Data Structures, Dynamic Memory allocation & the Heap

(Chapter 19)

1

Quick Review: Structures in C

- 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

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

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.
- Another example: linked list
  - We need to keep adding/deleting nodes in the list….
    - Size of the data (structure) varies during run time
- Solution:

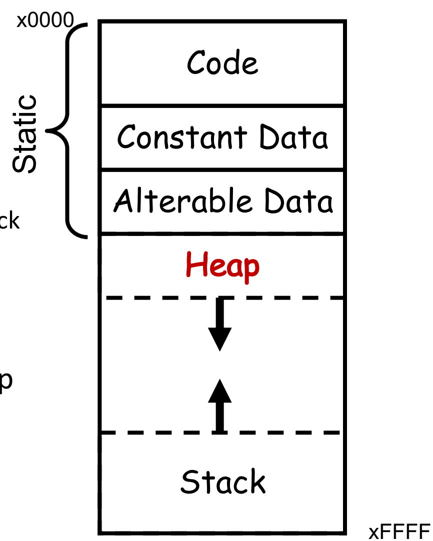
Allocate storage for data dynamically as needed.

7

7

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
- Heap grows towards xFFFF
- We've used Stack and static area so far – where does Heap come in ?



8

8

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 (local variables) Memory Allocation

- Automatic (local var) 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
- Ask for space at run-time…how ?
  - Need run-time support – call system to allocate memory: provide library call in C for users = malloc()
- where do you allocate this in memory…the Heap

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
    - **malloc()**
- Where do you allocate this space – heap

12

12

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… `malloc` and `free`

13

13

Heap API: Malloc Package

- `#include <stdlib.h>`
- `void *malloc(size_t size)`
  - If successful:
    - Returns a pointer to a *contiguous memory block* of at least `t_size` bytes, (typically) aligned to 8-byte boundary.
    - If `size == 0`, returns NULL
      - Note: void* is generic pointer (C for "just an address")
  - If unsuccessful: returns NULL (0) and sets `errno`.
- `void free(void *p)`
  - Returns the block pointed at by **p** back to heap
    - Its now in the 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.

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.
- Example: assume we want memory to store **n** number of students to enroll
  - `student` is a struct
  - ask for space to 'store' **n** student structures and **enroll** points to this space

```
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; /* this is a local variable –
                    pointer to a struct */

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…array since contiguous memory!

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 – array size num_students of type
 student*/
enroll =(student*)
     malloc(num_students * sizeof(struct student));

/* do some processing with enroll – data on the heap */

/* when done processing the data, free the data on the heap */
free(enroll);
```

18

18

Memory Layout – Heap & Stack

19

19

1 Picture == 1024 Words

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

Globals

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

"Heap" starts here

"Heap" storage doesn't have names

```
Enroll=(student*)
    malloc(num_students*
            sizeof(student))
```

20

1 Picture == 1024 Words

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

Globals

- malloc returns x4002
  - Heap region of size 10

"Heap" starts here

"Heap" storage doesn't have names

21

1 Picture == 1024 Words

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

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

Globals

"Heap" starts here

"Heap" storage doesn't have names

22

Example: Memory layout with heap variables

```
int test( int a){
int* x,B;
int i= 10;
   x = (int*)malloc(1*sizeof(int));
   B= (int*) malloc(2*sizeof(int));
……/* use X,B in body of function */

   free(x);  Free memory pointed to by x
   return i;
}
```

x,B are local variables of type pointers (to int)

malloc returns address x,B point to these addresses on the Heap

23

State of the Memory:

Address	Content	Value
3000		
3001		
3002		
3998	i	10
3999	B	
4000	x	

Heap starts at #3000

R5 (frame ptr for function test)

```
int test( int a){
int* x,B;
int i= 10;
…HERE….
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

24

24

Malloc & local vars

Address	Content	Value
3000	*x	
3001	*B: B[0]	
3002	B[1]	
3998	i	10
3999	B	3001
4000	x	3000

Heap starts at #3000

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));
……HERE…
free(x);
return i;
}
```

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

25

25

Malloc, free & memory leaks

Address	Content	Value
3000		
3001	*B: B[0]	
3002	B[1]	

