Aggregate Data — `struct`

Rob Hackman

Winter 2024

University of Alberta

Table of Contents

Aggregate Data

We're writing a simple video game and want to check if two objects in our game world touch eachother.

In order to do this we put an imaginary rectangle around each object in our world, and check if the rectangles overlap.

For each rectangle we need to know its height, width, and the x and y coordinates for where in our world its located.

The `collides` function

So let's try and write our function `collides` that returns true if two rectangles overlaps.

```
int collides(int h1, int w2, int x1, int y1,
             int h2, int w2, int x2, int y2) {
  ...
}
```

That is a lot of parameters, which also means a lot of arguments we have to pass when calling this function, which also means lots of places for mistakes to occur.

Aggregate Data

In Python you learned about *Classes* and *Objects*, that is because Python is an object-oriented language.

C is not object-oriented. C does not have classes.

C does support *structures*, which are just a way to define an aggregate data type that groups together some other data types.

Defining the rectangle struct

Here we define a structure type named `Rect`.

```
1   struct Rect {
2     int x, y, w, h;
3   }; // semicolon important here!
```

Structures are *not* objects. We don't have constructors, we don't have methods, we only declare the data we're grouping together. A structure is just a way to give a name to a particular set of data types in a particular order.

General `struct` definition

In C you may define your `structs` as such:

```
struct yourDataType {
  T fieldName1;
  Q fieldName2;
  ...
}; // Semicolon necessary!!!
```

Where `T` and `Q` are types.

Using structures

Here we have a declaration of the variable `r1` whose type is a `Rect` structure.

```
1       int main() {
2         struct Rect r1; // Note struct in type name
3         r1.x = 5;
4         r1.y = 10;
5         r1.w = 0;
6         r1.h = 7;
7         printf("%lu\n", sizeof(r1));
8       }
```

We can also observe the dot operator for member access. We use the dot operator when the operand on the left hand side is a structure, the right hand side should be the name of the field we want to access.

Structures and Functions

Let's write our `collides` function

```
1   int collides(struct Rect r1, struct Rect r2) {
2     return (r1.x < r2.x + r2.w && // r1 is not right of r2
3             r1.x + r1.w > r2.x && // r1 is not left of r2
4             r1.y < r2.y + r2.h && // r1 is not below r2
5             r1.y + r1.h > r2.y) // r1 is not above r2
6
7   }
```

Structures as parameters

Let's write a function `translate` that takes a `Rect` and an `x` and `y` amount to translate the rectangle by, and translates the rectangle by that amount.

Structures as parameters

Let's write a function `translate` that takes a `Rect` and an `x` and `y` amount to translate the rectangle by, and translates the rectangle by that amount.

```
1  void translate(struct Rect r, int x, int y) {
2    r.x = r.x + x;
3    r.y = r.y + y;
4  }
```

Structures as parameters

Let's write a function `translate` that takes a `Rect` and an `x` and `y` amount to translate the rectangle by, and translates the rectangle by that amount.

```
1  void translate(struct Rect r, int x, int y) {
2    r.x = r.x + x;
3    r.y = r.y + y;
4  }

5  int main() {
6    struct Rect r1 = {.x=1,.y=5,.h=1,.w=1};
7    translate(r1, 3, -2);
8    printf("Rect at (%d, %d)\n", r1.x, r1.y);
9    // r1 hasn't changed... why?
10 }
```

Structure memory layout

Unlike arrays, C is perfectly happy to copy an structure to and from a functions stack frame.

Structure memory layout

Unlike arrays, C is perfectly happy to copy an structure to and from a functions stack frame.

What do structures look like in memory?

