ECE 2400 Computer Systems Programming
Fall 2018

Topic 6: C Dynamic Allocation

School of Electrical and Computer Engineering
Cornell University

revision: 2018-09-20-23-40

1 Bookcase Analogy 2
2 Using malloc to Allocate Memory 3
3 Using free to Deallocate Memory 8
4 Mapping Conceptual Storage to Machine Memory 10
1. Bookcase Analogy

- Consider what happens if we run out of space in our bookcase
- We might create a “heap” of books next to the bookcase; this is a completely different space to store books with different mechanisms for adding and removing books
- The heap is a little less organized compared to the bookcase
- We use rolled up pieces of paper on the bookcase to point to various books in the heap
2. **Using malloc to Allocate Memory**

- Let’s revisit an example we saw in a previous topic
- Assume we wish to refactor appending a node to a chain into its own function

```c
#include <stddef.h>

typedef struct _node_t {
  int value;
  struct _node_t* next_ptr;
} node_t;

node_t* append( node_t* n_ptr, int value ) {
  node_t node;
  node.value = value;
  node.next_ptr = n_ptr;
  return &node;
}

int main( void ) {
  node_t* n_ptr = NULL;
  n_ptr = append( n_ptr, 3 );
  n_ptr = append( n_ptr, 4 );
  n_ptr = append( n_ptr, 5 );
  return 0;
}
```
2. Using malloc to Allocate Memory

```c
#include <stdlib.h>
#include <stddef.h>

int* rand_array( size_t n )
{
    int a[n];
    for ( size_t i=0; i<n; i++ )
        a[i] = rand() % 100;
    return a;
}

int main( void )
{
    int* a = rand_array(3);
    return 0;
}
```
2. Using malloc to Allocate Memory

- Dynamic memory allocation uses the heap (new region of memory)
- Because dynamically allocated variables are not on a function's stack frame, they are not deallocated when a function returns
- We can dynamically allocate variables on the heap using malloc
- malloc is defined in stdlib.h
- malloc takes the number of bytes to allocate as a parameter and returns a pointer to the new variable allocated on the heap

```c
int* a_ptr = malloc( sizeof(int) );
*a_ptr = 42;

int* b_ptr = malloc( 4 * sizeof(int) );

b_ptr[0] = 1;
b_ptr[1] = 2;
b_ptr[2] = 3;
b_ptr[3] = 4;

// typedef struct
// {
//   double real;
//   double imag;
// }
// complex_t;
complex_t* complex_ptr = malloc( sizeof(complex_t) );
complex_ptr->real = 1.5;
complex_ptr->imag = 3.5;
```
```c
#include <stdlib.h>
#include <stddef.h>

typedef struct _node_t
{
    int value;
    struct _node_t* next_ptr;
} node_t;

node_t* append( node_t* n_ptr, int value )
{
    node_t* new_ptr = malloc( sizeof(node_t) );
    new_ptr->value = value;
    new_ptr->next_ptr = n_ptr;
    return new_ptr;
}

int main( void )
{
    node_t* n_ptr = NULL;
    n_ptr = append( n_ptr, 3 );
    n_ptr = append( n_ptr, 4 );
    n_ptr = append( n_ptr, 5 );
    return 0;
}
```
2. Using malloc to Allocate Memory

```c
#include <stdlib.h>
#include <stddef.h>

int* rand_array( size_t n )
{
    int* x = malloc( n * sizeof(int) );
    for ( size_t i=0; i<n; i++ )
        x[i] = rand() % 100;
    return x;
}

int main( void )
{
    int* a = rand_array(3);
    return 0;
}
```

stack

heap
3. Using `free` to Deallocation Memory

- What happens if we call `rand_array` multiple times?
- What happens if we reuse the pointer to the result?

```c
#include <stdlib.h>
#include <stddef.h>

int* rand_array( size_t n )
{
    int* a = malloc( n * sizeof(int) );
    for ( size_t i=0; i<n; i++ )
        a[i] = rand() % 100;
    return a;
}

int main( void )
{
    int* a = rand_array(3);
    a = rand_array(3);
    return 0;
}
```
2. Using malloc to Allocate Memory

- Memory leaks can cause programs to crash
- Every call to malloc must eventually correspond to a call to free
- free is defined in stdlib.h
- free takes a pointer to a dynamically allocated variable

```c
#include <stdlib.h>
#include <stddef.h>

int* rand_array( size_t n )
{
    int* a = malloc( n * sizeof(int) );
    for ( size_t i=0; i<n; i++ )
        a[i] = rand() % 100;
    return a;
}

int main( void )
{
    int* a = rand_array(3);
    free(a);
    a = rand_array(3);
    free(a);
    return 0;
}
```
4. Mapping Conceptual Storage to Machine Memory

```c
int main( void ) {
    char a[] = "foo";
    const char* b = "bar";
    char* c = malloc( 4 * sizeof(char) );
    strcpy( c, a );
    free(c);
    return 0;
}
```

Memory
(4B word addr)

- stack
- heap
- static data
4. Mapping Conceptual Storage to Machine Memory