Pointers and Debugging

15-123
Systems Skills in C and Unix

Learning Objectives

- At the end of this lecture
  - Understand the relation between 1D Arrays and Pointers
  - Understand pointer arithmetic with arrays
  - Understand the common errors introduced by pointers
  - Understand how to use a debugger to isolate and fix critical errors

Pointers are challenging

Pointers introduce hard to catch errors

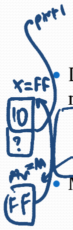
Why pointers cause errors?

- Many reasons
  - Dereferencing a pointer that has not being initialized

```
Char* s;        s          printf("%s", *s);
S=malloc(5);   [4bytes]
```

  - Dereferencing a pointer that is pointing to an illegal memory

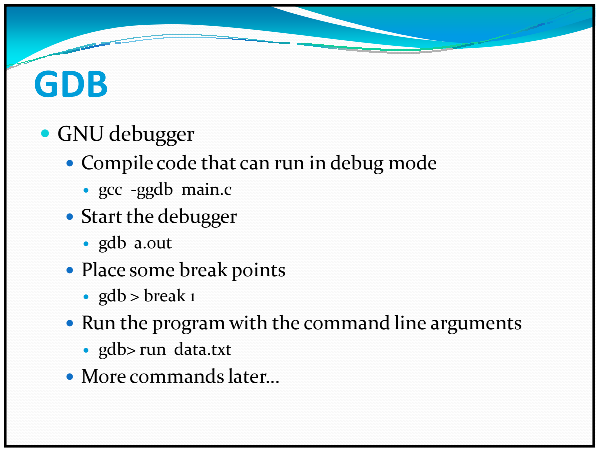
```
strcpy(s, "game");
int x=10;
int* ptr = &x;        A[n-1]='\0';     ptr++;
                                       printf("%d", *ptr);
```



  - Mixing pointers and integers

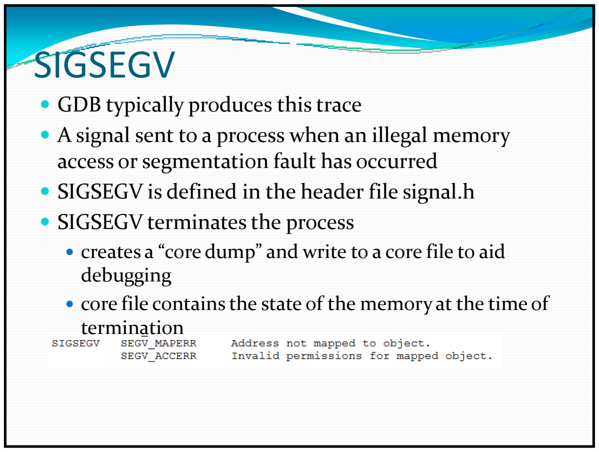
```
int x;
int* ptr = x;
```

GDB
GNU Debugger



GDB

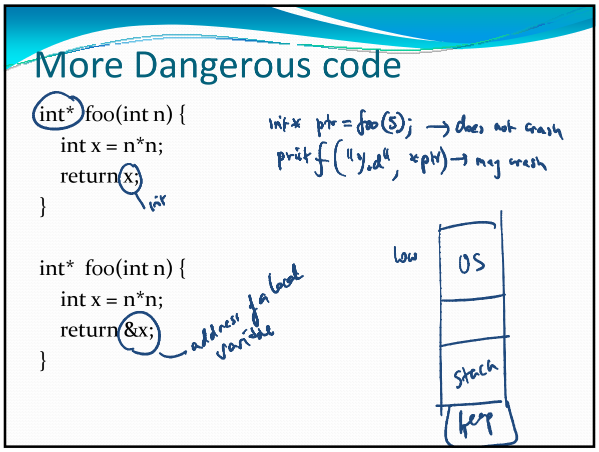
- GNU debugger
  - Compile code that can run in debug mode
    - gcc -ggdb main.c
  - Start the debugger
    - gdb a.out
  - Place some break points
    - gdb > break 1
  - Run the program with the command line arguments
    - gdb> run data.txt
  - More commands later…



