Data Structures, Dynamic Memory allocation & the Heap

(Chapter 19)

1

Recap

- Memory layout…Run-time stack
- Implementing function calls
  - Activation record for each function
  - Push record onto stack when function called
  - Pop record from stack when function returns
  - Supports recursion
- Arrays
  - Consecutive locations in memory…
    - Define higher dim arrays
- Pointers
  - Link between arrays and pointers

2

2

Structures

- Programs are solving a 'real world' problem
  - Entities in the real world are real 'objects' that need to be represented using some data structure
    - With specific attributes
  - Objects may be a collection of basic data types
    - In C we call this a structure

3

3

Example..Structures in C

- represent all information pertaining to a student

```
char GWID[9],char lname[16],char fname[16],float gpa;
```

- We can use a **struct** to group these data together for each student, and use typedef to give this type a name called student

```
struct student_data {
   char GWID[9];
  char lname[16];
  char fname[16];
  float gpa
   };
```

```
typedef struct student_data {
   char GWID[9];
  char lname[16];
  char fname[16];
  float gpa
  } student;
student seniors; // seniors is
              var of type student
```

4

4

Arrays and Pointers to Struct

•We can declare an array of structs

```
student enroll[100];
Enroll[25].GWID='G88881234';
```

•We can declare and create a pointer to a struct:

```
student *stPtr; //declare pointer to student type
stPtr = &enroll[34]; //points to enroll[34]
```

•To access a member of the struct addressed by Ptr:

```
(*stPtr).lname = 'smith'; //dereference ptr and
                           access field in struct
```

•Or using special syntax for accessing a struct field through a pointer:

- `stPtr-> lname = 'smith';`

5

Passing Structs as Arguments

•Unlike an array, a struct is always passed by value into a function.

- This means the struct members are copied to the function's activation record, and changes inside the function are not reflected in the calling routine's copy.

•Most of the time, you'll want to pass a pointer to a struct.

```
int similar(student *studentA, student *studentB)
{
  if (studentA->lname == studentB->lname) {
     ...
  }
  else
    return 0;
}
```

6

Dynamic Allocation

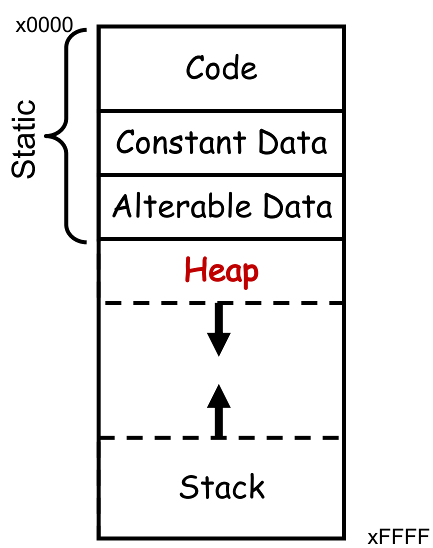
•Suppose we want our program to handle
a variable number of students – as many as the user wants to enter.

- We can't allocate an array, because we don't know the maximum number of students that might be required.
- Even if we do know the maximum number,
it might be wasteful to allocate that much memory
because most of the time only a few students' worth of data is needed.

•Solution:
Allocate storage for data dynamically, as needed.

Recall: Memory Layout

- Global data grows towards xFFFF
  - Global ptr R4
- Stack grows towards zero
  - Top of stack R6
  - Frame pointer R5
  - ALL local vars allocated on stack with address R5 + offset (-ve)
- Heap grows towards xFFFF
- We've used Stack and static area so far – where does Heap come in ?



Memory allocation review: Static Memory Allocation

- In this context "static" means "at compile time"
  - I.e., compiler has information to make final, hard-coded decisions
- Static memory allocation
  - Compiler knows all variables and how big each one is
  - Can allocate space to them and generate code accordingly
    - Ex, global array: compiler knows to allocate 100 slots to `my_static_array[100]`
    - Address of array is known statically, access with `LEA`

```
#define MAX_INTS 100
int my_static_array [MAX_INTS];

my_static_array .BLKW #100 ; # 100 words, 16-bits each on LC3
```

9

9

Automatic (variables) Memory Allocation

- Automatic memory allocation
  - Used for stack frames
  - Similar to static allocation in many ways
  - Compiler knows position and size of each variable in stack frame
    - Symbol table generated at compile time
    - Offset values for local variables (negative values for local var)
    - Local variables have address R5 + offset
  - Can generate code accordingly
  - Can "allocate" memory in hardcoded chunks
    - Relative to stack pointer (R6) and frame pointer (R5)

```
ADD R6, R6, #-3   ;; allocate space for 3 variables
LDR R0, R5, #-2   ; loads local variable intro R0
```

10

10

What Is Dynamic Memory Allocation?

- “Dynamic” means “at run-time”
  - Compiler doesn’t have enough information to make final decision

- Dynamic memory allocation
  - Compiler may not know how big a variable is
    - Most common example…how many elements in an array
  - Compiler may not even know that some variables exist
    - ?

- How does it figure out what to do then?
  - It doesn’t, programmer has to orchestrate this manually

11

11

Dynamic Allocation

- What if size of array is only known at run-time ?
- Dynamic allocation
  - Ask for space at run-time…How?
  - Need run-time support – call system to do this allocation
  - Provide a library call in C for users
- Where do you allocate this space – heap

12

12

Static vs. Dynamic Allocation

- There are two different ways that multidimensional arrays could be implemented in C.
- Static: When you know the size at compile time
  - A Static implementation which is more efficient in terms of space and probably more efficient in terms of time.
- Dynamic: what if you don't know the size at compile time?
  - More flexible in terms of run time definition but more complicated to understand and build
  - Dynamic data structures
- Need to allocate memory at run-time – malloc
  - Once you are done using this, then release this memory – free
- Next: Dynamic Memory Alloction

13

13

Heap API

•How does programmer interface with "heap"?
- Heap is managed by user-level C runtime library (**`libc`**)
- Interface function declarations found in **`"stdlib.h"`**
- Two basic functions…

void *malloc(size_t size); /* Ask for memory
- Gives a pointer to (address of) heap region of size **`size`**
  - If success, space is contiguous
- **`Returns NULL if heap can't fulfill request`**
- Note: **`void *`** is "generic pointer" (C for "just an address")
  - Can pass it around, but not dereference

void free(void *ptr); /* release memory
- Returns region pointed to by **`ptr`** back to "heap"

14

14

Using malloc

•To use malloc, we need to know how many bytes to allocate. The `sizeof` operator asks the compiler to calculate the size of a particular type.

