Yet more about pointers…

Within, Map, Filter and Reduce

Passing Function Pointers into other Functions

- ***Question.*** Why?
- ***Answers***
  - give us the ability to have a programer that is using our function to tailor it to their needs (called a *callback*)
  - helps us generalize code (as does the ADT)
  - reduces code redundancy
- ***Example***
  - Important missing generalization of the List ADT

List ADT Interface

Operations previously defined for the List ADT

- `List * create_list ()`
- `int  size (list)`
- `int  is_empty (list)`
- `int  push (list, value)`
- `int  append (list, value)`
- `int  add_after (list, value)`
- `int  in_list (list, value)`
- `int  remove_value (list, value)`
- `void print_list_all (list);`
- `void empty_list (list);`
- `void destroy_list(list);`

What is missing?

1.
2.
3.
4.
5.

List ADT Interface

Operations previously defined for the List ADT

- `List * create_list ()`
- `int  size (list)`
- `int  is_empty (list)`
- `int  push (list, value)`
- `int  append (list, value)`
- `int  add_after (list, value)`
- `int  in_list (list, value)`
- `int  remove_value (list, value)`
- `void print_list_all (list);`
- `void empty_list (list);`
- `void destroy_list(list);`

What is missing?

Getting the stored values back out without "guessing"

Basic Function to Complete the List ADT

- **first**, last, nth

```
int first( List * list , value_t * value) {
   int result = FAILURE;
   Node * node = list->head;
   if ( node != NULL ) {
      *value = node->value;
      result = SUCCESS;
   }
   return result;
}
```

Basic Function to Complete the List ADT

- first, **last**, nth

```
int last ( List * list , value_t * value) {
   int result = FAILURE;
   Node * node = list->tail;
   if ( node != NULL ) {
      *value = node->value;
      result = SUCCESS;
   }
   return result;
}
```

Basic Function to Complete the List ADT

- first, last, **nth**

```
int nth  ( List * list , int n, value_t * value) {
   int result = FAILURE;
   Node * node = nth_node(list->head, n);
   if ( node != NULL ) {
      *value = node->value;
      result = SUCCESS;
   }
   return result;
}

Node * nth_node ( Node * node , int n) {
   for ( i = 1 ; i < n && node != NULL && node->next != NULL ; i++ )
      node = node->next;
   return node;
}
```

Basic Function to Complete the List ADT

- **remove_first**, remove_last, remove_nth

```
int remove_first( List * list , value_t * value) {
   int result = FAILURE;
   Node * node = list->head;
   if ( node != NULL ) {
      *value = node->value;
      result = SUCCESS;
      list->head = node->next;
      if ( node == list->tail ) list->tail = NULL;
      remove_node(node);
      free(node);
   }
   return result;
}
```

Basic Function to Complete the List ADT

- remove_first, **remove_last**, remove_nth

```
int remove_last ( List * list , value_t * value) {
   int result = FAILURE;
   Node * node = list->tail;
   if ( node != NULL ) {
      *value = node->value;
      result = SUCCESS;
      if ( node == list->head ) list->head = NULL;
      list->tail = node->prev;
      remove_node(node);
      free(node);
   }
   return result;
}
```

remove_last is more efficient if list also stores, the node pointing to the tail or is doubly linked

Basic Function to Complete the List ADT

- remove_first, remove_last, **remove_nth**

```
int remove_nth  ( List * list , int n, value_t * value) {
   int result = FAILURE;
   Node * node = nth_node(list->head, n);
   if ( node != NULL ) {
      *value = node->value;
      result = SUCCESS;
      if ( node == list->head ) list->head = node->next;
      if ( node == list->tail ) list->tail = node->prev;
      remove_node(node);
      free(node);
   }
   return result;
}
```

remove_nth is more efficient
if list is implemented
using a doubly linked list

Passing Function Pointers into other Functions

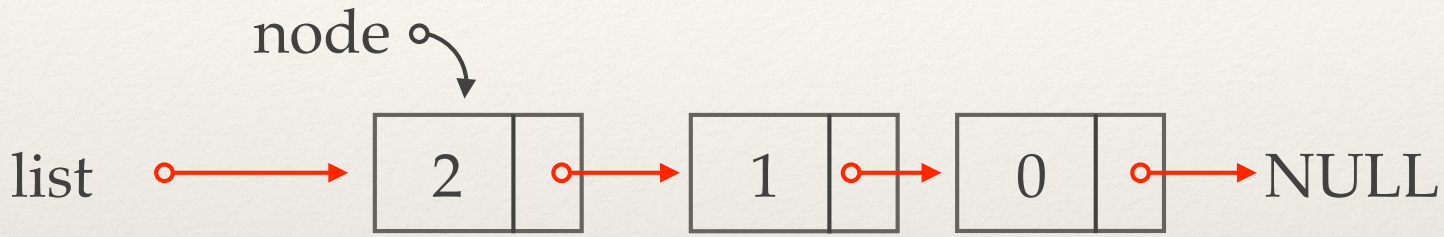
- ***Question.*** Why?
- ***Answers***
  - give us the ability to have a programer that is using our fn to tailor it to their needs (called a *callback*)
  - helps us generalize code (as does the ADT)
  - reduces code redundancy
- ***Example***
  - Important missing generalization of the List ADT
    - `within(list, void fn_ptr)`
    - `map, reduce, filter`

```
typedef struct element {
   value_t value;
   struct element *next;
} Node;
```

