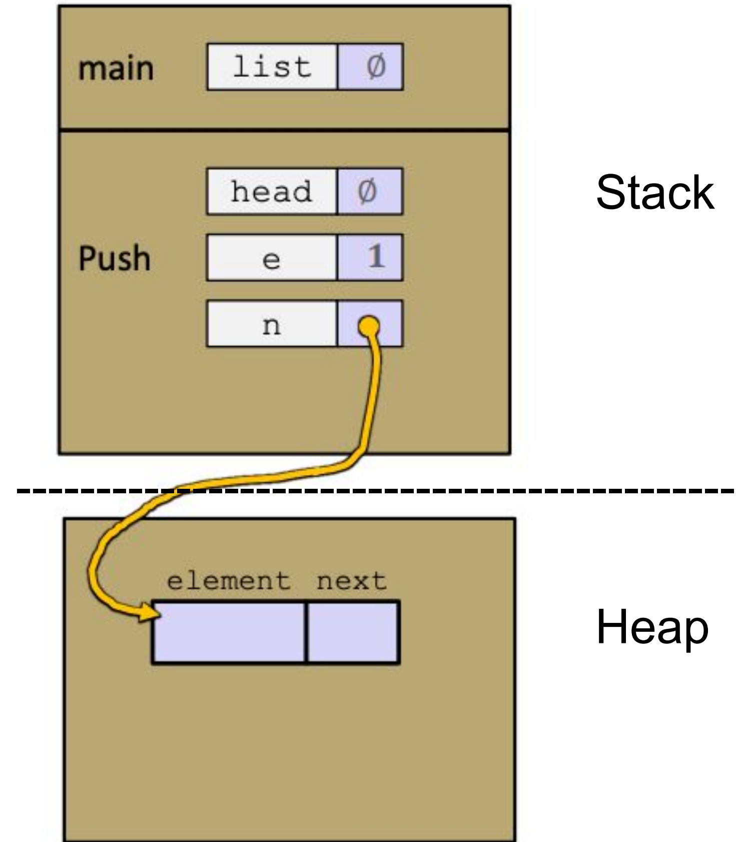
Memory diagrams, sometimes called box-and-arrow diagrams, visually describe the state of the program



Memory Diagrams / Box-and-Arrow Diagrams

A good memory diagram will have:

- An area for the stack and (if dynamic memory) heap, ideally both labeled
  - Be sure to label stackframes!
- Variables will be located in their stack frames or the heap. Every variable will have its own box, labeled with its name. Its value is drawn within the box.
- If a struct/class's fields are clear from context, there's no need to label them; if it's unclear, then label