SIGSEGV

- GDB typically produces this trace
- A signal sent to a process when an illegal memory access or segmentation fault has occurred
- SIGSEGV is defined in the header file signal.h
- SIGSEGV terminates the process
  - creates a "core dump" and write to a core file to aid debugging
  - core file contains the state of the memory at the time of termination

```
SIGSEGV   SEGV_MAPERR     Address not mapped to object.
          SEGV_ACCERR     Invalid permissions for mapped object.
```



More Dangerous code

```
int* foo(int n) {
   int x = n*n;
   return x;
}

int* foo(int n) {
   int x = n*n;
   return &x;
}
```

int

address of a local variable

int* ptr = foo(5); → does not crash

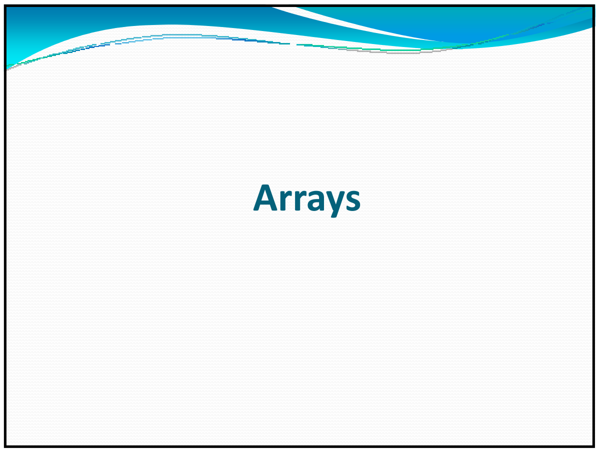
printf("%d", *ptr) → may crash

low

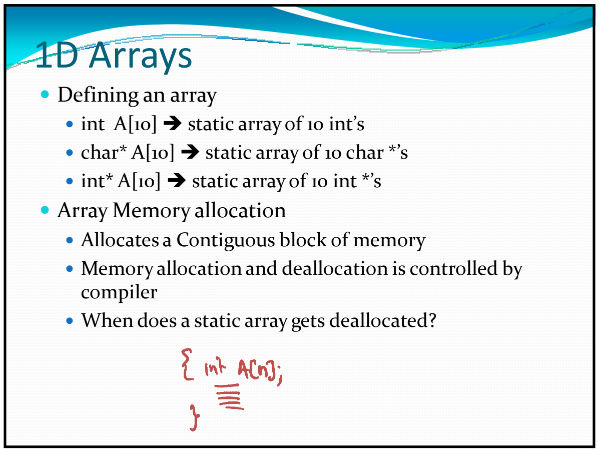
OS

stack

heap



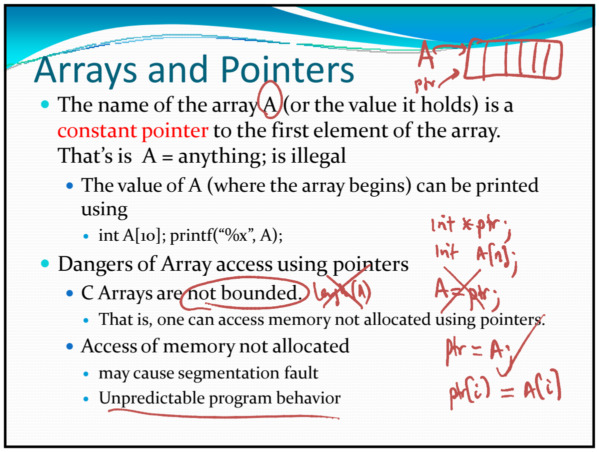
Arrays



1D Arrays

- Defining an array
  - int A[10] ➔ static array of 10 int's
  - char* A[10] ➔ static array of 10 char *'s
  - int* A[10] ➔ static array of 10 int *'s
- Array Memory allocation
  - Allocates a Contiguous block of memory
  - Memory allocation and deallocation is controlled by compiler
  - When does a static array gets deallocated?

```
{ int A[n];
     ≡
}
```