Structure memory layout

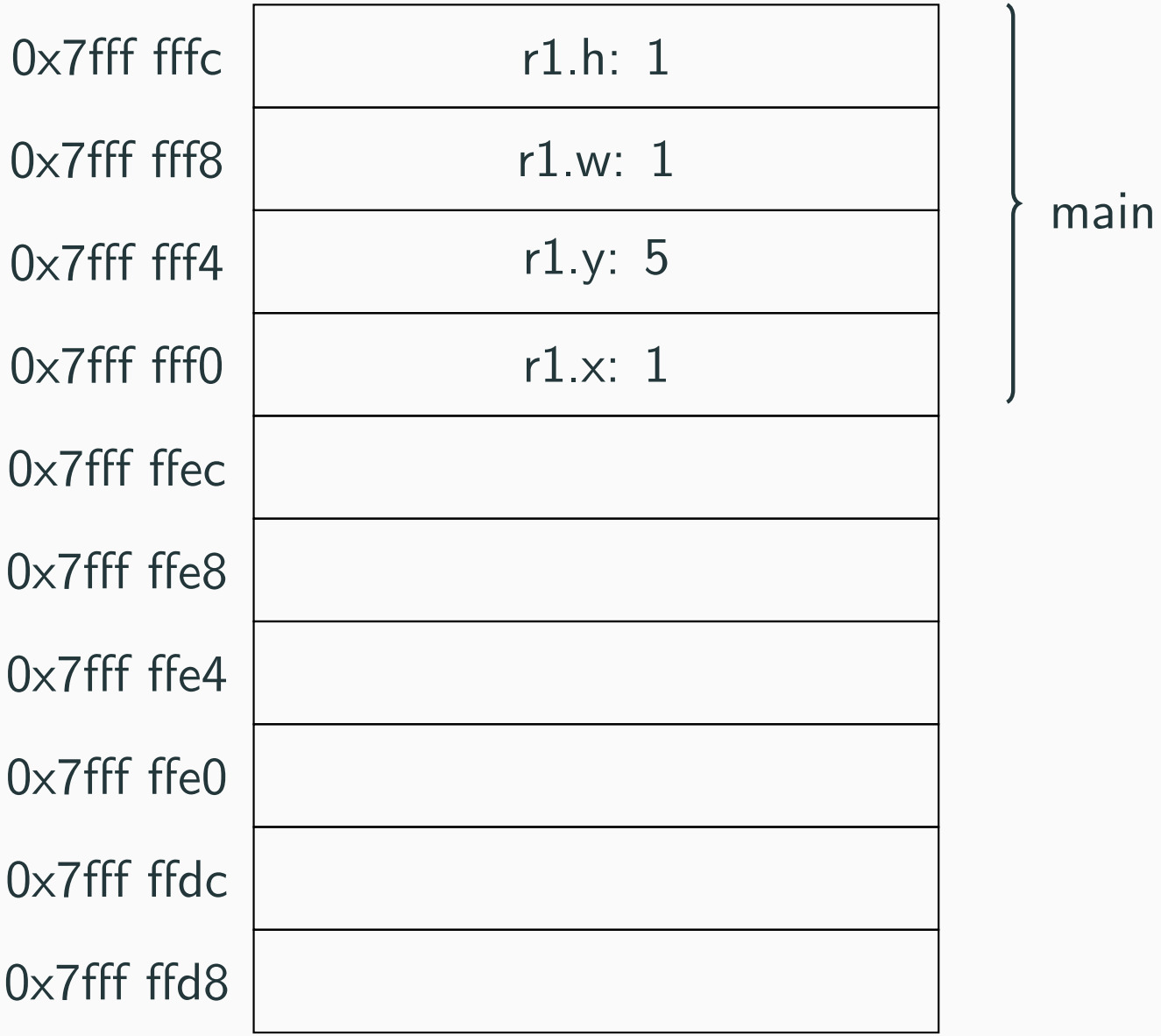
Unlike arrays, C is perfectly happy to copy an structure to and from a functions stack frame.

What do structures look like in memory?