- Assume `n` is number of students to enroll
- (student is struct)

- `enroll = malloc(n * sizeof(student));`

•We also need to change the type of the return value to the proper kind of pointer – this is called "casting."

- ```
  enroll =
    (student*) malloc(n* sizeof(student));
  ```

15

15

Example

```
int num_students;
student *enroll;

printf("How many students are enrolled?");
scanf("%d", &num_students);

enroll =
  (student*) malloc(sizeof(student) *num_students);
if (enroll == NULL) {
  printf("Error in allocating the data array.\n");
  ...
}
enroll[0].lname = 'smith';
```

If allocation fails, malloc returns NULL.

Note: Can use array notation or pointer notation.

16

16
```

free

•Once the data is no longer needed,
it must be released back into the heap for later use.

•This is done using the free function,
passing it the same address that was returned by malloc.

```
void free(void*);
free(enroll[0]);
```

•If allocated data is not freed, the program might run out of heap memory and be unable to continue.

- *Even though it is a local variable, and the values are 'destroyed', the allocator assumes the memory is still in use!*

17

17

Heap API Example

```
unsigned int i, num_students;
struct enroll *student; /* assume student has size 5 */

/* prompt user for number of students */
printf("enter maximum number of students: ");
scanf("%u\n", &num_students);

/* allocate student array */
enroll =
     malloc(num_students * sizeof(struct student));

/* do something with them */

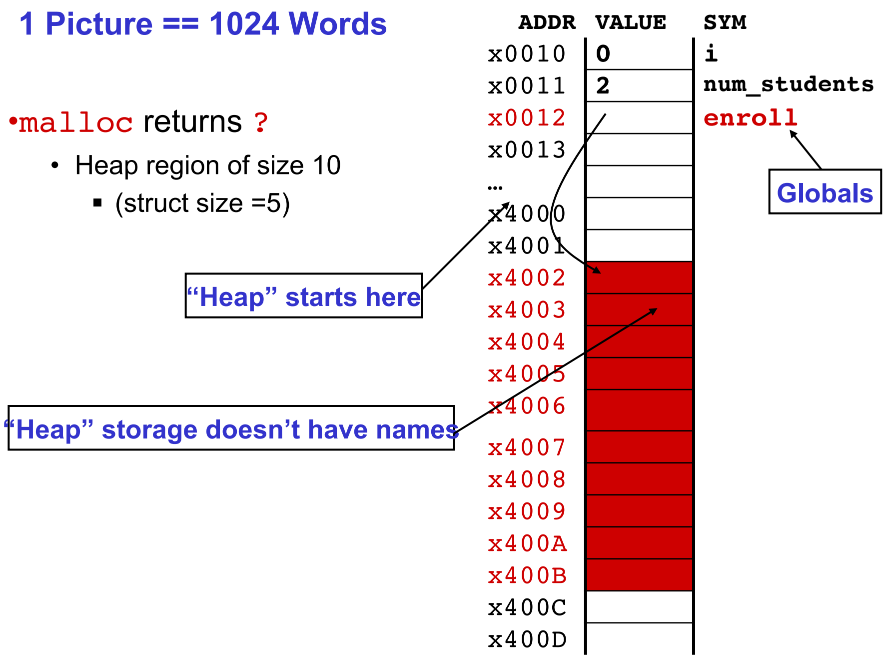
/* free students array */
free(enroll);
```

18

18

1 Picture == 1024 Words

- `malloc` returns ?
  - Heap region of size 10
    - (struct size =5)

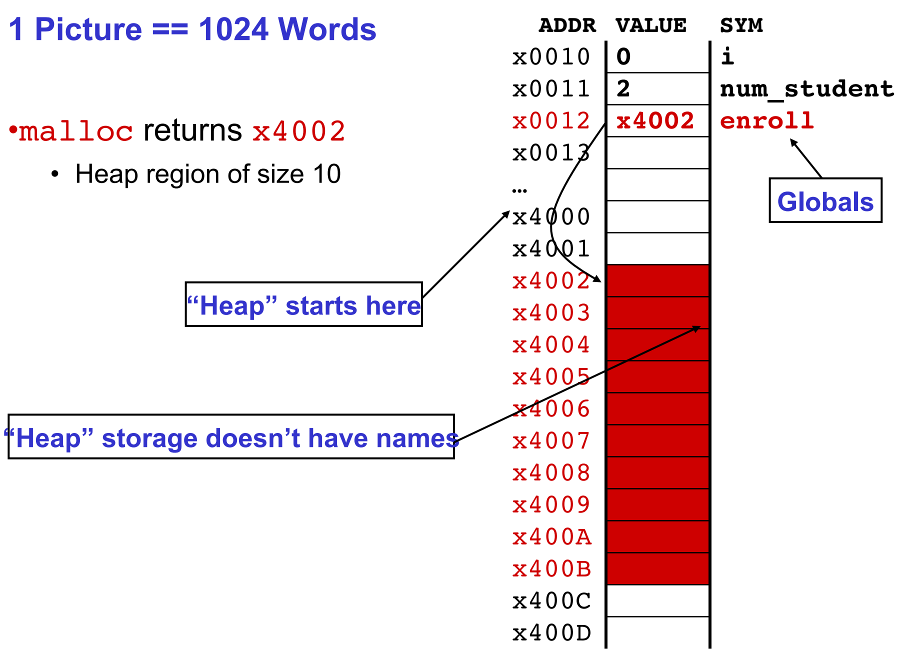


19

19

1 Picture == 1024 Words

- `malloc` returns `x4002`
  - Heap region of size 10



20

20

1 Picture == 1024 Words

- malloc returns x4002
  - Heap region of size 10
- What if enroll was local var ?

"Heap" starts here

"Heap" storage doesn't have names

ADDR	VALUE	SYM
x0010	0	i
x0011	2	num_bullets
x0012		
x0013		
…		
x4000		
x4001		
x4002		
x4003		
x4004		
x4005		
x4006		
x4007		
x4008		
x4009		
x400A		
x400B		
x400C		
x400D		

Globals

21

Malloc & local vars

Address	Content	Value
x3000	*x	
x3001	*B: B[0]	
x3002	B[1]	
x3FFE	i	10
x3FFF	B	x3001
x4000	x	x3000

Heap starts at x3000

R5
(frame ptr for function test)

