CIS 607:

Unix and C/C++

Structs,
File I/O,
Finish up memory overview
Talk about gdb / valgrind
• Enums make your own type
  – Type is “list of key words”
• Enums are useful for code clarity
  – Always possible to do the same thing with integers
• Be careful with enums
  – ... you can “contaminate” a bunch of useful words
C keyword “enum” – means enum definition is coming

```c
enum StudentType {
    HighSchool, Freshman, Sophomore, Junior, Senior, GradStudent
};
```

This enum contains 6 different student types

semi-colon!!!
enum example

int AverageAge(enum StudentType st)
{
    if (st == HighSchool)
        return 16;
    if (st == Freshman)
        return 18;
    if (st == Sophomore)
        return 19;
    if (st == Junior)
        return 21;
    if (st == Senior)
        return 23;
    if (st == GradStudent)
        return 26;

    return -1;
}
enums translate to integers ... and you can set their range

128-223-223-72-wireless:330 hank$ cat enum2.c
#include <stdio.h>

enum StudentType
{
    HighSchool = 105,
    Freshman,
    Sophomore,
    Junior,
    Senior,
    GradStudent
};

int main()
{
    printf("HighSchool = %d, GradStudent = %d\n", HighSchool, GradStudent);
}

128-223-223-72-wireless:330 hank$ gcc enum2.c
128-223-223-72-wireless:330 hank$ ./a.out
HighSchool = 105, GradStudent = 110
But enums can be easier to maintain than integers

```java
enum StudentType {
    HighSchool,
    Freshman,
    Sophomore,
    Junior,
    Senior,
    PostBacc,
    GradStudent
};
```

```java
int AverageAge(enum StudentType st) {
    if (st == HighSchool)
        return 16;
    if (st == Freshman)
        return 18;
    if (st == Sophomore)
        return 19;
    if (st == Junior)
        return 21;
    if (st == Senior)
        return 23;
    if (st == PostBacc)
        return 24;
    if (st == GradStudent)
        return 26;

    return -1;
}
```

If you had used integers, then this is a bigger change and likely to lead to bugs.
Structs, typedef, union
Simple Data Types

- float
- double
- int
- char
- unsigned char

All of these are simple data types
Structs: a complex data type

• Structs: mechanism provided by C programming language to define a group of variables
  – Variables must be grouped together in contiguous memory

• Also makes accessing variables easier ... they are all part of the same grouping (the struct)
struct syntax

C keyword "struct" – means struct definition is coming

```
struct Ray {
    double origin[3];
    double direction[3];
};
```

This struct contains 6 doubles, meaning it is 48 bytes

semi-colon!!!

```
int main() {
    struct Ray r;
    r.origin[0] = 0;
    r.origin[1] = 0;
    r.origin[2] = 0;
    r.direction[0] = 1;
    r.direction[1] = 0;
    r.direction[2] = 0;
}
```

Declaring an instance

“.” accesses data members for a struct
Nested structs

```c
struct Origin
{
    double originX;
    double originY;
    double originZ;
};

struct Direction
{
    double directionX;
    double directionY;
    double directionZ;
};

struct Ray
{
    struct Origin ori;
    struct Direction dir;
};

int main()
{
    struct Ray r;
    r.ori.originX = 0;
    r.ori.originY = 0;
    r.ori.originZ = 0;
    r.dir.directionX = 0;
    r.dir.directionY = 0;
    r.dir.directionZ = 0;
}
```

- access dir part of Ray
- accesses directionZ part of Direction (part of Ray)
typedef

• typedef: tell compiler you want to define a new type

```c
typedef struct Ray
{
    double origin[3];
    double direction[3];
} Ray;
```

```c
int main()
{
    struct Ray r;
    r.origin[0] = 0;
    r.origin[1] = 0;
    r.origin[2] = 0;
    r.direction[0] = 1;
    r.direction[1] = 0;
    r.direction[2] = 0;
}
```

saves you from having to type “struct” every time you declare a struct.
Other uses for typedef

• Declare a new type for code clarity
  – typedef int MilesPerHour;
    • Makes a new type called MilesPerHour.
    • MilesPerHour works exactly like an int.

