
C Programming Language Review and Dissection IV

Lecture 6



Today

Dynamic Memory Allocation

Linked Lists

Dynamic Memory Allocation

Why

How it's used in C

Example with linked lists

Dangers

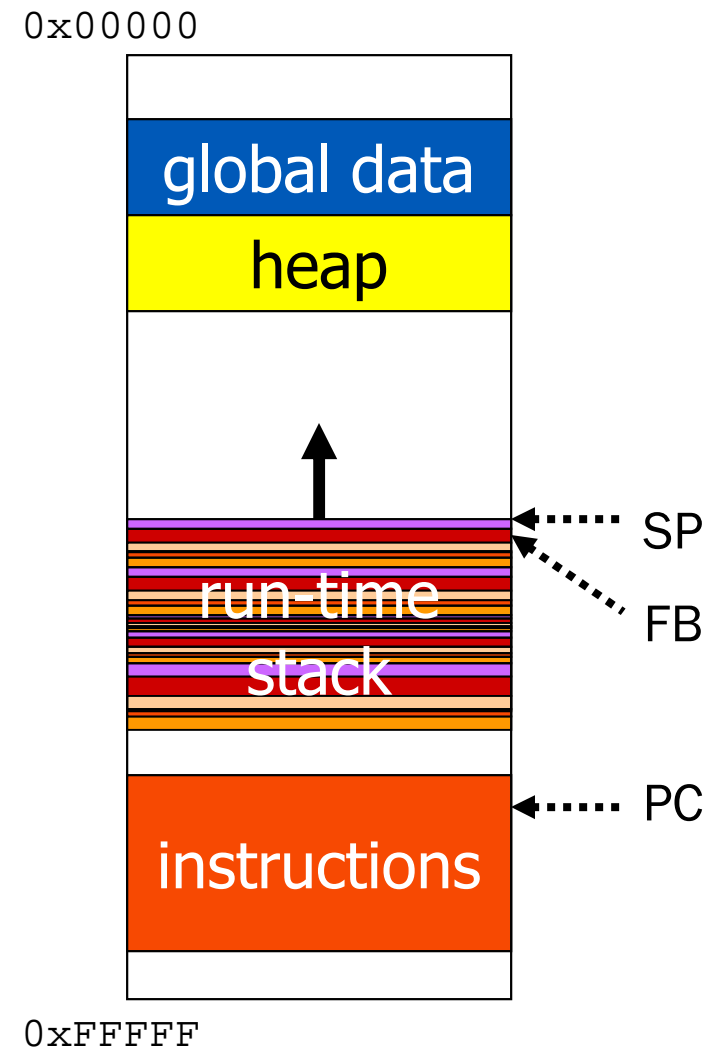
Dynamic Memory Management

In addition to storing variables in global data section and run-time stack, we can *dynamically* allocate memory from a *heap* of free space (Patt & Patel 19.4)

Allows more flexible programming

- Can allocate memory as needed, deallocate when done

Function interfaces in `stdlib.h` (C Standard Library)



Dynamic Memory Allocation in C

Why?

- Some systems have changing memory requirements, and stack variables (automatic) aren't adequate
- Example: Voice recorder needs to store recordings of different lengths. Allocating the same size buffer for each is inefficient

How?

- Allocate *nbytes* of memory and return a start pointer
 - `void * malloc (size_t nbytes);`
- Allocate *nelements***size* bytes of memory and return a start pointer
 - `void * calloc (size_t nelements, size_t size);`
- Change the size of a block of already-allocated memory
 - `void * realloc (void * pointer, size_t size);`
- Free a block of allocated memory
 - `void free (void * pointer);`

Using Dynamic Memory Management

Request space for one or more new variables

- Request pointer to space for one element

```
int * j, *k;
```

```
j = (int *) malloc (sizeof(int));
```

```
*j = 37;
```

- Request pointer to space for array of elements and initialize to zero

```
k = (int *) calloc(num_elements, sizeof(int));
```

```
k[0] = 55;
```

```
k[1] = 31;
```

- These return NULL if there isn't enough space

- Program has to deal with failure -- embedded program probably shouldn't just quit or reset....

