601.220 Intermediate Programming

Spring 2023, Day 11 (February 15)

Today's agenda

- Exercise 10 review
- Dynamic memory allocation, Valgrind
- Exercise 11

Reminders/Announcements

- HW2: due Friday
  - Written assignment, late submissions not accepted

Exercise 10 review

Important application of pointers: *pass by reference* semantics for normal variables.

(Arrays are always passed by reference, but ordinary variables are passed by value by default.)

Exercise 10 review

`getDate` function:

```
int getDate(int *m, int *d, int *y) {
  int month, day, year;
  int rc = scanf("%d/%d/%d", &month, &day, &year);
  if (rc == 3) {
    *m = month;
    *d = day;
    *y = year;
  }
  return rc;
}
```

Exercise 10 review

`months` array:

```
const char *months[] = {
  "Jan", "Feb", "Mar", "Apr", "May", "Jun",
  "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"
};
```

Calling `getDate` from `main`:

```
printf("Enter a date: ");
while (getDate(&mon, &day, &yr) == 3) {
  printf("%s %d, %d\n", months[mon-1], day, yr);
  printf("Enter a date: ");
}
```

Exercise 10 review

Even more concise version of `getDate`:

```
int getDate(int *m, int *d, int *y) {
  return scanf("%d/%d/%d", m, d, y);
}
```

Day 11 recap questions

1. What is the difference between stack and heap memory?
2. What is dynamic memory allocation in C?
3. What is the memory leak problem?
4. What is the difference between *malloc*, *realloc*, and *calloc*?
5. What do we use valgrind to check for?
6. Consider the `exclaim` function below. Do you see any problems with this function?

1. What is the difference between stack and heap memory?

Stack memory: used for local variables, parameters, and other data required by an in-progress function call.

Key characteristic: the *lifetime* of local variables and parameters is limited to the duration of the function call for which they are allocated. (Storage for local variables, parameters, etc. is allocated automatically in a *stack frame* created when a function is called.)

Heap memory: chunks of memory can be allocated in a dedicated region of memory (the "heap").

Key characteristic: the lifetime of variables allocated in the heap is under the explicit control of the program. (I.e., the program decides when a dynamically allocated variable is no longer needed.)

2. What is dynamic memory allocation in C?

The program uses `malloc` (or `calloc`, or `realloc`) to dynamically allocate a chunk of memory of a specified size. The program can then use the chunk as a single variable, an array, etc. For example:

```
// dynamically allocate an array of 10 int elements
int *arr = (int *) malloc(10 * sizeof(int));
for (int i = 0; i < 10; i++) {
  arr[i] = (i + 1);
}
```

Dynamically allocated memory must be explicitly de-allocated with `free` when the program no longer needs it:

```
free(arr);
```

3. What is the memory leak problem?

If a program dynamically allocates memory but does not free it, it continues to exist in the heap.

The maximum amount of memory which can be allocated in the heap is finite, so a program that repeatedly allocates memory without freeing it could eventually exhaust the heap, which would cause subsequent attempts to allocate memory to fail.

Programs must take care to de-allocate dynamically allocated memory after the last use.

4. What is the difference between *malloc*, *realloc*, and *calloc*?

`malloc`: dynamically allocate block of memory of specified size.

`calloc`: like `malloc`, but contents are filled with zeroes (useful for arrays, guarantees that all elements are 0.)

`realloc`: attempt to reallocate an existing chunk of memory. Reallocation could be done "in place", or could involve allocating a new chunk of memory and copying the contents of the original block of memory.

5. What do we use valgrind to check for?

`valgrind` can check for:

1. Memory leaks (detected when the program exist)
2. Memory errors, such as

- out of bounds array accesses
- use of an uninitialized value
- access to heap memory not currently in use (e.g., dereferencing a pointer to a de-allocated block of dynamically allocated memory)

Why valgrind is useful

Testing your program regularly using `valgrind` is incredibly helpful!

- Just because your program "works" when you run it, doesn't mean that it is free from bugs
- The kinds of bugs `valgrind` finds often lead to subtle data corruptions that can be difficult to track down by other means

Use it!

6. Consider the `exclaim` function below. Do you see any problems with this function?

The code:

```
char* exclaim(int n) {
  char s[20];
  assert(n < 20);
  for (int i = 0; i < n; i++) {
    s[i] = '!';
  }
  s[n] = '\0';
  return s;
}
```

Exercise 11

- Dynamic allocation
- Using `valgrind` to detect memory leaks and other memory errors
- Pointers to pointers
- Talk to us if you have questions!

Notes

Notes

Notes

Notes

Notes