C: Memory Management and Usage

Instructor: Nick Riasanovsky
Review

- Pointers and arrays are very similar
- Strings are just char pointers/arrays with a null terminator at the end
- Pointer arithmetic moves the pointer by the size of the thing it’s pointing to
- Pointers are the source of many C bugs!
Great Idea #1: Levels of Representation/Interpretation

Higher-Level Language Program (e.g. C) → Compiler
Assembly Language Program (e.g. MIPS) → Assembler
Machine Language Program (MIPS) → Machine Interpretation

Hardware Architecture Description (e.g. block diagrams) → Architecture Implementation
Logic Circuit Description (Circuit Schematic Diagrams)

This week:

```
temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;

lw  $t0, 0($2)
lw  $t1, 4($2)
sw  $t1, 0($2)
sw  $t0, 4($2)
```

```
0000 1001 1100 0110 1010 1111 0101 1000 1010 1111 0101 1000 1100 0110 1100 0110 1010 1111
0101 1000 0000 1001 1100 0110 1010 1111
```
Agenda

• C Memory Layout
  – Stack, Static Data, and Code
• Administrivia
• Dynamic Memory Allocation
  – Heap
• Common Memory Problems
• C Wrap-up: Linked List Example
C Memory Layout

- Program’s \textit{address space} contains 4 regions:
  - \textbf{Stack}: local variables, grows downward
  - \textbf{Heap}: space requested via \texttt{malloc()} and used with pointers; resizes dynamically, grows upward
  - \textbf{Static Data}: global and static variables, does not grow or shrink
  - \textbf{Code}: loaded when program starts, does not change

\textit{OS prevents accesses between stack and heap (via virtual memory)}
Where Do the Variables Go?

- Declared outside a function: Static Data
  - main() is a function
  - Freed when function returns

- Declared inside a function: Stack

- Dynamically allocated: Heap
  - i.e. malloc (we will cover this shortly)
The Stack

• Each stack frame is a contiguous block of memory holding the local variables of a single procedure
• A stack frame includes:
  – Location of caller function
  – Function arguments
  – Space for local variables
• Stack pointer (SP) tells where lowest (current) stack frame is
• When procedure ends, stack pointer is moved back (but data remains (garbage!)); frees memory for future stack frames;
The Stack

• Last In, First Out (LIFO) data structure

```c
int main() {
    a(0);
    return 1; }
void a(int m) {
    b(1); }
void b(int n) {
    c(2);
    d(4); }
void c(int o) {
    printf("c"); }
void d(int p) {
    printf("d"); }
```
Stack Misuse Example

int *getPtr() {
    int y;
    y = 3;
    return &y;
};

int main () {
    int *stackAddr, content;
    stackAddr = getPtr();
    content = *stackAddr;
    printf("%d", content); /* 3 */
    content = *stackAddr;
    printf("%d", content); /* ? */
};

Never return pointers to local variable from functions

Your compiler will warn you about this – don’t ignore such warnings!

overwrites stack frame
C Memory Layout

• Program’s \textit{address space} contains 4 regions:
  
  – \textbf{Stack}: local variables, grows downward
  
  – \textbf{Heap}: space requested via \texttt{malloc}() and used with pointers; resizes dynamically, grows upward
  
  – \textbf{Static Data}: \texttt{global} and \texttt{static} variables, does not grow or shrink
  
  – \textbf{Code}: loaded when program starts, does not change

\textit{OS prevents accesses between stack and heap (via virtual memory)}
Static Data

- Place for variables that persist
  - Data not subject to comings and goings like function calls
  - Examples: **String literals, global variables**
    - String literal example: `char * str = "hi";`
  - Size does not change, but sometimes data can
    - Notably string literals cannot

Code

- Copy of your code goes here
  - C code becomes data too!

