ECE 2400 Computer Systems Programming Spring 2025

Topic 6: C Dynamic Allocation

School of Electrical and Computer Engineering
Cornell University

revision: 2026-02-18-10-18

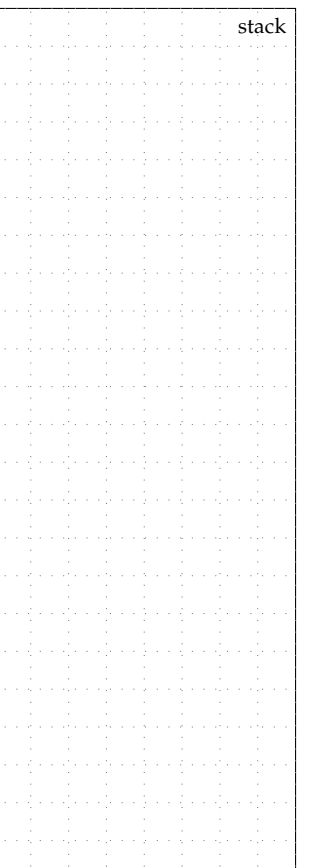
zyBooks logo indicates readings and coding labs in the course zyBook which will not be discussed in detail in lecture. Students are responsible for all material covered in lecture and in the course zyBook.

1. Using `malloc` to Allocate Memory

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

Draw a state diagram corresponding to the execution of this program

```
01 typedef struct _node_t
02 {
03   int              value;
04   struct _node_t* next_p;
05 }
06 node_t;
07
08 node_t* prepend( node_t* head_p,
09                  int v )
10 {
11   node_t node;
12   node.value  = v;
13   node.next_p = head_p;
14   return &node;
15 }
16
17 int main( void )
18 {
19   node_t* head_p = NULL;
20   head_p = prepend( head_p, 3 );
21   head_p = prepend( head_p, 4 );
22   return 0;
23 }
```



- 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  int* init_array( int n )
2  {
3    int x[n];
4
5    for ( int i=0; i<n; i++ )
6      x[i] = 0;
7
8    return x;
9  }
10
11 int main( void )
12 {
13   int* a = init_array(3);
14   return 0;
15 }
```

List two errors with this function:

1. _____

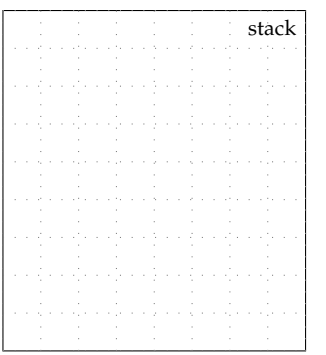
   _____

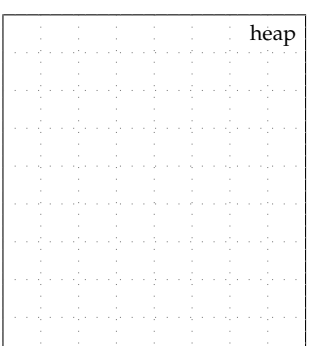
2. _____

   _____

- 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`

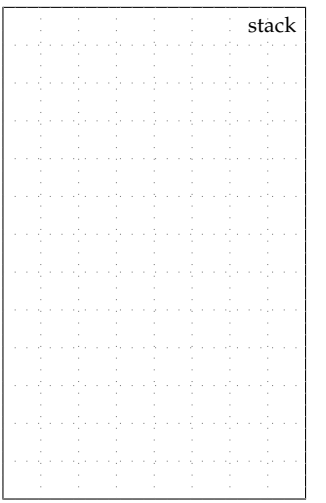
```
01 int* a_ptr =
02   malloc( sizeof(int) );
03
04 *a_ptr = 42;
05
06 int* b_ptr =
07   malloc( 4 * sizeof(int) );
08
09 b_ptr[0] = 10;
10 b_ptr[1] = 11;
11 b_ptr[2] = 12;
12 b_ptr[3] = 13;
```

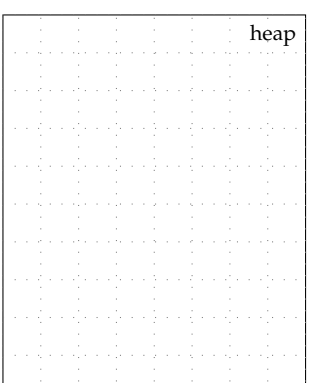




Draw a state diagram corresponding to the execution of this program

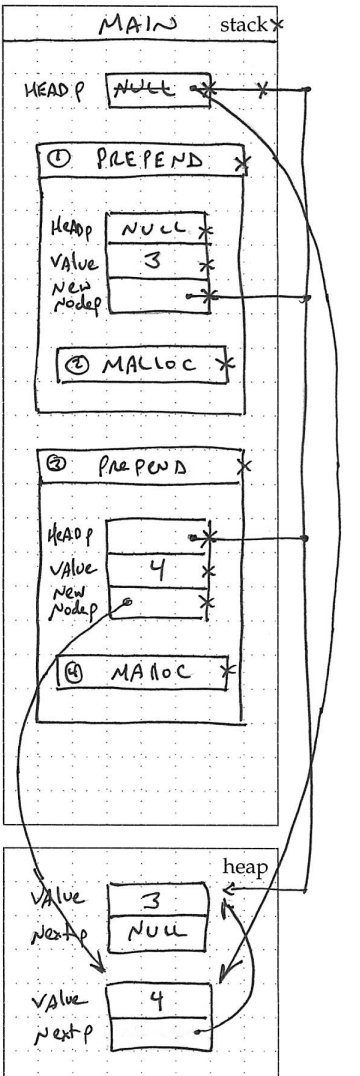
```
01 typedef struct
02 {
03   double real;
04   double imag;
05 }
06 complex_t;
07
08 int main( void )
09 {
10   complex_t* c_ptr0 =
11     malloc( sizeof(complex_t) );
12
13   c_ptr0->real = 1.5;
14   c_ptr0->imag = 3.5;
15
16   complex_t* c_ptr1 =
17     malloc( sizeof(complex_t) );
18
19   c_ptr1->real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





