

C Primer

CS351 : Saelee

“

I have stopped reading
Stephen King novels.
Now I just read C code
instead.

- Richard A. O'Keefe

Game plan

1. Overview
2. Operators, expressions, primitive types
3. Control constructs and functions
4. Compilation
5. Pointers, arrays, and aggregate types
6. Essential libraries

not a language course

K&R

UNIX man pages

comp.lang.C FAQ

NAME

strlen - find length of string

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

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

```
size_t
```

```
strlen(const char *s);
```

DESCRIPTION

The `strlen()` function computes the length of the string `s`.

RETURN VALUES

The `strlen()` function returns the number of characters that precede the terminating NUL character.

SEE ALSO

string(3)

> man strlen

imperative
strongly typed
weakly type checked
procedural
low level

C	Java
Procedural	Object-oriented
Source-level portability	Compiled-code portability
Manual memory management	Garbage collected
Pointers reference addresses	Opaque memory references
Manual error code checking	Exception handling
Manual namespace partitioning	Namespaces with packages
Lower level libraries	High level class libraries

C: “Make it efficient and simple, and let the programmer do whatever she wants”

Java: “Make it portable, provide a huge class library, and try to protect the programmer from doing stupid things.”

“

A language that **doesn't
have everything**
is actually **easier to
program in** than some
that do.

- Dennis M Ritchie

**American National
Standards Institute**

Limited standard library

Primitive Types

- `char`: single byte, “character”
- `short`: “short” integer, at least 16 bits
- `int`: integer, at least 16 bits
- `long`: “long” integer, at least 32 bits
- `float`: single precision floating point number
- `double`: double precision floating point number

Basic Operators

- Arithmetic: +, -, *, /, %, ++, --
- Relational: <, >, <=, >=, ==, !=
- Logical: &&, ||, &, |, !
- Assignment: =, +=, *=, ...

True/False

- 0 → “false”
- **Everything else** → “true”

!(0) → 1

!(1234) → 0

!!(-1020) → 1

Control Structures

- `if-else`
- `switch-case`
- `while, for, do-while`
- `continue, break`
- “Infinately abusable” `goto`

Variables

declare before use

scope:

global

local

static

Functions = top level module

```
public class Demo {  
    public static void main (String[] args) {  
        System.out.println("Hello world!");  
    }  
}
```

VS.

```
int main (int argc, char *argv[]) {  
    printf("Hello world!");  
}
```

declaration vs. definition
many-to-one

header files (.h)

vs.

source files (.c)

```
linux> gcc -o hello hello.c  
linux> ./hello  
Hello world!  
linux>
```