Stepping through a list

abstracted with function pointers

- Start at the **head** and follow the pointers / **links**!




```
Node * node = head;
while ( node != NULL ) {
   /*  code using the node goes here */
   node = node->next;
}
```

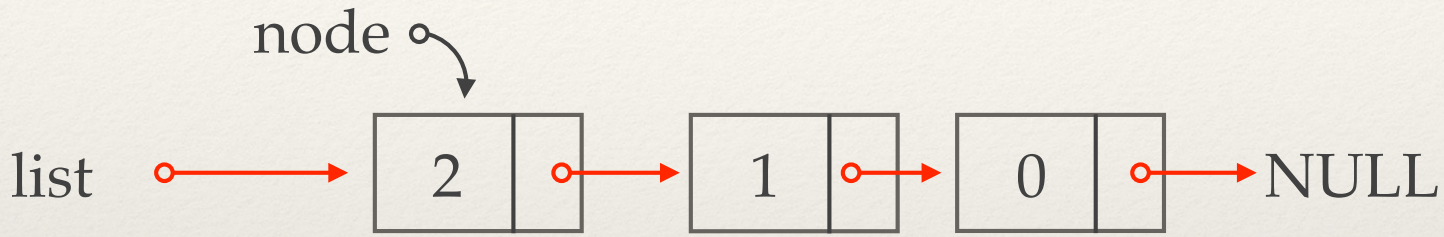
as originally taught (called a pattern)

```
typedef struct element {
   value_t value;
   struct element *next;
} Node;
```

Stepping through a list

abstracted with function pointers

- Start at the **head** and follow the pointers / **links**!



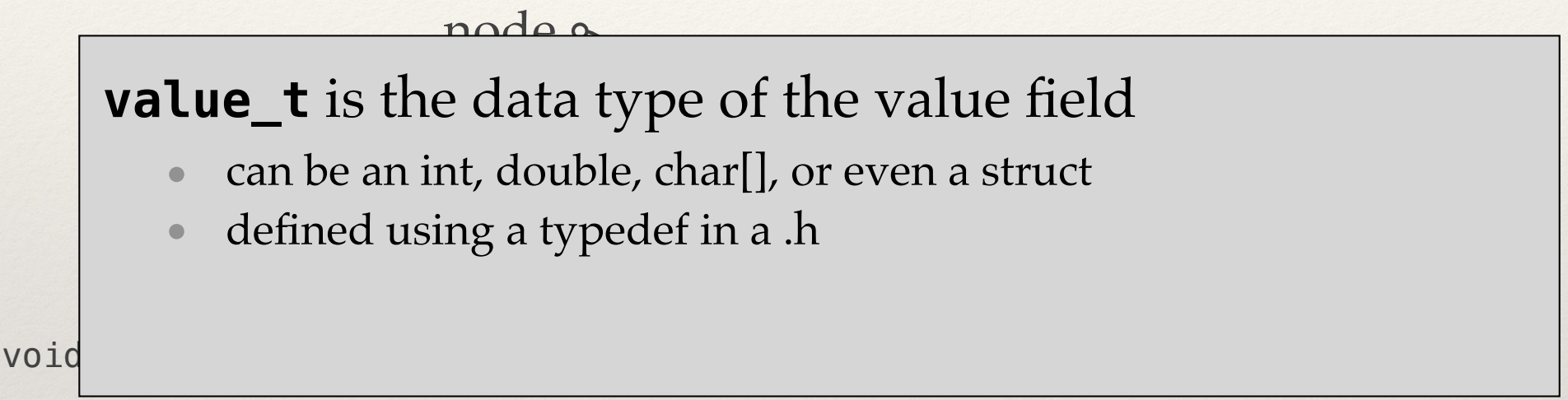

```
void within ( List * list, void (*fn_ptr)(value_t) )
   Node * node = list->head;
   while ( node != NULL ) {
      fn_ptr ( node->value );
      node = node->next;
   }
}
```

now generalized using a function pointer
to pass in externally written code
to be run for each node (aka "a callback")

```
typedef struct element {
   value_t value;
   struct element *next;
} Node;
```

Stepping through a list
abstracted with function pointers

- Start at the **head** and follow the pointers / **links**!



