Review of C Programming

MTSU CSCI 3240

Spring 2016

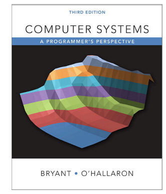
Dr. Hyrum D. Carroll

Materials from CMU and Dr. Butler

Textbooks

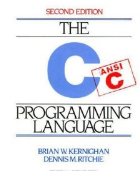
Required

- Randal E. Bryant and David R. O'Hallaron,
  - "Computer Systems: A Programmer's Perspective 3rd Edition", Prentice Hall 2015.
  - csapp.cs.cmu.edu
  - Most of the slide materials in this class are based on material provided by Bryant and O'Hallaron



Recommended

- Brian Kernighan and Dennis Ritchie,
  - "The C Programming Language, Second Edition", Prentice Hall, 1988



Why C?

Used prevalently

- Operating systems (e.g. Linux, FreeBSD/OS X, windows)
- Web servers (apache)
- Web browsers (firefox)
- Mail servers (sendmail, postfix, uw-imap)
- DNS servers (bind)
- Video games (any FPS)
- Graphics card programming (OpenCL GPGPU programming based on C)

Why?

- Performance
- Portability
- Wealth of programmers

Why C?

Compared to other high-level languages (HLLs)

- Maps almost directly into hardware instructions making code potentially more efficient
  - Provides minimal set of abstractions compared to other HLLs
  - HLLs make programming simpler at the expense of efficiency

Compared to assembly programming

- Abstracts out hardware (i.e. registers, memory addresses) to make code portable and easier to write
- Provides variables, functions, arrays, complex arithmetic and boolean expressions

Why assembly along with C?

Learn how programs map onto underlying hardware

- Allows programmers to write efficient code

Perform platform-specific tasks

- Access and manipulate hardware-specific registers
- Interface with hardware devices
- Utilize latest CPU instructions

Reverse-engineer unknown binary code

- Analyze security problems caused by CPU architecture
- Identify what viruses, spyware, rootkits, and other malware are doing
- Understand how cheating in on-line games work

The C Programming Language

Simpler than C++, C#, Java

- No support for
  - Objects
  - Memory management
  - Array bounds checking
  - Non-scalar operations
- Simple support for
  - Typing
  - Structures
- Basic utility functions supplied by libraries
  - libc, libpthread, libm
- Low-level, direct access to machine memory (pointers)
- Easier to write bugs, harder to write programs, typically faster
  - Looks better on a resume

C based on updates to ANSI-C standard

- Current version: C99

The C Programming Language

Compilation down to machine code as in C++

- Compiled, assembled, linked via gcc

Compared to interpreted languages…

- Python / Perl / Ruby / Javascript
  - Commands executed by run-time interpreter
  - Interpreter runs natively
- Java
  - Compilation to virtual machine "byte code"
  - Byte code interpreted by virtual machine software
  - Virtual machine runs natively
  - Exception: "Just-In-Time" (JIT) compilation to machine code

Our environment

All programs must run on `system64`

- ssh USER@system64.cs.mtsu.edu

Architecture this semester will be x86-64

GNU gcc compiler

- gcc –o hello hello.c

GNU gdb debugger

- ddd is a graphical front end to gdb
- "gdb -tui" is a graphical curses interface to gdb
- Must use "-g" flag when compiling and remove –O flags
  - gcc –g hello.c
  - Add debug symbols and do not reorder instructions for performance

GCC

- Used to compile C/C++ projects
  - List the files that will be compiled to form an executable
  - Specify options via flags
- Important Flags:
  - -g: produce debug information (important; used by GDB/valgrind)
  - -Werror: treat all warnings as errors (this should be your default)
  - -Wall/-Wextra: enable all construction warnings
  - -pedantic: indicate all mandatory diagnostics listed in C-standard
  - -O0/-O1/-O2: optimization levels
  - -o <filename>: name output binary file 'filename'
- Example:
  - gcc -g -Werror -Wall -Wextra -pedantic foo.c bar.c -o baz

Variables

Named using letters, numbers, some special characters

- By convention, not all capitals

Must be declared before use

- Contrast to typical scripting languages (Python, Perl, PHP, JavaScript)
- C is statically typed (for the most part)

Data Types and Sizes

C Data Type	Typical 32-bit	Typical 64-bit	x86-64
`char`	1	1	1
`short`	2	2	2
`int`	4	4	4
`long`	4	8	8
`float`	4	4	4
`double`	8	8	8
`long double`	–	–	10/16
pointer	4	8	8

Constants

Integer literals

1234, 077
0xFE, 0xab78

Character constants

'a' – numeric value of character 'a'
char letterA = 'a';
int asciiA = 'a';
} What's the difference?

