Lecture 3: C Memory Management

January 15, 2013
Today...

- Computer Memory
- Pointers/Addresses
- Arrays
- Memory Allocation
Today...

- Computer Memory
- Pointers/Addresses
- Arrays
- Memory Allocation
Heap

- Heap is a chunk of memory that users can use to dynamically allocated memory.

- Lasts until freed, or program exits.
Stack

• Stack contains local variables from functions and related book-keeping data. LIFO structure.
  ▫ Function variables are pushed onto stack when called.
  ▫ Functions variables are popped off stack when return.
Memory Layout

Memory Layout diagram courtesy of bogotobogo.com, and used with permission.
Call Stack

- Example: `DrawSquare` called from `main()`

```java
void DrawSquare(int i){
    int start, end, .... //other local variables
    DrawLine(start, end);
}
```

```java
void DrawLine(int start, int end){
    //local variables
    ...
}
```
Call Stack

• Example:

```c
void DrawSquare(int i){
    int start, end, ...
    DrawLine(start, end);
}
```

```c
void DrawLine(int start, int end){
    //local variables
    ...
}
```
Call Stack

- Example:

  ```
  void DrawSquare(int i){
      int start, end, ...
      DrawLineLine(start, end);
  }

  void DrawLine(int start, int end){
      //local variables
      ...
  }
  ```
Call Stack

• Example:

```c
void DrawSquare(int i){
    int start, end, ...
    DrawLine(start, end);
}
```

```c
void DrawLine(int start, int end){
    //local variables
    ...
}
```

Top of Stack
- main() book-keeping
- int i (DrawSquare arg)

Lower address
- Higher address
Call Stack

- Example:

```c
void DrawSquare(int i){
    int start, end, ...
    DrawLine(start, end);
}

void DrawLine(int start, int end){
    //local variables
    ...
}
```

Diagram:

- **Top of Stack**
  - local variables (start, end)
  - main() book-keeping
  - int i (DrawSquare arg)

- **Higher address**
- **Lower address**
Call Stack

- Example:

```c
void DrawSquare(int i){
    int start, end, ...
    DrawLine(start, end);
}
void DrawLine(int start, int end){
    //local variables
    ...
}
```

Top of Stack:
- start, end (DrawLine args)
- local variables (start, end)
- main() book-keeping
- int i (DrawSquare arg)

Lower address
Call Stack

• Example:

```c
void DrawSquare(int i){
    int start, end, ...
    DrawLine(start, end);
}

void DrawLine(int start, int end){
    //local variables
    ...
}
```

- Top of Stack:
  - DrawSquare book-keeping
  - start, end (DrawLine args)
  - local variables (start, end)
  - main() book-keeping
  - int i (DrawSquare arg)

- Lower address

- Higher address
Call Stack

- Example:

  ```c
  void DrawSquare(int i){
    int start, end, ...
    DrawLine(start, end);
  }
  
  void DrawLine(int start, int end){
    //local variables
    ...
  }
  ```
Call Stack

- Example: `DrawLine` returns

```c
void DrawSquare(int i){
    int start, end, ...
    DrawLine(start, end);
}
```

```c
void DrawLine(int start, int end){
    //local variables
    ...
}
```

Lower address

Top of Stack

- `DrawLine` local vars
- `DrawSquare` book-keeping
- `start, end (DrawLine args)`
- `local variables (start, end)`
- `main() book-keeping`
- `int i (DrawSquare arg)`

Higher address
Call Stack

- Example: DrawLine returns

```c
void DrawSquare(int i){
    int start, end, ...
    DrawLine(start, end);
}
```
```c
void DrawLine(int start, int end){
    //local variables
    ...  
}
```

![Stack diagram]

- **Top of Stack**
  - main() book-keeping
  - int i (DrawSquare arg)
- **Lower address**
  - Local variables (start, end)
Call Stack

- Example: `DrawSquare` returns

```c
void DrawSquare(int i) {
    int start, end, ...
    DrawLine(start, end);
}
```

```c
void DrawLine(int start, int end) {
    //local variables
    ...
}
```

Local variables (start, end)

Main() book-keeping

Int i (DrawSquare arg)

Higher address
Call Stack

• Example: **DrawSquare** returns

```c
void DrawSquare(int i) {
    int start, end, ...
    DrawLine(start, end);
}
```

```c
void DrawLine(int start, int end) {
    //local variables
    ...
}
```
Today...

- Computer Memory
- Pointers/Addresses
- Arrays
- Memory Allocation
Pointers and Addresses

Courtesy of xkcd at http://xkcd.com/138/, available under a CC by-nc license
Addresses

• Each variable represents an address in memory and a value.

• Address: \&\textit{variable} = address of variable

\&X = \text{Address of X}

Image by MIT OpenCourseWare.
A pointer is a variable that “points” to the block of memory that a variable represent.

- Declaration: data_type *pointer_name;
- Example:
  ```c
  char x = 'a';
  char *ptr = &x; // ptr points to a char x
  ```
Pointers

A pointer is a variable that “points” to the block of memory that a variable represent.