- Does not change
Question: Which statement below is FALSE? All statements assume each variable exists.

```c
void funcA() {int x; printf("A");}
void funcB() {
    int y;
    printf("B");
    funcA();
}
void main() {char *s = "s"; funcB();}
```

(A) `&x < &y`
(B) `x` and `y` are in adjacent frames
(C) `&x < s`
(D) `y` is in the 2nd frame from the top of the Stack
Question: Which statement below is FALSE? All statements assume each variable exists.

```c
void funcA() { int x; printf("A"); }  
void funcB() {
    int y;
    printf("B");
    funcA();
}
void main() { char *s = "s"; funcB(); }
```

(A) &x < &y
(B) x and y are in adjacent frames
(C) &x < s
(D) y is in the 2\textsuperscript{nd} frame from the top of the Stack

This is a string literal, and thus stored in STATIC DATA.

Note: We’re talking about *s, not s, i.e. the location where s points!
Addresses

- The size of an address (and thus, the size of a pointer) in bytes depends on architecture (e.g., 32-bit Windows, 64-bit Mac OS)
  - e.g., for 32-bit, have $2^{32}$ possible addresses
- If a machine is **byte-addressed**, then each of its addresses points to a unique **byte**
  - word-addresses = address points to a word
- Question: on a byte-addressed machine, how can we order the bytes of an integer in mem?
  - Answer: it depends
Ketchup on Shelf Analogy

- An integer’s MSBs have more value than its LSBs
- eg:

- Think of two ways of putting the number into memory
Endianness

• Big Endian:
  – Descending numerical significance with ascending memory addresses

• Little Endian
  – Ascending numerical significance with ascending memory addresses

Source: https://en.wikipedia.org/wiki/Endianness
Endianness

• In what order are the bytes within a data type stored in memory? Remember: 28 = 0x 00 00 00 1C

• Big Endian:
  – Descending numerical significance with ascending memory addresses

• Little Endian
  – Ascending numerical significance with ascending memory addresses
Common Mistakes

• Endianness ONLY APPLIES to values that occupy multiple bytes
• Endianness refers to STORAGE IN MEMORY NOT number representation
• Ex: char c = 97
  – c == 0b01100001 in both big and little endian
• Arrays and pointers still have the same order
  – int a[5] = {1, 2, 3, 4, 5} (assume address 0x40)
  – &(a[0]) == 0x40 && a[0] == 1
    • in both big and little endian
Agenda

• C Memory Layout
  – Stack, Static Data, and Code
• Administrivia
• Dynamic Memory Allocation
  – Heap
• Common Memory Problems
• C Wrap-up: Linked List Example
Administrivia

• Meet your fellow classmates! Form study groups and get your questions answered
  – Utilize Piazza, labs, discussions, and OHs
• End of the first week!
  – HW0 and mini-bio due Monday
  – HW1 released. Proj1 will be released tonight!
• Suggestion for weekend:
  – Finish HW, catch up on K&R, start early on the project (actually though!)
• DSP students should contact head TA (Damon)
• Fill out the exam conflicts form on piazza for conflicts
Agenda

• C Memory Layout
  – Stack, Static Data, and Code
• Administrivia
• Dynamic Memory Allocation
  – Heap
• Common Memory Problems
• C Wrap-up: Linked List Example
C Memory Layout

- Program’s **address space** contains 4 regions:
  - **Stack**: local variables, grows downward
  - **Heap**: space requested via `malloc()` and used with pointers; resizes dynamically, grows upward
  - **Static Data**: global and static variables, does not grow or shrink
  - **Code**: loaded when program starts, does not change

*OS prevents accesses between stack and heap (via virtual memory)*
Dynamic Memory Allocation

• Want persisting memory (like static) even when we don’t know size at compile time?
  – e.g. input files, user input
  – Stack won’t work because stack frames aren’t persistent

• Dynamically allocated memory goes on the Heap
  – more permanent than Stack

• Need as much space as possible without interfering with Stack
  – Start at opposite end and grow towards Stack
sizeof()

• If integer sizes are machine dependent, how do we tell?
• **Use `sizeof()` function**
  – Returns size in bytes of variable or data type name
    
    Examples: `int x; sizeof(x); sizeof(int);`

• Acts differently with arrays and structs, which we will cover later
  – Arrays: returns size of whole array
  – Structs: returns size of one instance of struct (sum of sizes of all struct variables + padding)
Allocating Memory