• Also used for enums & unions
  – same trick as for structs ... typedef saves you a word
  – Note: enums discussed in lab, unions discussed next
So important: struct data member access is different with pointers

```c
typedef struct
{
    double origin[3];
    double direction[3];
} Ray;

int main()
{
    Ray r;
    r.origin[0] = 0;
    r.origin[1] = 0;
    r.origin[2] = 0;
    r.direction[0] = 1;
    r.direction[1] = 0;
    r.direction[2] = 0;
}
```

```c
typedef struct
{
    double origin[3];
    double direction[3];
} Ray;

int main()
{
    Ray *r = malloc(sizeof(Ray));
    r->origin[0] = 0;
    r->origin[1] = 0;
    r->origin[2] = 0;
    r->direction[0] = 1;
    r->direction[1] = 1;
    r->direction[2] = 1;
}
```

Pointers: use “->”
Instances (i.e., not pointers): use “.”
File I/O
File I/O: streams and file descriptors

- Two ways to access files:
  - File descriptors:
    - Lower level interface to files and devices
      - Provides controls to specific devices
    - Type: small integers (typically 20 total)
  - Streams:
    - Higher level interface to files and devices
      - Provides uniform interface; easy to deal with, but less powerful
    - Type: FILE *

Streams are more portable, and more accessible to beginning programmers. (I teach streams here.)
File I/O

• Process for reading or writing
  – Open a file
    • Tells Unix you intend to do file I/O
    • Function returns a “FILE *
      – Used to identify the file from this point forward
    • Checks to see if permissions are valid
  – Read from the file / write to the file
  – Close the file
Opening a file

- FILE *handle = fopen(filename, mode);

The argument mode points to a string beginning with one of the following sequences (Additional characters may follow these sequences.):

```
``'
r'' Open text file for reading. The stream is positioned at the beginning of the file.
``'
r+'' Open for reading and writing. The stream is positioned at the beginning of the file.
``'
a+'' Open for reading and writing. The file is created if it does not exist, otherwise it is truncated. The stream is positioned at the beginning of the file.
```

Example: FILE *h = fopen("/tmp/212", "wb");

Close when you are done with “fclose”

Note: #include <stdio.h>
Reading / Writing

FREAD(3) BSD Library Functions Manual FREAD(3)

NAME
fread, fwrite — binary stream input/output

LIBRARY
Standard C Library (libc, -lc)

SYNOPSIS
#include <stdio.h>

size_t
fread(void *restrict ptr, size_t size, size_t nitems, FILE *restrict stream);

size_t
fwrite(const void *restrict ptr, size_t size, size_t nitems,
      FILE *restrict stream);

DESCRIPTION
The function fread() reads nitems objects, each size bytes long, from the stream
pointed to by stream, storing them at the location given by ptr.

The function fwrite() writes nitems objects, each size bytes long, to the stream
pointed to by stream, obtaining them from the location given by ptr.

RETURN VALUES
The functions fread() and fwrite() advance the file position indicator for the
stream by the number of bytes read or written. They return the number of objects
read or written. If an error occurs, or the end-of-file is reached, the return
value is a short object count (or zero).
Example

```c
C02LN00GFD58:330 hank$ cat rw.c
#include <stdio.h>
#include <string.h>

int main(int argc, char *argv[])
{
    char *hello = "hello world: file edition\n";
    FILE *f = fopen("330", "w");
    fwrite(hello, sizeof(char), strlen(hello), f);
    fclose(f);
}
C02LN00GFD58:330 hank$ gcc rw.c
C02LN00GFD58:330 hank$ ./a.out
C02LN00GFD58:330 hank$ cat 330
hello world: file edition
```
File Position Indicator

• File position indicator: the current location in the file

• If I read one byte, the one byte you get is where the file position indicator is pointing.
  – And the file position indicator updates to point at the next byte
  – But it can be changed...
The `fseek()` function sets the file position indicator for the stream pointed to by `stream`. The new position, measured in bytes, is obtained by adding `offset` bytes to the position specified by `whence`. If `whence` is set to SEEK_SET, SEEK_CUR, or SEEK_END, the offset is relative to the start of the file, the current position indicator, or end-of-file, respectively. A successful call to the `fseek()` function clears the end-of-file indicator for the stream and undoes any effects of the ungetc(3) and ungetwc(3) functions on the same stream.
The `ftell()` function obtains the current value of the file position indicator for the stream pointed to by `stream`. 

```c
long
ftell(FILE *stream);
```
We have everything we need to make a copy command...