String Literals

"I am a string"
"" // empty string

Constant pointers

Used for static arrays

- Symbol that points to a fixed location in memory

char amsg[ ] = "This is a test"; → This is a test\0

- Can change change characters in string (amsg[8] = '!';)
- Can not reassign amsg to point elsewhere (i.e. amsg = p)

Declarations and Operators

Variable declaration can include initialization

```
int foo = 34;
char *ptr = "fubar";
float ff = 34.99;
```

Arithmetic operators

- +, - , *, /, %
- Modulus operator (%)

Expressions

In C, oddly, assignment is an expression

- "x = 4" has the value 4

if (x == 4)
y = 3; /* sets y to 3 if x is 4 */

if (x = 4)
y = 3; /* always sets y to 3 (and x to 4) */

while ((c=getchar()) != EOF)

Increment and Decrement

Comes in prefix and postfix flavors

- i++, ++i
- i--, --i

Makes a difference in evaluating complex statements

- A major source of bugs
- Prefix: increment happens before evaluation
- Postfix: increment happens after evaluation

When the actual increment/decrement occurs is important to know about

- Is "i++ * 2" the same as "++I * 2" ?

Simple data types

datatype	size	values
char	1	-128 to 127
short	2	-32,768 to 32,767
int	4	-2,147,483,648 to 2,147,483,647
long	4	-2,147,483,648 to 2,147,483,647
float	4	3.4E+/-38 (7 digits)
double	8	1.7E+/-308 (15 digits long)

Error-handling Note

Error handling

- No "throw/catch" exceptions for functions in C
- Must look at return values or install global signal handlers (see Chapter 8)

Dynamic memory-allocation note

Dynamic memory

- Managed languages such as Java perform memory management (ie garbage collection) for programmers
- C requires the programmer to *explicitly* allocate and deallocate memory
- No “new” for a high-level object
- Memory can be allocated dynamically during run-time with `malloc()` and deallocated using `free()`
- Must supply the size of memory you want explicitly

“Typical” program

```
#include <stdio.h>
int main(int argc, char* argv[])
{
  /* print a greeting */
  printf("Good evening!\n");
  return 0;
}
```

```
$ gcc -o goodevening goodevening.c
$ ./goodevening
Good evening!
$
```

Breaking down the code

`#include <stdio.h>`

- Include the contents of the file stdio.h
  - Case sensitive – lower case only
- No semicolon at the end of line

`int main(…)`

- The OS calls this function when the program starts running.

`printf(format_string, arg1, …)`

- Call function from libc library
- Prints out a string, specified by the format string and the arguments.

Command Line Arguments (1)

main has two arguments from the command line

`int main(int argc, char* argv[])`

`argc`

- Number of arguments (including program name)

`argv`

- Pointer to an array of string pointers
  - `argv[0]`: = program name
  - `argv[1]`: = first argument
  - `argv[argc-1]`: last argument

- Example: find . –print
  - argc = 3
  - argv[0] = “find”
  - argv[1] = “.”
  - argv[2] = “-print”

Command Line Arguments (2)

```
#include <stdio.h>

int main(int argc, char* argv[])
{
  int i;
  printf("%d arguments\n", argc);
  for(i = 0; i < argc; i++)
    printf("  %d: %s\n", i, argv[i]);
  return 0;
}
```

Command Line Arguments (3)

```
$ ./cmdline The Class That Gives MTSU Its Zip
8 arguments
  0: ./cmdline
  1: The
  2: Class
  3: That
  4: Gives
  5: MTSU
  6: Its
  7: Zip
$
```

Arrays

```
char foo[80];
```

- An array of 80 characters (stored contiguously in memory)
  - `sizeof(foo)`
    = 80 × `sizeof(char)`
    = 80 × 1 = 80 bytes

```
int bar[40];
```

- An array of 40 integers (stored contiguously in memory)
  - `sizeof(bar)`
    = 40 × `sizeof(int)`
    = 40 × 4 = 160 bytes

Structures (structs)