Arrays and Pointers

- The name of the array A (or the value it holds) is a constant pointer to the first element of the array. That's is A = anything; is illegal
  - The value of A (where the array begins) can be printed using
    - int A[10]; printf("%x", A);
- Dangers of Array access using pointers
  - C Arrays are not bounded. ~~length(A)~~
    - That is, one can access memory not allocated using pointers.
  - Access of memory not allocated
    - may cause segmentation fault
    - Unpredictable program behavior

A

ptr

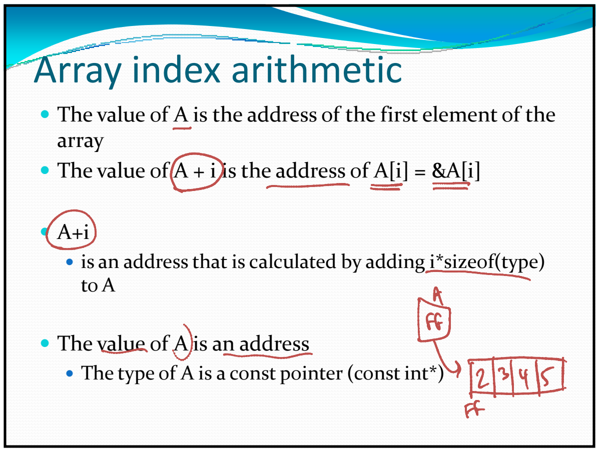
int *ptr;

int A[n];

~~A = ptr;~~

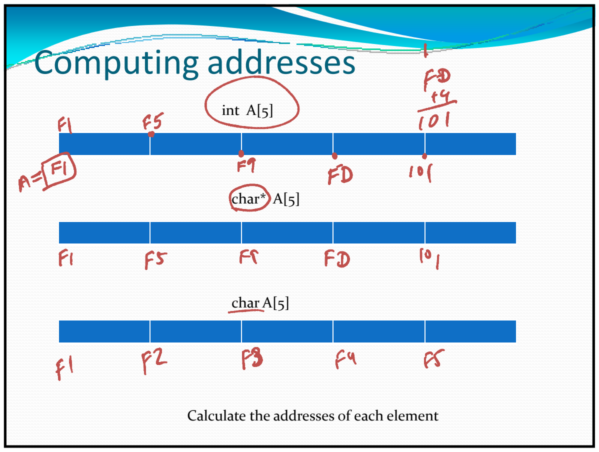
ptr = A; ✓

ptr[i] = A[i]



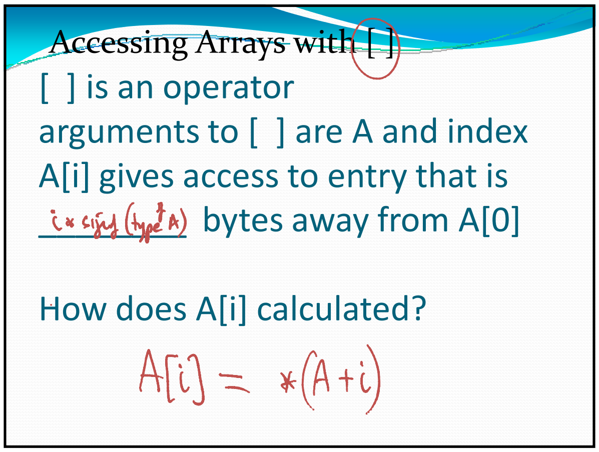
Array index arithmetic

- The value of A is the address of the first element of the array
- The value of A + i is the address of A[i] = &A[i]
- A+i
  - is an address that is calculated by adding i*sizeof(type) to A
- The value of A is an address
  - The type of A is a const pointer (const int*)



Computing addresses

Calculate the addresses of each element



Accessing Arrays with [ ]

