CS 61C:
Great Ideas in Computer Architecture

*Introduction to C, Part III*

Instructors:
Krste Asanovic & Vladimir Stojanovic

http://inst.eecs.Berkeley.edu/~cs61c/sp15
Review, Last Lecture

• Pointers are abstraction of machine memory addresses
• Pointer variables are held in memory, and pointer values are just numbers that can be manipulated by software
• In C, close relationship between array names and pointers
• Pointers know the type of the object they point to (except void *)
• Pointers are powerful but potentially dangerous
Review: C Strings

• String in C is just an array of characters
  
  ```c
  char string[] = "abc";
  ```

• How do you tell how long a string is?
  – Last character is followed by a 0 byte (aka “null terminator”)

  ```c
  int strlen(char s[])
  {
      int n = 0;
      while (s[n] != 0) n++;
      return n;
  }
  ```
Concise strlen()

```c
int strlen(char *s)
{
    char *p = s;
    while (*p++)
        ; /* Null body of while */
    return (p - s - 1);
}
```

What happens if there is no zero character at end of string?
Point past end of array?

• Array size $n$; want to access from 0 to $n-1$, but test for exit by comparing to address one element past the array

```c
int ar[10], *p, *q, sum = 0;
...
p = &ar[0]; q = &ar[10];
while (p != q)
    /* sum = sum + *p; p = p + 1; */
    sum += *p++;
```

– Is this legal?

• C defines that one element past end of array must be a valid address, i.e., not cause an error
Valid Pointer Arithmetic

• Add an integer to a pointer.
• Subtract 2 pointers (in the same array)
• Compare pointers (<, <=, ==, !=, >, >=)
• Compare pointer to NULL (indicates that the pointer points to nothing)

Everything else illegal since makes no sense:
• adding two pointers
• multiplying pointers
• subtract pointer from integer
Arguments in \texttt{main()} 

• To get arguments to the main function, use:
  – \texttt{int main(int argc, char *argv[])}

• What does this mean?
  – \texttt{argc} contains the number of strings on the command line (the executable counts as one, plus one for each argument). Here \texttt{argc} is 2:
    \begin{verbatim}
    unix% sort myFile
    \end{verbatim}
  – \texttt{argv} is a \textit{pointer} to an array containing the arguments as strings
Example

- foo hello 87
- argc = 3 /* number arguments */
- argv[0] = "foo",
  argv[1] = "hello",
  argv[2] = "87"

—Array of pointers to strings
C Memory Management

• How does the C compiler determine where to put all the variables in machine’s memory?
• How to create dynamically sized objects?
• To simplify discussion, we assume one program runs at a time, with access to all of memory.
• Later, we’ll discuss virtual memory, which lets multiple programs all run at same time, each thinking they own all of memory.
C Memory Management

- Program’s address space contains 4 regions:
  - stack: local variables inside functions, grows downward
  - heap: space requested for dynamic data via `malloc()`; resizes dynamically, grows upward
  - static data: variables declared outside functions, does not grow or shrink. Loaded when program starts, can be modified.
  - code: loaded when program starts, does not change
Where are Variables Allocated?

• If declared outside a function, allocated in “static” storage
• If declared inside function, allocated on the “stack” and freed when function returns
  – main() is treated like a function

```c
int myGlobal;
main() {
    int myTemp;
}
```
The Stack

• Every time a function is called, a new frame is allocated on the stack
• Stack frame includes:
  – Return address (who called me?)
  – Arguments
  – Space for local variables
• Stack frames contiguous blocks of memory; stack pointer indicates start of stack frame
• When function ends, stack frame is tossed off the stack; frees memory for future stack frames
• We’ll cover details later for MIPS processor

```c
fooA() { fooB(); }  
fooB() { fooC(); }  
fooC() { fooD(); }  
```

Stack Pointer ➔
Stack Animation

• Last In, First Out (LIFO) data structure

```c
main ()
{ a(0);
}
void a (int m)
{ b(1);
}
void b (int n)
{ c(2);
}
void c (int o)
{ d(3);
}
void d (int p)
{
}
```
Managing the Heap