Compilation

Preprocessing



Compilation



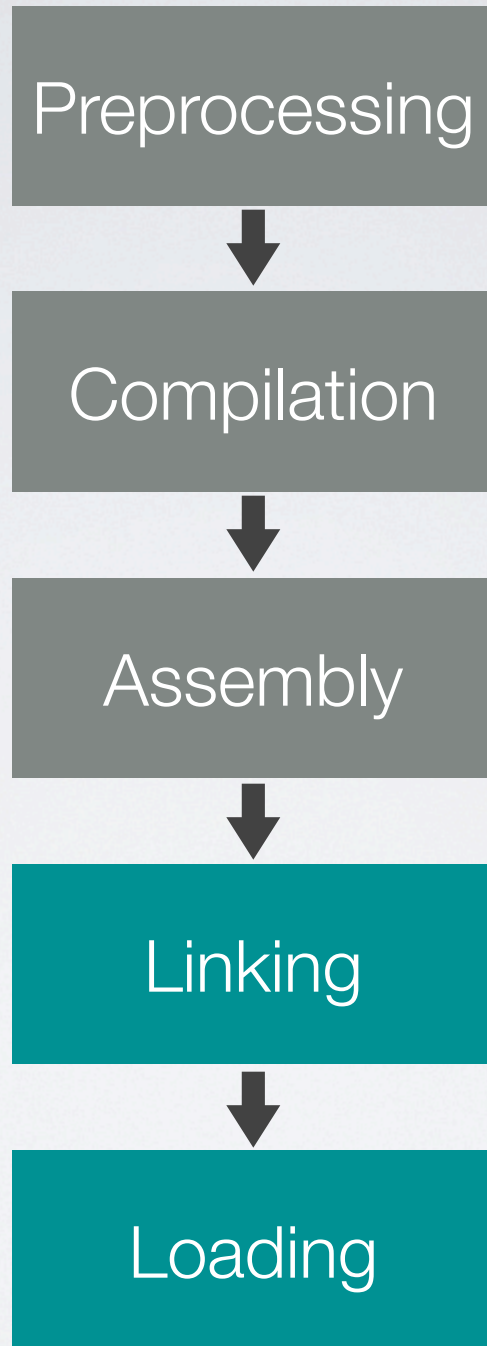
Assembly



Linking



Loading



preprocessor directives

text substitution

macros

conditional compilation

```
#define msg "Hello world!\n"

int main () {
    printf(msg);
    return 0;
}
```

```
linux> gcc -E hello.c
```

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

```
#define PLUS1(x) (x+1)

int main () {
    int y;
    y = y * PLUS1(y);
    return 0;
}
```

```
linux> gcc -E plus1.c
```

```
int main () {
    int y;
    y = y * (y+1);
    return 0;
}
```

```
#define SAYHI

int main () {
#ifdef SAYHI
    printf("Hi!");
#else
    printf("Bye!");
#endif
    return 0;
}
```

```
linux> gcc -E hello.c
```

```
int main () {
    printf("Hi!");
    return 0;
}
```

Linkage + Scoping

functions = external

variables = internal

sum.c

```
int sumWithI (int x, int y) {  
    return x + y + I;  
}
```

main.c

```
#include <stdio.h>  
  
int I = 10;  
  
int main () {  
    printf("%d\n", sumWithI(1, 2));  
    return 0;  
}
```

```
linux> gcc -Wall sum.c main.c  
sum.c: In function `sumWithI':  
sum.c:3: error: `I' undeclared (first use in this function)  
main.c: In function `main':  
main.c:10: warning: implicit declaration of function `sumWithI'
```

sum.c

```
int sumWithI (int x, int y) {  
    int I;  
    return x + y + I;  
}
```

main.c

```
#include <stdio.h>  
  
int sumWithI (int, int);  
  
int I = 10;  
  
int main () {  
    printf("%d\n", sumWithI(1, 2));  
    return 0;  
}
```

```
linux> gcc -Wall sum.c main.c  
linux> ./a.out  
-1073743741  
linux>
```

sum.c

```
int sumWithI (int x, int y) {  
    extern int I;  
    return x + y + I;  
}
```

main.c

```
#include <stdio.h>  
  
int sumWithI (int, int);  
  
int I = 10;  
  
int main () {  
    printf("%d\n", sumWithI(1, 2));  
    return 0;  
}
```

```
linux> gcc -Wall sum.c main.c  
linux> ./a.out  
13  
linux>
```

Pointers
variables
that store addresses

declaration: `type *var`

Operators:

& : address-of

* : dereference

```
int i = 5;    /* i is an int containing 5 */
int *p;      /* p is a pointer to an int */
p = &i;      /* p holds the address of i */
int j = *p;  /* j gets what p points to */
```

```
int main () {  
    int i, j, *p, *q;  
    i = 10;  
    p = &j;  
    q = p;  
    *q = i;  
    *p = (*q) * 2;  
    printf("i=%d, j=%d, *p=%d, *q=%d\n",  
           i, j, *p, *q);  
}
```

i=10, j=20, *p=20, *q=20

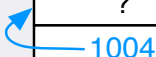
Step by step

```
int i, j, *p, *q;  
i = 10;
```