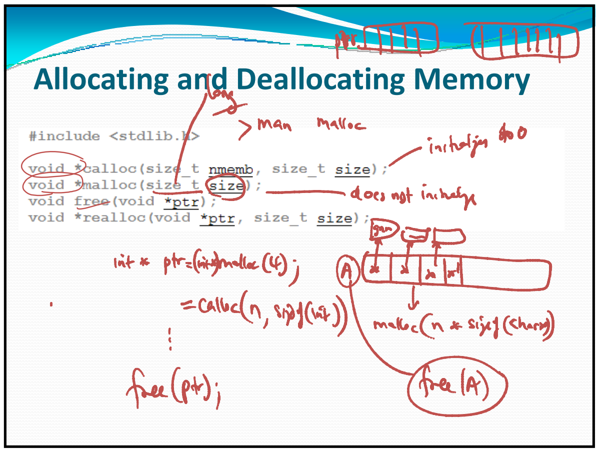
[ ] is an operator

arguments to [ ] are A and index

A[i] gives access to entry that is i * sizof(type of A) bytes away from A[0]

How does A[i] calculated?

$$A[i] = *(A+i)$$

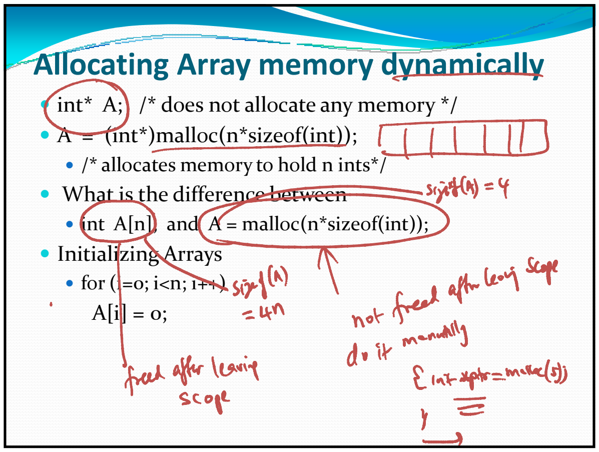


Allocating and Deallocating Memory

```
#include <stdlib.h>

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

ptr
long
> man malloc
initialize to 0
does not initialize
int * ptr = (int*)malloc(4);
= calloc(n, sizof(int))
free(ptr);
A
garb
malloc(n * sizof(char))
free(A)

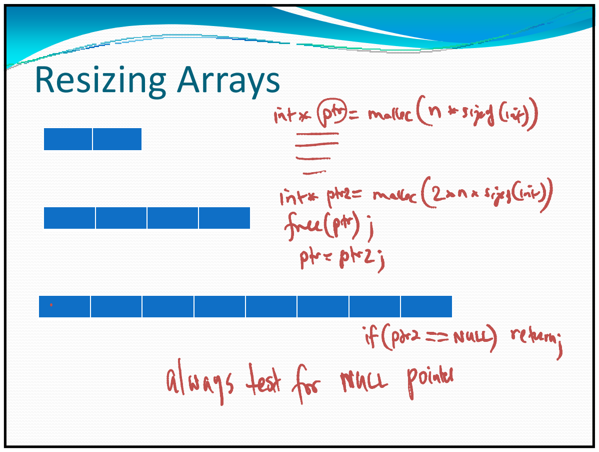


Allocating Array memory dynamically

- int* A; /* does not allocate any memory */
- A = (int*)malloc(n*sizeof(int));
  - /* allocates memory to hold n ints*/
- What is the difference between
  - int A[n], and A = malloc(n*sizeof(int));
- Initializing Arrays
  - for (i=0; i<n; i++)

    A[i] = 0;

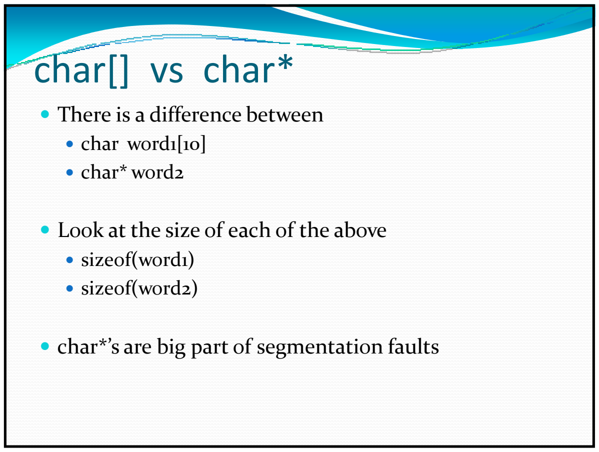
sizof(A) = 4
sizof(A) = 4n
not freed after leaving scope
do it manually
