A Brief Review of C
(and Beards)

Benjamin Brewster

A Brief History

- Developed by Dennis Ritchie (1941-2011) between 1969 and 1973 at Bell Labs

- C is a successor to B; however, B's inability to take advantage of the PDP-11's advanced features (to which computer Ritchie and Ken Thompson were busily porting UNIX) caused Ritchie to develop C

- UNIX was then re-written in C in 1972, which had been in development at the same time

UNIX Beard Comparison – Round 1

Dennis Ritchie – restrained, non-ironic

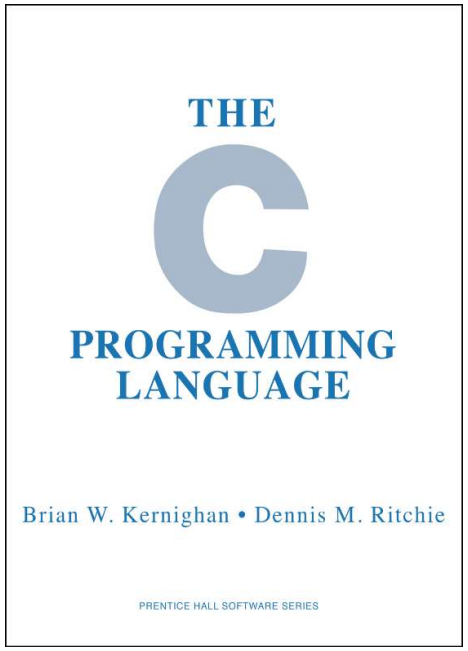
Richard Stallman – enough said

C is A High-Level Language

- As opposed to a low level language, like assembly
- The original version of C (C89) has 32 reserved keywords, and 50+ operators and syntax characters
- C syntax widely influences programming language syntax development today

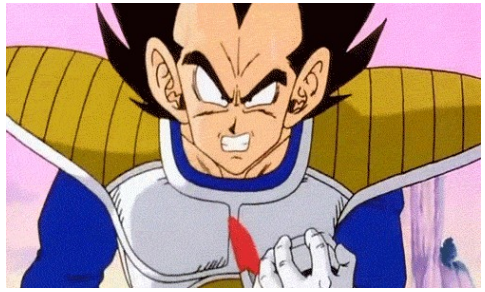
HELLO FREAKING WORLD

```
#include <stdio.h>

int main()
{
  printf("Hello world\n");
  return 0;
}
```



The first C book, written by Ritchie and Brian Kernighan, contains the first usage of a Hello World program put in book form

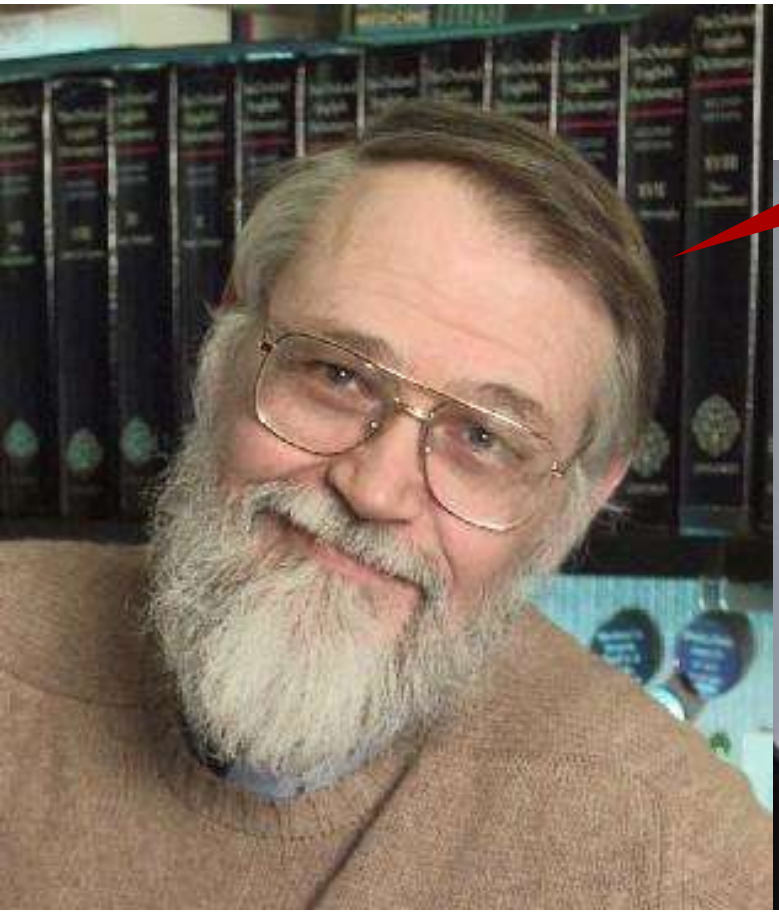
PART DEUX



```
#include <stdio.h>

int main()
{
    char* oblig = "Hello World";
    float itsOver = 9000.0f;
    printf("%s\n", oblig);
    printf("IT\'S OVER %.2f!\n", itsOver);

    return 0;
}
```

```
$ hw2
Hello World
IT'S OVER 9000.00!
```

UNIX Beard Comparison – Round 2



Brian Kernighan – shaped yet voluminous

Richard Stallman – enough said

Common String Shenanigans - Comparing

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

void main()
{
        char* boring = "boring";
        char* weirdDadSaying = "Eat more beef, kick less cats\n";
        int length;

        length = strlen(weirdDadSaying);
        printf("Length of entered string is = %d\n", length);

        if (strcmp(boring, weirdDadSaying) == 0)
                printf("Entered strings are equal.\n");
        else
                printf("Entered strings are not equal.\n");
}
```

```
$ stringshenanigans
Length of entered string is = 30
Entered strings are not equal.
```

Why Only Two Arguments? That's Weird Design

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

void main()
{
   char a[1000], b[1000];

   printf("Enter the first string\n");
   gets(a);

   printf("Enter the second string\n");
   gets(b);

   strcat(a, b);

   printf("String obtained on concatenation is %s\n", a);
 }
```

Function signature:
`char* gets(char *str)`

`strcat()` dumps the results into a and returns the same

Why Only Two Arguments? That's Weird Design

```
$ gcc -o getsstrcat getsstrcat.c
/tmp/ccJ1Kg0x.o: In function `main':
getsstrcat.c:(.text+0x20): warning: the `gets' function is dangerous and should not be used.
$ getsstrcat
Enter the first string
mystring!
Enter the second string
so col!!@
String obtained on concatenation is mystring!so col!!@
```

```
  printf("Enter the second string\n");
  gets(b);

  strcat(a, b);

  printf("String obtained on concatenation is %s\n", a);
}
```

`strcat()` dumps the results into a and returns the same

Substrings - Not Built-In!

```
#include <stdio.h>

void main()
{
   char string[1000], sub[1000];
   int position, length, c = 0;

   printf("Input a string\n");
   gets(string);

   printf("Enter the position of first char, a space, and length of substring\n");
   scanf("%d%d", &position, &length);

   while (c < length) {
      sub[c] = string[position + c - 1];
      c++;
   }
   sub[c] = '\0';

   printf("Required substring is \"%s\"\n", sub);
}
```

Substrings - Not Built-In!

```
$ gcc -o substrings substrings.c
/tmp/ccdSGmo9.o: In function `main':
substrings.c:(.text+0x27): warning: the `gets' function is dangerous and should not be used.
$ substrings
Input a string
test string!
Enter the position of first char, a space, and length of substring
2 6
Required substring is "est st"
```

```
   printf("Enter the position of first char, a space, and length of substring\n");
   scanf("%d%d", &position, &length);

   while (c < length) {
      sub[c] = string[position + c - 1];
      c++;
   }
   sub[c] = '\0';

   printf("Required substring is \"%s\"\n", sub);
}
```

Array Stuff

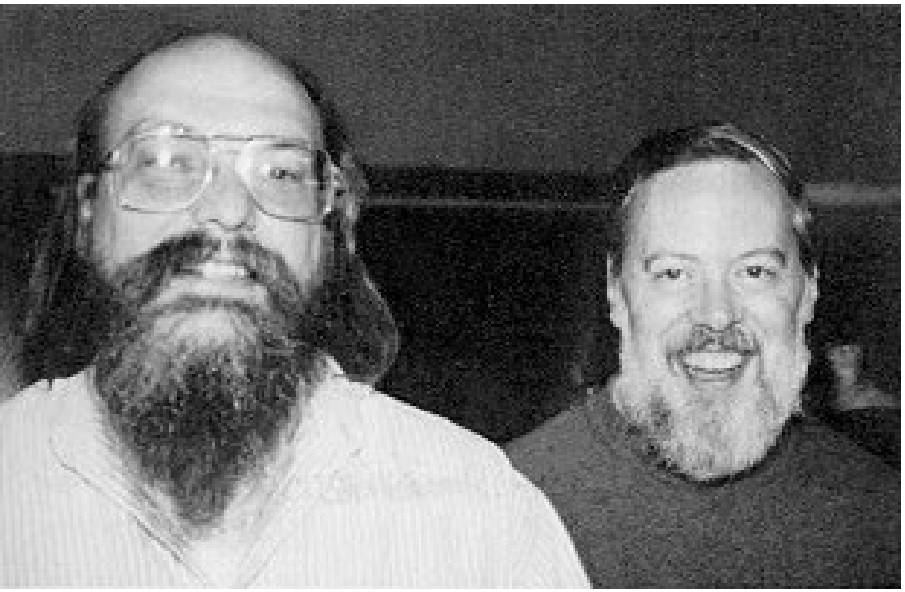
```
#include <stdio.h>

void main()
{
  int array[100], maximum, size, c, location = 1;

  printf("Enter the number of elements in array\n");
  scanf("%d", &size);

  printf("Enter %d integers\n", size);

  for (c = 0; c < size; c++)
    scanf("%d", &array[c]);

  maximum = array[0];

  for (c = 1; c < size; c++)
  {
    if (array[c] > maximum)
    {
       maximum  = array[c];
       location = c + 1;
    }
  }

  printf("Max element at location %d, value is %d.\n", location, maximum);
}
```

```
$ gcc -o arraystuff arraystuff.c
$ arraystuff
Enter the number of elements in array
5
Enter 5 integers
1 9 3 7 4
Max element at location 2, value is 9.
```

UNIX Beard Comparison – Round 3

Ken Thompson – Unrestrained, yet directed

Bonus Dennis! Does this guy know how to party!



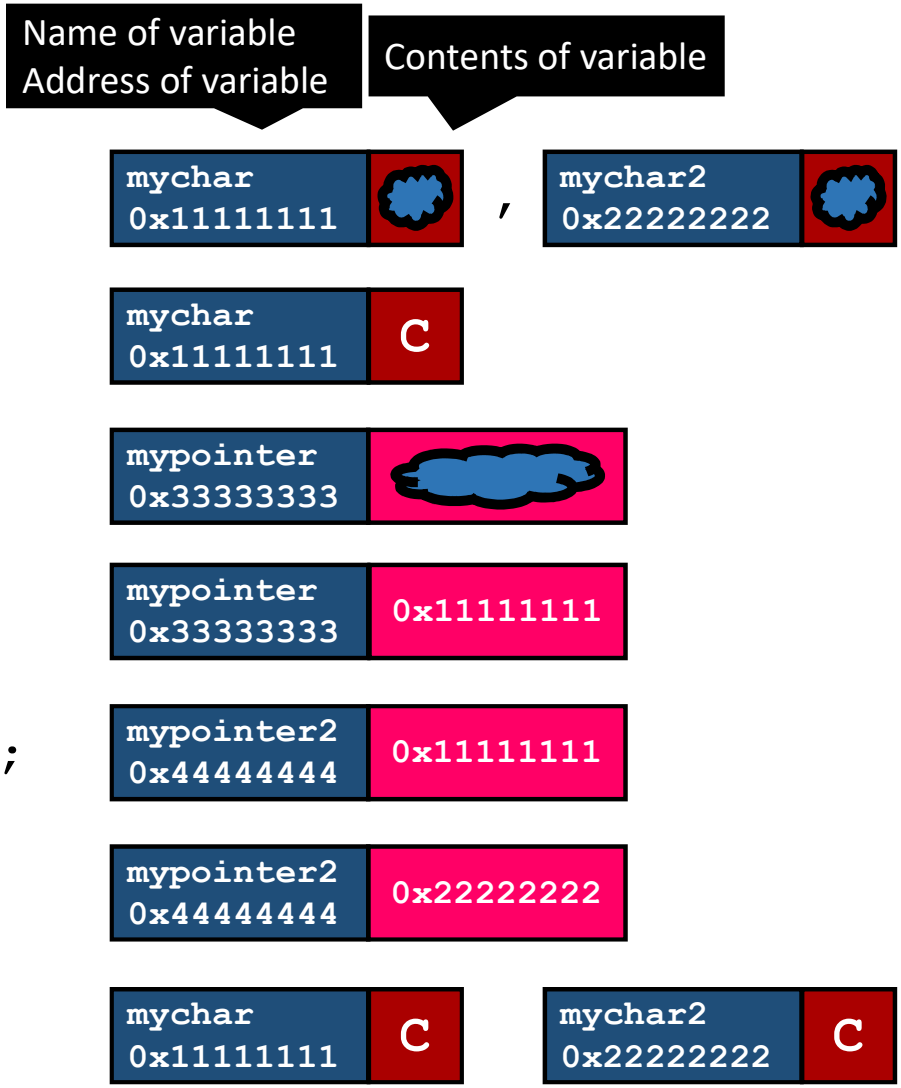
Gandalf the White

Richard Stallman

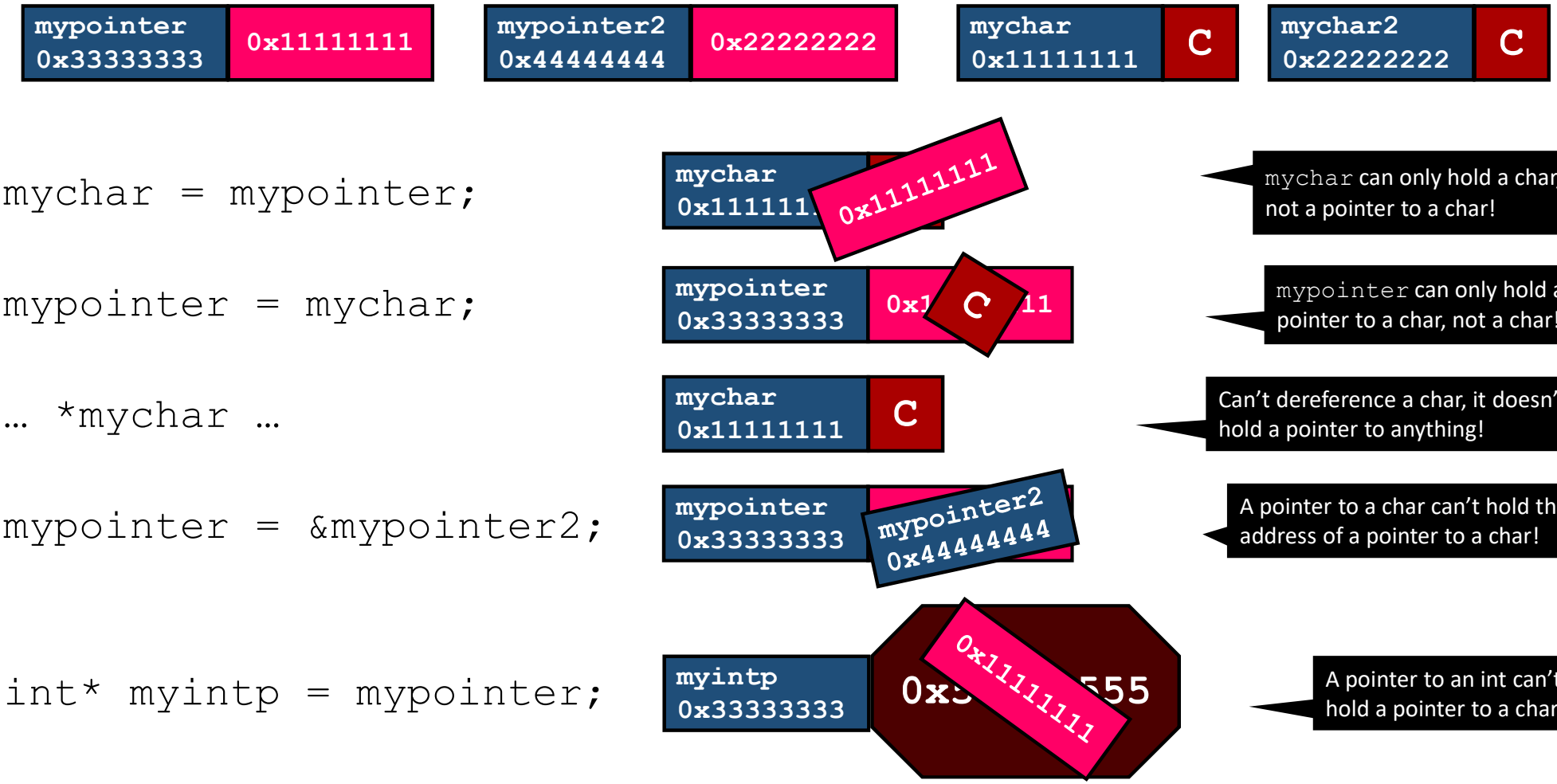
http://book.roomofthings.com/2012/05/15/enchanted_objects_IoT_for_humans

OH CRAP POINTERS

```
char mychar, mychar2;

mychar = 'C';

char* mypointer;

mypointer = &mychar;

char* mypointer2 = mypointer;

mypointer2 = &mychar2;

*mypointer2 = *mypointer;
```



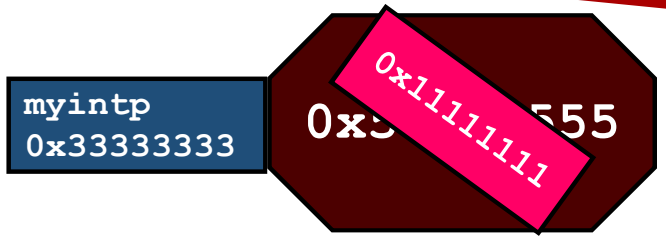
OH CRAP POINTERS - Illegal Commands

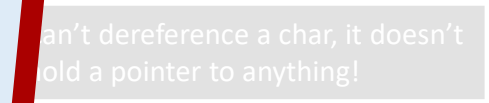


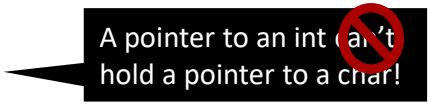
OH CRAP POINTERS - Illegal Commands


```
mychar = mypointer;

mypointer = mychar;

… *mychar …

mypointer = &mypointe

int* myintp = mypointer;
```

Except, C allows these four items, giving you a suitably dire **warning only** for each problem at compile time



