C: Linked Lists

Department of Computer Science
College of Engineering
Boise State University

August 25, 2017

Singly Linked List (1)

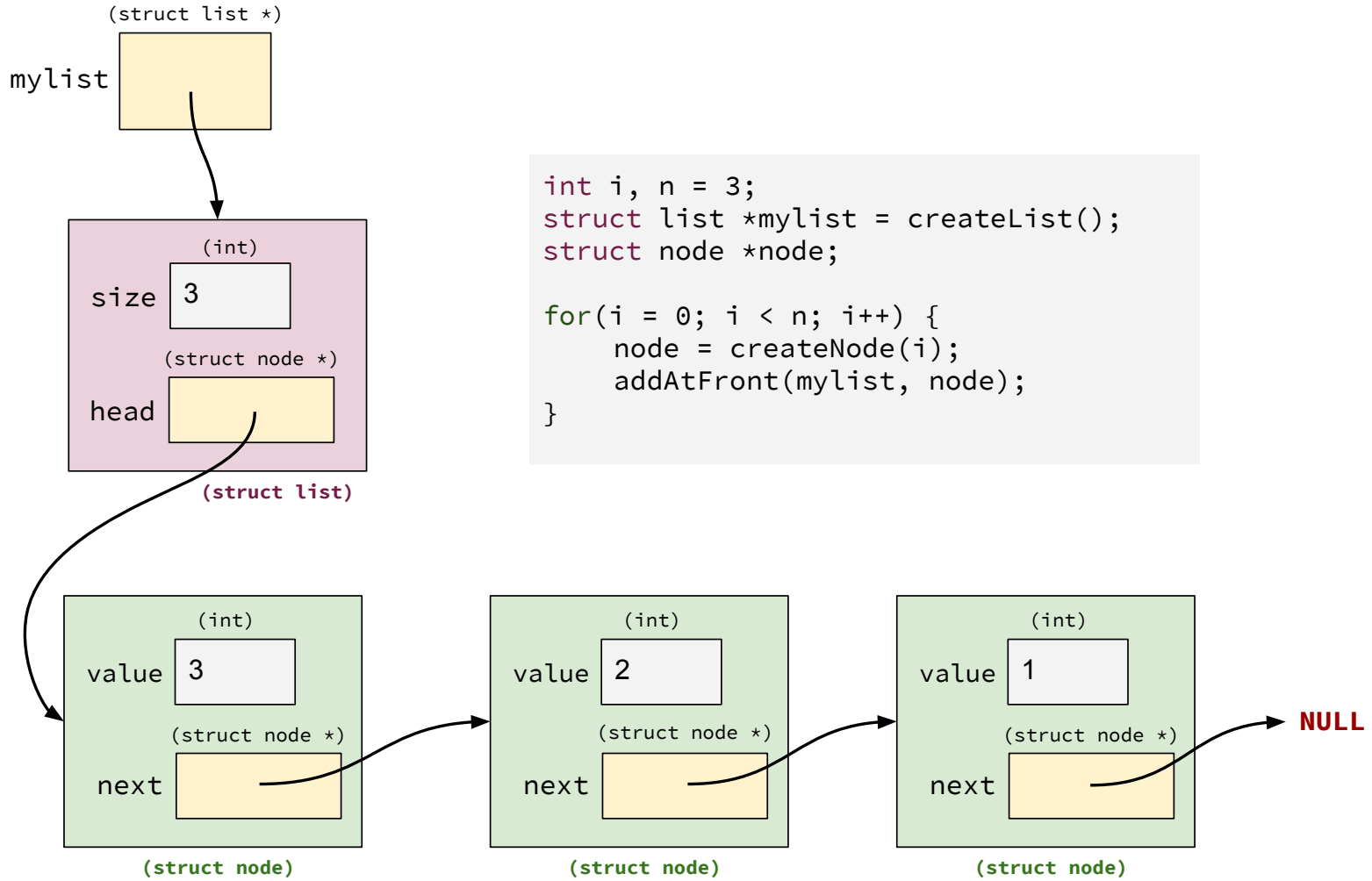
- Here is a setup for a linked list with a header represented by the following structure. The list structure keeps track of the head node of the list as well as its size.

```
struct list {
    int size;
    struct node *head;
};
```

- Each node contains pointers to the next node as well as to the data stored within the node.

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

Singly Linked List (2)




```
int i, n = 3;
struct list *mylist = createList();
struct node *node;

for(i = 0; i < n; i++) {
     node = createNode(i);
     addAtFront(mylist, node);
}
```

Singly Linked List (2)

Singly Linked List Code: C-examples/linked-lists/singly-linked

- `SinglyLinkedList.h`: Header contains node struct and function prototypes.
- `SinglyLinkedList.c`: Source contains function implementations.
- `SimpleTest.c`: Basic list test. Creates and reverses list.
- `Makefile`: Compiles source files.

Singly Linked List Code Version 2:
C-examples/linked-lists/singly-linked-better

- `libsrc/List.h, libsrc/Node.h`: Header contains node struct and function prototypes.
- `libsrc/List.c, libsrc/Node.c`: Source contains function implementations.
- `testsuite/SimpleTestList.c`: Basic list test. Creates and reverses list.
- `Makefile`: Compiles source files.

Debugging the list

- What happens if the list isn't null-terminated properly? Suppose the last node has a bad pointer? Or if some nodes get bypassed due to bugs in the list code?
- We need help! Valgrind is a memory-checker tool that catches memory errors (reading/writing to invalid memory locations due to bad pointers) and memory leaks (memory that is allocated but not freed that the program cannot reference).
- Run SimpleTest with valgrind.

```
valgrind --leak-check=yes SimpleTest 10
```

- There are memory leaks. Identify why?