C supports five functions for heap management:

- **malloc()**: allocate a block of uninitialized memory
- **calloc()**: allocate a block of zeroed memory
- **free()**: free previously allocated block of memory
- **cfree()**: DON’T USE THIS, USE FREE!
- **realloc()**: change size of previously allocated block
  - careful – it might move!
Malloc()

**void *malloc(size_t n):**
- Allocate a block of uninitialized memory
- NOTE: Subsequent calls might not yield blocks in contiguous addresses
- n is an integer, indicating size of allocated memory block in bytes
- size_t is an unsigned integer type big enough to “count” memory bytes
- sizeof returns size of given type in bytes, produces more portable code
- Returns void* pointer to block; NULL return indicates no more memory
- Think of pointer as a handle that describes the allocated block of memory; Additional control information stored in the heap around the allocated block!

**Examples:**

```c
int *ip;
ip = (int *) malloc(sizeof(int));

typedef struct { ... } TreeNode;
TreeNode *tp = (TreeNode *) malloc(sizeof(TreeNode));
```

“Cast” operation, changes type of a variable. Here changes (void *) to (int *)
Managing the Heap

• **void free(void *p):**
  – Releases memory allocated by `malloc()`
  – `p` is pointer containing the address *originally* returned by `malloc()`

```c
int *ip;
ip = (int *) malloc(sizeof(int));
...
free((void*) ip);  /* Can you free(ip) after ip++ ? */
```

```c
typedef struct { ... } TreeNode;
TreeNode *tp = (TreeNode *) malloc(sizeof(TreeNode));
...
f
free((void *) tp);
```

– When insufficient free memory, `malloc()` returns **NULL** pointer; **Check for it!**

```c
if ((ip = (int *) malloc(sizeof(int))) == NULL){
    printf("\nMemory is FULL\n");
    exit(1);  /* Crash and burn! */
}
```

– When you free memory, you must be sure that you pass the **original address** returned from `malloc()` to `free()`; Otherwise, system exception (or worse)!
Using Dynamic Memory

typedef struct node {
    int key;
    struct node *left;
    struct node *right;
} Node;

Node *root = 0;

Node *create_node(int key, Node *left, Node *right)
{
    Node *np;
    if ( (np = (Node*) malloc(sizeof(Node))) == NULL)
    { printf("Memory exhausted!\n"); exit(1); }
    else
    { np->key = key;
        np->left = left;
        np->right = right;
        return np;
    }
}

void insert(int key, Node **tree)
{  
    if ( (*tree) == NULL)
    { (*tree) = create_node(key, NULL, NULL); return; }

    if (key <= (*tree)->key)
        insert(key, &((*tree)->left));
    else
        insert(key, &((*tree)->right));
}
Observations

- Code, Static storage are easy: they never grow or shrink
- Stack space is relatively easy: stack frames are created and destroyed in last-in, first-out (LIFO) order
- *Managing the heap is tricky*: memory can be allocated / deallocated at any time
int x = 2;
int result;

int foo(int n)
{
    int y;
    if (n <= 0) { printf("End case!\n"); return 0; }
    else
    {
        y = n + foo(n-x);
        return y;
    }
}
result = foo(10);

Right after the `printf` executes but before the `return 0`, how many copies of `x` and `y` are there allocated in memory?

A: #x = 1, #y = 1
B: #x = 1, #y = 5
C: #x = 5, #y = 1
D: #x = 1, #y = 6
E: #x = 6, #y = 6
Administrivia

• We can accommodate all those on the wait list, but you have to enroll in a lab section with space!
  – Lab section is important, but you can attend different discussion section
  – Enroll into lab with space, and try to swap with someone later
• HW0 due 11:59:59pm Sunday 2/1
  – Right after the Superbowl...
• Midterm-II now Thursday April 9 in class
How are Malloc/Free implemented?