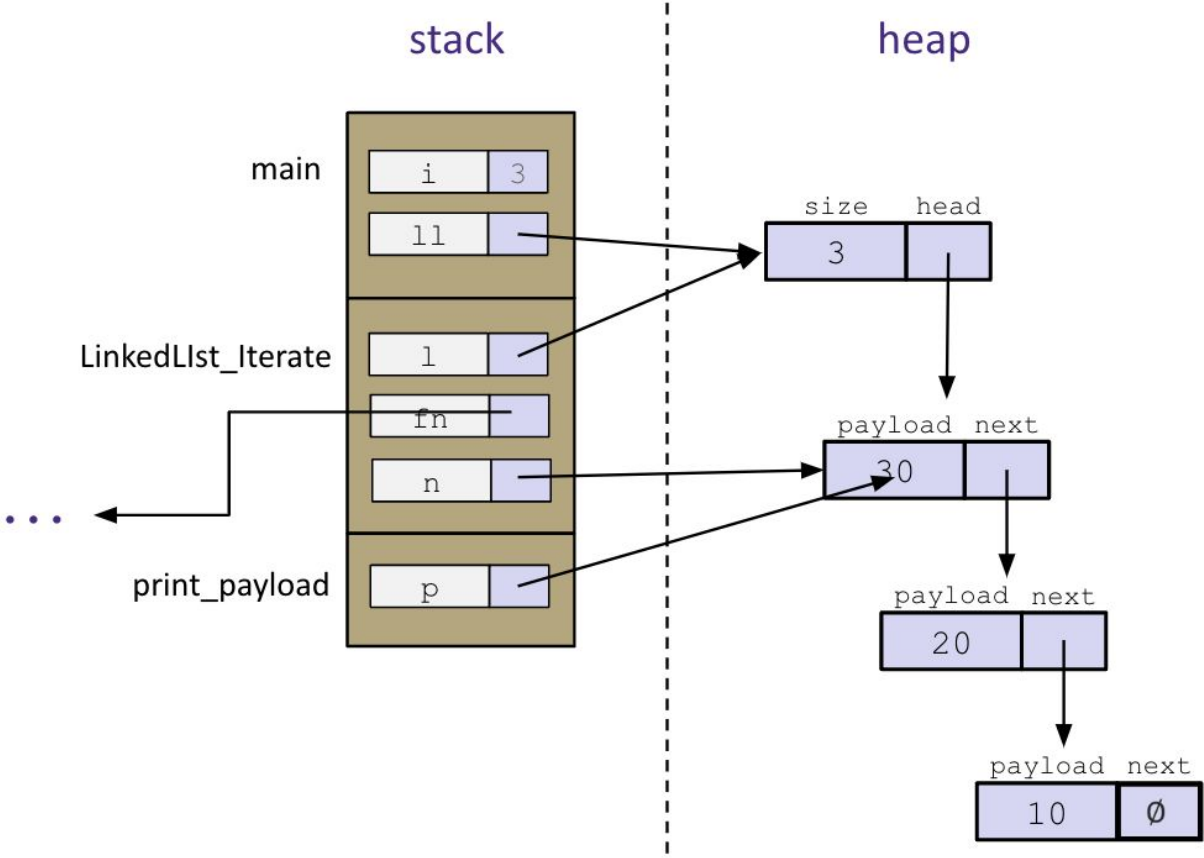


Exercise 1

```c
int main(void) {
 int i;
 LinkedList *ll = LinkedList_Allocate();
 printf("Please enter a list of integers you'd like reversed: ");
 while (1) {
   if (scanf("%d", &i) != 1) break;
   LinkedList_Push(ll, i);
 }
 printf("\nYour reversed numbers are:\n");
 LinkedList_Iterate(ll, &print_payload);
 printf("\n");
 LinkedList_Deallocate(ll);
 return 0;
}
```

Assume that you've set a breakpoint at the **first** call to `print_payload()`. Draw the memory diagram at that breakpoint, assuming that the program's input was **"10 20 30"**.

Exercise 1



Exercise 1

Things to note about this diagram:

- The stack is currently 3 frames deep, so we've drawn all 3 and labeled each one
- The linked list was dynamically allocated, so it's drawn in a labeled area for the heap
- It's clear from context that the `LinkedListNode` struct consists of an integer and a pointer, so we don't have to label each field (but it's also ok to do so)!
- Each variable, even if it's a copy, has a box containing its value and a label with its name.
- We haven't discussed how to draw function pointers, so it's ok to have different-looking answers. The important idea is that a function pointer is an address, and therefore it requires a variable to store that value.

Smart Pointers!

C++

raw pointers

trying to prevent people from using raw pointers

Review: Smart Pointers

- **`std::unique_ptr`** (Documentation) – Uniquely manages a raw pointer
  - Used when you want to declare unique ownership of a pointer
  - Disabled `cctor` and `op=`
- **`std::shared_ptr`** (Documentation) – Uses reference counting to determine when to delete a managed raw pointer
  - **`std::weak_ptr`** (Documentation) – Used in conjunction with `shared_ptr` but does **not** contribute to reference count

Using Smart Pointers

- Treat a smart pointer like a **normal (raw) pointer**, except now you **won't** have to use **`delete`** to deallocate memory!
  - You can use `*`, `->`, `[]` as you would with a raw pointer!

- **Initialize** a smart pointer by passing in a pointer to **heap memory**:

```
unique_ptr<int[]> u_ptr(new int[3]);
```

  - For **`shared_ptr`** and **`weak_ptr`**, you can use cctor and op= to get a copy

```
shared_ptr<int[]> s_ptr(another_shared_ptr);
```

Using Smart Pointers cont.

