INTRO TO LINKED LISTS

LECTURE 04-2

JIM FIX, REED COLLEGE CS2-F20

SOLUTIONS TO LAB 04

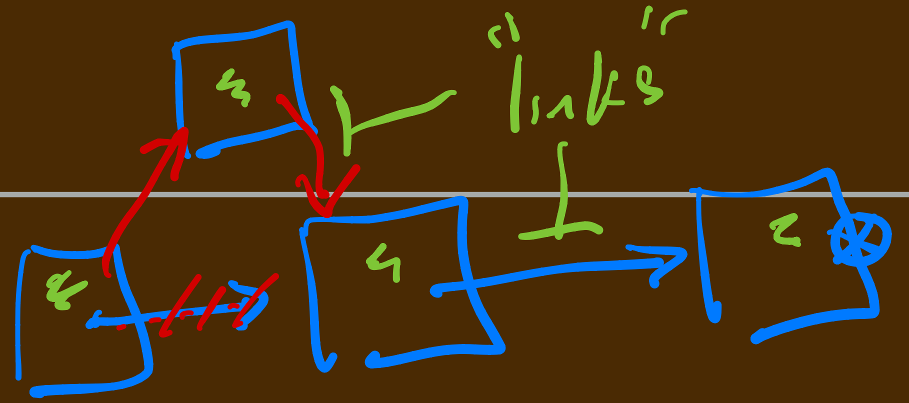
Two versions:

- One that uses `(*c).odometer` notation.
- One that uses `c->odometer` notation.

NEED FOR LINKED DATA STRUCTURES

C++ arrays can be used to hold collections of data items, but they are limited in their "direct" application:

- They cannot be resized; their length is set at allocation time.
- The valid data items held in an array are normally contiguously laid out.
  - To add items, we normally must shift items; much copying.
  - Removing normally also requires shifting items, or marking unused items.
    - Marking forces us to sift through the array, looking for valid items.
  - Resizing often requires a new allocation and a copying of the items.
- Looking for items might require "overlay" structures; clever organization.



LINKED DATA STRUCTURES

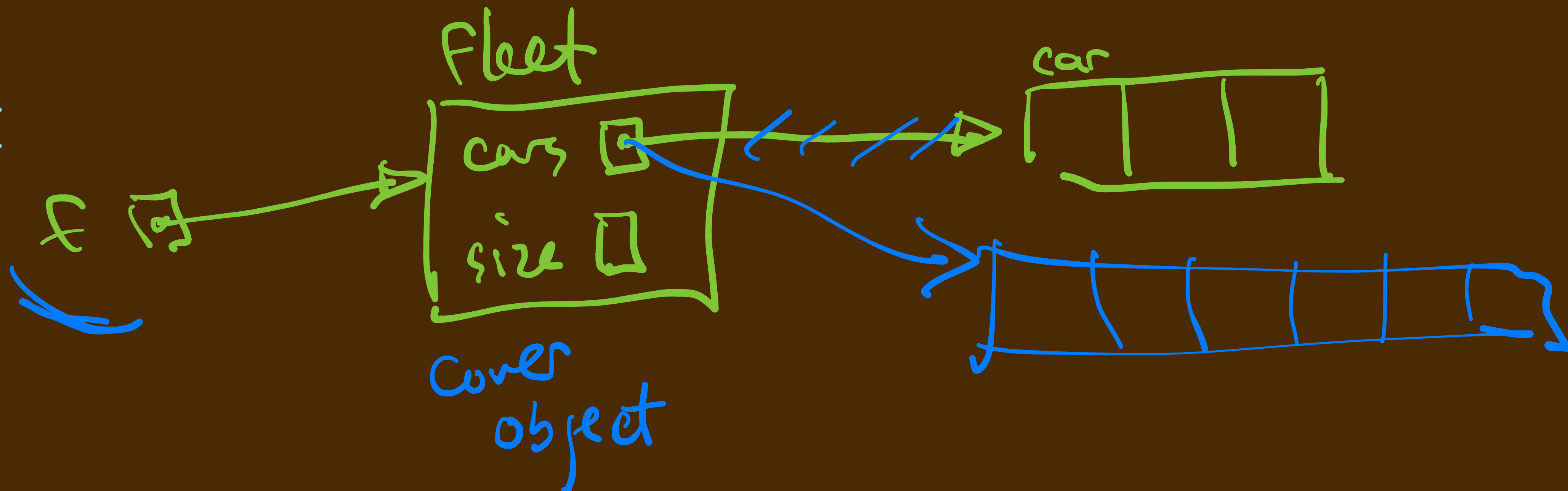
- Using pointers, we can organize a data structure as a collection of components and then "link" several components together.
  - A component containing one/several data items can point to other components containing related data.
- We can link an arbitrary number of these components to make a collection.
  - This makes it possible to **add** or **remove** items from the collection.
  - To **resize** a linked collection, we simply link in more components.
    - We just allocate more cnew omponents from the heap, any number

Today we study data structures called ***linked lists***. Gateway to trees, graphs, ...