Address	Data	
1000	10	(i)
1004	?	(j)
1008	?	(p)
1012	?	(q)

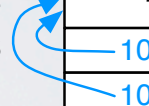
```
p = &j;
```

Address	Data	
1000	10	(i)
1004	?	(j, *p)
1008	?	(p)
1012	?	(q)




```
q = p;
```

Address	Data	
1000	10	(i)
1004	?	(j, *p, *q)
1008	?	(p)
1012	?	(q)




```
*q = i;
```

Address	Data	
1000	10	(i)
1004	10	(j, *p, *q)
1008	?	(p)
1012	?	(q)



```
*p = (*q) * 2;
```

Address	Data	
1000	10	(i)
1004	20	(j, *p, *q)
1008	?	(p)
1012	?	(q)



pass-by-value

vs.

pass-by-reference

vs.

passing references

```
int main () {  
    int a = 5, b = 10;  
    swap(a, b);  
    /* want a == 10, b == 5 */  
}
```

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

```
int main () {  
    int a = 5, b = 10;  
    swap(&a, &b);  
    /* want a == 10, b == 5 */  
}
```

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

swapping pointers?

```
int *pa, *pb;  
...  
swap(pa, pb);
```

```
int main () {  
    int *pa, *pb;  
    /* initialize pointers... */  
    swapp(&pa, &pb);  
}
```

```
void swapp (int **p, int **q) {  
    int *tmp = *p;  
    *p = *q;  
    *q = tmp;  
}
```

uninitialized pointers

= **garbage**

if lucky → crash

if unlucky → ?

the **null** pointer
never returned by &
safe to initialize and check

0

NULL

(int*)0

```
/* Arrays */
```

```
int arr1[5];
```

```
int arr2[] = {1, 2, 3, 4, 5};
```

```
arr1[pos] = val;
```

garbage

no metadata

no bounds checking

pointers ♥ arrays
array name = address

```
int *pa;  
int arr[5];
```

```
pa = &arr[0]      ⇔      pa = arr
```

```
arr[i]            ⇔      pa[i]
```

```
arr[i]            ⇔      *(pa+i)
```

Arrays \neq Pointers
static vs. dynamic binding

pointer arithmetic
pointer magic

```
int arr[] = {10, 20, 30, 40, 50},  
    *p     = arr,  
    *q     = &arr[4];
```

```
printf("%d\n", p==0);           // => 0  
printf("%d\n", *++p);          // => 20  
p += 2;  
printf("%d\n", *p);            // => 40  
printf("%d\n", *(q-3));        // => 20  
printf("%d\n", p < q);         // => 1  
printf("%d\n", q - p);         // => 1
```

string = `'\0'` terminated array

```
char str[]      = "hello!";  
char *p        = "hi";  
char tarr[][5] = {"max", "of", "four"};  
char *sarr[]   = {"variable", "length", "strings"};
```

```
int main (int argc, char *argv[]) {  
    for (; argc>0; argc--)  
        printf("%s%s", *argv++, argc==1?"":", ");  
}
```

```
linux> ./a.out testing one two three  
./a.out, testing, one, two, three
```

strcpy

```
void strcpy (char s[], char t[]) {  
    int i = 0;  
    while ((s[i] = t[i]) != '\0')  
        i++;  
}
```

```
void strcpy (char *s, char *t) {  
    while ((*s = *t) != '\0') {  
        s++;  
        t++;  
    }  
}
```

```
void strcpy (char *s, char *t) {  
    while ((*s++ = *t++) != '\0');  
}
```

```
void strcpy (char *s, char *t) {  
    while (*s++ = *t++);  
}
```

