CSE 333
Lecture 5 - data structures & modules

Hal Perkins
Department of Computer Science & Engineering
University of Washington
HW1 due in just over a week. How’s it going?

- Yes, you can use up to 2 late days on it
- No, you don’t want to

New exercise #5 due before class Monday

- Out after sections tomorrow and uses material from there
  - A bit longer than usual - you have to clean up some messy code
- No exercise #4 this quarter because of the July 4 break
Today’s topics:

- implementing data structures in C
- multi-file C programs
- brief intro to the C preprocessor
Let’s build a simple linked list

You’ve seen a linked list in CSE143

- each node in a linked list contains:
  ‣ some element as its payload
  ‣ a pointer to the next node in the linked list
- the last node in the list contains a NULL pointer (or some other indication that it is the last node)
Linked list node

Let’s represent a linked list node with a struct
- and, for now, assume each element is an int

```c
#include <stdio.h>

typedef struct node_st {
  int element;
  struct node_st *next;
} Node;

int main(int argc, char **argv) {
  Node n1, n2;

  n2.element = 2;
  n2.next = NULL;
  n1.element = 1;
  n1.next = &n2;
  return 0;
}
```

manual_list.c
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

typedef struct node_st {
  int element;
  struct node_st *next;
} Node;

Node *Push(Node *head, int e) {
  Node *n = (Node *) malloc(sizeof(Node));

  assert(n != NULL);  // crashes if false
  n->element = e;
  n->next = head;

  return n;
}

int main(int argc, char **argv) {
  Node *list = NULL;

  list = Push(list, 1);
  list = Push(list, 2);

  return 0;
}
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

typedef struct node_st {
    int element;
    struct node_st *next;
} Node;

Node *Push(Node *head, int e) {
    Node *n = (Node *) malloc(sizeof(Node));
    assert(n != NULL); // crashes if false
    n->element = e;
    n->next = head;
    return n;
}

int main(int argc, char **argv) {
    Node *list = NULL;
    list = Push(list, 1);
    list = Push(list, 2);
    return 0;
}
```c
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

typedef struct node_st {
  int element;
  struct node_st *next;
} Node;

Node *Push(Node *head, int e) {
  Node *n = (Node *) malloc(sizeof(Node));
  assert(n != NULL);    // crashes if false
  n->element = e;
  n->next = head;
  return n;
}

int main(int argc, char **argv) {
  Node *list = NULL;

  list = Push(list, 1);
  list = Push(list, 2);

  return 0;
}
```
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

typedef struct node_st {
  int element;
  struct node_st *next;
} Node;

Node *Push(Node *head, int e) {
  Node *n = (Node *) malloc(sizeof(Node));
  assert(n != NULL); // crashes if false
  n->element = e;
  n->next = head;
  return n;
}

int main(int argc, char **argv) {
  Node *list = NULL;
  list = Push(list, 1);
  list = Push(list, 2);
  return 0;
}
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

typedef struct node_st {
  int element;
  struct node_st *next;
} Node;

Node *Push(Node *head, int e) {
  Node *n = (Node *) malloc(sizeof(Node));
  assert(n != NULL); // crashes if false
  n->element = e;
  n->next = head;
  return n;
}

int main(int argc, char **argv) {
  Node *list = NULL;

  list = Push(list, 1);
  list = Push(list, 2);

  return 0;
}
```c
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

typedef struct node_st {
  int element;
  struct node_st *next;
} Node;

Node *Push(Node *head, int e) {
  Node *n = (Node *) malloc(sizeof(Node));
  assert(n != NULL); // crashes if false
  n->element = e;
  n->next = head;
  return n;
}

int main(int argc, char **argv) {
  Node *list = NULL;
  list = Push(list, 1);
  list = Push(list, 2);
  return 0;
}
```
```c
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

typedef struct node_st {
  int element;
  struct node_st *next;
} Node;

Node *Push(Node *head, int e) {
  Node *n = (Node *) malloc(sizeof(Node));
  assert(n != NULL);  // crashes if false
  n->element = e;
  n->next = head;
  return n;
}

int main(int argc, char **argv) {
  Node *list = NULL;
  list = Push(list, 1);
  list = Push(list, 2);
  return 0;
}
```
Push onto list

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

typedef struct node_st {
  int element;
  struct node_st *next;
} Node;

Node *Push(Node *head, int e) {
  Node *n = (Node *) malloc(sizeof(Node));
  assert(n != NULL); // crashes if false
  n->element = e;
  n->next = head;
  return n;
}

int main(int argc, char **argv) {
  Node *list = NULL;
  list = Push(list, 1);
  list = Push(list, 2);
  return 0;
}
```
```
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

typedef struct node_st {
  int element;
  struct node_st *next;
} Node;