Comments on Singly Linked List Code

- How would you use the code for storing data item of another type?
- How can we store different data types without modifying the code? (generic programming)
- How would use use the singly linked list code in multiple projects?
- How can we use compiled code in multiple projects without recompiling and including into each project? (libraries and plugins)
- Coming soon to a classroom near you!

Generic Coding in C

- Go to: Generic Coding in C notes.

Generic Singly Linked List (1)

- In order to create a generic linked list each node struct will contain a `void *` pointer to a generic "object" that will be stored in our list.
- We will need the user to pass us three function pointers: one for freeing the object, one foe testing if two objects are equal and one for converting the object into a string representation suitable for printing. We will store these function pointers in the list structure.

```
struct node {
    void *object;
    struct node *next;
};
struct list {
    struct node *head;
    int size;

    char *(*toString)(void *);
    void (*freeObject)(void *);
    int (*equals)(void *, void *);
};

/* constructor */
struct list *createList(char *(*toString)(void *),
                        void (*freeObject)(void *)
                        int (*equals)(void *, void *));
```

Generic Singly Linked List (2)

- Now, instead of storing a primitive type, this list will store a `void *` pointer as data in the node. This allows us to store any type in the node.
- For our example, the data stored could be a *job*. Each *job* has an *id* and some associated *info*.

```
struct job {
    int jobid;
    char *info;
};

char *toString(void *);
void freeObject(void *);
int equals(void *, void *);
```

Using the Generic List

- In order to use the generic list, the user will have to create an object type that they want to use (can be any type with any name).
- Then they have to provide the function pointers for `toString`, `equals` and `freeObject` to the `createList` function.
- **Recommended Exercise**. Finish the conversion of the singly linked list to a generic version on your own!

Doubly Linked List Example

Doubly Linked List Code: C-examples/linked-lists/doubly-linked

- `libsrc/List.h`: List header contains list struct and function prototypes.
- `libsrc/List.c`: Source contains incomplete function implementations.
- `libsrc/Node.h`: Node header contains node struct and function prototypes.
- `libsrc/Node.c`: Source contains function implementations.
- `libsrc/Makefile`: Compiles source files into library.
- `testsuite/Object.h`: Object header contains object struct and function prototypes.
- `testsuite/Object.c`: Source contains function implementations.
- `testsuite/SimpleTestList.c`: Basic list test. Creates and reverses list.
- `testsuite/RandomTestList.c`: Random list test. Creates and executes random operation on list.
- `testsuite/UnitTestList.c`: Unit tests for list. Incomplete.
- `Makefile`: Compiles test source files using the list library.

Using the Doubly-Linked List

- **In-class Exercise 1**: Create a doubly linked list named L1 and create and add two nodes to it with jobs names "job1" with id 16000 and "job2" with id 3200.
- **In-class Exercise 2**: Write a declaration for an array of `n` doubly linked lists and allocate space for it and fill it with pointers to `n` empty lists.
- **In-class Exercise 3**: How would you declare a list of queues? A list of stacks? A queue of lists? A stack of lists? A list of lists?

Testing

- Use unit-tests to increase confidence in your code.
- Check for boundary conditions!
- Use assertions to check that the program satisfies certain conditions at particular points in its execution. Assertions can be of three types:
  - Preconditions: at start of a function
  - Postconditions: at the end of a function
  - Invariants: over a block of code, for example, a loop
- In C, we can use the `assert` macro from the `assert.h` header file as shown below:

```
#include <assert.h>
assert (size <= LIMIT)
```

- If the assert fails, the program is terminated with an error message similar to shown below:

```
a.out: test.c:9: main: Assertion `size < 100' failed.
Aborted (core dumped)
```

Unit Tests for Doubly Linked List

- We will write our own version of assert that just breaks out of the current function, which represents a test case for our unit test program.

```
#define myassert(expr) if(!(expr)){ fprintf(stderr,
    "\t[assertion failed] %s: %s\n", __PRETTY_FUNCTION__
                                __STRING(expr)); return FALSE;
```

- Here is a good discussion on the use of assert statement.
- Let's review the file:

  C-examples/linked-lists/doubly-linked/testsuite/UnitTestList.c for

  a skeleton of a unit test program for the doublylinked list. You will complete this in your project!

Doubly Linked List as a Library

- For more information on how the example is compiled into a library, go to: Generic Coding in C notes.

Recommended Exercises

1. Write the header file for declaring a queue that uses dynamically allocated nodes? What typical operations would you provide. Write their prototypes.
2. Write the header file for declaring a stack that uses dynamically allocated nodes? What typical operations would you provide. Write their prototypes.
3. Compare an array-based implementation for a queue/stack with the dynamically allocated version.
4. Write the declaration for a list of queues.

Trees

- **Binary Tree**. To declare a binary tree, we can use something like the following declaration.

```
struct TreeNode {
    int key;
    void *data;
    struct TreeNode *left;
    struct TreeNode *right;
}
```

- **M-ary Tree**. Here is an example of a tree where each node can have up to `MAX_DEGREE=M` child nodes.

```
struct TreeNode {
    int key;
    void *data;
    TreeNode *child[MAX_DEGREE];
}
```

Recommended Exercise

- Develop a complete header file for a "generic" Binary Search Tree class in C. What operations would you want to provide? How would you store data in the Binary Search Tree? We will at least need methods for searching, inserting, deleting, inorder traversal among others.