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

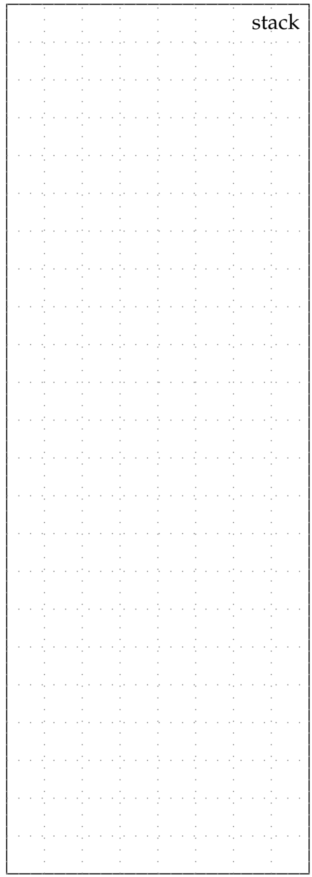
zyBooks The zyBooks logo is used to indicate additional material included in the course zyBook which will not be discussed in detail in lecture. Students are responsible for all material covered in lecture and in the course zyBook.

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

```c
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 );
  return 0;
}
```



- 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

```c
#include <stddef.h>

int* init_array( int n )
{
  int x[n];

  for ( int i=0; i<n; i++ )
    x[i] = 0;

  return x;
}

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

List two errors with this function:

1. ______________________________

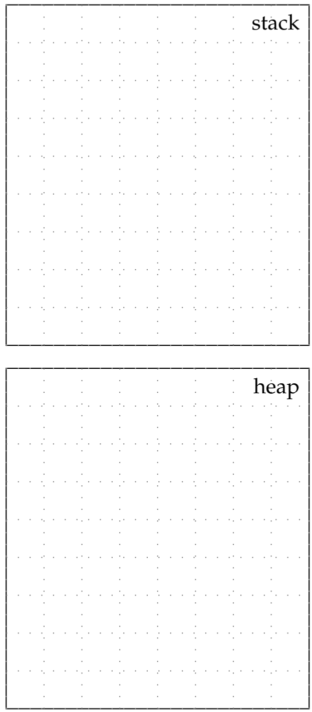
   ____

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

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

*a_ptr = 42;

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

b_ptr[0] = 10;
b_ptr[1] = 11;
b_ptr[2] = 12;
b_ptr[3] = 13;
```



Draw a state diagram corresponding to the execution of this program

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

int main( void )
{
  complex_t* c_ptr0 =
    malloc( sizeof(complex_t) );

  c_ptr0->real = 1.5;
  c_ptr0->imag = 3.5;

  complex_t* c_ptr1 =
    malloc( sizeof(complex_t) );

  c_ptr1->real = c_ptr0->real;
  c_ptr1->imag = c_ptr0->imag;

  return 0;
}
```

stack

heap

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

```
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 );
  return 0;
}
```

stack

heap

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

```c
#include <stddef.h>

int* init_array( int n )
{
  int* x = malloc( n * sizeof(int) );

  for ( int i=0; i<n; i++ )
    x[i] = 0;

  return x;
}

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

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

```
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;

  c_ptr =
    malloc( sizeof(complex_t) );

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

  return 0;
}
```

stack

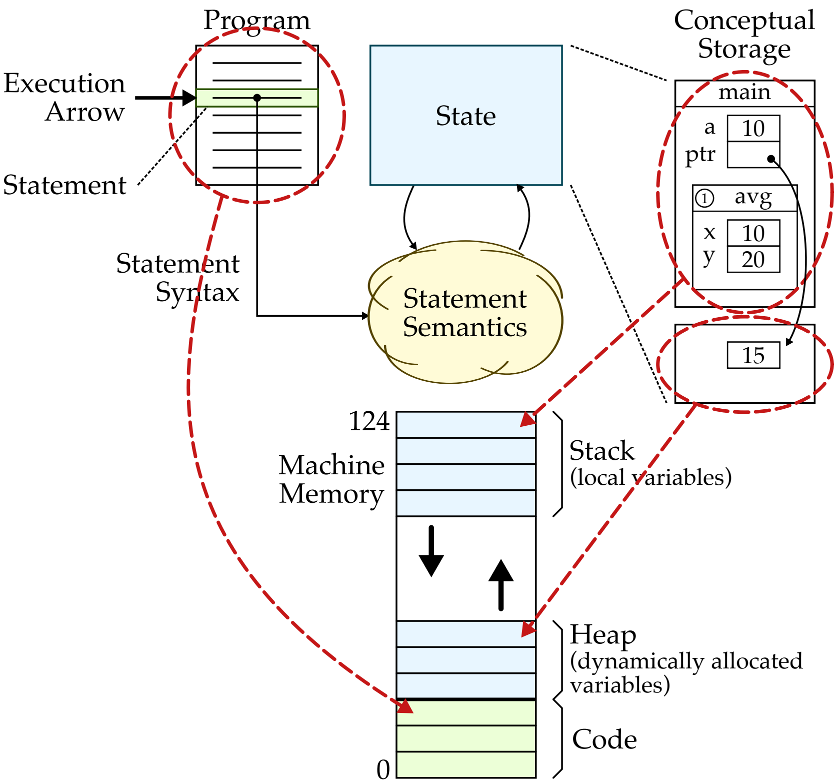
heap

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

```c
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;
}
```

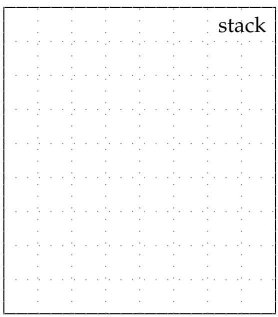
3. Mapping Conceptual Storage to Machine Memory

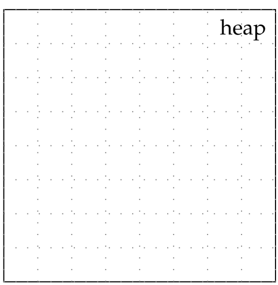
- Recall that our current use of state diagrams is conceptual
- Real machine uses memory to store variables
- Real machine does not use "arrows", uses memory addresses
- Heap is stored above code and grows *up*

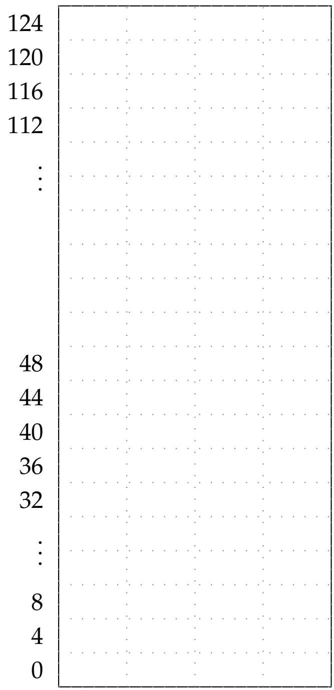


```c
int  a = 3;
int* a_ptr = &a;

int* b_ptr =
  malloc( sizeof(int) );
*b_ptr = 42;

int* c =
  malloc( 4 * sizeof(int) );
c[0] = 10;
c[1] = 11;
c[2] = 12;
c[3] = 13;
```

Memory
(4B word addr)







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*