Node *Push(Node *head, int e) {
  Node *n = (Node *) malloc(sizeof(Node));
  assert(n != NULL); // crashes if false
  n->element = e;
  n->next = head;
  return n;
}

int main(int argc, char **argv) {
  Node *list = NULL;
  list = Push(list, 1);
  list = Push(list, 2);
  return 0;
}
```
Push onto list

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

typedef struct node_st {
  int element;
  struct node_st *next;
} Node;

Node *Push(Node *head, int e) {
  Node *n = (Node *) malloc(sizeof(Node));
  assert(n != NULL); // crashes if false
  n->element = e;
  n->next = head;
  return n;
}

int main(int argc, char **argv) {
  Node *list = NULL;
  list = Push(list, 1);
  list = Push(list, 2);
  return 0;
}
```
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

typedef struct node_st {
    int element;
    struct node_st *next;
} Node;

Node *Push(Node *head, int e) {
    Node *n = (Node *) malloc(sizeof(Node));
    assert(n != NULL); // crashes if false
    n->element = e;
    n->next = head;
    return n;
}

int main(int argc, char **argv) {
    Node *list = NULL;
    list = Push(list, 1);
    list = Push(list, 2);
    return 0;
}
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

typedef struct node_st {
  int element;
  struct node_st *next;
} Node;

Node *Push(Node *head, int e) {
  Node *n = (Node *) malloc(sizeof(Node));
  assert(n != NULL); // crashes if false
  n->element = e;
  n->next = head;
  return n;
}

int main(int argc, char **argv) {
  Node *list = NULL;
  list = Push(list, 1);
  list = Push(list, 2);
  return 0;
}
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

typedef struct node_st {
  int element;
  struct node_st *next;
} Node;

Node *Push(Node *head, int e) {
  Node *n = (Node *) malloc(sizeof(Node));
  assert(n != NULL); // crashes if false
  n->element = e;
  n->next = head;
  return n;
}

int main(int argc, char **argv) {
  Node *list = NULL;
  list = Push(list, 1);
  list = Push(list, 2);
  return 0;
}
```
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

typedef struct node_st {
  int element;
  struct node_st *next;
} Node;

Node *Push(Node *head, int e) {
  Node *n = (Node *) malloc(sizeof(Node));
  assert(n != NULL); // crashes if false
  n->element = e;
  n->next = head;
  return n;
}

int main(int argc, char **argv) {
  Node *list = NULL;
  list = Push(list, 1);
  list = Push(list, 2);
  return 0;
}
```
Push onto list

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

typedef struct node_st {
  int element;
  struct node_st *next;
} Node;

Node *Push(Node *head, int e) {
  Node *n = (Node *) malloc(sizeof(Node));
  assert(n != NULL); // crashes if false
  n->element = e;
  n->next = head;
  return n;
}

int main(int argc, char **argv) {
  Node *list = NULL;
  list = Push(list, 1);
  list = Push(list, 2);
  return 0;
}
```
Push onto list

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

typedef struct node_st {
  int element;
  struct node_st *next;
} Node;

Node *Push(Node *head, int e) {
  Node *n = (Node *) malloc(sizeof(Node));

  assert(n != NULL); // crashes if false
  n->element = e;
  n->next = head;

  return n;
}

int main(int argc, char **argv) {
  Node *list = NULL;

  list = Push(list, 1);
  list = Push(list, 2);

  return 0;
}
```
Push onto list

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

typedef struct node_st {
    int element;
    struct node_st *next;
} Node;

Node *Push(Node *head, int e) {
    Node *n = (Node *) malloc(sizeof(Node));
    assert(n != NULL); // crashes if false
    n->element = e;
    n->next = head;
    return n;
}

int main(int argc, char **argv) {
    Node *list = NULL;
    list = Push(list, 1);
    list = Push(list, 2);
    return 0;
}
```

Why is this leak not a serious problem?
A generic linked list

Previously, our linked list elements were of type `int`

- what if we want to let our customer decide the element type?
- idea: let them push a generic pointer -- i.e., a `(void *)`

```c
typedef struct node_st {
  void *element;
  struct node_st *next;
} Node;

Node *Push(Node *head, void *e) {
  Node *n = (Node *) malloc(sizeof(Node));
  assert(n != NULL);  // crashes if false
  n->element = e;
  n->next = head;
  return n;
}
```
Using a generic linked list

To use it, customers will need to use type casting

- convert their data type to a (void *) before pushing
- convert from a (void *) back to their data type when accessing

```c
typedef struct node_st {
  void *element;
  struct node_st *next;
} Node;