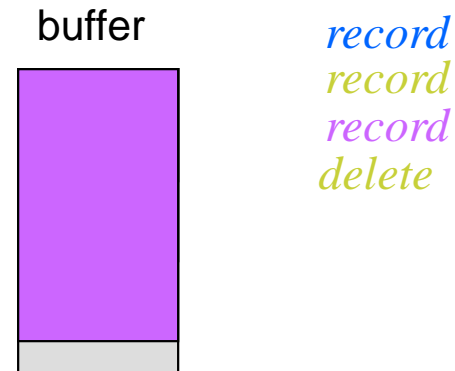
Free up space when done using variables

```
free(k);
```

Example Application: Voice Recorder

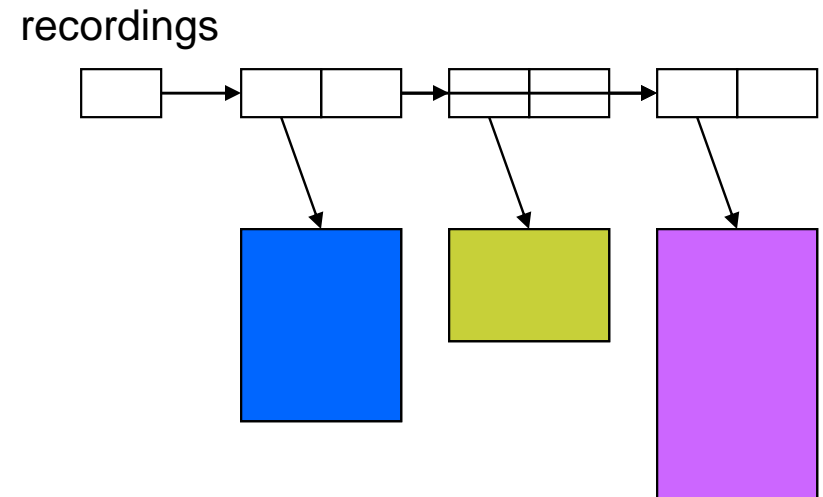
Recording

- While *record* switch is pressed
 - sample microphone
 - store in temporary RAM buffer
- When *record* switch is released
 - copy audio to a permanent buffer
 - add to end of list of recordings



Playback and skipping

- *forward* switch: skip forward over one recording, wrap around at end
- *play* switch: play the current recording
- *delete* switch: delete the current recording



Data Structure: linked list of recordings

Data Structure Detail: Linked List

Each list element is defined as a structure with fields

- AudioSize: Number of bytes
- AudioData: ...
- Next: Pointer to next list element

```
typedef struct {  
    unsigned AudioSize;  
    char * AudioData;  
    struct List_T * Next;  
} List_T;
```

Code for Voice Recorder main

```
unsigned char buffer[MAX_BUFFER_SIZE];
struct List_T * recordings = NULL, * cur_recording = NULL;

void main(void) {
    while (1) {
        while (NO_SWITCHES_PRESSED)
            ;
        if (RECORD)
            handle_record();
        else if (PLAY)
            handle_play();
        else if (FORWARD)
            handle_forward();
        else if (DELETE)
            handle_delete();
    }
}
```

Code for handle_forward

```
void handle_forward(void) {  
    if (cur_recording)  
        cur_recording = cur_recording->Next;  
    if (!cur_recording)  
        cur_recording = recordings;  
}
```

Code for handle_record

```
void handle_record(void) {
    unsigned i, size;
    unsigned char * new_recording;
    struct List_T * new_list_entry;
    i = 0;
    while (RECORD)
        buffer[i++] = sample_audio();
    size = i;
    new_recording = (unsigned char *) malloc (size);
    for (i=0; i<size; i++) /* could also use memcpy() */
        new_recording[i] = buffer[i];
    new_list_entry = (List_T *) malloc ( sizeof(List_T) );
    new_list_entry->AudioData = new_recording;
    new_list_entry->AudioSize = size;
    new_list_entry->Next = NULL;
    recordings = Append(recordings, new_list_entry);
}
```

Code for handle_delete

```
void handle_delete(void) {
    List_T * cur = recordings;
    if (cur == cur_recording)
        recordings = recordings->Next;
    else {
        while (cur->Next != cur_recording)
            cur = cur->Next;
        /* cur now points to previous list entry */
        cur->Next = cur_recording->Next;
    }
    free(cur_recording->AudioData);
    free(cur_recording);
}
```

