Characters, Strings, Larger C Program organization

Shuai Mu

based on Tiger Wang's and Jinyang Li's slides

Characters

How to represent text characters?

- How to associate bit patterns to integers?
  - base 2
  - 2’s complement
- How to associate bit patterns to floats?
  - IEEE floating point representation (based on normalized scientific notation)
- How to associate bit patterns to characters?
  - by convention
  - ASCII, UTF

ASCII: American Standard Code for Information Exchange

- Developed in 60s, based on the English alphabet
- use one byte (with MSB=0) to represent each character
- How many unique characters can be represented?

128

ASCII TABLE

Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char
0	0	*[NULL]*	32	20	*[SPACE]*	64	40	@	96	60	`
1	1	*[START OF HEADING]*	33	21	!	65	41	A	97	61	a
2	2	*[START OF TEXT]*	34	22	"	66	42	B	98	62	b
3	3	*[END OF TEXT]*	35	23	#	67	43	C	99	63	c
4	4	*[END OF TRANSMISSION]*	36	24	$	68	44	D	100	64	d
5	5	*[ENQUIRY]*	37	25	%	69	45	E	101	65	e
6	6	*[ACKNOWLEDGE]*	38	26	&	70	46	F	102	66	f
7	7	*[BELL]*	39	27	'	71	47	G	103	67	g
8	8	*[BACKSPACE]*	40	28	(	72	48	H	104	68	h
9	9	*[HORIZONTAL TAB]*	41	29	)	73	49	I	105	69	i
10	A	*[LINE FEED]*	42	2A	*	74	4A	J	106	6A	j
11	B	*[VERTICAL TAB]*	43	2B	+	75	4B	K	107	6B	k
12	C	*[FORM FEED]*	44	2C	,	76	4C	L	108	6C	l
13	D	*[CARRIAGE RETURN]*	45	2D	-	77	4D	M	109	6D	m
14	E	*[SHIFT OUT]*	46	2E	.	78	4E	N	110	6E	n
15	F	*[SHIFT IN]*	47	2F	/	79	4F	O	111	6F	o
16	10	*[DATA LINK ESCAPE]*	48	30	0	80	50	P	112	70	p
17	11	*[DEVICE CONTROL 1]*	49	31	1	81	51	Q	113	71	q
18	12	*[DEVICE CONTROL 2]*	50	32	2	82	52	R	114	72	r
19	13	*[DEVICE CONTROL 3]*	51	33	3	83	53	S	115	73	s
20	14	*[DEVICE CONTROL 4]*	52	34	4	84	54	T	116	74	t
21	15	*[NEGATIVE ACKNOWLEDGE]*	53	35	5	85	55	U	117	75	u
22	16	*[SYNCHRONOUS IDLE]*	54	36	6	86	56	V	118	76	v
23	17	*[ENG OF TRANS. BLOCK]*	55	37	7	87	57	W	119	77	w
24	18	*[CANCEL]*	56	38	8	88	58	X	120	78	x
25	19	*[END OF MEDIUM]*	57	39	9	89	59	Y	121	79	y
26	1A	*[SUBSTITUTE]*	58	3A	:	90	5A	Z	122	7A	z
27	1B	*[ESCAPE]*	59	3B	;	91	5B	[	123	7B	{
28	1C	*[FILE SEPARATOR]*	60	3C	<	92	5C	\	124	7C	|
29	1D	*[GROUP SEPARATOR]*	61	3D	=	93	5D	]	125	7D	}
30	1E	*[RECORD SEPARATOR]*	62	3E	>	94	5E	^	126	7E	~
31	1F	*[UNIT SEPARATOR]*	63	3F	?	95	5F	_	127	7F	*[DEL]*

C exercise 1: tolower

```
// tolower returns the corresponding
// lowercase character for c if c is an
// uppercase letter. Otherwise, it returns c.
char tolower(char c) {






}

int main() {
   char c = tolower(‘A’);
   printf(“resulting c is %c\n”, c);
}
```

C exercise 1: tolower

```
// tolower returns the corresponding
// lowercase character for c if c is an
// uppercase letter. Otherwise, it returns c.
char tolower(char c) {

    // test if c is an uppercase letter
    if (c < ‘A’ || c > ‘Z’) {
        return c;
    }


}
```