```
int test( int a){
int* x,B;
int i= 10;
x = (int*)malloc(1*sizeof(int));
B= (int*) malloc(2*sizeof(int));
……
free(x);
return i;
}
```

x, B are local vars of type pointer
in Function test..
Allocated on stack
but to addresses on Heap

22

malloc, free & memory leaks

Address	Content	Value
x3000		
x3001	*B: B[0]	
x3002	B[1]	
x3FFF		
x4000		

- After function test Returns…
  - x was freed
  - B was not
- Stack does not contain
  Local vars x,B
- Heap still thinks B[0],B[1]
  are being used

➡ Program no longer has pointers that. can access this space….
memory leak !

23

23

Heap: Managing Malloc

How does it work?
What is a good malloc implementation ?

24

Malloc Package

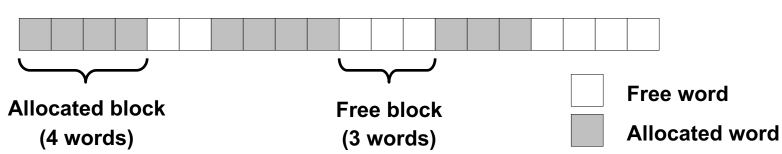
- `#include <stdlib.h>`
- `void *malloc(size_t size)`
  - If successful:
    - Returns a pointer to a memory block of at least `size` bytes, (typically) aligned to 8-byte boundary.
    - If `size == 0`, returns NULL
  - If unsuccessful: returns NULL (0) and sets `errno`.

- `void free(void *p)`
  - Returns the block pointed at by `p` to pool of available memory
  - `p` must come from a previous call to `malloc` or `realloc`.

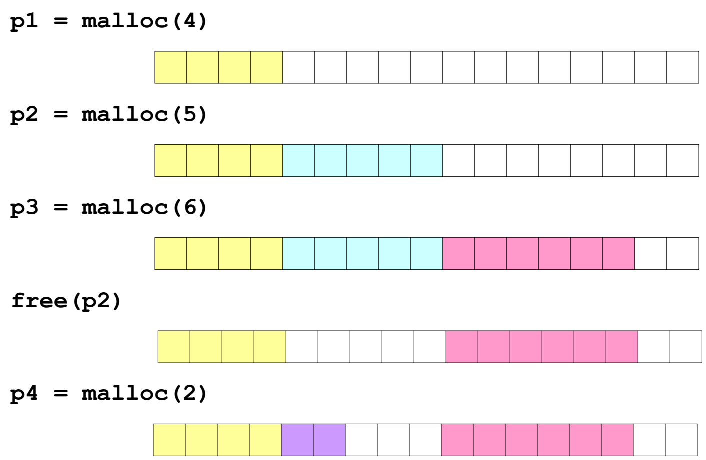
- `void *realloc(void *p, size_t size)`
  - Changes size of block `p` and returns pointer to new block.

Assumptions

- Assumptions
  - Memory is word addressed (each word can hold a pointer)



Allocation Examples



27

Goals of Good malloc/free

- Primary goals
  - Good time performance for `malloc` and `free`
    - Ideally should take constant time (not always possible)
    - Should certainly not take linear time in the number of blocks
  - Good space utilization
    - User allocated structures should be large fraction of the heap.
    - Want to minimize "fragmentation".
- Some other goals
  - Good locality properties – *motivation for this will be discussed later in course*
    - Structures allocated close in time should be close in space
    - "Similar" objects should be allocated close in space
  - Robust
    - Can check that `free(p1)` is on a valid allocated object `p1`
    - Can check that memory references are to allocated space

28

Challenges & problems with Dynamic allocation

- What can go wrong ?
- How to fix it ?

29

29

Memory leak

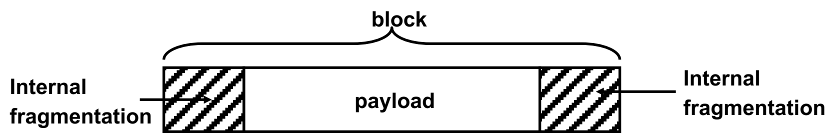
- forgot to free()
  - Allocator assumes the memory is still in use
- Overwrote pointer to block…oops: cannot get to the memory anymore
- Thumb rule:
  - For every malloc there should be an associated free
  - Will this solve all your problems ?

30

30

Internal Fragmentation

- Poor memory utilization caused by *fragmentation*.
  - Comes in two forms: internal and external fragmentation
- Internal fragmentation
  - For some block, internal fragmentation is the difference between the block size and the payload size.

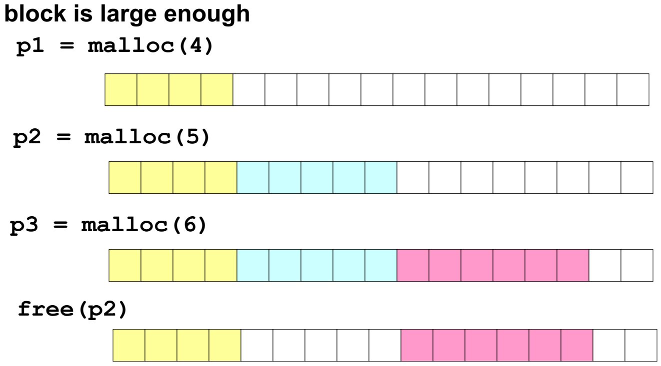


  - Caused by overhead of maintaining heap data structures, padding for alignment purposes, or explicit policy decisions (e.g., not to split the block).
  - Depends only on the pattern of *previous* requests, and thus is easy to measure.

31

External Fragmentation

Occurs when there is enough aggregate heap memory, but no single free block is large enough



`p4 = malloc(6)` **oops!** We have 7 free blocks but not 6 contigious free blocks.

External fragmentation depends on the pattern of *future* requests, and thus is difficult to measure.

32

Implementation issues

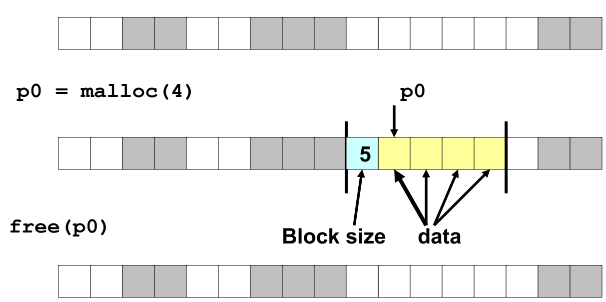
- How to 'collect' all the free blocks
  - How to keep track of them
  - Where to insert free block
  - How to determine amount of free memory ?
- Compaction ?
  - Can alleviate external fragmentation problems

33

33

Knowing How Much to Free

- Standard method
  - Keep the length of a block in the word preceding the block.
    - This word is often called the *header field* or *header*
  - Requires an extra word for every allocated block

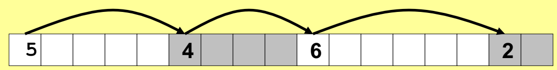


34

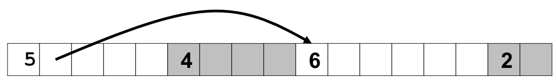
34

Keeping Track of Free Blocks

•*Method 1*: *Implicit list* using lengths -- links all blocks



•*Method 2*: *Explicit list* among the free blocks using pointers within the free blocks



•*Method 3*: *Segregated free list*
- Different free lists for different size classes
  - Ex: one list for size 4, one for size 8, etc.

•*Method 4*: Blocks sorted by size
- Can use a balanced tree (e.g. Red-Black tree) with pointers within each free block, and the length used as a key

35

35

What to do with sets of free blocks?

•During program run-time, blocks are no longer in use but may not have been freed
- So need to determine blocks no longer in use
- Keeping track of free blocks allows us to navigate the memory to determine blocks
- But when should we 'run' this process?
- But we still have fragmentation problem
- So what do we do…

Garbage Collection !

36

36

Implicit Memory Management: Garbage Collection

• *Garbage collection:* automatic reclamation of heap-allocated storage -- application never has to free