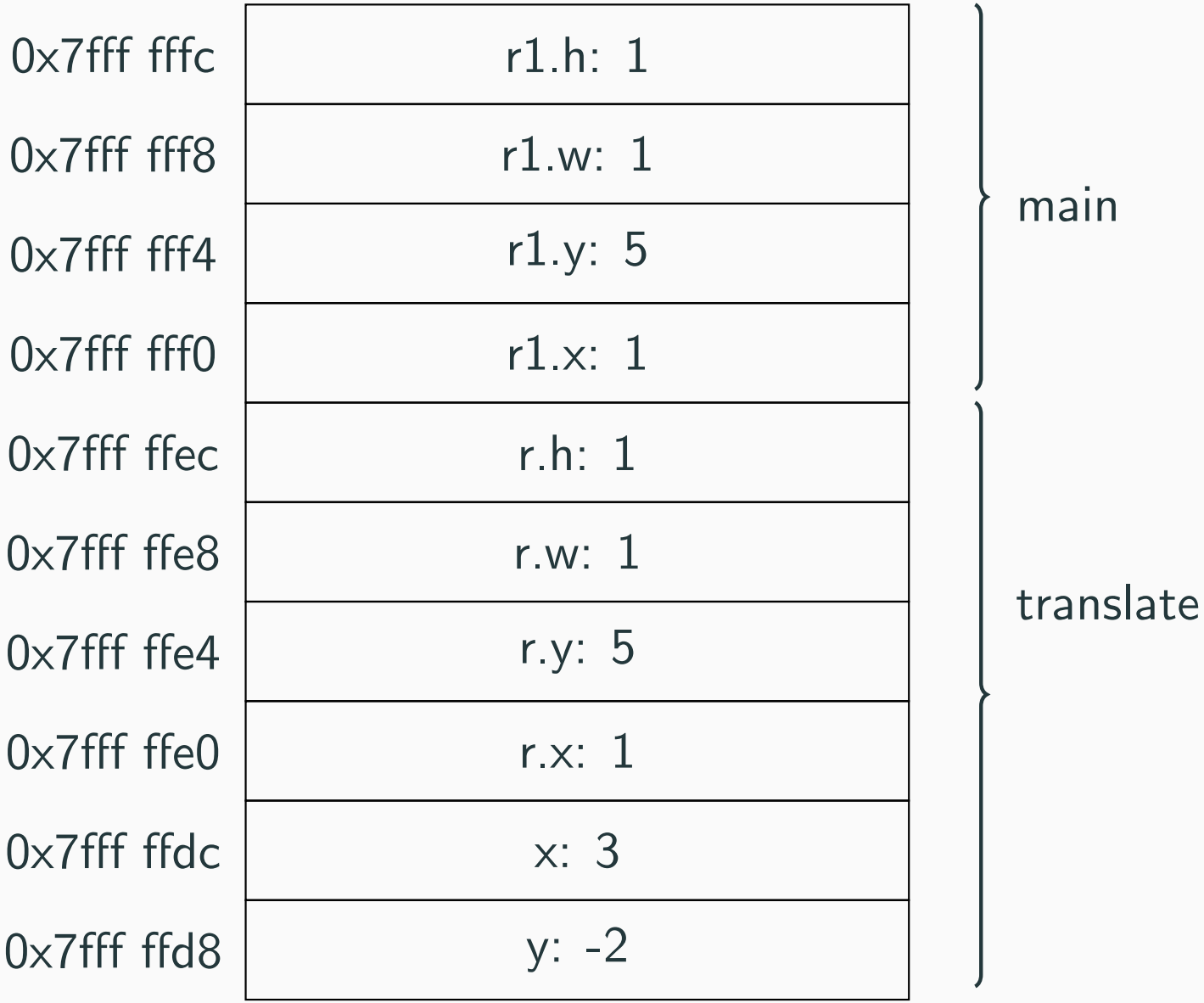
Structures fields are contiguous[1] in memory. They are laid out in the order they were declared in the `struct` definition.

[1] There may be *padding* in-between the fields for byte alignment.