C exercise 1: tolower

```
// tolower returns the corresponding
// lowercase character for c if c is an
// uppercase letter. Otherwise, it returns c.
char tolower(char c) {

    // test if c is an uppercase letter
    if (c < ‘A’ || c > ‘Z’) {
        return c;
    }

    return c + ‘a’ – ‘A’;

}
```

C’s standard library includes tolower, toupper

C exercise 2: toDigit

```
// toDigit returns the corresponding integer for c
// if c is a valid digit character, e.g ‘1’, ‘2’,
// Otherwise, it returns -1.
int toDigit(char c) {





}

int main() {
   int d = toDigit(‘8’);
   printf(“int is %d, multiply-by-2 %d\n”, d, 2*d);
}
```

C exercise 2: toDigit

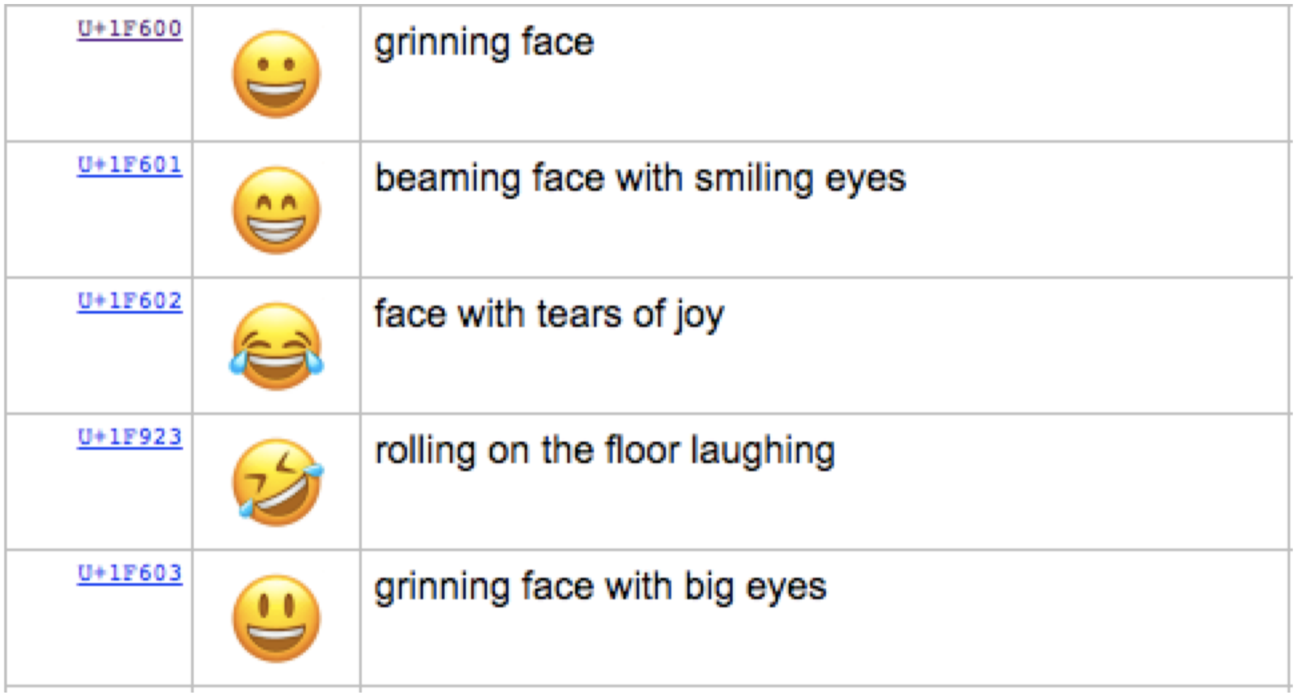
```
// toDigit returns the corresponding integer for c
// if c is a valid digit character, e.g ‘1’, ‘2’,
// Otherwise, it returns -1.
int toDigit(char c) {
     // test if c is a valid character
     if (c < ‘0’ || c > ‘9’) {
         return -1;
     }

}
int main() {
   int d = toDigit(‘8’);
   printf(“int is %d, multiply-by-2 %d\n”, d, 2*d);
}
```