- Want to transfer ownership from one `unique_ptr` to another ?

```
unique_ptr<T> V = std::move(unique_ptr<T> U);
```

- Want to get the reference count of a `shared_ptr`?

```
int count = s.use_count();
```

- Want to convert your `weak_ptr` to a `shared_ptr`?

```
std::shared_ptr s = w.lock();
```

Exercise 2

Exercise 2

Change the following code to use smart pointers. Should each field be a unique, shared or weak pointer?

```
#include <memory>
using std::shared_ptr;
using std::unique_ptr;
using std::weak_ptr;

struct IntNode {
  IntNode(int* val, IntNode* node): value(val), next(node) {}

  ~IntNode() { delete value; }

  int* value;
  IntNode* next;
};
```

Exercise 2

```
#include <memory>
using std::shared_ptr;
using std::unique_ptr;
using std::weak_ptr;

struct IntNode {
  IntNode(int* val, IntNode* node) :
   value(unique_ptr<int>(val)), next(shared_ptr<IntNode>(node)) {}

  ~IntNode() { delete value; }

  unique_ptr<int> value;
  shared_ptr<IntNode> next;
};
```

Exercise 2

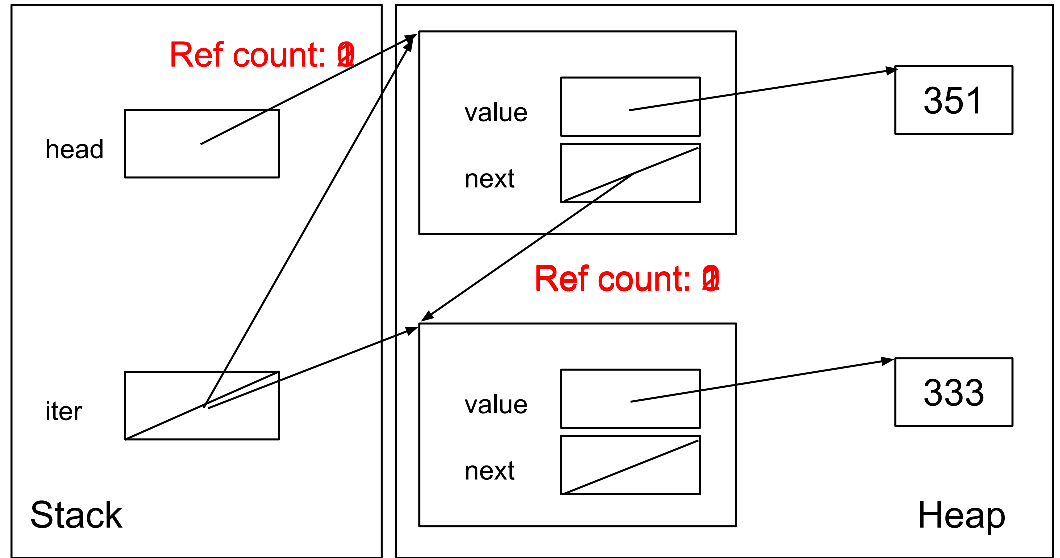
```
#include <memory>
using std::shared_ptr;
using std::unique_ptr;
using std::weak_ptr;

struct IntNode {
  IntNode(int* val, IntNode* node) :
   value(unique_ptr<int>(val)), next(shared_ptr<IntNode>(node)) {}

  ~IntNode() { delete value; }

  unique_ptr<int> value;
  shared_ptr<IntNode> next;
};
```

Example: Client Code




```cpp
#include <iostream>

using std::cout;
using std::endl;
using std::shared_ptr;

int main() {
  shared_ptr<IntNode> head(new IntNode(new int(351), nullptr));
  head->next = shared_ptr<IntNode>(new IntNode(new int(333), nullptr));
  shared_ptr<IntNode> iter = head;
  while (iter != nullptr) {
    cout << *(iter->value) << endl;
    iter = iter->next;
  }
}
```

Example: Client Code

Nothing left on the heap!

```
#include <iostream>

using std::cout;
using std::endl;
using std::shared_ptr;

int main() {
  shared_ptr<IntNode> head(new IntNode(new int(351), nullptr));
  head->next = shared_ptr<IntNode>(new IntNode(new int(333), nullptr));
  shared_ptr<IntNode> iter = head;
  while (iter != nullptr) {
    cout << *(iter->value) << endl;
    iter = iter->next;
  }
}
```