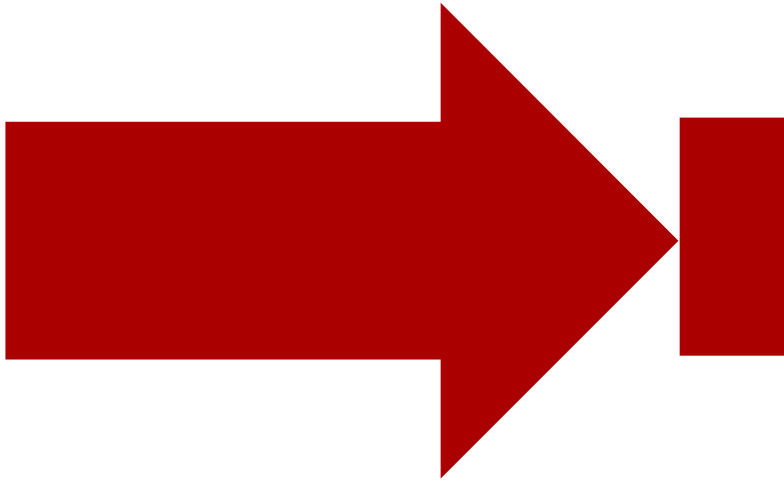




OH CRAP POINTERS

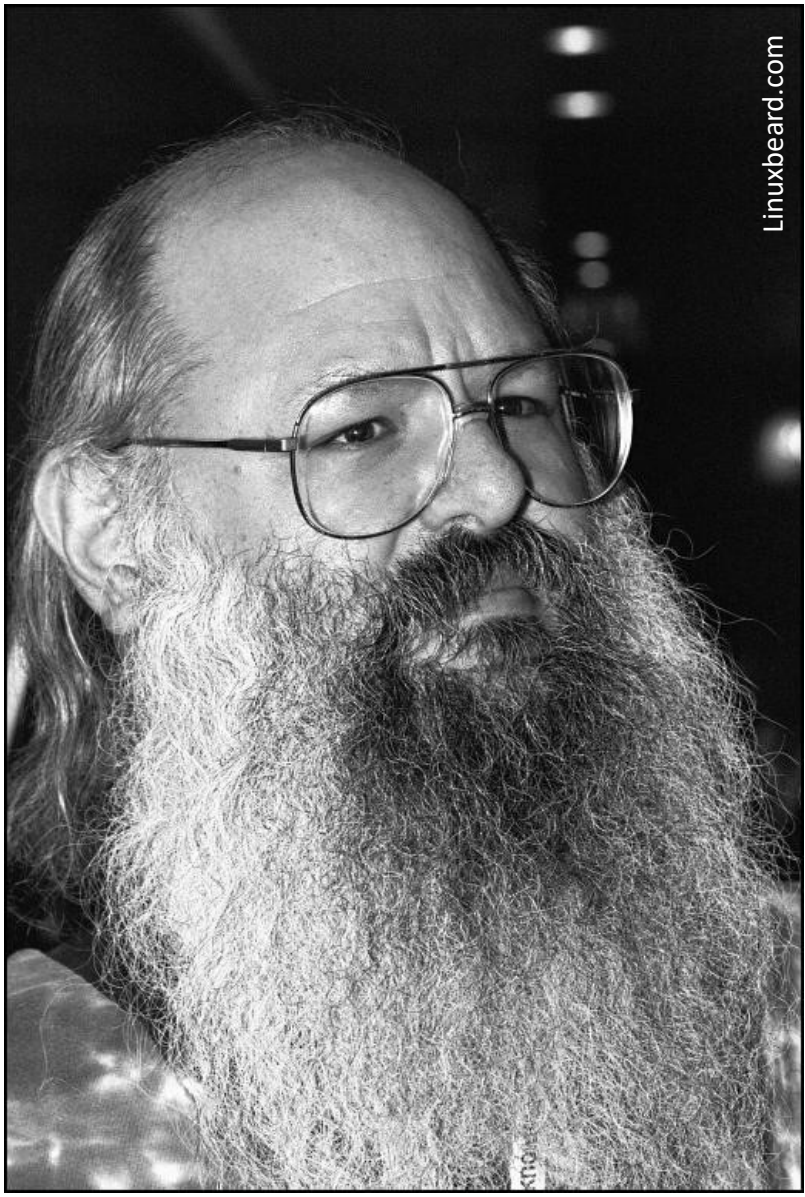
```
#include <stdio.h>

void CopyString(char* tgt, char* src)
{
        while (*src)
        {
                *tgt = *src;
                src++;
                tgt++;
        }

        *tgt = '\0';
}

void main()
{
        char target[1000];
        char* source = "COPY ME!";

        CopyString(target, source);
        printf("Target is: \'%s\'\n", target);
}
```

```
$ gcc -o ohcrappointers ohcrappointers.c
$ ohcrappointers
Target is: 'COPY ME!'
```



UNIX Beard Winner

Ed Gould
BSD Pioneer



From Programs to Execution

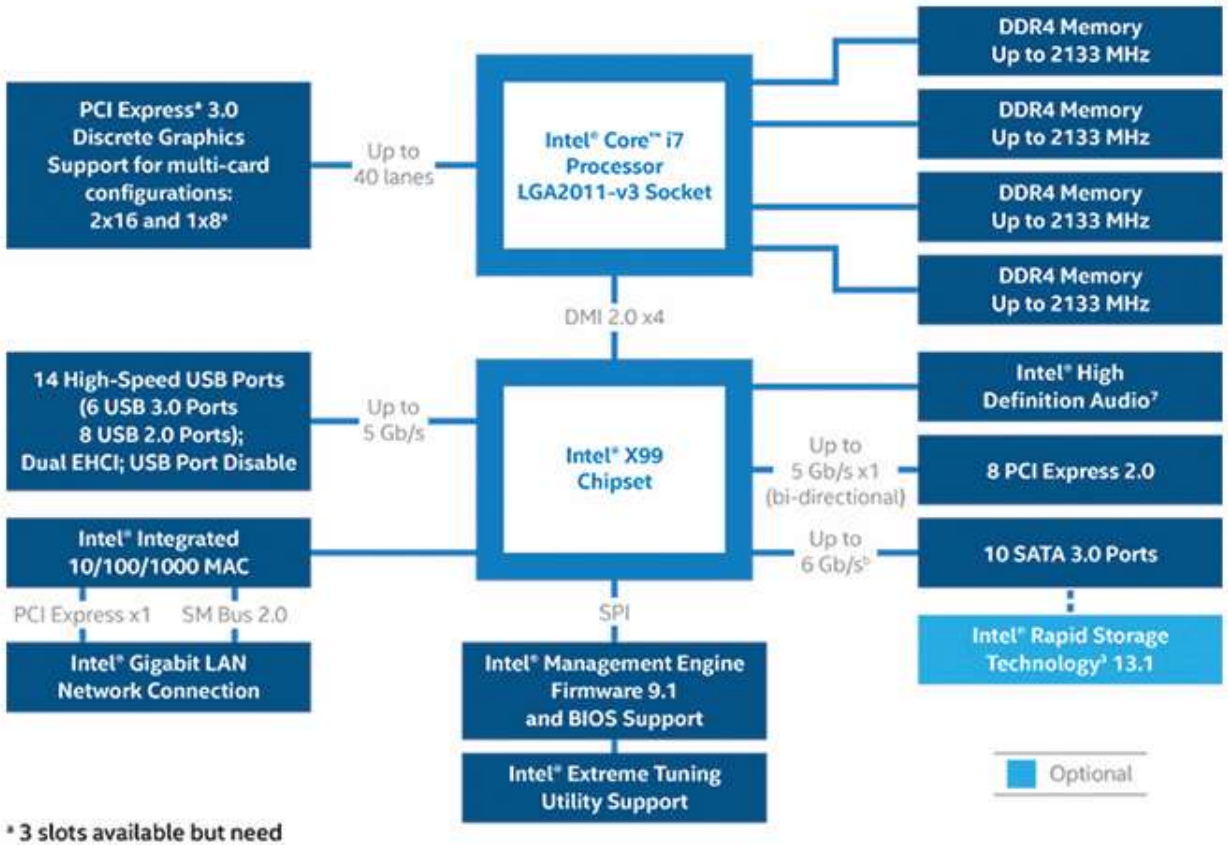
Benjamin Brewster

The Underlying Hardware

- The OS provides software access to the hardware in an *abstracted* manner
- What does that hardware actually look like?

In this case, this ancient 3rd party wireless(!) NES hardware looks pretty awesome

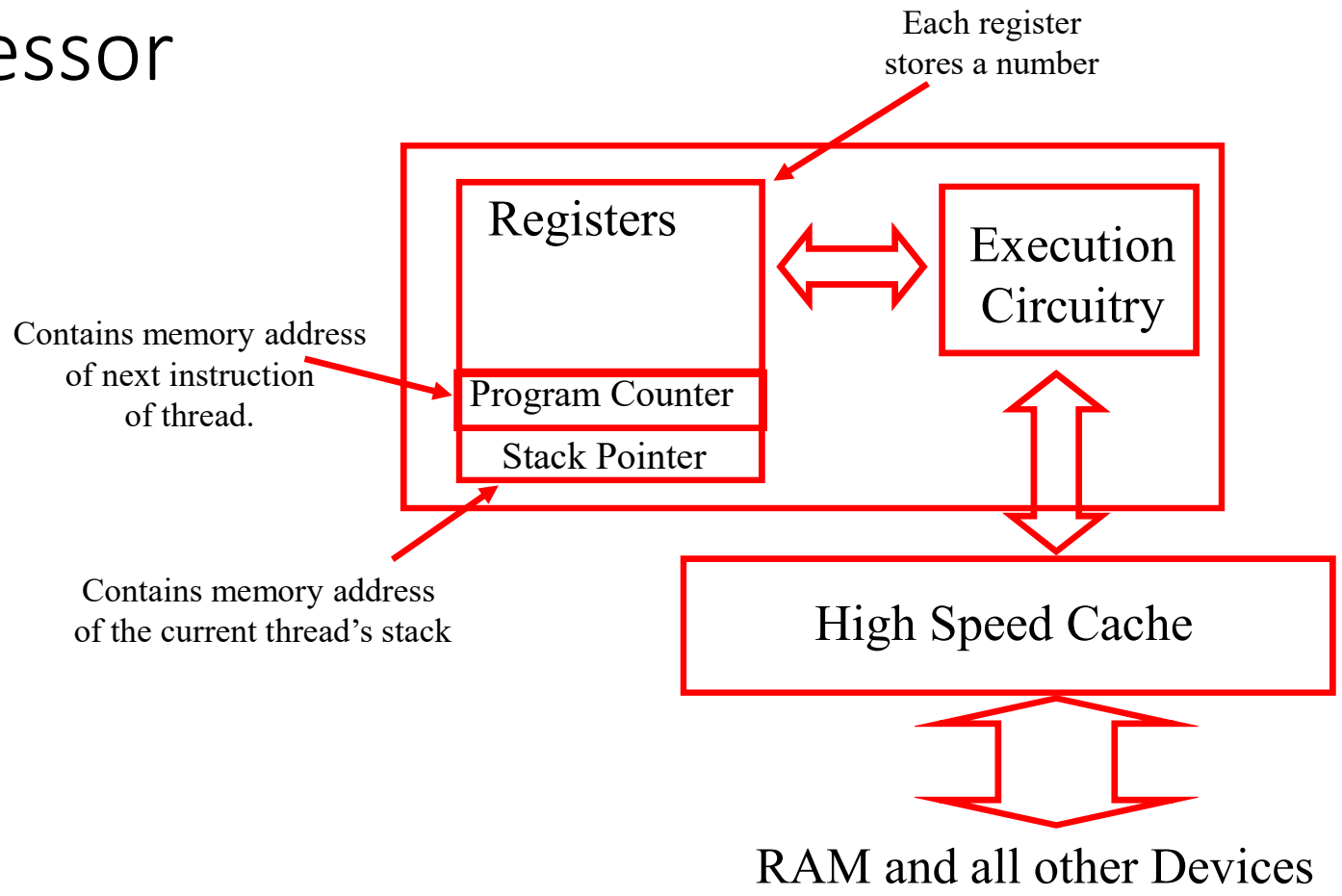
Intel® X99 Chipset Block Diagram



* 3 slots available but need additional logic onboard to support more slots. 5x8 configuration requires additional system clocks to be provided by third party components.

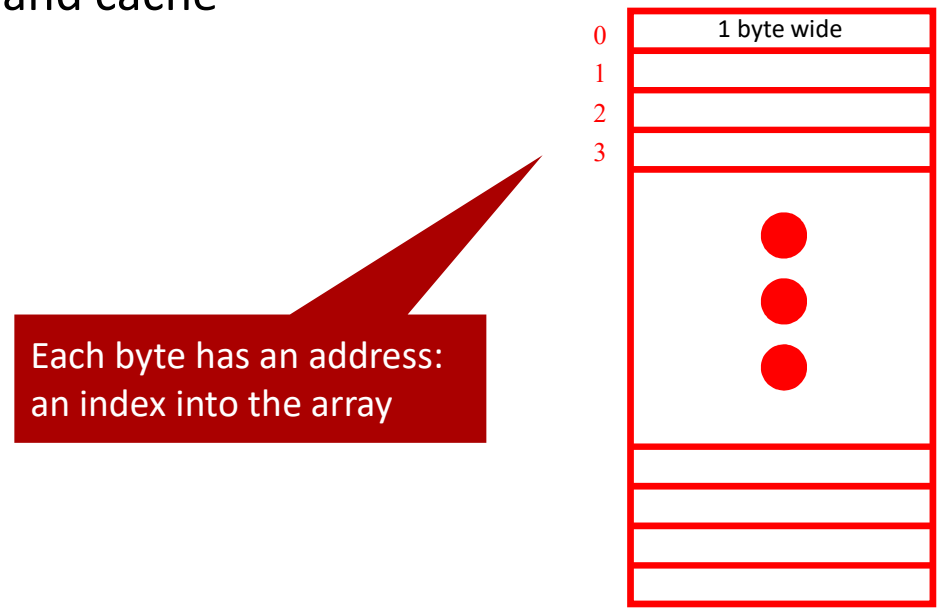
b All SATA ports capable of 6 Gb/s.

A Processor


Each register
stores a number
Registers
Execution
Circuitry
Contains memory address
of next instruction
of thread.
Program Counter
Stack Pointer
Contains memory address
of the current thread's stack
High Speed Cache
RAM and all other Devices

Memory

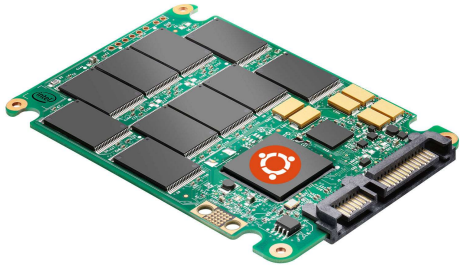
- Memory is an array of *bytes*
- Temporary storage only:
  - Much slower than the processor and cache
  - Much faster than a disk

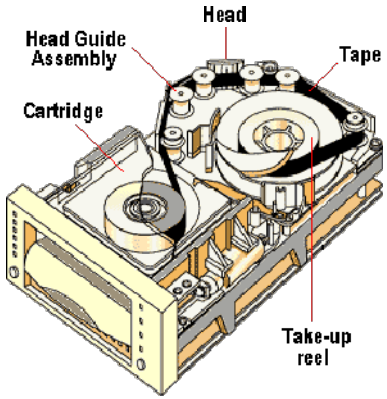


Other Storage Devices

- Persistent storage
  - Magnetic Hard Disk Drives (HDD)
  - Non-volatile memory
    - Solid State Drives (SSD)
    - Flash memory
  - Magnetic tapes
  - Optical (CD-ROM, DVDs, BD, etc.)

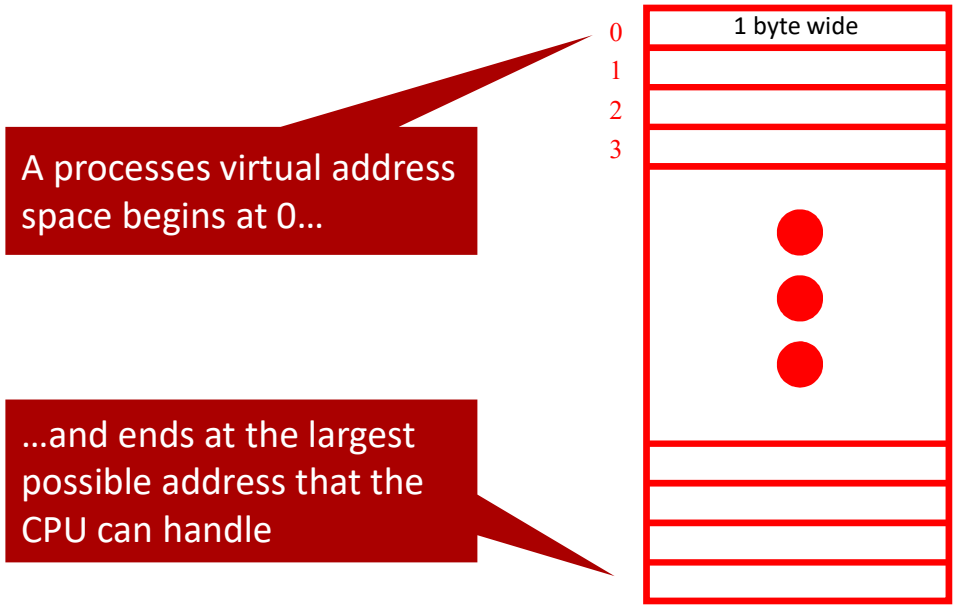
- All slower than RAM, but keep their memory without power



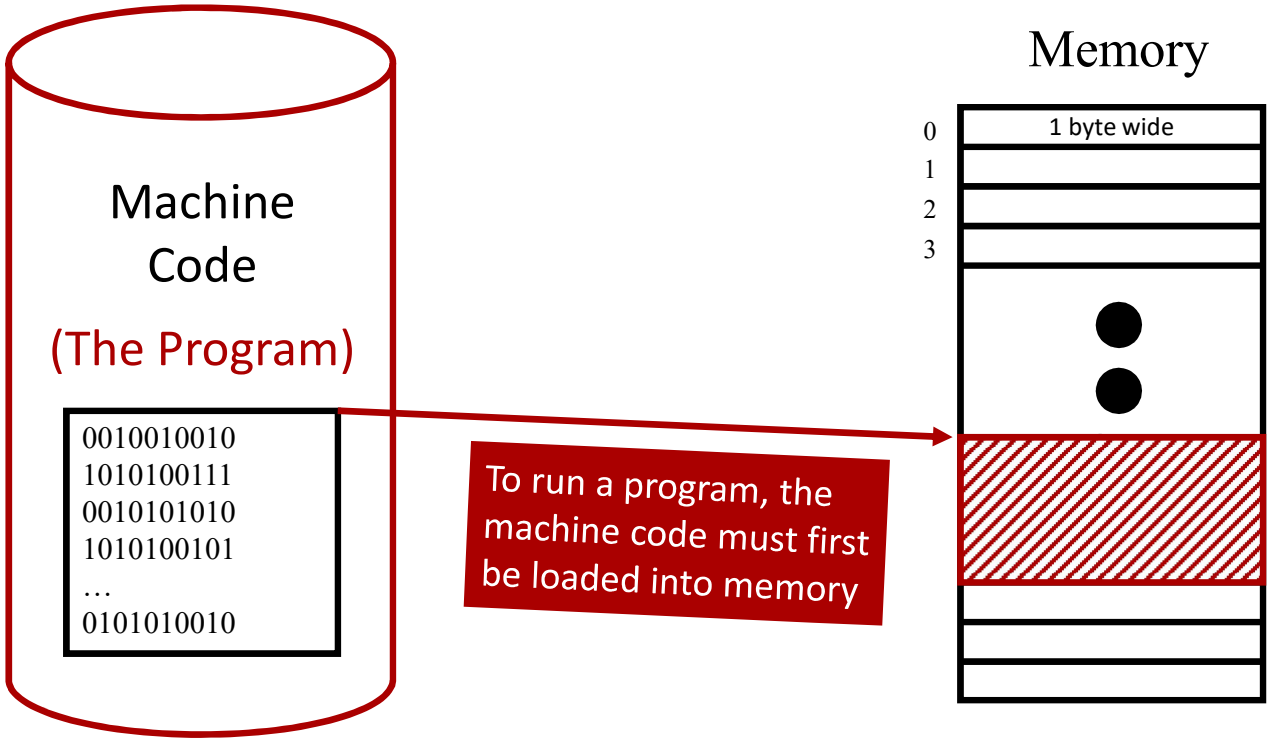


Virtual Memory

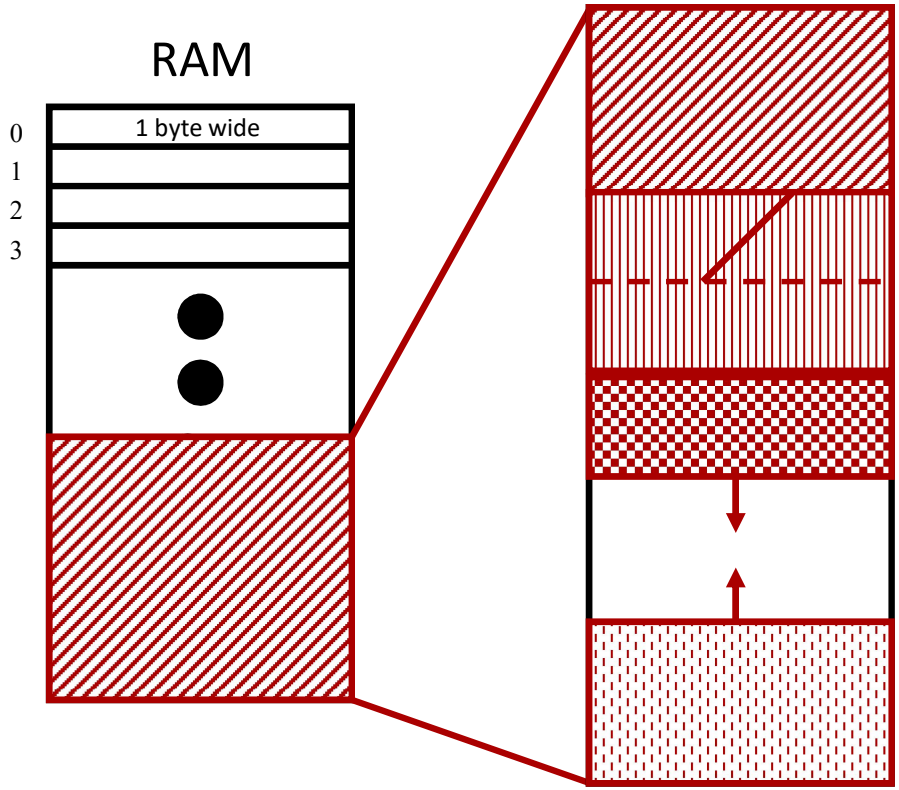
- Virtual memory hardware creates the illusion of:
  - Un-shared, exclusive memory
  - Unlimited memory (up to the maximum address size)