• 3 functions for requesting memory: `malloc()`, `calloc()`, and `realloc()`

• `malloc(n)`
  – Allocates a continuous block of `n bytes` of uninitialized memory (contains garbage!)
  – Returns a pointer to the beginning of the allocated block; NULL indicates failed request (check for this!)
  – Different blocks not necessarily adjacent
Using malloc()

• Almost always used for arrays or structs
• Good practice to use `sizeof()` and typecasting
  
  ```c
  int *p = (int *) malloc(n*sizeof(int));
  ```

  – `sizeof()` makes code more portable

  – `malloc()` returns `void *`; typecast will help you catch coding errors when pointer types don’t match

• Can use array or pointer syntax to access
Releasing Memory

• Release memory on the Heap using `free()`
  – Memory is limited, release when done

• `free(p)`
  – Pass it pointer `p` to beginning of allocated block; releases the whole block
  – `p` must be the address *originally* returned by `m/c/realloc()`, otherwise throws system exception
  – Don’t call `free()` on a block that has already been released or on NULL
  – Make sure you don’t lose the original address
    • eg: `p++` is a **BAD IDEA**; use a separate pointer
Dynamic Memory Example

• Need `#include <stdlib.h>`

```c
typedef struct {
    int x;
    int y;
} point;

point *rect; /* opposite corners = rectangle */
...
if( !(rect=(point *) malloc(2*sizeof(point))) )
{
    printf("\nOut of memory!\n");
    exit(1);
}
...
Do NOT change rect during this time!!!
free(rect);
```
Question: Want output: a[] = {0,1,2} with no errors. Which lines do we need to change?

```c
#define N 3
int *makeArray(int n) {
    int *ar;
    ar = (int *) malloc(n);
    return ar;
}
void main() {
    int i,*a = makeArray(N);
    for(i=0; i<N; i++)
        *a++ = i;
    printf("a[] = {%i,%i,%i}\n",a[0],a[1],a[2]);
    free(a);
}
```

(A) 4, 12  
(B) 5, 12  
(C) 4, 10  
(D) 5, 10
Question: Want output: $a[] = \{0, 1, 2\}$ with no errors. Which lines do we need to change?

```c
#define N 3

int *makeArray(int n) {
    int *ar;
    ar = (int *) malloc(n * sizeof(int));
    return ar;
}

void main() {
    int i,*a = makeArray(N);
    for(i=0; i<N; i++)
        *(a+i) = i;
    printf("a[] = 
        \{%i, %i, %i\}\", a[0], a[1], a[2]);
    free(a);
}
```

(A) 4, 12
(B) 5, 12
(C) 4, 10
(D) 5, 10
Agenda

• C Memory Layout
  – Stack, Static Data, and Code
• Administrivia
• Dynamic Memory Allocation
  – Heap
• Common Memory Problems
• C Wrap-up: Linked List Example
Know Your Memory Errors
(Definitions taken from http://www.hyperdictionary.com)

• Segmentation Fault  More common in 61C
  “An error in which a running Unix program attempts to access memory not allocated to it and terminates with a segmentation violation error and usually a core dump.”

• Bus Error  Less common in 61C
  “A fatal failure in the execution of a machine language instruction resulting from the processor detecting an anomalous condition on its bus. Such conditions include invalid address alignment (accessing a multi-byte number at an odd address), accessing a physical address that does not correspond to any device, or some other device-specific hardware error.”
Common Memory Problems

1) Using uninitialized values
2) Using memory that you don’t own
   – Using NULL or garbage data as a pointer
   – De-allocated stack or heap variable
   – Out of bounds reference to stack or heap array
3) Freeing invalid memory
4) Memory leaks
Using Uninitialized Values

• What is wrong with this code?

```c
void foo(int *p) {
    int j;
    *p = j;  // j is uninitialized (garbage), copied into *p
}

void bar() {
    int i=10;
    foo(&i);
    printf("i = %d\n", i);
}
```
Using Memory You Don’t Own (1)

• What is wrong with this code?

```c
typedef struct node {
    struct node* next;
    int val;
} Node;

int findLastNodeValue(Node* head) {
    while (head->next != NULL)
        head = head->next;
    return head->val;
}
```

What if `head` is NULL?