Aggregate data

```
#include <stdio.h>

struct person
{
  char*     name;
  int       age;
}; /* <== DO NOT FORGET the semicolon */

int main(int argc, char* argv[])
{
  struct person potter;
  potter.name = "Harry Potter";
  potter.age = 15;

  printf("%s is %d years old\n", potter.name, potter.age);

  return 0;
}
```

Structs

- Collection of values placed under one name in a single block of memory
  - Can put structs, arrays in other structs
- Given a struct *instance*, access the fields using the '.' operator
- Given a struct *pointer*, access the fields using the '->' operator

```
struct foo_s {
   int a;
   char b;
};
```

```
struct bar_s {
   char ar[10];
   foo_s baz;
};
```

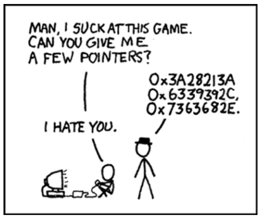
```
bar_s biz; // bar_s instance
biz.ar[0] = 'a';
biz.baz.a = 42;
bar_s* boz = &biz; // bar_s ptr
boz->baz.b = 'b';
```

Pointers

Pointers are variables that hold an address in memory.

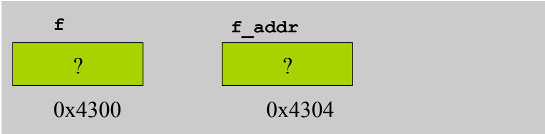
That address contains another variable.

Unique to C and C-like languages

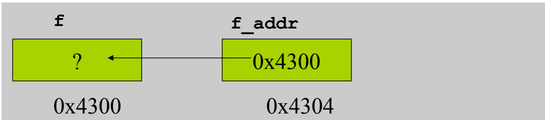


Using Pointers (1)

```
float f;        /* data variable */
float *f_addr;  /* pointer variable */
```

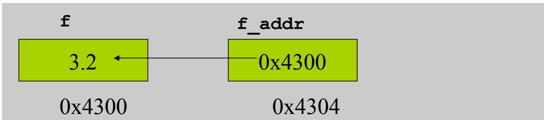



```
f_addr = &f;    /* & = address operator */
```

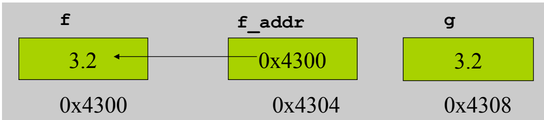


Using Pointers (2)

```
*f_addr = 3.2; /* indirection operator */
```

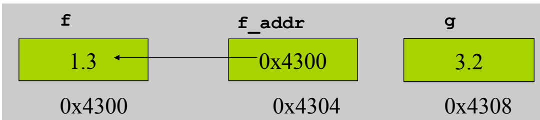



```
float g = *f_addr;/* indirection: g is now 3.2 */
```



Using Pointers (3)

```
f = 1.3;                /* but g is still 3.2 */
```



Pointers To Pointers (etc)

```
int i, j;
int  *v;
int  **m;
v = malloc(NROWS * NCOLS * sizeof(int));
m = malloc(NROWS * sizeof(int *));
for (i=0; i < NROWS; i++)
    m[i] = v + (NCOLS * i);
```

cReview/malloc2DArray.c

Pointer Arithmetic

- Can add/subtract from an address to get a new address
  - Generally, you should avoid doing this (Only perform when absolutely necessary)
  - Result depends on the pointer type
- `A+i`, where `A` is a pointer: `0x100`, `i` is an `int` *(x86-64)*
  - `int*  A`: `A+i = 0x100 + sizeof(int)  * i = 0x100 + 4 * i`
  - `char* A`: `A+i = 0x100 + sizeof(char) * i = 0x100 + i`
  - `int** A`: `A+i = 0x100 + sizeof(int*) * i = 0x100 + 8 * i`
- Rule of thumb: cast pointer explicitly to avoid confusion
  - Prefer `(char*)(A) + i` vs `A + i`, even if `char* A`
  - Absolutely do this in macros

Function calls (static)

Calls to functions typically static (resolved at compile-time)

```
void print_ints(int a, int b)  {
  printf("%d %d\n",a,b);
}

int main(int argc, char* argv[]) {
   int i=3;
   int j=4;
   print_ints(i,j);
}
```

Function call parameters

Function arguments are passed "by value".

What is "pass by value"?

- The called function is given a copy of the arguments.

What does this imply?

- The called function can't alter a variable in the caller function, but its private copy.