Running a Program



Typical Organization of Program in Memory



Code Segment - program instructions

Data Segment - Initialized global & static variables in read/write and read-only sections

Uninitialized global & static variables

Stack - automatic variables, function return pointers

Heap - dynamically allocated memory

Stack Versus Heap

The Stack

- Stores local automatic variables and function return pointers as the program enters and exits scoped blocks of code
- Memory managed efficiently by CPU
- Variable size is limited by OS settings
- Variables cannot be resized

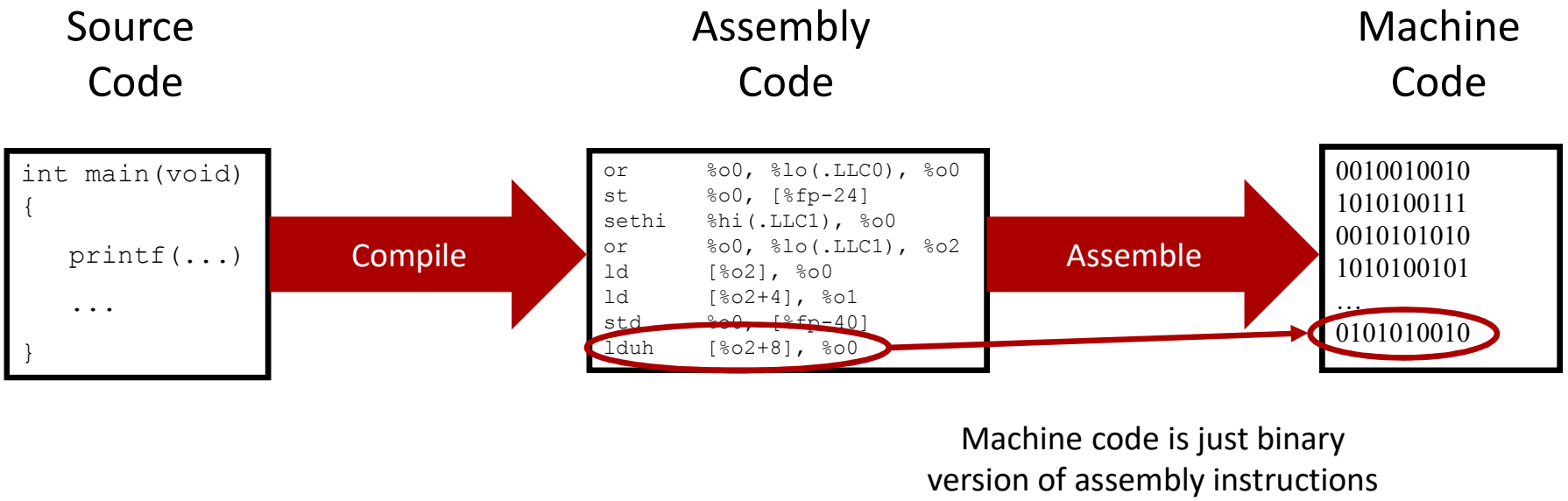
The Heap

- Variables are allocated manually (`malloc()`, `calloc()`)
- Memory is unmanaged, so fragmentation can occur; heap access is slower than stack
- Variable size is unlimited (other than virtual memory limits)
- Variables can be resized with `realloc()`

Creating The Program Code

- How do we turn a high-level program (C++, Java) into something that the computer can run?

Creating The Program Code – High Level



- By default, most compilers will compile, link, and assemble your source code, though you can split those steps up for more control

The compile/link process

1. The C pre-processor expands macros and strips out comments
   `#include`, `#define`, `#ifdef`, `//`, `/* */`, etc.
2. The compiler parses your source, checks for errors and generates assembly language code
3. The compiler calls the assembler, which converts assembly code to machine binary code
4. If you are compiling an executable, the linker step tries to match function calls to function code (they might be in different files!)

GCC - the Standard UNIX Compiler

- Basic compilation options

`-g`	Compile with debugging info for GDB
`-c`	Compiles only, without linking (more later)
`-S`	Generates assembly code
`-O3`	Optimizes as much as possible
`-o`	Specifies the *name* of the output file
`-Wall`	Turns on all warnings
`-l`*`library`*	Adds support for library *`library`* when linking (for example, `-lpthread`)

- These should work with any of the Unix CLI C/C++ compilers

Compiling an Executable

- If you have *one* source (.c) file:

```
$ gcc -o dbtest dbtest.c
```

Here I've used `dbtest`, instead of `test`. Why?

- If you have *multiple* source (.c) files:
  - Option A *(simpler)*: compile them all at once, together into one executable:

```
$ gcc -o dbtest dbtest.c dbcreate.c dbopen.c
```

Separate Compile and Link

- If you have *multiple* source (.c) files:
  - Option B *(more efficient)* : compile them one at a time without linking, then link them all together at the end

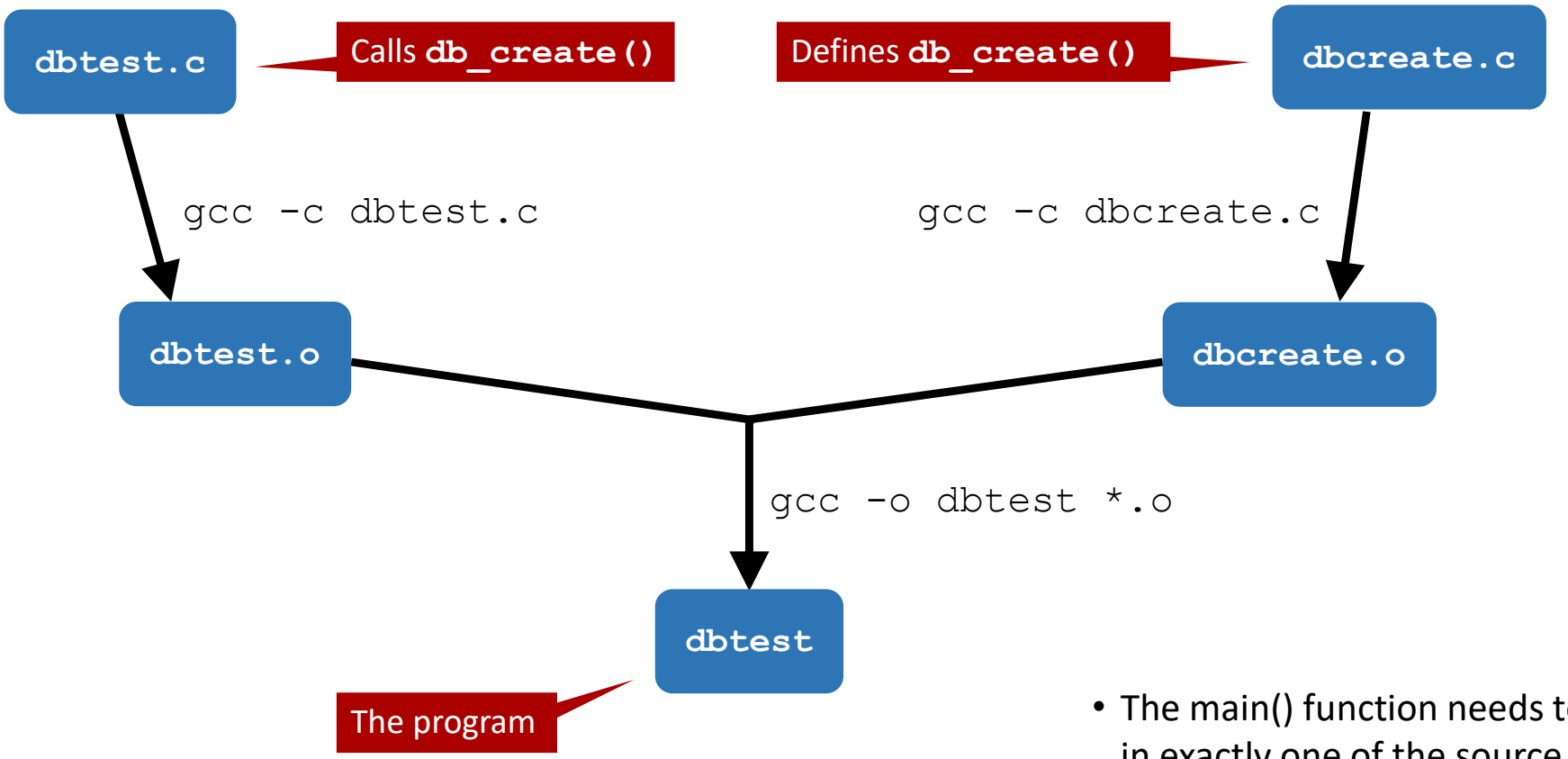
1. First compile all source files separately into object files (.o):

```
$ gcc -c dbtest.c
$ gcc -c dbcreate.c
$ gcc -c dbopen.c
$ gcc -c dbread.c
```

2. Link all the object (.o) files together to create an executable:

```
$ gcc -o dbtest dbtest.o dbcreate.o dbopen.o
```

Compile & Link



- The main() function needs to be in exactly one of the source files

Library Archives

- Library archives are collections of object files (.o) gathered into a single large file, with indexes to make accessing them fast
  - Usually faster than having to read every .o file
  - Easier to link with if you aren't changing the library object files frequently

- To create a library
  - First create all the object files (see previous slide)
  - Then use the `ar` command:

```
$ ar -r libdb.a dbcreate.o dbopen.o dbread.o
```

Using Library Archives

- Include the library anywhere you can use an object file:

```
$ gcc -o dbtest dbtest.o libdb.a
```

Hello World!

- A complete C compilation and execution example:

```
$ cat hw.c
#include <stdio.h>

void main()
{
        printf("Hello World!\n");
}
$ gcc -o hw hw.c
$ hw
Hello World!
```

Concurrency

Benjamin Brewster

Concurrency

- Concurrency (an informal term): doing multiple things at the same time
  - Program decomposition into order-independent chunks

- On UNIX concurrency is easy
  - Multiple processes *can* be running simultaneously
  - Multiple copies of the same program can be running
  - CPU time can be split and shared

- Concurrency is very powerful
  - Greatly increases the efficiency of an OS: while one process is waiting for I/O, another process can use the CPU

Definitions: System Calls vs. Library Functions

- System Calls
  - A request for service that causes the normal operation of a process to be *interrupted* and control passed to the OS
  - Typically, the process is now blocked and won't do anything else until the system call returns
  - read(), write(), etc.

- C Library Functions
  - Faster, as they have no permissions or blocking issues
  - sqrt(), printf(), etc.

Illusions of Simultaneous Execution

- Multiprogramming
  - More than one process can be ready to execute
  - System calls trigger “context switches”, which let the next process run
  - The process will not execute again until its system call returns

- Timesharing
  - CPU time split between multiple processes
    - Gives illusion that many processes are running at once

Processes can also communicate amongst themselves - more on this later

Multiprocessing

- Executing multiple processes at the actual same time is called **multiprocessing**

- Today's CPUs have 4, 6, 8, and 10 cores, with more coming
  - Each core acts like a mini-CPU, each of which can do **multiprogramming** AND **timesharing**

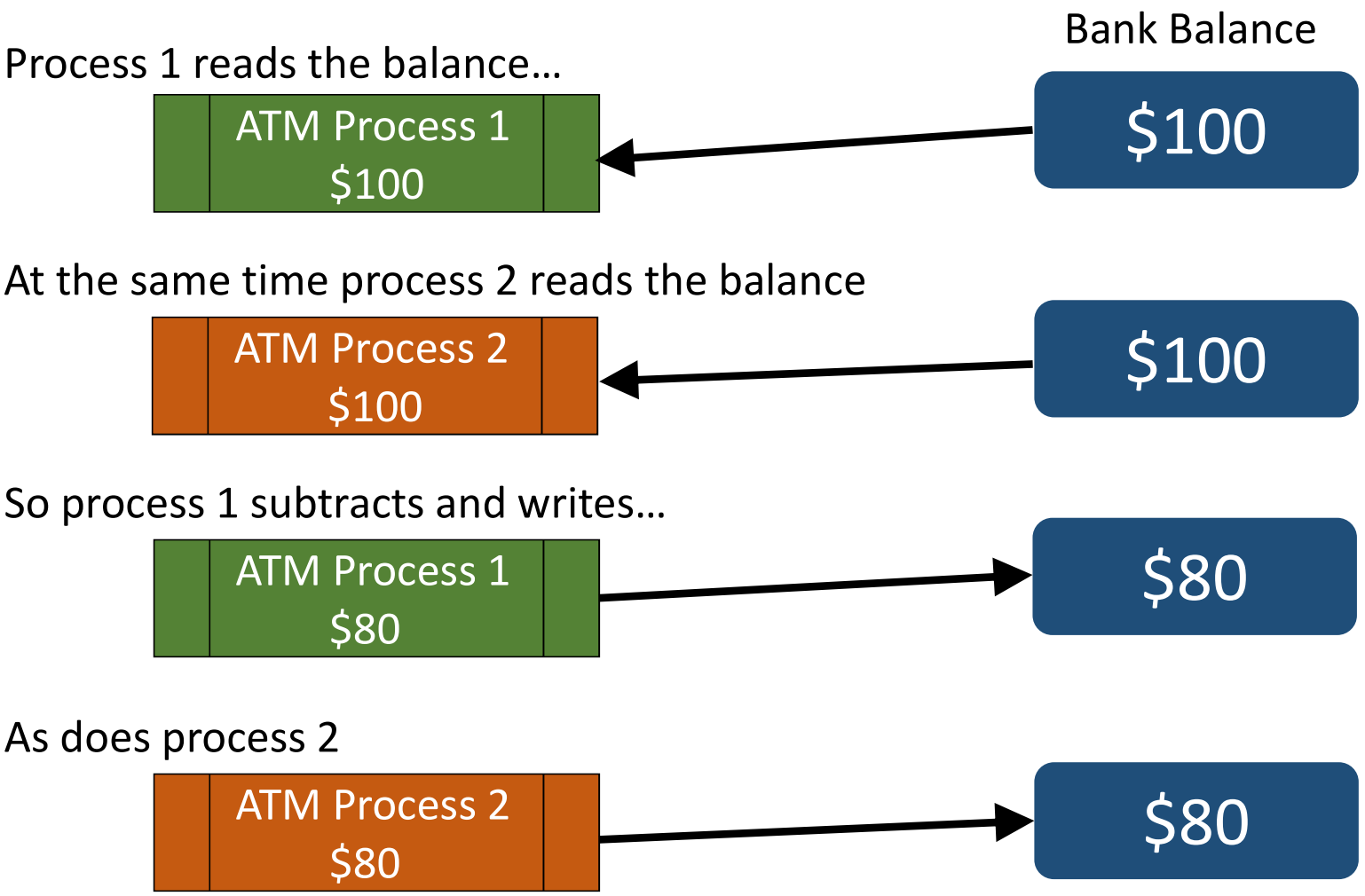
Possible Complications

- Concurrently running processes can share data and/or resources
- What if multiple processes access the same resource at the same time?
- This is most likely a disaster
  - aka, “Race Condition”, “Oops”, or “Aw carp”

The Classic Example

- Two ATM machines each withdraw $20 from the same account

- To update bank account balance:
  - Read current balance into memory
  - Subtract $20
  - Write new balance to the bank account


Bank Balance
Process 1 reads the balance…
ATM Process 1
$100
$100
At the same time process 2 reads the balance
ATM Process 2
$100
$100
So process 1 subtracts and writes…
ATM Process 1
$80
$80
As does process 2
ATM Process 2
$80
$80

Race conditions

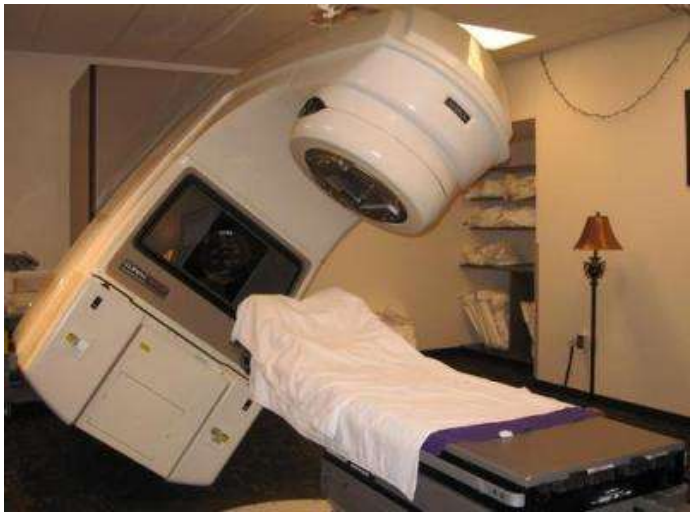
- Why are race conditions hard to detect?
  - They may only ever show up once, in a particularly strange set of conditions

- Another way of saying it: “A race hazard (or race condition) is a flaw in a system or process where the output exhibits unexpected critical **dependence** on the relative timing of events.”