- Declaration: data_type *pointer_name;
- Example:

```c
    char x = 'a';
    char *ptr = &x; // ptr points to a char x
```
- Pointers are integer variables themselves, so can have pointer to pointers: char **ptr;
## Data type sizes

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Size*</th>
<th>Range*</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>Character or small integer.</td>
<td>1byte</td>
<td>signed: -128 to 127</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>unsigned: 0 to 255</td>
</tr>
<tr>
<td>short int (short)</td>
<td>Short Integer.</td>
<td>2bytes</td>
<td>signed: -32768 to 32767</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>unsigned: 0 to 65535</td>
</tr>
<tr>
<td>int</td>
<td>Integer.</td>
<td>4bytes</td>
<td>signed: -2147483648 to 2147483647</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>unsigned: 0 to 4294967295</td>
</tr>
<tr>
<td>long int (long)</td>
<td>Long integer.</td>
<td>4bytes</td>
<td>signed: -2147483648 to 2147483647</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>unsigned: 0 to 4294967295</td>
</tr>
<tr>
<td>bool</td>
<td>Boolean value. It can take one of two values: true or false.</td>
<td>1byte</td>
<td>true or false</td>
</tr>
<tr>
<td>float</td>
<td>Floating point number.</td>
<td>4bytes</td>
<td>+/- 3.4e +/- 38 (~7 digits)</td>
</tr>
<tr>
<td>double</td>
<td>Double precision floating point number.</td>
<td>8bytes</td>
<td>+/- 1.7e +/- 308 (~15 digits)</td>
</tr>
<tr>
<td>long double</td>
<td>Long double precision floating point number.</td>
<td>8bytes</td>
<td>+/- 1.7e +/- 308 (~15 digits)</td>
</tr>
</tbody>
</table>
Dereferencing = Using Addresses

• Also uses * symbol with a pointer. Confusing? I know!!!

• Given pointer ptr, to get value at that address, do: *ptr
  ▫ int x = 5;
  int *ptr = &x;
  *ptr = 6; // Access x via ptr, and changes it to 6
  printf("%d", x); // Will print 6 now

• Can use void pointers, just cannot dereference without casting
Pointer Arithmetic

- Can do math on pointers
  - Ex: char* ptr
Pointer Arithmetic

• Can do math on pointers
  – Ex: char* ptr

\[
\text{ptr} + 1
\]
Pointer Arithmetic

• Can do math on pointers
  – Ex: char* ptr

\[ \text{ptr + 1} \]

\[ \text{ptr+i has value: ptr + i \times sizeof(data\_type\_of\_ptr)} \]
Pointer Arithmetic

• Can do math on pointers
  ▫ p1 = p2: sets p1 to the same address as p2
  ▫ Addition/subtraction:
    • p1 + c, p1 - c
  ▫ Increment/decrement:
    • p1++, p1--
Why use pointers? They so confuzin...

- Pass-by-reference rather than value.
  ```c
  void sample_func( char* str_input);
  ```
- Manipulate memory effectively.
- Useful for arrays (next topic).
Today...

- Computer Memory
- Pointers/Addresses
- Arrays
- Memory Allocation
C Arrays (Statically Allocated)

• Arrays are really chunks of memory!

• Declaration:
  ▫ Data_type array_name[num_elements];

• Declare array size, cannot change.
C Arrays (Statically Allocated)

- Can initialize like:
  - `int data[] = {0, 1, 2};` //Compiler figures out size
  - `int data[3] = {0, 1, 2};`
  - `int data[3] = {1};` // data[0] = 1, rest are set to 0
  - `int data[3];` //Here, values in data are still junk
    
    ```c
    data[0] = 0;
    data[1] = 1;
    data[2] = 2;
    ```
Array and Pointers

- Array variables are pointers to the array start!
  - `char *ptr;`
  - `char str[10];`
  - `ptr = str; //ptr now points to array start`
  - `ptr = &str[0]; //Same as above line`

- Array indexing is same as dereferencing after pointer addition.
  - `str[1] = 'a'` is same as `*(str+1) = 'a'`
C-Style Strings

- No string data type in C. Instead, a string is interpreted as a null-terminated char array.
- Null-terminated = last char is null char ‘\0’, not explicitly written
  
  ```
  char str[] = “abc”;
  ```

- String literals use “ “. Compiler converts literals to char array.
C-Style Strings

• Char array can be larger than contained string

```c
char str[5] = “abc”;
```

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>\0</th>
</tr>
</thead>
</table>

• Special chars start with ‘\’:
  ▫ \n, \t, \b, \r: newline, tab, backspace, carriage return
  ▫ \\, ‘’, “”: backslash, apostrophe, quotation mark
String functionalities

- #include <string.h>

- char pointer arguments: char str1[14]
  - char* strcpy(char* dest, const char* source);
    strcpy(str1, "hakuna ");
  - char* strcat(char* dest, const char* source);
    strcat(str1, "matata"); //str1 now has “hakuna matata”

- More in documentation...
Today...

- Computer Memory
- Pointers/Addresses
- Arrays and Strings
- Memory Allocation
Dynamic Allocation

- `#include <stdlib.h>`
- `sizeof` (a C language keyword) returns number of bytes of a data type.
- `malloc/realloc` finds a specified amount of free memory and returns a `void` pointer to it.
  - `char * str = (char *) malloc( 3 * sizeof(char) );`  
    `strcpy(str, “hi”);`
  - `str = (char *) realloc( str , 6 * sizeof(char) );`  
    `strcpy(str, “hello”);`
Dynamic Deallocation

- `#include <stdlib.h>`

- **free** declares the memory pointed to by a pointer variable as free for future use:

```c
char * str = (char *) malloc( 3 * sizeof(char) );
strcpy(str, "hi");
... use str ...
free(str);
```
Dynamically Allocated Arrays

• Allows you to avoid declaring array size at declaration.

• Use malloc to allocate memory for array when needed:
  - `int *dynamic_array;`
  - `dynamic_array = malloc( sizeof(int) * 10 );`
  - `dynamic_array[0]=1; // now points to an array`
Summary

• Memory has stack and heap.

• Pointers and addresses access memory.

• Arrays are really chunks of memory. Strings are null-terminated char arrays.

• C allows user memory allocation. Use malloc, realloc and free.