Examples

Example 1: swap_1

```
void swap_1(int a, int b)
{
  int temp;
  temp = a;
  a = b;
  b = temp;
}
```

Q: Let x=3, y=4,
after swap_1(x,y);
x =? y=?

~~A: x=4; y=3;~~

B: x=3; y=4;

Example 2: swap_2

```
void swap_2(int *a, int *b)
{
  int temp;
  temp = *a;
  *a = *b;
  *b = temp;
}
```

Q: Let x=3, y=4, after swap_2(&x,&y); x =? y=?

A: x=4; y=3;

~~B: x=3; y=4;~~

Is this pass by value?

Call by value vs. reference in C

Call by reference implemented via pointer passing

```
void swap(int *px, int *py) {
  int tmp;
  tmp = *px;
  *px = *py;
  *py = tmp;
}
```

- Swaps the values of the variables x and y if px is &x and py is &y
- Uses integer pointers instead of integers

Otherwise, call by value...

```
void swap(int x, int y) {
    int tmp;
    tmp = x;
    x = y;
    y = tmp;
  }
```

Function calls (dynamic)

Using function pointers, C can support late-binding of functions where calls are determined at run-time

```
#include <stdio.h>
void print_even(int i){ printf("Even %d\n",i);}
void print_odd(int i) { printf("Odd %d\n",i); }

int main(int argc, char **argv) {
     void (*fp)(int);
     int i = argc;

     if(argc%2){
          fp=print_even;
     }else{
          fp=print_odd;
     }
     fp(i);
}
```

```
% ./funcp a
Even 2
% ./funcp a b
Odd 3
```

Casting

- Can cast a variable to a different type
- Integer Type Casting:
  - signed <-> unsigned: change interpretation of most significant bit
  - smaller signed -> larger signed: sign-extend (duplicate the sign bit)
  - smaller unsigned -> larger unsigned: zero-extend (duplicate 0)
- Cautions:
  - cast explicitly, out of practice. C will cast operations involving different types implicitly, often leading to errors
  - never cast to a smaller type; will truncate (lose) data
  - never cast a pointer to a larger type and dereference it, this accesses memory with undefined contents

Typedefs

- Creates an *alias* type name for a different type
- Useful to simplify names of complex data types

```
struct list_node {
   int x;
};

typedef int pixel;
typedef struct list_node* node;
typedef int (*cmp)(int e1, int e2);

pixel x; // int type
node foo; // struct list_node* type
cmp int_cmp; // int (*cmp)(int e1, int e2) type
```

Macros

- Fragment of code given a name; replace occurrence of name with contents of macro
  - No function call overhead, type neutral
- Uses:
  - defining constants (INT_MAX, ARRAY_SIZE)
  - defining simple operations (MAX(a, b))
- Warnings:
  - Use parentheses around arguments/expressions, to avoid problems after substitution
  - Do not pass expressions with side effects as arguments to macros

```
#define INT_MAX 0x7FFFFFFF
#define MAX(A, B) ((A) > (B) ? (A) : (B))
#define REQUIRES(COND) assert(COND)
#define WORD_SIZE 4
#define NEXT_WORD(a) ((char*)(a) + WORD_SIZE)
```

Header Files

- Includes C declarations and macro definitions to be shared across multiple files
  - Only include function prototypes/macros; no implementation code!
- Usage: #include <header.h>
  - #include <lib> for standard libraries (eg #include <string.h>)
  - #include "file" for your source files (eg #include "header.h")
  - Never include .c files (bad practice)

```
// list.h
struct list_node {
   int data;
   struct list_node* next;
};
typedef struct list_node* node;

node new_list();
void add_node(int e, node l);
```

```
// list.c
#include "list.h"

node new_list() {
   // implementation
}

void add_node(int e, node l) {
   // implementation
}
```

```
// stacks.h
#include "list.h"
struct stack_head {
   node top;
   node bottom;
};
typedef struct stack_head* stack

stack new_stack();
void push(int e, stack S);
```

Header Guards

- Double-inclusion problem: include same header file twice

```
//grandfather.h
```

```
//father.h
#include "grandfather.h"
```

```
//child.h
#include "father.h"
#include "grandfather.h"
```

Error: child.h includes grandfather.h twice

- Solution: header guard ensures single inclusion

```
//grandfather.h
#ifndef GRANDFATHER_H
#define GRANDFATHER_H


#endif
```

```
//father.h
#ifndef FATHER_H
#define FATHER_H


#endif
```

```
//child.h
#include "father.h"
#include "grandfather.h"
```