  -Wikipedia

The Seriousness of Software Engineering

- Most infamous race condition:
  - http://en.wikipedia.org/wiki/Therac_25
- People could outrun the system
  - The system was counting on a normally slow human, and didn't take into account people learning to use the system faster
- 3 people died, and 3 were injured as a result of this software engineering disaster



http://hackaday.com/2015/10/26/killed-by-a-machine-the-therac-25

The Lesson - Provide Access Control

- Concurrent update situation
  - 2+ processes accessing resource concurrently
  - At least one process might write
- **Must provide access control**
  - If one process is writing, no other process should access (read OR write) the resource
- "Locks" solve these problems
  - Only process owning the lock may access (r/w) the resource
  - Many ways to do locking
  - Locking usually requires support from the OS
    - But you can do it in software, too

Access Control with a Lock File

```
do
{
   lock_fd = open(lock_file_path, O_WRONLY | O_CREAT | O_EXCL, 0644);

   if (lock_fd == -1)
   {
      if (errno == EEXIST)
      {
         // File already exists - wait a while, then try again
         sleep(1);
      }
      else
      {
         // An unexpected error - bail out
         perror("Couldn't open lock file\n");
         return(-1);
      }
   }
}
while (lock_fd == -1);
```

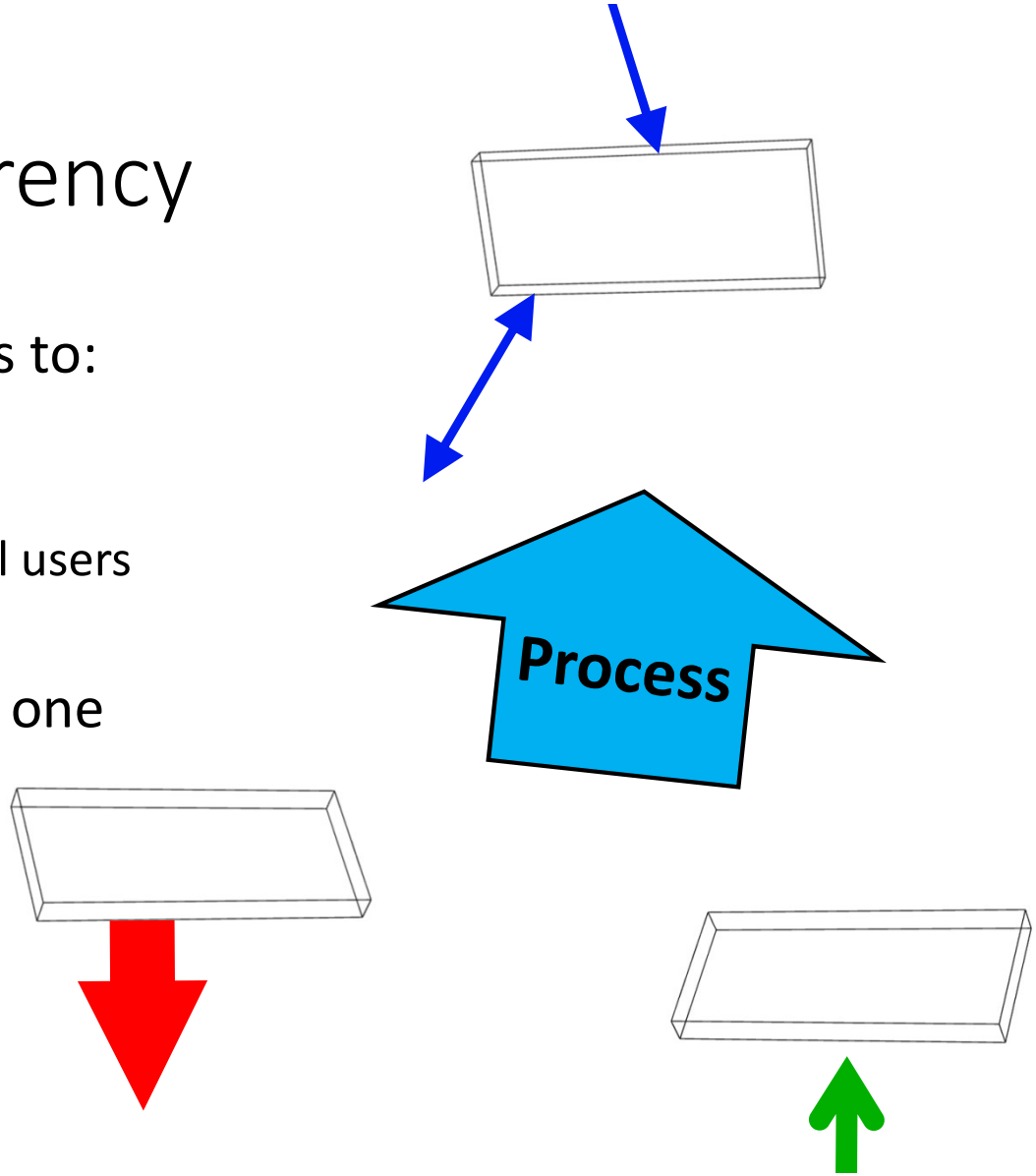
Deeper Definitions

- Kernel
  - The central part of an Operating System
  - Manages hardware (and drivers)
  - Provides the Scheduler
  - Not interacted with by users: system calls are requests to the kernel
- Scheduler
  - Distributes prioritized and/or fair CPU time

Process-Level Concurrency

- Imagine a chat server that needs to:
  - Watch for users connecting
  - Send chat output to all users
  - Receive chat input from individual users

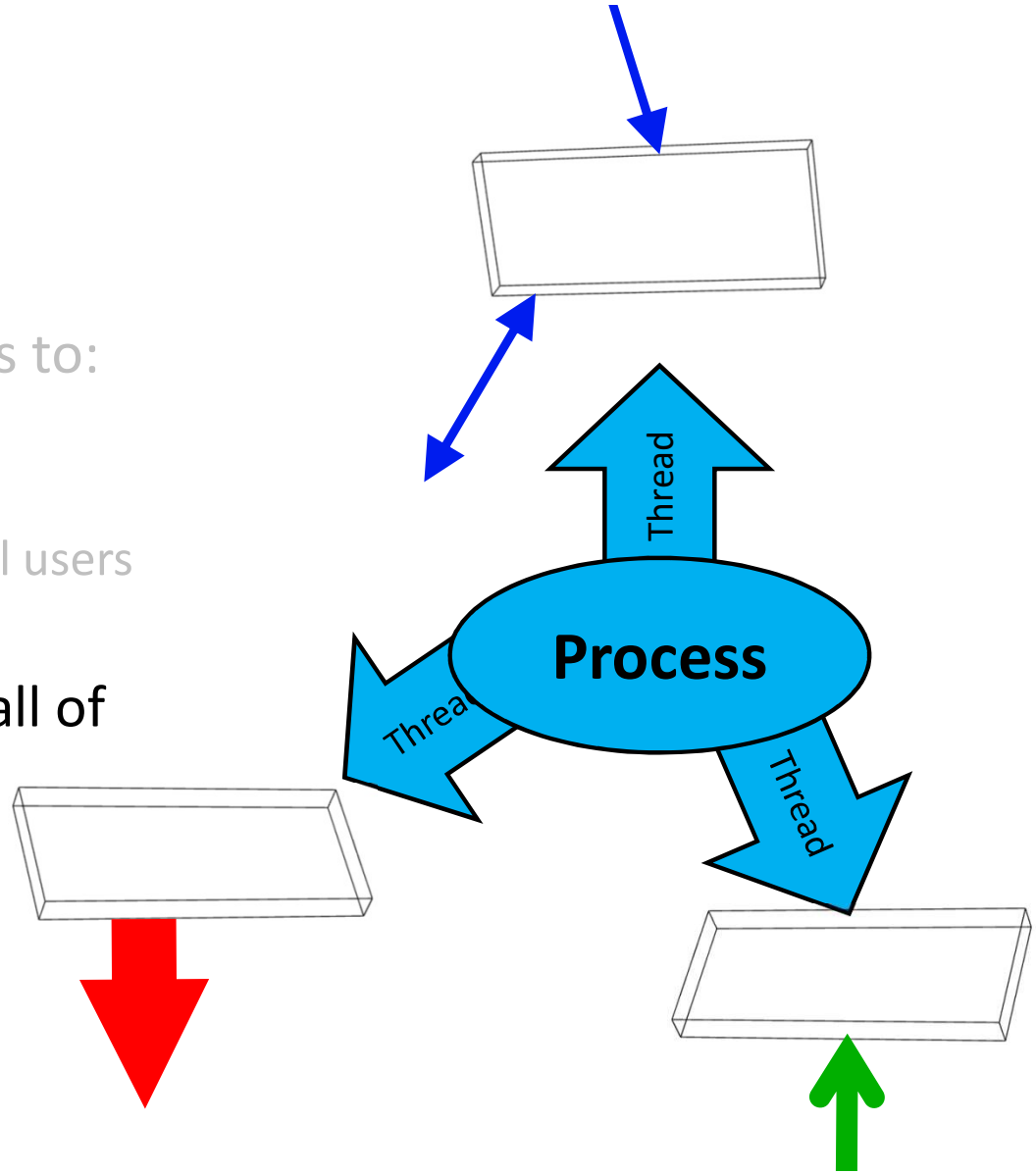
- But a single **process** can only do one of these things at a time!



Enter Threads

- Imagine a chat server that needs to:
  - Watch for users connecting
  - Send chat output to all users
  - Receive chat input from individual users

- **Threads** allows a process to do all of these things at the same time



Thread Advantages over Processes

- Communication between threads is vastly simpler than IPC:
  - Threads share the following:
    - Code, Heap, Data
  - They each have their own Stack, but they can access the Stacks of other threads(!)

- The CPU switches between executing threads much faster than switching between processes, because of what's shared above
  - On a multi-core CPU, the CPU can run each kernel-level thread on a separate core, yielding true concurrence

Thread Disadvantages

- Shared resources means increased possibilities of:
  - Race conditions
  - Resource contention, including file access
  - Leaking of sensitive data across threads

Types of Threads

- Kernel-Level Threads
  - Controlled by the kernel, given time by the scheduler
  - Akin to a mini-process

- User-Level Threads
  - The kernel doesn't know about these – they are entirely within a process and do not involve the scheduler; switching between them is done cooperatively by the protocol of the software itself to manage the task
  - Really just *emulation* of threading
  - Sometimes called green threads

Thread Type Comparison

- Kernel-Level Threads
  - Controlled by the kernel, given time by the scheduler
  - Simple to create with built-in libraries in UNIX
  - Generally considered the better choice in almost all circumstances

- User-Level Threads
  - Switching between user-level threads is even faster than switching between kernel-level threads, because it doesn't require swapping memory protection to the in-kernel scheduler and back to the process
  - The entire process can be pre-empted by the scheduler
  - A blocking system call blocks the entire process, and thus all of the threads
  - More difficult to use; libraries aren't built-in to UNIX

Thread Implementation in UNIX

- Implemented with the POSIX Threads API in UNIX

```
#include <pthread.h>
```

Compile with `-lpthread` option in `gcc`

Creating a Thread

```
int pthread_create(      pthread_t* thread,
                         const pthread_attr_t* attr,
                         void* (*start_routine) (void *),
                         void* arg
                   );
```

- **`thread`**: points to the variable in which the ID of the new thread is written into; depending on OS implementation, sometimes this is an int, sometimes it's a struct

Creating a Thread

```
int pthread_create(      pthread_t* thread,
                         const pthread_attr_t* attr,
                         void* (*start_routine) (void *),
                         void* arg
                   );
```

- **`attr`**: points to a `pthread_attr_t` struct that contains option flags (`NULL` if none)

Creating a Thread

```
int pthread_create(      pthread_t* thread,
                         const pthread_attr_t* attr,
                         void* (*start_routine) (void *),
                         void* arg
                   );
```

- **`start_routine`**: points to a function (in the current program) that will be the start point of execution for the *new* thread that copies this one

Similar to `fork()`, which we'll cover later

Creating a Thread

```
int pthread_create(      pthread_t* thread,
                         const pthread_attr_t* attr,
                         void* (*start_routine) (void *),
                         void* arg
                   );
```

- **`arg`**: points to the sole argument that is passed into `start_routine` (`NULL` if none); if multiple arguments are desired, pass a struct

Creating a Thread - Example

```
int resultInt;
pthread_t myThreadID;

resultInt =  pthread_create( &myThreadID,
             NULL,
             start_routine,
             NULL );
```

Destroying a Thread

- Threads can be killed by:
  - The thread calling `pthread_exit()`
  - The thread returns from `start_routine()`
  - The thread gets cancelled by another thread calling `pthread_cancel()`
  - Any thread in the process calls `exit()`

Identifying the Executing Thread

- Here's the function for getting the "thread ID" of an executing thread; note that this ID is not necessarily an integer:

```
pthread_t myThreadID = pthread_self();
```

- Testing for equality:

```
pthread_equal(myThreadID, unknownThreadID);
```

Thread Example – Page 1 of 3

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

#define NUM_THREADS     5

void* perform_work(void* argument)
{
   int passed_in_value;

   passed_in_value = *((int *) argument);
   printf("Hello World! It's me, thread with argument %d!\n", passed_in_value);
   return NULL;
}
```

Thread Example – Page 2 of 3

```
int main(void)
{
   pthread_t threads[NUM_THREADS];
   int thread_args[NUM_THREADS];
   int result_code, index;

   for (index = 0; index < NUM_THREADS; ++index) {
      // create all threads one by one
      thread_args[index] = index;
      printf("In main: creating thread %d\n", index);
      result_code = pthread_create(&threads[index], NULL,
                                   perform_work, (void *) &thread_args[index]);
      assert(0 == result_code);
   }
…
```

Thread Example – Page 3 of 3

```
…
    // wait for each thread to complete
    for (index = 0; index < NUM_THREADS; ++index)
    {
        result_code = pthread_join(threads[index], NULL);
        printf("In main: thread %d has completed\n", index);
        assert(0 == result_code);
    }

    printf("In main: All threads completed successfully\n");
    exit(EXIT_SUCCESS);
}
```

Block until thread ‘index’ completes

Thread Example - Results

```
$ gcc -o threadtest threadtest.c -lpthread
$ threadtest
In main: creating thread 0
In main: creating thread 1
In main: creating thread 2
Hello World! It's me, thread with argument 0!
In main: creating thread 3
Hello World! It's me, thread with argument 1!
Hello World! It's me, thread with argument 2!
In main: creating thread 4
Hello World! It's me, thread with argument 3!
In main: thread 0 has completed
In main: thread 1 has completed
In main: thread 2 has completed
Hello World! It's me, thread with argument 4!
In main: thread 3 has completed
In main: thread 4 has completed
In main: All threads completed successfully
```

Mutexes

- An abbreviation for “mutual exclusion”
- Implemented as part of the POSIX `pthread` API to provide thread synchronization via “locks”
- Provides the programmer the ability to protect data from multiple reads and writes on the same files
- There are several types which check variously for errors, perform faster, etc.

The Lifespan of a Mutex

```
pthread_mutex_t myMutex = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_destroy(myMutex);
```

Mutex Locking

- A thread acquires a lock on a mutex variable by attempting to call a special lock function:

```
pthread_mutex_t myMutex = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_lock(myMutex);
```

- Once the mutex is locked, any other thread attempting to lock it will block!

Mutex Unlocking

- A thread unlocks a mutex variable *that it has previously locked* like so:

```
pthread_mutex_unlock(myMutex);
```

- Once unlocked, one of the other blocked threads (chosen essentially randomly) currently blocked on the mutex will unblock and gain the lock

- A non-blocking attempt to lock the mutex:

```
int resultCode = pthread_mutex_trylock(myMutex);
```

Mutexes Generalized

- A mutex is a form of *semaphore*, which come in two types:
  - Counting semaphore
    - Allow some sort of arbitrary resource count, e.g number of available buffers
  - Binary semaphore
    - Equal to 1 or 0, indicating locked/unavailable or unlocked/available

- Invented by Edsger Dijkstra in 1962 or 1963
  - More on him later!

File Access in C

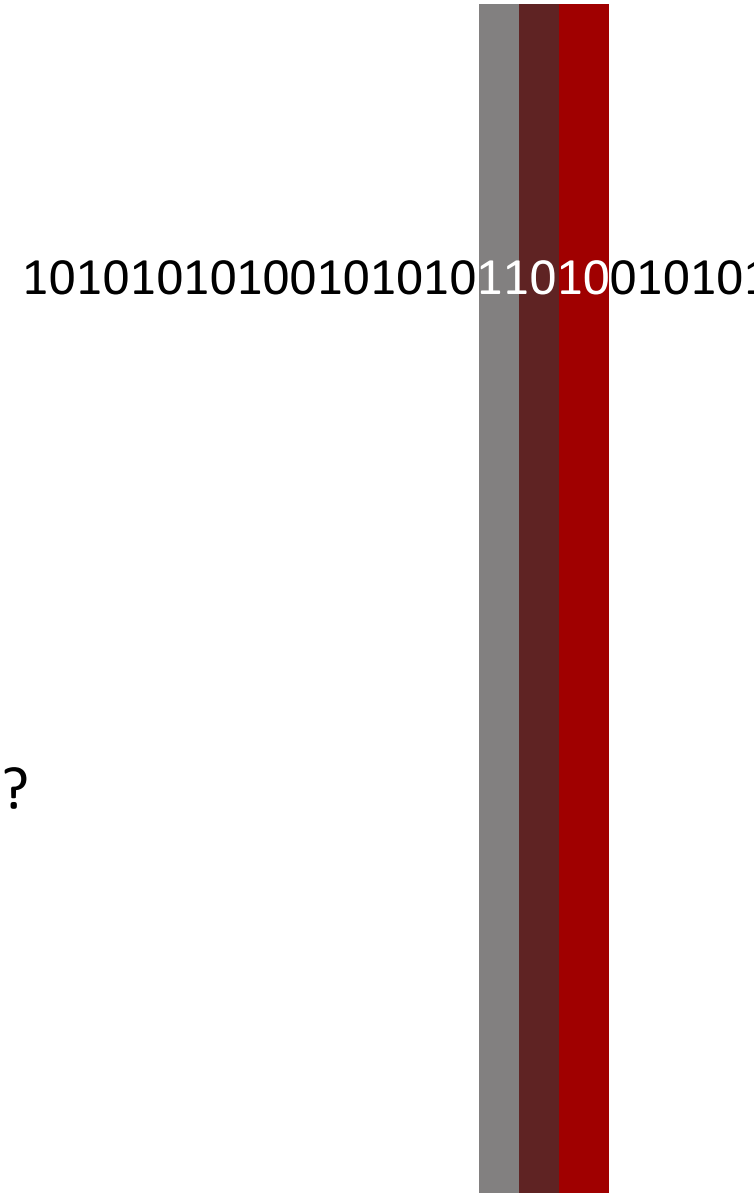
Benjamin Brewster

Unix Paradigm

- Everything is a file!
  - Except processes

- Directory contents could include:
  - Hard links
  - Symbolic links
  - Named pipes
  - Device character special file
  - Device block special file
  - Named socket

What is a File in UNIX?



- System Programmer View:
  - A stream of bytes
    - Could be accessed as an array
    - Newlines/carriage returns & tabs are all just bytes, too!
  - Persistent

- How do we access files for reading and writing?

Opening a File

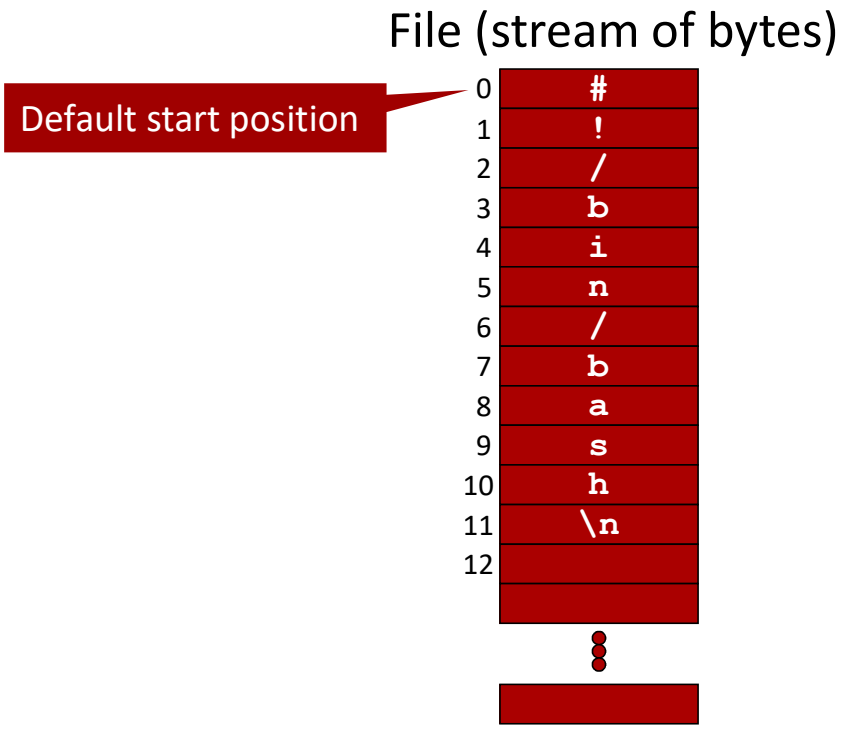
- Files can be open for:
  - read only :: O_RDONLY
  - write only :: O_WRONLY
  - read and write :: O_RDWR

Reminder: We're talking about C programming now, not shell scripts

- When you open a file for writing…
  - Should you delete an existing file with the same name?
  - If not, where do you want to start writing
    - Beginning? End? Somewhere else?
  - If the file doesn't exist, should you create it?
  - If you create it, what should the initial access permissions be?

The File Pointer

- Tracks where the next file operation occurs in an open file
- A separate file pointer is maintained for each open file
- All of the operations we're talking about:
  - Directly impact which byte in a file is pointed to by the file pointer when the file is opened
  - Move the file pointer



Open for Read

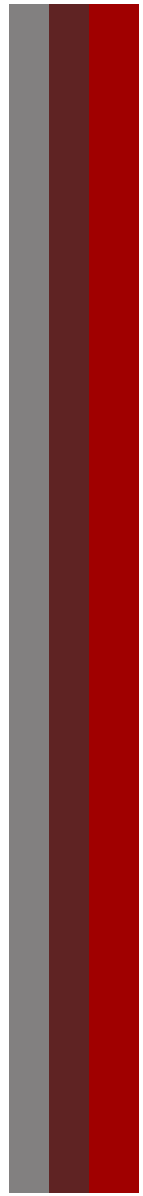
```
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
        char file[] = "cs344/grades.txt";
        int file_descriptor;

        file_descriptor = open(file, O_RDONLY);

        if (file_descriptor < 0)
        {
                fprintf(stderr, "Could not open %s\n", file);
                exit(1);
        }

        close(file_descriptor);
        return 0;
}
```

Using `open()` and `close()` allows us to represent file descriptors as **int**s.

The more modern `fopen()` and `fclose()` require a special file descriptor type.

Open for Write

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

int main(void)
{
        char file[] = "cs344/grades.txt";
        int file_descriptor;

        file_descriptor = open(file, O_WRONLY);

        if (file_descriptor < 0)
        {
                fprintf(stderr, "Could not open %s\n", file);
                perror("Error in main()");
                exit(1);
        }

        close(file_descriptor);
        return 0;
}
```