```
void foo() {
   int *p = malloc(128);
   return; /* p block is now garbage */
}
```

Common in functional languages, scripting languages, and modern object oriented languages:

- Lisp, ML, Java, Perl, Mathematica,
- **This is why you never worried about this problem in Java!**

Variants (conservative garbage collectors) exist for C and C++

- Cannot collect all garbage

37

37

Garbage Collection

• How does the memory manager know when memory can be freed?
  - In general we cannot know what is going to be used in the future since it depends on conditionals
  - But we can tell that certain blocks cannot be used if there are no pointers to them

• Need to make certain assumptions about pointers
  - Memory manager can distinguish pointers from non-pointers
  - All pointers point to the start of a block
  - Cannot hide pointers (e.g., by coercing them to an `int`, and then back again)

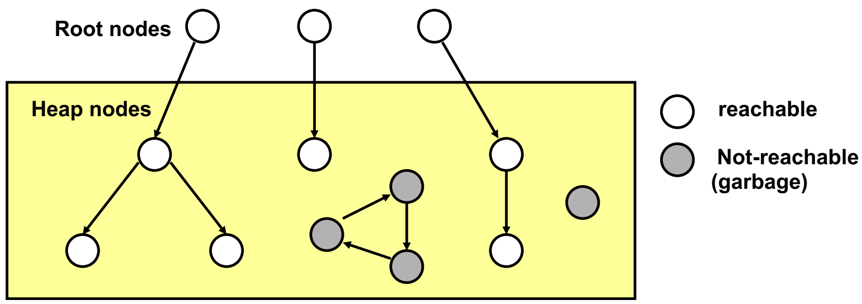
• Garbage collection process runs during program execution!

38

38

Garbage Collection: Memory as a Graph

- We view memory as a directed graph
  - Each block is a node in the graph
  - Each pointer is an edge in the graph
  - Locations not in the heap that contain pointers into the heap are called *root* nodes (e.g. registers, locations on the stack, global variables)



A node (block) is *reachable* if there is a path from any root to that node.

Non-reachable nodes are *garbage* (never needed by the application)

And you thought graphs were not useful ☺

39

39

Dynamic Data Structures

40

Recap…

•Static vs dynamic allocation

•Dynamic mem allocated on Heap

•Interface to Heap via:

- Malloc – ask for space, it returns pointer to space
- Free – return space to allocator when done

41

41

dynamic data structures …review

- Example 1: Linked List
  - Read example in textbook (or review your code from prior classes)
- Example 2. Dynamic arrays
- Example 3. Hash Tables
  - Project 5 and HW7

42

42

Data Structures

•A data structure is a particular organization of data in memory.

- We want to group related items together.
- We want to organize these data bundles in a way that is convenient to program and efficient to execute.

•An array is one kind of data structure.

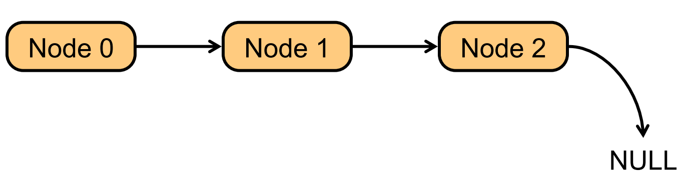
- `struct` – directly supported by C
- linked list – built from `struct` and **dynamic memory allocation**

43

Example 1: The Linked List Data Structure

•A linked list is an ordered collection of nodes, each of which contains some data, connected using pointers.

- Each node points to the next node in the list.
- The first node in the list is called the head.
- The last node in the list is called the tail.



44

student data structure in linked list

•Each student has the following characterics:
GWID, lname, fname, gpa.

•Because it's a linked list, we also need a pointer to the next node in the list:

•

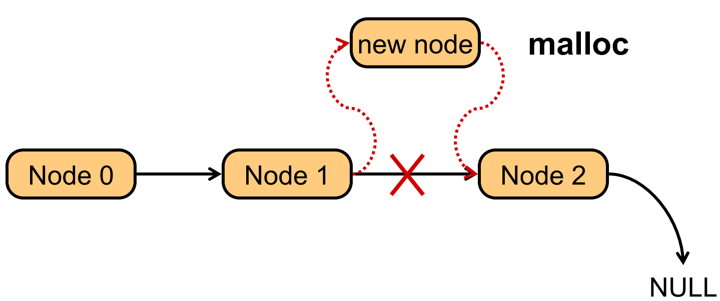
```
struct student {
  char[9] GWID;
  char lname[10];
  char fname[10];
  float gpa;
  student *next; /* ptr to next student in list */
}
```

45

45

Adding a Node

•Create a new node with the proper info.
Find the node (if any) with a greater GWID.
"Splice" the new node into the list:

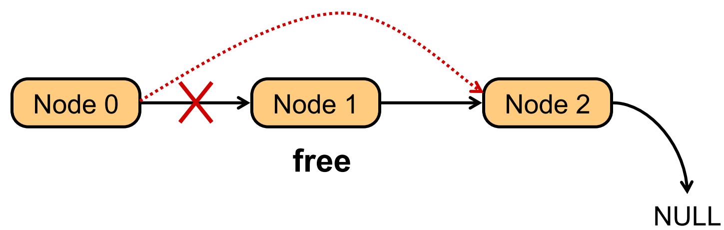


46

46

Deleting a Node

•Find the node that points to the desired node.
Redirect that node's pointer to the next node (or NULL).
Free the deleted node's memory.



47

47

Building on Linked Lists

•The linked list is a fundamental data structure.

- Dynamic
- Easy to add and delete nodes
- Doubly linked list – is more efficient in some apps

•The concepts described here are helpful
when learning about more elaborate data structures:

- *Trees, linked lists, array list,… – CS1112?*
- Hash Tables – HW7, project 5

48

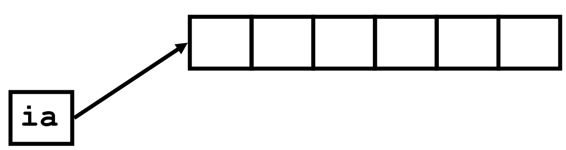
48

Example 2:
Dynamic Arrays & Multi-dimensional arrays

49

Static 1-D Arrays