`value_t` is the data type of the value field

- can be an int, double, char[], or even a struct
- defined using a typedef in a .h

```
void
   Node * node = list->head;
   while ( node != NULL ) {
      fn_ptr ( node->value );
      node = node->next;
   }
}
```

```
typedef struct element {
   value_t value;
   struct element *next;
} Node;
```

Stepping through a list
abstracted with function pointers

- Start at the **head** and follow the pointers / **links**!

node a

`value_t` is the data type of the value field

- as the List ADT would exist in its own file,
  if you want to change the value_t datatype (say from int to char[])
  you need to recompile it to create a new listADT.o to link with
- Consequently you need access to the original List ADT source code

```
void
   Node * node = list->head;
   while ( node != NULL ) {
      fn_ptr ( node->value );
      node = node->next;
   }
}
```

```
typedef struct element {
   value_t value;
   struct element *next;
} Node;
```

Stepping through a list
abstracted with function pointers

- Start at the **head** and follow the pointers / **links**!

node *

`value_t` is the data type of the value field

- There are more flexible ways of generalizing the code, but tricky techniques such as using void * with casting (*see next section*)

```
void
   Node * node = list->head;
   while ( node != NULL ) {
      fn_ptr ( node->value );
      node = node->next;
   }
}
```

```
typedef struct element {
   value_t value;
   struct element *next;
} Node;
```

Stepping through a list

abstracted with function pointers

- Start at the **head** and follow the pointers / **links**!

list ... NULL

```
void within ( List
   Node * node = l
   while ( node !=
      fn_ptr ( noc
      node = node-
   }
}
```

Using within to print a list

using `typedef value_t int;`

```
void print_list_all(List * list) {
   printf("%s", "< “);
   within(list, print_value);
   printf("%s", ">\n ");
}

void print_value( int value) {
    printf ( “d ", value );
}
```

Problem: Can only perform with “side-effect” such as print

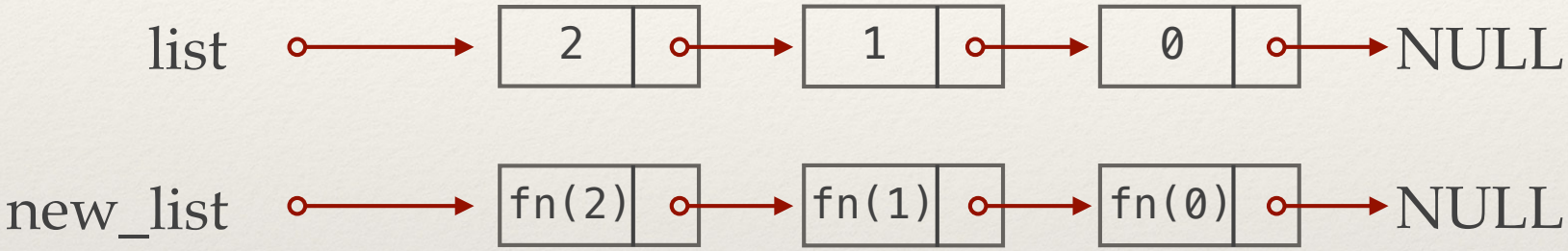
Need to apply the code and then get information out of the list

```
typedef struct element {
   value_t value;
   struct element *next;
} Node;
```

Stepping through a list

abstracted with function pointers

map: creates a new list where the value of each node is equal to the value from the node at the same position in the original list after the function has been applied




```
List * map ( List * list, value_t (*fn_ptr)(value_t) )
   Node * node = list->head;
   List * new_list = create_list();

   for ( i = 0; i < size(list) ; i++) {
        append ( new_list, fn_ptr ( node->value ) );
        node = node->next;
   }

   return new_list;
}
```

Stepping through a list
abstracted with function pointers

```
typedef value_t int;

value_t exp2(value_t value){
  return exp(2, value);
}

value_t exp3(value_t value){
  return exp(3, value);
}

inline int exp(int x, int y){
   return (int) pow((double)x,(double)y);
}
```

map in use: simple example

```
void main() {
   List * list = create_list();
   List * list_e2 = create_list();
   List * list_e3 = create_list();

   for (i=0; i < 5; i++)
      push(list1, i);

   list_e2 = map(list, exp2);

   list_e3 = map(list, exp3);

   print_list_all(list);
   print_list_all(list_e2);
   print_list_all(list_e3);

   destroy_list(list);
   destroy_list(list_e2);
   destroy_list(list_e3);
}
```

output

```
< 4, 3, 2, 1, 0 >
< 16, 8, 4, 2, 1 >
< 81, 27, 9, 3, 1 >
```

```
typedef struct element {
   value_t value;
   struct element *next;
} Node;
```