Truncating an Existing File

- When you open a file for writing, should you delete all contents of an existing file with the same name, or write over existing contents?
  - To delete it and start fresh: `O_TRUNC`

- Example:

```
file_descriptor = open(file, O_WRONLY | O_TRUNC);
```

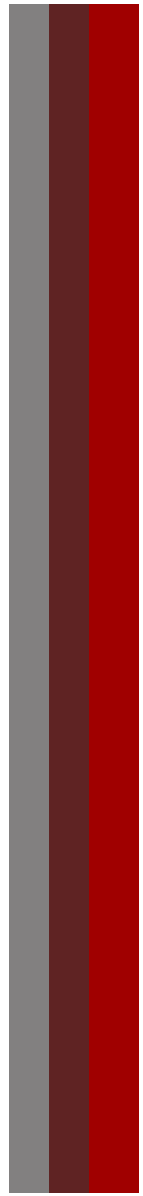
  - Opens an existing file for writing only, then deletes all the data in it
  - Sets the file pointer to position 0

Appending to an Existing File

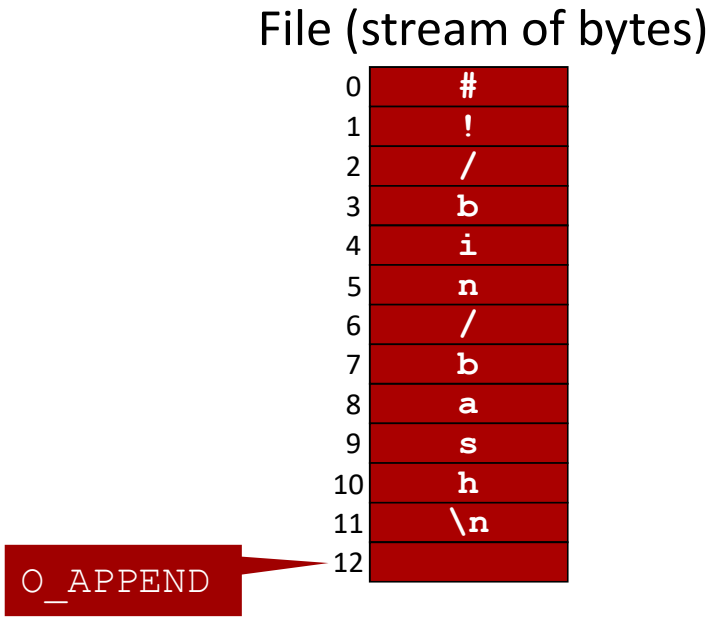
- Open the file in append mode with flag: `O_APPEND`
- Before *every* write, the file pointer will be automatically set to the end of the file
- Example

```
file_descriptor = open(filepath, O_WRONLY | O_APPEND);
```

  - Opens an existing file for writing only in append mode



O_APPEND and the File Pointer



Creating a New File

- To open (or create) a file that doesn't exist, use flag: `O_CREAT`

- Example: open a file for writing only, creating it if it doesn't exist:

```
file_descriptor = open(filepath, O_WRONLY | O_CREAT, 0600);
```

- The third parameter of `open()` *must* be used when the creation of a new file is requested (i.e. using `O_CREAT` or `O_TMPFILE`)

Even though the `open()` call will probably fail in bizarre ways if you don't include the third argument here, it still compiles! Thanks, C!

Creating a New File - Access Permissions

- Again, the third parameter of `open()` *must* be used when the creation of a new file is requested (i.e. using `O_CREAT` or `O_TMPFILE`)

- Third parameter contains octal number permissions bits:
  - Specify directly as with chmod: `0600`
  - Or you can bit-wise OR flags together: `S_IRUSR | S_IWUSR`

- Example:

```
file_descriptor = open(file, O_WRONLY | O_CREAT, 0600);
file_descriptor = open(file, O_WRONLY | O_CREAT, S_IRUSR | S_IWUSR);
```

User has read and write permission

User has read permission

User has write permission

`lseek()`

- Manipulates a file pointer in a file
- Used to control where you're messing with da bitz

- Examples:
  - Move to byte #16

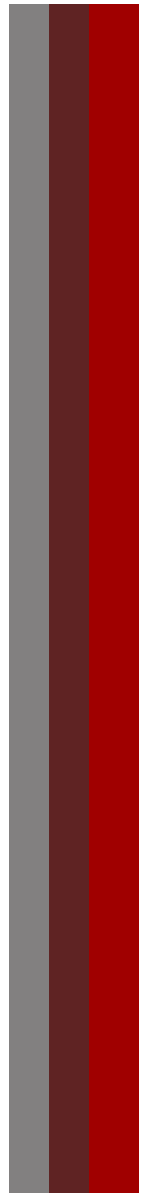
```
newpos = lseek(file_descriptor, 16, SEEK_SET);
```

  - Move forward 4 bytes

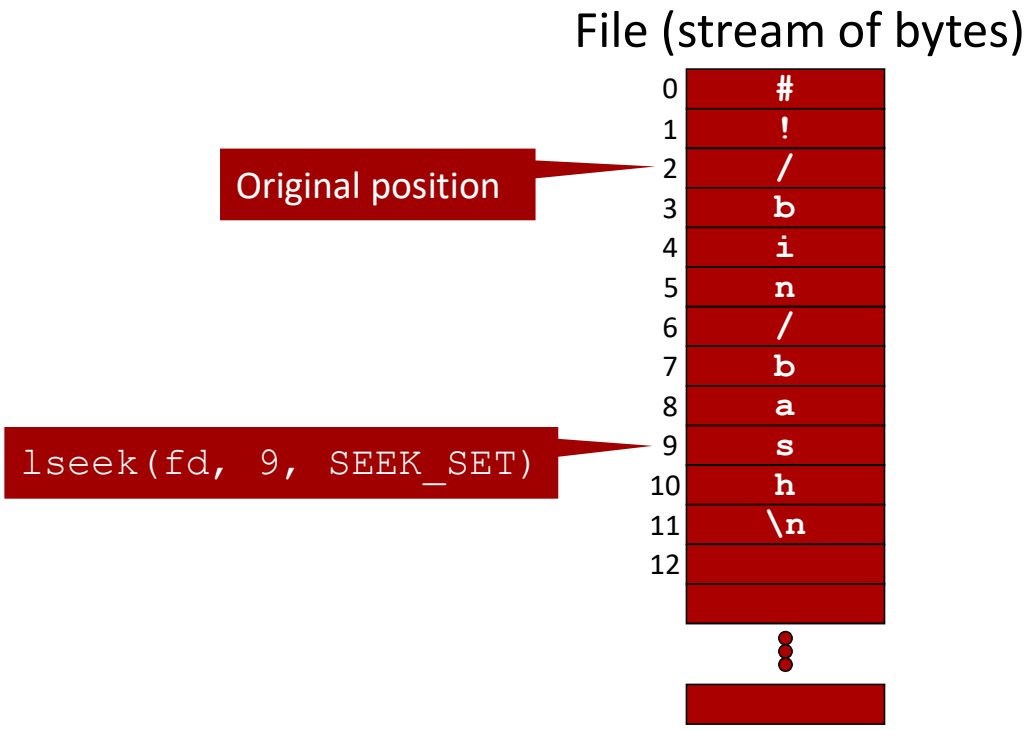
```
newpos = lseek(file_descriptor, 4, SEEK_CUR);
```

  - Move to 8 bytes from the end

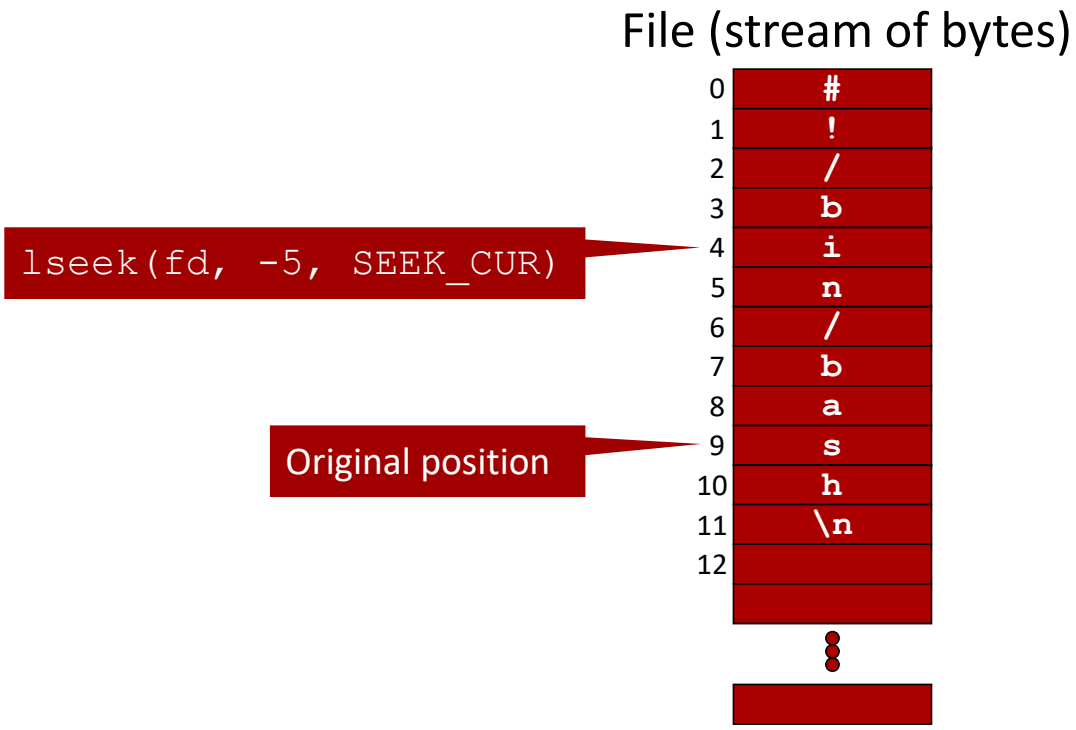
```
newpos = lseek(file_descriptor, -8, SEEK_END);
```



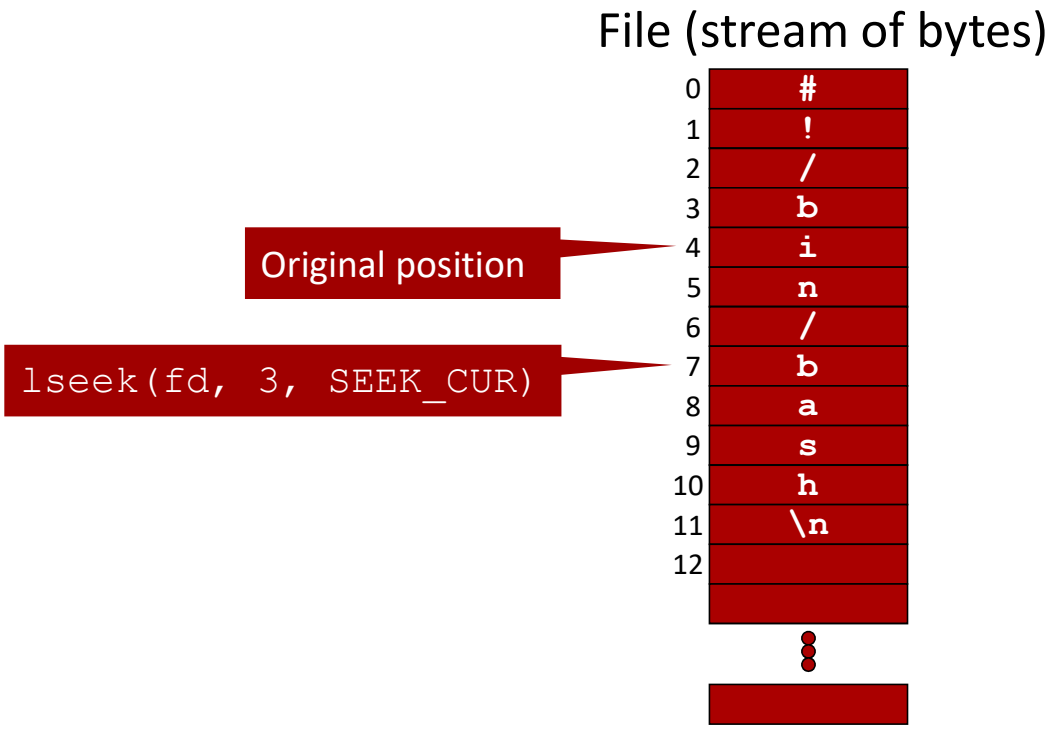
`lseek()` :: `SEEK_SET` :: Setting Position



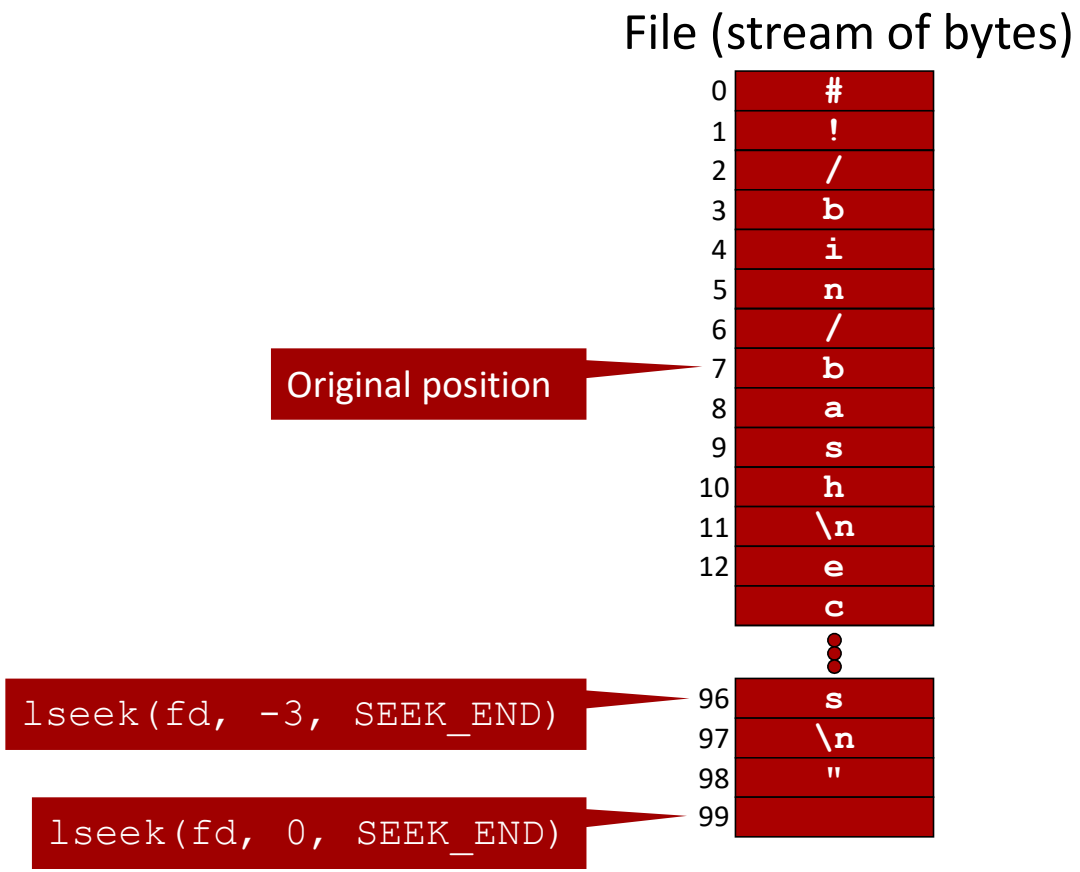
`lseek()` :: `SEEK_CUR` :: Moving backwards



`lseek()` :: `SEEK_CUR` :: Moving Forwards



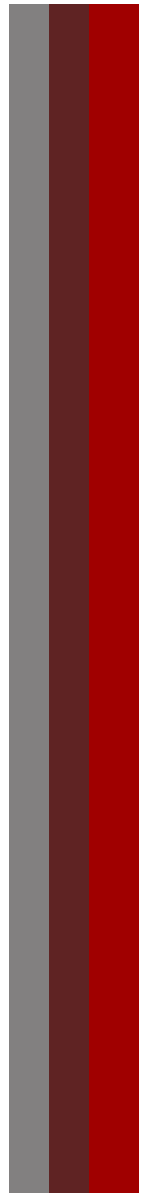
`lseek()` :: `SEEK_END` :: Moving Relative to the End



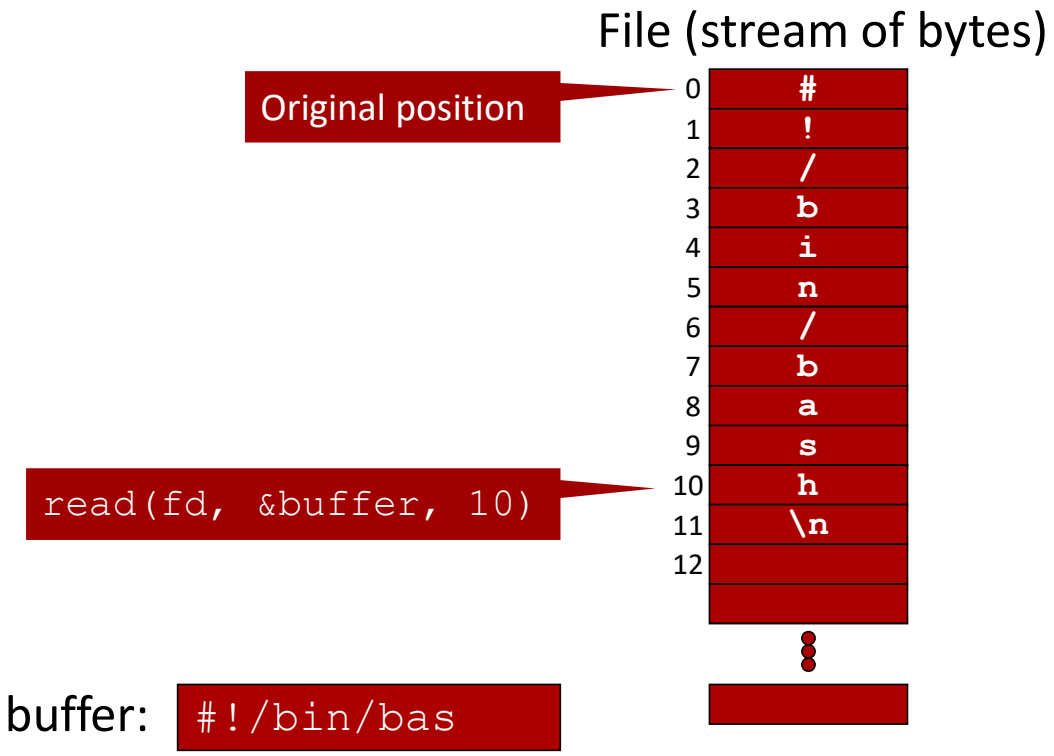
Read/Write and the File Pointer

- If you've opened a file for reading and/or writing, be aware that *both* of these operations will change the file pointer location!

