CS 11 C track: lecture 6

- Last week: pointer arithmetic

- This week:
  - The **gdb** program
  - **struct**
  - **typedef**
  - linked lists
gdb for debugging (1)

- **gdb**: the **Gnu DeBugger**
- [http://courses.cms.caltech.edu/cs11/material/c/mike/misc/gdb.html](http://courses.cms.caltech.edu/cs11/material/c/mike/misc/gdb.html)
- Use when program core dumps
- or when want to walk through execution of program line-by-line
Before using **gdb**:

- Must compile C code with additional flag: `-g`
- This puts all the source code into the binary executable

Then can execute as: **gdb myprogram**

Brings up an interpreted environment
gdb for debugging (3)

\texttt{gdb} > \texttt{run}

- Program runs...
- If all is well, program exits successfully, returning you to prompt
- If there is (e.g.) a core dump, \texttt{gdb} will tell you and abort the program
If your program needs command-line arguments, e.g. `myprogram 1 2 3`, then you should do this in `gdb`:

```
gdb> run 1 2 3
```

This will run `myprogram` with the command-line arguments 1, 2, and 3.
Stack backtrace ("where")

- Your program core dumps
- Where was the last line in the program that was executed before the core dump?
- That's what the **where** command tells you
gdb – basic commands (2)

```
gdb> where

#0 0x4006cb26 in free () from /lib/libc.so.6
#1 0x4006ca0d in free () from /lib/libc.so.6
#2 0x8048951 in board_updater (array=0x8049bd0, ncells=2) at 1dCA2.c:148
#3 0x80486be in main (argc=3, argv=0xbffff7b4) at 1dCA2.c:44
#4 0x40035a52 in __libc_start_main () from /lib/libc.so.6

stack backtrace
```
**gdb – basic commands (3)**

- Look for topmost location in stack backtrace that corresponds to your code

- Watch out for
  - freeing memory you didn't allocate
  - accessing arrays beyond their maximum elements
  - dereferencing pointers that don't point to part of a malloc()ed block
**gdb – basic commands (4)**

- **break**, **continue**, **next**, **step** commands
- **break** causes execution to stop on a given line
  
  `gdb> break foo.c: 100` (setting a breakpoint)
- **continue** resumes execution from that point
- **next** executes the next line, then stops
- **step** executes the next statement
  - goes into functions if necessary (**next** doesn't)
print and display commands

print prints the value of any program expression

```
  gdb> print i
  $1 = 100
```

display prints a particular value every time execution stops

```
  gdb> display i
```
print will print arrays as well

```c
int arr[] = { 1, 2, 3 };
```

gdb> print arr

$1 = {1, 2, 3}

N.B. the $1 is just a name for the result

print $1

$2 = {1, 2, 3}
**gdb – printing arrays (2)**

- `print` has problems with dynamically-allocated arrays

```c
int *arr;

arr = (int *)malloc(3 * sizeof(int));

arr[0] = 1; arr[1] = 2; arr[2] = 3;
```

```
gdb> print arr
$1 = (int *) 0x8094610
```

- Not very useful...
Can print this array by using `@` (gdb special syntax)

```c
int *arr;

arr = (int *)malloc(3 * sizeof(int));

arr[0] = 1; arr[1] = 2; arr[2] = 3;
```

gdb> print *arr@3

$2 = {1, 2, 3}
Common `gdb` commands have abbreviations:

- **p** (same as `print`)
- **c** (same as `continue`)
- **n** (same as `next`)
- **s** (same as `step`)

More convenient to use when interactively debugging.
structs (1)

- Way to package primitive data objects into an aggregate data object
  
- **struct** declaration:

```c
struct point {
  int x;
  int y;
  double dist; /* from origin */
}; /* MUST have semicolon! */
```
struct declaration usually done outside of function, like a function prototype

