Dynamic Memory Allocation
(and Multi-Dimensional Arrays)

Professor Hugh C. Lauer
CS-2303, System Programming Concepts

(Slides include materials from *The C Programming Language*, 2nd edition, by Kernighan and Ritchie, *Absolute C++*, by Walter Savitch, *The C++ Programming Language,* Special Edition, by Bjarne Stroustrup, and from *C: How to Program*, 5th and 6th editions, by Deitel and Deitel)

Problem

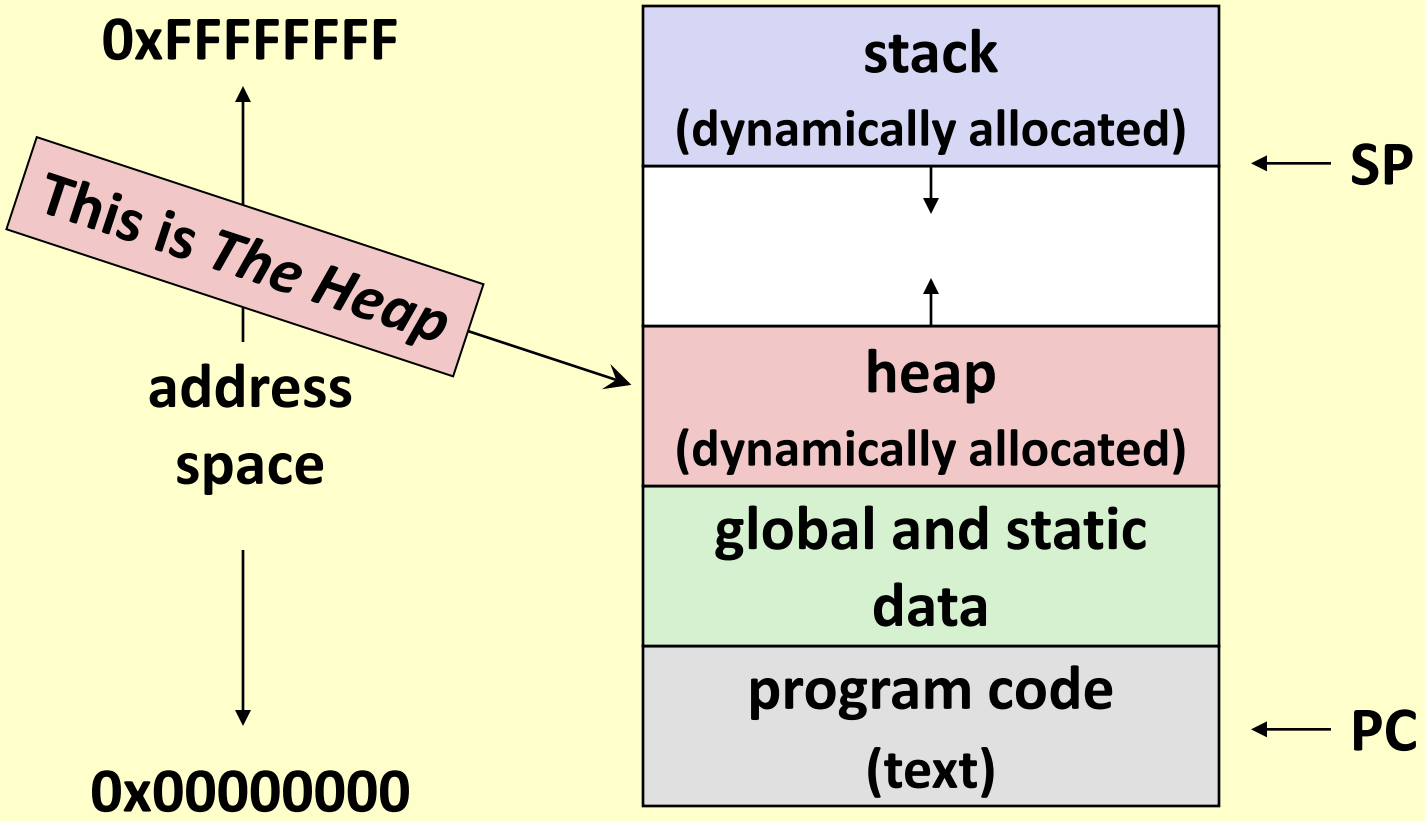
- **What do you do if:–**
  - … the size of an array is not known until run time?