<string.h>

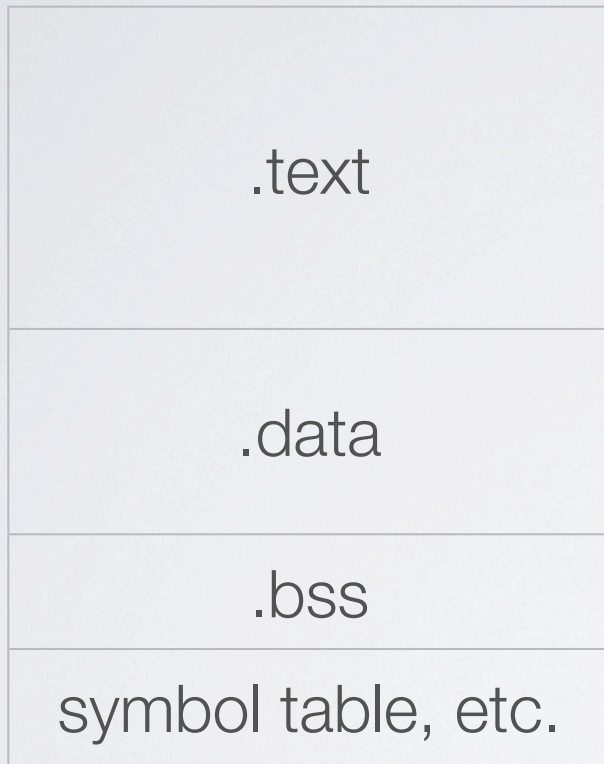
strcpy, strcat

strcmp, strlen, strchr

memcpy, memmove

lifetime
scope

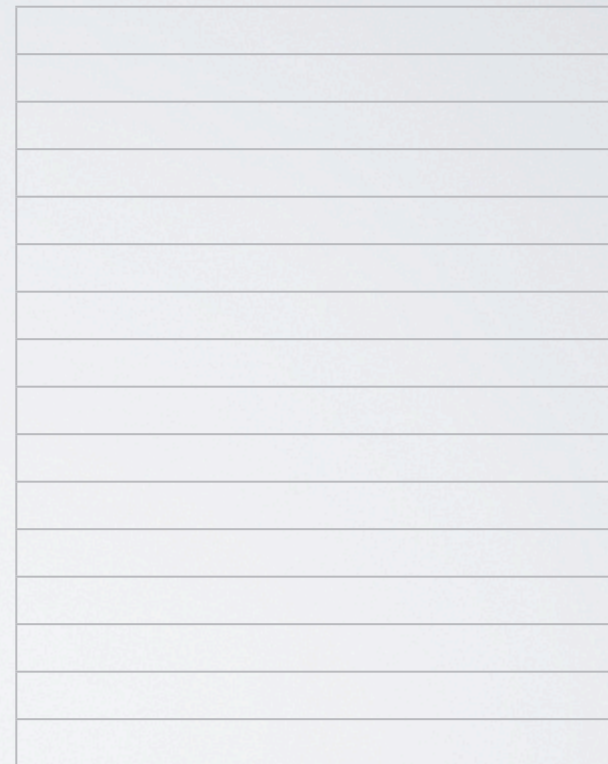
ELF (file)



loader



RAM



code, **global** & **static** data

```
void bar() { foo (); }
```



local variables
automatically reclaimed
LIFO alloc/dealloc

dynamic scope

dynamic memory allocation

<stdlib.h>

malloc

free

realloc

explicit memory management

freeing is critical

memory leaks

long running processes

generic pointers
(void *)
default for malloc

```
/* single int */
int *ip = (int *) malloc( sizeof(int) );

/* N char buffer */
char *buf = (char *) malloc(N * sizeof(char));

/* ROWS x COLS int matrix */
int **mtx = (int **) malloc(ROWS * sizeof(int *));
for (i=0; i<ROWS; i++)
    mtx[i] = (int *) malloc(COLS * sizeof(int));

free(ip);
free(buf);
for (i=0; i<ROWS; i++)
    free(mtx[i]);
free(mtx);
```