Create/initialize struct like this:

```c
struct point p;
p.x = 0;  /* "dot syntax" */
p.y = 0;
p.dist = sqrt(p.x*p.x + p.y*p.y);
```
Using a `struct`:

```c
void foo(void) {
    struct point p;
    p.x = 10; p.y = -3;
    p.dist = sqrt(p.x*p.x + p.y*p.y);
    /* do stuff with p */
}
```
Using `malloc()` with structs:

```c
struct point *make_point(void) {
    struct point *p;
    p = (struct point *) malloc(sizeof(struct point));
    return p;
} /* free struct elsewhere */
```
Using pointers to structs:

```c
void init_point(struct point *p) {
    (*p).x = (*p).y = 0;
    (*p).dist = 0.0;
    /* syntactic sugar: */
    p->x = p->y = 0;
    p->dist = 0.0;
}
```
structs (6)

- structs can contain arrays or other structs
- Usually use pointers to structs instead of just plain structs

```c
struct foo {
    int x;
    struct point p1; /* Unusual */
    struct point *p2; /* Typical */
};
```
structs (7)

- structs can be "recursive":

```c
struct node {
    int value;
    struct node *next;
};
```

- but can't have `struct node next` inside declaration (why?)
Typing `struct point` all the time is tedious.

Use a `typedef` (type alias):

```c
typedef struct point Point;
```

```c
typedef int Length;
```

Original type comes first.

New name is at the end.
**typedef (2)**

- Type component of `typedef` can also be a struct

```c
typedef struct { /* no name for struct */
    int x;
    int y;
    double dist;
} Point;
```

`Point p1, p2; /* no "struct" */`

- N.B. This is an *anonymous* struct
typedef (3)

- Recursively defined structs:

```c
typedef struct _node {
    int value;
    struct _node *next;
} node;
```
typedef (4)

- Read this as:

```c
typedef

struct _node {
    int value;
    struct _node *next;
}

node;
```
Linked lists

- `node` is the linked list struct!
- Set `next` pointer to next node in list
- If `next` is `NULL`, then at end of list
- Linked lists are just chains of `nodes`
Creating a linked list (1)

```c
node *list, *n, *prev;
```
Linked list (diagram)

list

n

prev
n = (node *)malloc(sizeof(node));
list = n;  /* list points to first node */
n->value = 10;
prev = n;  /* pointer to previous node */
Linked list (diagram)

list

n

prev
Linked list (diagram)

\[ \text{list} \rightarrow \text{n} \rightarrow \text{prev} \rightarrow (\text{node}) \]
Linked list (diagram)
Linked list (diagram)

- `list` points to `n` which points to a node with value 10 and `next:`.
- `prev` points to `n`.

`value: 10`
`next:`

(node)
Linked list (diagram)

- list
- n
- prev

(node) value: 10
next:
Creating a linked list (3)

n = (node *)malloc(sizeof(node));
prev->next = n; /* connect nodes */
prev = n;
n->value = 20;
/* ... continued on next slide ... */
Linked list (diagram)

- `list` arrow pointing to a node
- `n` arrow pointing to a node
- `prev` arrow pointing to a node

Node:
- `value: 10`
- `next:`

(node)
Linked list (diagram)

- **list**
- **n**
- **prev**

<table>
<thead>
<tr>
<th>value: 10</th>
</tr>
</thead>
<tbody>
<tr>
<td>next:</td>
</tr>
</tbody>
</table>

(node)

(node)
Linked list (diagram)

list

n

prev

value: 10
next:

(node)

(node)
Linked list (diagram)

- `list`
- `n`
- `prev`

Value: 10
Next: (node)

(prev (node))
Linked list (diagram)

- `list`
- `n`
- `prev`

[value: 10]
[next:]
Linked list (diagram)
Creating a linked list (4)

/* Continued... */
n = (node *) malloc(sizeof(node));
prev->next = n;
prev = n;
prev = n;
n->value = 30;
n->next = NULL; /* End of list marker. */
Linked list (diagram)

```
list

n
prev

value: 10
next:

value: 20
next:

(node)

(node)
```
Linked list (diagram)

```
value: 10
next:

value: 20
next:

prev

n

list

(node)
(node)

(node)
```
Linked list (diagram)
Linked list (diagram)