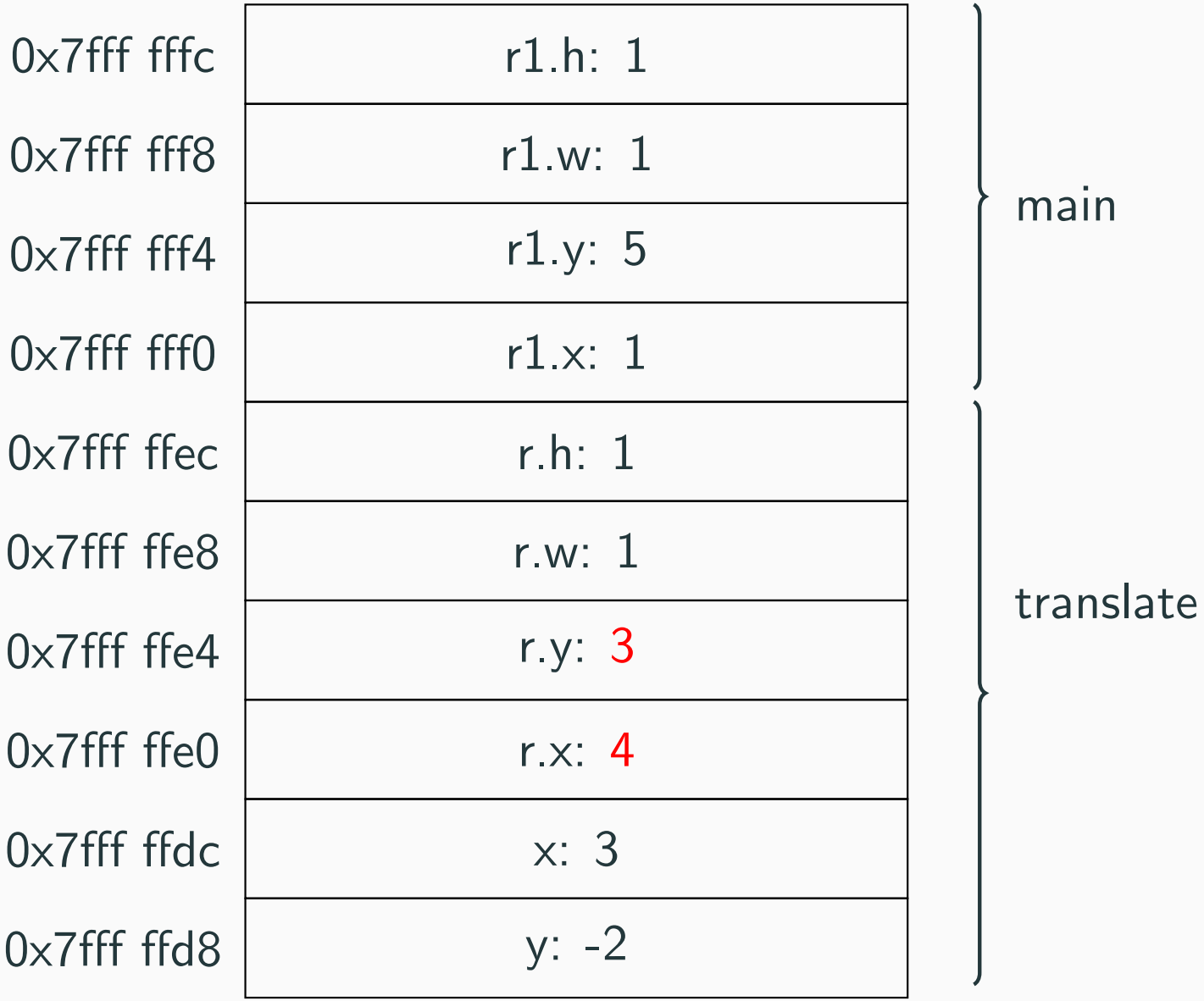
Structure memory layout



Structure memory layout



Structure memory layout



Passing pointers to structures

We wanted the `translate` function to mutate our structure.

As before, if we want the function to be able to mutate our local data we must provide the address of our data.

Also, copying large structures is inefficient. Passing a pointer is fast. In C programmers will often pass structures by pointer to save copying costs.

If we don't want to mutate the structure we pass it by a pointer to a `const` structure.

Fixed `translate`

```
1   void translate(struct Rect *r, int x, int y) {
2     // Parentheses necessary...
3     // Member of operator binds tighter
4     // than the dereference operator... annoying.
5     (*r).x = (*r).x + x;
6     (*r).y = (*r).y + y;
7   }
8   int main() {
9     struct r1 = {.x=1,.y=5,.h=1,.w=1};
10    translate(&r1, 3, -2);
11    printf("Rect at (%d, %d)\n", r1.x, r1.y);
12  }
```

The arrow operator

The dot operator cannot be applied to pointers — so we cannot write `r.x` when `r` is a pointer.

To get the structure `r` points at we dereference it, but the dot operator binds tighter than the dereference operator. So, `*r.x` would mean dereference the `x` field of the `r` structure, but in our case `r` is not a struct so we cannot access its field.

The designers of the language acknowledge it would be annoying to write `(*r).x` every time. So, there is an operator specifically for member access through a pointer — *the arrow operator*.

The arrow operator in use

In general the expression `p->f` is short hand for `(*p).f` where `p` is a pointer to a structure, and `f` is the name of a field of that structure.

```
1  void translate(struct Rect *r, int x, int y) {
2    r->x = r->x + x;
3    r->y = r->y + y;
4  }
```

Practice Questions

Create a `struct` to store our array of integers.

Update `push` and `pop` so that they work on the `struct` type.
You'll find they're much easier to write!

