

**CS 201**

**The Standard C Library**

# Announcements

## **MECOP (Multiple Engineering Cooperative Program)**

- Cooperative industrial program. The five-year undergraduate program with two six-month paid internships.
- Interested students need to apply at the end of their sophomore year.
- Applications are accepted between March 1 and April 10.
- <http://mecop.ous.edu>

## **Intel Ultimate Engineering Experience**

- 6-week summer program.
- \$1000 stipend, \$2000 scholarship for fall studies
- Freshmen/sophomore engineering students
- Application due April 20
- [http://www.pdx.edu/mme/sites/www.pdx.edu.mme/files/Intel\\_Ultimate\\_Engineering%5B2%5D.pdf](http://www.pdx.edu/mme/sites/www.pdx.edu.mme/files/Intel_Ultimate_Engineering%5B2%5D.pdf)

# 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” on linuxlab shows which header file to include
- K&R Appendix B lists all functions

## Code linked in automatically

- At compile time (if statically linked gcc -static)
- At run time (if dynamically linked)
  - file <executable>
  - ldd <executable>

# The C Standard Library

## Examples

- I/O
  - `printf, scanf, fprintf, fscanf, putc, getc, puts, gets, fgetc, fgets, fputc, fputs, sprintf, sscanf`
  - `fopen, fclose, fflush, fread, fwrite, fseek`
  - `remove, rename, setbuf, tmpfile`
  - `feof, ferror, perror`
- Memory operations
  - `memcpy, memcmp, memset, malloc, free`
- Types
  - `isalpha, isdigit, islower, isupper, toupper, tolower`
- String operations
  - `strlen, strncpy, strncat, strncmp, strchr, strtok`
  - `atof, atoi, atol, strtod, strtol, strtoul`

# The C Standard Library

## Examples

- Math operations
  - `sin`, `cos`, `tan`, `exp`, `log`, `pow`, `sqrt`, `ceil`, `floor`, `div`, `abs`
- Utility functions
  - `rand`, `srand`, `exit`, `abort`, `system`, `getenv`
- Time
  - `clock`, `time`, `difftime`, `mktime`, `asctime`, `ctime`
- Variable arguments
  - `va_list`, `va_start`, `va_arg`, `va_end`
- Jumps
  - `setjmp`, `longjmp`
- Signals
  - `signal`, `raise`
- 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 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
- Format directives
  - Character (`%c`), String (`%s`), Integer (`%d`), Float (`%f`)
  - Formatting commands
    - » `%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

# 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

```
mashimaro <class/03> 2:25PM % ./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 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

```
mashimaro <class/03> 2:25PM % ./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`

# More on 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 80
int main(int argc, char* argv[])
{
    int f1,n;
    char buf[BUFSIZE];

    if ((f1 = open(argv[1], O_RDONLY, 0)) == -1)
        error("cp: can't open %s", argv[1]);

    while ((n = read(f1, buf, BUFSIZ)) > 0)
        if (write(1, buf, n) != n)
            error("cp: write error to stdout");
}
```

```
mashimaro <~> 2:01PM % cat opentest.txt
This is a test of CS 201
and the open(), read(),
and write() calls.
mashimaro <~> 2:01PM % ./opentest opentest.txt
This is a test of CS 201
and the open(), read(),
and write() calls.
mashimaro <~> 2:01PM % c
```

# 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:

```
mashimaro <class/03> 2:31PM % ./fops HELLO
mashimaro <class/03> 2:31PM % cat tmpfile.txt
HELLO
mashimaro <class/03> 2:32PM %
```

# Memory Allocation

## **malloc, calloc, free**

- C library calls that manage memory at run-time
- The actual system call is *sbrk* (low-level memory allocator)

# malloc

## Dynamically allocates memory

- Allocates memory in the heap
  - Lives between function invocations
- 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

## 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);
```

**Caveat: don't free the same memory block twice!**

# Zeroing Memory

**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  memory to  value for  length

# calloc

```
void *calloc(size_t nelem, size_t elsize);
```

- Allocates  $nelem * elsize$  bytes
- A lot like malloc except will always zero memory that is returned

