The Standard C Library

The C Standard Library

Common functions we don’t need to write ourselves

- A portable interface to many system calls

Analogous to class libraries in Java or C++

Function prototypes declared in standard header files

- Must include the appropriate “.h” in source code

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

  - “man 3 printf” shows which header file to include

- K&R Appendix B describes the 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

The C Standard Library

I/O **stdio.h**

`printf, scanf, puts, gets, open, close, read, write,`
`fprintf, fscanf, fseek, …`

Memory and string operations **string.h**

`memcpy, memcmp, memset,`
`strlen, strncpy, strncat, strncmp,`
`strtod, strtol, strtoul, …`

Character Testing **ctype.h**

`isalpha, isdigit, isupper,`
`tolower, toupper, …`

Argument Processing **stdarg.h**

`va_list, va_start, va_arg, va_end, …`

The C Standard Library

Utility functions **stdlib.h**

`rand, srand, exit, system, getenv,`
`malloc, free, atoi, …`

Time **time.h**

`clock, time, gettimeofday, …`

Jumps **setjmp.h**

`setjmp, longjmp, …`

Processes **unistd.h**

`fork, execve, …`

Signals **signals.h**

`signal, raise, wait, waitpid, …`

Implementation-defined constants **limits.h, float.h**

`INT_MAX, INT_MIN, DBL_MAX, DBL_MIN, …`

Formatted Output

```
int printf(char *format, …)
```

Sends output to standard output

```
int fprintf(FILE *stream, char *format, …);
```

Sends output to a file

```
int sprintf(char *str, char *format, …)
```

Sends output to a string variable

Return Value: The number of characters printed

(not including trailing \0)

On Error: A negative value is returned.

Formatted Output

The format string is copied as-is to output.

Except the % character signals a formatting action.

Format directives specifications

- Character (%c), String (%s), Integer (%d), Float (%f)
- Fetches the next argument to get the value
- 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
  - %10.15s → Pad to 10, but truncate after 15, right justified

For more details: `man 3 printf`

Formatted Output

```c
#include <stdio.h>

int main() {
  char *p;
  float f;

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

  printf(":%10.15s:\n", p); // right justified, truncate to 15, pad to 10
  printf(":%15.10s:\n", p); // 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:

```
% test_printf_example
:This is a test:
:     This is a :
:909.22:
:      909.21533:
%
```

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 that the arguments are pointers!

Example: scanf

```
#include <stdio.h>

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

Why are pointers given to scanf?

Example: scanf

```
#include <stdio.h>

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

Why are pointers given to scanf?

Input Error Checking

```
#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:

```
% test_scanf_example
Enter the first value: 20
Enter the second value: 30
20 + 30 = 50
%
```

Line-Based I/O

`int puts(char *line)`

Outputs string pointed to by `line` followed by newline character to `stdout`

`char *gets(char *s)`

Reads the next input line from `stdin` into buffer pointed to by `s`

Null terminates

`char *fgets(char *s, int size, FILE * stream)`

"size" is the size of the buffer.

Stops reading before buffer overrun.

Will store the \n, if it was read.

`int getchar()`

Reads a character from stdin

Returns it as an int (0..255)

Returns EOF (i.e., -1) if "end-of-file" or "error".

General 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` (i.e., 0)

Standard output (display)

`stdout` (i.e., 1)

Standard error (display)

`stderr` (i.e., 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

```c
#include <stdio.h>
#include <fcntl.h>
#define BUFSIZE 16

int main(int argc, char* argv[]) {
    int fd,n;
    char buf[BUFSIZE];

    if ((fd = open(argv[1], O_RDONLY)) == -1)
        perror("cp: can't open file");
    do {
        if ((n=read(fd, 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 CS 201
and the open(), read(),
and write() calls.
% ./opentest opentest.txt
This is a test of CS 201
and the open(), read(),
and write() calls.
% ./opentest asdfasdf
cp: can't open file: No such file or directory
%
```

I/O Redirection in the Shell

File Redirection

```
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 programs to each other via pipes

```
ls –l | egrep tar
```

Standard output of “`ls`” sent to standard input of “`egrep`”

I/O via “File” Interface

Similar interface

`fscanf, fread, fgets, fprintf, fwrite, fputs`

Must supply `FILE*` argument for each call

Note: `FILE*` ≠ file descriptor

