ECE 2400 Computer Systems Programming
Fall 2017

Topic 6: C Dynamic Allocation

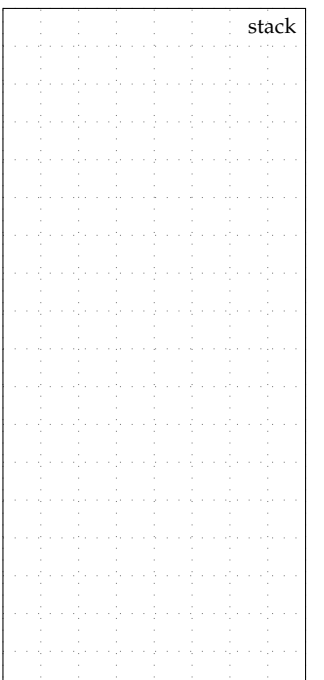
School of Electrical and Computer Engineering
Cornell University

revision: 2017-09-22-07-11

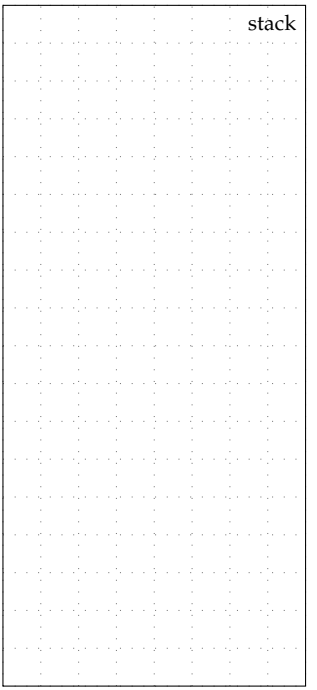
1. Using `malloc` to Allocate Memory

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

```
1  #include <stddef.h>
2
3  typedef struct _node_t
4  {
5    int              value;
6    struct _node_t* next_ptr;
7  }
8  node_t;
9
10 node_t* append( node_t* n_ptr,
11                 int value )
12 {
13   node_t node;
14   node.value    = value;
15   node.next_ptr = n_ptr;
16   return &node;
17 }
18
19 int main( void )
20 {
21   node_t* n_ptr = NULL;
22   n_ptr = append( n_ptr, 3 );
23   n_ptr = append( n_ptr, 4 );
24   n_ptr = append( n_ptr, 5 );
25   return 0;
26 }
```