C exercise 2: toDigit

```
// toDigit returns the corresponding integer for c
// if c is a valid digit character, e.g ‘1’, ‘2’,
// Otherwise, it returns -1.
int toDigit(char c) {
     // test if c is a valid character
     if (c < ‘0’ || c > ‘9’) {
         return -1;
     }
     return c – ‘0’;
}
int main() {
   int d = toDigit(‘8’);
   printf(“int is %d, multiply-by-2 %d\n”, d, 2*d);
}
```

The Modern Standard: UniCode

- ASCII can only represent 128 characters
  - How about Chinese, Korean, all of the worlds languages? Symbols? Emojis?
- Unicode standard represents >135,000 characters



U+1F600	😀	grinning face
U+1F601	😁	beaming face with smiling eyes
U+1F602	😂	face with tears of joy
U+1F923	🤣	rolling on the floor laughing
U+1F603	😃	grinning face with big eyes

UTF-8

- UTF-8 is one encoding form for Unicode
  - use 1, 2, or 4 byte to represent a character
  - Unicode for ASCII characters have the same ASCII value → UTF-8 one byte code is the same as ASCII
- C has no primitive support for Unicode

C Strings

Strings

- String is represented as an array of chars.
  - Array has no space to encode its length.
- How to determine string length?
  - explicitly pass around an integer representing length

```
// tolower_string turns every character in character array s
// into lower case
void tolower_string(char *s, int len) {
     for (int i = 0; i < len; i++) {
          s[i] = tolower(s[i]);
      }
}
```

Strings

- String is represented as an array of chars.
  - Array has no space to encode its length.
- How to determine string length?
  - explicitly pass around an integer representing length
  - C string stores a NULL character to mark the end (by convention)

```
void tolower_string(char *s) {


}
```

Strings

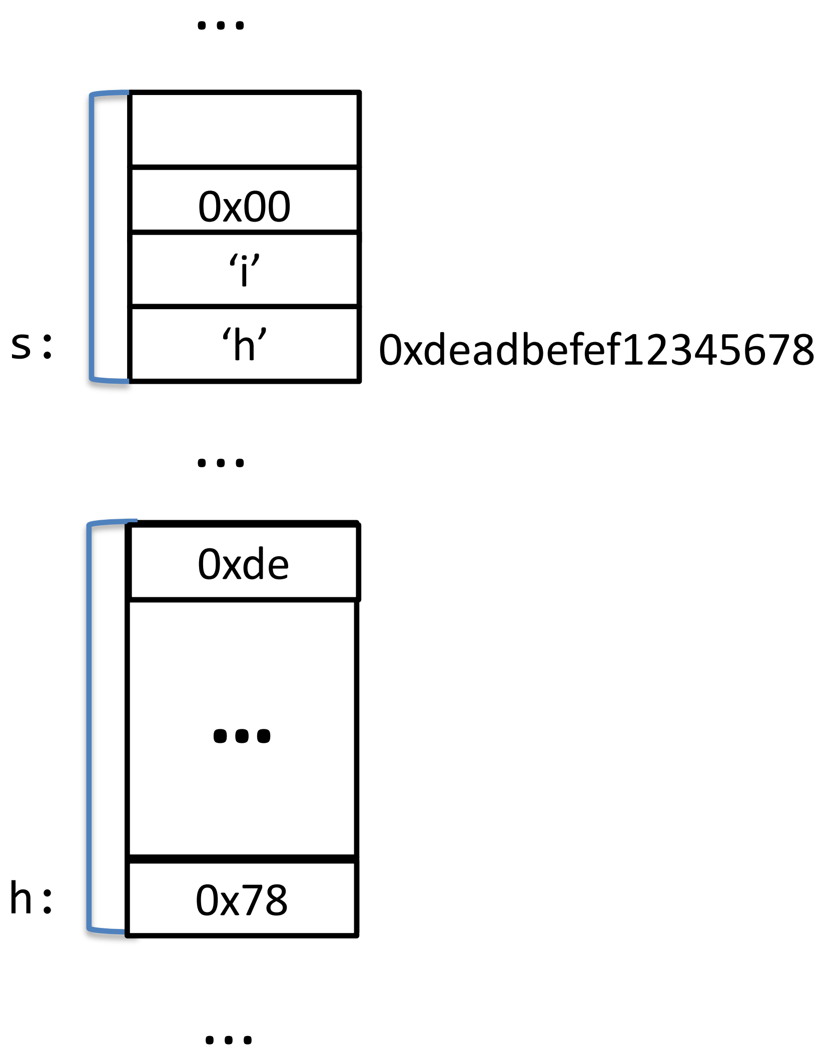
- String is represented as an array of chars.
  - Array has no space to encode its length.
- How to determine string length?
  - explicitly pass around an integer representing length
  - C string stores a NULL character to mark the end (by convention)

```
void tolower_string(char *s) {
    int i = 0;
    while (s[i] != ‘\0’) {
        s[i] = tolower(s[i]);
        i++;
    }
}
```