Heap starts at #3000

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; }
```

…HERE…

- After function returns
  - x was freed, B was not!
- Stack does not contain local var x,B
- Heap still thinks B[0],B[1] are being used

→ Program no longer has pointers that can access this space…

Memory leak !

Check your code for mem leaks….

Valgrind !

26

26

Questions ?

- Work through HW5 : Capture state of memory through code execution
  - function call and return
  - global, stack and heap
  - (and code generation)

27

27

Dynamic Data Structures

Super fast Review !

28

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 HW6
- Project 5 – a "search engine"
  - Read documents in a directory
  - Enter a search phrase and find the most relevant document
  - Under the hood: build a hashmap
    - HW6 code….or CS2113 hashmap code

29

29

Example 2:
Dynamic Arrays & Multi-dimensional arrays

36

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

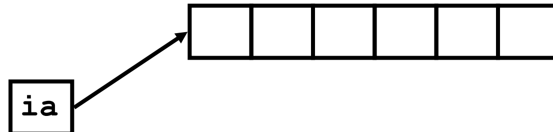
- Using static array of max size is a bad idea
  - Wasting space

37

37

Static 1-D Arrays

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

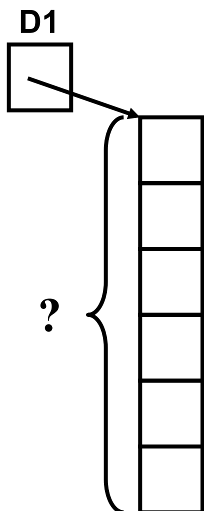


- `ia[4]` means `*(ia + 4)`
- Observe: ia is a pointer to a contiguous block of memory locations

38

38

Pictorially



1-D Array D1[n] – n known at run-time

Type of D1 = pointer to <arraytype>

Call malloc() and ask for space for n blocks