Odds and Ends

- Prefix vs Postfix increment/decrement
  - a++: use a in the expression, then increment a
  - ++a: increment a, then use a in the expression
- Switch Statements:
  - remember break statements after every case, unless you want fall through (may be desirable in some cases)
  - should probably use a default case
- Variable/function modifiers:
  - global variables: defined outside functions, seen by all files
  - static variables/functions: seen only in file it's declared in
  - Refer to K&R for other modifiers and their meanings

The Standard C Library

The C Standard Library

Common functions we don't need to write ourselves
- Provides a portable interface to many system calls

Analogous to class libraries in Java or C++

Function prototypes declared in standard header files

```
#include <stdio.h>     #include <stddef.h>
#include <time.h>      #include <math.h>
#include <string.h>    #include <stdarg.h>
#include <stdlib.h>
```

- Must include the appropriate ".h" in source code
  - "man 3 printf" shows which header file to include
- K&R Appendix B lists many original functions

Code linked in automatically
- At compile time (if statically linked gcc -static)
- At run time (if dynamically linked)
  - Use "ldd" command to list dependencies
- Use "file" command to determine binary type

The C Standard Library

Examples (for this class)
- I/O
  - printf, scanf, puts, gets, open, close, read, write
  - fprintf, fscanf, … , fseek
- Memory operations
  - memcpy, memcmp, memset, malloc, free
- String operations
  - strlen, strncpy, strncat, strncmp
  - strtod, strtol, strtoul

The C Standard Library

Examples

- Utility functions
  - `rand, srand, exit, system, getenv`
- Time
  - `clock, time, gettimeofday`
- Jumps
  - `setjmp, longjmp`
- Processes
  - `fork, execve`
- Signals
  - `signal, raise, wait, waitpid`
- Implementation-defined constants
  - `INT_MAX, INT_MIN, DBL_MAX, DBL_MIN`

I/O

Formatted output

- `int printf(char *format, …)`
  - Sends output to standard output
- `int fprintf(FILE *stream, const char *format, ...);`
  - Sends output to a file
- `int sprintf(char *str, char *format, …)`
  - Sends output to a string variable
- Return value
  - Number of characters printed (not including trailing \0)
  - On error, a negative value is returned

I/O

Format string composed of ordinary characters (except '%')

- Copied unchanged into the output

Format directives specifications (start with %)

- Character (%c), String (%s), Integer (%d), Float (%f)
- Formatting commands for padding or truncating output and for left/right justification
  - %10s => Pad short string to 10 characters, right justified
  - %-10s => Pad short string to 10 characters, left justified
  - %.10s => Truncate long strings after 10 characters
  - %10.15 => Pad to 10, but truncate after 15, right justified
- Fetches one or more arguments

For more details: `man 3 printf`

I/O

```
#include <stdio.h>
main() {
  char *p;
  char *q;

  float f,g;

  p = "This is a test";
  q = "This is a test";
  f = 909.2153258;

  printf(":%10.15s:\n",p); /* right justified, truncate to 15, pad to 10 */
  printf(":%15.10s:\n",q); /* right justified, truncate to 10, pad to 15 */
  printf(":%0.2f:\n",f);  /* Cut off anything after 2nd decimal, No pad */
  printf(":%15.5f:\n",f); /* Cut off anything after 5th decimal, Pad to 15 */

  return 0;
}
```

```
OUTPUT
% ./strs
:This is a test:
:     This is a :
:909.22:
:      909.21533:
```

I/O

Formatted input

- `int scanf(char *format, …)`
  - Read formatted input from standard input
- `int fscanf(FILE *stream, const char *format, ...);`
  - Read formatted input from a file
- `int sscanf(char *str, char *format, …)`
  - Read formatted input from a string
- Return value
  - Number of input items assigned
- Note
  - Requires pointer arguments

Example: scanf

```
#include <stdio.h>

int main()
{
  int  x;
  scanf("%d\n", &x);
  printf("%d\n", x);
}
```

Q: Why are pointers given to scanf?

A: We need to assign the value to x.