“first class” functions

```
> (define square
    (lambda (x) (* x x)))

> (square 5)
25

> (map square
    '(1 2 3 4))
(1 4 9 16)

> (map (lambda (x) (* x x x))
    '(1 2 3 4))
(1 8 27 64)
```

... in Scheme (*lambdas*)

map is a
“higher order” function

can't quite in C

function **pointers**

```
int *p;           /* pointer to int */
int arr[10];     /* array of 10 ints */
int *arr[10];    /* array of 10 pointers to int */
int *(arr[10]); /* array of 10 pointers to int */
int (*arr)[10]; /* pointer to array of 10 ints */
int *bar ();    /* function that returns a pointer
                to an int */

int (*fp) ();   /* pointer to a function that
                returns an int */

int ((*f())[10])() /* function returning pointer to
                  an array of 10 pointers to
                  functions returning int */
```

```
int times_two (int x) {
    return x*2;
}

void map (int (*f)(int), int *arr, int n) {
    int i;
    for (i=0; i<n; i++) {
        arr[i] = (*f)(arr[i]);
    }
}

main () {
    int i,
    arr[] = {1, 2, 3, 4, 5};
    map(times_two, arr, 5);
    for (i=0; i<5; i++) {
        printf("%d ", arr[i]); // => 2, 4, 6, 8, 10
    }
}
```

```
int even_p (int n) {  
    return n%2 == 0;  
}
```

```
int sum_if (int *vec, int n, int (*p)(int) ) {  
    int i, sum = 0;  
    for (i=0; i<n; i++) {  
        if ((*p)(vec[i]))  
            sum += vec[i];  
    }  
    return sum;  
}
```

```
main() {  
    int arr[] = {1, 2, 3, 4, 5};  
    printf("%d\n", sum_if(arr, 5, even_p)); // => 6  
}
```

aggregate data

Java/C++ object

C structs

```
/* type definition */  
struct point {  
    int x;  
    int y;  
}; /* don't forget the end ';' */
```

```
/* struct declarations */  
struct point p1, p2, *pp;
```

dot operator: ' . '

component access

arrow operator: '→'
component via pointer

```
/* type definition */  
struct point {  
    int x;  
    int y;  
}; /* don't forget the end ';' */
```

```
/* struct declarations */  
struct point p1, p2, *pp;
```

```
/* struct member access */  
p1.x = 0; p1.y = 0;  
pp = &p1;  
(*pp).x = 5;  
pp->y = 6;
```

```
/* dynamic alloc */
struct point *ptarr = (struct point *) malloc(
    N * sizeof(struct point) );

/* initialization */
for (i=0; i<N; i++) {
    ptarr[i].x = ptarr[i].y = 0;
}

/* .. do stuff .. */

free(ptarr);
```

```
struct point p1  
struct point *p2  
...wordy  
typedef feature
```

```
typedef struct pt {  
    int x;  
    int y;  
} Point, *PointPtr;
```

```
Point p;           // == struct pt p  
PointPtr pp;      // == struct pt *pp
```

pointer magic

```
PointPtr p = malloc(N * sizeof(Point));  
for (i=0; i<N; i++) {  
    p->x = p->y = 0;  
    p++;  
}
```

← address incremented by sizeof(Point)

Coding style

```
int **mtx = (int **) malloc(ROWS * sizeof(int));
if (mtx == NULL) {
    fprintf(stderr, "out of memory!");
    exit(1);
}
for (i=0; i<ROWS; i++) {
    mtx[i] = (int *) malloc(COLS * sizeof(int));
    if (mtx[i] == NULL) {
        fprintf(stderr, "out of memory!");
        exit(1);
    }
}
```

check return values

```
void *Malloc (size_t size) {  
    if ((p = malloc(size)) == NULL)  
        unix_error("malloc error");  
    return p;  
}
```

wrapper functions are handy

K&R

UNIX man pages
comp.lang.C FAQ