Table of Contents

Abstract Data Types

An abstract data type (ADT) is an abstract data type for use in a program.

The user of an ADT should not need to know the implementation of an ADT, but rather the behaviour it provides and only interact with it through the provided interface.

In Python `list`, `dict`, `tuple`, etc. are all ADTs.

Let's build a List ADT like Python's.

What our `List` ADT should provide

Users of our `List` ADT should be able to:

- Create an empty list
- Append items to a list
- Remove items from a list
- Access items in a list

Our `List` ADT

We'll call our ADT `List`. Though it won't be as general as Python `lists` as we will still only store one data type.

We'll implement our ADT with a *linked list*.
A linked list is a simple data structure that is made up of *nodes*.
Each node stores two things: an item in the list, and a pointer to the next node.

```
1  struct Node {
2    int data;
3    struct Node *next;
4  };
5  struct List {
6    struct Node *head;
7    size_t len;
8  };
```

Linked Lists

We'll draw linked lists as simple *box and pointer diagrams*.

The diagram below shoes a linked list that represents the list `[17, 32, 7]`.

Each pair of boxes represents a `Node`. The values in the first box represent the data that node stores, the arrow represents the pointer to the next node.

A function for intialization

For the user to not need to know how our ADT is implemented we need to provide them with functions to call for the operations they would desire.

That includes initializing the fields (in fact in a true ADT the user shouldn't even know what the fields are). So we'll provide a function for getting a new `List`.

```
1  struct List *createList() {
2    struct List *ret = malloc(sizeof(struct List));
3    ret->head = NULL;
4    ret->len = 0;
5    return ret;
6  }
```

A function for intialization

```
1  struct List *createList() {
2    struct List *ret = malloc(sizeof(struct List));
3    ret->head = NULL;
4    ret->len = 0;
5    return ret;
6  }
```

Why return a pointer? One reason is, as mentioned, it is typically more efficient to copy just a pointer to the caller's stackframe rather than a (potentially) large object.

Perhaps the more important reason we'll see later.

The `addToFront` function

Let's write a function, `addToFront`, for adding to the front of an existing `List`.

The `addToFront` function

Let's write a function, `addToFront`, for adding to the front of an existing `List`.

```
1  struct List *addToFront(struct List *l, int val) {
2    struct Node *node = malloc(sizeof(struct Node));
3    node->data = val;
4    node->next = l->head;
5    l->head = node;
6    l->len = l->len+1;
7    return l;
8  }
```

The `ith` function

Let's write a function, `ith`,

The `ith` function

Let's write a function, `ith`,

```
1   int ith(struct List *l, size_t ind) {
2     assert(ind < l->len);
3     struct Node *cur = l->head;
4     for (size_t i = 0; i < ind; ++i, cur=cur->next);
5     return cur->data;
6   }
```

Note: `ith` is $\mathcal{O}(n)$ which is very poor — it should be constant. This problem is only solved in C by exposing the implementation of our ADT (the nodes) or using a different underlying data structure. In C++ we'll see other solutions.

The `ith` function

Let's write a function, set

The `ith` function

Let's write a function, set

```
1   int ith(struct List *l, size_t ind) {
2     assert(ind < l->len);
3     struct Node *cur = l->head;
4     for (size_t i = 0; i < ind; ++i, cur=cur->next);
5     return cur->data;
6   }
```

Note: `ith` is $\mathcal{O}(n)$ which is very poor — it should be constant. This problem is only solved in C by exposing the implementation of our ADT (the nodes) or using a different underlying data structure. In C++ we'll see other solutions.

The `setElem` function

`ith` allows the user to access the value of an element, let's provide a function to mutate the value at an index.

The `setElem` function

`ith` allows the user to access the value of an element, let's provide a function to mutate the value at an index.

```
1   int setElem(struct List *l, size_t ind, int val) {
2     assert(ind < l->len);
3     struct Node *cur = l->head;
4     for (size_t i = 0; i < ind; ++i, cur=cur->next);
5     cur->data = val;
6   }
```

Note: this suffers from the same problem as `ith` and is $\mathcal{O}(n)$. Same solutions apply as to `ith`.

