

Lecture 6 – C Memory Management



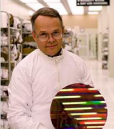
2007-01-29

There is one handout today at the front and back of the room!

Lecturer SOE Dan Garcia

www.cs.berkeley.edu/~ddgarcia

Intel: "Moore's law lives!" ⇒ Intel says it will be replacing their silicon dioxide insulators (which leak current as process size shrinks, resulting in lower battery life, more heat) with a Hafnium-based insulators. This is a big breakthrough.



www.nytimes.com/2007/01/27/technology/27chip.html

CS61C L06 C Memory Management (1)

Garcia, Spring 2007 © UC

Review

- Use handles to change pointers
- Create abstractions (and your own data structures) with structures
- Dynamically allocated heap memory must be manually deallocated in C.
 - Use `malloc()` and `free()` to allocate and de-allocate persistent storage.



CS61C L06 C Memory Management (2)

Garcia, Spring 2007 © UC

Don't forget the globals!

- Remember:
 - Structure declaration does not allocate memory
 - Variable declaration does allocate memory
- So far we have talked about several different ways to allocate memory for data:
 1. Declaration of a local variable

```
int i; struct Node list; char *string; int ar[n];
```
 2. "Dynamic" allocation at runtime by calling allocation function (`alloc`).

```
ptr = (struct Node *) malloc(sizeof(struct Node)*n);
```
 3. Data declared outside of any procedure (i.e., before `main`).
 - Similar to #1 above, but has "global" scope.

```
int myGlobal;
main() {
}
```



CS61C L06 C Memory Management (3)

Garcia, Spring 2007 © UC

C Memory Management

- C has 3 pools of memory
 - **Static storage**: global variable storage, basically permanent, entire program run
 - **The Stack**: local variable storage, parameters, return address (location of "activation records" in Java or "stack frame" in C)
 - **The Heap** (dynamic `malloc` storage): data lives until deallocated by programmer
- C requires knowing where objects are in memory, otherwise things don't work as expected
 - Java hides location of objects

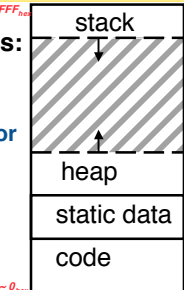


CS61C L06 C Memory Management (4)

Garcia, Spring 2007 © UC

Normal C Memory Management

- A program's **address space** contains 4 regions:
 - **stack**: local variables, grows downward
 - **heap**: space requested for pointers via `malloc()`; resizes dynamically, grows upward
 - **static data**: variables declared outside `main`, does not grow or shrink
 - **code**: loaded when program starts, does not change



For now, OS somehow prevents accesses between stack and heap (gray hash lines). Wait for virtual memory



CS61C L06 C Memory Management (5)

Garcia, Spring 2007 © UC

Where are variables allocated?

- If declared **outside** a procedure, allocated in "static" storage
- If declared **inside** procedure, allocated on the "stack" and freed when procedure returns.
 - NB: `main()` is a procedure

```
int myGlobal;
main() {
    int myTemp;
}
```

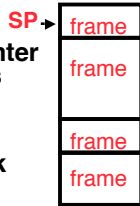


CS61C L06 C Memory Management (6)

Garcia, Spring 2007 © UC

The Stack

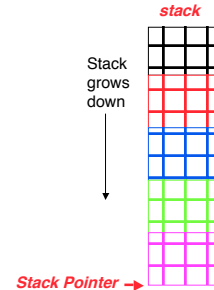
- Stack frame includes:
 - Return "instruction" address
 - Parameters
 - Space for other local variables
- Stack frames contiguous blocks of memory; stack pointer tells where top stack frame is
- When procedure ends, stack frame is tossed off the stack; frees memory for future stack frames



Stack

- Last In, First Out (LIFO) data structure

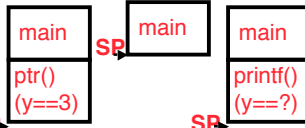
```
main ()
{ a(0);
}
void a (int m)
{ b(1);
}
void b (int n)
{ c(2);
}
void c (int o)
{ d(3);
}
void d (int p)
{
}
```



Who cares about stack management?