```
int ia[6];
```



- **ia[4]** means ***(ia + 4)**

50

50

Dynamic arrays

•Don’t’ know size of array until run time

•Example: store an array of student records

- Do not know number of students until run time
- Size if specified by user at run-time

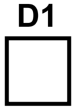
- Using static array of max size is a bad idea
  - Wasting space

51

51

Pictorially

D1



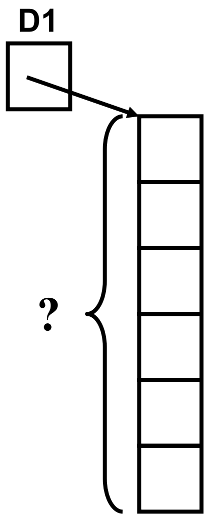
1-D array D1[n] – n determined at run-time

- D1 is an address – points to start of array!
- Type of D1 = pointer to < array type >

52

52

Pictorially



1-D array D1[n] – n determined at run-time

- D1 is an address – points to start of array!
- Type of D1 = pointer to < arraytype >

call malloc() and ask for space for n blocks

how much: n * sizeof(type)

D1 points to the start of these n blocks

After that access D1 as you would an array:

D1[i] accesses element i of the array

Once done, need to free D1

53

53

1-D Dynamic Array allocation

•Example 1-D dynamic array: `D-array1.c`

•C-code: download and go over the code

Outline:

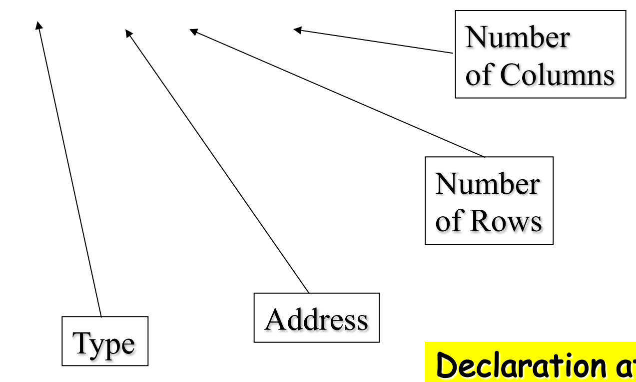
- Prompt user for size of array, call function **`allocarray`** to malloc space for the array, return to main and work on the array
- Declare dynamic array variable – pass this to function
- Call function **`allocarray`**: this function calls malloc to allocate space for the array
  - type returned by function: pointer to int
    - Pointer to a block of ints….array
  - Arguments to the function: size of array

54

54

Static 2-D array Declaration