How much: n `* sizeof(type)`

Once allocated, access D1 as you would an array:

D1[i] accesses element i in array

Once you are done, `free(D1)`

40

40

1-D Dynamic Array allocation

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

•C-code: download (from webpage) and go over the code

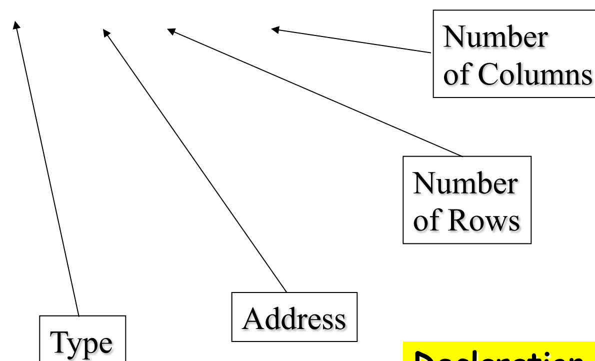
Outline:

- 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

41

Static 2-D array Declaration

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



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

42

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]`

43

2-D dynamic arrays

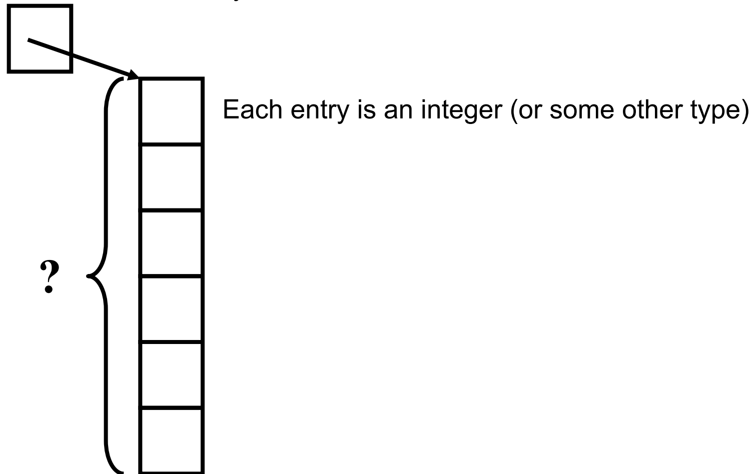
- We do not know #rows or #columns at compile time
  - Need to prompt user for this info
- How did 1-D arrays work?
  - Pointer to block of words
  - Block of words is the array
- How can we extend this
  - Pointer to 1-D array of “rows”
  - *Each entry in this array is a pointer to the row*
    - How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

44

44

Pictorially

D1: Pointer to 1-D array

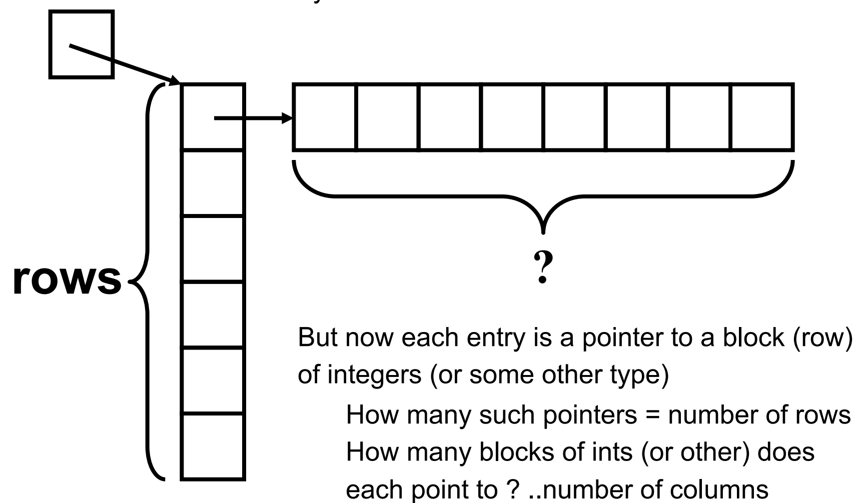


45

45

Pictorially: 2-D array

D2: Pointer to 2-D array



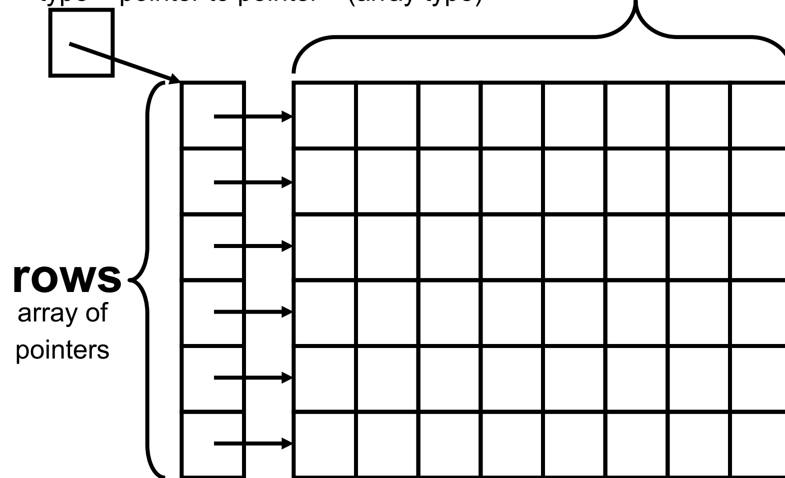
46

46

Pictorially

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

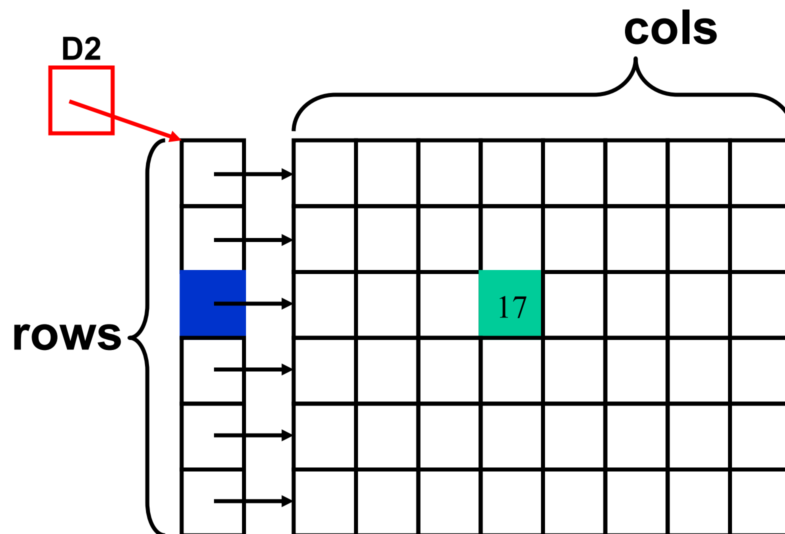
cols



47

47

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



48

48

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

