CS 11 C track: lecture 7

- Last week: structs, `typedef`, linked lists
- This week:
  - hash tables
  - more on the C preprocessor
  - `extern`
  - `const`
Hash tables (1)

- Data structures we've seen so far:
  - arrays
  - structs
  - linked lists
Hash tables (2)

- Hash tables are a new data structure
- Like an array indexed with strings *e.g.*
  - `height["Jim"] = 6; /* not C code */`
- Very fast lookup (O(1) *i.e.* constant time)
- Flexible: can add/delete elements easily
Hash tables (3)

- Want to associate a string (key) with a value
- Generate an integer hash value from the string key
  - different keys should generate different hash values
- Use hash value as index into an array of linked lists
  - array length is large (128 in lab 7)
  - array values start off as NULL pointers (empty lists)
  - no linked list should ever get larger than a few elements
Hash tables (4)

119 (hash value)

“Jim”

height

key

value

next

6

(NULL)
A hash table is an array of linked lists.
The linked lists all start off as empty lists.
Empty lists are represented as the NULL pointer.
So: the array of linked lists is actually an array of pointers to linked lists (pointers to nodes in a linked list).
If you use an array of nodes, your program is broken!
Generating the hash value from the string

- Many ways to do it
- We choose a particularly simple (and lame) way
- Treat the string as an array of `chars`
- Treat each `char` as a small integer (0 - 127)
  - C allows this
- Sum up the values of all the characters
- Take the sum mod 128 (the array length)
- Gives an integer in the range 0-127
  - that's our index into the array
Hash tables (6)

- Three things we can do with a hash table:
  - Look up the value corresponding to a particular key
  - Change the value corresponding to an existing key in the table
  - Add a new key/value pair to the table
Hash tables (7)

- How to find the value given the key
  - compute hash value to get array index
  - find array location
  - if NULL, not there (return "not found" value)
  - if not NULL, search for key in linked list
    - if found, return node value
    - if not found, not there (return "not found" value)
How to change the value corresponding to a given key (or add a new key/value pair):

- compute hash value to get array index
- find array location
- if NULL, add node with key/value pair
- if not NULL, search for key in linked list
  - if found, change node value
  - if not found, add new node to list
    - (anywhere in list!)
Hash tables (9)

- Adding nodes to linked list
  - nodes in linked list not in any order
  - so can add to any place in list
  - most people try to add to the end of the list
  - actually easier to add to \textit{beginning} of list
  - either way, have to set some pointer values to different values
Hash tables (10)

- Hash table itself is *not* the array of linked lists
  - It's a `struct` which contains that array
  - Easy to make mistakes with this
  - Think of it as a box containing the array
- Why use a `struct` if all it contains is one array?
  - Practice in handling more complex data structures
  - Real hash tables would have more fields e.g. length of array to permit resizing of the array
Lab 7

- Pretty routine application of hash tables
- One likely problem involving a memory leak
  - May be hard to figure out where to free memory
Sometimes want to conditionally compile code

If some condition met, compile this code

else do nothing, or do something else

Examples:

- debugging code
- compiling on different platforms
C preprocessor: #ifndef (2)

- Debugging code:

```c
#define DEBUG

int value = 10;

#ifndef DEBUG

    printf("value = %d\n", value);

#endif
```
C preprocessor: `#ifdef`

- Can leave out `#define` and choose at compile time:
  
  ```
  % gcc -DDEBUG foo.c -o foo
  ```

- `-D` option means to Define `DEBUG`

- This makes the debugging code compile

- Otherwise it won’t compile

- Usually best to do it this way
C preprocessor: `#else`

- Also use `#ifdef/#else` for portability e.g.:

```c
#ifdef WINDOWS
#include <windows.h>
#else
#include <X11/X.h>
#endif
```
C preprocessor: \texttt{#ifndef (1)}

- \texttt{#ifndef} includes code if something is not defined
- \texttt{assert} is defined using \texttt{#ifndef} \textit{e.g.}
  
  ```c
  assert(i == 0);  /* expands to: */
  ifndef NDEBUG
      if (!(i == 0)) { abort(); } 
  endif
  ```
C preprocessor: `#ifndef`

- Recall: to switch off assertions, define `NDEBUG`:

```
% gcc -DNDEBUG foo.c -o foo
```

- Then all assertions are removed from code during compilation
- Useful after code has been debugged
C preprocessor: `#if` (1)

- Can also test integer values with `#if/#elif/...`:

```c
#if REVISION == 1
/* revision 1 code */
#endif
```

```c
#elif REVISION == 2
/* revision 2 code */
#endif
```

```c
#else
/* generic code */
#endif
```
C preprocessor: `#if (2)`

- Use `#if 0` to comment out large blocks of code:

```
#if 0
/* This doesn’t get compiled. */
#endif
```

- Useful because can't nest `/* */` comments
Multiple inclusion of header files can cause problems

- *e.g.* multiple declarations of struct types

- Difficult to prevent
  - one include file includes another, etc.

- Need mechanism to prevent this
C preprocessor: include guards (2)

/* header file "foo.h": */
 ifndef FOO_H
 define FOO_H

 /* contents of file */

 endif  /* FOO_H */

 contents of foo.h only included once
extern (1)

- Sometimes many files need to share some data *e.g.* global variable
- Can only define in one place
- Put `extern` declaration in header file
- Means: this is defined somewhere else
extern (2)

/* In header file "foo.h": */

extern int max_value;

/* In file "foo.c": */

/* global variable: */

int max_value = 1000000;
We’ve seen this:

```c
#define SOME_CONSTANT 100
```

A better alternative is this:

```c
const int SOME_CONSTANT = 100;
```

Why is this better?

- get type checking on `SOME_CONSTANT`
Next week

- Most of C language has been covered
- Virtual machines (!)
- More integer types: *short, long, unsigned*
- Wrapping up