CS113: Lecture 9

Topics:

- Dynamic Allocation
- Dynamic Data Structures

What's wrong with this?

```
char *big_array( char fill ) {
   char a[1000]; int i;
   for( i = 0; i < 1000; i++ )
      a[i] = fill;
   return a;
}

void main() {
   char *b;
   b = big_array( 'z' );
}
```

It's the usual thing: the activation record for `big_array` gets destroyed after the function returns, so `b` isn't pointing to anything stable.

But what about dynamic allocation?

Remember the good old days of Java programming?

- If you want, say, a new Vector, you just use the `new` operator to create one! When you have it you can fill it with as much crap as you want.

- The following Java code works just fine:

```
Vector getVector() {
  return new Vector();
}

void main() {
  Vector v = getVector();
  /* do stuff with the vector */
}
```

Why does this work?

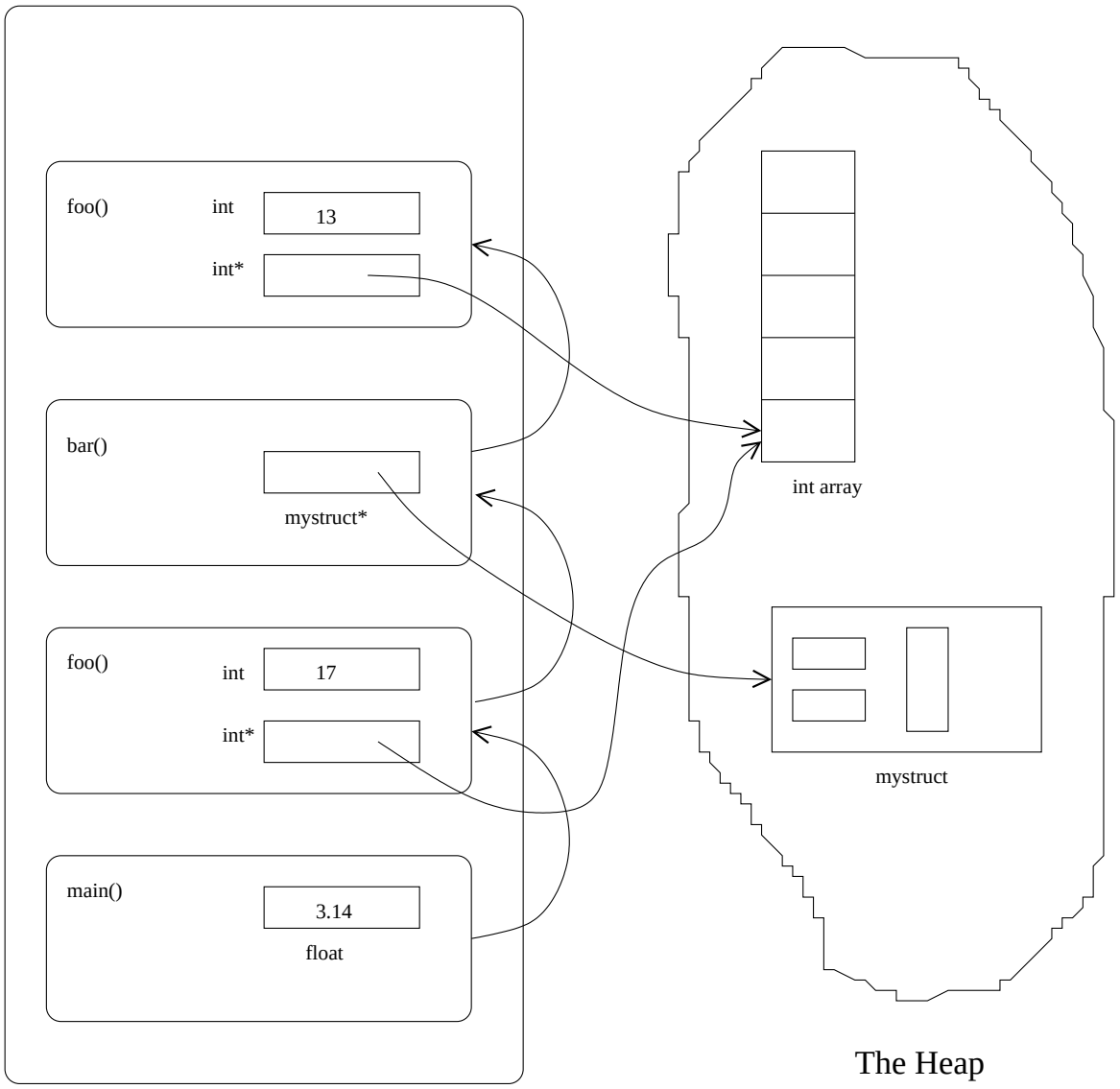
- Storage space for objects is allocated dynamically, *outside of the activation record*.