No warnings! Just Seg Fault that needs finding!
Using Memory You Don’t Own (2)

• What is wrong with this code?

```c
char *append(const char* s1, const char *s2) {
    const int MAXSIZE = 128;
    char result[MAXSIZE];
    int i=0, j=0;
    for (; i<MAXSIZE-1 && j<strlen(s1); i++, j++)
        result[i] = s1[j];
    for (j=0; i<MAXSIZE-1 && j<strlen(s2); i++, j++)
        result[i] = s2[j];
    result[++i] = '\0';
    return result;
}
```

Local array appears on Stack

Pointer to Stack (array) no longer valid once function returns
Using Memory You Don’t Own (3)

• What is wrong with this code?

typedef struct {
    char *name;
    int age;
} Profile;

Profile *person = (Profile *)malloc(sizeof(Profile));
char *name = getName();
person->name = malloc(sizeof(char)*strlen(name));
strcpy(person->name,name);
...
// Do stuff (that isn’t buggy)
free(person);
free(person->name);

Did not allocate space for the null terminator! Want (strlen(name)+1) here.

Accessing memory after you’ve freed it. These statements should be switched.
Using Memory You Haven’t Allocated

• What is wrong with this code?

```c
void StringManipulate() {
    const char *name = "Safety Critical";
    char *str = malloc(sizeof (char) * 10);
    strncpy(str, name, 10);
    str[10] = '\0';
    printf("%s\n", str);
}
```

Write beyond array bounds
Read beyond array bounds
Using Memory You Haven’t Allocated

• What is wrong with this code?

```c
char buffer[1024]; /* global */
int foo(char *str) {
    strcpy(buffer,str);
    ...
}
```

What if more than a kibi characters?

This is called BUFFER OVERRUN or BUFFER OVERFLOW and is a security flaw!!!
Freeing Invalid Memory

• What is wrong with this code?

```c
void FreeMemX() {
    int fnh = 0;
    free(&fnh);  // 1) Free of a Stack variable
}

void FreeMemY() {
    int *fum = malloc(4*sizeof(int));
    free(fum+1); // 2) Free of middle of block
    free(fum);
    free(fum);  // 3) Free of already freed block
    free(fum);
}
```
Memory Leaks

• What is wrong with this code?

```c
int *pi;
void foo() {
    pi = (int*)malloc(8*sizeof(int));
    ...
    free(pi);
}
void main() {
    pi = (int*)malloc(4*sizeof(int));
    foo();
}
```

- Overrode old pointer!
- No way to free those 4*sizeof(int) bytes now
- `foo()` leaks memory
Memory Leaks

• Remember that Java has garbage collection but C doesn’t
• Memory Leak: when you allocate memory but lose the pointer necessary to free it
• **Rule of Thumb:** More *mallocs than frees* probably indicates a memory leak

• Potential memory leak: Changing pointer – do you still have copy to use with *free* later?

```c
plk = (int *)malloc(2*sizeof(int));
...
plk++; // Mem Leak! Typically happens through incrementation or reassignment
```
Debugging Tools

• Runtime analysis tools for finding memory errors
  – Dynamic analysis tool:
    Collects information on memory management while program runs
  – No tool is guaranteed to find ALL memory bugs; this is a very challenging programming language research problem

• You will be introduced to Valgrind in Lab 3

http://valgrind.org
# Meet the Staff

<table>
<thead>
<tr>
<th></th>
<th>Emaan</th>
<th>Sruthi</th>
<th>Sean</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bribe Food</td>
<td>Sushi</td>
<td>Peanut Butter</td>
<td>Fresh Fruit</td>
</tr>
<tr>
<td>Biggest Fear</td>
<td>Elevators</td>
<td>Driving on highway</td>
<td>Spiders with hair</td>
</tr>
<tr>
<td>Unpopular Opinion</td>
<td>Avocados are disgusting</td>
<td>Star Wars prequels are great</td>
<td>Corgis aren't cute</td>
</tr>
<tr>
<td>Cafe Getaway</td>
<td>85°</td>
<td>Philz Coffee</td>
<td>UCha</td>
</tr>
</tbody>
</table>
Agenda