Stepping through a list
abstracted with function pointers

map: creates a new list where the value of each node is equal to the value from the node at the same position in the original list after the function has been applied

Limitation: this code can only map onto a list of the same value_t data type

Potential Solⁿ: replace value_t with a void pointer (covered later)
the void ptr is then cast to the data type required as the new list is used

```
List *
   Nod
   List
   for ( i = 0; i < size(list) ; i++) {
        append ( new_list, fn_ptr ( node->value ) );
        node = node->next;
   }

   return new_list;
}
```

```
typedef struct element {
   value_t value;
   struct element *next;
} Node;
```

Stepping through a list
abstracted with function pointers

map: creates a new list where the value of each node is equal to the value from the node at the same position in the original list after the function has been applied

map uses function pointers
to transform one list into another of the same size

However, this is not the only reason
to apply a function pointer to a list

```
List *
   Node
   List

   for ( i = 0; i < size(list) ; i++) {
        append ( new_list, fn_ptr ( node->value ) );
        node = node->next;
   }

   return new_list;
}
```

```
typedef struct element {
   value_t value;
   struct element *next;
} Node;
```

Stepping through a list

abstracted with function pointers

map: creates a new list where the value of each node is equal to the value from the node at the same position in the original list after the function has been applied

Rather, the number of elements in the list can be changed

A function can decide which elements stay and which go

This is called a filter

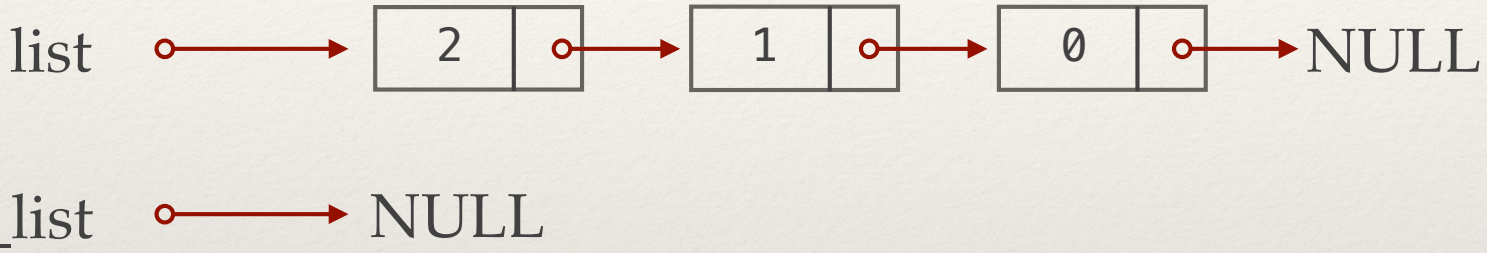
```
List *
   Node
   List
   for ( i = 0; i < size(list) ; i++) {
        append ( new_list, fn_ptr ( node->value ) );
        node = node->next;
   }
   return new_list;
}
```

```
typedef struct element {
   value_t value;
   struct element *next;
} Node;
```

Stepping through a list

abstracted with function pointers

filter: creates a new list where a node from the original list is copied and added if the *filter function* returns TRUE else the node is skipped



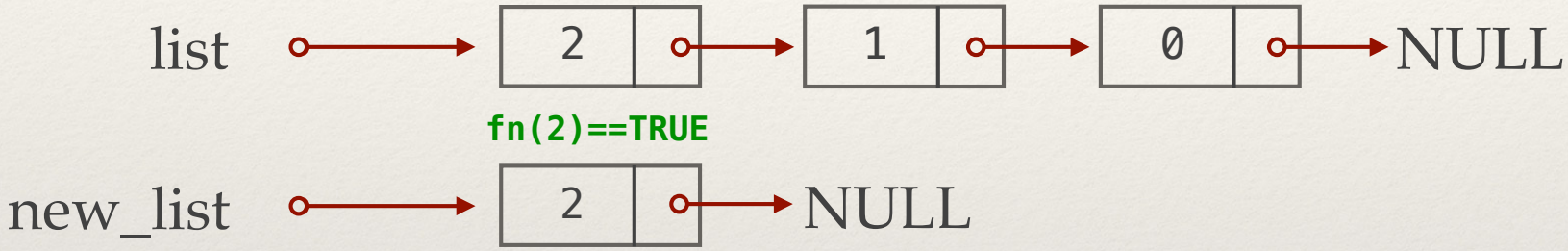

```
List * filter(List * list, int (*filter_fn)(value_t) )
    Node * node = list->head;
    List * new_list = create_list();
    for ( i = 0; i < size(list) ; i++) {
        if (filter_fn(node) == TRUE)
            append ( new_list, node->value );
        node = node->next;
    }
    return new_list;
}
```

```
typedef struct element {
   value_t value;
   struct element *next;
} Node;
```