Copying string?

does this make a copy of "hi"?

```
char s[4] = "hi";
char *h;
h = s;
h[0] = 'H';

printf("s=%s h=%s\n",s,h);
```



Copying string?

does this make a copy of “hi”?

```
char s[4] = “hi”;
char h[4];
h = s;
h[0] = ‘H’;

printf(“s=%s h=%s\n”,s,h);
```

Copying string

```
void strcpy(char *dst, char *src)
{

}

int main()
{
   char s[4] = “hi”;
   char h[4];
   strcpy(h, s);
   h[0] = ‘H’;

   printf(“s=%s h=%s\n”,s,h);
}
```

Copying string

```
void strcpy(char *dst, char *src) {
    int i = 0;
    while (src[i] != ‘\0’) {
       dst[i] = src[i];
       i++;
    }
}

int main() {
   char s[4] = “hi”;
   char h[4];
   strcpy(h, s);
   h[0] = ‘H’;

   printf(“s=%s h=%s\n”,s,h);
}
```

strcpy is included in C std library.

Copying string

```
void strcpy(char *dst, char *src) {
    int i = 0;
    while (src[i] != ‘\0’) {
       dst[i] = src[i];
       i++;
    }
}

int main() {
   char s[4] = “hi”;
   char h[2];
   strcpy(h, s);
   h[0] = ‘H’;

   printf(“s=%s h=%s\n”,s,h);
}
```

Results in out-of-bound write!
Buffer overflow!

Copying string

```
void strncpy(char *dst, char *src, int n) {
    int i = 0;
    while (src[i] != ‘\0’ && i < n) {
        dst[i] = src[i];
        i++;
    }
}

int main() {
   char s[4] = “hi”;
   char h[2];
   strncpy(h, s, 2);
   h[0] = ‘H’;

   printf(“s=%s h=%s\n”,s,h);
}
```

strncpy is included in C std library.

