601.220 Intermediate Programming

Dynamic memory allocation

Outline

- Stack frame vs. heap memory
- Dynamic memory allocation using `malloc` and `free`
- `realloc` and `calloc`

Limitations of arrays allocated within a stack frame

- Last time, we saw that arrays allocated within a stack frame ("static allocation") have several limitations
  - Size of array is limited by size of stack frame
  - Arrays created within a called functions stack frame can't be accessed by calling function (since lifetime of array ends when called function returns)
  - Prior to C99, another limitation existed:
    - Needed to know size of array prior to run-time - couldn't ask for array of size `n` when `n` was a value input by user!
- To get around these limitations, we can use *dynamic allocation*

Dynamically-allocated memory

- Dynamically-allocated memory is located in a part of memory separate from the stack; it lives on "the heap"
- Dynamically-allocated memory lives as long as we like (until entire program ends)
  - We don't necessarily lose access to it when function call returns
    - This means we can return it to a calling function!
- Dynamically-allocated memory is not subject to size limitations based on stack frame size, since it's not part of the stack
- The size of a dynamically-allocated block of memory can be decided at run time

Dynamically-allocated memory

- Dynamically-allocated memory solves lots of problems. . .
- But there is a catch: since it is not automatically reclaimed when function call ends, *we are responsible for telling system when we're through with this memory*
  - that is, we need to remember to *deallocate* it
  - allocated memory is not available to other programs/users until we deallocate it
  - failing to deallocate memory is the cause of "memory leaks"

Dynamically-allocated memory

- To allocate memory, we can use a command named `malloc` (memory allocate) from `<stdlib.h>` (need to `#include`):

```
// allocate space for one int on heap
int *ip = malloc(sizeof(int));
// check if allocation succeeded
if (ip == NULL) { /*output error message*/ }
```

- After allocation with `malloc`, memory has not been initialized

```
// give dynamically-allocated int an initial value
*ip = 0;
```

Dynamically-allocated memory

- When usage of dynamically-allocated int is complete, deallocate it using *free* command on address of the memory on the heap:

```
// notify system that we're through with heap int
free(ip);

// avoid accidental attempt to use this pointer
// to access the released space later
ip = NULL;
```

Where should deallocation occur?

- Deallocation need not happen in same function where allocation occurred...
- ...but *some* function needs to deallocate the block of memory!
  - Programmer's responsibility is to determine where deallocation will occur, and then ensure that it really does happen

realloc

- Reallocates the given area of memory
- Can be used for both expanding and contracting
- The area must have been previously (dynamically) allocated
- The reallocation is done either by:
  - expanding or contracting the existing area, if possible
  - allocating a new memory block of new size bytes
- On success:
  - returns the pointer to the beginning of newly allocated memory
- On failure:
  - returns a null pointer

realloc example

```c
// realloc_example.c:
#include <stdio.h>
#include <stdlib.h>

int main()
{
  int *ptr = malloc(sizeof(int)*100);
  int i = 0;
  for (; i < 100; ++i) {
    ptr[i] = i;
  }

  ptr = realloc(ptr, sizeof(int) * 10000); // reallocate to expand

  for(i = 0; i < 10000; ++i) { // start from index 0 again
    ptr[i] = i;
  }
  return 0;
}
```

```
$ gcc -std=c99 -Wall -Wextra -pedantic realloc_example.c
```

calloc

- Similar to `malloc`, but initializes all bits to 0

```
#include <stdio.h>
#include <stdlib.h>

int main () {
  int *pData =  calloc (10, sizeof(int));
  for (int i = 0; i < 10; i++) {
      printf("%d ", pData[i]);
  }
}
```

```
$ gcc -std=c99 -Wall -Wextra -pedantic calloc_example.c
$ ./a.out
0 0 0 0 0 0 0 0 0 0
```