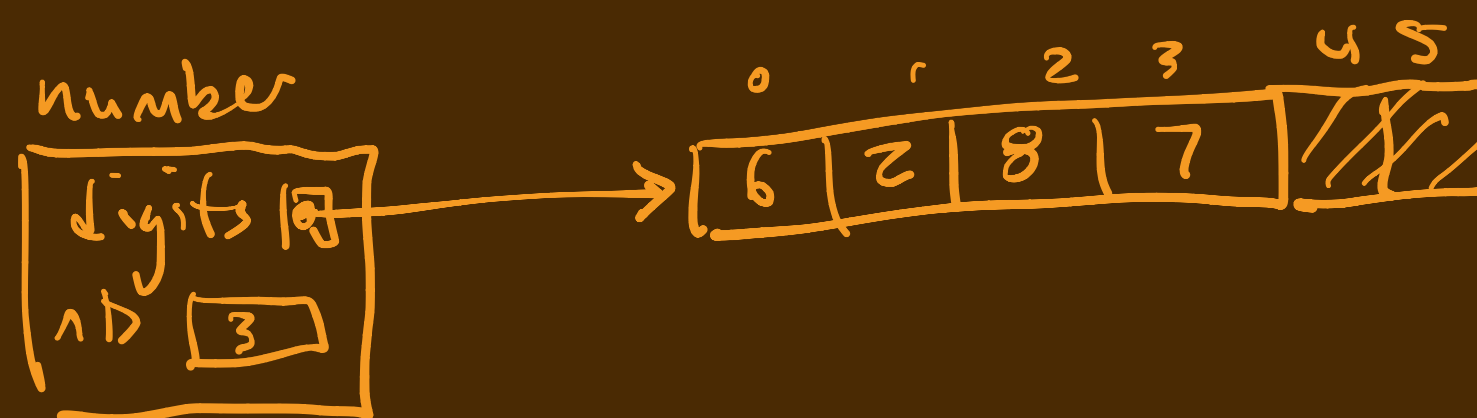
EXAMPLE STRUCTURES

Before looking at linked lists, consider some linked designs:

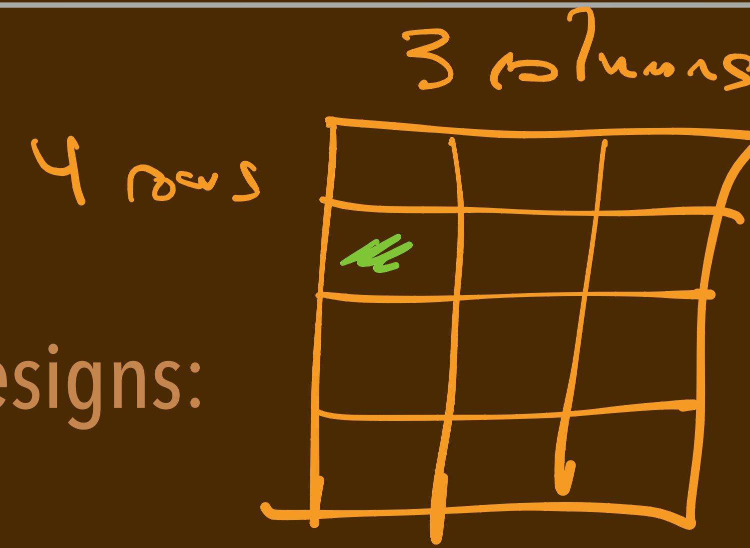
```
struct fleet {
   car* cars;
   int size;
};
```




```
struct number {
   int* digits;
   int numDigits;
   int capacity;
};
```

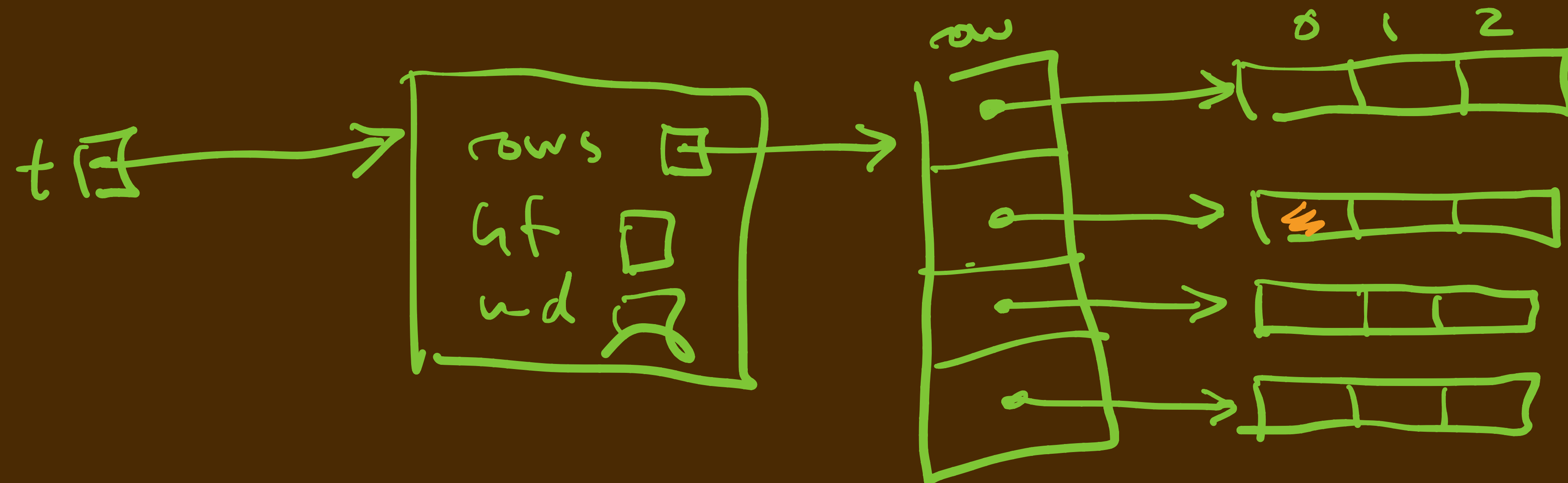


EXAMPLE STRUCTURES



Before looking at linked lists, consider some linked designs:

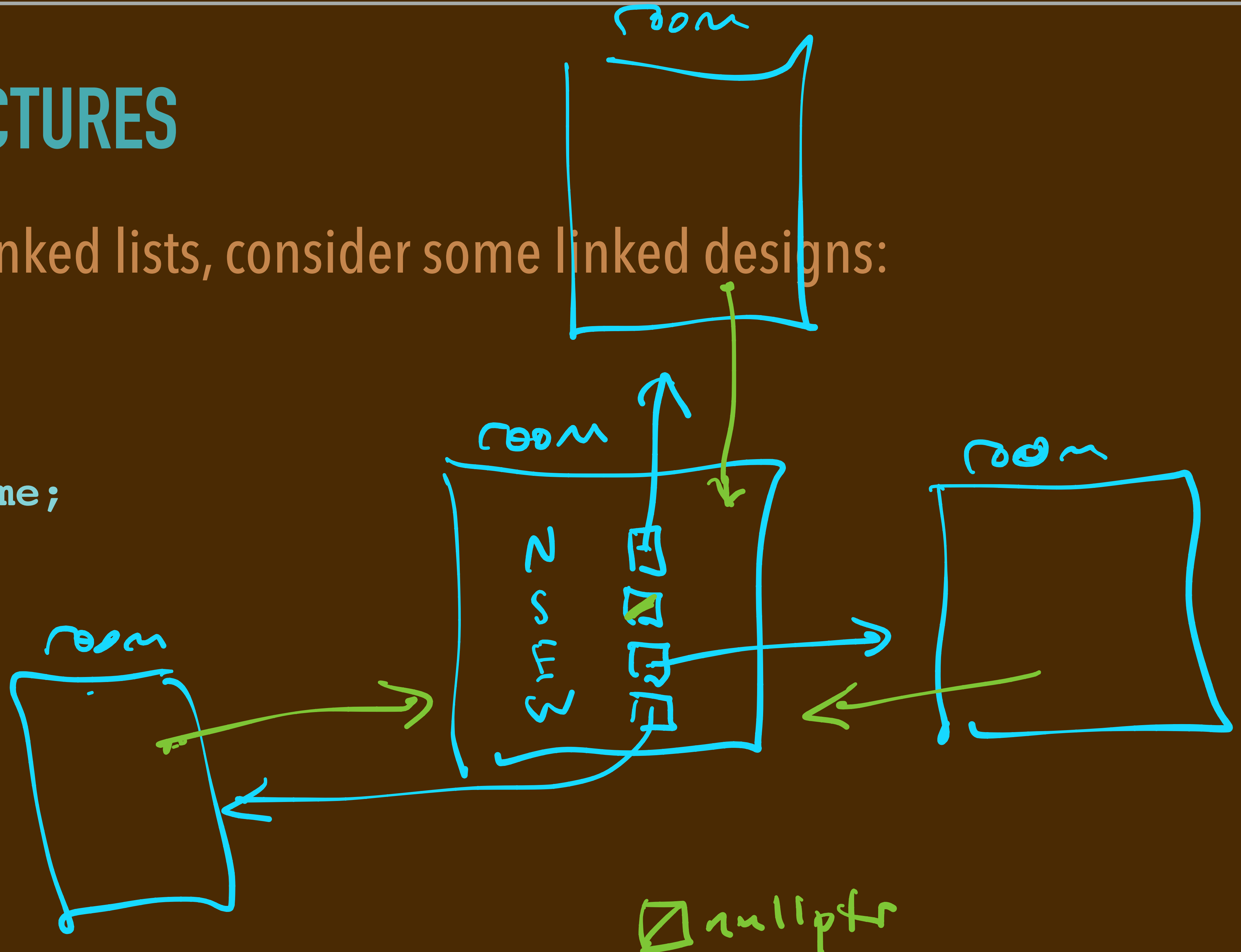
```
struct table {
  row* rows;
  int height;
  int width;
};

struct row {
  douuble* columns;
};
```



EXAMPLE STRUCTURES

Before looking at linked lists, consider some linked designs:

```
struct room {
   std::string name;
   room* north;
   room* south;
   room* east;
   room* west;
};
```



EXAMPLE STRUCTURES

Before looking at linked lists, consider some linked designs:

```
struct room {
  std::string name;
  struct room* north;
  struct room* south;
  struct room* east;
  struct room* west;
};
```

EXAMPLE STRUCTURES

Before looking at linked lists, consider some linked designs:

```
struct room {
  std::string name;
  struct room* north;
  struct room* south;
  struct room* east;
  struct room* west;
};
```

within room's definition!

room r;
room* p;

THROUGH SOME QUIRK OF C INHERITED BY C++, "STRUCT ROOM" IS DEFINED RIGHT AWAY BUT THE NEW TYPE "ROOM" IS DEFINED AFTER.

EXAMPLE STRUCTURES

Before looking at linked lists, consider some linked designs:

```
struct room {
  std::string name;
  struct room* north;
  struct room* south;
  struct room* east;
  struct room* west;
};
```

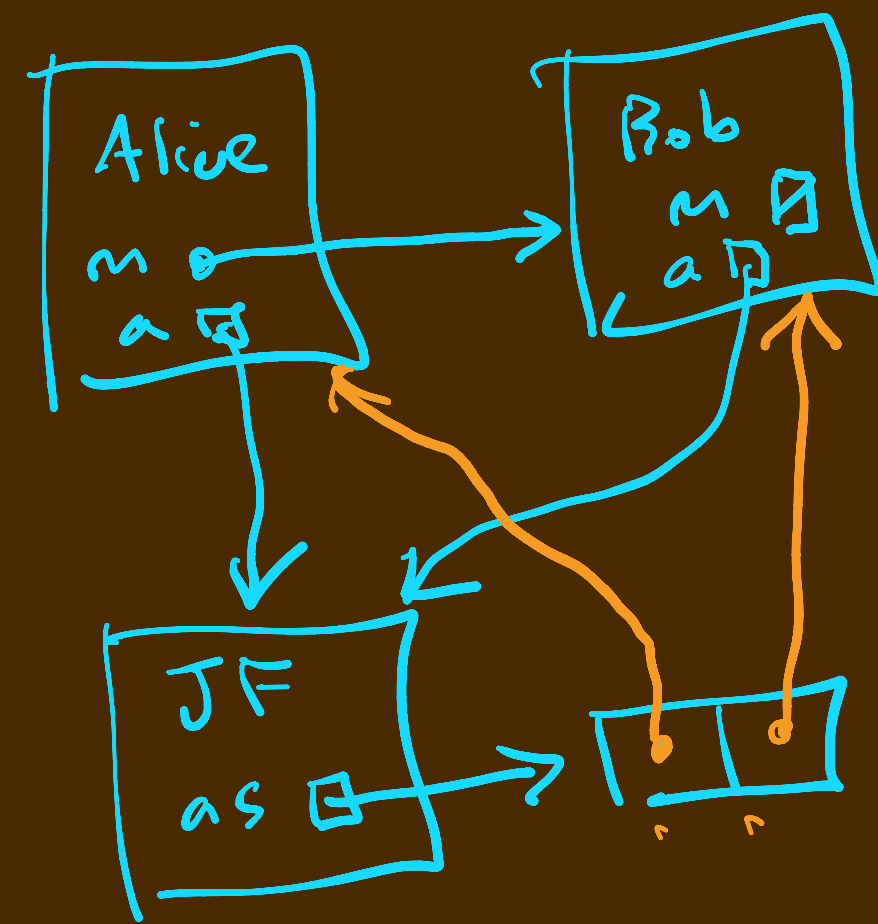
These can each be `nullptr` for walls in maze.