{ int *ptr = malloc(5);
}
freed after leaving scope



Resizing Arrays

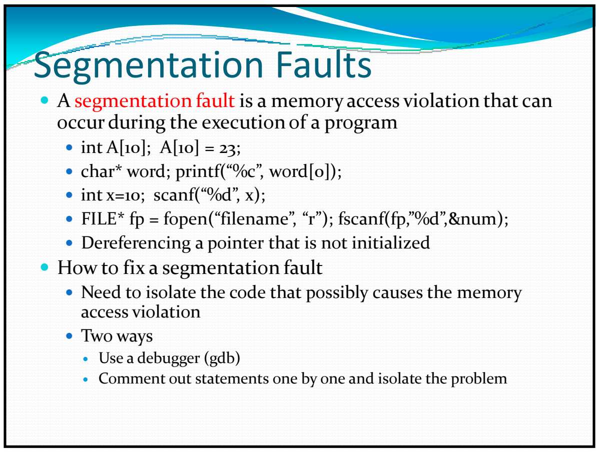
int * ptr = malloc(n * sizof(int))
int * ptr2 = malloc(2 * n * sizof(int))
free(ptr);
ptr = ptr2;
if (ptr2 == NULL) return;
always test for NULL pointer

Strings



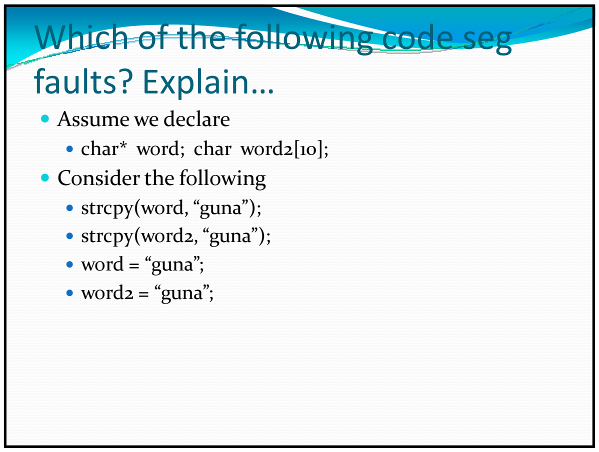
char[] vs char*

- There is a difference between
  - char word1[10]
  - char* word2
- Look at the size of each of the above
  - sizeof(word1)
  - sizeof(word2)
- char*'s are big part of segmentation faults



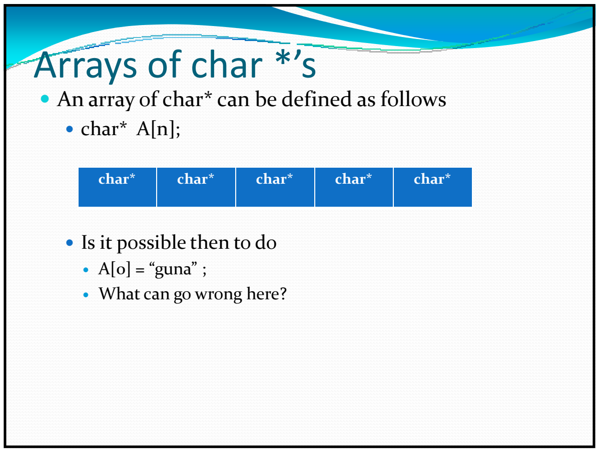
Segmentation Faults

- A segmentation fault is a memory access violation that can occur during the execution of a program
  - int A[10]; A[10] = 23;
  - char* word; printf("%c", word[0]);
  - int x=10; scanf("%d", x);
  - FILE* fp = fopen("filename", "r"); fscanf(fp,"%d",&num);
  - Dereferencing a pointer that is not initialized
- How to fix a segmentation fault
  - Need to isolate the code that possibly causes the memory access violation