- Assume we wish to refactor prepending a node to the front of a chain of nodes into its own function

```
01 typedef struct _node_t
02 {
03   int              value;
04   struct _node_t* next_p;
05 }
06 node_t;
07
08 node_t* prepend( node_t* head_p,
09                  int v )
10 {
11   node_t* new_node_p =
12     malloc( sizeof(node_t) );
13
14   new_node_p->value  = v;
15   new_node_p->next_p = head_p;
16   return new_node_p;
17 }
18
19 int main( void )
20 {
21   node_t* head_p = NULL;
22   head_p = prepend( head_p, 3 );
23   head_p = prepend( head_p, 4 );
24   return 0;
25 }
```



zyBooks The course zyBook includes a coding lab to implement a function to append a node to the *back* of a chain of nodes.

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

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

How does this address the two errors we identified earlier?

1. _____

_____

2. _____

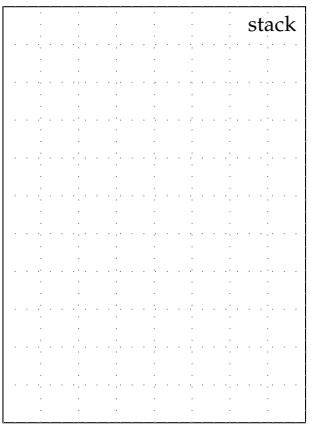
_____

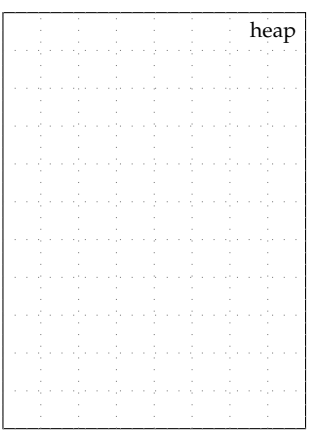
zyBooks The course zyBook includes a coding lab to implement a function to duplicate a given array on the heap and return a pointer to this newly allocated array.