THROUGH SOME QUIRK OF C INHERITED BY C++, "STRUCT ROOM" IS DEFINED RIGHT AWAY BUT THE NEW TYPE "ROOM" IS DEFINED AFTER.

EXAMPLE STRUCTURES

Before looking at linked lists, consider some linked designs:

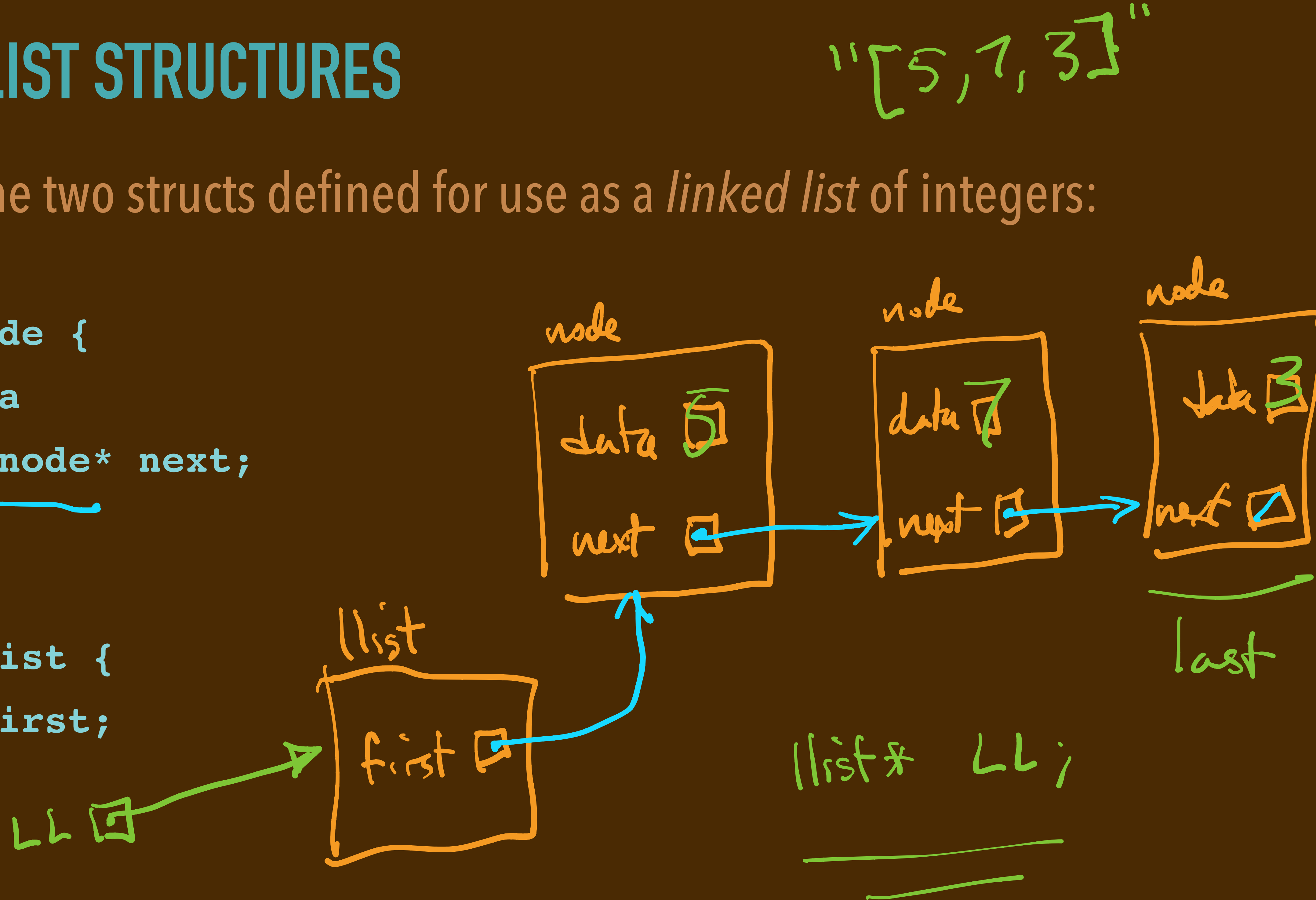
```
struct student {
  std::string name;
  std::string major;
  int year;
  struct student* mentor;
  struct prof* advisor;
};
struct prof {
  std::string name;
  std::string department;
  student* advisees;
};
```



LINKED LIST STRUCTURES

Here are the two structs defined for use as a *linked list* of integers:

```
struct node {
  int data
  struct node* next;
};

struct llist {
  node* first;
};
```



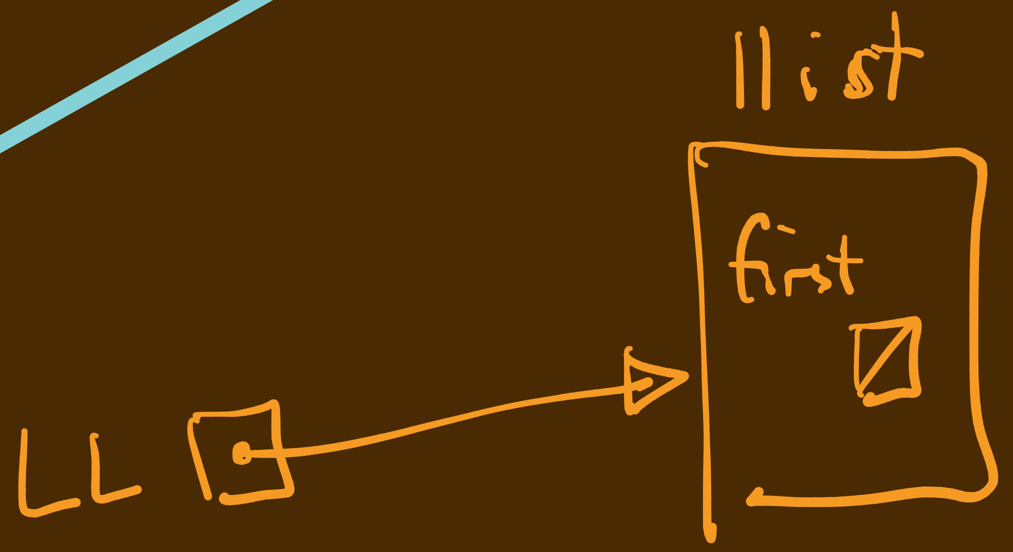
LINKED LIST STRUCTURES

"[ ]"