```
int ia[3][4];
```

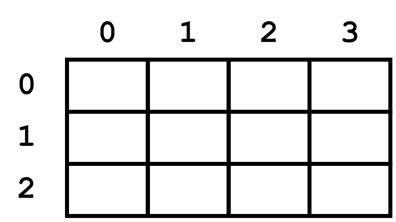


Declaration at compile time i.e. size must be known

55

55

How does a two dimensional array work?

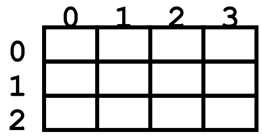


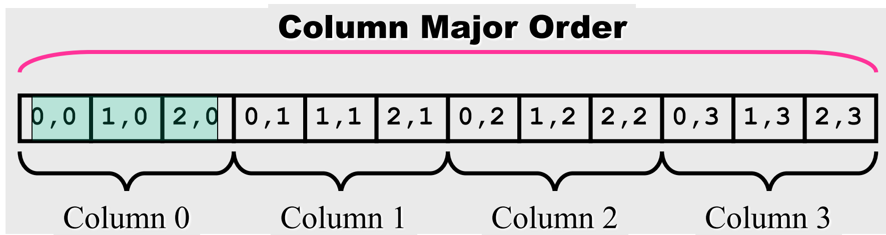
How would you store it?

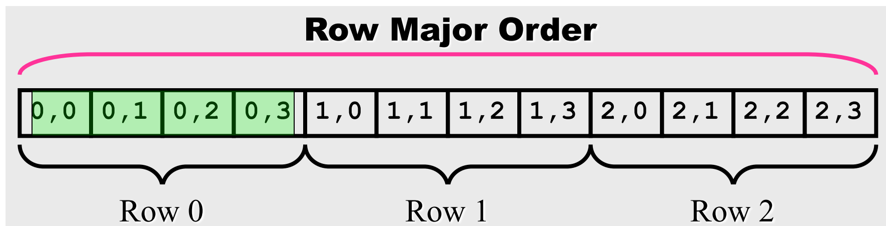
56

56

How would you store it?







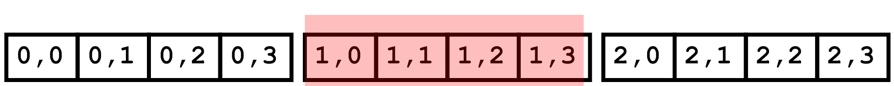
57

57

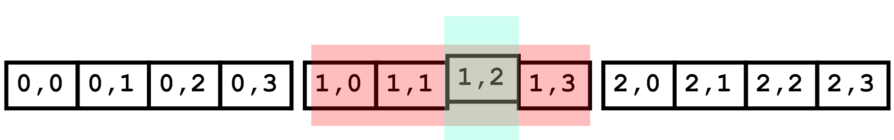
Advantage of row-major

- Using Row Major Order allows visualization as an array of arrays

`ia[1]`



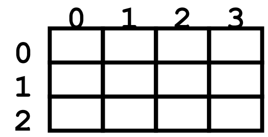
`ia[1][2]`



58

58

How does C store 2-D arrays ?

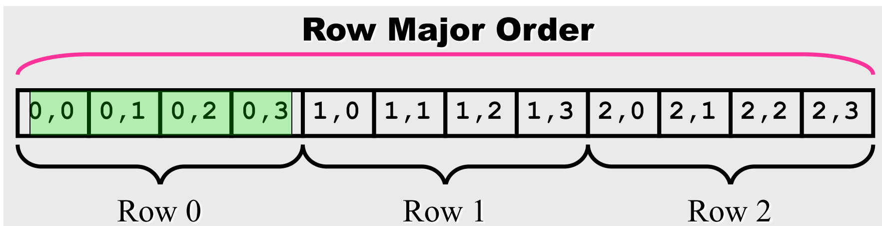


Row major

Pointer arithmetic stays unmodified

Remember this…..

Affects how well your program does when you access memory



59

59

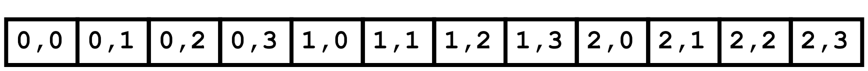
Element Access

- Given a row and a column index
- How to calculate location?
- To skip over required number of rows:

    `row_index * sizeof(row)`

    `row_index * Number_of_columns * sizeof(arr_type)`

- This plus *address of array* gives address of first element of desired row
- Add **`column_index * sizeof(arr_type)`** to get actual desired element



60

60

Element Access

```
Element_Address =

      Array_Address +
        Row_Index * Num_Columns * Sizeof(Arr_Type) +
        Column_Index * Sizeof(Arr_Type)


Element_Address =

      Array_Address +
        (Row_Index * Num_Columns + Column_Index) *
           Sizeof(Arr_Type)
```

61

61

Recall: pointers and arrays

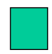
- One Dimensional Array
  `int ia[6];`
- Address of beginning of array:
  `ia ≡ &ia[0]`

- Two Dimensional Array
  `int ia[3][6];`
- Address of beginning of array:
  `ia ≡ &ia[0][0]`
- also
- Address of row 0:
  `ia[0] ≡ &ia[0][0]`
- Address of row 1:
  `ia[1] ≡ &ia[1][0]`
- Address of row 2:
  `ia[2] ≡ &ia[2][0]`

62

Now think about 3-D array

- A 3D array



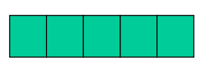
`int a`

63

63

Now think about

- A 3D array



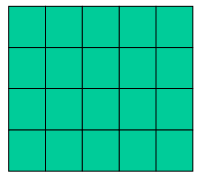
`int a[5]`

64

64

Now think about

- A 3D array



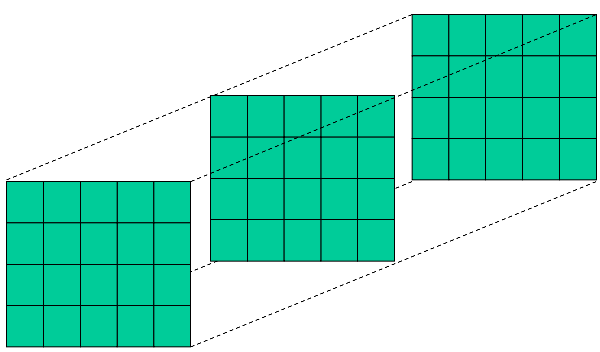
`int a[4][5]`

65

65

Now think about

- A 3D array



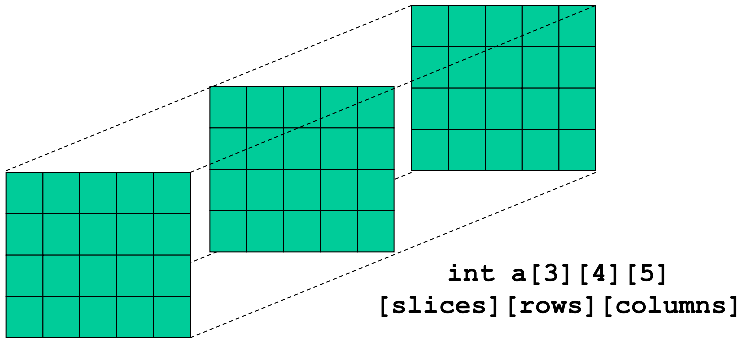
`int a[3][4][5]`

66

66

Offset to a[i][j][k]?

- A 3D array