- fopen
- fread
- fwrite
- fseek
- ftell

Can we do this together as a class?
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(int argc, char *argv[])
{
    FILE *f_in, *f_out;
    int buff_size;
    char *buffer;

    if (argc != 3)
    {
        printf("Usage: %s <file1> <file2>\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    f_in = fopen(argv[1], "r");
    fseek(f_in, 0, SEEK_END);
    buff_size = ftell(f_in);
    fseek(f_in, 0, SEEK_SET);

    buffer = malloc(buff_size);
    fread(buffer, sizeof(char), buff_size, f_in);

    printf("Copying %d bytes from %s to %s\n", buff_size, argv[1], argv[2]);

    f_out = fopen(argv[2], "w");
    fwrite(buffer, sizeof(char), buff_size, f_out);

    fclose(f_in);
    fclose(f_out);

    return 0;
}
Return values in shells

```
C02LN00GFD58:330 hank$ ./a.out copy.c copy2.c
Copying 697 bytes from copy.c to copy2.c
C02LN00GFD58:330 hank$ echo $?
0
C02LN00GFD58:330 hank$ ./a.out copy.c
Usage: ./a.out <file1> <file2>
C02LN00GFD58:330 hank$ echo $?
1
```

$? is the return value of the last executed command
Printing to terminal and reading from terminal

• In Unix, printing to terminal and reading from terminal is done with file I/O
• Keyboard and screen are files in the file system!
  – (at least they were ...
Standard Streams

• Wikipedia: “preconnected input and output channels between a computer program and its environment (typically a text terminal) when it begins execution”

• Three standard streams:
  – stdin (standard input)
  – stdout (standard output)
  – stderr (standard error)

What mechanisms in C allow you to access standard streams?
printf

• Print to stdout
  – printf("hello world\n");
  – printf("Integers are like this %d\n", 6);
  – printf("Two floats: %f, %f", 3.5, 7.0);
fprintf

• Just like printf, but to streams
• `fprintf(stdout, “helloworld\n”);`
  – same as printf
• `fprintf(stderr, “helloworld\n”);`
  – prints to “standard error”
• `fprintf(f_out, “helloworld\n”);`
  – prints to the file pointed to by FILE *f_out.
buffering and printf

• Important: printf is buffered
• So:
  – printf puts string in buffer
  – other things happen
  – buffer is eventually printed
• But what about a crash?
  – printf puts string in buffer
  – other things happen ... including a crash
  – buffer is never printed!

Solutions: (1) fflush, (2) fprintf(stderr) always flushed
DRAM vs NV-RAM

- **DRAM**: Dynamic Random Access Memory
  - stores data
  - each bit in separate capacitor within integrated circuit
  - loses charge over time and must be refreshed
  - \( \rightarrow \) volatile memory

- **NV-RAM**: Non-Volatile Random Access Memory
  - stores data
  - information unaffected by power cycle
  - examples: Read-Only Memory (ROM), flash, hard drive, floppy drive, ...
Seagate Expansion 5TB Desktop External Hard Drive USB 3.0 (STEB5000100) by Seagate
$133.99 $169.99
Get it by Friday, Nov 20
More Buying Choices
$133.99 new (68 offers)
$117.24 used (1 offer)

Crucial Ballistix Sport 16GB Kit (8GBx2) DDR3 1600 MT/s (PC3-12800) UDIMM Memory BLS2KIT8G3D1609DS1S00/ BLS2CP8G3D1609DS1S00 by Crucial
$74.99 $150.99
Get it by Thursday, Nov 19
More Buying Choices
$69.95 new (73 offers)