• Underlying operating system allows `malloc` library to ask for large blocks of memory to use in heap (e.g., using Unix `sbrk()` call)

• C standard `malloc` library creates data structure inside unused portions to track free space
Simple Slow Malloc Implementation

- Initial Empty Heap space from Operating System
- Free Space
  - Malloc library creates linked list of empty blocks (one block initially)
  - First allocation chews up space from start of free space
  - After many mallocs and frees, have potentially long linked list of odd-sized blocks
  - Frees link block back onto linked list – might merge with neighboring free space
Faster malloc implementations

• Keep separate pools of blocks for different sized objects
• “Buddy allocators” always round up to power-of-2 sized chunks to simplify finding correct size and merging neighboring blocks:
Power-of-2 “Buddy Allocator”

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Malloc Implementations

• All provide the same library interface, but can have radically different implementations
• Uses headers at start of allocated blocks and space in unallocated memory to hold malloc’s internal data structures
• Rely on programmer remembering to free with same pointer returned by malloc
• Rely on programmer not messing with internal data structures accidentally!
Common Memory Problems

• Using uninitialized values
• Using memory that you don’t own
  – Deallocated stack or heap variable
  – Out-of-bounds reference to stack or heap array
  – Using NULL or garbage data as a pointer
• Improper use of free/realloc by messing with the pointer handle returned by malloc/calloc
• Memory leaks (you allocated something you forgot to later free)
Using Memory You Don’t Own

• What is wrong with this code?

```c
int *ipr, *ipw;
void ReadMem() {
    int i, j;
    *ipr = malloc(4 * sizeof(int));
    i = *(ipr - 1000); j = *(ipr + 1000);
    free(ipr);
}

void WriteMem() {
    *ipw = malloc(5 * sizeof(int));
    *(ipw - 1000) = 0; *(ipw + 1000) = 0;
    free(ipw);
}
```
Using Memory You Don’t Own

• Using pointers beyond the range that had been malloc’d
  – May look obvious, but what if mem refs had been result of pointer arithmetic that erroneously took them out of the allocated range?

```c
int *ipr, *ipw;
void ReadMem() {
    int i, j;
    *ipr = malloc(4 * sizeof(int));
    i = *(ipr - 1000); j = *(ipr + 1000);
    free(ipr);
}

void WriteMem() {
    *ipw = malloc(5 * sizeof(int));
    *(ipw - 1000) = 0; *(ipw + 1000) = 0;
    free(ipw);
}
```
Faulty Heap Management

• What is wrong with this code?

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

void main() {
    pi = malloc(4*sizeof(int));
    foo();
    ...
}
```
Faulty Heap Management

• Memory leak: *more mallocs than frees*

```c
int *pi;
void foo() {
    pi = malloc(8*sizeof(int));
    /* Allocate memory for pi */
    /* Oops, leaked the old memory pointed to by pi */
    ...
    free(pi); /* foo() is done with pi, so free it */
}

void main() {
    pi = malloc(4*sizeof(int));
    foo(); /* Memory leak: foo leaks it */
    ...
}
```
Faulty Heap Management

• What is wrong with this code?

```c
int *plk = NULL;
void genPLK() {
    plk = malloc(2 * sizeof(int));
    ... ... ...
    plk++;
}
```
Faulty Heap Management

- Potential memory leak – handle has been changed, do you still have copy of it that can correctly be used in a later free?

```c
int *plk = NULL;
void genPLK() {
    plk = malloc(2 * sizeof(int));
    ... ... ...
    plk++; /* Potential leak: pointer variable incremented past beginning of block! */
}
```
In the News, Smallest Chess Program

• Written by Olivier Poudade in x86 assembly code
• Fits in “a 512-byte x86 boot sector for Windows / Linux / OS X / DOS / BSD”

http://olivier.poudade.free.fr/src/BootChess.asm
Faulty Heap Management

• What is wrong with this code?

```c
void FreeMemX() {
    int fnh = 0;
    free(&fnh);
}

void FreeMemY() {
    int *fum = malloc(4 * sizeof(int));
    free(fum+1);
    free(fum);
    free(fum);
    free(fum);
}
```
Faulty Heap Management

• Can’t free non-heap memory; Can’t free memory that hasn’t been allocated

```c
void FreeMemX() {
  int fnh = 0;
  free(&fnh); /* Oops! freeing stack memory */
}

void FreeMemY() {
  int *fum = malloc(4 * sizeof(int));
  free(fum+1);
  /* fum+1 is not a proper handle; points to middle of a block */
  free(fum);
  free(fum);
  free(fum);
  /* Oops! Attempt to free already freed memory */
}
```
Using Memory You Haven’t Allocated

• What is wrong with this code?