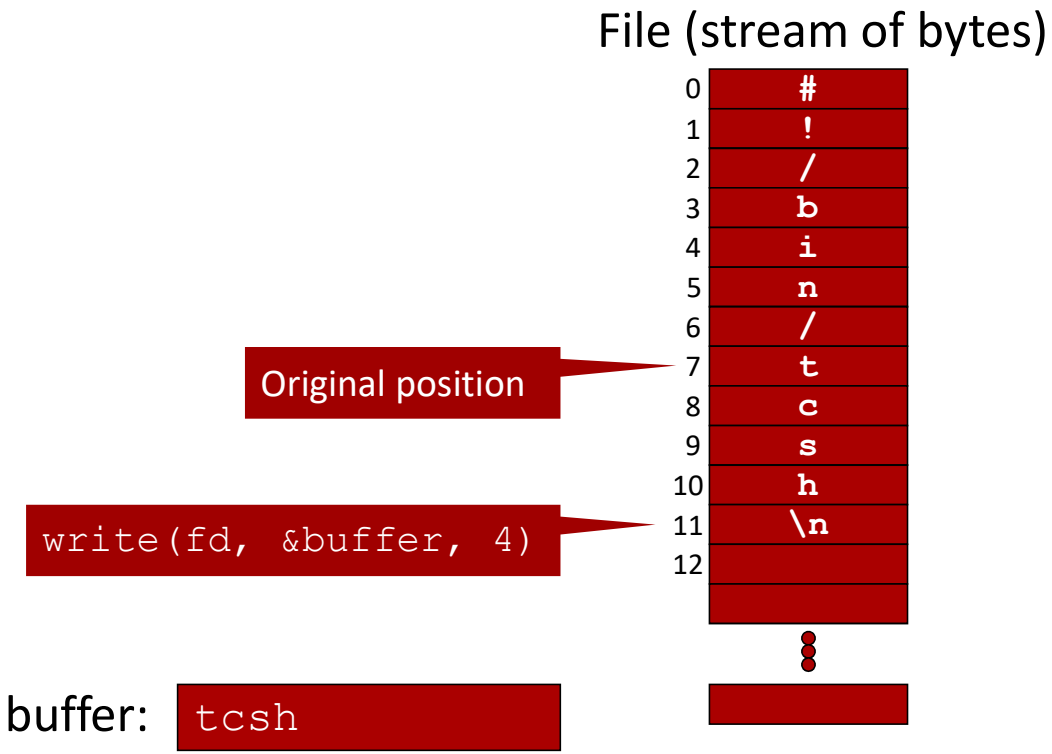
- The pointer will be incremented by exactly the number of bytes read or written



`read()` and the File Pointer



`write()` and the File Pointer



Complete Read/Write Example

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

int main(void)
{
        int file_descriptor;
        char* newFilePath = "./newFile.txt";
        char* giveEm = "THE BUSINESS\n";
        ssize_t nread, nwritten;
        char readBuffer[32];

        file_descriptor = open(newFilePath, O_RDWR | O_CREAT | O_TRUNC, S_IRUSR | S_IWUSR);

        if (file_descriptor == -1)
        {
                printf("Hull breach - open() failed on \"%s\"\n", newFilePath);
                perror("In main()");
                exit(1);
        }

        nwritten = write(file_descriptor, giveEm, strlen(giveEm) * sizeof(char));

        memset(readBuffer, '\0', sizeof(readBuffer)); // Clear out the array before using it
        lseek(file_descriptor, 0, SEEK_SET); // Reset the file pointer to the beginning of the file
        nread = read(file_descriptor, readBuffer, sizeof(readBuffer));

        printf("File contents:\n%s", readBuffer);
        return 0;
}
```

Mode bits

Permissions

These two steps are really important to avoid nasty bugs

The Standard IO Library in C

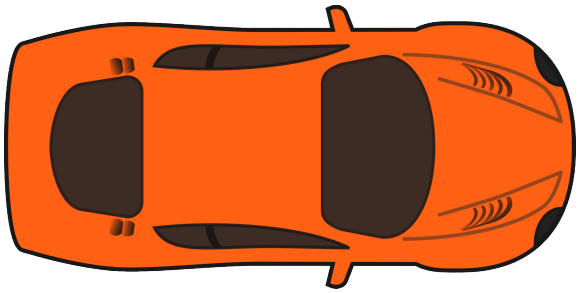
- `fopen`, `fclose`, `printf`, `fprintf`, `sprintf`, `scanf`, `fscanf`, `getc`, `putc`, `gets`, `fgets`, `fseek`, etc.
- Automatically buffers input and output intelligently
- Easy to work in line mode
  - i.e., read one line at a time
  - write one line at a time
- Powerful string and number formatting
- To use them:

  #include <stdio.h>

Why Teach and Use `read()` & `write()`?

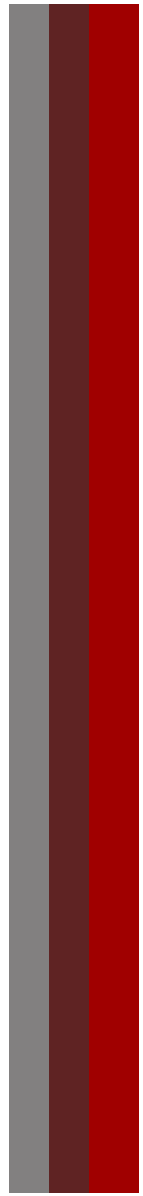
- Maximum performance
  - IF you know exactly what you are doing
  - No additional hidden overhead from stdio, which is much slower!
  - No hidden system calls behind stdio functions which may be non-reentrant

- Control exactly what is written/read and at what times



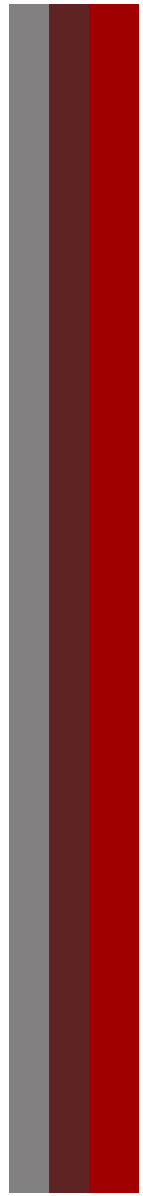
Some stdio Functions

- `fclose` Close a stream
- `feof` Check if End Of File has been reached
- `fgetc` Get next character from a stream
- `fgetpos` Get position in a stream
- `fopen` Open a file
- `fprintf` Print formatted data to a stream
- `fputc` Write character to a stream
- `fread` Read block of data from a stream
- `fseek` Reposition stream's position indicator (stdio version of `lseek`)
- `getc` Get the next character
- `getchar` Get the next character from stdin



Some More stdio Functions

- `gets` Get a string from stdin
- `printf` Print formatted data to stdout
- `putc` Write character to a stream
- `putw` Write an integer to a stream
- `remove` Delete a file
- `rename` Rename a file or directory
- `rewind` Reposition file pointer to the beginning of a stream
- `scanf` Read formatted data from stdin
- `sprintf` Format data to a string
- `sscanf` Read formatted data from a string
- `ungetc` Push a character back into stream



Files or Streams?

- **stdin**, **stdout**, and **stderr** are actually *file streams*, not file system files
- File streams wrap around, and provide buffering to, the underlying file descriptor among other features
- The stdio library streams are connected with the `fopen()` call to a variable of type `FILE*`:

```
FILE* myFile = fopen("datafile103", "r");
```

- Streams are closed when a process terminates
- Raw file descriptors with open files are passed on to child processes:
  - A process spawns a new child process with `fork()`, all open files are shared between parent and child processes

Getting Input From the User: userinput.c

When `bufferSize = 0` and `lineEntered = NULL`, `getline()` allocates a buffer for you with `malloc()`

Could also be a regular file opened as a stream with `fopen()`

`getline()` is my preferred tool to get user input

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

void main() {
        int numCharsEntered = -5; // How many chars we entered
        int currChar = -5; // Tracks where we are when we print out every char
        size_t bufferSize = 0; // Holds how large the allocated buffer is
        char* lineEntered = NULL; // Points to a buffer allocated by getline() that holds our entered string + \n + \0

        while(1)
        {
                // Get input from the user
                printf("Enter in a line of text (CTRL-C to exit):");
                numCharsEntered = getline(&lineEntered, &bufferSize, stdin); // Get a line from the user
                printf("Allocated %zu bytes for the %d chars you entered.\n", bufferSize, numCharsEntered);
                printf("Here is the raw entered line: \"%s\"\n", lineEntered);

                // Print out the actual contents of the string that was entered
                printf("Here are the contents of the entire buffer:\n");
                printf("  # CHAR INT\n");
                for (currChar = 0; currChar < bufferSize; currChar++) // Display every character in both dec and ASCII
                        printf("%3d `%c\' %3d\n", currChar, lineEntered[currChar], lineEntered[currChar]);
                free(lineEntered); // Free the memory allocated by getline() or else memory leak
                lineEntered = NULL; // Reset pointer to allow getline() magic to work again
        }
}
```

Results - Getting Input From the User: userinput.c

```
$ gcc -o userinput userinput.c
$ userinput
Enter in a line of text (CTRL-C to exit):abc o0OlI1
Allocated 120 bytes for the 11 chars you entered.
Here is the raw entered line: "abc o0OlI1
"
Here are the contents of the entire buffer:
  # CHAR INT
  0 `a'  97
  1 `b'  98
  2 `c'  99
  3 ` '  32
  4 `o' 111
  5 `0'  48
  6 `O'  79
  7 `l' 108
  8 `I'  73
  9 `1'  49
 10 `
'  10
 11 `'   0
 12 `'   0
            ...(cut)...
118 `'   0
119 `'   0
Enter in a line of text (CTRL-C to exit):^C
```

Newline; if you don't want this, just add:

```
lineEntered[numCharsEntered - 1] = '\0';
```

after calling `getline()`

Null terminator

Obtaining File Information

- `stat()` and `fstat()`

- Retrieve all sorts of information about a file or directory
  - Which device it is stored on
  - Ownership/permissions of that file
  - Number of hard links pointing to it
  - Size of the file
  - Timestamps of last modification and access
  - Ideal block size for I/O to this file

Strings in C
A Programmer's Nightmare

Benjamin Brewster

C Strings

- Strings in C are sequences of characters contiguously stored
  - Not a native type like `int` or `float` in more advanced languages
- A “string” terminates with the null character
  `\0`
- That’s *it!* Any further programmatic use of strings requires functions and procedures that work within this format

Displaying C Strings with Formatted Printing

- Formatted means numbers correctly printed with text

- Formatted printing is done with:
  - `printf()` Prints to standard out
  - `sprintf()` Prints to a string (a char array)
  - `fprintf()` Prints to a file

- These functions look for null terminators to know when to stop

Basic C String Functions

Warning: do not use the == operator!

- Use the string library functions:
  - `strcmp()` Compares two strings for equality
  - `strlen()` Returns the length of the string in characters, not including null terminator
  - `strcpy()` Copies one string into another
  - `strcat()` Returns one string that is a concatenation of itself with another string
- n-character versions:
  - `strncpy()` Copy only n characters - won't null-terminate a full array, or actually prevent you from over-writing an array
  - `strncat()` Appends only a portion of a string to another

Declaring C Strings

- Three ways of declaring the same string

```
1. char* mystring     = "my string";
2. char  mystring[]   = "my string";
3. char  mystring[20] = "my string";
```

- Are they really the same? And why do we care in OS?

Declaring C Strings

- Three ways of declaring the same string

```
1. char* mystring     = "my string";
2. char  mystring[]   = "my string";
3. char  mystring[20] = "my string";
```

- Are they really the same? And why do we care in OS?
- Because this one difference shows how close C is to the underlying memory management being performed by UNIX

Declaring C Strings

- Three ways of declaring the same string

```
1. char* mystring     = "my string";
2. char  mystring[]   = "my string";
3. char  mystring[20] = "my string";
```

- Are they really the same? And why do we care in OS?
- Because this one difference shows how close C is to the underlying memory management being performed by UNIX
- I.e. you need to know this, because otherwise you'll break all the things and not know why

Declaring C Strings – Method 1

- Three ways of declaring the same string

```
1. char* mystring     = "my string";
2. char  mystring[]   = "my string";
3. char  mystring[20] = "my string";
```

- At compile time, creates a sequence of bytes in the **read-only initialized data segment** portion of memory with the contents `"my string"`
- During execution, creates a pointer on the **stack** (automatic variable) called `mystring` that points to the read-only sequence of characters in the **data segment**
- `mystring` can be pointed to other addresses (it doesn't hold chars by itself, as it's a pointer)

Declaring C Strings – Method 1 – Example

```
#include <stdio.h>

void main()
{
        char* mystring = "my string";
        printf("Var is: %s\n", mystring);
        mystring[3] = 'Q';
        printf("Var is: %s\n", mystring);
}
```

`mystring` is a pointer put on the stack

"`my string`" is a string literal defined and stored in the read-only portion of the data segment

Index 3 bytes off of where `mystring` is pointing too, then change whatever is there to 'Q'…

Result:

Declaring C Strings – Method 1 – Example

```
#include <stdio.h>

void main()
{
        char* mystring = "my string";
        printf("Var is: %s\n", mystring);
        mystring[3] = 'Q';
        printf("Var is: %s\n", mystring);
}
```

`mystring` is a pointer put on the stack

"`my string`" is a string literal defined and stored in the read-only portion of the data segment

Result:

```
Var is: my string
Segmentation fault (core dumped)
```

… except you can't do that, because your program cannot change memory in the read-only portion of the data segment

Declaring C Strings – Method 2

- Three ways of declaring the same string

```
1. char* mystring     = "my string";
2. char  mystring[]   = "my string";
3. char  mystring[20] = "my string";
```

- During execution, creates space for 10 bytes on the **stack** as an automatic variable, names that variable `mystring`
- Puts **`"my string"`** into the variable `mystring` with a null terminator after it
- The variable `mystring` is editable, as it is an array

Declaring C Strings – Method 2 – Example

```
#include <stdio.h>

void main()
{
        char mystring[] = "my string";
        printf("Var is: %s\n", mystring);
        mystring[3] = 'Q';
        printf("Var is: %s\n", mystring);
}
```

mystring is an array

Result:

```
Var is: my string
Var is: my Qtring
```

Declaring C Strings – Method 3

- Three ways of declaring the same string

```
1. char* mystring     = "my string";
2. char  mystring[]   = "my string";
3. char  mystring[20] = "my string";
```

- Creates space for 20 bytes on the **stack** as an automatic variable, names that variable `mystring`
- Puts **`"my string"`** into the variable `mystring` with a null terminator after it
- The variable `mystring` is editable, as it is an array

String Literals (Again)

- What's wrong with this code:

```
char* mystring = "my string";
strcpy(mystring, "AA string");
printf(mystring);
```

String Literals (Again)

- What's wrong with this code:

```
char* mystring = "my string";
strcpy(mystring, "AA string");
printf(mystring);
```

- String literals cannot be changed in C - they are initialized in the **read-only** section of the **initialized data segment**

- When is this error caught?
  - Only at run-time, as a seg-fault; this compiles just fine

Buffer Overrun

- What's wrong with this?

```
char fiveStr[5] = "five";
strcpy(fiveStr, "five6");
printf(fiveStr);
```

Buffer Overrun

- What's wrong with this?

```
char fiveStr[5] = "five";
strcpy(fiveStr, "five6");
printf(fiveStr);
```

- "`five6`" is too long to store in `fiveStr`

- When is this error caught?
  - Never!
  - Unless something you needed is overwritten and a segfault occurs because a just-accessed pointer no longer points to where it was supposed to!

Fully Initializing C String Arrays

```
char mystring[20];
strcpy(mystring, "my string");
```


```
printf("%s", mystring);
```

How do we deal with this uninitialized data?

Result:

```
my string
```

What's In that Uninitialized Data?

```
$ cat cstring-array-unint.c
#include <stdio.h>
#include <string.h>

void main()
{
  int i = -5;
  char mystring[20];

  strcpy(mystring, "my string");

  printf("Char => Int :: ASCII Table Lookup\n");

  for (i = 0; i < 19; i++)
    printf("%c    => %d\n", mystring[i], mystring[i]);
}
$ gcc -o cstring-array-unint cstring-array-unint.c
```

```
$ cstring-array-unint
Char => Int :: ASCII Table Lookup
m    => 109
y    => 121
     => 32
s    => 115
t    => 116
r    => 114
i    => 105
n    => 110
g    => 103
    => 0
@    => 64
    => 0
    => 0
    => 0
    => 0
    => 0
@    => 64
W    => 87
▒    => -26
```

Printing chars as ints is a great way to debug C string arrays!

ASCII -26 summons Cthulu

Initializing C String Arrays - The Bad

- Depending on how you declare them, C string arrays may be full of uninitialized data - it's best to clear them before use

- What happens if we somehow remove the automatic null terminator?

```
$ cat cstring-array.c
#include <stdio.h>
#include <string.h>

void main()
{
        int i = -5;
        char mystring[20];
        strcpy(mystring, "my string");

        printf("Var is: %s\n", mystring);
        mystring[3] = 'Q';
        printf("Var is: %s\n", mystring);

        mystring[9] = '#';
        mystring[19] = '\0';
        for (i = 10; i < 19; i++)
                if (mystring[i] == '\0')
                        mystring[i] = '#';

        printf("Var is: %s\n", mystring);
}
```

Uninitialized!

Initializing C String Arrays - The Bad

- Depending on how you declare them, C string arrays may be full of uninitialized data - it's best to clear them before use

- What happens if we somehow remove the automatic null terminator?

```
$ cat cstring-array.c
#include <stdio.h>
#include <string.h>

void main()
{
        int i = -5;
        char mystring[20];
        strcpy(mystring, "my string");

        printf("Var is: %s\n", mystring);
        mystring[3] = 'Q';
        printf("Var is: %s\n", mystring);

        mystring[9] = '#';
        mystring[19] = '\0';
        for (i = 10; i < 19; i++)
                if (mystring[i] == '\0')
                        mystring[i] = '#';

        printf("Var is: %s\n", mystring);
}
$ gcc -o cstring-array cstring-array.c
$ cstring-array
Var is: my string
Var is: my Qtring
Var is: my Qtring#@#####▒t▒
```

Uninitialized!