• C Memory Layout
  – Stack, Static Data, and Code
• Administrivia
• Dynamic Memory Allocation
  – Heap
• Common Memory Problems
• C Wrap-up: Linked List Example
Linked List Example

• We want to generate a linked list of strings
  – This example uses structs, pointers, \texttt{malloc()}\texttt{,} \texttt{free()}\texttt{)

• Create a structure for nodes of the list:

  \begin{verbatim}
  struct Node {
    char *value;
    struct Node *next;
  } node;
  \end{verbatim}

The link of the linked list
Adding a Node to the List

- Want to write addNode to support functionality as shown:

```c
char *s1 = "start", *s2 = "middle", *s3 = "end";
struct node *theList = NULL;
theList = addNode(s3, theList);
theList = addNode(s2, theList);
theList = addNode(s1, theList);
```

- In what part of memory are these stored?
- Must be able to handle a NULL input
- If you’re more familiar with Lisp/Scheme, you could name this function `cons` instead.
Adding a Node to the List

• Let's examine the 3rd call ("start"): 

```c
node *addNode(char *s, node *list) {
    node *new = (node *) malloc(sizeof(NodeStruct));
    new->value = (char *) malloc(strlen(s) + 1);
    strcpy(new->value, s);
    new->next = list;
    return new;
}
```

Don’t forget this for the null terminator!
Removing a Node from the List

• Delete/free the first node ("start"): 

```c
node *deleteNode(node *list) {
    node *temp = list->next;
    free(list->value);
    free(list);
    return temp;
}
```

What happens if you do these in the wrong order?
Additional Functionality

• How might you implement the following?
  – Append node to end of a list
  – Delete/free an entire list
  – Join two lists together
  – Reorder a list alphabetically (sort)
Summary

• C Memory Layout
  – **Stack**: local variables (grows & shrinks in LIFO manner)
  – **Static Data**: globals and string literals
  – **Code**: copy of machine code
  – **Heap**: dynamic storage using `malloc` and `free`

  The source of most memory bugs!

• Common Memory Problems

• Last C Lecture!
Bonus Slides!!!11!1!one!!
Memory Management

• Many calls to `malloc()` and `free()` with many different size blocks – where are they placed?
• Want system to be fast with minimal memory overhead
  – Versus automatic garbage collection of Java
• Want to avoid `fragmentation`, the tendency of free space on the heap to get separated into small chunks
Fragmentation Example

1) Block 1: malloc(100)
2) Block 2: malloc(1)
3) Block 1: free(B1)
4) Block 3: malloc(50)
   – What if malloc(101)?
5) Block 4: malloc(60)
Basic Allocation Strategy: K&R

• Section 8.7 offers an implementation of memory management (linked list of free blocks)
  – If you can decipher the code, you’re well-versed in C!
• This is just one of many possible memory management algorithms
  – Just to give you a taste
  – No single best approach for every application
K&R Implementation

• Each block holds its own size and pointer to next block
• `free()` adds block to the list, combines with adjacent free blocks
• `malloc()` searches free list for block large enough to meet request
  – If multiple blocks fit request, which one do we use?
Choosing a Block in malloc()

• **Best-fit:** Choose smallest block that fits request
  – Tries to limit wasted fragmentation space, but takes more time and leaves lots of small blocks

• **First-fit:** Choose first block that is large enough (always starts from beginning)
  – Fast but tends to concentrate small blocks at beginning

• **Next-fit:** Like first-fit, but resume search from where we last left off
  – Fast and does not concentrate small blocks at front
Simplify Code with `typedef`

- It gets annoying to type out `struct ListNode` over and over again
  - Define new variable type for struct:

  Method 1:  Method 2:
  
  ```c
  typedef struct Node ListNode;
  ```

  ```c
  typedef struct Node {
    char *value;
    struct Node *next;
  } ListNode;
  ```

- Can further rename pointers:

  ```c
  typedef ListNode * List;  
  typedef char * String;
  List myLinkedList;
  String value;
  ```