```
FILE *fopen(char *name, char *mode);
```

Opens a file if we have access permission

Returns a “FILE pointer” which you use in `fread`, `fwrite`, …

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

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

```
int fclose(fp);
```

Flush any pending output and clean up.

I/O via “File” Interface

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

main (int argc, char** argv) {
   char *p = argv[1];
   FILE *fp;

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

OUTPUT:

```
% test_file_ops HELLO
% cat tmpfile.txt
HELLO
%
```

Memory allocation and management

```
(void *) malloc (int numberOfBytes)
```

Dynamically allocates memory from the heap

- Memory persists between function invocations (unlike local variables)

Returns a pointer to a block of at least numberOfBytes 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 “n” elements

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

Memory allocation and management

```
(void *) malloc (int numberOfBytes)
```

Be careful to allocate enough memory!

Overrun on the space is undefined!!!

Common error:

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

NOTE: `strlen` *doesn’t account for the NULL terminator!*

Fix:

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

Memory allocation and management

```
void free(void * p)
```

Deallocates memory in heap.

Pass in a pointer that was returned by `malloc`.

Example

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

Example

```
        struct table* tp =

               (struct table*) malloc(sizeof(struct table));

        free(tp);
```

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

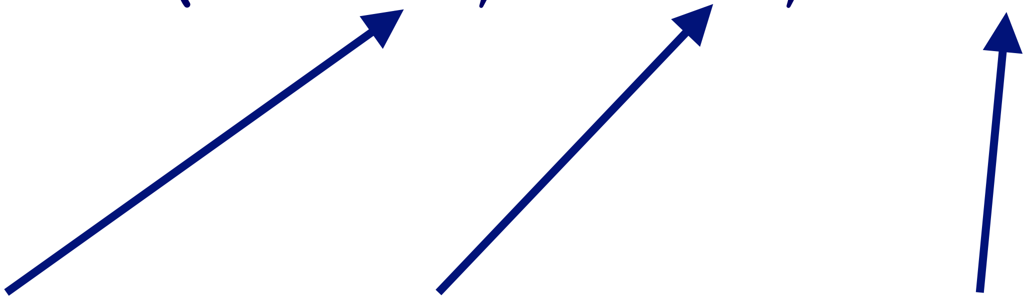
Memory allocation and management

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 ch, int n);
```



Set this memory to this value for this number of bytes

Memory allocation and management

How to move a block of bytes efficiently?

```
void *memmove(void *dest, void *src, int n);
```

How to allocate zero-filled chunk of memory?

```
void *calloc(int numberThings, int sizeOfThings);
```

Note:

These slides use "`int`"
However, "`size_t`" is better.
Makes code more portable.
"`size_t`" → unsigned integer.

Strings

String functions are provided in the string library.

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

Includes functions such as:

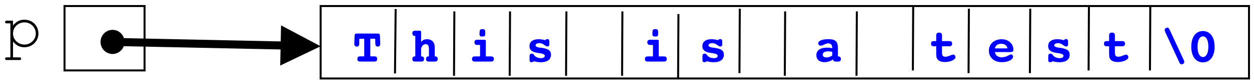
- Compute length of string
- Copy strings
- Concatenate strings
- …

Strings

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

Set p to the address of a character array

```
char *p = "This is a test";
```

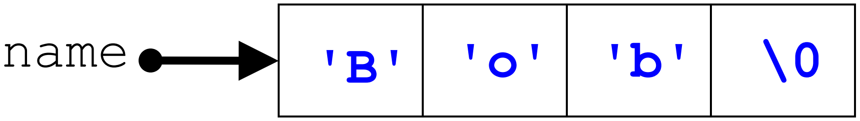


p

T	h	i	s		i	s		a		t	e	s	t	\0

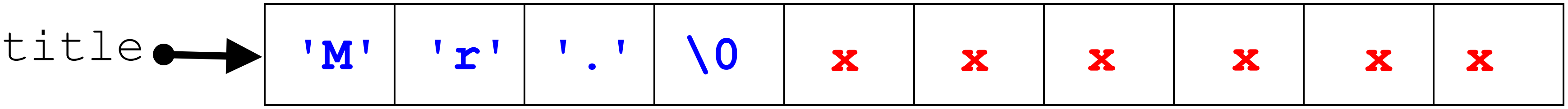
NOTE: p can be reassigned to a different address

```
char name[4] = "Bob";
char title[10] = "Mr.";
```



