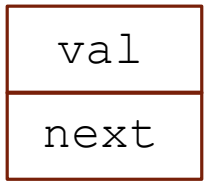
Lab 7

Demo and Helpful Tips

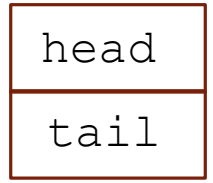
List.h -> 2 structures

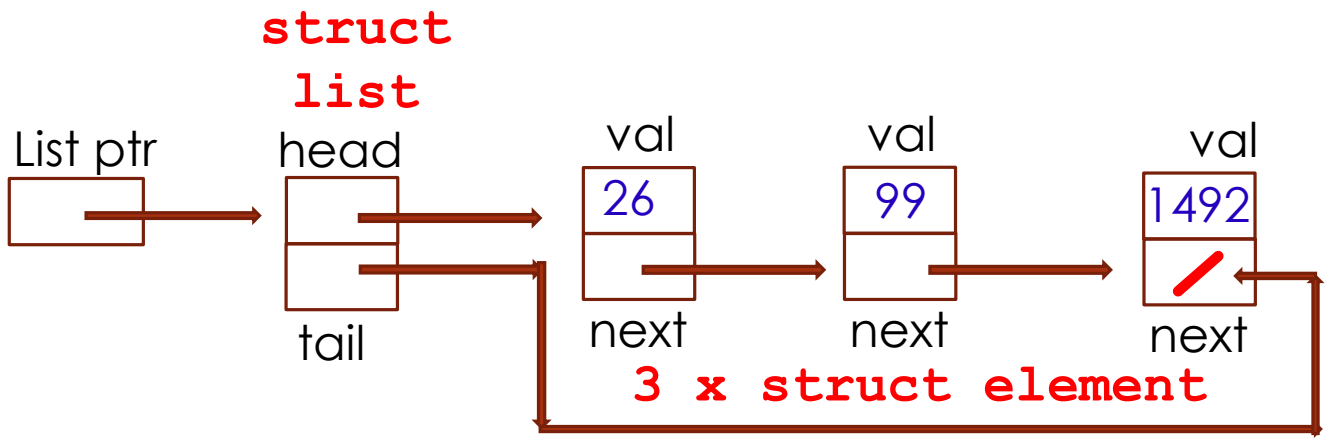
```
// List element: a list is a chain of these
typedef struct element
{
  int val;
  struct element* next;
} element_t;

// List header - keep track of the first and last list elements
typedef struct list
{
  element_t* head;
  element_t* tail;
} list_t;
```

element_t



list_t





List.h -> 7 functions

```
// returns a pointer to a new header for an empty list, or NULL if
// memory allocation fails.
list_t* list_create( void );

// frees all the memory used by the list
void list_destroy( list_t* list );

// returns a pointer to a new list element containing integer i and
// next-pointer set to NULL, or NULL if memory allocation fails.
element_t* element_create( int i );

// Appends a new element containing integer i to the end of the
// list. Returns 0 on success, else 1.
int list_append( list_t* list, int i );

// Prepends a new element containing integer i to the head of the
// list. Returns 0 on success, else 1.
int list_prepend( list_t* list, int i );

// Returns a pointer to the ith list element, where the list head is
// 0, head->next is 1, etc., or NULL if i is out of range (i.e. larger
// than (number of list elements -1 ))
element_t* list_index( list_t* list, unsigned int i );

// Prints a list in human-readable form from the first to last
// elements, between curly braces.
void list_print( list_t* list );
```

The idea of Task 1 to Task 5 is …

- We are given 5 different buggy implementations of the functions described in `List.h`
- We are given a test driver: `main.c`
- We are to discover these bugs by adding code to our test driver
- Our test driver has found a bug when it either
  - Returns 1

  OR

  - It 'crashes' with a segmentation fault
- In Task 1 to Task 5, we are not to fix these bugs

In this lab, getting a segmentation fault is a good and useful thing!