49

49

Okay…practice what you learnt…

•Read through 1-D dynamic array code – linked on webpage and then use the template for 2-D arrays to write code to create 2-D dynamic arrays

50

50

Hash Tables/Hash Map

HW 6 and Project 5

51

Hash functions

- Array GW[ ] of students
  - To find student with ID x, GW*[x]*
- Domain of student IDs ? G followed by 8 digits
  - GWID: G_ _ _ _ _ _ _ _
  - 10^8
- 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!
- *hash function h can be viewed as mapping the value to the array index*:
  - *h[name]= i*
  - Array[*h(name)*] now works like an array!

53

53

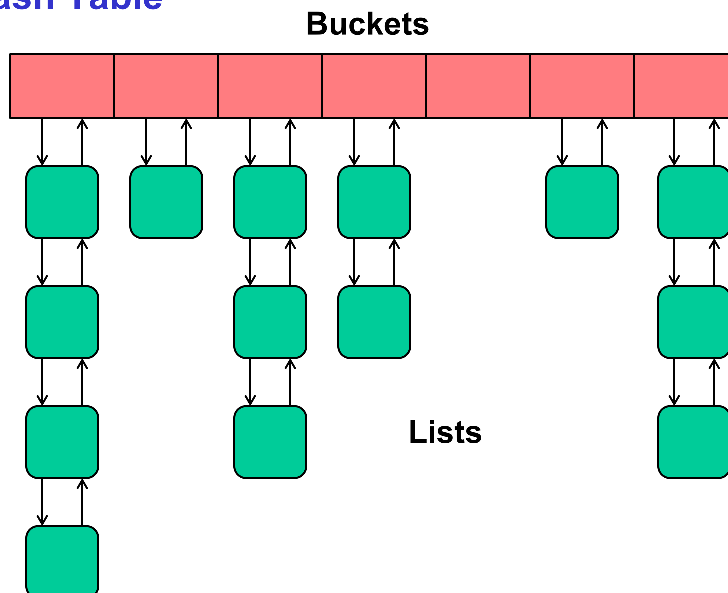
The Hash Table

- 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
- Example: hash function h = modulo 4
  - Maps to 4 buckets
  - h(10)= 2 – input data with key=10 is placed in bucket 2
  - h(15)=3 – input data with key=15 is placed in bucket 3
  - h(19)=3 – input data with key=19 is placed in bucket 3

54

54

Hash Table



55

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

56

56

56

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 if domain is strings and not integers
  - **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

58

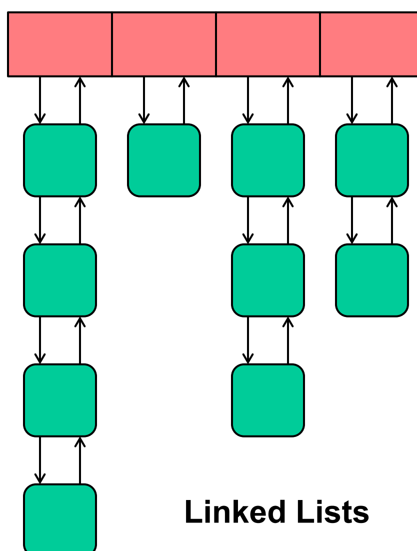
58

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

59

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

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) ;
}
```

