MEMORY MANAGEMENT

CS 23200

Big Picture

- ☑ Writing programs on *nix computers
  - ☑ File system commands
  - ☑ Editing, compiling, running, and debugging programs
- C Language
  - ☑ Familiar aspects of C (variables, operators, basic I/O, control flow, functions)
  - ☑ Basics of pointers and structures
  - Memory management
  - Advanced pointers
  - More structures and related constructs
  - Multi-file programs
  - …

Outline

- The stack and heap
- malloc and free
- Heap errors and heap checkers

An Example

```
int main(){
   int lineLength;
   char *pLine = readALine();
   lineLength = strlen(pLine);
   /* CRASH, BOOM, BANG */
```

```
char* readALine(){
   const int bufsize=1000;
   char buffer[bufsize];
   int c, nextIndex = 0;
   do{
      c = getchar();
      if(c == EOF || c == '\n'){
         buffer[nextIndex] = '\0';
         break;
      }
      buffer[nextIndex] = (char)c;
      ++nextIndex;
   } while(nextIndex < bufsize);
   buffer[bufsize-1] = '\0';
   return buffer;
}
```

- The problem: the pointer returned by `readALine` is no longer valid
  - Because of memory organization

(Typical) Memory Organization

- Each program has several sections of memory that serve different purposes
- Two most important sections:
  - (Call or Function) Stack: contains functions' local variables, parameters, etc.
  - Heap: contains memory for which the program explicitly asks (e.g., using `alloc` or `malloc`)

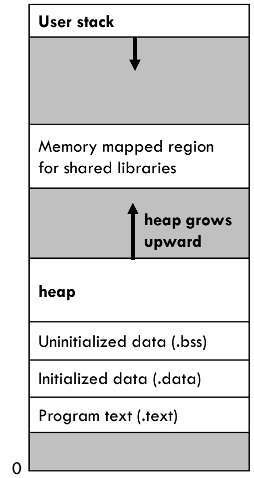


Figure 10.35 from CS:APP (Bryant & O'Hallaron)

Pieces of Data

- Abstract view: any variable or array is just a piece of data
- Every piece of data resides somewhere in memory
  - Each piece: a string of bits
  - Compiler knows the type and interprets bit string as an int, double, etc.

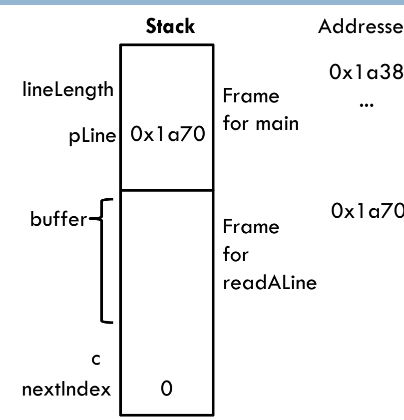
… | 32 | … | -4 | … | 3.14 | … | 3 | 9 | 2 | 3 | … | 7.8 | … | 'h' | 'i' | '\0' | …

Memory Management

- For **EVERY** piece of data in your program, YOU must know if it is on the heap or the stack
- Compiler takes care of the stack
- You take care of the heap
- Either way, you have to know when your piece of data is available and when it is not
- Thus, we need to know how the stack works and how the heap works …

A Stack Example

```
char* readALine(){
   char buffer[1000];
   int c, nextIndex = 0;
...
   return buffer;
}

int main(){
   int lineLength;
   char *pLine = readALine();
   lineLength = strlen(pLine);
   /* INVALID MEMORY ACCESS */
...
```



- The problem: the pointer returned by `readALine` points to memory that is invalid after the return

The Stack

- Items in a program stack are called **frames**
- One frame for every function that is waiting for a return value, and one for the function that is executing
- Compiler generates the code that maintains the stack
- **Local variables** of a function f are in f's **stack frame**
  - Available from when f is called **until f returns**

The Heap to the Rescue

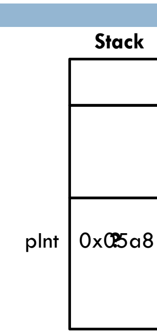
- Want to create something in a function and return a pointer it
- The problem: local variables in functions get destroyed when the function exits
- The solution: the heap
  - Memory that the programmer explicitly **requests** and **releases**
  - Memory is destroyed when the programmer decides

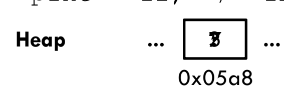
Outline

- The stack and heap
- malloc and free
- Heap errors and heap checkers

The Heap: An Example

```
int *pInt;
/* pInt is a local variable: on the stack */

pInt = malloc(sizeof(int));
/* malloc returns address
   of space on the heap */

*pInt = 5;
pInt[0] = 7;

/* When done with chunk on heap, call free */
free(pInt);

*pInt = 12;  /* INVALID */
```