Let's have a look at `main.c`

```c
#include <stdio.h>
#include <stdlib.h>

#include "list.h"

int main( int argc, char* argv[] )
{
  // test the create function
  list_t* list = list_create();

  // check to see if the create function did everything it was supposed to
  if( list == NULL )
    {
      printf( "create: create failed to malloc\n" );
      return 1;
    }

  if( list->head != NULL )
    {
      printf( "create: head is not null!\n" );
      return 1;
    }

  if( list->tail != NULL )
    {
      printf( "create: tail is not null!\n" );
      return 1;
    }

  // now test all the other functions (except list_print) to see if
  // they do what they are supposed to

  // you code goes here

  return 0; // tests pass
}
```

We are checking to see if any of the 5 implementations of `list_create()` contains a bug

We are expecting the function `list_create()` to

- Get memory for a `list`, and to verify that it was successful at getting this memory,
- To set the `head` of the `list` to NULL
- To set the `tail` of the `list` to NULL

So, we add code to our `main.c` to check that `list_create()` did indeed do all that we expected. If not, then it's a bug and we report it by printing a useful message and returning 1.

Let's compile our `main.c`

Makefile

```
all: t1 t2 t3 t4 t5

t1: main.c t1.c
    gcc -Wall -std=c99 -o $@ main.c t1.c

t2: main.c t2.c
    gcc -Wall -std=c99 -o $@ main.c t2.c

t3: main.c t3.c
    gcc -Wall -std=c99 -o $@ main.c t3.c

t4: main.c t4.c
    gcc -Wall -std=c99 -o $@ main.c t4.c

t5: main.c t5.c
    gcc -Wall -std=c99 -o $@ main.c t5.c

clean:
    rm -f t1 t2 t3 t4 t5 *.o
```

At the command line:

```
$ make all
or
$ make
or
$ make t1
$ make t2
$ make t3
$ make t4
$ make t5
or
$ for i in {1..5}; do make t$i; done
or
$ for i in {1..5};
> do
> make t$i;
> done
```

Let's execute our `main`

- At the command line:

```
$ for i in {1..5}; do ./t$i; echo $?; done
```

or

```
$ for i in {1..5};
> do
> ./t$i; echo $?;
> done
```

- Are we getting the results we are expecting?

Result of executing `main`

```
alavergn@cs-moyie:~/sfuhome/cmpt-127/cmpt127-1194-alavergn/7$ make
gcc -Wall -std=c99 -o t1 main.c t1.c
gcc -Wall -std=c99 -o t2 main.c t2.c
gcc -Wall -std=c99 -o t3 main.c t3.c
gcc -Wall -std=c99 -o t4 main.c t4.c
gcc -Wall -std=c99 -o t5 main.c t5.c
alavergn@cs-moyie:~/sfuhome/cmpt-127/cmpt127-1194-alavergn/7$ for i in {1..5}; do ./t$i ; echo $? ; done
0
0
0
0
0
alavergn@cs-moyie:~/sfuhome/cmpt-127/cmpt127-1194-alavergn/7$
```

Hum… no bugs has been discovered
in the 5 implementations of `list_create()` !!!

How to proceed!

- Do not look at `t1.c`, `t2.c`, `t3c`, `t4.c`, `t5.c` – searching for the bugs! Nope! That would be cheating!
- Instead, work only with `main.c`, extending it, i.e., adding code to it in order to verify that all 7 functions behave as you expect
- So, what is the expected behaviour of these 7 functions?

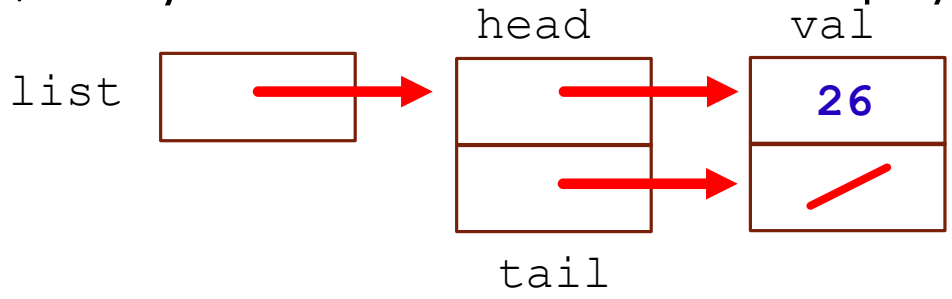
- Hint: no bugs in implementation of `list_create()`, `list_print()` and `list_destroy()`

Let's investigate `list_prepend( )`

- What are we expecting from `list_prepend( )`?
- From `List.h`:

```
// Prepends a new element containing integer i to the head of the
// list. Returns 0 on success, else 1.
int list_prepend( list_t* list, int i );
```

- For example, which result are we expecting from `list_prepend(list, 26)` if we start with an empty list?
- We are expecting:



- Let's add code to our `main.c` to confirm that the expected result depicted above is indeed what we obtain from `list_prepend(list, 26)`

Let's investigate `list_prepend( )`

- What if we were prepending to an already existing list?
- For example, which result would we be expecting from `list_prepend(list, 26)` if our list already had 3 elements: 45, 23, 19?