Corsair Vengeance 16GB (2x8GB) DDR3 1600 MHz (PC3 12800) Desktop Memory (CMZ16GX3M2A1600C10) by Corsair
$83.90 $149.79
Get it by Thursday, Nov 19
More Buying Choices
$72.50 new (101 offers)
$74.99 used (3 offers)

Crucial 16GB Kit (8GBx2) DDR3/DDR3L-1600 MHz (PC3-12800) CL11 204-Pin SODIMM Memory for Mac CT2K8G3S160BM / CT2C8G3S160BM by Crucial
$72.99 $169.99
Get it by Thursday, Nov 19
More Buying Choices
$71.29 new (99 offers)
$62.00 used (8 offers)
Relationship to File Systems

• File Systems could be implemented in DRAM.
• However, almost exclusively on NV-RAM
  – Most often hard drives
• Therefore, properties and benefits of file systems are often associated with properties and benefits of NV-RAM.
## DRAM vs NV-RAM properties

<table>
<thead>
<tr>
<th>Property</th>
<th>DRAM</th>
<th>NV-RAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Capacity</td>
<td>~10GB</td>
<td>~10TB</td>
</tr>
<tr>
<td>Cost</td>
<td>$5/GB</td>
<td>$0.03/GB</td>
</tr>
<tr>
<td>Latency</td>
<td>&lt;100 nanoseconds</td>
<td>~10 milliseconds</td>
</tr>
</tbody>
</table>

### Distance:
A 20” map of Oregon is 1:100,000 scale.

### Time:
- 1 second to 27 hours is 1:100,000 scale
- 1 minute to 69 days is 1:100,000 scale
- 1 hour to 11 years is 1:100,000 scale
- 1 day to 273 years is 1:100,000 scale
More on Memory
Memory Segments

• Von Neumann architecture: one memory space, for both instructions and data
• → so break memory into “segments”
  – ... creates boundaries to prevent confusion
• 4 segments:
  – Code segment
  – Data segment
  – Stack segment
  – Heap segment
Code Segment

• Contains assembly code instructions
• Also called text segment
• This segment is modify-able, but that’s a bad idea
  – “Self-modifying code”
    • Typically ends in a bad state very quickly.
Data Segment

• Contains data not associated with heap or stack
  – global variables
  – statics (to be discussed later)
  – character strings you’ve compiled in
    char *str = "hello world\n"
Stack: data structure for collection

• A stack contains things
• It has only two methods: push and pop
  – Push puts something onto the stack
  – Pop returns the most recently pushed item (and removes that item from the stack)
• LIFO: last in, first out

Imagine a stack of trays.
You can place on top (push).
Or take one off the top (pop).
Stack

- Stack: memory set aside as scratch space for program execution
- When a function has local variables, it uses this memory.
  - When you exit the function, the memory is lost
Stack

• The stack grows as you enter functions, and shrinks as you exit functions.
  – This can be done on a per variable basis, but the compiler typically does a grouping.
    • Some exceptions (discussed later)

• Don’t have to manage memory: allocated and freed automatically
Heap

• Heap (data structure): tree-based data structure

• Heap (memory): area of computer memory that requires explicit management (malloc, free).

• Memory from the heap is accessible any time, by any function.
  – Contrasts with the stack
Memory Segments

- text (fixed size)
- data (fixed size)
- stack | growth
- free
- heap | growth
## Stack vs Heap: Pros and Cons

