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.
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. Oh my!
Freeing the list

- **Inclass Exercise.** Add a new function to free a singly linked list.

```c
void freeList(struct node *list)
{
}
```
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!
To create a more flexible list than the previous example, we will describe the 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.

```c
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.

```c
struct node {
    int item;
    struct node *next;
};
```
Generic Coding in C

Go to: Generic Coding in C notes.
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 two function pointers: one for freeing the object and one for converting the object into a string representation suitable for printing. We will store these function pointers in the list structure.

```c
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 *));
```
Now, instead of storing a primitive type, this list will store a void * pointer as data in the node. This allows us to store anything in the node.

For our example, the data stored could be a job. Each job has an id and some associated info.

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

```c
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 two function pointers for `toString` and `freeItem` to the `createList` function.
- **Recommended Exercise.** Go convert 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 list library.
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 doubly linked lists of size n and allocate space for it and fill it with pointers to n empty lists.

In Class Pondering How would you declare a list of lists? A queue of lists? A stack of lists? A list of queues?
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:
  ```c
  #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)
  ```
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.

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

Let's review the file:

`C-examples/link-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 Assignments

- Read Sections 6.1 through 6.9. Skim though Sections 6.5 and 6.6 as they are more difficult.
- Techniques introduced:
  - Using doxygen tool for javadoc style comments.
  - Using assert macros for effective unit-testing.
  - Using valgrind tool to check for memory errors and leaks.
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.

```c
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 \(\text{MAX\_DEGREE}=M\) child nodes.

```c
struct TreeNode {
    int key;
    void *data;
    TreeNode *child[\text{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.