

ECE 2400 Computer Systems Programming

Fall 2021

Topic 6: C Dynamic Allocation

School of Electrical and Computer Engineering
Cornell University

revision: 2021-08-28-13-33

1	Using <code>malloc</code> to Allocate Memory	2
2	Using <code>free</code> to Deallocate Memory	8
3	Mapping Conceptual Storage to Machine Memory	10

zyBooks The zyBooks logo is used to indicate additional material included in the course zyBook which will not be discussed in detail in lecture. Students are responsible for all material covered in lecture and in the course zyBook.

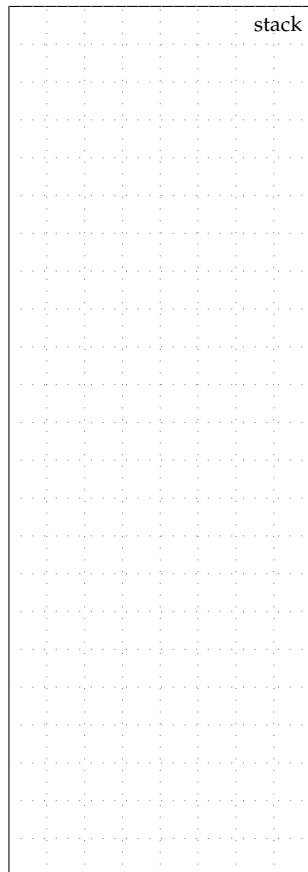
Copyright © 2021 Christopher Batten. All rights reserved. This handout was prepared by Prof. Christopher Batten at Cornell University for ECE 2400 / ENGRD 2140 Computer Systems Programming. Download and use of this handout is permitted for individual educational non-commercial purposes only. Redistribution either in part or in whole via both commercial or non-commercial means requires written permission.

1. 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

Draw a state diagram corresponding to the execution of this program

```
0001 typedef struct _node_t
0002 {
0003     int          value;
0004     struct _node_t* next_ptr;
0005 }
0006 node_t;
0007
0008 node_t* append( node_t* n_ptr,
0009                int value )
0010 {
0011     node_t node;
0012     node.value = value;
0013     node.next_ptr = n_ptr;
0014     return &node;
0015 }
0016
0017 int main( void )
0018 {
0019     node_t* n_ptr = NULL;
0020     n_ptr = append( n_ptr, 3 );
0021     n_ptr = append( n_ptr, 4 );
0022     return 0;
0023 }
```



- Let's consider a similar idea for arrays
- Assume we wish to refactor allocating an array and then initializing all elements to zero into its own function

```
1  #include <stddef.h>
2
3  int* init_array( int n )
4  {
5      int x[n];
6
7      for ( int i=0; i<n; i++ )
8          x[i] = 0;
9
10     return x;
11 }
12
13 int main( void )
14 {
15     int* a = init_array(3);
16     return 0;
17 }
```

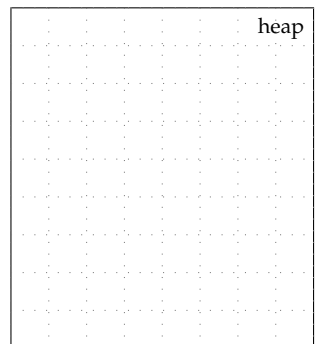
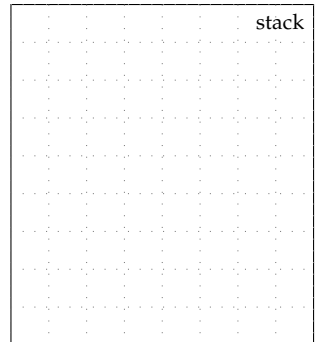
List two errors with this function:

1. _____

2. _____

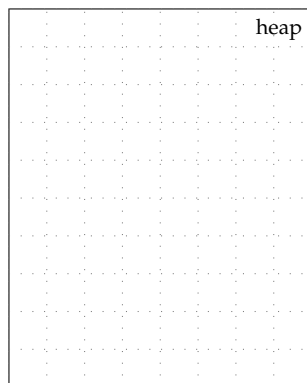
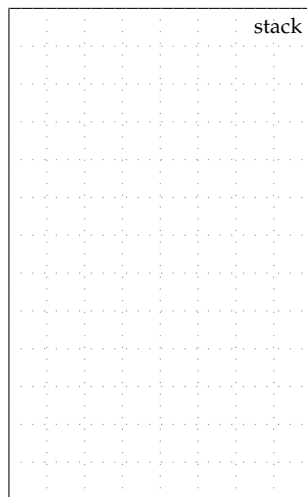
- **Dynamic memory allocation** uses the **heap** (new region of memory)
- Because dynamically allocated variables are not on a function's stack frame, they are not deallocated when a function returns
- We can dynamically allocate variables on the heap using `malloc`
- `malloc` takes the number of bytes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

```
□□□ 01 int* a_ptr =
□□□ 02     malloc( sizeof(int) );
□□□ 03
□□□ 04 *a_ptr = 42;
□□□ 05
□□□ 06 int* b_ptr =
□□□ 07     malloc( 4 * sizeof(int) );
□□□ 08
□□□ 09 b_ptr[0] = 10;
□□□ 10 b_ptr[1] = 11;
□□□ 11 b_ptr[2] = 12;
□□□ 12 b_ptr[3] = 13;
```



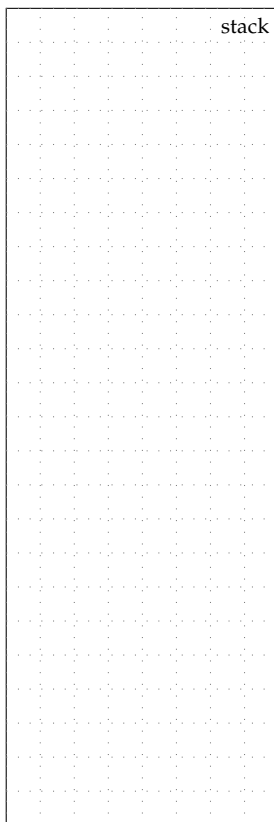
Draw a state diagram corresponding to the execution of this program

```
□□□ 01 typedef struct
□□□ 02 {
□□□ 03     double real;
□□□ 04     double imag;
□□□ 05 }
□□□ 06 complex_t;
□□□ 07
□□□ 08 int main( void )
□□□ 09 {
□□□ 10     complex_t* c_ptr0 =
□□□ 11         malloc( sizeof(complex_t) );
□□□ 12
□□□ 13     c_ptr0->real = 1.5;
□□□ 14     c_ptr0->imag = 3.5;
□□□ 15
□□□ 16     complex_t* c_ptr1 =
□□□ 17         malloc( sizeof(complex_t) );
□□□ 18
□□□ 19     c_ptr1->real = c_ptr0->real;
□□□ 20     c_ptr1->imag = c_ptr0->imag;
□□□ 21
□□□ 22     return 0;
□□□ 23 }
```



- Assume we wish to refactor appending a node to a chain into its own function

```
01 typedef struct _node_t
02 {
03     int          value;
04     struct _node_t* next_ptr;
05 }
06 node_t;
07
08 node_t* append( node_t* n_ptr,
09                int value )
10 {
11     node_t* new_ptr =
12         malloc( sizeof(node_t) );
13
14     new_ptr->value    = value;
15     new_ptr->next_ptr = n_ptr;
16     return new_ptr;
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     return 0;
25 }
```



- Assume we wish to refactor allocating an array and then initializing all elements to zero into its own function

```
1  #include <stddef.h>
2
3  int* init_array( int n )
4  {
5      int* x = malloc( n * sizeof(int) );
6
7      for ( int i=0; i<n; i++ )
8          x[i] = 0;
9
10     return x;
11 }
12
13 int main( void )
14 {
15     int* a = init_array(3);
16     return 0;
17 }
```

How does this address the two errors we identified earlier?

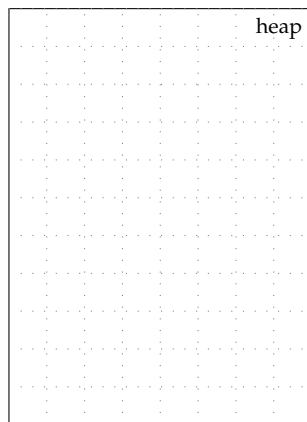
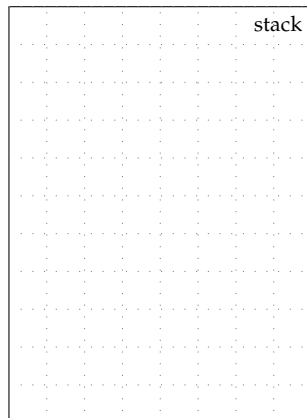
1. _____