```
offset = (i * rows * columns) + (j * columns)
         + k
```

67

67

2-D dynamic arrays

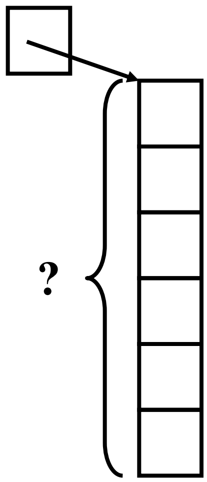
- We do not know #rows or #columns at compile time
  - Need to prompt user for this info
- How did 1-D arrays work?
  - Pointer to block of words
  - Block of words is the array
- How can we extend this
  - Pointer to 1-D array of "rows"
  - *Each entry in this array is a pointer to the row*
    - How many elements in the row = number of columns

68

68

Pictorially

D1: Pointer to 1-D array



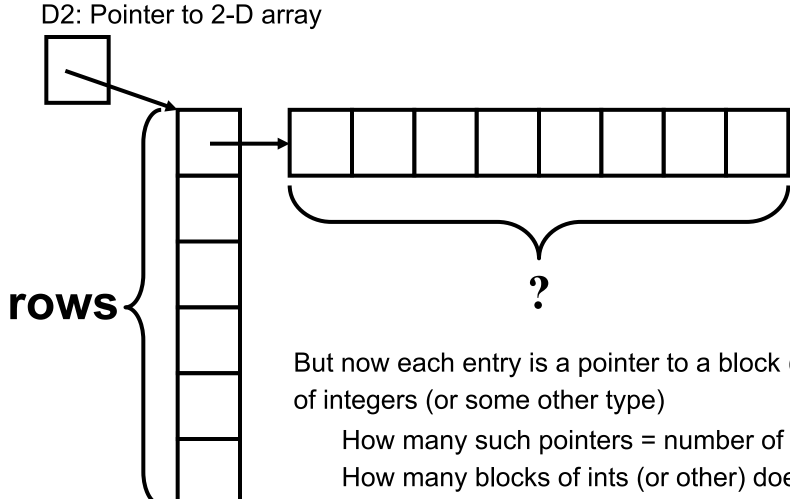
Each entry is an integer (or some other type)

69

69

Pictorially: 2-D array

D2: Pointer to 2-D array



But now each entry is a pointer to a block (row) of integers (or some other type)

How many such pointers = number of rows

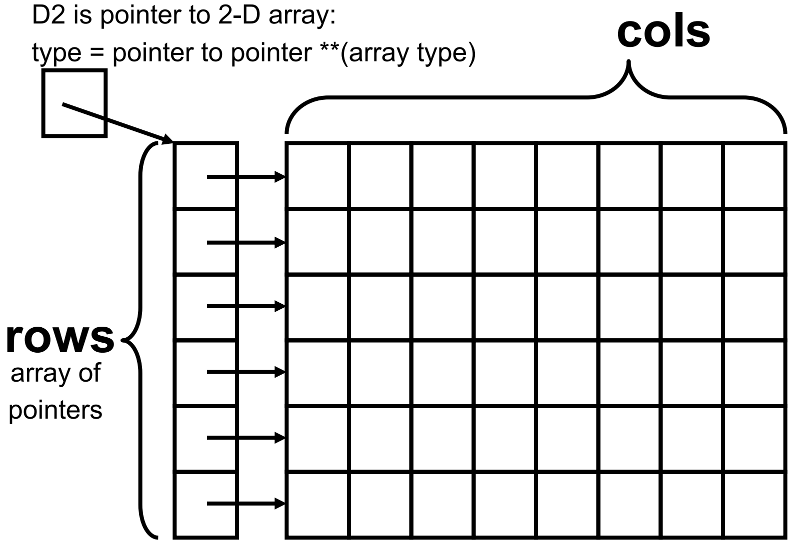
How many blocks of ints (or other) does each point to ? ..number of columns

70

70

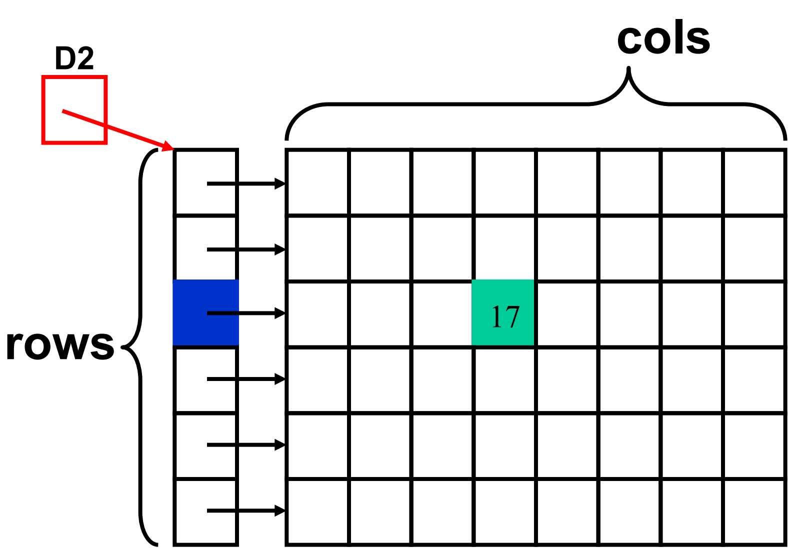
Pictorially

D2 is pointer to 2-D array:
type = pointer to pointer **(array type)



71