**Essentially, a malloc with a paired memset call**

**It takes time to zero the memory, so frequent calls to calloc can be more costly than just malloc**

- A design consideration for your program

# Memory Operations

**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);
```

- in case the memory areas overlap

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

# The C String Library

**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
- This library is guaranteed to be there in any ANSI standard implementation of C.

# Strings

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

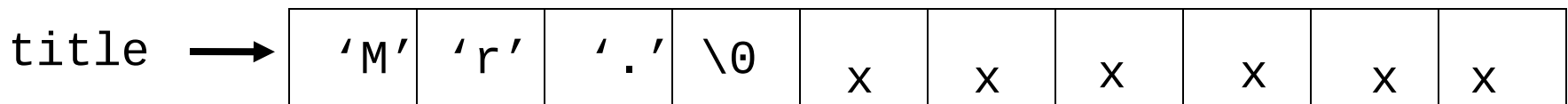
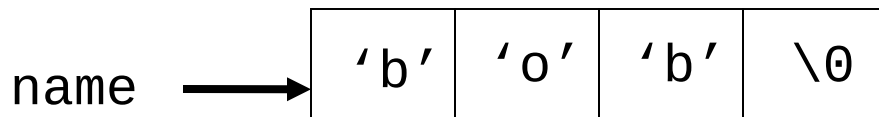
- **Character pointer p**

- 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

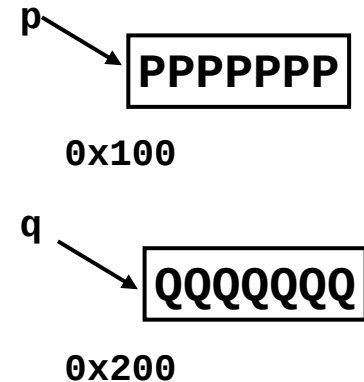
```
char* p="PPPPPPP";
```

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

```
p = q;
```

## What does this do?

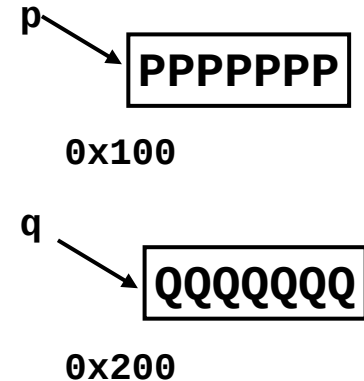
1. Copy QQQQQQ into 0x100?
2. Set p to 0x200



# Copying strings

## Consider

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

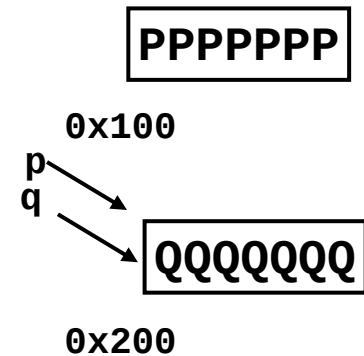


## What does this do?

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

## Copying strings

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



# C String Library

**C provides a library of string functions. Note that assignment( = ) and equality (==) operators don't do the job.**

```
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 *)`

- Returns the number of characters in the string, not including the “null” character

`strcpy(char *s1, char *s2)`

- Copies s2 on top of s1. The order of the parameters mimics the assignment operator

`strcmp(char *s1, char *s2)`

- Returns  $< 0$ ,  $0$ ,  $> 0$  if  $s1 < s2$ ,  $s1 == s2$  or  $s1 > s2$  lexicographically

`strcat(char *s1, char *s2)`

- Appends (concatenates) s2 to s1

# C String Library

**Secure versions of the C String library have an additional size parameter.**

`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` lexicographically

`strncat(char *s1, char *s2, int n)`

- Appends at most `n` characters of `s2` to `s1`

# 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));
}
```

# Other string functions

## Converting strings to numbers

Contained in `<stdlib.h>`

```
int atoi (char *ptr);
```

**Takes a character string and converts it to an integer.**

- White space and + or - are OK.
- Starts at beginning and continues until something non-convertible is encountered.

### Some examples:

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

# Other string functions

**long atol (char \*ptr) ;**

- Same as `atoi` except it returns a long.

**double atof (char \* str);**

- 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

```
/* atoi 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 = atoi(my_string);
```

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

```
/*atol Converts an ASCII string to its long integer  
equivalent; for example, converts +2000000000 to the value  
2000000000. */
```

```
long my_value;
```

```
char my_string[] = "+2000000000";
```

```
my_value = atol(my_string);
```

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

# Examples

```
/*atof Converts an ASCII string to its floating-point  
equivalent; for example, converts +1776.23 to the value  
1776.23. */  
  