- list
- n
- prev

- value: 10
  next: (node)

- value: 20
  next: (node)

- (node)
Linked list (diagram)
Linked list (diagram)
Linked list (diagram)

- list
- prev
- n

Value: 10
Next: (node)

Value: 20
Next: (node)

Value: 30
Next: NULL
Linked list (final diagram)

```
list

value: 10
next:

value: 20
next:

value: 30
next: NULL
```
Creating a linked list (5)

- Can also create linked lists from the end back to the front
- Actually easier to do it that way when possible
  - example: lab 6 command-line arguments
- End-of-list is represented as NULL pointer
- add nodes to previous list (or to NULL)
Creating a linked list (6)

list = NULL;    /* Empty list. */
node *n = (node *) malloc(sizeof(node));
n->value = 30;
n->next = list;
list = n;     /* now 1-node list */
Linked list (diagram)

list → NULL
Linked list (diagram)
Linked list (diagram)

```
list -> NULL

n -> value: 30 (node)
```
Linked list (diagram)

- `n` is a node with `value: 30` and `next: NULL`.
- `list` points to `NULL`.

The diagram shows the structure of a single node in a linked list with a value of 30 and a null next pointer.
Linked list (diagram)

(n) value: 30
next: NULL
(list)
Linked list (diagram)

list

value: 30
next: NULL

(none)
Creating a linked list (7)

node *n = (node *) malloc(sizeof(node));
n->value = 20;
n->next = list;
list = n;     /* now 2-node list */
Linked list (diagram)

`list` → `n` → `value: 30` → `next: NULL` → `(node)`
Linked list (diagram)

```
list ->
   value: 30
   next: NULL

n -> (node)
```

(node)
Linked list (diagram)

```
list
  n

(node)
  value: 20

(node)
  value: 30
  next: NULL
```
Linked list (diagram)

```
value: 20
next:

value: 30
next: NULL
```

n → (node)

list → (node)
Linked list (diagram)

list

n

value: 20
next:

value: 30
next: NULL

(node)

(node)
Creating a linked list (8)

```c
node *n = (node *) malloc(sizeof(node));
n->value = 10;
n->next = list;
list = n; /* now 3-node list */
```
Linked list (diagram)

list

n

value: 20
next:

value: 30
next: NULL

(node)

(node)
Linked list (diagram)

```
list
  value: 20
  next: 

  value: 30
  next: NULL
```

(none)
Linked list (diagram)

```
  n
  value: 10

  list
  value: 20
  next: 

  value: 30
  next: NULL
```

(node)
Linked list (diagram)

```
list
  → n
  ↓
  value: 10
  next:

value: 20
next:

value: 30
next: NULL
```
Linked list (diagram)

- value: 10
  - next:
    - value: 20
      - next:
        - value: 30
          - next: NULL
Linked list (final diagram)

```
list

<table>
<thead>
<tr>
<th>value: 10</th>
</tr>
</thead>
<tbody>
<tr>
<td>next:</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>value: 20</th>
</tr>
</thead>
<tbody>
<tr>
<td>next:</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>value: 30</th>
</tr>
</thead>
<tbody>
<tr>
<td>next: NULL</td>
</tr>
</tbody>
</table>

(node)

(node)

(node)
```
Checking `malloc()`

- Previous code simplified to fit on slide
- Actually should check every `malloc` call for failure

```c
n = (node *)malloc(sizeof(node));
if (n == NULL)
{
    fprintf(stderr,
        "Error: out of memory. \n");
    exit(1);
}
```
Iterating through a linked list

- Standard idiom for going through linked lists:

```c
node *n;

/* Set all node values to zero. */
for (n = list; n != NULL; n = n->next) {
    n->value = 0;
}
```

- You should be able to figure out how this works
This week's lab:

- New sorting algorithm: "quicksort"
- More efficient than ME sort, bubblesort
- Use on linked lists, not arrays
- Memory management will be a challenge!
Next time

- Hash tables
- More "fun" with pointers ;-)