Different almost every time it runs, as memory is used

Initializing C String Arrays - The Suspicious

- Depending on how you declare them, C string arrays may be full of uninitialized data - it's best to clear them before use

Seems to initialize entire array to \0 but is this portable?

```
$ cat cstring-array.c
#include <stdio.h>
#include <string.h>

void main()
{
        int i = -5;
        char mystring[20] = "my string";

        printf("Var is: %s\n", mystring);
        mystring[3] = 'Q';
        printf("Var is: %s\n", mystring);

        mystring[9] = '#';
        mystring[19] = '\0';
        for (i = 10; i < 19; i++)
                if (mystring[i] == '\0')
                        mystring[i] = '#';

        printf("Var is: %s\n", mystring);
}

$ gcc -o cstring-array cstring-array.c

$ cstring-array
Var is: my string
Var is: my Qtring
Var is: my Qtring##########
```

Initializing C String Arrays - The Preferred

- Depending on how you declare them, C string arrays may be full of uninitialized data - it's best to clear them before use

```
$ cat cstring-array.c
#include <stdio.h>
#include <string.h>

void main()
{
        int i = -5;
        char mystring[20];
        memset(mystring, '\0', 20);
        strcpy(mystring, "my string");
        printf("Var is: %s\n", mystring);
        mystring[3] = 'Q';
        printf("Var is: %s\n", mystring);
        mystring[9] = '#';
        mystring[19] = '\0';
        for (i = 10; i < 19; i++)
                if (mystring[i] == '\0')
                        mystring[i] = '#';
        printf("Var is: %s\n", mystring);
}
$ gcc -o cstring-array cstring-array.c
$ cstring-array
Var is: my string
Var is: my Qtring
Var is: my Qtring##########
```

Fully Initialized

Meanwhile Back on the Ranch…

- C continues to provide dangerous string functions

`strtok()` :: String tokenizer

- Splits strings into chunks
- Makes your hair fall out
- Maxes out your credit cards
- Unfriends all your social media friends
- Sometimes the best/only tool for the job :/

`strtok` Example

```
char input[18] = "This.is my/string";

char* token = strtok(input, " ./");
token = strtok(NULL, " ./");
token = strtok(NULL, " ");
```

This

is

my/string

Changing the delimiter as strtok() tokenizes the string is neat

A Major `strtok()` Drawback *(the first of many)*

```
char* input = "This.is my/string";
char* token = strtok(input, " ./");
token = strtok(NULL, " ./");
token = strtok(NULL, " ");
```

- Fails miserably. Why?

A Major `strtok()` Drawback *(the first of many)*

```
char* input = "This.is my/string";
char* token = strtok(input, " ./");
token = strtok(NULL, " ./");
token = strtok(NULL, " ");
```

- Fails miserably, crashing on execution: Why?
  - Because `input` is a string literal, and `strtok()` is about to mess with your strings

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

void main()
{
	char input[50];
	char* token = 0; // Set null pointer
	int inputsize = -5;
	int currChar = -5;

	memset(input, '\0', 50);
	strcpy(input, "A.B C/D");
	inputsize = strlen(input);

	printf("Input: "); for (currChar = 0; currChar < inputsize; currChar++) printf("%2d ", input[currChar]);
	printf(" = \"%s\", token: \"%s\"\n", input, token);
	token = strtok(input, " ./");
	printf("Input: "); for (currChar = 0; currChar < inputsize; currChar++) printf("%2d ", input[currChar]);
	printf(" = \"%s\", token: \"%s\"\n", input, token);
	token = strtok(NULL, " ./");
	printf("Input: "); for (currChar = 0; currChar < inputsize; currChar++) printf("%2d ", input[currChar]);
	printf(" = \"%s\", token: \"%s\"\n", input, token);
	token = strtok(NULL, " ./");
	printf("Input: "); for (currChar = 0; currChar < inputsize; currChar++) printf("%2d ", input[currChar]);
	printf(" = \"%s\", token: \"%s\"\n", input, token);
}
```

strtok Example Results

```
Input: 65 46 66 32 67 47 68  = "A.B C/D", token: "(null)"
Input: 65  0 66 32 67 47 68  = "A", token: "A"
Input: 65  0 66  0 67 47 68  = "A", token: "B"
Input: 65  0 66  0 67  0 68  = "A", token: "C"
```

- `input` gets jacked up by `strtok()` as the delimiters encountered during parsing get nulled
- Further, this can only work because `strtok()` keeps a hidden static variable in the data segment up to date while parsing

Further `strtok` Horrors

- Not only does `strtok()` modify the input…

  (You don't even specify which string to tokenize past the first call! Hidden vars!)

```
char input[18] = "This.is my/string";
char* token = strtok(input, " ./");
token = strtok(NULL, " ./");
token = strtok(NULL, " ");
```

- Mixing calls of `strtok()` between different strings is not allowed because it can only process ONE string with its hidden variables!
  - But there is a `strtok_r()` that achieves re-entrancy, allowing the mixing of calls, by requiring you pass in a pointer to a temp variable for it to use

Horrors Explained

- This mixing of `strtok()` calls is easy to do on accident in a large program, especially with functions involved:

```
strtok(input1, …)
function()
    strtok(input2, …)
strtok(input1, …)
```

- The solution is to simply use a more modern language with a string type

Combining Declaration Methods

- What does this mean:

```
char* mystring[3];
```

Combining Declaration Methods

- What does this mean:

```
char* mystring[3];
```

Declare an array of pointers, each of which points to a string; each of these pointers can be pointed at either array names *or* string literals

Remember that an array name is a pointer to the first element's address in memory

Arrays of Pointers to Strings - Example

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

void main()
{
  int currElem = -5;
  int numElems = 3;
  char* mystring[numElems];
  char myarray[10];

  strcpy(myarray, "1ARRAY");

  printf("Size of char*: %d\n", sizeof(char*));
  printf("Size of one array element: %d\n", sizeof(mystring[0]));
  printf("Size of all array elements: %d\n", sizeof(mystring));
  printf("Number of elements in array: %d = %d\n", sizeof(mystring) / sizeof(mystring[0]), numElems);

  //strcpy(mystring[0], "strcpy string");          // Causes seg fault, that's a pointer!
  //printf("mystring[0]: %s\n", mystring[0]);
  mystring[0] = "string literal";                  // Set the first pointer to point to the address of a string literal
  printf("mystring[0]: %s\n", mystring[0]);        // (which is the address of the literal's first element)
  mystring[0] = myarray;                           // Set the first pointer to point to the name of a C string array
  printf("mystring[0]: %s\n", mystring[0]);        // (which is the address of the array's first element)
}
```

Results:

```
Size of char*: 8
Size of one array element: 8
Size of all array elements: 24
Number of elements in array: 3 = 3
mystring[0]: string literal
mystring[0]: 1ARRAY
```

Combining Declaration Methods

- What does this mean:

```
char* mystring[3];
```

Declare an array of pointers, each of which points to a string; each of these pointers can be pointed at either array names *or* string literals

Remember that an array name is a pointer to the first element's address in memory

We can change where the pointers point, but how do we create new strings for this new array to hold?

Dynamically Allocating a String

- To create a string variable dynamically, and thus use it like an array, use `malloc()` and `free()`:

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

void main()
{
  char* mystring;
  char* literal = "literal";

  mystring = malloc(20 * sizeof(char));

  if (mystring == 0)
    printf("malloc() failed!\n");

  memset(mystring, '\0', 20);

  sprintf(mystring, "yay! %s\n", literal);
  printf("%s", mystring);
  mystring[3] = 'Q';
  printf("%s", mystring);

  free(mystring);
}
```

Note that `char* mystring` is editable!

```
$ gcc -o malloctest malloctest.c
$ malloctest
yay! literal
yayQ literal
```

Malloc Memory Leaks

- If you don’t free dynamically allocated memory, it still takes up space
- If you have a long-running program, like a server process, this could eventually use up all of your memory
- Process memory is normally all freed automatically when a process is terminated
  - At least in UNIX, Windows, etc. - some real-time operating systems don’t!

Malloc Memory Leaks

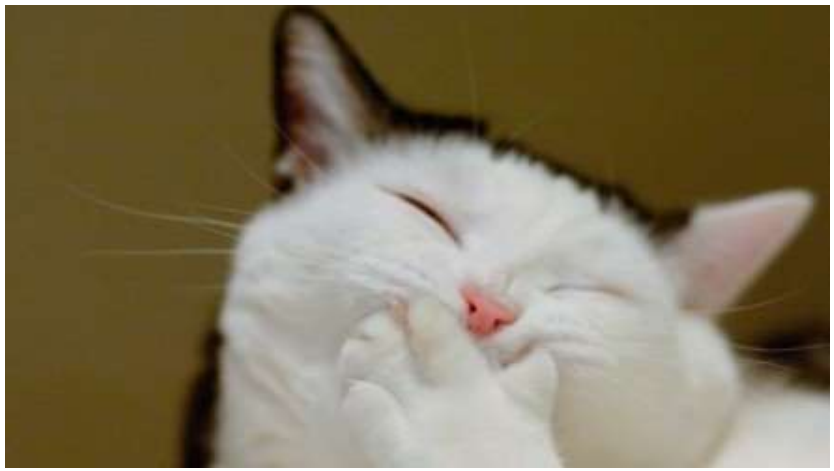
- Here's a classic way to hide and cause a leak:

```
char* mystring = malloc(20 * sizeof(char));
...
mystring = "hello";
```

- This leaks because you no longer have the start address of the dynamically allocated space; `mystring` now points to a string literal

```
free(mystring);  // And if you try this later, it fails spectacularly
```

Spectacular Failing

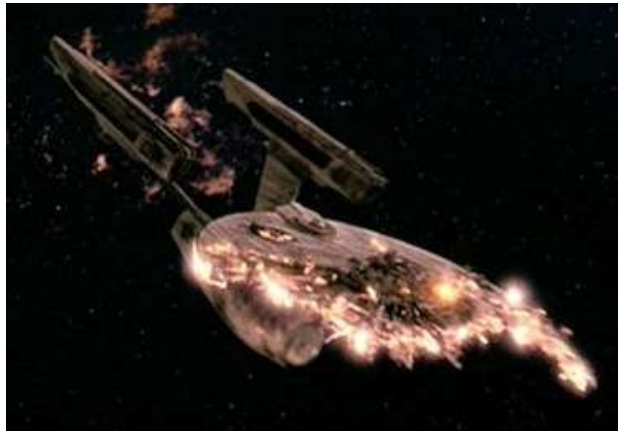


Same program, but let's just put this right in here…

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

void main()
{
  char* mystring;
  char* literal = "literal";

  mystring = malloc(20 * sizeof(char));

  if (mystring == 0)
    printf("malloc() failed!\n");

  memset(mystring, '\0', 20);

  sprintf(mystring, "yay! %s\n", literal);
  printf("%s", mystring);
  mystring[3] = 'Q';
  printf("%s", mystring);
  mystring = "test\n";
  free(mystring);
}
```

Spectacular Failing Results

```
$ malloctest
yay! literal
yayQ literal
*** Error in `malloctest': free(): invalid pointer: 0x0000000000400805 ***
======= Backtrace: =========
/lib64/libc.so.6(+0x7d053)[0x7fc92849c053]
malloctest[0x40074a]
/lib64/libc.so.6(__libc_start_main+0xf5)[0x7fc928440b15]
malloctest[0x4005d9]
======= Memory map: ========
00400000-00401000 r-xp 00000000 00:39 3238103636          /nfs/stak/faculty/b/brewsteb/tempdir/malloctest
00600000-00601000 r--p 00000000 00:39 3238103636          /nfs/stak/faculty/b/brewsteb/tempdir/malloctest
00601000-00602000 rw-p 00001000 00:39 3238103636          /nfs/stak/faculty/b/brewsteb/tempdir/malloctest
01195000-011b6000 rw-p 00000000 00:00 0                   [heap]
7fc924000000-7fc924021000 rw-p 00000000 00:00 0
7fc924021000-7fc928000000 ---p 00000000 00:00 0
7fc928209000-7fc92821e000 r-xp 00000000 fd:02 16777347    /usr/lib64/libgcc_s-4.8.5-20150702.so.1
7fc92821e000-7fc92841d000 ---p 00015000 fd:02 16777347    /usr/lib64/libgcc_s-4.8.5-20150702.so.1
7fc92841d000-7fc92841e000 r--p 00014000 fd:02 16777347    /usr/lib64/libgcc_s-4.8.5-20150702.so.1
7fc92841e000-7fc92841f000 rw-p 00015000 fd:02 16777347    /usr/lib64/libgcc_s-4.8.5-20150702.so.1
7fc92841f000-7fc9285d6000 r-xp 00000000 fd:02 16811513    /usr/lib64/libc-2.17.so
7fc9285d6000-7fc9287d6000 ---p 001b7000 fd:02 16811513    /usr/lib64/libc-2.17.so
7fc9287d6000-7fc9287da000 r--p 001b7000 fd:02 16811513    /usr/lib64/libc-2.17.so
7fc9287da000-7fc9287dc000 rw-p 001bb000 fd:02 16811513    /usr/lib64/libc-2.17.so
7fc9287dc000-7fc9287e1000 rw-p 00000000 00:00 0
7fc9287e1000-7fc928802000 r-xp 00000000 fd:02 16811685    /usr/lib64/ld-2.17.so
7fc9289cf000-7fc9289d2000 rw-p 00000000 00:00 0
7fc9289ff000-7fc928a02000 rw-p 00000000 00:00 0
7fc928a02000-7fc928a03000 r--p 00021000 fd:02 16811685    /usr/lib64/ld-2.17.so
7fc928a03000-7fc928a04000 rw-p 00022000 fd:02 16811685    /usr/lib64/ld-2.17.so
7fc928a04000-7fc928a05000 rw-p 00000000 00:00 0
7ffe32184000-7ffe321a5000 rw-p 00000000 00:00 0           [stack]
7ffe321cd000-7ffe321cf000 r-xp 00000000 00:00 0           [vdso]
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0   [vsyscall]
Aborted (core dumped)
```

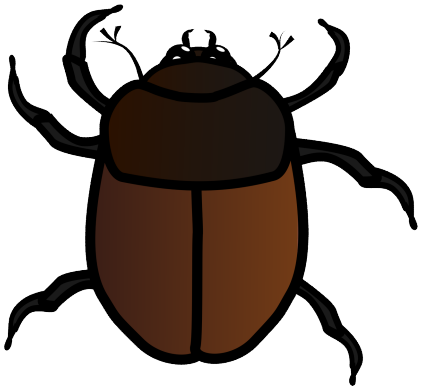


C Debugging

Benjamin Brewster

UNIX C Debugging

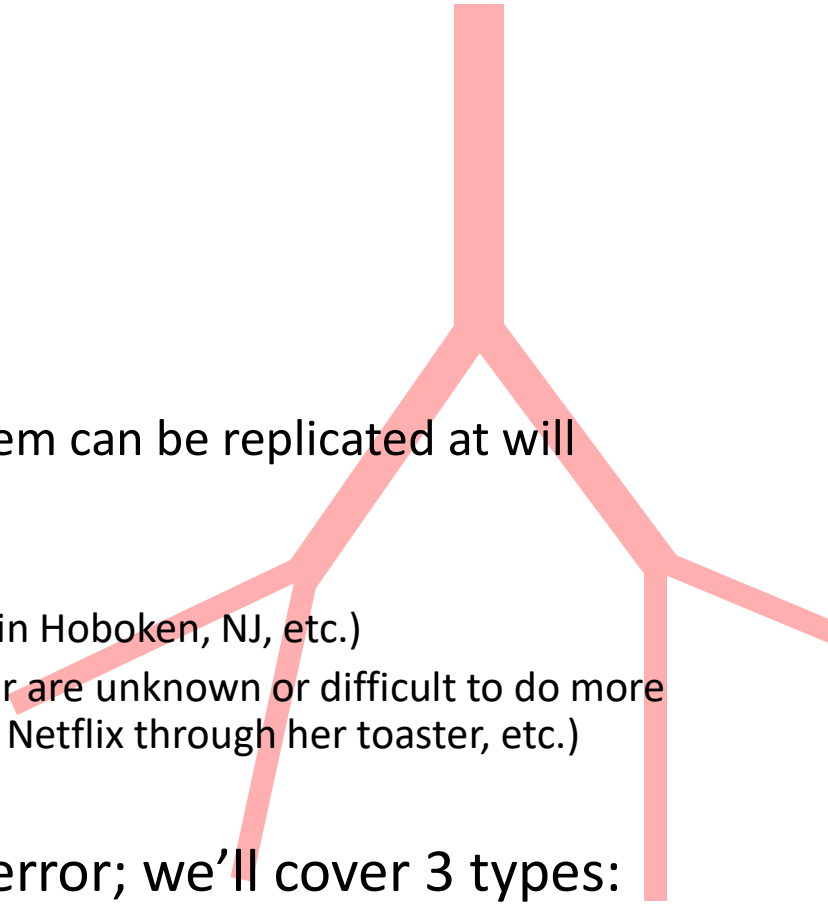
- Just a few notes on debugging…
  - http://en.wikipedia.org/wiki/Software_bug#Etymology



Debugging Process Review

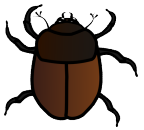
1. Reproduce the problem reliably
   - Simplify input and environment until the problem can be replicated at will
     - e.g. Wolf Fence algorithm
   - Challenges:
     - Unique environment (space station, aunt Edna's PC in Hoboken, NJ, etc.)
     - Particular sequence of events leading up to the error are unknown or difficult to do more than once (lightning strike, aunt Edna tries to watch Netflix through her toaster, etc.)

2. Examine the process state at the time of error; we'll cover 3 types:
   1. Live Examination
   2. Post-mortem Debugging
   3. Trace Statement



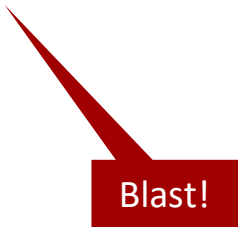
Using a debugger with `gcc`

- Compile with the "-g" option.
  ```
  $ gcc -g testit.c -o testit
  ```
- Then start the debugger on the program
  ```
  $ gdb ./testit
  ```
- In the debugger, some key commands:
  - `run` :: (re)starts the program running; will stop at breakpoint (can add args, e.g.: `run 6 myfile`)
  - `break` :: sets a breakpoint where the debugger will stop and allow you to examine variables or single step
  - `step` :: executes a single line of C code; will enter a function call
  - `next` :: executes a single line of C code; will not enter a function call
  - `continue` :: continues execution again until another breakpoint is hit or the program completes
  - `print` :: prints out a variable
  - `quit` :: stop debugging (exit gdb)



Demo with testit.c

```
$ gcc -o testit testit.c
$ testit
Segmentation fault (core dumped)
```



Demo with testit.c

-g flag compiles with debug symbols

```
$ gcc -g -o testit testit.c
$ gdb testit
[…]
Reading symbols from /nfs/stak/faculty/b/brewsteb/codesamples/gdbdemos/testit...done.
(gdb) run
Starting program: /nfs/stak/faculty/b/brewsteb/codesamples/gdbdemos/testit
Program received signal SIGSEGV, Segmentation fault.
0x00000000004004e3 in main () at testit.c:12
12              temp[2]='F';
```

Set a breakpoint at line 12

```
(gdb) break 12
Breakpoint 1 at 0x4004db: file testit.c, line 12.
(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n) y

Starting program: /nfs/stak/faculty/b/brewsteb/codesamples/gdbdemos/testit

Breakpoint 1, main () at testit.c:10
12              temp[2]='F';
```

Execution pauses just before running line 12

Demo with testit.c

```
(gdb) print temp
$1 = 0x400628 "CS344"
(gdb) list 12
7               char* temp = "CS344";
8
9               int i;
10               i=0;
11
12              temp[2]='F';
13
14              for (i = 0; i < 5 ; i++ )
15                      printf("%c\n", temp[i]);
16
```

Show the contents and address of the `temp` variable

Display the five lines before and after line 12

Demo with testit.c

```
(gdb) print temp
$1 = 0x400628 "CS344"
(gdb) list 12
7               char* temp = "CS344";
8
9               int i;
10               i=0;
11
12              temp[2]='F';
13
14              for (i = 0; i < 5 ; i++ )
15                      printf("%c\n", temp[i]);
16
```

Demo with testit.c

```
7               char* temp = "CS344";
8
9               int i;
10              i=0;
11
12              temp[2]='F';
13
14              for (i = 0; i < 5 ; i++ )
15                      printf("%c\n", temp[i]);
16
```

```
(gdb) jump 13
Continuing at 0x4004e6.
C
S
3
4
4
Adding 6 to 3: 10
Program exited with code 022.
```

Let's see if the rest of this works:

Jump to line 13, skipping line 12, and continue

???