  - Two ways
    - Use a debugger (gdb)
    - Comment out statements one by one and isolate the problem



Which of the following code seg faults? Explain…

- Assume we declare
  - char* word; char word2[10];
- Consider the following
  - strcpy(word, "guna");
  - strcpy(word2, "guna");
  - word = "guna";
  - word2 = "guna";

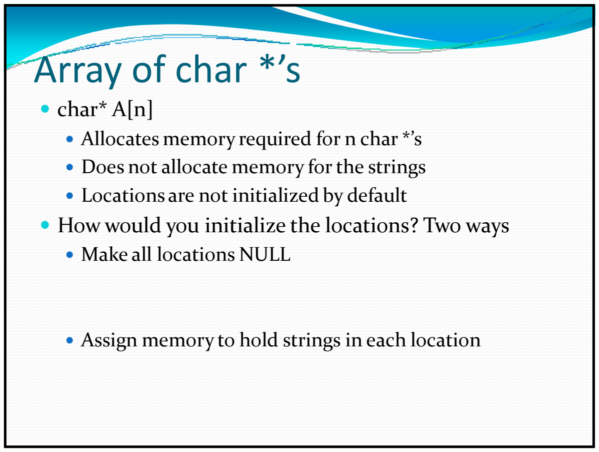


Arrays of char *'s

- An array of char* can be defined as follows
  - char* A[n];

char*	char*	char*	char*	char*

  - Is it possible then to do
    - A[0] = "guna" ;
    - What can go wrong here?



Array of char *'s

- char* A[n]
  - Allocates memory required for n char *'s
  - Does not allocate memory for the strings
  - Locations are not initialized by default
- How would you initialize the locations? Two ways
  - Make all locations NULL
  - Assign memory to hold strings in each location

Reading words

- char* A[n];
  - Does not allocate memory for Strings
- Allocate memory for each location
  - for (int i=0; i<n; i++)