A different way of initializing string …

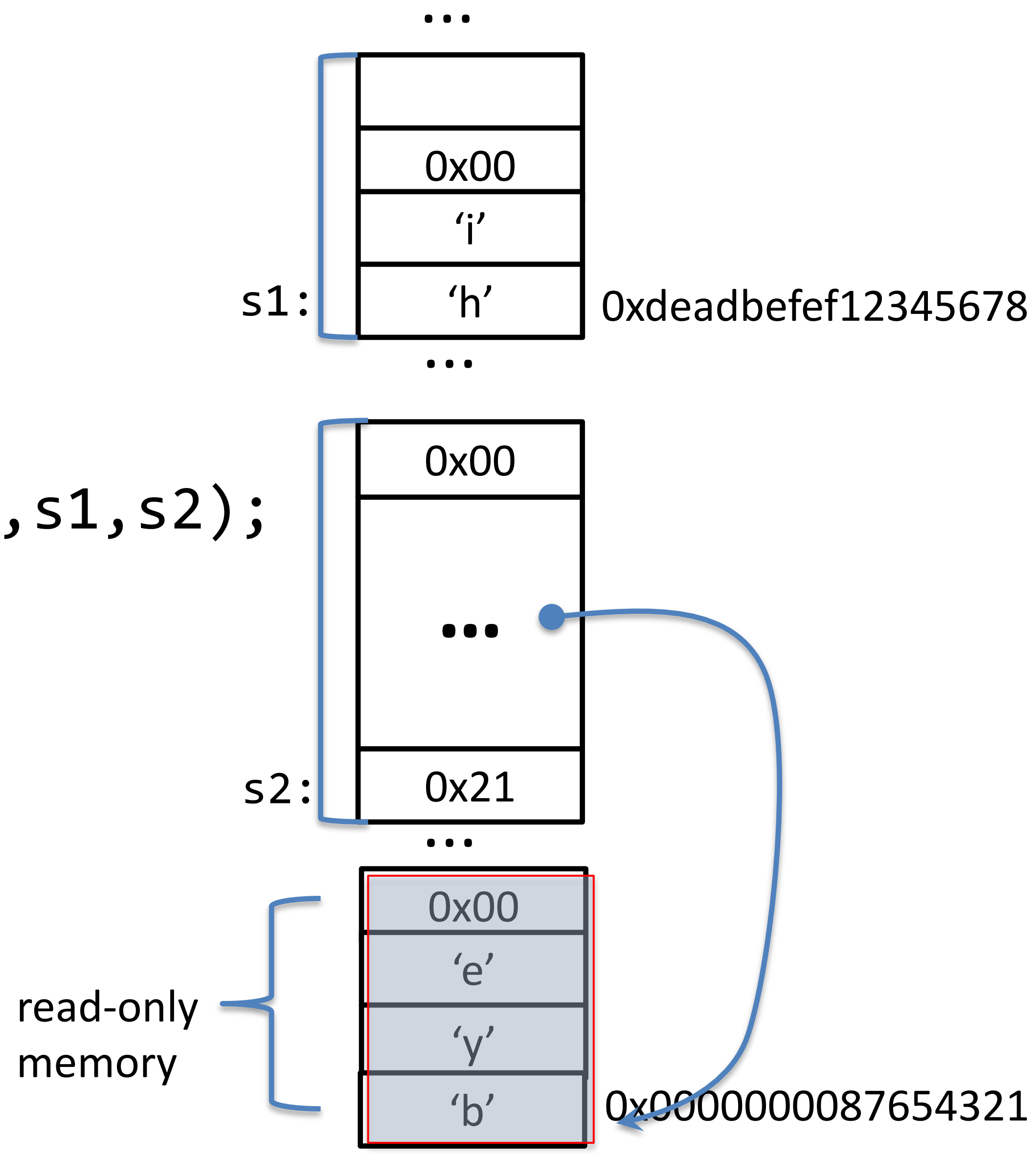
```
char s1[4] = “hi”;
char *s2 = “bye”;
s1[0] = ‘H’;
s2[0] = ‘B’;

printf(“s1=%s s2=%s\n”,s1,s2);
```

OK

Segmentation fault (bus error)

A different way of initializing string

```
char s1[4] = “hi”;
char *s2 = “bye”;
s1[0] = ‘H’;
s2[0] = ‘B’;

printf(“s1=%s s2=%s\n”,s1,s2);
```

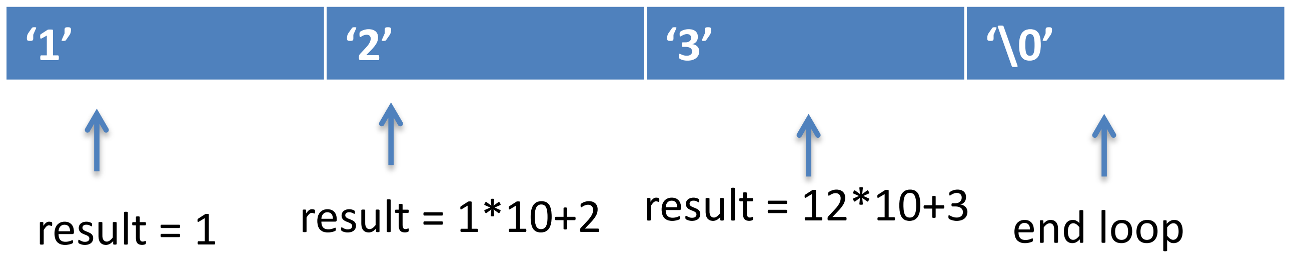


The Atoi function

```
// atoi returns the integer
// corresponding to the string of digits
int atoi(char *s)
{

}

int main()
{
   char *s= “123”;
   printf(“integer is %d\n”, atoi(s));
}
```