```c
void StringManipulate() {
    const char *name = "Safety Critical";
    char *str = malloc(10);
    strncpy(str, name, 10);
    str[10] = '\0';
    printf("%s\n", str);
}
```
Using Memory You Haven’t Allocated

- Reference beyond array bounds

```c
void StringManipulate() {
    const char *name = "Safety Critical";
    char *str = malloc(10);
    strncpy(str, name, 10);
    str[10] = '\0';
    /* Write Beyond Array Bounds */
    printf("%s\n", str);
    /* Read Beyond Array Bounds */
}
```
Using Memory You Don’t Own

• What’s wrong with this code?

```c
char *append(const char* s1, const char *s2) {
    const int MAXSIZE = 128;
    char result[128];
    int i=0, j=0;
    for (j=0; 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;
}
```
Using Memory You Don’t Own

• Beyond stack read/write

```c
char *append(const char* s1, const char *s2) {
    const int MAXSIZE = 128;
    char result[128];
    int i=0, j=0;
    for (j=0; 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;
}
```

- `result` is a local array name – stack memory allocated
- Function returns pointer to stack memory – won’t be valid after function returns
Using Memory You Don’t Own

• What is wrong with this code?

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

int findLastNodeValue(Node* head) {
    while (head->next != NULL) {
        head = head->next;
    }
    return head->val;
}
Using Memory You Don’t Own

• Following a NULL pointer to mem addr 0!

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

int findLastNodeValue(Node* head) {
    while (head->next != NULL) {
        /* What if head happens to be NULL? */
        head = head->next;
    }
    return head->val; /* What if head is NULL? */
}
```
Managing the Heap

- `realloc(p, size)`:
  - Resize a previously allocated block at `p` to a new `size`
  - If `p` is `NULL`, then `realloc` behaves like `malloc`
  - If `size` is 0, then `realloc` behaves like `free`, deallocating the block from the heap
  - Returns new address of the memory block; NOTE: it is likely to have moved!

E.g.: allocate an array of 10 elements, expand to 20 elements later

```c
int *ip;
ip = (int *) malloc(10*sizeof(int));
/* always check for ip == NULL */
...
...
ip = (int *) realloc(ip, 20*sizeof(int));
/* always check for ip == NULL */
/* contents of first 10 elements retained */
...
...
realloc(ip, 0); /* identical to free(ip) */
```
Using Memory You Don’t Own

• What is wrong with this code?

```c
int* init_array(int *ptr, int new_size) {
    ptr = realloc(ptr, new_size*sizeof(int));
    memset(ptr, 0, new_size*sizeof(int));
    return ptr;
}

int* fill_fibonacci(int *fib, int size) {
    int i;
    init_array(fib, size);
    /* fib[0] = 0; */ fib[1] = 1;
    for (i=2; i<size; i++)
        fib[i] = fib[i-1] + fib[i-2];
    return fib;
}
```
Using Memory You Don’t Own

• Improper matched usage of mem handles

```c
int* init_array(int *ptr, int new_size) {
    ptr = realloc(ptr, new_size*sizeof(int));
    memset(ptr, 0, new_size*sizeof(int));
    return ptr;
}
```

```c
int* fill_fibonacci(int *fib, int size) {
    int i;
    /* oops, forgot: fib = */ init_array(fib, size);
    /* fib[0] = 0; */ fib[1] = 1;
    for (i=2; i<size; i++)
        fib[i] = fib[i-1] + fib[i-2];
    return fib;
}
```

Remember: `realloc` may move entire block

What if array is moved to new location?
And In Conclusion, ...

- C has three main memory segments in which to allocate data:
  - Static Data: Variables outside functions
  - Stack: Variables local to function
  - Heap: Objects explicitly malloc-ed/free-d.

- Heap data is biggest source of bugs in C code