

## Lecture 6 – C Memory Management

2010-02-01



Lecturer SOE Dan Garcia

[www.cs.berkeley.edu/~ddgarcia](http://www.cs.berkeley.edu/~ddgarcia)

Tape ogre awakens! ⇒  
IBM Zurich has made  
a new tape material that can store 29.5  
gigabits/in<sup>2</sup>, i.e., a cartridge that can  
hold 35 terabytes of data, more than 40  
times the current capacity.



[www.technologyreview.com/computing/24406](http://www.technologyreview.com/computing/24406)

CS61C L06 C Memory Management (1)

Garcia, Spring 2010 © UCB

## 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 2010 © UCB

## 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];
```
  3. "Dynamic" allocation at runtime by calling allocation function (`alloc`).  

```
ptr = (struct Node *) malloc(sizeof(struct Node)*n);
```
- One more possibility exists...
  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 2010 © UCB

## 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



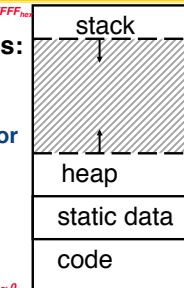
CS61C L06 C Memory Management (4)

Garcia, Spring 2010 © UCB

## 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 2010 © UCB

## 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;  
}
```

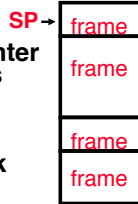


CS61C L06 C Memory Management (6)

Garcia, Spring 2010 © UCB

## The Stack

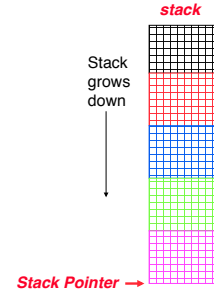
- Stack frame includes:
  - Return "instruction" address
  - Parameters
  - Space for other local variables
- Stack frames contiguous blocks of memory; stack pointer tells where top stack frame is
- When procedure ends, stack frame is tossed off the stack; frees memory for future stack frames



## Stack

- Last In, First Out (LIFO) data structure

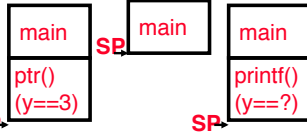
```
main ()
{ a(0);
}
void a (int m)
{ b(1);
}
void b (int n)
{ c(2);
}
void c (int o)
{ d(3);
}
void d (int p)
{
}
```



## 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.

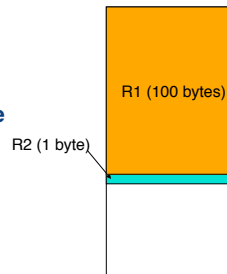


\* This is technically called *external fragmentation*

## Heap Management

### • An example

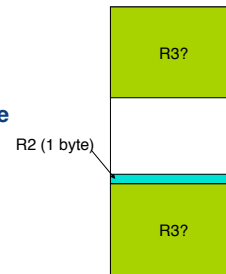
- Request R1 for 100 bytes
- Request R2 for 1 byte
- Memory from R1 is freed
- Request R3 for 50 bytes



## Heap Management

### • An example

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



## 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



## 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



## 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



## Peer Instruction – Pros and Cons of fits

- 1) **first-fit** results in many **small blocks** at the beginning of the free list
- 2) **next-fit** is **slower than first-fit**, since it takes longer in steady state to find a match
- 3) **best-fit** leaves lots of **tiny blocks**

	1	2	3
a)	FF	FF	FF
b)	FF	FF	FF
c)	TF	FF	FF
d)	TF	FF	FF
e)	TF	TF	FF



### 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)



### Bonus slides

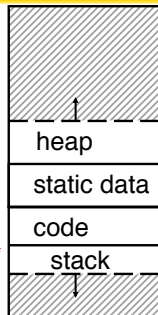
- These are extra slides that used to be included in lecture notes, but have been moved to this, the "bonus" area to serve as a supplement.
- The slides will appear in the order they would have in the normal presentation

# Bonus



### 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



### 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.