Stepping through a list

abstracted with function pointers

filter: creates a new list where a node from the original list is copied and added if the *filter function* returns TRUE else the node is skipped



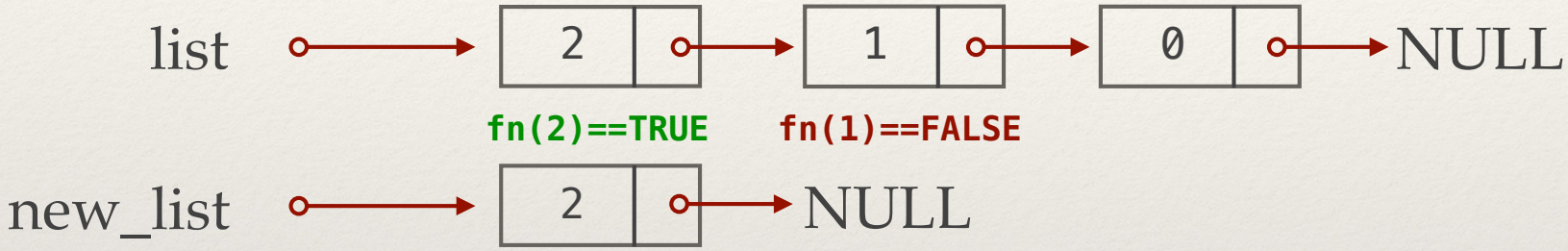

```
List * filter(List * list, int (*filter_fn)(value_t) )
    Node * node = list->head;
    List * new_list = create_list();
    for ( i = 0; i < size(list) ; i++) {
        if (filter_fn(node) == TRUE)
            append ( new_list, node->value );
        node = node->next;
    }
    return new_list;
}
```

```
typedef struct element {
   value_t value;
   struct element *next;
} Node;
```

Stepping through a list

abstracted with function pointers

filter: creates a new list where a node from the original list is copied and added if the *filter function* returns TRUE else the node is skipped



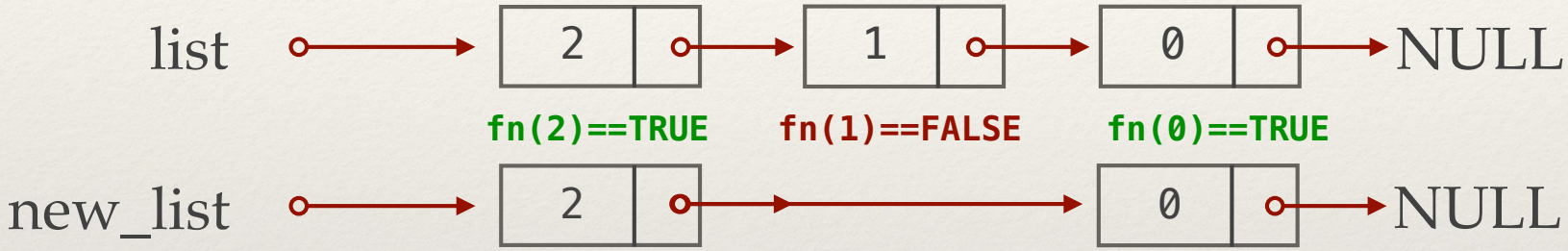

```
List * filter(List * list, int (*filter_fn)(value_t) )
    Node * node = list->head;
    List * new_list = create_list();
    for ( i = 0; i < size(list) ; i++) {
        if (filter_fn(node) == TRUE)
            append ( new_list, node->value );
        node = node->next;
    }
    return new_list;
}
```

```
typedef struct element {
   value_t value;
   struct element *next;
} Node;
```

Stepping through a list

abstracted with function pointers

filter: creates a new list where a node from the original list is copied and added if the *filter function* returns TRUE else the node is skipped




```
List * filter(List * list, int (*filter_fn)(value_t) )
    Node * node = list->head;
    List * new_list = create_list();
    for ( i = 0; i < size(list) ; i++) {
        if (filter_fn(node) == TRUE)
            append ( new_list, node->value );
        node = node->next;
    }
    return new_list;
}
```