- Pointers in C allow access to deallocated memory, leading to hard-to-find bugs!

```
int *ptr () {
  int y;
  y = 3;
  return &y;
};
main () {
  int *stackAddr, content;
  stackAddr = ptr();
  content = *stackAddr;
  printf("%d", content); /* 3 */
  content = *stackAddr;
  printf("%d", content); /*13451514 */
}
```



The Heap (Dynamic memory)

- Large pool of memory, **not** allocated in contiguous order
 - back-to-back requests for heap memory could result blocks very far apart
 - where Java `new` command allocates memory
- In C, specify number of **bytes** of memory explicitly to allocate item

```
int *ptr;
ptr = (int *) malloc(sizeof(int));
/* malloc returns type (void *),
so need to cast to right type */
```

- `malloc()`: Allocates raw, uninitialized memory from heap



Memory Management

- How do we manage memory?
- Code, Static storage are easy: they never grow or shrink
- Stack space is also 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



Heap Management Requirements

- Want `malloc()` and `free()` to run quickly.
- Want minimal memory overhead
- Want to avoid **fragmentation*** – when most of our free memory is in many small chunks
 - In this case, we might have many free bytes but not be able to satisfy a large request since the free bytes are not contiguous in memory.

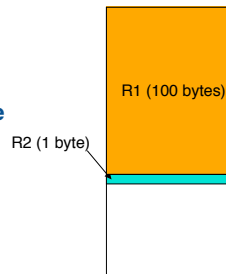


* This is technically called *external fragmentation*

Heap Management

• An example

- Request R1 for 100 bytes
- Request R2 for 1 byte
- Memory from R1 is freed
- Request R3 for 50 bytes



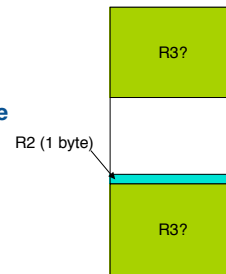
CS61C L06 C Memory Management (13)

Garcia, Spring 2007 © UCB

Heap Management

• An example

- Request R1 for 100 bytes
- Request R2 for 1 byte
- Memory from R1 is freed
- Request R3 for 50 bytes



CS61C L06 C Memory Management (14)

Garcia, Spring 2007 © UCB

K&R Malloc/Free Implementation

• From Section 8.7 of K&R

- Code in the book uses some C language features we haven't discussed and is written in a very terse style, don't worry if you can't decipher the code
- Each block of memory is preceded by a header that has two fields: **size of the block** and a **pointer to the next block**
- All **free blocks** are kept in a circular linked list, the pointer field is unused in an allocated block



CS61C L06 C Memory Management (15)

Garcia, Spring 2007 © UCB

K&R Implementation

- `malloc()` searches the free list for a block that is big enough. If none is found, more memory is requested from the operating system. If what it gets can't satisfy the request, it fails.
- `free()` checks if the blocks adjacent to the freed block are also free
 - If so, adjacent free blocks are merged (**coalesced**) into a single, larger free block
 - Otherwise, the freed block is just added to the free list



CS61C L06 C Memory Management (16)

Garcia, Spring 2007 © UCB

Choosing a block in `malloc()`

- If there are multiple free blocks of memory that are big enough for some request, how do we choose which one to use?
 - **best-fit**: choose the smallest block that is big enough for the request
 - **first-fit**: choose the first block we see that is big enough
 - **next-fit**: like first-fit but remember where we finished searching and resume searching from there



CS61C L06 C Memory Management (17)