<table>
<thead>
<tr>
<th></th>
<th>Stack</th>
<th>Heap</th>
</tr>
</thead>
<tbody>
<tr>
<td>Allocation/Deal location</td>
<td>Automatic</td>
<td>Explicit</td>
</tr>
</tbody>
</table>
How stack memory is allocated into Stack Memory Segment

```java
void foo()
{
    int stack_varA;
    int stack_varB;
}

int main()
{
    int stack_varC;
    int stack_varD;
    foo();
}
```
How stack memory is allocated into Stack Memory Segment

```c
void foo()
{
    int stack_varA;
    int stack_varB;
}

int main()
{
    int stack_varC;
    int stack_varD;
    foo();
}
```
How stack memory is allocated into Stack Memory Segment

```c
void foo()
{
    int stack_varA;
    int stack_varB;
}

int main()
{
    int stack_varC;
    int stack_varD;
    foo();
}
```
How stack memory is allocated into Stack Memory Segment

```c
void foo()
{
    int stack_varA;
    int stack_varB;
}

int main()
{
    int stack_varC;
    int stack_varD;
    foo();
}
```
How stack memory is allocated into Stack Memory Segment

```c
int doubler(int A)
{
    int stack_varA;
    stack_varA = 2*A;
    return stack_varA;
}

int main()
{
    int stack_varC;
    int stack_varD = 3;
    stack_varC = doubler(stack_varD);
}```
How stack memory is allocated into Stack Memory Segment

```c
int doubler(int A)
{
    int stack_varA;
    stack_varA = 2*A;
    return stack_varA;
}

int main()
{
    int stack_varC;
    int stack_varD = 3;
    stack_varC = doubler(stack_varD);
}
```
int doubler(int A) {
    int stack_varA;
    stack_varA = 2*A;
    return stack_varA;
}

int main() {
    int stack_varC;
    int stack_varD = 3;
    stack_varC = doubler(stack_varD);
}
How stack memory is allocated into Stack Memory Segment

```c
int doubler(int A) {
    int stack_varA;
    stack_varA = 2*A;
    return stack_varA;
}
int main() {
    int stack_varC;
    int stack_varD = 3;
    stack_varC = doubler(stack_varD);
}
```

Return copies into location specified by calling function.
How stack memory is allocated into Stack Memory Segment

```c
int doubler(int A)
{
    int stack_varA;
    stack_varA = 2*A;
    return stack_varA;
}

int main()
{
    int stack_varC;
    int stack_varD = 3;
    stack_varC = doubler(stack_varD);
}
```
This code is very problematic ... why?

```c
int *foo()
{
    int stack_varC[2] = { 0, 1 };
    return stack_varC;
}

int *bar()
{
    int stack_varD[2] = { 2, 3 };
    return stack_varD;
}

int main()
{
    int *stack_varA, *stack_varB;
    stack_varA = foo();
    stack_varB = bar();
    stack_varA[0] *= stack_varB[0];
}
```

foo and bar are returning addresses that are on the stack ... they could easily be overwritten (and bar’s stack_varD overwrites foo’s stack_varC in this program)
int main()
{
    int stack_varA;
    {
        int stack_varB = 3;
    }
}

1. **Code**
2. **Data**
3. **Stack**
   - stack_varA
4. **Free**
5. **Heap**
Nested Scope

```c
int main()
{
    int stack_varA;
    {
        int stack_varB = 3;
    }
}
```
You can create new scope within a function by adding '{' and '}'.

```c
int main()
{
    int stack_varA;
    {
        int stack_varB = 3;
    }
}
```
# Stack vs Heap: Pros and Cons

<table>
<thead>
<tr>
<th></th>
<th>Stack</th>
<th>Heap</th>
</tr>
</thead>
<tbody>
<tr>
<td>Allocation/Deal</td>
<td>Automatic</td>
<td>Explicit</td>
</tr>
<tr>
<td>location</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Access</td>
<td>Fast</td>
<td>Slower</td>
</tr>
</tbody>
</table>

Memory pages associated with stack are almost always immediately available.

Memory pages associated with heap may be located anywhere ... may be caching effects.
## Stack vs Heap: Pros and Cons

<table>
<thead>
<tr>
<th>Stack</th>
<th>Heap</th>
</tr>
</thead>
<tbody>
<tr>
<td>Allocation/Deal location</td>
<td>Automatic</td>
</tr>
<tr>
<td>Access</td>
<td>Fast</td>
</tr>
<tr>
<td>Variable scope</td>
<td>Limited</td>
</tr>
</tbody>
</table>
Variable scope: stack