2. _____

2. Using free to Deallocate Memory

Draw a state diagram corresponding to the execution of this program

```
□□□ 01 typedef struct
□□□ 02 {
□□□ 03     double real;
□□□ 04     double imag;
□□□ 05 }
□□□ 06 complex_t;
□□□ 07
□□□ 08 int main( void )
□□□ 09 {
□□□ 10     complex_t* c_ptr =
□□□ 11         malloc( sizeof(complex_t) );
□□□ 12
□□□ 13     c_ptr->real = 1.5;
□□□ 14     c_ptr->imag = 3.5;
□□□ 15
□□□ 16     c_ptr =
□□□ 17         malloc( sizeof(complex_t) );
□□□ 18
□□□ 19     c_ptr->real = 2.5;
□□□ 20     c_ptr->imag = 4.5;
□□□ 21
□□□ 22     return 0;
□□□ 23 }
```

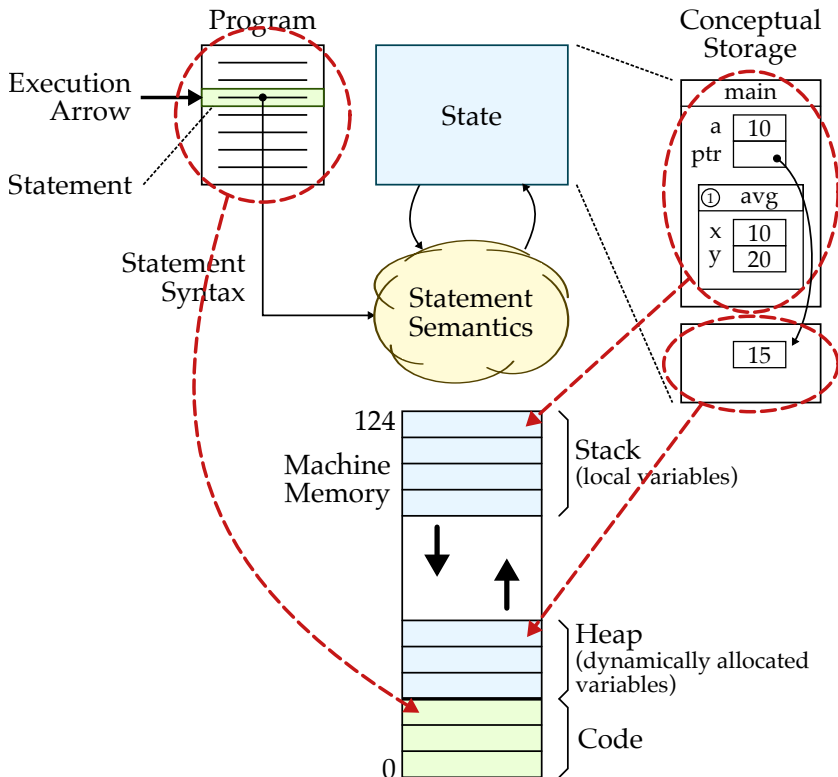


- Every call to malloc must have corresponding call to free
- free takes a pointer to a dynamically allocated variable

```
1  typedef struct
2  {
3      double real;
4      double imag;
5  }
6  complex_t;
7
8  int main( void )
9  {
10     complex_t* c_ptr =
11         malloc( sizeof(complex_t) );
12
13     c_ptr->real = 1.5;
14     c_ptr->imag = 3.5;
15
16     free( c_ptr );
17
18     c_ptr =
19         malloc( sizeof(complex_t) );
20
21     c_ptr->real = 2.5;
22     c_ptr->imag = 4.5;
23
24     free( c_ptr );
25
26     return 0;
27 }
```

3. Mapping Conceptual Storage to Machine Memory

- Recall that our current use of state diagrams is conceptual
- Real machine uses **memory** to store variables
- Real machine does not use “arrows”, uses **memory addresses**
- Heap is stored above code and grows *up*



Machine memory in real systems

- Machine memory size ranges from KBs (embedded) to TBs (server)
- Lowest address range reserved to detect NULL pointer dereference
- Static data region is used for global variables
- Machine memory as shown is really the *virtual memory space*
- Different programs have their own virtual memory spaces mapped to a single large *physical memory space*

