foo(Char *** ptr, ---){
*ptr = malloc(---);
AF

A[][]

AF

Advanced Pointers

15-123

Effective Programming in C and Unix

Quick review

Pointers

- A pointer variable contains an address (eg: address of an int variable, address of a char, address of a char* etc)
- Any variable defined as
  - int x = 10;
  - has a value 10 and its address given by the unary operator & acting on x, That is, &x is the address of x
  - Later we learn that a pointer to x, can be passed to a function if the x needs to be changed inside the function
- Pointer variables can be declared as
  - int* ptr, char* ptr, …
- Declaration of a pointer variable does not allocate memory to dereference the pointer
  - Memory must be explicitly allocated before dereferencing the pointer using *ptr
  - Memory can be allocated using malloc(n), where malloc returns an address of a contiguous memory block of n bytes
    - Malloc returns a void*

Potential pointer (and other) errors

Run time errors

A) dereference of uninitialized or otherwise invalid pointer
B) insufficient (or none) allocated storage for operation
C) storage used after free
D) allocation freed repeatedly
E) free of unallocated or potentially storage
F) free of stack space
G) return, directly or via argument, of pointer to local variable
H) dereference of wrong type
I) assignment of incompatible types
J) program logic confuses pointer and referenced type
K) incorrect use of pointer arithmetic
L) array index out of bounds

---

int* x, ptr;

Identify the lines that cause errors

```
#include <stdio.h>
#include <stdlib.h>

int main(){
  int x=10, *ptr=&x ;
  printf("%x \n", ptr+1);
  printf("%d \n", *(ptr+1));
  printf("%d \n", *ptr+1);
  return 0;
}
```

next int to X

Crash

11

ptr

ptr+1

X	??

```
#include <stdio.h>
#include <stdlib.h>

int main(){
  int *ptr=malloc(25);
  for (int i=0; i<25; i++)
    ptr[i] = 0;
  return 0;
}
```

25 * sizeof(int)

*(ptr+i)

insufficient memory for int operations

Identify the lines that cause errors

```c
#include <stdio.h>
#include <stdlib.h>

int main(){
  int *ptr=malloc(20) ;
  int* A = ptr;
  for (int i=0; i<5; i++)
    A[i]=i;
  free(ptr) ;
  for (int i=0; i<5; i++)
    printf("%d",A[i]);
  return 0;
}
```

access freed memory

ptr=NULL

```c
#include <stdio.h>
#include <stdlib.h>

int main(){
  int *ptr= malloc(10) ;
  printf("%x \n", ptr+1);
  printf("%d \n", *(ptr+1));
  printf("%d \n", *ptr+1);
  free(ptr);
  printf("%d \n", *(ptr+1));
  printf("%d \n", *ptr+1);
  return 0;
}
```

FF, FF+4, garbage, garbage+1, ptr+1, heap

Identify the lines that cause errors

```c
#include <stdio.h>
#include <stdlib.h>

int* square(int x){
  int y = x*x;
  return &x;
}

int main(){
  int x = 10;
  printf("%x", square(x));
  printf("%d", *square(x));
  return 0;
}
```

&x

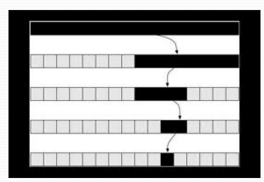
dereference an address of a local variable that does not exist

```c
#include <stdio.h>
#include <stdlib.h>

int main(){
  int x=10, *iptr=&x ;
  char ch='c', *cptr = &x;
  iptr = (int*)cptr;
  printf("%x", cptr+1);
  printf("%d", *cptr+1);
  printf("%d",*iptr);
  return 0;
}
```

Now to algorithms

Binary Search

- The idea of the binary search is that given a sorted array, one can efficiently search for a target in O(log n) time

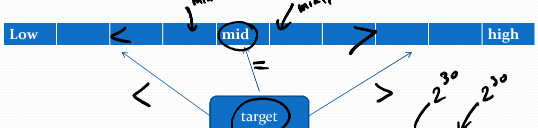


Source: withfriendship.com

$n = 2^{30} \approx$ billion

$\log_2 n \approx 30$

The binary search algorithm



mid-1, mid+1, <, =, >, 2^{30}, 2^{30}

$\text{mid} = \frac{\text{low}+\text{high}}{2} \stackrel{?}{=} \text{low} + \frac{(\text{high}-\text{low})}{2}$