Another malloc Example

- Can use malloc to create arrays

```
int arraySize; /* size unknown at compile time */
...
{
   int i;
   int *arr = malloc(sizeof(int) * arraySize);

   /* initialize the array */
   for(i=0; i<arraySize; i++){
      arr[i] = 0;
   }
   ...

   free(arr);
}
```

Using malloc to Return Arrays

```
/* WRONG WAY: buffer points at the STACK */
char* readALine(){
   char buffer[1000];
   int c, nextIndex = 0;
...
   return buffer;
}

/* CORRECT WAY: buffer points at the HEAP */
char* readALine(){
   const int maxLineLength = 1000;
   int c, nextIndex = 0;
   char* buffer = malloc(sizeof(char) * maxLineLength);
...
   return buffer;
}
/* caller must free the return value! */
```

Memory Management in a Nutshell

- Stack
  - Holds local variables, which are invalid after function return
- Heap
  - Several chunks of space, each allocated by malloc
    - malloc(x) returns pointer to x bytes of space on the heap
      - Returns NULL if space is not available
    - sizeof(type) tells how many bytes one "type" item needs
  - Each chunk remains valid until program calls "free" with that chunk's address
  - Must free each chunk exactly once

Exercise: Validity of Function Return Values

Using malloc to Return Structures

```
struct listnode {
   int value;
   struct listnode *next;
   struct listnode *prev;
};

/* caller must free the return value */
struct listnode* createNode(const int value){
   struct listnode *pNode
      = malloc(sizeof(struct listnode));

   if(pNode == NULL)
      return NULL;

   pNode->value = value;
   pNode->next = NULL;
   pNode->prev = NULL;

   return pNode;
}
```

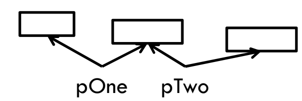
Using malloc to Return Structures

```
struct listnode {
   int value;
   struct listnode *next;
   struct listnode *prev;
};

/* caller must free the return value */
struct listnode* createNode(const int value);
int main(){
   struct listnode *pList = createNode(1);
   if(pList == NULL){
      return -1;
   }
   pList->next = createNode(2);
   if(pList->next == NULL){ ... }
   pList->next->prev = pList;
   ...
   if(pList->next != NULL){
      free(pList->next);
   }
   free(pList);
   return 0;
```

When to Use malloc/free

- To create some data in a function that needs to persist after the function returns
- To copy some data that is already on the heap
- For large amounts of data
  - The limit on stack size is typically smaller than the limit on heap size
- Do not use it for single integers, characters, etc.
  - Using the heap is less efficient than the stack for such things

Every successful call to **malloc adds a new chunk on the heap**
Every successful call to **free removes a chunk from the heap**
Pointer assignments do not change the heap



- Every successful call to malloc returns an address
- That address is stored in a list
  - Maintained by the standard library
- free every address in the list exactly once
  - Doesn't have to be the same pointer variable that came from malloc, just the same address

```
int *pOne, *pTwo;

pOne = malloc(sizeof(int) * 88);
pTwo = pOne;

pOne = malloc(sizeof(int) * 14);

free(pTwo);

pOne = pTwo;
/* chunk is now lost! */
pTwo = malloc(sizeof(int) * 73);

free(pOne);
/* CRASH: double-free (above) */
free(pTwo);
```

Three Important Rules (Repeated)

- Every successful call to **malloc adds a new chunk on the heap**
- Every successful call to **free removes a chunk from the heap**
- **Pointer assignments do not change the heap**

Memory Management Practice

Outline

- The stack and heap
- malloc and free
- Heap errors and heap checkers

No, really. USE CAUTION!!!

- Memory management bugs are very easily introduced into C code, even for experienced programmers
  - One of the drawbacks of the C language
- Reading or writing memory in a place you are not intending can cause…
  - Immediate crashes
  - Delayed crashes
  - Unexpected results
  - No apparent effects…
    - … until you change unrelated code and the program crashes

Heap Checkers

- ALWAYS run your program through a heap checker
- Will catch things that otherwise go unnoticed
  - Array out of bounds access
  - Uninitialized variables
  - Memory leaks (i.e., not freeing a malloc-ed chunk)
  - Variety of other memory errors

Some Heap Checkers

- **Memcheck (part of Valgrind)**
  - Quick start guide at http://valgrind.org/docs/manual/quick-start.html#quick-start.mcrun
  - Does not require recompiling, but uses **much** more memory and time
- Dmalloc
  - Available at http://dmalloc.com/
  - Requires recompiling the program, but does not need as much memory as Memcheck

valgrind examples

valgrind --leak-check=yes ./myprog arg1 arg2

https://www.youtube.com/watch?v=fvTsFjDuag8

Summary

- The stack and heap
  - Know where every piece of your data resides!
- malloc and free
  - malloc adds a chunk
  - free removes a chunk
  - Pointer assignment does not alter the heap
- Heap errors and heap checkers
  - Run with valgrind to diagnose errors

Example: Binary Tree

- Design structure
- Write insert and destroy functions