1 Administrative Information
2 Function Pointers
3 Data structures in C
4 Poor man’s inheritance
5 Typedefs
6 Building an OS Queue
7 The Road Ahead
Getting ready for Project 1

Administrative Information

- Change in office hours
  - Monday 2.30pm - 4.25pm (Z)
  - Wednesday 1.25pm - 3.20pm (Ki Suh)
- For next week only
  - Z’s office hours will be moved to Tuesday, 4.10pm - 5.00pm
  - Venue: 4132 Upson Hall
- Project 1 will be released next week.
Project groups are due by Tuesday, 6 September.
E-mail Z if you have not signed up on paper yet.
Update: individual project weightages will not be equal.
Function Pointers

- Instead of referencing data, references code.
- Don’t need to dereference the function pointer to use it.
- When assigning a value to the function pointer, use the function name directly, don’t prepend the & sign to it.
Function Pointer example

```c
int inc(int i) { return i + 1; }
int dec(int i) { return i - 1; }

int apply(int (*f)(int), int i)
{
    return f(i);
}

int main(int argc, char** argv)
{
    printf("++:%i\n", apply(inc, 10));
    printf("--:%i\n", apply(dec, 10));
    return 0;
}
```
Dissecting the Function Pointer

```
int (*f)(int)
```

- return type of the function is int
- function takes one parameter of type int
- function pointer will be henceforth referred to as ‘f’

Any function that takes an int and returns an int (ie. of the form `int foo(int param)`) can be assigned to `f`. 
Structs

- An aggregated set of related variables.
- Each variable can be a different data type.
- Provides a way to attain OO-like behavior in C.
Internally...

- structs have predictable layout.
- Each struct is contiguous in memory.
- Variables are packed together in the order they are specified in the definition.
Example

```c
struct coordinates
{
    int x;
    int y;
};
```

- `sizeof(struct coordinates)` is 8
Another example

```c
struct telephone_entry
{
    char name[15];
    short area_code;
    int local_number;
};
```

- `sizeof(struct telephone_entry)` is not `15 + 2 + 4`
Variables are aligned to certain power-of-2 numbered boundaries for faster access.

Padding can be controlled through compiler options.

Pad bytes may be non-zero.

Not important to know the rules of padding, just need to know that padding can occur in structs.
Padding Illustration

- **name (15 bytes)**
- **local_number (4 bytes)**
- **padding (unknown values)**
- **area_code (2 bytes)**
structs are contiguous in memory and have predictable layout.

dissimilar structs with similar initial fields will have similar initial memory layout.

exploit this fact to create "base classes".
Example

```c
struct generic_tree_node {
    struct generic_tree_node* parent;
    struct generic_tree_node* left_child;
    struct generic_tree_node* right_child;
};

struct my_tree_node {
    struct my_tree_node* parent;
    struct my_tree_node* left_child;
    struct my_tree_node* right_child;
    int node_value;
    char* data;
};
```
void swap_children(struct generic_tree_node* node)
{
    struct generic_tree_node* temp;

    temp = node->left_child;
    node->left_child = node->right_child;
    node->right_child = temp;
}

- **Cast** `struct my_tree_node*` into `struct generic_tree_node*` and call the function.
typedef struct list_elem
{
    int data;
    struct list_elem* next;
} list_elem_t;
typedef int *int_ptr;

typedef void *any_t;

typedef struct {
    int x;
    int y;
} coordinates, *coordinates_ptr;
typedef int *int_ptr;

int main(int argc, char** argv) {
    int x = 5;
    int_ptr = &x;
    printf("value of x is %d\n", *int_ptr);
    return 0;
}
Building an OS Queue

- Build some queue functions that can be reliably used by your OS.
- Your scheduler will depend heavily on these functions, so performance is important.
  - Enqueue and dequeue should run in $O(1)$ time.
Typical implementation

- Singly or doubly linked list can both satisfy $O(1)$. 

![Diagram of linked list with head and tail pointers]
Enqueue operation

- To enqueue, allocate a piece of memory, set the pointers.
- Two different cases to consider:
  - normal case: append list node to the end and update tail pointer.
  - boundary case: queue is empty; set head and tail to point to the same node.
Normal case
Boundary case

tail →

head →

tail →

head →
Dequeue operation

- To dequeue, remove the first list node and free up its associated memory.
- Two different cases to consider:
  - normal case: update head pointer and new head node.
  - boundary case 1: queue contains only 1 node; update head and tail pointers.
  - boundary case 2: queue is empty, return an error.
Normal case
Boundary case 1
Alternative queue implementation

- Motivation: it is not always possible to safely call allocate in a real OS.
  - An interrupt may arrive in the middle of a malloc() call.
  - If the interrupt does queue operations and calls malloc(), a deadlock could occur.

- Have to think of some other way to allow queue operations without allocating memory.
Solution

- Make sure queueable objects are augmented with some extra space.
  - Use "poor man’s inheritance" to set aside space for the pointers your queue structure needs.
  - Perform queue operations by casting queue objects into the "base class".

- Objects not augmented with this extra space cannot be queued.

- Challenge seekers: implement your queue using this method!
Project 1 will be released next Friday, 9 September.
Topics: scheduling, threads, semaphores (and of course queues).
Come to class to hear more about it.