Here are the two structs defined for use as a *linked list* of integers:

```
struct node {
  int data
  struct node* next;
};

struct llist {
  node* first;
};
```

This is `nullptr` if the list is empty.



LINKED LIST STRUCTURES

Here are the two structs defined for use as a *linked list* of integers:

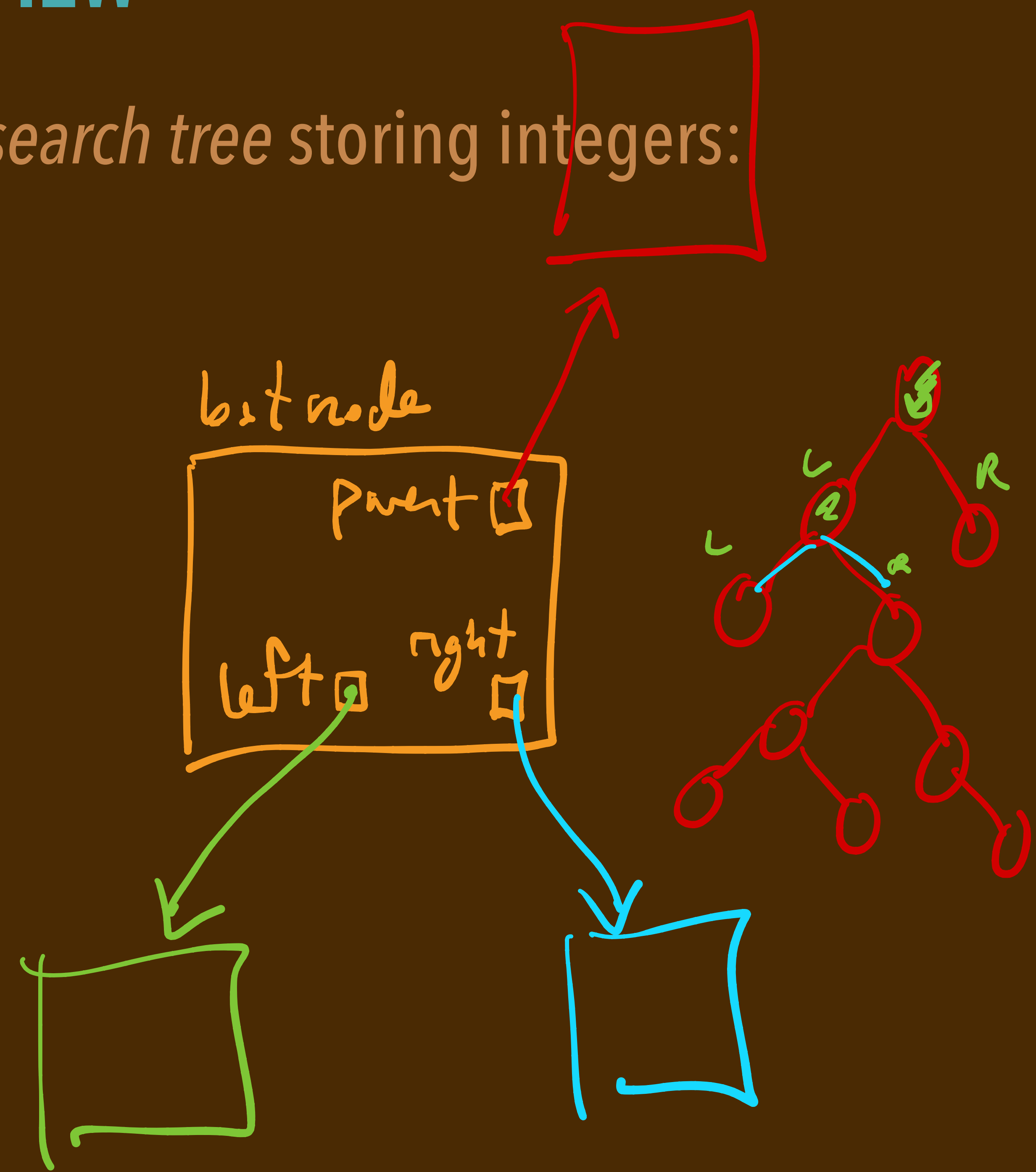
```
struct node {
   int data
   struct node* next;
};

struct llist {
   node* first;
};
```

This is `nullptr` if the holding the last item.

SEARCH TREE STRUCTURES, A PREVIEW

Here are the two structs used for a *binary search tree* storing integers:

```
struct bstnode {
  int key;
  struct bstnode* parent;
  struct bstnode* left;
  struct bstnode* right
};

struct bst {
  bstnode* root;
};
```



SEARCH TREE STRUCTURES, A PREVIEW

Here are the two structs used for a *binary search tree* storing integers:

```
struct bstnode {
  int key;
  struct bstnode* parent;
  struct bstnode* left;
  struct bstnode* right
};
```

This is `nullptr` for the tree's *root*.

```
struct bst {
  bstnode* root;
};
```

This is `nullptr` if the tree's collection is empty.