#include <stdio.h>  
  
#include <stdlib.h>  
  
int main()  
{  
  
    double my_value;  
  
    char my_string[] = "+1776.23";  
  
    my_value = atof(my_string);  
  
    printf("%f\n", my_value);  
  
    return 0;  
  
}
```

**The output from the above code is 1776.230000.**

# Static character arrays

## Static arrays of characters are different

- Static character array `amsg`
  - Allocates characters at a fixed location that never changes (although you may change characters in array)
  - Can not reassign `amsg` to point elsewhere (i.e. `amsg = p`)

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

# 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:

```
mashimaro <class/03> 2:41PM % ./myrand 30
0 : 493850533
1 : 1867792571
2 : 1191308030
3 : 1240413721
4 : 2134708252
5 : 1278462954
6 : 1717909034
7 : 1758326472
8 : 1352639282
9 : 1081373099
mashimaro <class/03> 2:41PM %
```

**So how do you remember what arguments all these functions take, and what they return?**

**Use the “man” command.**

**Important: always include the header files shown by the “man” command**

# Makefiles

**Recipe for compiling and running your code**

**Call it makefile or Makefile (big or little M)**

- The “make” utility will use that by default
- You only have to specify the name if it’s called something else

**The first rule in the Makefile is used by default if you just say “make” with no arguments**

**The second line of each rule (the command) must start with a tab, not spaces!**

# A simple Makefile

```
sd: sd.c  
    cc -Wall -g -o sd sd.c
```

# A little more complex

```
all: sd test1 t1check test2
```

```
sd: sd.c  
    cc -g -o sd sd.c
```

```
test1: test1.c  
      cc -o test1 test1.c
```

```
test2: test2.c  
      cc -o test2 test2.c
```

```
t1check: t1check.c  
        cc -o t1check t1check.c
```

```
clean:  
      rm sd test1 t1check test2 t1sort
```

# A slightly more complex makefile

```
CC = gcc
CFLAGS = -Wall -O2
LIBS = -lm

OBJS = driver.o kernels.o fcyc.o clock.o

all: driver

driver: $(OBJS) config.h defs.h fcyc.h
    $(CC) $(CFLAGS) $(OBJS) $(LIBS) -o driver

driver.o: driver.c defs.h
kernels.o: kernels.c defs.h
fcyc.o: fcyc.c
clock.o: clock.c
```

# How to make a tar file

```
mkdir john
```

```
cp *.c *.h Makefile john
```

```
tar cvf john.tar john
```

How to extract the tar file:

```
tar xvf john.tar
```

# For Further Information

**W. Richard Stevens, Advanced Programming in the Unix Environment, Addison Wesley, 1993.**

**Brian W. Kernighan and Rob Pike, The Unix Programming Environment, Prentice-Hall, 1984**

**CS 201**

**GDB debugger**

# **gdb is Our Friend**

**Some might say “our best friend”**

**To compile a program for use with gdb, use the ‘-g’ compiler switch**

## **Better graphical interfaces**

- Most debuggers provide the same functionality
- Insight: <http://sourceware.org/insight/>
- DDD: <http://www.gnu.org/software/ddd/>
- KDbg: <http://www.kdbg.org/>

# Controlling program execution

## ***run***

- Starts the program

## ***step***

- Step program until it reaches a different source line.

## ***next***

- Step program, proceeding through subroutine calls.
- Single step to the next source line, not into the call.

## ***continue***

- Continue program execution after signal or breakpoint.

# Controlling program execution

## ***break, del***

- Set and delete breakpoints at particular lines of code

## ***watch, rwatch, awatch***

- Data breakpoints
- Stop when the value of an expression changes (*watch*), when expression is read (*rwatch*), or either (*awatch*)

# Printing out code and data

## *print*

- Print expression
- Basic
  - `print argc`
  - `print argv[0]`
- `print {type} addr`
  - `(gdb) p {char *} 0xbfffdce4`
- `print /x addr`
  - `'/x'` says to print in hex. See “help x” for more formats
  - Same as examine memory address command (`x`)
- `printf “format string” arg-list`
  - `(gdb) printf "%s\n", argv[0]`

## *list*

- Display source code (useful for setting breakpoints)

# Other Useful Commands

## ***where, backtrace***

- Produces a backtrace - the chain of function calls that brought the program to its current place.

## ***up, down***

- Change scope in stack

## ***info***

- Get information
- 'info' alone prints a list of info commands
- 'info br' : a table of all breakpoints and watchpoints
- 'info reg' : the machine registers

## ***quit***

- Exit the debugger

# Example Program