name

'B'	'o'	'b'	\0



title

'M'	'r'	'.'	\0	x	x	x	x	x	x

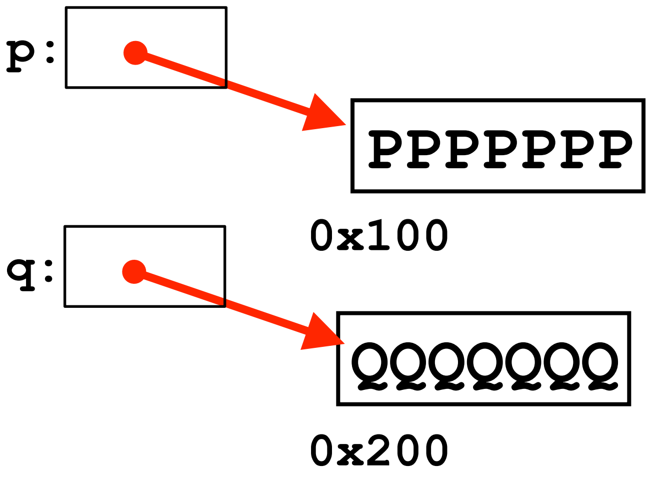
Copying strings

Consider

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

What does this do?

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



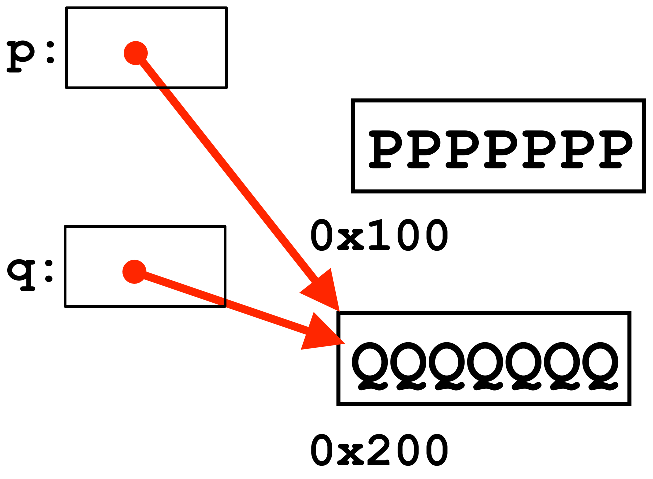
Copying strings

Consider

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

What does this do?

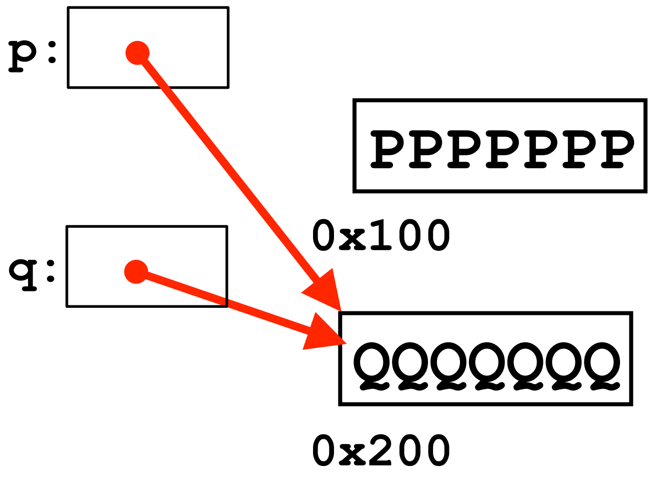
1. ~~Copy QQQQQQ into 0x100?~~
2. Set p to 0x200



Copying strings

Consider

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



What does this do?