SEARCH TREE STRUCTURES, A PREVIEW

Here are the two structs used for a *binary search tree* storing integers:

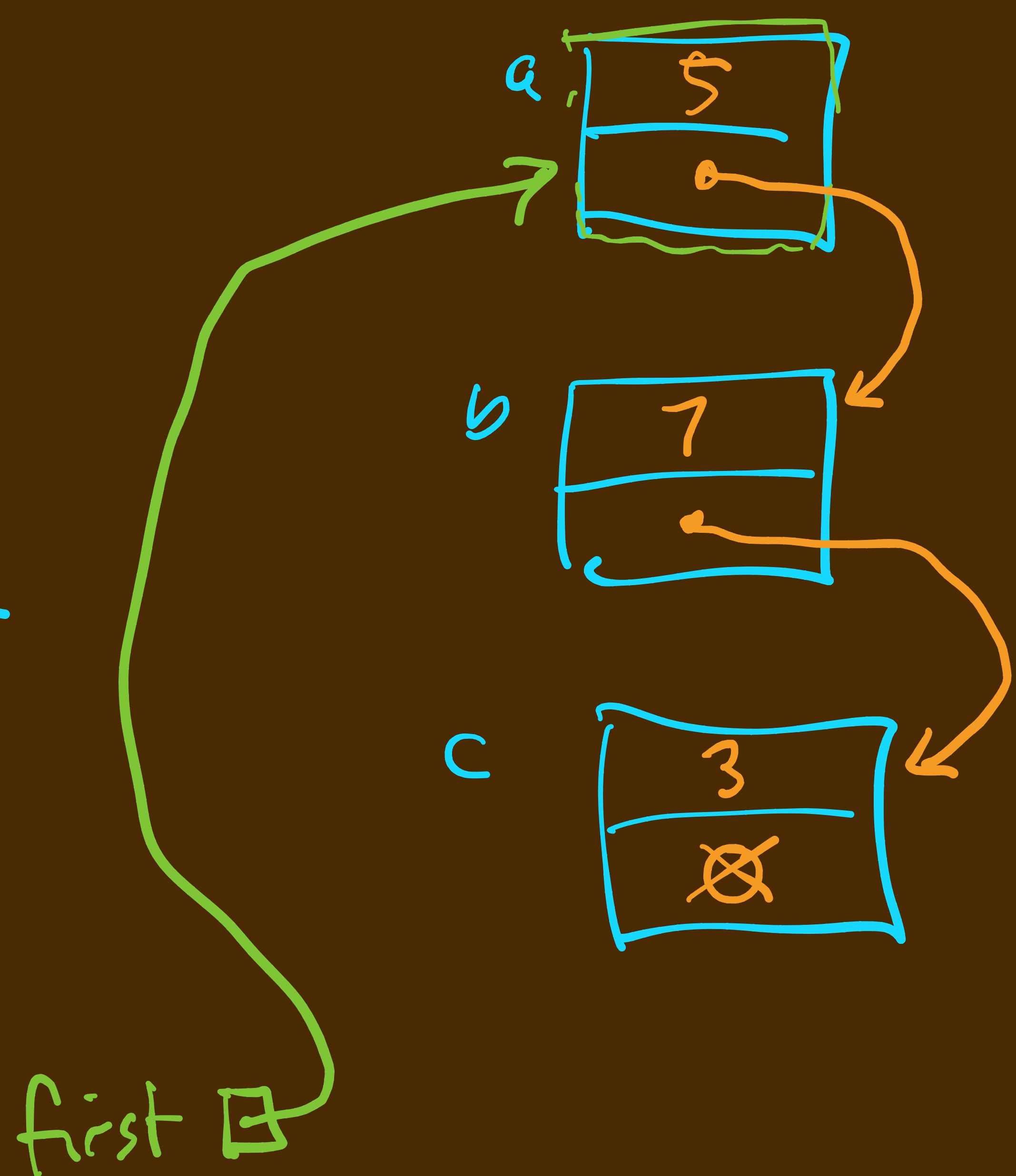
```
struct bstnode {
  int key;
  struct bstnode* parent;
  struct bstnode* left;
  struct bstnode* right
};

struct bst {
  bstnode* root;
};
```

These are `nullptr` at a *leaf node*.

SOME LINKED LIST CODE

Consider this code. What does it do?

```
struct node {
  int data;
  struct node* next;
};

int main(void) {
  node a;
  node b;
  node c;
  a.data = 5;
  b.data = 7;
  c.data = 3;
  node* first = &a;
  a.next = &b;
  b.next = &c;
  c.next = nullptr;
}
```