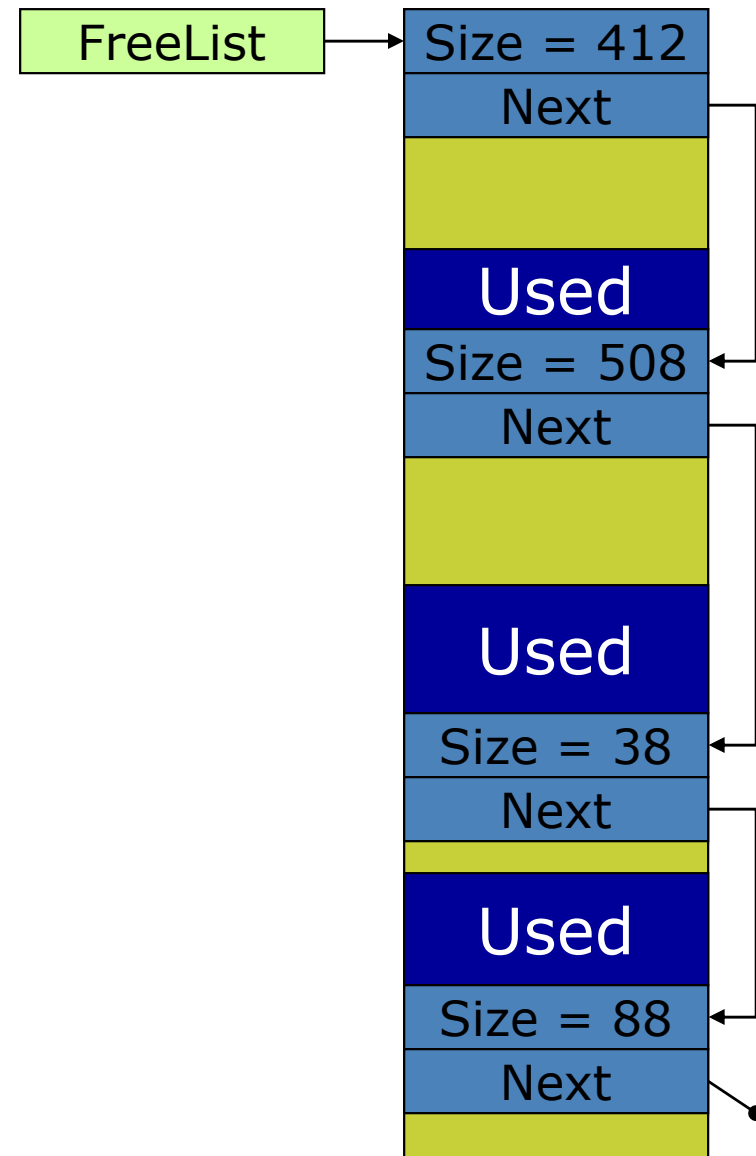
Allocation Data Structures

Keep free memory in
sorted list of free blocks

```
typedef struct hdr {  
    struct hdr * next;  
    unsigned int size;  
};  
hdr * FreeList;
```

*Assume hdr takes no
space for examples*

More details in “Memory
Allocation in C,” Leslie
Alridge, *Embedded
Systems Programming*,
August 1989



Allocation Operations

To allocate memory

- find first block of size \geq requested_size
- modify list to indicate space isn't free
 - if sizes match exactly, remove free block from list
 - else split memory
 - reduce size field by requested_size, keeping first part of block in free space
 - allocate memory in second part of block
 - return pointer to newly allocated block

To free memory depends on block's memory location

- If before first free block, prepend it at head of free list
- If between free list entries, insert in list
- If after last free block, append it at tail of free list

Freed memory block may be adjacent to other free blocks. If so, merge contiguous blocks

Dangers of Dynamic Memory Allocation

Memory leaks waste memory

- Never freeing blocks which are no longer needed. User's responsibility.

May accidentally use freed memory

- User's responsibility.

Allocation speed varies

- Linked list must be searched for a block which is large enough
- Bad for a real-time system, as worst case may be large.

Fragmentation

- Over time free memory is likely to be broken into smaller and smaller fragments.
- Eventually there won't be a block large enough for an allocation request, even though there is enough total memory free

Heap and Fragmentation

Problem:

- malloc/calloc/free use a *heap* of memory; essentially a list of blocks of empty and used memory
- Repeated allocation/free cycles with differently sized allocation units leads to *fragmentation*
 - Although there may be enough memory free, it may be fragmented into pieces too small to meet request

Solutions (none optimal):

- Always allocate a fixed size memory element
- Use multiple heaps, each with a fixed element size