```
      A[i] = malloc(strlen(word)+1)
    /* just allocate memory required for the current word*/
```

Dealing with runtime errors

Run time errors

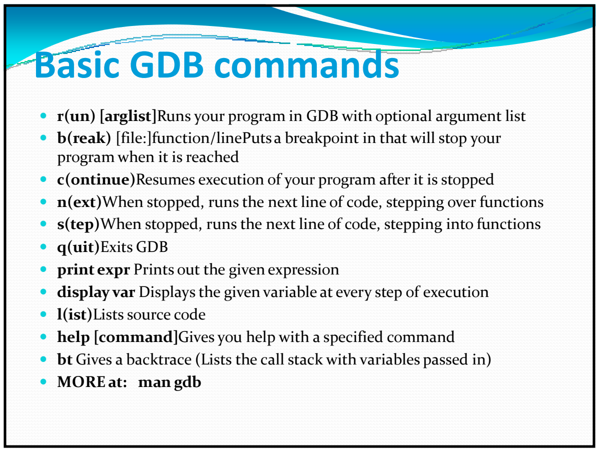
A) dereference of uninitialized or otherwise invalid pointer
B) insufficient (or none) allocated storage for operation
C) storage used after free
D) allocation freed repeatedly
E) free of unallocated or potentially storage
F) free of stack space
G) return, directly or via argument, of pointer to local 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

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

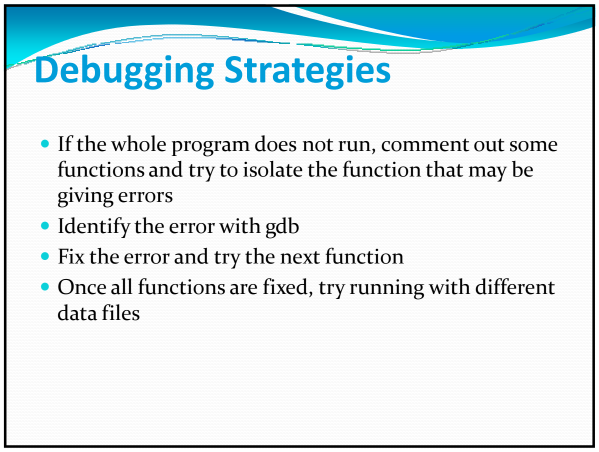
Process of debugging

- Need to develop a disciplined approach to programming
  - Best way to avoid errors is not to introduce in the first place
- When errors occur, find out where the program crashes
  - Sometimes with printf statements (be aware of buffer)
  - Most times printf's cannot tell us much
- Ideal way is to use a debugger
  - A program that can run your program step-by-step and provide an execution trace



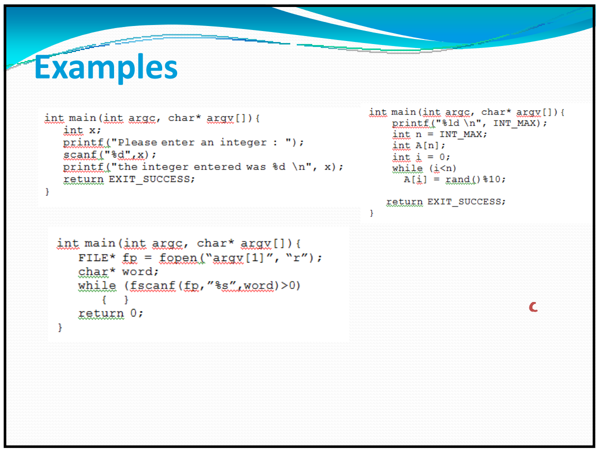
Basic GDB commands

- **r(un) [arglist]**Runs your program in GDB with optional argument list
- **b(reak)** [file:]function/linePuts a breakpoint in that will stop your program when it is reached
- **c(ontinue)**Resumes execution of your program after it is stopped
- **n(ext)**When stopped, runs the next line of code, stepping over functions
- **s(tep)**When stopped, runs the next line of code, stepping into functions
- **q(uit)**Exits GDB
- **print expr** Prints out the given expression
- **display var** Displays the given variable at every step of execution
- **l(ist)**Lists source code
- **help [command]**Gives you help with a specified command
- **bt** Gives a backtrace (Lists the call stack with variables passed in)
- **MORE at: man gdb**



Debugging Strategies

- If the whole program does not run, comment out some functions and try to isolate the function that may be giving errors
- Identify the error with gdb
- Fix the error and try the next function
- Once all functions are fixed, try running with different data files



Examples

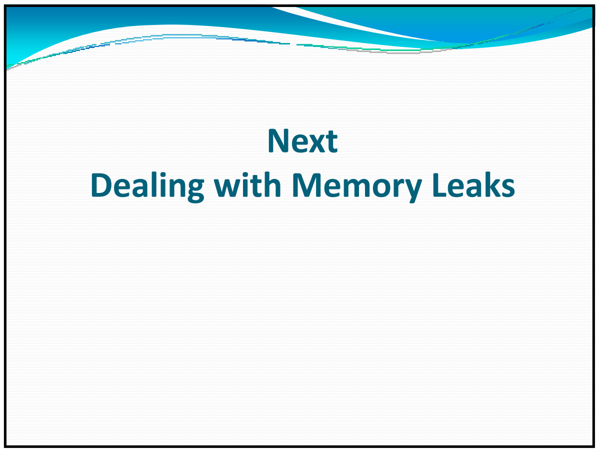
```
int main(int argc, char* argv[]){
   int x;
   printf("Please enter an integer : ");
   scanf("%d",x);
   printf("the integer entered was %d \n", x);
   return EXIT_SUCCESS;
}
```

```
int main(int argc, char* argv[]){
   printf("%ld \n", INT_MAX);
   int n = INT_MAX;
   int A[n];
   int i = 0;
   while (i<n)
     A[i] = rand()%10;

  return EXIT_SUCCESS;
}
```

```
int main(int argc, char* argv[]){
   FILE* fp = fopen("argv[1]", "r");
   char* word;
   while (fscanf(fp,"%s",word)>0)
       {   }
   return 0;
}
```

C



Next
Dealing with Memory Leaks