```
1  #include <stdio.h>
2  void sub(int i)
3  {
4      char* here[900];
5      sprintf((char *)here,"Function %s in %s", __FUNCTION__ , __FILE__);
6      printf("%s @ line %d\n", here, __LINE__);
7  }
8
9  void sub2(int j)
10 { printf("%d\n",j); }
11
12 int main(int argc, char** argv)
13 {
14     int x;
15     x = 30;
16     sub2(x);
17     x = 90;
18     sub2(x);
19     sub(3);
20     printf("%s %d\n",argv[0],argc);
21     return(0);
22 }
```

# Walkthrough example

```
% gcc -g -o gdb_ex gdb_ex.c
% gdb gdb_ex
(gdb) set args a b c d    set program arguments
(gdb) list 1,22          list source file
(gdb) break 14           break at source line at program start
(gdb) break sub          subroutine break
(gdb) break 6
(gdb) run                start program (breaks at line 14)
(gdb) p argv             hex address of argv (char**)
(gdb) p argv[0]          prints "gdb_ex"
(gdb) p argv[1]          prints "a"
(gdb) p strlen(argv[1])  prints 1
(gdb) p argc             prints 5
(gdb) p /x argc          prints 0x5
(gdb) p x                uninitialized variable
(gdb) n                  go to next line
(gdb) p x                x now 30
(gdb) p /x &x            print address of x
(gdb) x/w &x             print contents at address of x
```

# Walkthrough example

(gdb) <b>n</b>	go to next line (execute entire call)
(gdb) <b>s</b>	go to next source instr
(gdb) <b>s</b>	go to next source instr (follow call)
(gdb) <b>continue</b>	go until next breakpoint (breaks at line 6 in sub)
(gdb) <b>where</b>	list stack trace
(gdb) <b>p x</b>	x no longer scoped
(gdb) <b>up</b>	change scope
(gdb) <b>p x</b>	x in scope, prints 90
(gdb) <b>del 3</b>	delete breakpoint
(gdb) <b>continue</b>	finish
(gdb) <b>info br</b>	get breakpoints
(gdb) <b>del 1</b>	delete breakpoint
(gdb) <b>break main</b>	breakpoint main
(gdb) <b>run</b>	start program
(gdb) <b>watch x</b>	set a data write watchpoint
(gdb) <b>c</b>	watchpoint triggered

# Top-level Help

(gdb) help

List of classes of commands:

aliases -- Aliases of other commands

breakpoints -- Making program stop at certain points

data -- Examining data

files -- Specifying and examining files

internals -- Maintenance commands

obscure -- Obscure features

running -- Running the program

stack -- Examining the stack

status -- Status inquiries

support -- Support facilities

tracepoints -- Tracing of program execution without stopping the program

user-defined -- User-defined commands

Type "help" followed by a class name for a list of commands in that class.

Type "help" followed by command name for full documentation.

Command name abbreviations are allowed if unambiguous.

# Documentation

## What to do when you need to look something up?

- The gdb man page is intimidating

## Go to the web, do Google on gdb, look for online documentation

- [http://sources.redhat.com/gdb/current/onlinedocs/gdb\\_toc.html](http://sources.redhat.com/gdb/current/onlinedocs/gdb_toc.html)
- [http://sunsite.ualberta.ca/Documentation/Gnu/gdb-5.0/html\\_chapter/gdb\\_toc.html](http://sunsite.ualberta.ca/Documentation/Gnu/gdb-5.0/html_chapter/gdb_toc.html)
- There are others

# Extra slides

# A simple shell script

```
test1
sd t1file t1sort
t1check
echo
test2
sd t2file t1sort
t1check
echo
```

## A plain text file containing commands

- Known as a batch file in some O. S.'s

**Make sure the file has “executable” permission**

**Invoke it by name, just like a binary program**

## Which of these has a problem?

- `char c = 'z';`
- `char c = "zzz";`      */\* c is a char const not a pointer \*/*
- `char *p = "zzz";`
- `char *p = 'z';`      */\* p is a pointer not a char constant \*/*
- `char buf[256] = "zzz";`

## What about these? (Depends on compiler C89 or C99)

```
#define BUFSIZ 256
int bufsize = 256;
extern int BSIZE;
char buf[BUFSIZ];
char buf[bufsize];
char buf[BSIZE];
char buf [(BUFSIZ * 34) + 99];
char buf[BUFSIZ + bufsize];
```