```c
int *foo()
{
    int stack_varA[2] = { 0, 1 };
    return stack_varA;
}

int *bar()
{
    int *heap_varB;
    heap_varB = malloc(sizeof(int)*2);
    heap_varB[0] = 2;
    heap_varB[1] = 2;
    return heap_varB;
}

int main()
{
    int *stack_varA;
    int *stack_varB;
    stack_varA = foo(); /* problem */
    stack_varB = bar(); /* still good */
}
```

foo is bad code ... never return memory on the stack from a function

bar returned memory from heap

The calling function – i.e., the function that calls bar – must understand this and take responsibility for calling free.

If it doesn’t, then this is a “memory leak”.
Memory leaks

It is OK that we are using the heap ... that’s what it is there for

The problem is that we lost the references to the first 49 allocations on heap

The heap’s memory manager will not be able to re-claim them ... we have effectively limited the memory available to the program.

```c
{ int i;
  int stack_varA;
  for (i = 0 ; i < 50 ; i++)
    stack_varA = bar();
}
```
Running out of memory (stack)

```c
int endless_fun()
{
    endless_fun();
}

int main()
{
    endless_fun();
}
```

Stack overflow: when the stack runs into the heap. There is no protection for stack overflows. (Checking for it would require coordination with the heap’s memory manager on every function calls.)
Running out of memory (heap)

```c
int *heaps_o_fun()
{
    int *heap_A = malloc(sizeof(int)*1000000000);
    return heap_A;
}

int main()
{
    int *stack_A;
    stack_A = heaps_o_fun();
}
```

If the heap memory manager doesn’t have room to make an allocation, then malloc returns NULL .... a more graceful error scenario.
## Stack vs Heap: Pros and Cons

<table>
<thead>
<tr>
<th></th>
<th>Stack</th>
<th>Heap</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Allocation/Deallocation</strong></td>
<td>Automatic</td>
<td>Explicit</td>
</tr>
<tr>
<td><strong>Access</strong></td>
<td>Fast</td>
<td>Slower</td>
</tr>
<tr>
<td><strong>Variable scope</strong></td>
<td>Limited</td>
<td>Unlimited</td>
</tr>
<tr>
<td><strong>Fragmentation</strong></td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Memory Fragmentation

- Memory fragmentation: the memory allocated on the heap is spread out of the memory space, rather than being concentrated in a certain address space.
Memory Fragmentation

```c
int *bar()
{
    int *heap_varA;
    heap_varA = malloc(sizeof(int)*2);
    heap_varA[0] = 2;
    heap_varA[1] = 2;
    return heap_varA;
}

int main()
{
    int i;
    int stack_varA[50];
    for (i = 0 ; i < 50 ; i++)
        stack_varA[i] = bar();
    for (i = 0 ; i < 25 ; i++)
        free(stack_varA[i*2]);
}
```

Negative aspects of fragmentation?
(1) can’t make big allocations
(2) losing cache coherency
Even if there is lots of memory available, the memory manager can only accept your request if there is a big enough contiguous chunk.
## Stack vs Heap: Pros and Cons

<table>
<thead>
<tr>
<th></th>
<th>Stack</th>
<th>Heap</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Allocation/Deallocation</strong></td>
<td>Automatic</td>
<td>Explicit</td>
</tr>
<tr>
<td><strong>Access</strong></td>
<td>Fast</td>
<td>Slower</td>
</tr>
<tr>
<td><strong>Variable scope</strong></td>
<td>Limited</td>
<td>Unlimited</td>
</tr>
<tr>
<td><strong>Fragmentation</strong></td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Memory Errors

- Array bounds read

```java
int main()
{
    int var;
    int arr[3] = { 0, 1, 2 };
    var=arr[3];
}
```

- Array bounds write

```java
int main()
{
    int var = 2;
    int arr[3];
    arr[3]=var;
}
```
Memory Errors

• Free memory read / free memory write

```c
int main()
{
    int *var = malloc(sizeof(int)*2);
    var[0] = 0;
    var[1] = 2;
    free(var);
    var[0] = var[1];
}
```

When does this happen in real-world scenarios?
Memory Errors

• Freeing unallocated memory