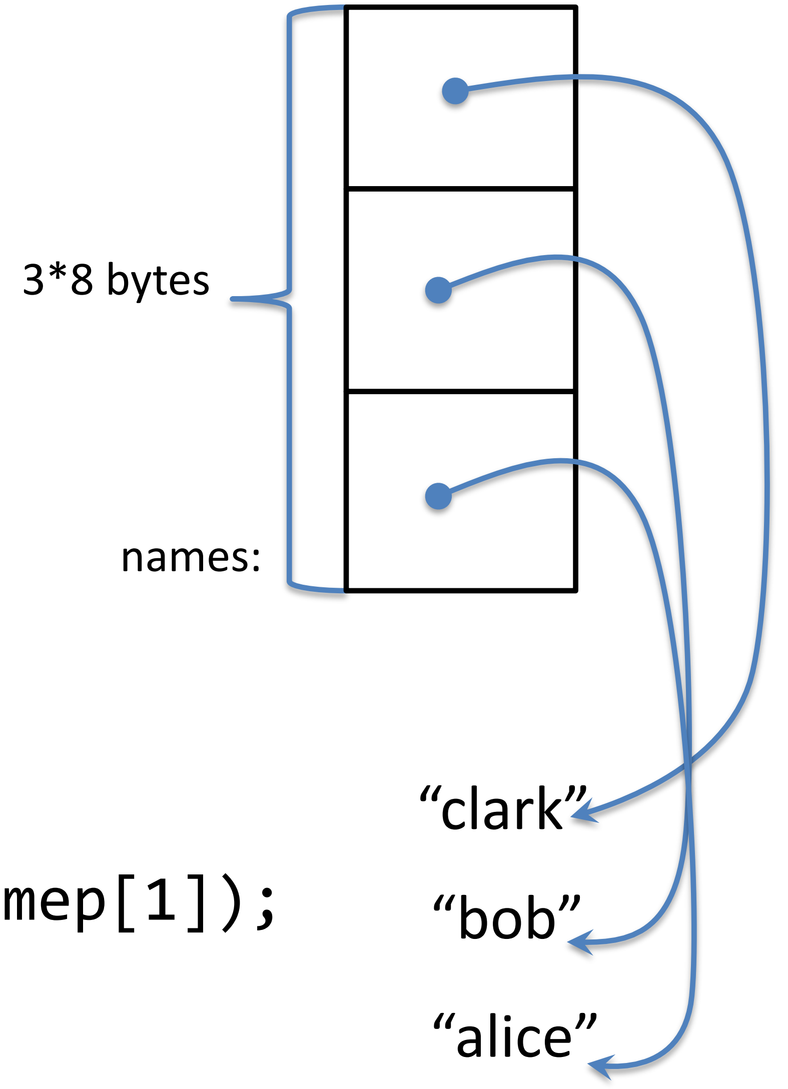
The Atoi function

```
// atoi returns the integer
// corresponding to the string of digits
int atoi(char *s) {
    int result = 0;
    int i = 0;
    while (s[i] >= '0' && s[i] <= '9') {
            result = result * 10 + (s[i] -'0');
            i++;
     }
     return result;
}
```



Array of pointers

```
char* names[3] = {
   “alice”,
   “bob”,
   “clark”
};
```



```
char **namep;
namep = names;

printf(“name is %s”, namep[1]);
```

The most commonly used array of pointers: argv

```
int main(int argc, char **argv)
{
   for (int i = 0; i < argc; i++) {
      printf(“%s\n”, argv[i]);
   }
}
```

```
$ ./a.out 1 2 3
./a.out 1 2 3
```

argv[0] is the name of the executable

Organization of large C programs

Linked list: one big file

```
typedef struct {
     int val;
     struct node *next;
 }node;

node* insert(node *head, int val) {
    node *new_head = (node *)malloc(sizeof(node));
    new_head->next = head;
    new_head->val = val;
}

int main() {
    node *head = NULL;
    for (int i = 0; i < 3; i++)
        head = insert(head, i);
}
```

list.c

What if another program also wants to use this linked list implementation?