The `removeItem` function

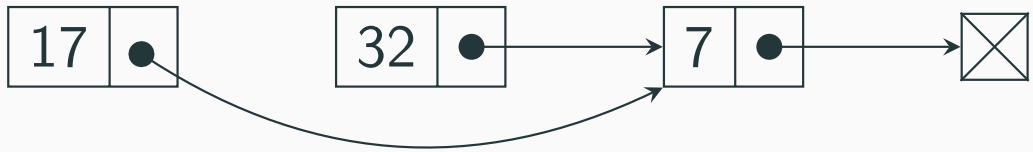
`removeItem` allows the user to remove an item at a particular index from the list. How do we achieve this? We must be careful to remove the old node from the list as well as freeing it.

If we want to remove the second item (32) from the list below, how do we do it?

The `removeItem` function

`removeItem` allows the user to remove an item at a particular index from the list. How do we achieve this? We must be careful to remove the old node from the list as well as freeing it.

If we want to remove the second item (32) from the list below, how do we do it?



Updating the node before the one we want to remove to point at the node *after* the one we want to remove effectively removes the desired node from the chain. We must be careful to not forget to free this removed node, however!

The `removeItem` function

`removeItem` allows the user to remove an item at a particular index from the list. How do we achieve this? We must be careful to remove the old node from the list as well as freeing it.

If we want to remove the second item (32) from the list below, how do we do it?

Updating the node before the one we want to remove to point at the node *after* the one we want to remove effectively removes the desired node from the chain. We must be careful to not forget to free this removed node, however!

The `removeItem` function

```
1  struct List *removeItem(struct List *l, size_t ind) {
2    assert(ind < l->len);
3    if (ind == 0) {
4      struct Node *tmp = l->head;
5      l->head = l->head->next;
6      free(tmp);
7      l->len = l->len-1;
8      return l;
9    }
10   struct Node *cur = l->head;
11   struct Node *prev = NULL;
12   for (size_t i = 0; i < ind; ++i, prev=cur, cur=cur->next);
13   prev->next = cur->next;
14   free(cur);
15   l->len = l->len-1;
16   return l;
17 }
```

Freeing a linked list

To free our linked list we must make sure to free *every* `Node`, as well as freeing the `List` structure itself.

In order to free a `Node`, we must make sure to either first record its `next` pointer, or have already freed the `Node` after it. If we free a `Node` first, and then try to acces its `next` pointer, we would be accessing the field of a structure that we already freed (accessing a dangling pointer).

One of the most effective methods to freeing an entire linked is is recursion.

Freeing a linked list — code

```
1  void deleteNode(struct Node *n) {
2    if (!n) return;
3    // Recursively delete our next node
4    // BEFORE freeing this node.
5    deleteNode(n->next);
6    free(n);
7  }
8
9  void deleteList(struct List *l) {
10   if (!l) return;
11   deleteNode(l->head);
12   free(l);
13 }
```

Using our ADT

Now, we've build our `List` ADT and our client progrmamer can use it with our provided interface.

```
1  int main() {
2    struct List *l = createList();
3    addToFront(l, 3);
4    addToFront(l, 2);
5    addToFront(l, 1);
6    for (size_t i = 0; i < len(l); ++i) {
7      printf("%d\n", ith(l, i));
8    }
9    deleteList(l);
10 }
```

Too exposed

We have a problem. Currently our ADT is revealed to the client programmer and they can see and use all the details.

This means the client programmer could use our ADT improperly, for example:

```
1  int main() {
2    struct List l; struct Node n1;
3    struct Node n2;
4    n1.next = &n2; n2.next = &n1;
5    l.head = n1;
6  }
```

Too exposed

We have a problem. Currently our ADT is revealed to the client programmer and they can see and use all the details.

This means the client programmer could use our ADT improperly, for example:

```
1  int main() {
2    struct List l; struct Node n1;
3    struct Node n2;
4    n1.next = &n2; n2.next = &n1;
5    l.head = n1;
6  }
```

The client shouldn't even be able to create `List` structures directly, or `Node` structures at all! They certainly shouldn't be able to modify the fields of these structures.

Reusability

Also, our `List` structure is an ADT. The purpose of an ADT is to be a general data type that could apply to many scenarios. We'd like to be able to use our `List` ADT in several programs.

Right now that means copying our structure definition, and all functions, to each program that uses it.

What if we decide to change how our `List` is implemented? We'd have to change it in every program we ever used it. High chance we make a mistake in one of those programs, or forget one.

How can we hide our ADT's details from the client programmer, and make the ADT avaialble to any program we want to use it in?