Stepping through a list

abstracted with function pointers

```
typedef value_t int;

int odd (Node * node){
  return node->value % 2;
}

int even (Node * node){
  return !(node->value % 2);
}

void fib(int *curr, int *prev){
      int new = *curr + *prev;
      *prev = *curr;
      *curr = new;
}
```

```
void main() {
   List * fib_seq  = create_list();
   List * fib_odd  = create_list();
   List * fib_even = create_list();
   int value = 0, prev = 0;

   while (value < 50){
      append(fib_seq, value);
      fib(&value, &prev);
   }

   fib_odd  = filter(fib_seq, odd);
   fib_even = filter(fib_seq, even);

   print_list_all(fib_seq);
   print_list_all(fib_odd);
   print_list_all(fib_even);

   destroy_list(fib_seq);
   destroy_list(fib_odd);
   destroy_list(fib_even);
}
```

output

```
< 0, 1, 1, 2, 3, 5, 8, 13, 21, 34 >

< 1, 1, 3, 5, 13, 21 >

< 0, 2, 8, 34 >
```

```
typedef struct element {
   value_t value;
   struct element *next;
} Node;
```

Stepping through a list

abstracted with function pointers

filter: creates a new list where a node from the original list is copied and added if the *filter function* returns TRUE else the node is skipped

One more way to process a list:

Combine or accumulate the values of a list into a single value

called reduce

```
List * filte
    Node * node = list->head;
    List * new_list = create_list();
    for ( i = 0; i < size(list) ; i++) {
        if (filter_fn(node) == TRUE)
            append ( new_list, node->value );
        node = node->next;
    }
    return new_list;
}
```

```
typedef struct element {
   value_t value;
   struct element *next;
} Node;
```

Stepping through a list
abstracted with function pointers

reduce: step through a list and accumulate the values

- by applying a reducing function to the value of the node and the current accumulator value
- the accumulator is set to the result of the reducing fn, becoming the input for the reduction at the next node

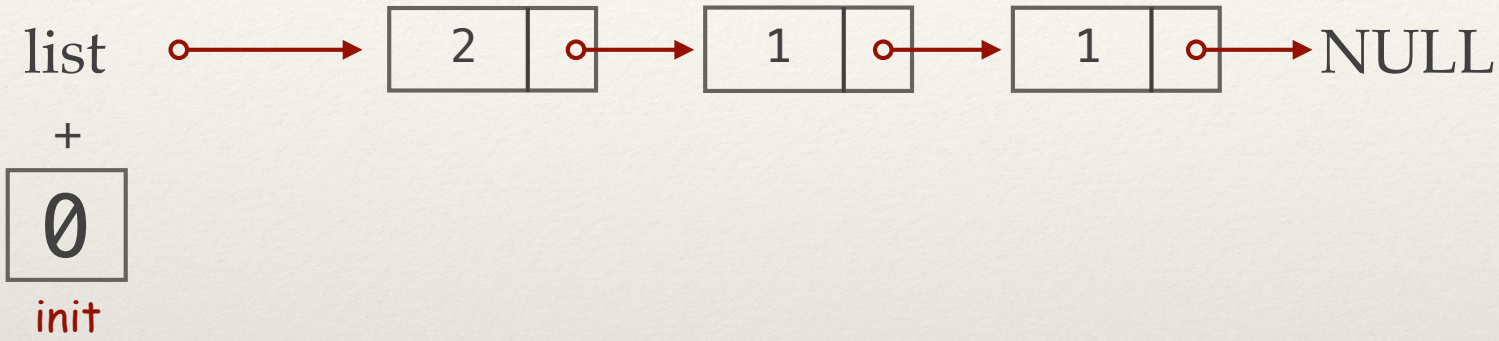
```
value_t reduce(List * list, value_t (*reduce)(value_t, value_t), value_t init )
   Node * node = list->head;
   value_t result = init;
   for ( i = 0; i < size(list) ; i++) {
      result = reduce( node->value, result );
      node = node->next;
   }
   return result;
}
```

```
typedef struct element {
   value_t value;
   struct element *next;
} Node;
```