Modified `main.c`

```
// Testing list_prepend( )
int val = 26;
int ret = list_prepend( list, val );
// list_prepend(...) returns 0 on success, else 1.
if ( ret ) {
    puts( "list_prepend() failed." );
    return 1;
}
if( list->head == NULL ) {
    puts( "list_prepend(): list->head NULL." );
    return 1;
}
if( list->tail== NULL ) {
    puts( "list_prepend(): list->tail NULL." );
    return 1;
}
if( list->head != list->tail ) {
    puts( "list_prepend(): first prepend: head != tail." );
    return 1;
}
if( list->head->next != NULL ) {
    puts( "list_prepend(): list->head->next != NULL." );
    return 1;
}
if( list->head->val != val ){
    puts( "list_prepend(): list->head->val != val." );
    return 1;
}
```

Result of executing `main`

- Recompiling and executing the new executable, we get:

```
alavergn@cs-moyie:~/sfuhome/cmpt-127/cmpt127-1194-alavergn/7$ make
gcc -Wall -std=c99 -o t1 main.c t1.c
gcc -Wall -std=c99 -o t2 main.c t2.c
gcc -Wall -std=c99 -o t3 main.c t3.c
gcc -Wall -std=c99 -o t4 main.c t4.c
gcc -Wall -std=c99 -o t5 main.c t5.c
alavergn@cs-moyie:~/sfuhome/cmpt-127/cmpt127-1194-alavergn/7$ for i in {1..5}; do ./t$i ; echo $? ; done
0
0
0
list_prepend(): list->tail NULL.
1
0
alavergn@cs-moyie:~/sfuhome/cmpt-127/cmpt127-1194-alavergn/7$
```

Bingo!!! We discovered a bug in `t4.c`
i.e., the implementation of `list_prepend()` in t4.c contains a bug!

Helpful tip when testing `element_create()`

```
// Testing element_create( )
element_t* el = malloc( sizeof( element_t ) );
assert(el);
memset( el, 0xFF, sizeof( element_t ) );
free(el);

el = element_create( 1492 );
assert(el);

if( el->next )
{
   puts( "element_create(): el->next not NULL." );
   return 1;
}

if( el->val != 1492 )
{
   puts( "element_create(): el->val not correct." );
   return 1;
}
```

We shall assume that when we ask for memory twice in a row, we are given the same memory both times.

Based on this assumption, we ask for memory once, set this memory to the value 1 (11111111_2 -> FF_{16}) which cannot be mistaken for NULL (i.e., 0).

Then we free the memory and ask for it again via `element_create()`. If `element_create()` does not set pointers to NULL as expected, this will be detected because these pointers will have the default value of 1 i.e., not NULL.

Result of our testing

- Recompiling and executing the new executable, we get:

```
alavergn@cs-moyie:~/sfuhome/cmpt-127/cmpt127-1194-alavergn/7$ make
gcc -Wall -std=c99 -o t1 main.c t1.c
gcc -Wall -std=c99 -o t2 main.c t2.c
gcc -Wall -std=c99 -o t3 main.c t3.c
gcc -Wall -std=c99 -o t4 main.c t4.c
gcc -Wall -std=c99 -o t5 main.c t5.c
alavergn@cs-moyie:~/sfuhome/cmpt-127/cmpt127-1194-alavergn/7$ for i in {1..5}; do ./t$i ; echo $? ; done
0
element_create(): el->next not NULL.
1
0
list_prepend(): list->tail NULL.
1
0
alavergn@cs-moyie:~/sfuhome/cmpt-127/cmpt127-1194-alavergn/7$
```

Bingo!!! We discovered a bug in `t2.c`
i.e., the implementation of `list_create()` in `t2.c` contains a bug!

Continue modifying `main.c`

- … by adding code to `main.c` that verifies the expectations you have of the behaviour of each of the functions called from `main.c`
- Once you have detected the bug in each of the 5 implementations of the `List.h` functions (`t1.c`, `t2.c`, …, `t5.c`), commit your `main.c` to Gitlab and move on to Lab 7 Task 6 and Task 7

Task 6

- To compile:
  - at the command line:

```
$ gcc -o t6 list.c main.c
```

- To execute:
  - at the command line:

```
$ ./t6; echo $?
```

or

```
$ ./t6
$ echo $?
```

- Are we getting the results we are expecting?