I/O

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
        int a, b, c;
        printf("Enter the first value: ");
        if (scanf("%d",&a) == 0) {
            perror("Input error\n");
            exit(255);
        }
        printf("Enter the second value: ");
        if (scanf("%d",&b) == 0) {
            perror("Input error\n");
            exit(255);
        }
        c = a + b;
        printf("%d + %d = %d\n", a, b, c);
     return 0;
}


OUTPUT
% ./scanf
Enter the first value: 20
Enter the second value: 30
20 + 30 = 50
```

I/O

Line-based input

- `char *gets(char *s);`
  - Reads the next input line from `stdin` into buffer pointed to by `s`
  - Null terminates

Line-based output

- `int puts(char *line);`
  - Outputs string pointed to by `line` followed by newline character to `stdout`

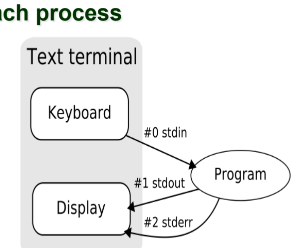
I/O

Direct system call interface

- `open()` = returns an integer file descriptor
- `read()`, `write()` = takes file descriptor as parameter
- `close()` = closes file and file descriptor

Standard file descriptors for each process

- Standard input (keyboard)
  - `stdin or 0`
- Standard output (display)
  - `stdout or 1`
- Standard error (display)
  - `stderr or 2`



Error handling

Standard error (`stderr`)

- Used by programs to signal error conditions
- By default, `stderr` is sent to display
- Must redirect explicitly even if `stdout` sent to file

```
fprintf(stderr, "getline: error on input\n");
perror("getline: error on input");
```

- Typically used in conjunction with `errno` return error code
  - `errno` = single global variable in all C programs
  - Integer that specifies the type of error
  - Each call has its own mappings of `errno` to cause
  - Used with `perror` to signal which error occurred

Example

```
#include <stdio.h>
#include <fcntl.h>
#define BUFSIZE 16
int main(int argc, char* argv[]) {
        int f1,n;
        char buf[BUFSIZE];
        long int f2;

        if ((f1 = open(argv[1], O_RDONLY, 0)) == -1)
            perror("cp: can't open file");
        do {
          if ((n=read(f1,buf,BUFSIZE)) > 0)
            if (write(1, buf, n) != n)
                perror("cp: write error to stdout");
        } while(n==BUFSIZE);
        return 0;
}
```

```
% cat opentest.txt
This is a test of
the open(), read(),
and write() calls.
% ./opentest opentest.txt
This is a test of
the open(), read(),
and write() calls.
% ./opentest asdfasdf
cp: can't open file: No such file or directory
%
```

I/O

Using standard file descriptors in shell

- Redirecting to/from files
  - `ls -l > outfile`
    - » redirects output to "`outfile`"
  - `./a.out < infile`
    - » standard input taken from "`infile`"
  - `ls -l > outfile 2> errorfile`
    - » `sends standard error and standard out to separate files`
- Connecting them to each other via Unix pipes
  - `ls -l | egrep tar`
    - » standard output of "`ls`" sent to standard input of "`egrep`"

I/O via file interface

Supports formatted, line-based and direct I/O

- Calls similar to analogous calls previously covered

Opening a file

- `FILE *fopen(char *name, char *mode);`
  - Opens a file if we have access permission
  - Returns a pointer to a file

```
FILE *fp;
fp = fopen("/tmp/x", "r");
```

Once the file is opened, we can read/write to it

- `fscanf, fread, fgets, fprintf, fwrite, fputs`
- Must supply `FILE*` argument for each call

Closing a file after use

- `int fclose(fp);`
  - Closes the file pointer and flushes any output associated with it

I/O via file interface

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

main(int argc, char** argv)
{
  int i;
  char *p;
  FILE *fp;

  fp = fopen("tmpfile.txt","w+");
  p = argv[1];
  fwrite(p, strlen(p), 1, fp);
  fclose(fp);
  return 0;
}


OUTPUT:
% ./fops HELLO
% cat tmpfile.txt
HELLO
%
```

Memory allocation and management

malloc

- Dynamically allocates memory from the heap
  - Memory persists between function invocations (unlike local variables)
- Returns a pointer to a block of at least `size` bytes – not zero filled!
  - Allocate an integer

```
int* iptr =(int*) malloc(sizeof(int));
```

  - Allocate a structure

```
struct name* nameptr = (struct name*)
   malloc(sizeof(struct name));
```

  - Allocate an integer array with "value" elements

```
int *ptr = (int *) malloc(value * sizeof(int));
```

Be careful to allocate enough memory

- Overrun on the space is undefined
- Common error:

```
char *cp = (char *) malloc(strlen(buf)*sizeof(char))
```

