Lecture 11

Writing modular code in C

- Java's design is centered around modularity and abstraction
  - Objects (subtyping) and parameterization (e.g., List<T>) are powerful techniques to foster code reuse and improve code quality
- C provides a "poor man's" form of modular programming via the use of a special type (void *) and intelligent use of low-level memory operations and casting
- Enables functions that take pointer-valued arguments to be applied "polymorphically" i.e., applied to pointers that point to objects which have different representations
  - The key observation is that as long as the function does not need to directly inspect the contents of these objects, it can be applied to objects of many different types
    - E.g, swap, reverse, map, concat, …

Example

```
void swap_int (int* a, int* b) {
   int tmp = *a;
   *a = *b;
   *b = temp;
}
…
int x = 5; int y = 10;
swap_int(&x, &y);
…
```

Example

Suppose you want to swap two shorts

```
void swap_short (short *a, short* b) {
  short tmp = *a;
  *a = *b;
  *b = temp;
 }
```

Or two strings ...

```
void swap_string (char **a, char **b) {
  char *tmp = *a;
  …
}
```

Or...

```
void swap_float(float *a, float *b) { ... }
void swap_size_t(size_t *a, size_t *b) { ... }
void swap_double(double *a, double *b) { ... }
void swap_mystruct(mystruct *a, mystruct *b) { ... }
```

Design Pattern

- Take pointers to the values that need to be swapped
- Create temporary storage for one of the values
- Perform the move

```
void swap(t *data1, t *data2) {
      t tmp = *data1;
      *data1 = *data2;
      *data2 = t;
}
```

Challenge: each concrete instance of swap may need to allocate temporary storage with differing storage requirements:

- when swapping ints, `tmp` occupies 4 bytes
- when swapping shorts, `tmp` occupies 2 bytes
- when swapping strings, `tmp` occupies 8 bytes (on a 64-bit machine)
- …

Generic Swap

- Abstract datatype representation using void*
- Supply details about the required size necessary to implement tmp as an extra argument
- Leverage low-level memory operation routines (and casts) to deal with the contents of the objects, when necessary

```
void swap (void *x, void *y, size_t nbytes) {
    char tmp[nbytes];
    // how do we assign to tmp?
```

- Problem: can’t dereference a ptr of type void * since the type does not provide enough information to indicate what the pointer points to!

Low-level memory operations

```
void *memcpy(void *dest, const void *src, size_t n);
```

- K & R pp. 250
- Copies the next `n` bytes pointed to by `src` to the location pointed to by `dest`. Does not support overlapping memory regions.

```
int   x = 10;
int   y = 13;
memcpy(&x, &y, sizeof(x)); // the same as x = y
```

```
void *memmove(void *dest, const void *src, size_t n);
```

- Copies the next `n` bytes pointed to by `src` to the location pointed to by `dest`. Works even if the regions overlap.

```
int main() {
  int x[8] = {1,2,3,4,5,6,7,8};
  memmove(&x[0],&x[4],sizeof(x[0]) * 3);
  for (int i = 0; i < 8; i++) {
    printf("%d ", x[i]);
  }
}
```

```
prints:
5 6 7 4 5 6 7 8
```

Generic Swap

```
void swap(void *x, void *y, size_t nbytes) {
  char tmp[nbytes];
  // store a copy of x in temporary storage
  memcpy(tmp, x, nbytes);
  // copy y to location of x
  memcpy(x, y, nbytes);
  // copy data in temporary storage to location of data2
  memcpy(y, tmp, nbytes);
}
```

- The use of void* allows a pointer to become 'polymorphic', capable of pointing to different objects at different times (e.g. across different calls)

- But, since the underlying type system doesn't explicitly support polymorphic types (e.g., no overloading, overriding, type parameters), we resort instead to manifesting polymorphism by operating on a uniform representation (i.e., a collection of bytes) whose contents can be ascribed a specific type by its consumers

Generics

- C's approach to generics can be effective, but is very weakly constrained
- No enforcement that underlying byte representation is being manipulated according to their logical type structure

```
int main() {
  int x = 256;
  int y = 255;
  swap(&x,&y,1);
  printf("%d\n", x);
  printf("%d\n", y);
}
```

This program prints:

```
0
511
```

Why?

Another Example

- Write a function that swaps the first and last element of an array.
  - Would like the function to operate over arrays containing arbitrary elements
  - Can we leverage our generic swap function for this purpose?

```
void swap_ends(void *arr, size_t nelems) {
    swap(arr, arr + nelems – 1, sizeof(*arr));
}
```

pointer to first element of the array

pointer to last element of the array

This doesn’t work … why?