Node *Push(Node *head, void *e);  // assume last slide’s code

int main(int argc, char **argv) {
  char *hello = "Hi there!";
  char *goodbye = "Bye bye."
  Node *list = NULL;

  list = Push(list, (void *) hello);
  list = Push(list, (void *) goodbye);
  printf("payload: '%s'\n", (char *) ((list->next)->element) );
  return 0;
}
```
Using a generic linked list

Results in:

(main) list

element

next

NULL

(main) hello

Hello!

Bye bye.

(main) goodbye

(main) goodbye

Hello!

Bye bye.
Multi-file C programs

Let’s create a linked list module

- a module is a self-contained piece of an overall program
  ‣ has externally visible functions that customers can invoke
  ‣ has externally visible typedefs, and perhaps global variables, that customers can use
  ‣ may have internal functions, typedefs, global variables that customers should not look at

- the module’s interface is its set of public functions, typedefs, and global variables
Modularity

The degree to which components of a system can be separated and recombined

- “loose coupling” and “separation of concerns”
- modules can be developed independently
- modules can be re-used in different projects
C header files

header: a C file whose only purpose is to be `#include’d`

- generally a filename with the .h extension
- holds the variables, types, and function prototype declarations that make up the interface to a module

the main idea

- every `name.c` intended to be a module has a `name.h`
- `name.h` declares the interface to that module
- other modules that want to use `name` will `#include name.h`
  - and they should assume as little as possible about the implementation in `name.c`
C module conventions

Most C projects adhere to the following rules:

- .h files never contain definitions, only declarations

- .c files never contain prototype declarations for functions that are intended to be exported through the module interface
  
    ▷ those function prototype declarations belong in the .h file

- *never* #include a .c file -- only #include .h files

- #include all of the headers you reference, even if another header (accidentally) includes some of them

- any .c file with an associated .h file should be able to be compiled into a .o file
#include and the C preprocessor

The C preprocessor (cpp) transforms your source code before the compiler runs:

- transforms your original C source code into transformed C source code

- processes the directives it finds in your code (#something)
  
  ‣ #include "ll.h"  -- replaces with post-processed content of ll.h
  
  ‣ #define PI 3.1415  -- defines a symbol, replaces later occurrences
  
  ‣ and there are several others we’ll see soon...

- run on your behalf by gcc during compilation
Example

Let’s manually run the pre-processor on cpp_example.c:

- cpp is the preprocessor
- "-P" suppresses some extra debugging annotations
- (can also use gcc -E)

bash$ cpp -P cpp_example.c out.c
bash$ cat out.c
typedef long long int verylong;
int main(int argc, char **argv) {
  int x = 1;
  int y = 2 + 1;
  verylong z = 1 + 2 + 1;
  return 0;
}
Program that uses a linked list

```c
#include <stdlib.h>
#include <assert.h>
#include "ll.h"

Node *Push(Node *head, void *element) {
   ...
}
```

```c
typedef struct node_st {
   void *element;
   struct node_st *next;
} Node;

Node *Push(Node *head, void *element);
```

```c
#include "ll.h"

int main(int argc, char **argv) {
   Node *list = NULL;
   char *hi = "hello";
   char *bye = "goodbye";

   list = Push(list, hi);
   list = Push(list, bye);

   return 0;
}
```

```c
example_ll_customer.c
```
Compiling the program

Four steps:
- compile `example_ll_customer.c` into an object file
- compile `ll.c` into an object file
- link `ll.o`, `example_ll_customer.o` into an executable
- test, debug, rinse, repeat

```bash
bash$ gcc -Wall -g -o example_ll_customer.o -c example_ll_customer.c
bash$ gcc -Wall -g -o ll.o -c ll.c
bash$ gcc -o example_ll_customer -g ll.o example_ll_customer.o
bash$
bash$ ./example_ll_customer
```

Payload: 'yo!'
Payload: 'goodbye'
Payload: 'hello'

```bash
bash$ valgrind --leak-check=full ./example_customer
...etc.
```
Where do the comments go?

If a function is declared in a header file (.h) and defined in a C file (.c)

- The header needs full documentation. It is the public specification.
- No need to cut/paste the comment in the C file
  ‣ Don’t want two copies that can get out of sync
  ‣ But help the reader with a “specified in foo.h” comment on the code

If a function has a prototype and implementation in the same C file

- One school: full comment on the prototype at the top of the file, no comment (or “declared above”) on code (e.g., project code is like this)
- Another: prototype is for the compiler, doesn’t need a comment; put the comments with the code to keep them together (my preference)
Exercise 1

Extend the linked list program we covered in class:

- add a function that returns the number of elements in a list
- implement a program that builds a list of lists
  - i.e., it builds a linked list
    - but each element in the list is a (different) linked list
- bonus: design and implement a “Pop” function
  - removes an element from the head of the list
  - make sure your linked list code, and customers’ code that uses it, contains no memory leaks
Exercise 2

Implement and test a binary search tree

  
  ‣ don’t worry about making it balanced

- implement key insert() and lookup() functions
  
  ‣ bonus: implement a key delete() function

- implement it as a C module
  
  ‣ bst.c, bst.h

- implement test_bst.c
  
  ‣ contains main(), tests out your BST
Exercise 3

Implement a Complex number module

- complex.c, complex.h
- includes a typedef to define a complex number
  ‣ a + bi, where a and b are doubles
- includes functions to:
  ‣ add, subtract, multiply, and divide complex numbers
- implement a test driver in test_complex.c
  ‣ contains main()
See you on Friday Monday!