61

61

Course…what remains.

- Rest of the course is mostly "C stuff"

- Topic remaining: 'real' memory organization and program performance
  - Simple code optimization techniques
  - Final Project handed out on last day – serves as your "final"
  - It is C code that you have to rewrite to improve performance AND you have to write a detailed report…your grade depends as much on the analysis in the report as the performance of your code.

62

62

Heap: Managing Malloc

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

You will learn more about this in Systems Prog

63

Malloc Package

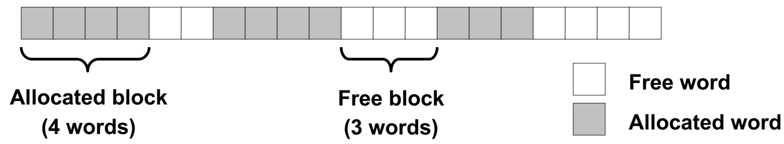
- **`#include <stdlib.h>`**
- **`void *malloc(size_t size)`**
  - If successful:
    - Returns a pointer to a contiguous memory block of at least `size` bytes, (typically) aligned to 8-byte boundary.
    - If `size == 0`, returns NULL
  - If unsuccessful: returns NULL (0) and sets `errno.`
- **`void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

64

64

Assumptions

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

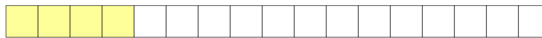


65

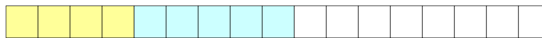
65

Allocation Examples

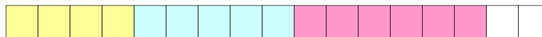
```
p1 = malloc(4)
```



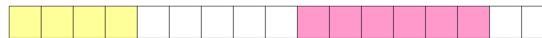
```
p2 = malloc(5)
```



```
p3 = malloc(6)
```



```
free(p2)
```



```
p4 = malloc(2)
```

66

66

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

67

67

Challenges & problems with Dynamic allocation

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

68

68

Memory leak

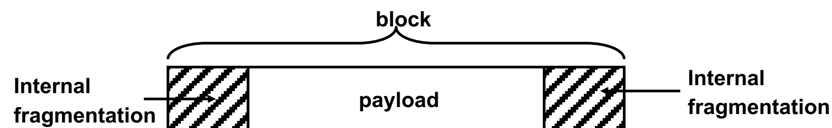
- forgot to free()
  - Allocator assumes the memory is still in use
- Overwrote pointer to block…oops: cannot get to the memory anymore
- Thumb rule:
  - For every malloc there should be an associated free
  - Will this solve all your problems ?
  - Exercise: Try to run the code we gave last class in the inclass exercise – but insert a free(x2) before return

69

69

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.

70

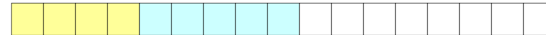
70

External Fragmentation

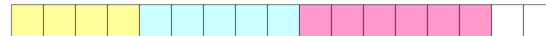
Occurs when there is enough aggregate heap memory, but no single free block is large enough

`p1 = malloc(4)`

`p2 = malloc(5)`



`p3 = malloc(6)`



`free(p2)`

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

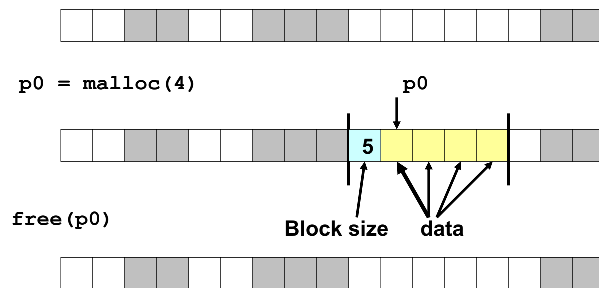
External fragmentation depends on the pattern of *future* requests, and thus is difficult to measure.

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

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

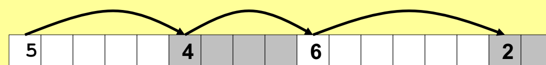


73

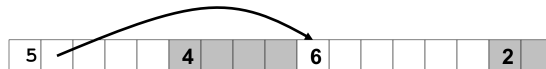
73

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

74

74

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 !

75

75

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

76

76

Garbage Collection

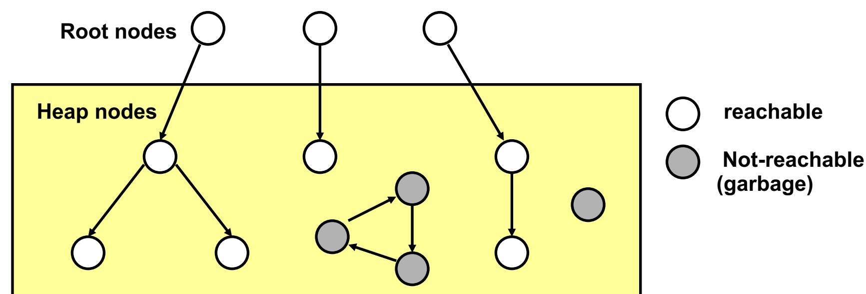
- How does the memory manager know when memory can be freed?
  - In general we cannot know what is going to be used in the future since it depends on conditionals
  - But we can tell that certain blocks cannot be used if there are no pointers to them

- 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!

77

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 ☺

78