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 }
```



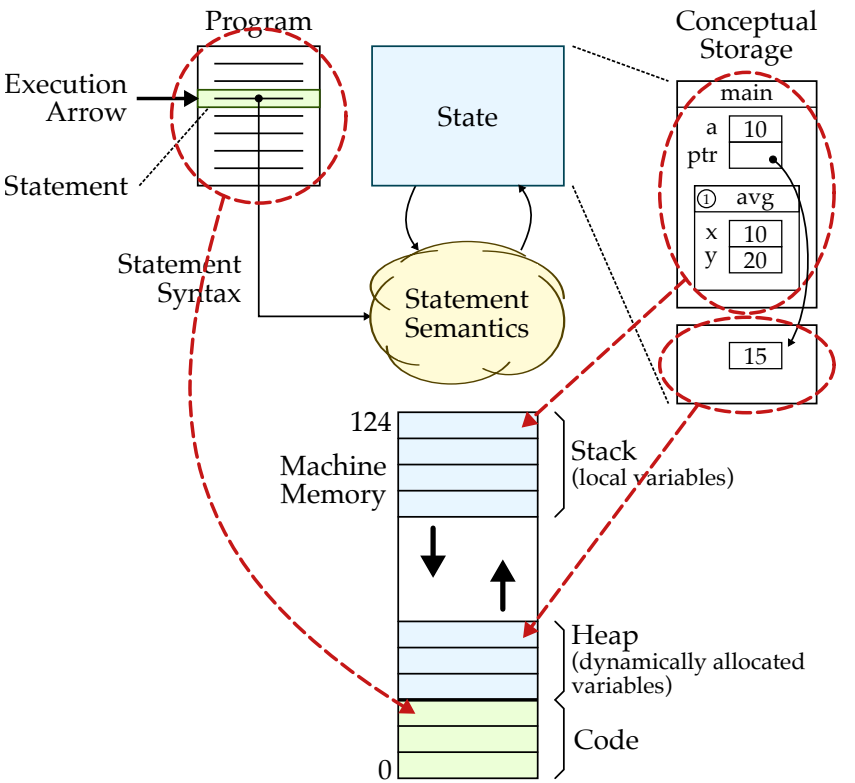


- Every call to malloc must have corresponding call to free
- free takes a pointer to a dynamically allocated variable

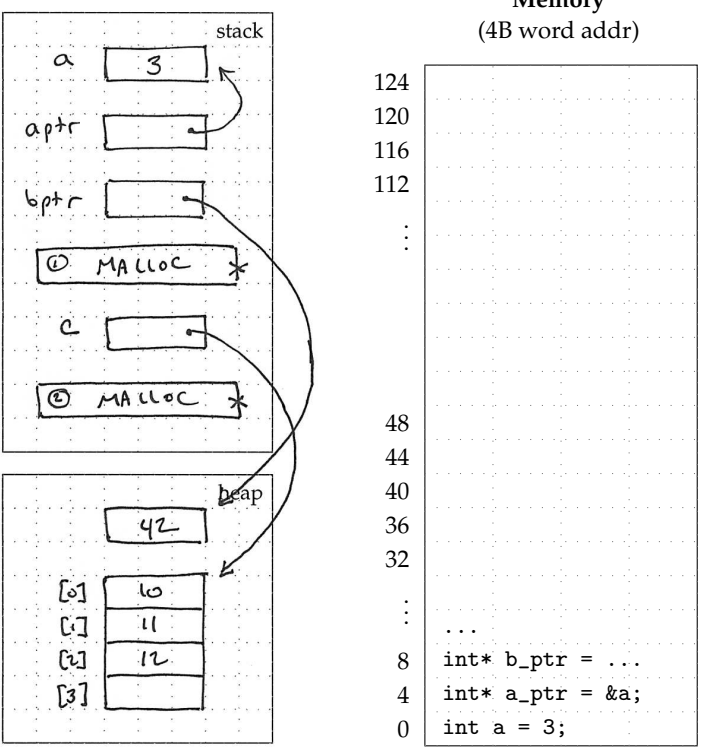
```
typedef struct
{
  double real;
  double imag;
}
complex_t;

int main( void )
{
  complex_t* c_ptr =
    malloc( sizeof(complex_t) );

  c_ptr->real = 1.5;
  c_ptr->imag = 3.5;

  free( c_ptr );

  c_ptr =
    malloc( sizeof(complex_t) );

  c_ptr->real = 2.5;
  c_ptr->imag = 4.5;

  free( c_ptr );

  return 0;
}
```

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*



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



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*