- Within a function, the activation record keeps a *pointer* to the dynamic object. (You've been using pointers all this time!)

- When you're done with an object, the Java VM determines this and *garbage collects* the memory.

I present to you: The Heap

C lets you allocate memory dynamically too, but it's all explicitly controlled by the programmer. Dynamic memory is stored outside of the activtion records for functions, in a memory area called "the heap":



Your keys to the heap: malloc and free

Use `malloc` (for "memory allocate") to allocate memory in the heap.

- `malloc()` takes an unsigned integer representing the number of bytes to allocate
- It has type `(void*)`, so you *must* cast it to another pointer type before using it

When you're done with the memory, use `free` to free up the memory space in the heap.

- `free` takes a pointer to a chunk of memory freed by `malloc`
- K & R say: "it is a ghastly error to free something not obtained by calling `malloc`"

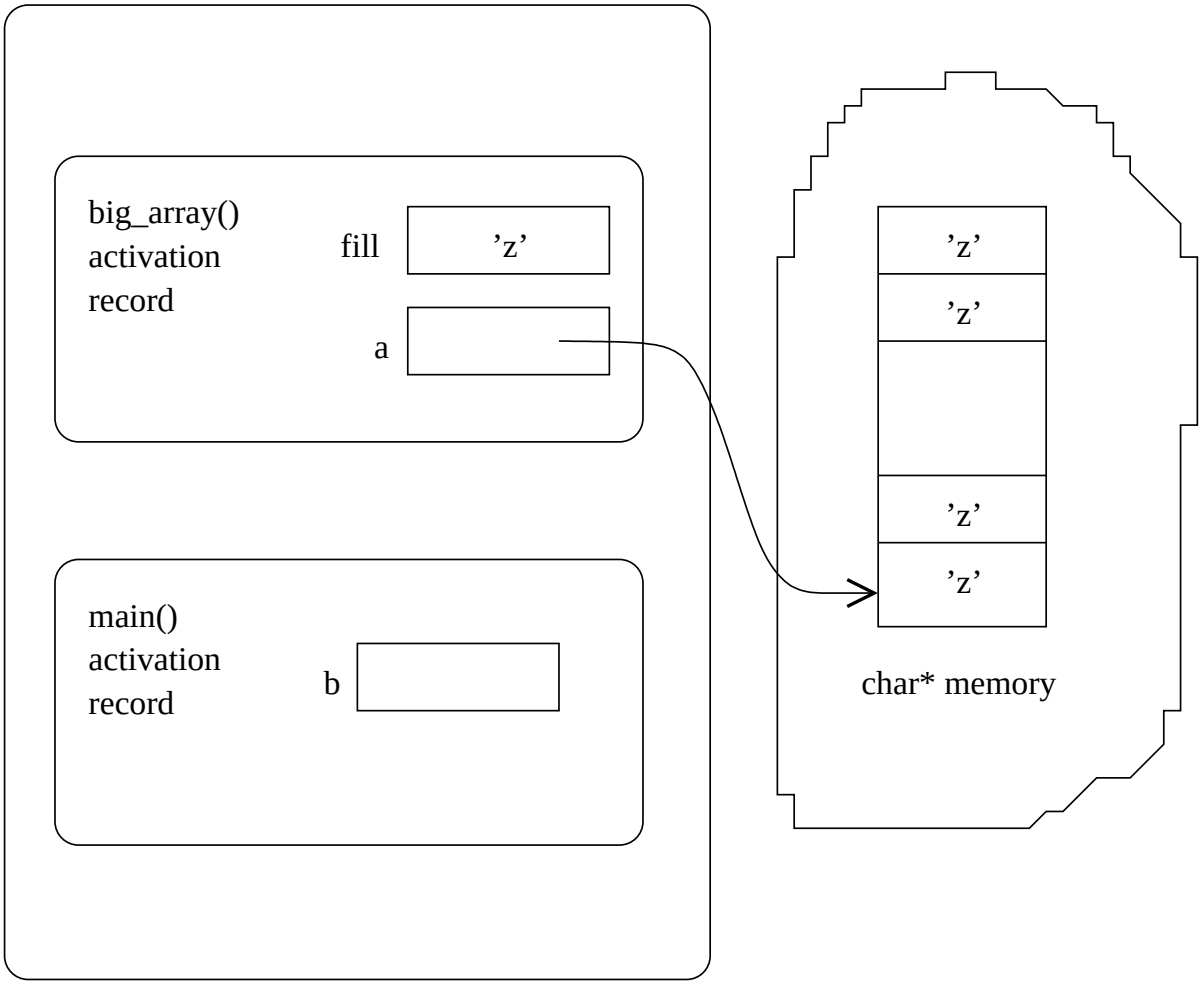
The fixed program

The paradigm: allocate the memory, check to make sure it worked, use it, then free up the memory.

```
char *big_array( char fill ) {
   char *a; int i;
   a = (char *) malloc( sizeof(char) * 1000 );
   /* Don't forget to check if it's null! */
   if( a == NULL ) return a;
   for( i = 0; i < 1000; i++ )
      a[i] = fill;
   return a;
}

void main() {
   char *b;
   b = big_array( 'z' );
   /* do something with b */
   free( b );
}
```

The Stack and the Heap



When `big_array` finishes its activation record is destroyed, but it returns (to `b`) a pointer to the new array in the heap.

You can (and should) think of the heap as a big array of indeterminate type, with static extent.

A more flexible version

The following function lets us make arrays of any size we want!

```
char *big_array( char fill, int size ) {
   char *a;
   a = (char *) malloc( sizeof(char) * size );
   if( a == NULL ) return a;
   for( i = 0; i < size; i++ )
      a[i] = fill;
   return a;
}
```

What good is this malloc thing?

- Suppose you want to write a program which stores names (of people) along with their addresses.

- One way to implement would be to define a struct holding all of this information, and then define an array of structs at the beginning of the program:

```
struct person_struct {
   char name[30];
   char address[60];
};

struct person_struct  database[6000];
```

- Difficulties:

  Need to know ahead of time the maximum size of the database.

  If the maximum size is 6000 and only 50 people stored, much memory is wasted.