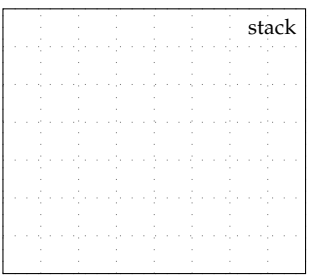


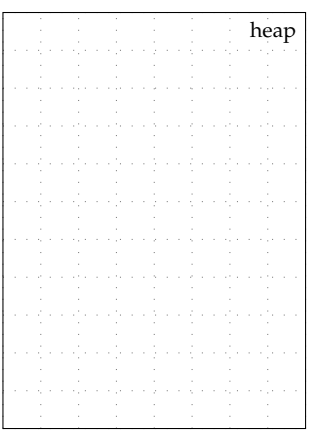
```c
#include <stdlib.h>
#include <stddef.h>

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

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



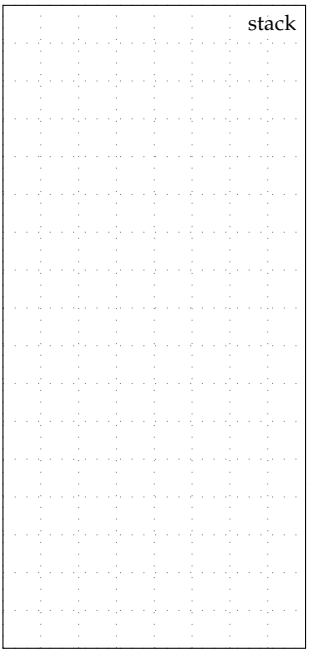
- Dynamic memory allocation uses the heap (new region of memory)
- Because dynamically allocated variables are not on a function's stack frame, they are not deallocated when a function returns
- We can dynamically allocate variables on the heap using `malloc`
- `malloc` is defined in `stdlib.h`
- `malloc`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

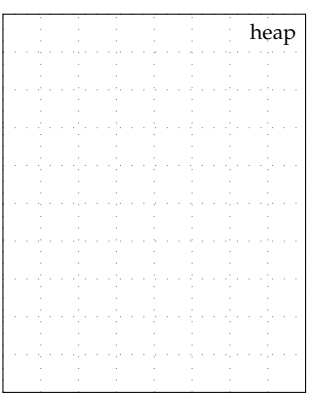
```
1  int* a_ptr =
2    malloc( sizeof(int) );
3
4  *a_ptr = 42;
5
6  int* b_ptr =
7    malloc( 4 * sizeof(int) );
8
9  b_ptr[0] = 1;
10 b_ptr[1] = 2;
11 b_ptr[2] = 3;
12 b_ptr[3] = 4;
13
14 // typedef struct
15 // {
16 //   double real;
17 //   double imag;
18 // }
19 // complex_t;
20
21 complex_t* complex =
22   malloc( sizeof(complex_t) );
23
24 complex->real = 1.5;
25 complex->imag = 3.5;
```



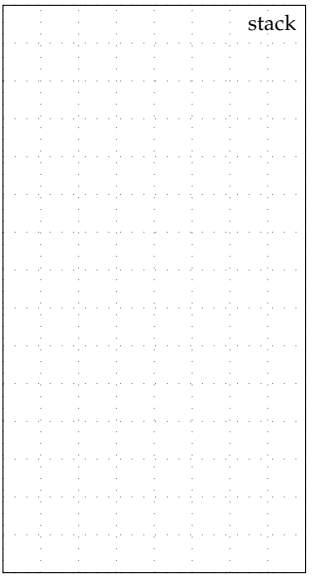


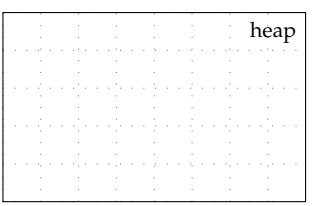
```c
#include <stdlib.h>
#include <stddef.h>