If (A[mid] == target) done

If (A[mid] < target) search A[mid+1, high]

If (A[mid] > target) search A[low, mid-1]

If (low > high) target not found

Only 10% of the programmers can correctly implement binary search

Idea of the insertion sort

Sorted, unsorted

0 1 2 i-1

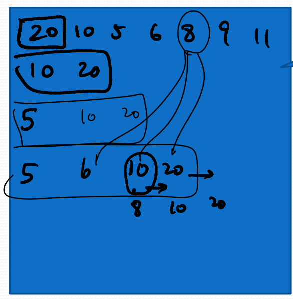
Assume A[0,0] is Sorted

Assume A[0,..i-1] is Sorted

Insert A[i] → A[0,...i]

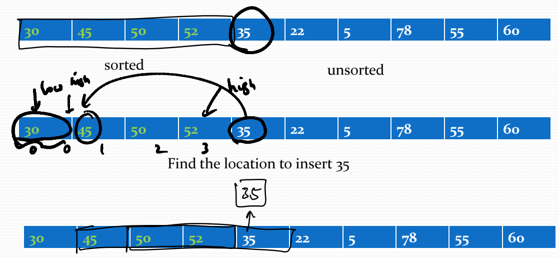
$i = 1, \dots, n-1$

Insertion Sort example



How efficient is insertion sort? (hint: count operations)

Making the insertion sort efficient



Find the location to insert 35

Move the block using memcpy

Cost of moving memory

for (int i=n-1; i>0 ; i--)
A[i] = A[i-1];

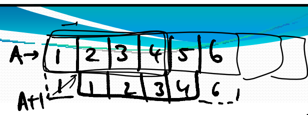
- How many bytes of memory was moved based on code logic?
- What if we can copy the entire block at once. How would we do that.
  - Computers perform bit shifting very efficiently

memcpy

```
MEMCPY(3)                Linux Programmer's Manual                MEMCPY(3)

NAME
       memcpy - copy memory area

SYNOPSIS
       #include <string.h>

       void *memcpy(void *dest, const void *src, size_t n);

DESCRIPTION
       The memcpy() function copies n bytes from memory area src to memory area
       dest.  The memory areas should not overlap.  Use memmove(3) if the  mem-
       ory areas do overlap.

RETURN VALUE
       The memcpy() function returns a pointer to dest.
```

Great way to move things around in an array

memmove



memmove(A+1, A, 4*sizeof(int))

```
NAME
       memmove - copy memory area

SYNOPSIS
       #include <string.h>

       void *memmove(void *dest, const void *src, size_t n);

DESCRIPTION
       The  memmove()  function  copies n bytes from memory area
       src to memory area dest.  The memory areas may overlap.

RETURN VALUE
       The memmove() function returns a pointer to dest.
```

Passing arguments to functions

```
#include <stdio.h>
#include <stdlib.h>

int sum(int* A, int n){
  for (int i=0, sum=0; i<n; i++){
    sum += i;
    return sum;
  }

int main(){
  int A[] ={1,2,3,4,5,6};
  printf("%d\n", sum(A,6));
  return 0;
}
```

How arguments are passed to functions

- Arguments to functions are passed by value
  - That is a copy of the value of the variable is given to the function
  - If the copy is just a value, function cannot change the original variable
  - If the copy is an address of a variable, the function can change the value of the calling variable
  - Arrays are always passed by “reference”. That is, the address of A is given to the function

```
#include <stdio.h>
#include <stdlib.h>

int sum(int* A, int n){
  for (int i=0, sum=0; i<n; i++){
    sum += i;
    return sum;
  }

int main(){
  int A[] ={1,2,3,4,5,6};
  printf("%d\n", sum(A,6));
  return 0;
}
```

Understanding **

Understanding **

- char** is an address of a variable of type char*
- char** reads
  - Pointer to a char*

char** char* char

- Recall : char* argv[]
  - Command line arguments are saved as an array of char*’s (or char**)

Passing an array of strings to a function

```
/*
   This function inserts the word to array[index].
   Must allocate memory to hold the word
*/

int insertWord(char **array, char *word, int cap, int index){


  return EXIT_SUCCESS;
}
```

Next lecture is on memory management

Go to recitation Wednesday

SL4 is optional but very helpful

Quiz 2 will be available shortly