Demo with testit.c

Set a breakpoint on line 13, because jump starts ongoing execution again

```
(gdb) break 13
Breakpoint 2 at 0x4004e6: file testit.c, line 13.
(gdb) info breakpoints
Num     Type           Disp Enb Address            What
1       breakpoint     keep y   0x00000000004004db in main at testit.c:12
        breakpoint already hit 1 time
2       breakpoint     keep y   0x00000000004004e6 in main at testit.c:13
(gdb) run
Starting program: /nfs/stak/faculty/b/brewsteb/codesamples/gdbdemos/testit

Breakpoint 1, main () at testit.c:12
12              temp[2]='F';
(gdb) jump 13
Continuing at 0x4004e6.

Breakpoint 2, main () at testit.c:14
14              for (i = 0; i < 5 ; i++ )
```

Skip line 13, stopping at the next breakpoint

Demo with testit.c

Run the next line of code (14), then display the *next* one (15)

```
(gdb) step
15                      printf("%c\n", temp[i]);
(gdb) print i
$2 = 0
(gdb) step
C
14              for (i = 0; i < 5 ; i++ )
(gdb) print i
$3 = 0
(gdb) step
15                      printf("%c\n", temp[i]);
(gdb) print i
$4 = 1
(gdb) where
#0  main () at testit.c:15
```

`i` has finally updated

Demo with testit.c

This is getting boring iterating through this for loop, so just set a breakpoint in the future (at line 17; we're currently at 15) to skip ahead to find this math bug

```
(gdb) break 17
Breakpoint 3 at 0x40051c: file testit.c, line 17.
(gdb) continue
Continuing.
S
3
4
4
```

Floor it

A function! Let's go in!

```
Breakpoint 3, main () at testit.c:17
17              printf("Adding 6 to 3: %d\n", Add6(3));
(gdb) next
Adding 6 to 3: 10
18      }
(gdb) where
#0  main () at testit.c:18
```

Oops: **next** goes to the next line but *won't* enter functions! I should have used **step**

Demo with testit.c

Alright, restart the whole thing, since we missed our function

```
(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n) y

Starting program: /nfs/stak/faculty/b/brewsteb/codesamples/gdbdemos/testit


Breakpoint 1, main () at testit.c:12
12              temp[2]='F';
(gdb) jump 13
Continuing at 0x4004e6.

Breakpoint 2, main () at testit.c:14
14              for (i = 0; i < 5 ; i++ )
```

Skip past the known seg fault

Demo with testit.c

```
(gdb) continue
Continuing.
C
S
3
4
4

Breakpoint 3, main () at testit.c:17
17              printf("Adding 6 to 3: %d\n", Add6(3));
```

Continue on to the function call

Demo with testit.c

```
(gdb) step
Add6 (in=3) at testit.c:22
22              int six = 7;
(gdb) watch six
Hardware watchpoint 8: six
(gdb) next
Hardware watchpoint 8: six
Old value = 52
New value = 7
Add6 (in=3) at testit.c:24
24              return six + in;
(gdb) continue
Continuing.

Watchpoint 8 deleted because the program has left the block in
which its expression is valid.
0x0000000000400526 in main () at testit.c:17
17              printf("Adding 6 to 3: %d\n", Add6(3));
(gdb) continue
Continuing.
Adding 6 to 3: 10

Program exited with code 022.
(gdb) quit
```

Step into the function!

`gdb` tells us it just entered a function

Oops! six = 6!

`watch` causes `gdb` to pause execution if the variable `six` changes. This could also have been an expression about it's value:

`(gdb) watch six if six > 6`

`gdb` pauses and tells us it deleted a watchpoint

Visual Studio Destroys `gdb`

- Any Integrated Development Environment destroys `gcc` and `gdb`
  - IDEs have code generation, compiling, optimization, organization, debugging, live code step-through, and documenting all built in
- Visual Studio 20XX rocks
- But we don't have access to that in UNIX, so how do we find nasty bugs like memory leaks?

valgrind

- valgrind helps us to find memory leaks in C programs
- Compile with –g to add better diagnostics

```
$ cat leaky.c
// leaky.c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
void main()
{
        char* dynamicBuffer;
        dynamicBuffer = malloc(10);
}
$ gcc –o leaky –g leaky.c
```

`malloc()` but no `free()`

Compile with debug symbols (line numbers, function & variable names, etc.

valgrind – leaky example

```
$ valgrind --leak-check=yes --show-reachable=yes ./leaky
…
==31186== HEAP SUMMARY:
==31186==     in use at exit: 10 bytes in 1 blocks
==31186==   total heap usage: 1 allocs, 0 frees, 10 bytes allocated
==31186==
==31186== 10 bytes in 1 blocks are still reachable in loss record 1 of 1
==31186==    at 0x4A06A2E: malloc (vg_replace_malloc.c:270)
==31186==    by 0x4004D5: main (leaky.c:8)
==31186==
==31186== LEAK SUMMARY:
==31186==    definitely lost: 0 bytes in 0 blocks
==31186==    indirectly lost: 0 bytes in 0 blocks
==31186==      possibly lost: 0 bytes in 0 blocks
==31186==    still reachable: 10 bytes in 1 blocks
==31186==         suppressed: 0 bytes in 0 blocks
==31186==
==31186== For counts of detected and suppressed errors, rerun with: -v
==31186== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 6 from 6)
```

Still reachable refers to `dynamicBuffer`, `malloc`'d on line 8, whose dynamically allocated memory pointer was not overwritten but simply didn't get freed before the program terminated.

Because the OS frees all memory when your process terminates, these can often be safely ignored without consequence, as long as you aren't allocating and forgetting about the memory in a loop...

But it's safest to de-allocate memory as needed to facilitate safe code revisions *later*!

valgrind – leaky2

- A note about valgrind and `printf()`
- This program is the same as leaky.c except for the `printf` statement

```
// leaky2.c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
void main()
{
        char* dynamicBuffer;
        dynamicBuffer = malloc(10);
        printf("This printf causes valgrind to think the malloc pointer is lost\n");
}
```

`malloc()` but no `free()`

`printf()` uses all kinds of internal variables that confuse valgrind into thinking things are worse than they are

valgrind – leaky2

```
$ valgrind --leak-check=yes --show-reachable=yes ./leaky2
…
==8303== Command: ./leaky2
…
This printf causes valgrind to think the malloc pointer is lost
==8303==
==8303== HEAP SUMMARY:
==8303==     in use at exit: 10 bytes in 1 blocks
==8303==   total heap usage: 1 allocs, 0 frees, 10 bytes allocated
==8303==
==8303== 10 bytes in 1 blocks are definitely lost in loss record 1 of 1
==8303==    at 0x4A06A2E: malloc (vg_replace_malloc.c:270)
==8303==    by 0x400515: main (leaky2.c:8)
==8303==
==8303== LEAK SUMMARY:
==8303==    definitely lost: 10 bytes in 1 blocks
==8303==    indirectly lost: 0 bytes in 0 blocks
==8303==      possibly lost: 0 bytes in 0 blocks
==8303==    still reachable: 0 bytes in 0 blocks
==8303==         suppressed: 0 bytes in 0 blocks
==8303==
==8303== For counts of detected and suppressed errors, rerun with: -v
==8303== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 6 from 6)
```

Despite this normally bad warning, you *can* still fix your code and `free()` it, but there's definitely still a memory leak if you don't.

Bad printf!

valgrind – leaky3

- valgrind can also help you discover when you use variables that are uninitialized:

```
// leaky3.c
#include <stdio.h>
void main()
{
        int six;
        printf("six: %d\n", six);
}
```

`six` isn't initialized!

valgrind – leaky3

```
$ valgrind --leak-check=yes --show-reachable=yes ./leaky3
…
==10122== Use of uninitialised value of size 8
==10122==    at 0x334E843A5B: _itoa_word (in /lib64/libc-2.12.so)
==10122==    by 0x334E846612: vfprintf (in /lib64/libc-2.12.so)
==10122==    by 0x334E84F149: printf (in /lib64/libc-2.12.so)
==10122==    by 0x4004E2: main (leaky3.c:6)
==10122==
==10122== Conditional jump or move depends on uninitialised value(s)
==10122==    at 0x334E843A65: _itoa_word (in /lib64/libc-2.12.so)
==10122==    by 0x334E846612: vfprintf (in /lib64/libc-2.12.so)
==10122==    by 0x334E84F149: printf (in /lib64/libc-2.12.so)
==10122==    by 0x4004E2: main (leaky3.c:6)
==10122==
==10122== Conditional jump or move depends on uninitialised value(s)
==10122==    at 0x334E8450A3: vfprintf (in /lib64/libc-2.12.so)
==10122==    by 0x334E84F149: printf (in /lib64/libc-2.12.so)
==10122==    by 0x4004E2: main (leaky3.c:6)
==10122==
==10122== Conditional jump or move depends on uninitialised value(s)
==10122==    at 0x334E8450C1: vfprintf (in /lib64/libc-2.12.so)
==10122==    by 0x334E84F149: printf (in /lib64/libc-2.12.so)
==10122==    by 0x4004E2: main (leaky3.c:6)
…
```

Knowing that the error happened on line 6 is priceless

That's a lot of whining

Mixing Languages

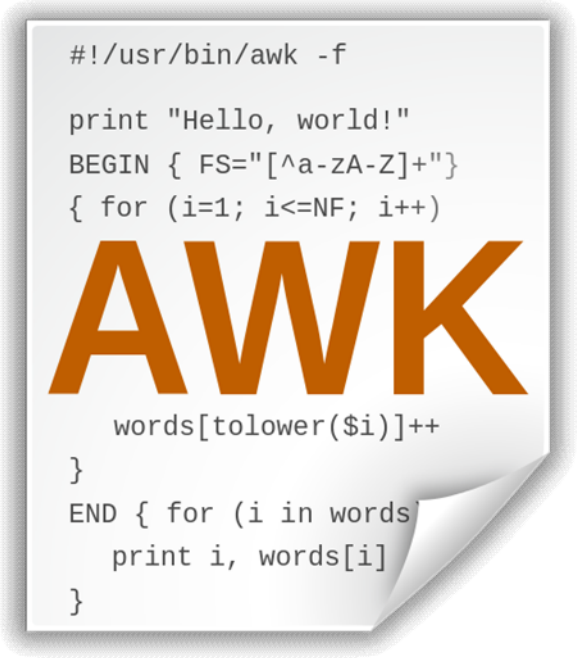
Benjamin Brewster

Scripting Languages

- The bash shell scripting language is not the only standard UNIX scripting language
- We can mix all of these languages and programs together!
- The only other always-built-in scripting language for a UNIX system is `awk`

awk

- awk was invented by
  - Alfred Aho
  - Peter Weinberger
  - Brian Kernighan
- It is commonly used for writing one-line programs on UNIX systems
- Popular early on because it adds computational ability to the command line
  - But now-a-days, we can do this directly in bash with e.g. $(( ))



awk example

- Hello world in awk

```
#!/usr/bin/awk -f
BEGIN { print "Hello, world!"; exit }
```

Associative Arrays

- `awk` features a kind of array called an associative array
- A normal array maps numbers to arbitrary objects (i.e., whatever you pick)
- Here are examples of a normal array mapping integer indexes to strings:
  - 6 maps to "jones"
  - 2 maps to "Nahasapeemapetilon"

Associative Arrays

- An associate array maps arbitrary objects to arbitrary objects
- Here is an example of mapping strings to other strings:
  - "Nahasapeemapetilon" maps to "Apu"
  - "Eat more beef" maps to "Kick less cats“
- Here, an object called MyObject maps to integers:
  - myObj1 maps to 6
  - myObj2 maps to 7

Associative Arrays

- Awk associative array example:

```
myarray[0] = "dog"
myarray["cat"] = "feline"
myarray[3] = 6
```

- This is a sparse array, because there are breaks in the integer numbering from 0 to 3
- An associative arrays is also called:
  - Map, hash, lookup table

Perl

- Perl is a general-purpose programming language
  - Practical Extraction and Report Language
- Written by Larry Wall, released in 1987
- Borrows features from C, shell scripting, awk, sed, Lisp, and others
- Designed to be easy to use, not necessarily elegant

Perl

```
#!/usr/bin/perl
# The traditional first program.

# Strict and warnings are recommended.
use strict;
use warnings;

# Print a message.
print "Hello, World!\n";
```

Perl Camel

- http://perl.postbit.com/photos/other/perl-camel-source-code.html

- Image comes from here, one of the classic O'Reilly books:


```
#!/usr/bin/perl -w
# camel code
                                        $_='ev
                                      al("seek\040D
         ATA,0,                     0;");foreach(1..3)
       {<DATA>}my                  @camel1hump;my$camel;
  my$Camel  ;while(                <DATA>){$_=sprintf("%-6
9s",$_);my@dromedary              1=split(//);if(defined($
_=<DATA>)){@camel1hum            p=split(//);}while(@dromeda
 ry1){my$camel1hump=0           ;my$CAMEL=3;if(defined($_=shif
       t(@dromedary1    ))&&/\S/){$camel1hump+=1<<$CAMEL;}
      $CAMEL--;if(d   efined($_=shift(@dromedary1))&&/\S/){
     $camel1hump+=1  <<$CAMEL;}$CAMEL--;if(defined($_=shift(
    @camel1hump))&&/\S/){$camel1hump+=1<<$CAMEL;}$CAMEL--;if(
    defined($_=shift(@camel1hump))&&/\S/){$camel1hump+=1<<$CAME
    L;;}$camel.=(split(//,"\040..m`{/J\047\134}L^7FX"))[$camel1h
     ump];}$camel.="\n";}@camel1hump=split(/\n/,$camel);foreach(@
     camel1hump){chomp;$Camel=$_;y/LJF7\173\175`\047/\061\062\063\
     064\065\066\067\070/;y/12345678/JL7F\175\173\047`/;$_=reverse;
      print"$_\040$Camel\n";}foreach(@camel1hump){chomp;$Camel=$_;y
       /LJF7\173\175`\047/12345678/;y/12345678/JL7F\175\173\0 47`/;
        $_=reverse;print"\040$_$Camel\n";}';;s/\s*//g;;eval;   eval
          ("seek\040DATA,0,0;");undef$/;$_=<DATA>;s/\s*//g;(   );;s
            ;^.*_;;;map{eval"print\"$_\"";}/.{4}/g; __DATA__   \124
              \1   50\145\040\165\163\145\040\157\1 46\040\1  41\0
                   40\143\141  \155\145\1 54\040\1   51\155\  141
                   \147\145\0  40\151\156 \040\141    \163\16 3\
                    157\143\   151\141\16  4\151\1     57\156
                    \040\167  \151\164\1   50\040\      120\1
                    45\162\   154\040\15    1\163\      040\14
                    1\040\1   64\162\1       41\144       \145\
                    155\14    1\162\        153\04        0\157
                     \146\     040\11      7\047\         122\1
                     45\15      1\154\1  54\171           \040
                     \046\        012\101\16             3\16
                     3\15          7\143\15              1\14
                     1\16           4\145\163            \054
                    \040            \111\156\14         3\056
                   \040\          125\163\145\14         4\040\
                   167\1          51\164\1  50\0         40\160\
                 145\162                                \155\151
               \163\163                                  \151\1
             57\156\056
```

Perl Camel

- http://perl.postbit.com/photos/other/perl-camel-source-code.html

Python

- Similar philosophies as Perl, but now far more widespread than Perl
- In active development and usage
- Python is faster, with better support for Object Oriented Programming

Perl & Python

- Perl & Python are interpreted languages
- When you want to run code you've written, it is first read by an interpreter, slightly optimized ("compiled"), and then executed.
- Perl can only be interpreted by `perl` (the Perl interpreter), Python is interpreted by `python`

Python – Example Scripts

```
#!/usr/bin/python
print "Hello World!";
```

```
#!/usr/bin/python
# Create a file for writing
file = open("myfile.dat", "w+")
file.write("STUFF N JUNK”)
```

Python – Running an Example Script

```
$ cat pythontest
#!/usr/bin/python
print "Hello, World!";
$ chmod +x pythontest
$ pythontest
Hello, World!
```

Building a String in Python

```
$ cat pythonstring
#!/usr/bin/python
# comments!
mystring = "SO"
mystring += " MUCH "
mystring += "EASIER"
print "THIS IS " + mystring + " " + str(9) + " TIMES";
```

Converts the number into a string

```
$ pythonstring
THIS IS SO MUCH EASIER 9 TIMES
```

Python Math (Python 3)

```
$ cat pythonmath
#!/usr/bin/python
six = 6;
seven = 7;
thirteen = six + seven;
print("How much: {0}".format(str(thirteen)));
$ pythonmath
How much: 13
```

`printf()`-like functionality

Python versus C – A Vast Difference in Speed

C

```
$ cat c-billion.c
void main()
{
        long i, j;
        for (i = 0; i <= 1000000000; i++)
                j = i * i;
}
$ gcc -o c-billion c-billion.c
$ /usr/bin/time --format='%C took %e seconds' c-billion
c-billion took 2.78 seconds
```

- Both of these programs count to a billion

Python

```
$ cat python-billion
#!/usr/bin/python
for i in range(0, 1000000000):
        j = i * i;
$ /usr/bin/time --format='%C took %e seconds' python-billion
python-billion took 160.15 seconds
```

Mixing Languages

- Scripting languages and compiled languages can call each other

- This allows us to combine the best parts of one with the other, e.g.:
  - Speed == C
  - Short and easy to program == Python

Mixing C into Python

- This Python program calls a C program (it could have been a binary from any language) which *doesn't* return any results back to the Python script:

```
$ gcc –o c-billion c-billion.c
$ cat python-billion-fast
#!/usr/bin/python
from subprocess import call
call("./c-billion")
$ /usr/bin/time --format='%C took %e seconds' python-billion-fast
python-billion-fast took 2.91 seconds
```

Mixing C into Python

- Ways to get data back into Python:
  - Have the C program write a datafile, which is read by Python
  - Create a UNIX pipe, from which both Python and C can read and write
  - Create a C function inside the Python program with the “instant” module
  - Several other complex ways involving the Python C API, ctypes, SWIG, Boost Python API, etc., all of which use additional wrappers or APIs to manipulate and transmit data
- By the end of the course, you should be able to do the first two
- The others are non-trivial but are effective

Mixing Python into C

- You can write a C program that calls Python and returns the value back to C

- But why? Because many file and string handling tasks, especially extensive ones, are easier in Python

- Official example of this:

  https://docs.python.org/release/2.6.5/extending/embedding.html#pure-embedding

Mixing C into bash Shell Scripting

```
$ cat addsix-c.c
#include <stdio.h>
int main(int argc, char* argv[])
{
        printf("%d", atoi(argv[1]) + 6);
        return 0;
}
$ gcc -o addsix-c addsix-c.c
$ cat addsix-bash
#!/bin/bash
value=4
printf "value: %d\n" $value
value=$(./addsix-c $value)
printf "value + addsix: %d\n" $value
$ chmod +x addsix-bash
$ addsix-bash
value: 4
value + addsix: 10
```

This could be any pre-compiled binary, not just a C binary

It could even be another shell script

Mixing C into bash Shell Scripting

```
$ cat addsix-c.c
#include <stdio.h>
int main(int argc, char* argv[])
{
        printf("%d", atoi(argv[1]) + 6);
        return 0;
}
$ gcc -o addsix-c addsix-c.c
$ cat addsix-bash
#!/bin/bash
value=4
printf "value: %d\n" $value
value=$(./addsix-c $value)
printf "value + addsix: %d\n" $value
$ chmod +x addsix-bash
$ addsix-bash
value: 4
value + addsix: 10
```

Note: if this line is instead:

```
value=$("./addsix-c 4")
```

Then this line fails with an error because `"./addsix-c 4"` is not the name of a program; program names don't have spaces