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;
}
```





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

int* rand_array( size_t size )
{
  int* a =
    malloc( size * sizeof(int) );

  for ( size_t i=0; i<size; i++ )
    a[i] = rand() % 100;

  return a;
}

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

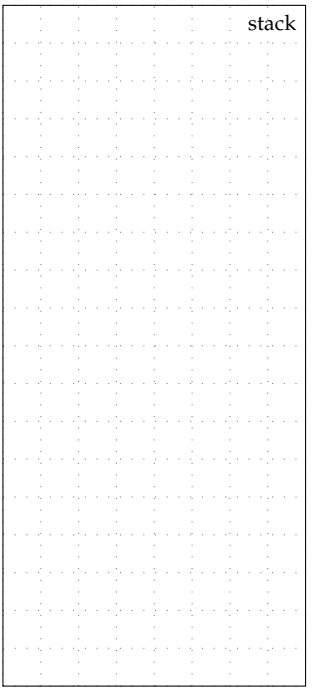


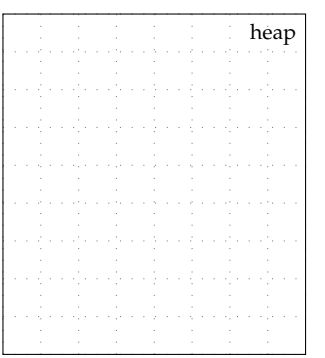


2. Using free to Deallocate Memory

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

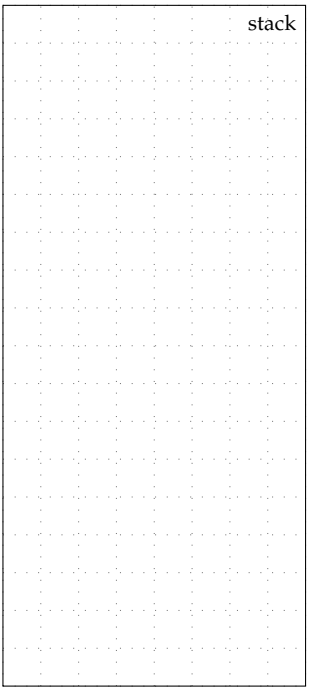
```
1  #include <stdlib.h>
2  #include <stddef.h>
3
4  int* rand_array( size_t size )
5  {
6    int* a =
7      malloc( size * sizeof(int) );
8
9    for ( size_t i=0; i<size; i++ )
10     a[i] = rand() % 100;
11
12   return a;
13 }
14
15 int main( void )
16 {
17   int* a = rand_array(3);
18   a = rand_array(3);
19   return 0;
20 }
```





- 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

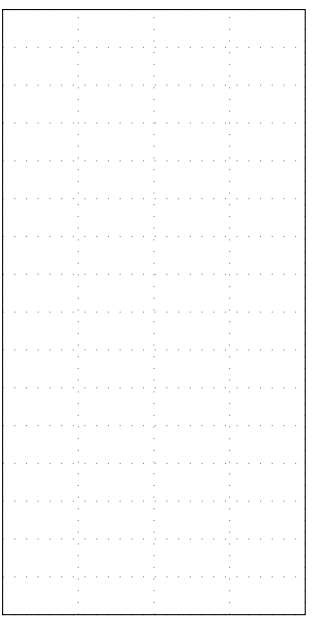
```
1  #include <stdlib.h>
2  #include <stddef.h>
3
4  int* rand_array( size_t size )
5  {
6    int* a =
7      malloc( size * sizeof(int) );
8
9    for ( size_t i=0; i<size; i++ )
10     a[i] = rand() % 100;
11
12   return a;
13 }
14
15 int main( void )
16 {
17   int* a = rand_array(3);
18   free(a);
19   a = rand_array(3);
20   free(a);
21   return 0;
22 }
```

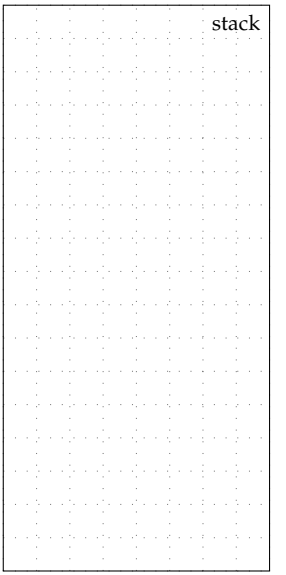


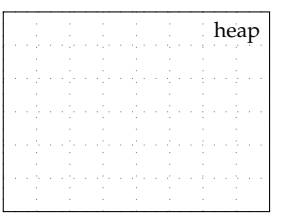
3. Mapping Conceptual Storage to Machine Memory

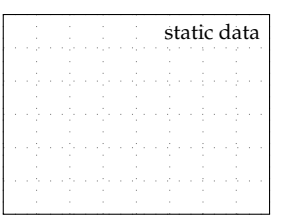
```
1  int main( void )
2  {
3    char a[] = "foo";
4    const char* b = "bar";
5    char* c =
6      malloc( 4 * sizeof(char) );
7    strcpy( c, a );
8    free(c);
9    return 0;
10 }
```

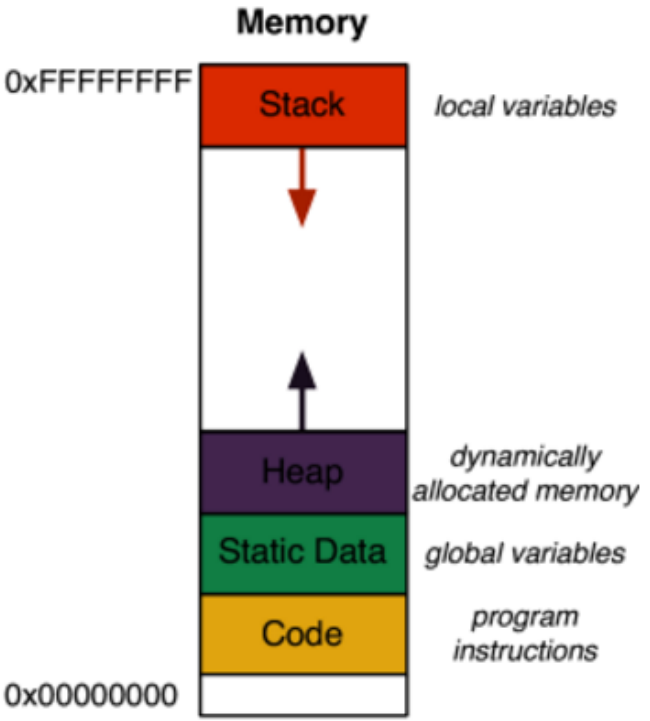
Memory
(4B word addr)










Memory
0xFFFFFFFF
Stack
local variables
Heap
dynamically
allocated memory
Static Data
global variables
Code
program
instructions
0x00000000