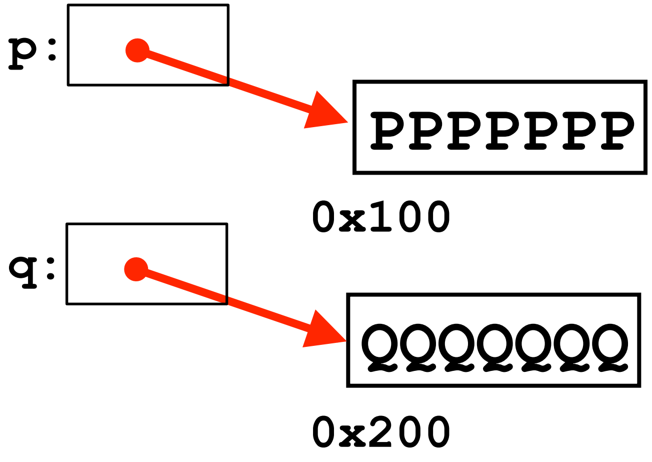
1. ~~Copy QQQQQQ into 0x100?~~
2. Set p to 0x200

To copy the strings?

- Manually copy characters

```
p[2] = q[2];
```

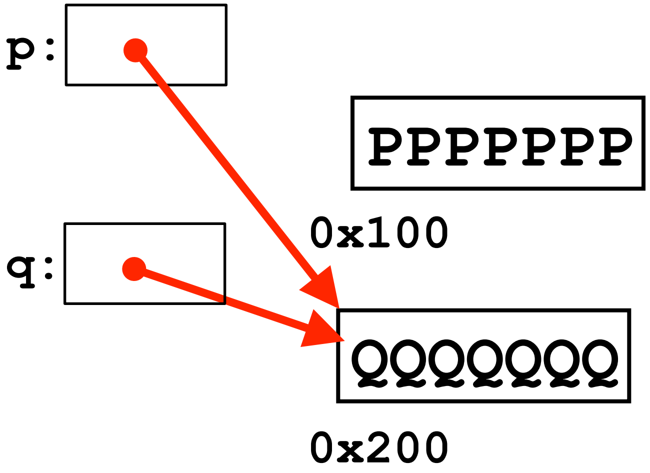
- Use `strncpy` to copy characters



Copying strings

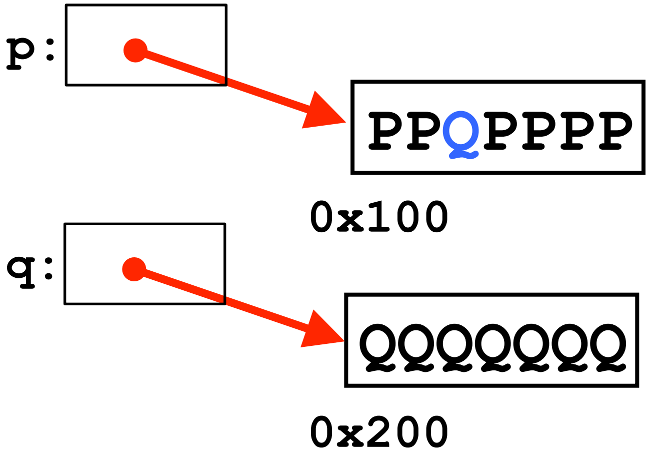
Consider

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



What does this do?

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



To copy the strings?

- Manually copy characters

```
p[2] = q[2];
```

- 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 lexigraphically.

Returns < 0, 0, > 0 if s1 < s2, s1 == s2 or s1 > s2

```
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

```c
#include <stdio.h>
#include <string.h>
int main() {
    char first[10] = "Harry ";
    char last[15] = "Porter";
    char name[30];
    char you[] = "Harold";

    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));

}
```

OUTPUT:

```
12, "Harry Porter"
0
```

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);
}
```

OUTPUT:

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

Other string functions

Converting strings to numbers

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

Takes a character string and converts it to a long (long) 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)

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 -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 the seed for Pseudo-Random Number Generator

For Unix/Linux documentation:

```
$ man 3 rand
```

On Internet:

www.man7.org

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
%
```

Makefiles

The make utility: Compile things as necessary:

```
make
```

The makefile: Recipe for compiling 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
   gcc -Wall -g sd.c -o sd
```

```
% make
gcc -Wall -g sd.c -o sd
%
```

A little more complex

```
all: sd test1 t1check test2


sd: sd.c
    gcc -Wall -g sd.c -o sd

test1: test1.c
    gcc -Wall -g test1.c -o test1

test2:  test2.c
    gcc -Wall -g test2.c -o test2

t1check: t1check.c
    gcc -Wall -g t1check.c -o t1check

clean:
    rm sd test1 t1check test2
```