```c
int main()
{
    int *var = malloc(sizeof(int)*2);
    var[0] = 0;
    var[1] = 2;
    free(var);
    free(var);
}
```

When does this happen in real-world scenarios?

Vocabulary: “dangling pointer”: pointer that points to memory that has already been freed.
Memory Errors

• Freeing non-heap memory

```c
int main()
{
    int var[2]
    var[0] = 0;
    var[1] = 2;
    free(var);
}
```

When does this happen in real-world scenarios?
Memory Errors

• NULL pointer read / write

```c
int main()
{
    char *str = NULL;
    printf(str);
    str[0] = 'H';
}
```

• NULL is never a valid location to read from or write to, and accessing them results in a “segmentation fault”
  – .... remember those memory segments?

When does this happen in real-world scenarios?
Memory Errors

• Uninitialized memory read

```c
int main()
{
    int *arr = malloc(sizeof(int)*10);
    int V2=arr[3];
}
```
SLIDES TO REVIEW ON YOUR OWN
Unix shells allows you to manipulate standard streams.

- “>” redirect output of program to a file
- Example:
  - `ls > output`
  - `echo “this is a file” > output2`
  - `cat file1 file2 > file3`
Unix shells allows you to manipulate standard streams.

- “<” redirect file to input of program
- Example:
  - python < myscript.py
    - Note: python quits when it reads a special character called EOF (End of File)
    - You can type this character by typing Ctrl-D
    - This is why Python quits when you type Ctrl-D
      - (many other programs too)
Unix shells allows you to manipulate standard streams.

• “>>” concatenate output of program to end of existing file
  – (or create file if it doesn’t exist)

• Example:
  – echo “I am starting the file” > file1
  – echo “I am adding to the file” >> file1
  – cat file1
    I am starting the file
    I am adding to the file
What’s happening here?

```
C02LN00GFD58:330  hank$  mkdir  tmp
C02LN00GFD58:330  hank$  cd  tmp
C02LN00GFD58:tmp  hank$  touch  f1
C02LN00GFD58:tmp  hank$  ls  f1  f2  >  out
ls:  f2:  No  such  file  or  directory
C02LN00GFD58:tmp  hank$  cat  out
f1
```

Ls is outputting its error messages to stderr
Redirecting stderr in a shell

```
C02LN00GFD58:Documents hank$ cd ~/330
C02LN00GFD58:330 hank$ mkdir tmp
C02LN00GFD58:330 hank$ cd tmp
C02LN00GFD58:tmp hank$ touch f1
C02LN00GFD58:tmp hank$ ls f1 f2 > out
ls: f2: No such file or directory
C02LN00GFD58:tmp hank$ cat out
f1
C02LN00GFD58:tmp hank$ ls f1 f2 > out 2>out_error
C02LN00GFD58:tmp hank$ cat out_error
ls: f2: No such file or directory
```
Redirecting stderr to stdout

Convenient when you want both to go to the same stream
Outline

• Review
• File I/O
• Project 2B
• Redirection
• Pipes
c functions: fork and pipe

• fork: duplicates current program into a separate instance
  – Two running programs!
  – Only differentiated by return value of fork (which is original and which is new)

• pipe: mechanism for connecting file descriptors between two forked programs

Through fork and pipe, you can connect two running programs. One writes to a file descriptor, and the other reads the output from its file descriptor.

Only used on special occasions. (And one of those occasions is with the shell.)
pipes in Unix shells

- represented with “|”
- output of one program becomes input to another program

```c
#include <stdio.h>
int main() { printf("Hello world\n"); } 
```
Very useful programs

- grep: keep lines that match pattern, discard lines that don’t match pattern