g++ --std=c++17 -o foo foo.cc

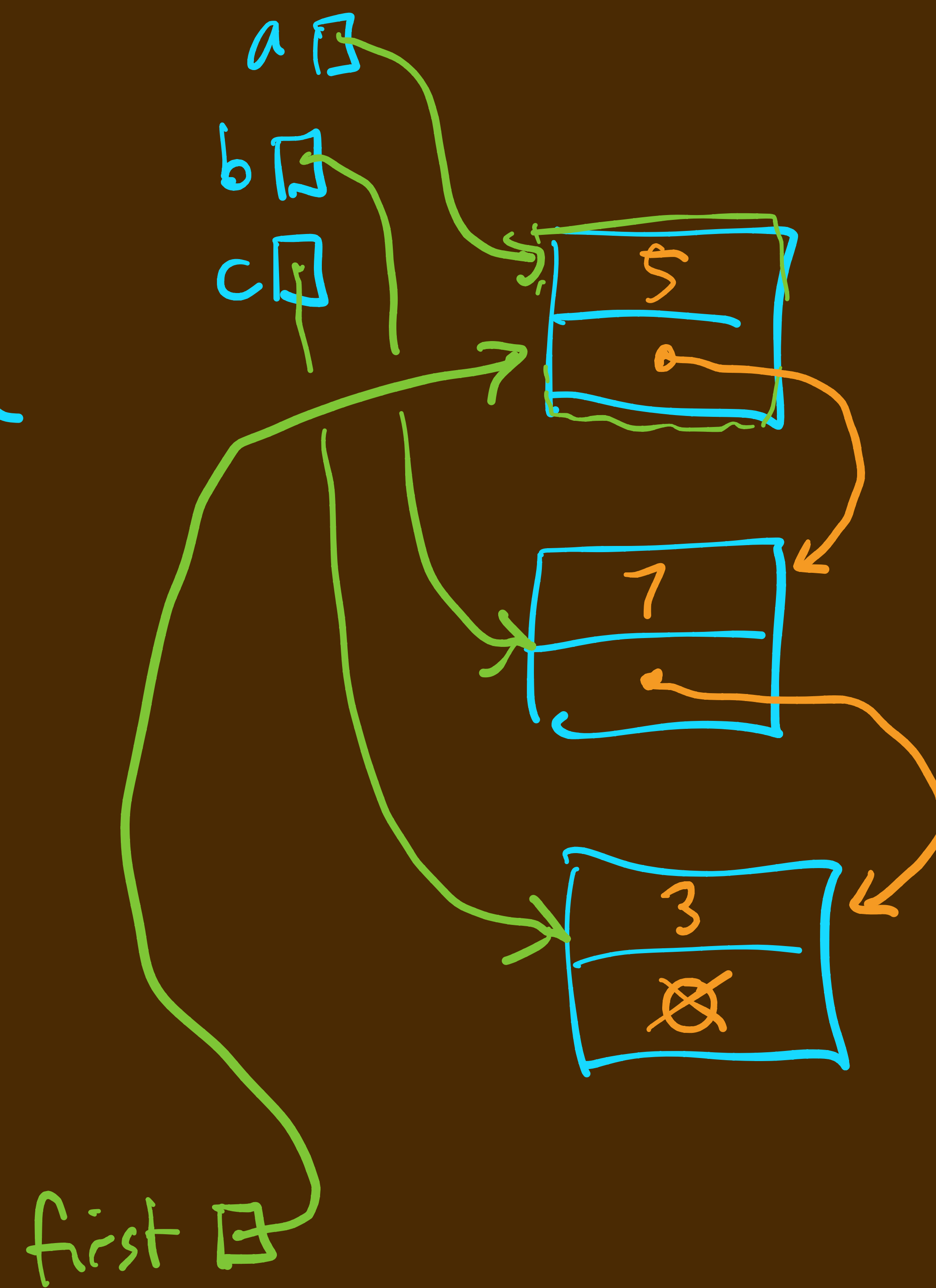
SOME LINKED LIST CODE

Consider this code. What does it do?

```
struct node {
  int data;
  struct node* next;
};

int main(void) {
  node* a = new node {5, nullptr};
  node* b = new node {7, nullptr};
  node* c = new node {3, nullptr};
  node* first = a;
  a.next = b;
  b.next = c;
}
```

heap allocated

C++11, C++17



SOME LINKED LIST CODE

Consider this code. What does it do?

```
struct node {
  int data;
  struct node* next;
};
struct llist {
  node* first;
};

int main(void) {
  node* a = new node {5, nullptr};
  node* b = new node {7, nullptr};
  node* c = new node {3, nullptr};
  llist* LL = new llist {a};
  node* first = a;
  a.next = b;
  b.next = c;
}
```

SOME LINKED LIST CODE

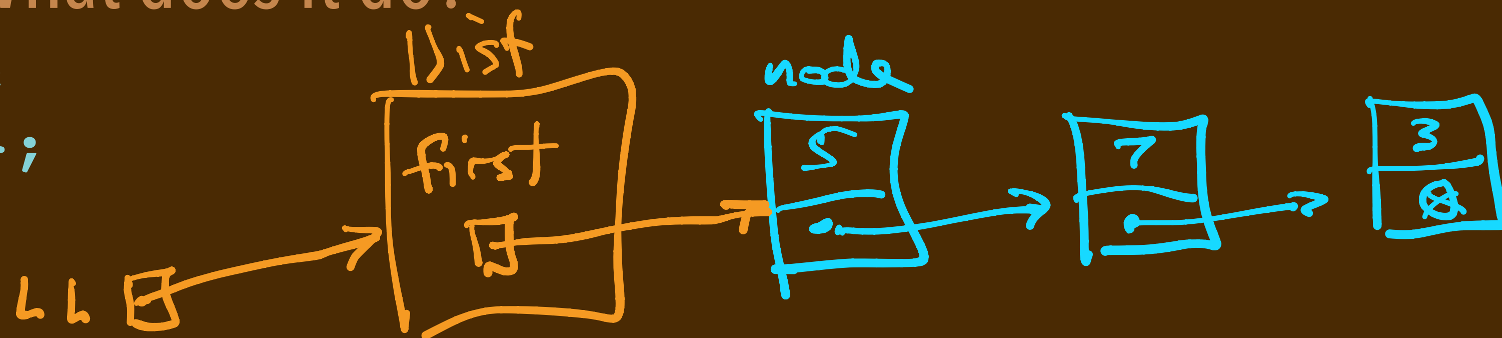
Consider this code. What does it do?

```
struct node {
  int data;
  struct node* next;
};

struct llist {
  node* first;
};


int main(void) {
  node* c = new node {3, nullptr};
  node* b = new node {7, c};
  node* a = new node {5, b};
  llist* LL = new llist {a};
}
```

SOME LINKED LIST CODE

Consider this code. What does it do?

```
struct node {...};
struct llist {...};
```




```
int main(void) {
  node* c = new node {3, nullptr};
  node* b = new node {7, c};
  node* a = new node {5, b};
  llist* LL = new llist {a};
  std::cout << LL->first->data << std::endl;
  std::cout << LL->first->next->data << std::endl;
  std::cout << LL->first->next->next->data << std::endl;
}
```

builds "[5, 7, 3]"