    OR
  - … a function must return an array that it creates?
- **How can we manipulate variable-length arrays and pass them around our programs?**
- **Answer:–**
  - Use dynamically allocated storage in *The Heap!*

Definition — The Heap

- **A region of memory provided by most operating systems for allocating storage *not* in *Last in, First out* discipline**
  - I.e., not a stack
- **Must be explicitly allocated *and* released**
- **May be accessed *only* with pointers**
  - Remember, an array is equivalent to a pointer
- **Many hazards to the *C* programmer**

Dynamic Data Allocation



Allocating Memory in The Heap

- See *<stdlib.h>*

void *malloc(size_t size);
void free(void *ptr);
void *calloc(size_t nmemb, size_t
void *realloc(void *ptr, size_t size);

free()* knows *size* of chunk allocated by *malloc()* or *calloc()

Segmentation *fault* and/or big-time error if bad pointer

- ***malloc()* — allocates *size* by** **heap and returns a pointer to it.**
  - *NULL* pointer if allocation fa reason
- ***free()* — returns the chunk of memory pointed to by *ptr* back to the heap**
  - *Must* have been allocated by *malloc()* or *calloc()*

Notes

- ***calloc()* is just a variant of *malloc()***
- ***malloc()* is analogous to *new* in *C++* and Java**
  - *new* in *C++* actually calls *malloc()*
- ***free()* is analogous to *delete* in *C++***
  - *delete* in *C++* actually calls *free()*
  - Java does not have *delete* — uses *garbage collection* to recover memory no longer in use

Example usage of *malloc()* and *free()*

```
#include <stdlib.h>
int PlayGame(int board[], int arraySize);

int main(){
    int s, t;
    int A[];
    s = …;          /* determine size of array from input */
    A = malloc(s * sizeof(int));
    …;
    t = PlayGame(A, s);
    …
    free(A);
    return t;
}
```

Alternate version of malloc() and free()

```
#include <stdlib.h>
int PlayGame(int *board, int arraySize);

int main(){
    int s, t;
    int *A;
    s = …;          /* determine size of array from input */
    A = malloc(s * sizeof(int));
    …;
    t = PlayGame(A, s);
    …
    free(A);
    return t;
}
```

Generalization

- ***malloc()* and *free()* are used …**
  - whenever you need a dynamically sized array
  - whenever you need an array that does not follow last in, first out rule of *The Stack*

- **Valid in all versions of *C***
  - See p. 167 of K&R (§7.8.5)

malloc vs. calloc

- ***void *malloc(size_t #ofBytes)***
  - Takes number of bytes as argument
  - Aligned to largest basic data type of machine
    - » Usually *long int*
  - Does *not* initialize memory
  - Returns pointer to *void*
- ***void *calloc(size_t #ofItems, size_t sizeOfItem)***
  - Takes number of # of items and size as argument
  - *Initializes memory to zeros*; returns pointer to void
- ***calloc(n, itemSize) ≡ malloc(n * itemSize)***
  - Plus initialization to zero

realloc()

void *realloc(void *p, size_t newSize)

- Changes the size of a *malloc'ed* piece of memory by allocating new area (or changing size of old area)
- May increase or decrease size
- Data copied from old area to new area
  - If larger, extra space is uninitialized
- *free(p)* – i.e., frees old area (if new area different from old)
- Returns pointer to *void* of new area
- See p. 252 of K&R

Rarely used in *C* programming!

Definition – Memory Leak

- **The steady loss of available memory in *Heap* due to program forgetting to *free* everything that was *malloc'*ed.**
  - Bug-a-boo of most large *C* and *C*++ programs
- **If you overwrite or lose the value of a pointer to a piece of *malloc'*ed memory, there is no way to find it again!**
  - Automatically creating a memory leak
- **Killing the program frees *all* memory!**

Questions?