  - `strlen` doesn't account for the NULL terminator
- Fix:

```
char *cp = (char *) malloc((strlen(buf)+1)*sizeof(char))
```

Memory allocation and management

free

- Deallocates memory in heap.
- Pass in a pointer that was returned by `malloc`.
- Integer example

```
int *iptr = (int*) malloc(sizeof(int));
free(iptr);
```

- Structure example

```
struct table* tp = (struct table*)malloc(sizeof(struct table));
free(tp);
```

Freeing the same memory block twice corrupts memory and leads to exploits

Memory allocation and management

Sometimes, before you use memory returned by malloc, you want to zero it

- Or maybe set it to a specific value

`memset()` sets a chunk of memory to a specific value

- `void *memset(void *s, int c, size_t n);`

Set this memory to this value for this length

Memory allocation and management

Because not all data consists of text strings…

```
void *memcpy(void *dest, void *src, size_t n);
void *memmove(void *dest, void *src, size_t n);
```

Malloc, Free, Calloc

- Handle dynamic memory
- `void* malloc (size_t size):`
  - allocate block of memory of `size` bytes
  - does not initialize memory
- `void* calloc (size_t num, size_t size):`
  - allocate block of memory for array of `num` elements, each `size` bytes long
  - initializes memory to zero values
- `void free(void* ptr):`
  - frees memory block, previously allocated by malloc, calloc, realloc, pointed by `ptr`
  - use exactly once for each pointer you allocate
- `size` argument*:*
  - *should* be computed using the `sizeof` operator
  - sizeof: takes a type and gives you its size
  - e.g., `sizeof(int)`, `sizeof(int*)`

Memory Management Rules

- Malloc what you free, free what you malloc
  - client should free memory allocated by client code
  - library should free memory allocated by library code
- Number mallocs = Number frees
  - Number mallocs > Number Frees: definitely a memory leak
  - Number mallocs < Number Frees: definitely a double free
- Free a malloced block exactly once
  - Should not dereference a freed memory block

Stack Vs Heap Allocation

- Local variables and function arguments are placed on the *stack*
  - deallocated after the variable leaves scope
  - *do not* return a pointer to a stack-allocated variable!
  - *do not* reference the address of a variable outside its scope!
- Memory blocks allocated by calls to malloc/calloc are placed on the *heap*
- Globals, constants are placed elsewhere
- Example:
  - // a is a pointer on the *stack* to a memory block on the *heap*
  - int* a = malloc(sizeof(int));

Strings

String functions are provided in an ANSI standard string library.

`#include <string.h>`

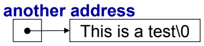
- Includes functions such as:
  - Computing length of string
  - Copying strings
  - Concatenating strings

Strings

In C, a string is an array of characters terminated with the “null” character (‘\0’, value = 0).

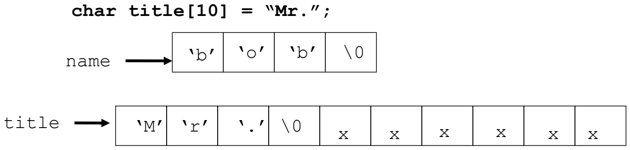
- Character pointer p
  - Sets p to address of a character array
  - p can be reassigned to another address

char *p = “This is a test”;



- Examples

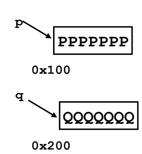
```
char name[4] = “bob”;
char title[10] = “Mr.”;
```



Copying strings

Consider

```
char* p=“PPPPPPP”;
char* q=“QQQQQQQ”;
p = q;
```



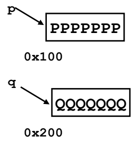
What does this do?

1. Copy QQQQQQ into 0x100?
2. Set p to 0x200

Copying strings

Consider

```
char* p="PPPPPPP";
char* q="QQQQQQQ";
p = q;
```

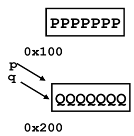


What does this do?