Garcia, Spring 2007 © UCB

Peer Instruction

Which are guaranteed to print out 5?

```
I: main() {
    int *a-ptr; *a-ptr = 5; printf("%d", *a-ptr); }
```

```
II: main() {
    int *p, a = 5;
    p = &a; ...
    /* code; a & p NEVER on LHS of = */
    printf("%d", a); }
```

```
III: main() {
    int *ptr;
    ptr = (int *) malloc (sizeof(int));
    *ptr = 5;
    printf("%d", *ptr); }
```

	I	II	III
0:	-	-	-
1:	-	-	YES
2:	-	YES	-
3:	-	YES	YES
4:	YES	-	-
5:	YES	-	YES
6:	YES	YES	-
7:	YES	YES	YES

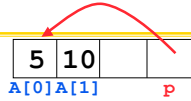


CS61C L06 C Memory Management (18)

Garcia, Spring 2007 © UCB

Peer Instruction

```
int main(void){
int A[] = {5,10};
int *p = A;
```



```
printf(“%u %d %d %d\n”, p, *p, A[0], A[1]);
p = p + 1;
printf(“%u %d %d %d\n”, p, *p, A[0], A[1]);
*p = *p + 1;
printf(“%u %d %d %d\n”, p, *p, A[0], A[1]);
}
```

If the first printf outputs `100 5 5 10`, what will the other two printf output?

- 1: 101 10 5 10 then 101 11 5 11
- 2: 104 10 5 10 then 104 11 5 11
- 3: 101 <other> 5 10 then 101 <3-others>
- 4: 104 <other> 5 10 then 104 <3-others>
- 5: One of the two printf causes an ERROR
- 6: I surrender!



CS61C L06 C Memory Management (19)

Garcia, Spring 2007 © UCB

Peer Instruction – Pros and Cons of fits

- A. The con of first-fit is that it results in many **small blocks** at the beginning of the free list
- B. The con of next-fit is it is **slower** than first-fit, since it takes longer in steady state to find a match
- C. The con of best-fit is that it **leaves lots of tiny blocks**

ABC
0: FFF
1: FTF
2: FTT
3: FTT
4: TFF
5: TFF
6: FTT
7: FTT



CS61C L06 C Memory Management (20)

Garcia, Spring 2007 © UCB

And in conclusion...

- C has 3 pools of memory
 - **Static storage**: global variable storage, basically permanent, entire program run
 - **The Stack**: local variable storage, parameters, return address
 - **The Heap** (dynamic storage): `malloc()` grabs space from here, `free()` returns it.
- `malloc()` handles free space with freelist. Three different ways to find free space when given a request:
 - First fit (find first one that's free)
 - Next fit (same as first, but remembers where left off)
 - Best fit (finds most "snug" free space)



CS61C L06 C Memory Management (21)

Garcia, Spring 2007 © UCB

Bonus slides

- These are extra slides that used to be included in lecture notes, but have been moved to this, the "bonus" area to serve as a supplement.
- The slides will appear in the order they would have in the normal presentation

Bonus

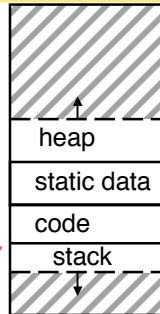


CS61C L06 C Memory Management (22)

Garcia, Spring 2007 © UCB

Intel 80x86 C Memory Management

- A C program's 80x86 address space :
 - **heap**: space requested for pointers via `malloc()`; resizes dynamically, grows upward
 - **static data**: variables declared outside main, does not grow or shrink
 - **code**: loaded when program starts, does not change
 - **stack**: local variables, grows downward



CS61C L06 C Memory Management (23)

Garcia, Spring 2007 © UCB

Tradeoffs of allocation policies

- **Best-fit**: Tries to limit fragmentation but at the cost of time (must examine all free blocks for each `malloc()`). Leaves lots of small blocks (why?)
- **First-fit**: Quicker than best-fit (why?) but potentially more fragmentation. Tends to concentrate small blocks at the beginning of the free list (why?)
- **Next-fit**: Does not concentrate small blocks at front like first-fit, should be faster as a result.



CS61C L06 C Memory Management (24)

Garcia, Spring 2007 © UCB