Multi-Dimensional Arrays

- ***int D[10][20]***
  - A *one-dimensional array* with 10 elements, each of which is an array with 20 elements
- **I.e., *int D[10][20]*** ***/*[row][col]*/***
- **Last subscript varies the fastest**
  - I.e., elements of last subscript are stored contiguously in memory
- **Also, three or more dimensions**

Limitations of Multi-Dimensional Arrays

- ***C* supports only fixed size multi-dimensional arrays**
- **Rows stored contiguously in memory ⇒**
  - Compiler must know how many elements in a row
  - Cannot pass an array to a function with open-ended number of elements per row

- ***void f(int A[][20], int nRows); /*okay*/***
- ***void f(int A[10][], int nCols); /*NOT okay */***
- ***void f(int A[][], int nRows, int nCols);***
  /*also NOT okay */
- **Same for three or more dimensions!**

Dynamic Two-Dimensional Arrays

- **Array of pointers, one per row**
  - Each row is a one-dimensional array

```
int x, y;     /* x = #rows, y = #columns */
int **B = calloc(x, sizeof(int *));
for (i = 0; i < x; i++)
        B[i] = calloc(y, sizeof(int));
…
/*   to access element of row i, column j */
        B[i][j]
```

Why does this work?

- ***int **B* means the same as *int *B[];***
  - I.e., ***B*** is an array of pointers to *int*
  - Therefore, ***B[i]*** is a pointer to *int*

- **But if *A* is a pointer to *int*, then *A[j]* is the *jth integer* of the array *A***

- **Substitute *B[i]* for *A***
- **Therefore, *B[i][j]* is the *jth integer* of the *ith* row of *B***

Why does this work? (continued)

- **When expression contains *B[i][j]*, compiler generates code resembling**
  - *int *temp = B + i;*
  - **(temp + j);*
- **… in order to access this element**

Alternative Method

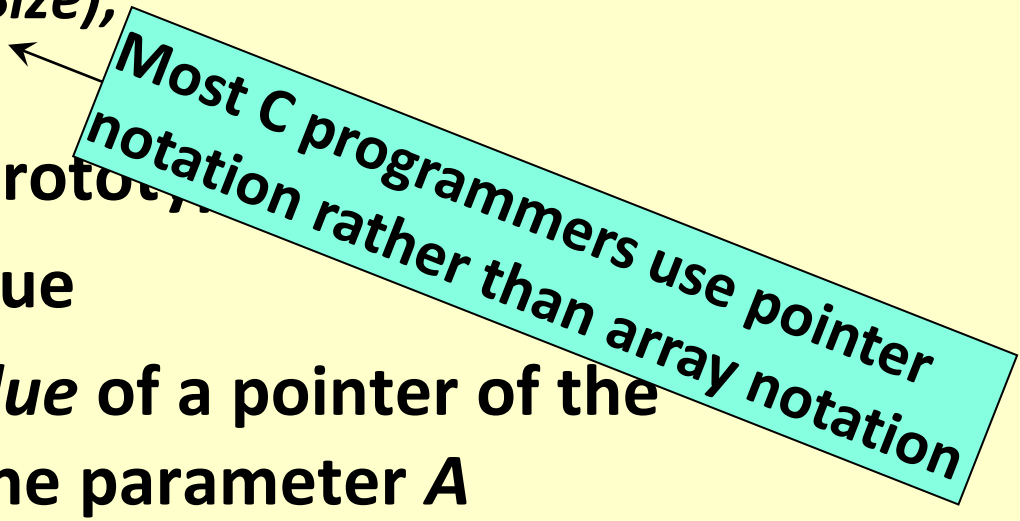
```
int x, y;     /* x = #rows, y = #columns */
int **B = calloc(x, sizeof(int *));
int elements[] = calloc(x*y, sizeof(int));
for (i = 0; i < x; i++)
        B[i] = &elements[i*y];
…
/*   to access element of row i, column j */
        B[i][j]
```

Why does this work?

Exercise for the student!

Arrays as Function Parameters (again)

- *void init(float A[], int arraySize);*
  *void init(float *A, int arraySize);*



- Are identical function protot
- Pointer is passed by value
- I.e. caller copies the *value* of a pointer of the appropriate type into the parameter *A*
- Called function can reference *through* that pointer to reach thing pointed to

Arrays as Function Parameters (continued)

- ***void init(float A[][], int rows, int columns);***
  void init(float **A, int rows, int columns);

- **Not Identical!**
- **Compiler complains that *A[][]* in header is incompletely specified**
  - Needs to know number of columns
  - As if constant!
- ***Must* use pointer notation in header or declaration**
- **In body of *init*, *A[i][j]* is still acceptable**

Questions?