Generics and arrays

Need to supply element size to the function:

```
void swap_ends(void *arr, size_t nelems, size_t elem_bytes) {
  swap(arr, arr + (nelems – 1) * elem_bytes, elem_bytes);


}
```

Reflects underlying memory representation

If nelems = 4, and
- array of integers, elem_bytes = 4, so end of array is 12 bytes after the beginning
- array of chars, elem_bytes = 1, so end of array is 4 bytes after the beginning
- array of strings, elem_bytes = 8 (on a 64-bit machine), so end of array is 24 bytes after the beginning

Generics and arrays

```
void swap_ends(void *arr, size_t nelems, size_t elem_bytes) {
  swap(arr, arr + (nelems – 1) * elem_bytes, elem_bytes);


}
```

- Still won't work: arr has type void* and we're not allowed to do pointer arithmetic (since we don't know the size of the elements being pointed to)
- Need to force C to think in terms of bytes!

```
void swap_ends(void *arr, size_t nelems, size_t elem_bytes) {
  swap(arr, (char *)arr + (nelems – 1) * elem_bytes, elem_bytes);


}
```

Casts the void* array to a char* array. Now, all computation is performed on bytes

Generics and arrays

```
void swap_ends(void *arr, size_t nelems, size_t elem_bytes) {
  swap(arr, (char *)arr + (nelems – 1) * elem_bytes, elem_bytes);


}
```

```
int nums[] = {5, 2, 3, 4, 1};
size_t nelems = sizeof(nums) / sizeof(nums[0]);
swap_ends(nums, nelems, sizeof(nums[0]));
```

```
char *strs[] = {"Hi", "Hello", "Howdy"};
size_t nelems = sizeof(strs) / sizeof(strs[0]);
swap_ends(strs, nelems, sizeof(strs[0]));
```

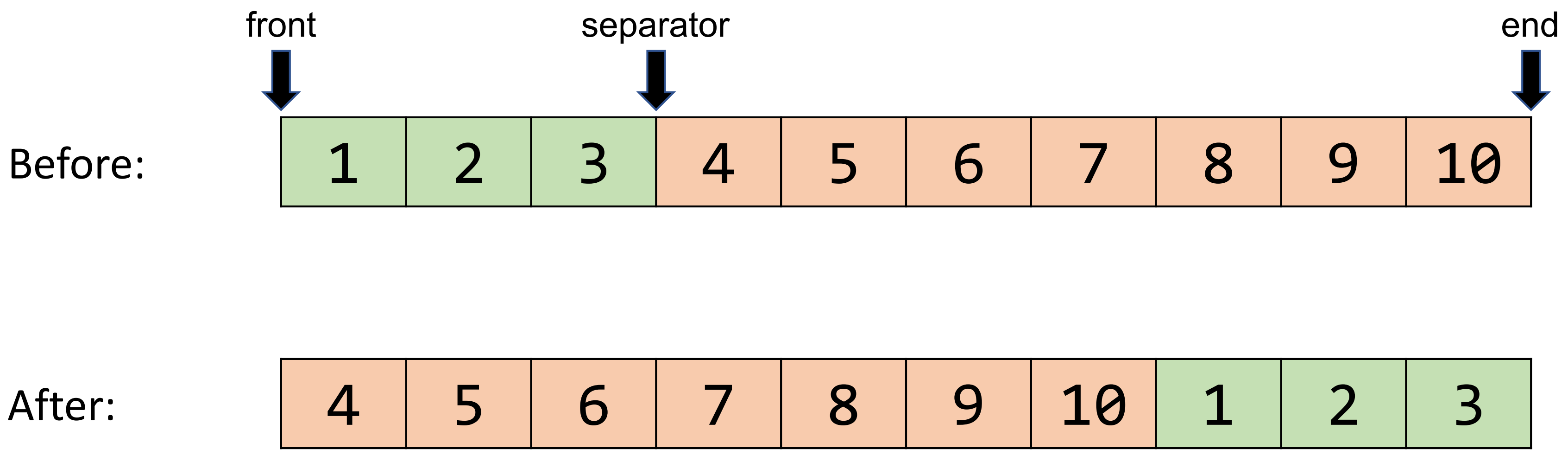
Exercise

Implement a function called `rotate` that has the following signature:

```
void rotate(void *front, void *sep, void *end);
```

The idea is that `front` represents the start of the array, `end` is the address of the end of the array, and `sep` is the address of some element in between. The function moves all elements between `front` and `sep` to the end of the array, and all elements between `sep` and `end` to the front.

```
int array[7] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
rotate(array, array + 3, array + 10);
```


front
separator
end
Before:
1 2 3 4 5 6 7 8 9 10
After:
4 5 6 7 8 9 10 1 2 3