A naive implementation

```
struct person_struct {
   char name[30];
   char address[60];
};

struct database_struct {
   struct person_struct people[100];
   int num_people;
};

void add_person( struct database_struct *db,
                 char *pname, char *add ) {
   strcpy( (db->people[db->num_people]).name, pname );
   strcpy( (db->people[db->num_people]).address, add );
   (db->num_people)++;
}

void main() {
   struct database_struct db;
   db.num_people = 0;

   add_person( &db, "O'Neill, Kevin",
      "1234 Street Ave., Ithaca, NY 14850" );
}
```

Linked list implementation

```
#include <stdio.h>

#define NAME_SIZE 30
#define ADD_SIZE  60

struct list_item_struct {
   char name[NAME_SIZE];
   char address[ADD_SIZE];
   struct list_item_struct *next;
};

struct database_struct {
  struct list_item_struct *first;
};

typedef struct list_item_struct list_item;
typedef struct database_struct database;

int initialize_db( database *db ) {
  db->first = NULL;
}
```

Adding list elements

```
int add_to_db( database *db, char *name, char *address ) {
  list_item *new_item_ptr;
  new_item_ptr = (list_item *) malloc(sizeof(list_item));

  if( new_item_ptr == NULL ) return -1;
  if( strlen(name) >= NAME_SIZE ||
      strlen(address) >= ADD_SIZE ) return -1;

  strcpy( new_item_ptr->name, name );
  strcpy( new_item_ptr->address, address );

  new_item_ptr->next = db->first;
  db->first = new_item_ptr;
  return 0;
}
```

Putting it together

```
void print_item( list_item l ) {
  printf( "Name: %s\n", l.name );
  printf( "Address: %s\n\n", l.address );
}

void print_db( database db ) {
  list_item *l = db.first;
  while( l != NULL ) {
     print_item( *l );
     l = l->next;
  }
}

void main() {
  database db;
  initialize_db( &db );
  add_to_db( &db, "Kevin O'Neill", "1234 Street Ave." );
  add_to_db( &db, "Homer Simpson",
                "742 Evergreen Terrace" );
  add_to_db( &db, "Tony Blair",
                "10 Downing Street" );

  print_db( db );
}
```

Removing elements

```
void remove_from_db( database *db, list_item *item ) {
  list_item *l = db->first;

  /* if database is empty */
  if( l == NULL ) return;

  /* if first element is the item */
  if( l == item ) {
    db->first = l->next;
    free( l );
    return;
  }

  /* otherwise, try to find it */
  while( l->next != NULL && l->next != item )
    l = l->next;

  /* we've either found item, or
     come to the end of the list */
  if( l->next == item ) {
    /* skip item in the list, and free memory */
    l->next = item->next;
    free( item );
  }
}
```