- ~~A) Copy QQQQQQ into 0x100?~~
- B) Set p to 0x200

Copying strings

1. Must manually copy characters
2. Or use `strncpy` to copy characters



Strings

Assignment(=) and equality (==) operators

```
char *p;
char *q;
if (p == q) {
  printf("This is only true if p and q point to the
  same address");
}
p = q;  /* The address contained in q is placed */
        /*    in p.  Does not change the memory */
        /*    locations p previously pointed to.*/
```

C String Library

Some of C's string functions

`strlen(char *s1)`
- Returns the number of characters in the string, not including the "null" character

`strncpy(char *s1, char *s2, int n)`
- Copies at most n characters of s2 on top of s1. The order of the parameters mimics the assignment operator

`strncmp (char *s1, char *s2, int n)`
- Compares up to n characters of s1 with s2
- Returns < 0, 0, > 0 if s1 < s2, s1 == s2 or s1 > s2 lexigraphically

`strncat(char *s1, char *s2, int n)`
- Appends at most n characters of s2 to s1

Insecure deprecated versions: strcpy, strcmp, strcat

String code example

```
#include <stdio.h>
#include <string.h>
int main() {
     char first[10] = "bobby ";
     char last[15] = "smith";
     char name[30];
     char you[5] = "bobo";

     strncpy( name, first, strlen(first)+1 );
     strncat( name, last, strlen(last)+1);
     printf("%d, %s\n", strlen(name), name );
     printf("%d \n",strncmp(you,first,3));

}
```

strncpy and null termination

strncpy does not guarantee null termination

- Intended to allow copying of characters into the middle of other strings
- Use `snprintf` to guarantee null termination

Example

```
#include <string.h>
main()  {
     char a[20]="The quick brown fox";
     char b[9]="01234567";
     strncpy(a,b,8);
     printf("%s\n",a);
}

% ./a.out
01234567k brown fox
```

Other string functions

Converting strings to numbers

```
#include <stdlib.h>
int strtol (char *ptr, char **endptr, int base);
```

Takes a character string and converts it to an integer.

- White space and + or - are OK.
- Starts at beginning of ptr and continues until something non-convertible is encountered.
- endptr (if not null, gives location of where parsing stopped due to error)

Some examples:

String	Value returned
"157"	157
"-1.6"	-1
"+50x"	50
"twelve"	0
"x506"	0

Other string functions

`double strtod (char * str, char **endptr);`

- String to floating point
- Handles digits 0-9.
- A decimal point.
- An exponent indicator (e or E).
- If no characters are convertible a 0 is returned.

Examples:

String	Value returned
"12"	12.000000
"-0.123"	-0.123000
"123E+3"	123000.000000
"123.1e-5"	0.001231

Examples

```
/* strtol Converts an ASCII string to its integer equivalent;
   for example, converts -23.5 to the value -23. */

   int my_value;

   char my_string[] = "-23.5";

   my_value = strtol(my_string, NULL, 10);

   printf("%d\n", my_value);

/* strtod Converts an ASCII string to its floating-point
   equivalent; for example, converts +1776.23 to the value
   1776.23. */

    double my_value;

    char my_string[] = "+1776.23";

    my_value = strtod(my_string, NULL);

    printf("%f\n", my_value);
```

Random number generation

Generate pseudo-random numbers

- `int rand(void);`
  - Gets next random number
- `void srand(unsigned int seed);`
  - Sets seed for PRNG
- man 3 rand

Random number generation

```
#include <stdio.h>
int main(int argc, char** argv) {
  int i,seed;

  seed = atoi(argv[1]);
  srand(seed);
  for (i=0; i < 10; i++)
         printf("%d : %d\n", i , rand());
}


OUTPUT:
% ./myrand 30
0 : 493850533
1 : 1867792571
2 : 1191308030
3 : 1240413721
4 : 2134708252
5 : 1278462954
6 : 1717909034
7 : 1758326472
8 : 1352639282
9 : 1081373099
%
```

Getopt

- Need to include `getopt.h` and `unistd.h` to use
- Used to parse command-line arguments.
- Typically called in a loop to retrieve arguments
- Switch statement used to handle options
  - colon indicates required argument
  - `optarg` is set to value of option argument
- Returns -1 when no more arguments present

```
int main(int argc, char** argv){
  int opt, x;
  /* looping over arguments */
  while(-1 != (opt = getopt(argc, argv, “x:”))){
    switch(opt) {
      case 'x':
        x = atoi(optarg);
        break;
      default:
        printf(“wrong argument\n");
        break;
      }
    }
}
```

Note about Library Functions

- These functions can return error codes
  - `malloc` could fail
  - a file couldn't be opened
  - a string may be incorrectly parsed
- Remember to check for the error cases and handle the errors accordingly
  - may have to terminate the program (eg `malloc` fails)
  - may be able to recover (user entered bad input)

Questions?