Stepping through a list

abstracted with function pointers

reduce: step through a list and accumulate the values



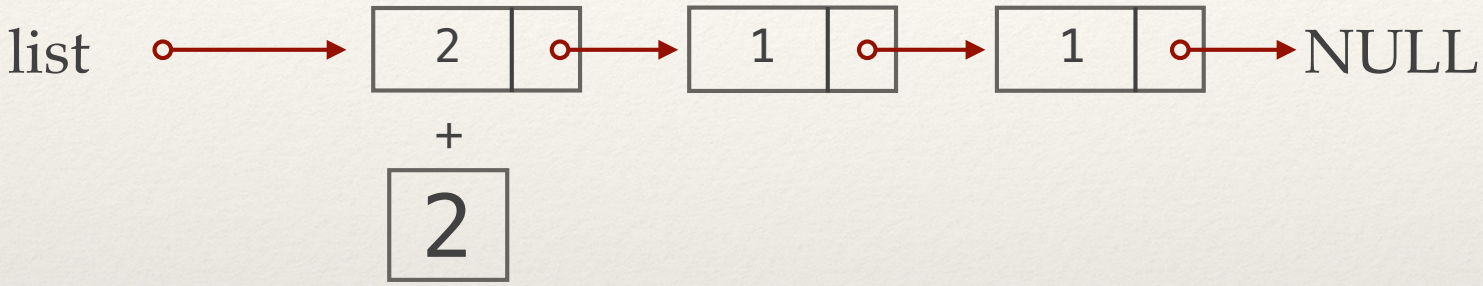

```
value_t reduce(List * list, value_t (*reduce)(value_t, value_t), value_t init )
   Node * node = list->head;
   value_t result = init;
   for ( i = 0; i < size(list) ; i++) {
      result = reduce( node->value, result );
      node = node->next;
   }
   return result;
}
```

```
typedef struct element {
   value_t value;
   struct element *next;
} Node;
```

Stepping through a list

abstracted with function pointers

reduce: step through a list and accumulate the values



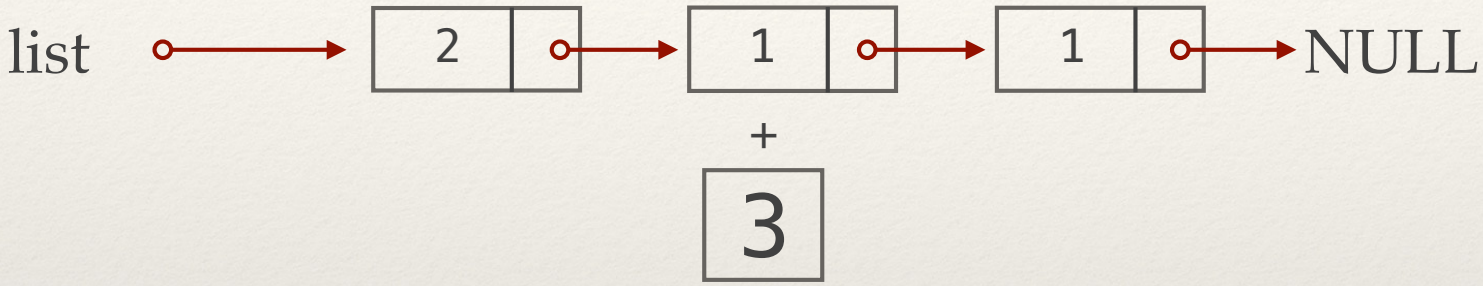

```
value_t reduce(List * list, value_t (*reduce)(value_t, value_t), value_t init )
   Node * node = list->head;
   value_t result = init;
   for ( i = 0; i < size(list) ; i++) {
      result = reduce( node->value, result );
      node = node->next;
   }
   return result;
}
```

```
typedef struct element {
   value_t value;
   struct element *next;
} Node;
```

Stepping through a list

abstracted with function pointers

reduce: step through a list and accumulate the values



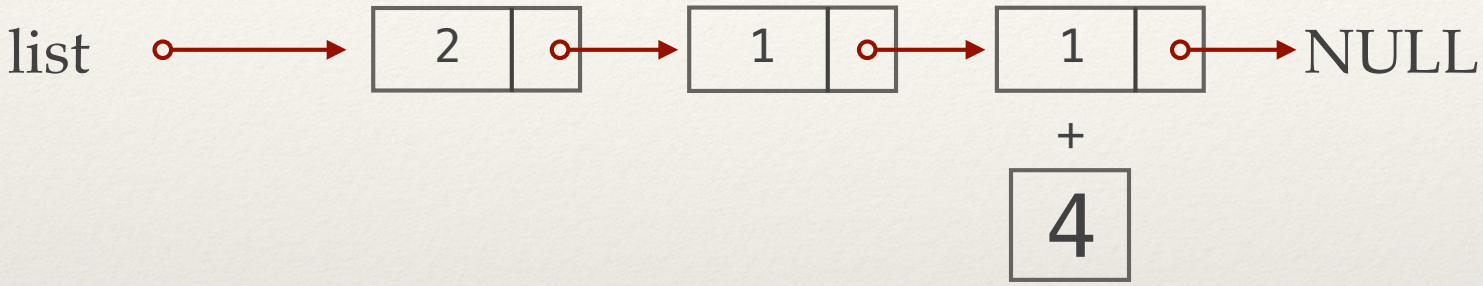

```
value_t reduce(List * list, value_t (*reduce)(value_t, value_t), value_t init )
   Node * node = list->head;
   value_t result = init;
   for ( i = 0; i < size(list) ; i++) {
      result = reduce( node->value, result );
      node = node->next;
   }
   return result;
}
```

```
typedef struct element {
   value_t value;
   struct element *next;
} Node;
```