//outputs 5

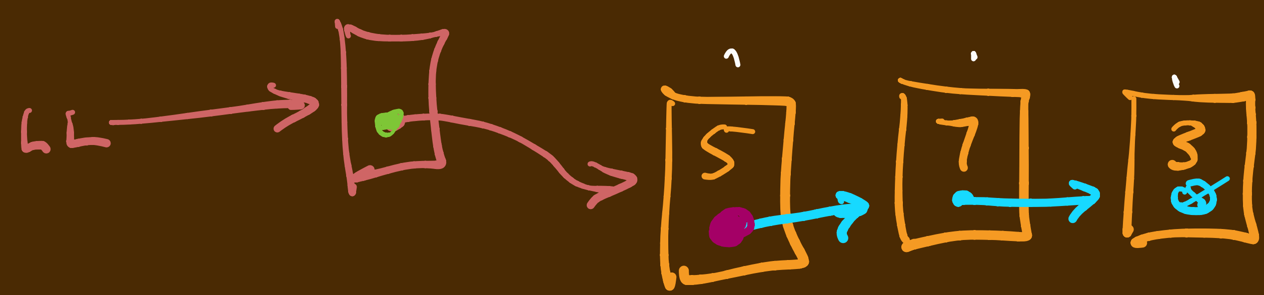
//outputs 7

//outputs 3

SOME LINKED LIST CODE

Consider this code. What does it do?

```
struct node {...};
struct llist {...};
```




```
int main(void) {
  node* c = new node {3, nullptr};
  node* b = new node {7, c};
  node* a = new node {5, b};
  llist* LL = new llist {a};

  node* current = LL->first;
  while (current != nullptr) {
    std::cout << current->data << std::endl;
    current = current->next;
  }
}
```

TRAVERSING A LIST: OUTPUT

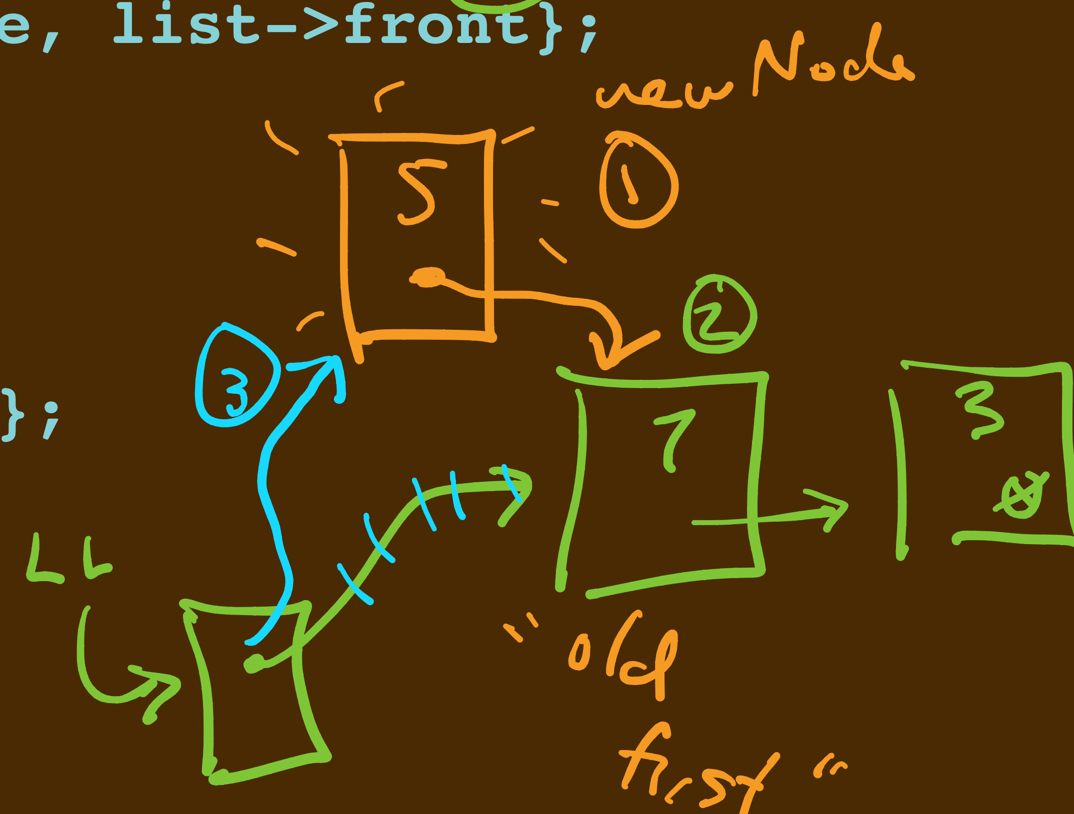
We can package that *list traversal* as a separate procedure:

```
struct node {...};
struct llist {...};

void output(llist* list) {
  node* current = list->first;
  while (current != nullptr) {
    std::cout << current->data << std::endl;
    current = current->next;
  }
}

int main(void) {
  node* c = new node {3, nullptr};
  node* b = new node {7, c};
  node* a = new node {5, b};
  llist* LL = new llist {a};
  output(LL);
}
```

BUILDING A LIST: ADDING AN ITEM IN FRONT

We can package the code that adds items as a separate procedure:

```
... // struct defs
void output(llist* list) {...}

void insertAtFront(int value, llist* list) {
  node* newNode = new node {value, list->front};
  list->front = newNode;
}

int main(void) {
  llist* LL = new llist {nullptr};
  insertAtFront(3, LL);
  insertAtFront(7, LL);
  insertAtFront(5, LL);
  output(LL);
}
```



GETTING THE LAST ITEM

Write the missing code:

```
... // struct defs
void output(llist* list) { ... }
void insertAtFront(int value, llist* list) { ... }

int outputLast(llist* list) {

  ????
}

int main(void) {
  llist* LL = new llist {nullptr};
  insertAtFront(3, LL);
  insertAtFront(7, LL);
  insertAtFront(5, LL);
  outputLast(LL);
}
```

ADDING AN ITEM ONTO THE END

Write the missing code:

```
... // struct defs
void output(llist* list) { ... }
void insertAtFront(int value, llist* list) { ... }
void outputLast(llist* list) { ... }

void insertAtEnd(int value, llist* list) {

  ????
}

int main(void) {
  llist* LL = new llist {nullptr};
  insertAtEnd(5, LL);
  insertAtEnd(7, LL);
  insertAtEnd(3, LL);
  output(LL);
}
```

NEXT

- ***MONDAY:***
  We'll continue to develop these ***linked list*** "methods."
  - We'll essentially build a class-like definition for linked lists.

- ***TOMORROW:***
  I'll post a Homework 04

- ***TONIGHT:***
  I'll post these annotated slides and also the linked list code.