```
D2[2][3] = 17;
```



72

2-D Dynamic Array allocation

•Example 2 2-D dynamic array: `D-array2.c`

Outline:

- Prompt user for size of array, call function allocarray to malloc space for the array, return to main and work on the array
- Declare dynamic array variable – pass this to function
  - This is a pointer to a pointer – i.e, **int
- Fill in function allocarray: this function calls malloc to allocate space for the array
  - Determine type returned by function
  - Arguments to the function
- Don't forget to **free**
  - Think about how to free all the space used by 2-D array

73

73

Hash Tables

HW 7 and Project 5

74

The Hash Table

- A useful general purpose data structure which can be scaled to store large numbers of items and offers relatively fast access times.
- Put another way, a Hashtable is similar to an *Array*
  - But with an index whose data type we define:
    - Normally:
      - array [int] = some value
    - But with a Hashtable it is “kind of” like:
      - array [*my_structure*] = *some value*
      - array [*float*] = *some value*
      - array [*string*] = *some value*
      - array [another_array] = *some value*
    - The programmer (YOU) defines a way to map your datatype to an actual integer index
- Why is this useful?
  - Makes “searching” easy

75

75

Hash functions

•Array GW[ ] of students

- To find student with ID x, GW*[x]*

•Domain of student IDs ?

- GWID: G_ _ _ _ _ _ _ _

•Range ?

•Ideally: array of size= number of GW students

- GW[x] will be entry for student with ID x
- But this is not definition of an array!

76

76

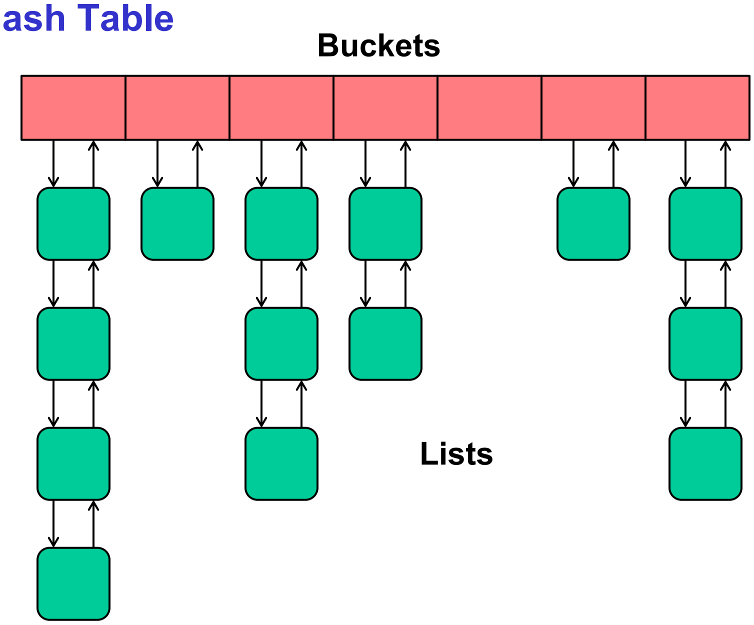
The Hash Table

- Designed to store (key,value) pairs
- Idea
  - Take every key and apply a hash function which returns an integer – this integer is the index of a bucket where you store that object.
  - These buckets are usually implemented as linked lists so if two or more keys hash to the same bucket they are all stored together.
  - The number of elements stored in each bucket should be roughly equal to the total number of elements divided by the total number of buckets

77

77

Hash Table



78

Some hash table vocabulary

- Key: portion of your data that you use to map to bucket
- Value: bucket # in hash table array (aka the index #)
- Hash Function: maps key to value
  - AKA: mapping function
  - AKA: map
  - AKA: "hashing"
- Associative Array
  - What a hash table actually is: an array whose index is associated with your custom datatype
- Collision:
  - When more than 1 key maps to the same value
    - AKA: bucket contains more than 1 data item
    - We used linked list to allow collisions
    - Perfect hash function yields no collisions!
- Load factor: # of entries in table / # of buckets

79

79

Dynamic Resizing of Buckets

- Resizing Hash Table Dynamically?
  - Adding more buckets at runtime (or reducing)
    - Ensures less collisions
  - Wouldn't be necessary if we knew how much data would be in table in advance!
    - This is just not practical most times
  - This is often implemented in practice once load factor ~ .75
    - The purpose is to keep search 'linear' in terms of time
      - And efficiently use memory
    - You wish to keep storage & lookup of items independent of # of items stored in hashtable
      - Refining the hash function is the key to this (not resizing)
  - Why not make it huge?
    - Diminishing returns: eventually too many buckets = wasted memory

80

Hash functions

- The purpose of this:
    - Have an exact way to “find” the data later on
    - We use hash function to “lookup” the bucket our data is in
        - We use our linked lists’s “find” to search within the bucket
- If we knew we had 1000 distinct values, then h(k) will be between 1 and 1000
- Simple hash function: Modulo B for B buckets
    - If B=100 then h(k) = K mod 100
- What is domain if strings and not int
    - **Example: Add up ASCII values of characters in string s to get an integer x, then apply modulo**
    - h(s) = x mod B
- How do you rebalance the load – extendible hashing functions
    - Mod k to start with (k is power of 2)
    - Mod 2k

81

81

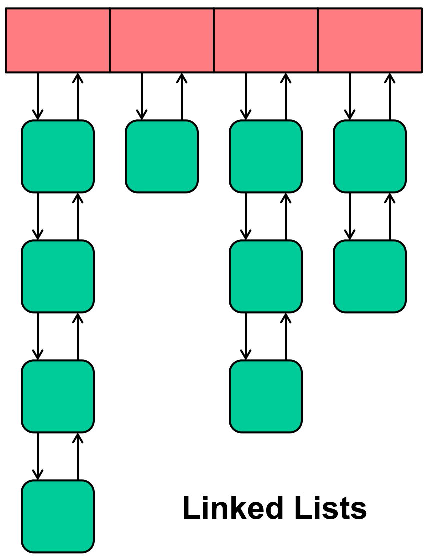
How do we implement a HashTable?

- It is almost like a 2D array:
    - Except the # of columns differs for each row
        - my_array [0] = {data1, data2}
        - my_array [1] = {data3}
        - my_array [2] = {data4, data5, data6}
- Under the hood, we usually do use an array
    - We call the # of rows the # of “buckets” in the table
    - But we usually make the columns a linked list
        - Example: my_linked_list* my_array [10]
            - my_array [0] = linked_list0
            - my_array [1] = linked_list1
            - my_array [2] = linked_list2
            - …
        - This would define an array of 10 linked lists

82

82

Struct_of_ints Hash Table



← 4 Buckets

Each bucket is really just a head pointer to 4 separate linked lists

Hash "mapping" function tells us which "bucket" data must go into

Linked lists hold onto data that fits into more than 1 bucket

83

Example using struct_of_ints linked list

- we created a linked list ..let's say of "ints"
  - Let's use them as the basis for our hash table

```
#include "linked_list.h"
#define BUCKETS 4
int main () {
    int i, bucket ;
    struct_of_ints* my_hash_tbl [BUCKETS] ;
    /* think about how you may need to change to deal with
    Case when number of buckets is input by the user */
    printf ("Enter INT\n" ) ;
    scanf ("%d", &i) ;

    bucket = i % BUCKETS ;    // maps key to value

    // store data in hashtable
    my_hash_tbl[bucket] =         // we access like an array
            add_to_list ( my_hash_tbl[bucket], i) ;
}
```

84

84