Stepping through a list

abstracted with function pointers

reduce: step through a list and accumulate the values



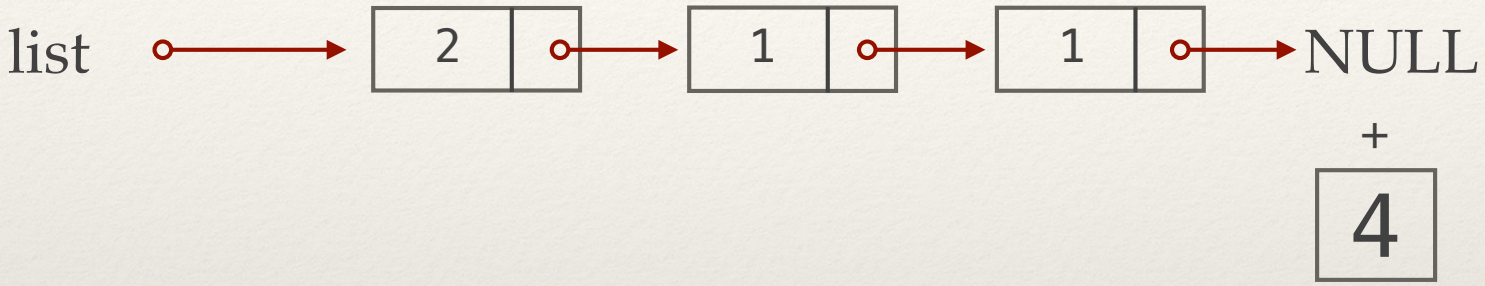

```
value_t reduce(List * list, value_t (*reduce)(value_t, value_t), value_t init )
   Node * node = list->head;
   value_t result = init;
   for ( i = 0; i < size(list) ; i++) {
      result = reduce( node->value, result );
      node = node->next;
   }
   return result;
}
```

```
typedef struct element {
   value_t value;
   struct element *next;
} Node;
```

Stepping through a list

abstracted with function pointers

reduce: step through a list and accumulate the values



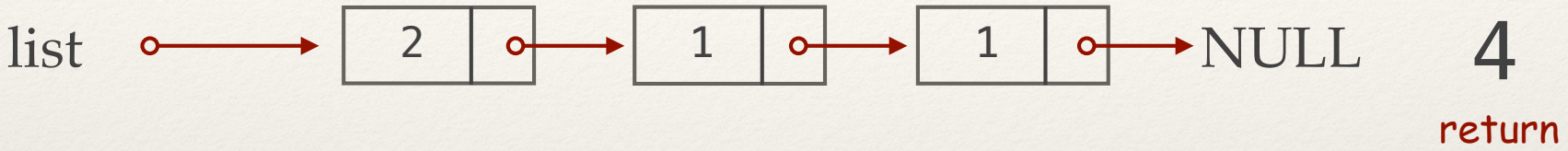

```
value_t reduce(List * list, value_t (*reduce)(value_t, value_t), value_t init )
   Node * node = list->head;
   value_t result = init;
   for ( i = 0; i < size(list) ; i++) {
      result = reduce( node->value, result );
      node = node->next;
   }
   return result;
}
```

```
typedef struct element {
   value_t value;
   struct element *next;
} Node;
```

Stepping through a list

abstracted with function pointers

reduce: step through a list and accumulate the values




```
value_t reduce(List * list, value_t (*reduce)(value_t, value_t), value_t init )
   Node * node = list->head;
   value_t result = init;
   for ( i = 0; i < size(list) ; i++) {
      result = reduce( node->value, result );
      node = node->next;
   }
   return result;
}
```

Stepping through a list

abstracted with function pointers

```
typedef value_t int;

value_t sum(List * list){
  return reduce(list, add, 0);
}

value_t sum_sq(List * list){
  return reduce(list, mult, 1);
}

value_t add(value_t v1, value_t v2){
  return v1 + v1;
}

value_t mult(value_t v1, value_t v2){
  return v1 * v2;
}
```

output

```
< 4, 3, 2, 1 >

Sum = 10

Product = 24
```

```
void main() {
   List * list = create_list();
   int i = 0;

   for (i=0; i < 5; i++)
      push(list, i+1);

   print_list_all(list);

   printf("Sum = %d\n", sum(list));

   printf("Product = %d\n", prod(list));

   destroy_list(list);
}
```