```bash
$ ls -l | grep ppt
```

```
-rw-r--r--@ 1 hank staff  3278589 Apr  5 11:40 CIS330_Lec2.pptx
-rw-r--r--@ 1 hank staff  2220104 Apr  8 20:57 CIS330_Lec3.pptx
-rw-r--r--@ 1 hank staff  3899863 Jan 21 09:26 CIS610_lec2.pptx
-rw-r--r--@ 1 hank staff  4629257 Jan 30 10:24 CIS610_lec3.pptx
-rw-r--r--@ 1 hank staff  21382185 Mar 25 12:40 CIS_colloquium2013.pptx
-rw-r--r--@ 1 hank staff  21382185 Jan  7 12:21 CIS_colloquium_2013.pptx
-rw-r--r--@ 1 hank staff  2172179 Dec 20 15:24 ICS_results.pptx
-rw-r--r--@ 1 hank staff  4841050 Nov 13 10:10 MBTI.pptx
-rw-r--r--@ 1 hank staff  2031749 Apr  5 16:20 SC14_flow.pptx
-rw-r--r--@ 1 hank staff  17972476 Mar 25 12:43 VMV_2013.pptx
-rw-r--r--@ 1 hank staff  98149068 Apr  1 10:25 aachen.pptx
-rw-r--r--@ 1 hank staff  9815146 Feb 24  07:00 childs_poster_SDAV_AHM_2014.pptx
-rw-r--r--@ 1 hank staff  592243 Feb 26 04:09 childs_sdaav_slides.pptx
-rw-r--r--@ 1 hank staff 15765504 Feb 13 14:57 cig_exascale.ppt
-rw-r--r--@ 1 hank staff 16699392 Jan  7 12:14 cis610_Lec1.ppt
-rw-r--r--@ 1 hank staff  3159872 Jan  7 11:15 egpgvCfg.pptx
-rw-r--r--@ 1 hank staff 15767552 Mar 23 02:48 eu_regional_school.ppt
-rw-r--r--@ 1 hank staff 35009136 Mar 25 09:42 eu_regional_school_part1.ppt
-rw-r--r--@ 1 hank staff 10775552 Mar 25 04:49 eu_regional_school_part1B.ppt
-rw-r--r--@ 1 hank staff  72966144 Mar 26 08:43 eu_regional_school_part2.ppt
-rw-r--r--@ 1 hank staff  7571317 Mar 25 12:53 ilm_booth_talk.pptx
```
Very useful programs

• **sed**: replace pattern 1 with pattern 2
  – `sed s/pattern1/pattern2/g`
    • `s` means substitute
    • `g` means “global” ... every instance on the line

**sed** is also available in “vi”
```bash
:%s/pattern1/pattern2/g (% means all lines)
:103,133s/p1/p2/g (lines 103-133)
```
## Wildcards

- ‘*’ is a wildcard with Unix shells

```bash
fawcett:tmp child$ ls
Abe       Chavarria  Hebb     Macy     Smith
Alajaji   Chen       Jia       Maguire  Steelhammer
Alamoudi  Clark      Kine      Michlanski Szczepanski
Anastas   Collier    Lee       Moreno   Totten
Andrade   Costello  Legge     Olson    Vega-Fujioka
Ballarche Donnelly  Li        Owen     Wang
Brennan   Etzel      Lin       Pogrebinsky Whiteley
Brockway  Friedrich  Liu       Qin      Woodruff
Brogan    Garvin     Lopes     Rhodes   Xu
Brooks    Gonzales   Luo       Roberts  Yaconelli
Bruce     Guo        Lynch     Rodriguez Young
Carlton   Hampton    Lyon      Roush     Zhang
Chalmers  Harris     Machado   Rozenboim de

fawcett:tmp child$ ls C*
Carlton   Chavarria  Clark    Costello
Chalmers  Chen       Collier

fawcett:tmp child$ ls *z
Rodriguez

fawcett:tmp child$ ls *ee*
Lee       Steelhammer

fawcett:tmp child$ ls *e*e*
Lee       Legge     Steelhammer  Whiteley
```

- ‘?’ is a wildcard that matches exactly one character
Other useful shell things

- ‘tab’: auto-complete
- esc=: show options for auto-complete
- Ctrl-A: go to beginning of line
- Ctrl-E: go to end of line
- Ctrl-R: search through history for command