A 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 fcyc.h

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
```

GDB debugger

The Unix/Linux Debugger: gdb

When all else fails…

- Stop the program
- Look at (or modify) registers
- Look at (or modify) memory
- Single-step the program
- Set a "breakpoint"

To compile a program for use with gdb

… use the '-g' compiler switch

Controlling program execution

`run`

Start 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.

Execute the whole routine at once; stop upon RETURN.

`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 expr

```
(gdb) print x
(gdb) print argv[0]
```

print {type} addr

```
(gdb) p {char *} 0xbfffdce4
(gdb) 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

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`

`(gdb) info`	prints a list of info commands
`(gdb) info br`	Print a table of all breakpoints and watchpoints
`(gdb) info r`	The machine registers

`quit`

Exit gdb

Example Program

```c
#include <stdio.h>

void sub(int i) {
    char here [900];
    sprintf ((char *) here, "Function %s in %s", __FUNCTION__ , __FILE__);
    printf ("%s @ line %d\n", here, __LINE__);
}

void sub2(int j) {
    printf ("%d\n″, j);
}

int main(int argc, char** argv)
{
    int x;
    x = 30;
    sub2 (x);
    x = 90;
    sub2 (x);
    sub (3);
    printf ("%s %d\n", argv[0], argc);
    return (0);
}
```

http://cs.pdx.edu/~harry/cs201/gdb_example.c

Walkthrough example

```
% gcc -g gdb_example.c -o b_example
% gdb gdb_example
(gdb) set args a b c d        set program arguments
(gdb) list 1,99               list source file through line 99
(gdb) break main              set breakpoint at beginning of “main” function
(gdb) break sub               set another breakpoint
(gdb) break 6                 set break at source line
(gdb) run                     start program (breaks at line 16)
(gdb) disass main             show assembly code for “main” function
(gdb) info r                  display register contents
(gdb) p argv                  hex address of argv (char**)
(gdb) p argv[0]               prints “gdb_example”
(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, prints some #
(gdb) n                       execute to the next line
(gdb) p x                     x is now 30
(gdb) p/x &x                  print address of x
(gdb) x/w &x                  print contents at address of x
```

Walkthrough example

`(gdb) n`	go to next line (execute entire call)
`(gdb) s`	go to next source instr
`(gdb) s`	go to next source instr (follow call)
`(gdb) continue`	go until next breakpoint (breaks at line 6 in sub)
`(gdb) where`	list stack trace
`(gdb) p x`	x no longer scoped
`(gdb) up`	change scope
`(gdb) p x`	x in scope, prints 90
`(gdb) del 3`	delete breakpoint
`(gdb) continue`	finish

`(gdb) info br`	get breakpoints
`(gdb) del 1`	delete breakpoint
`(gdb) break main`	breakpoint main
`(gdb) run`	start program
`(gdb) watch x`	set a data write watchpoint
`(gdb) c`	watchpoint triggered
`(gdb) quit`	quit

Different gdb interfaces

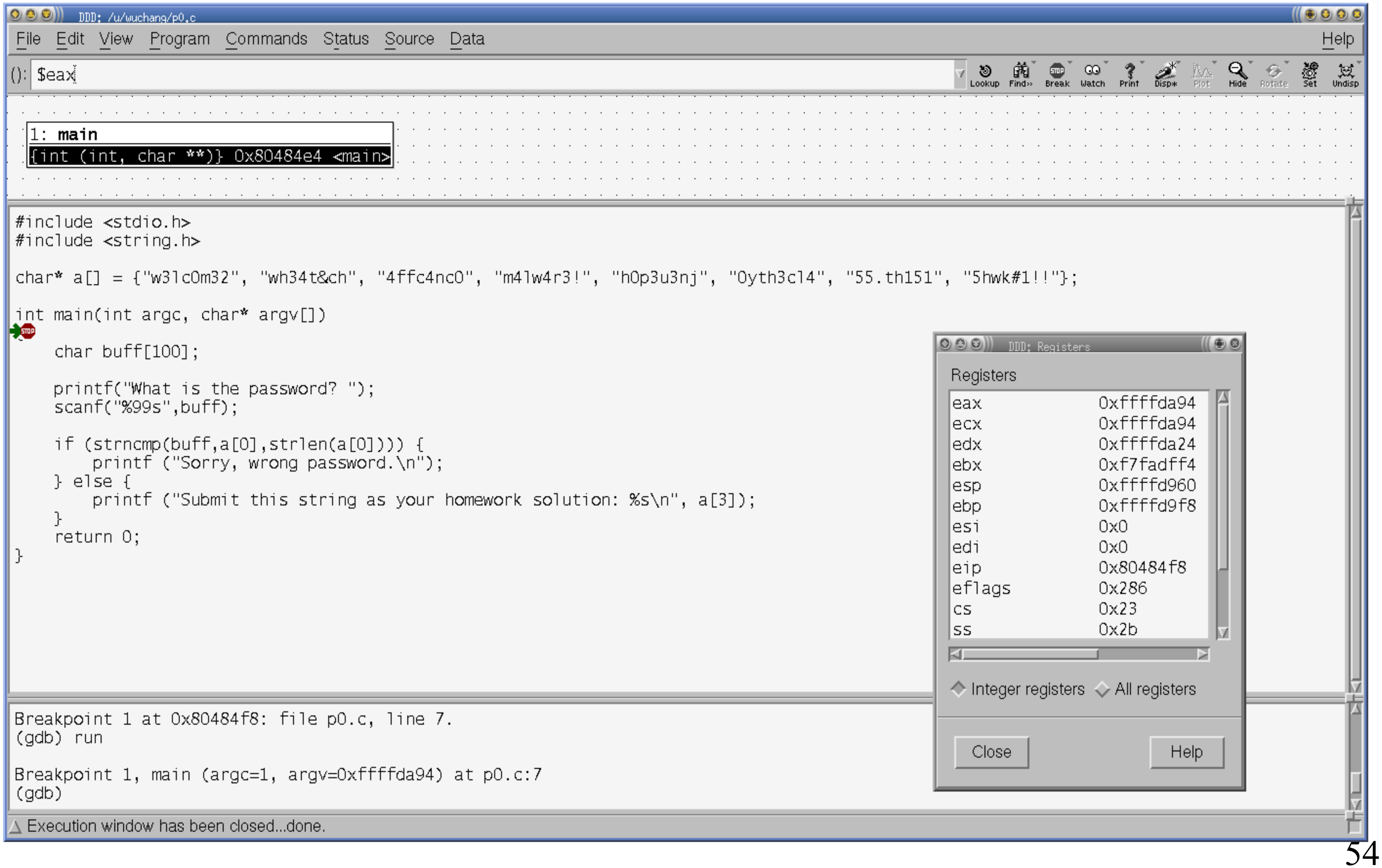
Better graphical interfaces

Most debuggers provide the same functionality

- `gdb –tui`

  Sort of graphical (like “vi”)
- Insight: http://sourceware.org/insight
- DDD: http://www.gnu.org/software/ddd
- TDB: http://pdqi.com/browsex/TDB.html
- KDbg: http://www.kdbg.org

DDD



DDD: /u/wuchang/p0.c
File Edit View Program Commands Status Source Data Help
():
Lookup Find» Break Watch Print Disp* Plot Hide Rotate Set Undisp
DDD: Registers
Close
Help

```
$eax
```

```
1: main
{int (int, char **)} 0x80484e4 <main>
```

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

char* a[] = {"w3lc0m32", "wh34t&ch", "4ffc4nc0", "m4lw4r3!", "h0p3u3nj", "0yth3cl4", "55.th151", "5hwk#1!!"};

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

    char buff[100];

    printf("What is the password? ");
    scanf("%99s",buff);

    if (strncmp(buff,a[0],strlen(a[0]))) {
        printf ("Sorry, wrong password.\n");
    } else {
        printf ("Submit this string as your homework solution: %s\n", a[3]);
    }
    return 0;
}
```

Registers

```
eax            0xffffda94
ecx            0xffffda94
edx            0xffffda24
ebx            0xf7fadff4
esp            0xffffd960
ebp            0xffffd9f8
esi            0x0
edi            0x0
eip            0x80484f8
eflags         0x286
cs             0x23
ss             0x2b
```

◇ Integer registers ◇ All registers

```
Breakpoint 1 at 0x80484f8: file p0.c, line 7.
(gdb) run

Breakpoint 1, main (argc=1, argv=0xffffda94) at p0.c:7
(gdb)
```

△ Execution window has been closed...done.