linked list: multiple files

```
typedef struct {
     int val;
     struct node *next;
 }node;
node *insert(node *he
```

header file includes type definitions and exported function signatures

list.h

If not included, gcc does have info on the node type to compile list.c

```
#include “list.h”
node* insert(node *head, int val) {
    node *new_head = (node *)malloc(sizeof(node));
    new_head->next = head;
    new_head->val = val;
}
```

list.c

```
$ gcc -c list.c
```

generate object file list.o

```
$ gcc list.c
```

will not work since main() is not defined

linked list: multiple files

```c
#include “list.h”
int main() {
   node *head = NULL;
   for (int i = 0; i < 3; i++)
      head = insert(head, i);
}
```

test1.c

```c
#include “list.h”
int main() {
   node *head;
   for (int i = 0; i < 3; i++)
      head=insert(head, i);
}
```

test2.c

```
$ gcc -c test1.c
$ gcc test1.o list.o
$ ./a.out
```

generate object file test1.o

link test1.o and list.o to form executable a.out

Exporting global variables

list.h

```
typedef struct {
     int val;
     struct node *next;
 }node;
node *insert(node *head, int val);
```

list.c

```
#include “list.h”
int debug;
node* insert(node *head, int val) {
    ...
    if (debug > 0)
       printf(“inserted val %d\n”, val);
}
```

test1.c

```
#inde “list.h”
int main() {
   debug = 1;
   ...
}
```

Exporting global variables

```c
typedef struct {
     int val;
     struct node *next;
 }node;
extern int debug;
node *insert(node *head, int val);
```

Declares debug variable but does not allocate space

list.h

```c
#include “list.h”
int debug;
node* insert(node *head, int val) {
    ...
    if (debug > 0)
       printf(“inserted val %d\n”, val);
}
```

list.c

```c
#inde “list.h”
int main() {
   debug = 1;
   ...
}
```

test1.c

C does not have explicit namespace

- Scope of a global variable / function by default is across all files (linked together)
- To restrict the scope of a global variable / function to this file only, prefix with “static” keyword

No other files can use the debug variable and insert function

```
#include “list.h”
static int debug;
static node* insert(node *head, int val) {
    ...
    if (debug > 0)
       printf(“inserted val %d\n”, val);
}
```

list.c

static prefixing local variables means different things

- Normal local variables are de-allocated upon function exit
- Static local variables are not de-allocated
  - offers private, persistent storage across function invocation

initialized once, never deallocated (like a global variable, except with local scape)

```
node* insert(node *head, int val) {
    static int n_inserts = 0;
    ...
    n_inserts++;
    printf("number of inserts %d\n", n_inserts);
}
```

C standard library

<assert.h> assert

<ctype.h> isdigit(c), isupper(c), isspace(c), tolower(c), toupper(c) ..

<math.h> log(f) log10(f) pow(f, f), sqrt(f), ...

<stdio.h> fopen, fclose, fread, fwrite, printf, ...

<stdlib.h> malloc, free, atoi, rand

<string.h> strlen, strcpy, strcat, strcmp

Section 3 of manpage is dedicated to C std library

To read manual, type

```
man 3 strlen
```

The C pre-processor

- All the hashtag directives are processed by C pre-processor before compilation
- #include <stdio.h>
  - insert text of included file in the current file
  - with <...> , preprocessor searches system path for specified file
  - with "...", preprocessor searches local directory as well as system path

C Macros

- #define name replacement_text

```
#define NITER 10000

int main()
   for (int i = 0; i < NITER; i++) {
       ....
   }
}
```

It's better to write:

```
static const int niter = 10000;
```

C Macros

- Macro can have arguments
- Macro is NOT a function call

```
#define SQUARE(X) X*X

a = SQUARE(2);                   a = 2*2;

b = SQUARE(i+1);                 b = i+1*i+1;

c = SQUARE(i++);
```

C Macros

- Macros can have arguments
- Macro is NOT a function call

```
#define SQUARE(X) (X)*(X)

a = SQUARE(2);                    a = (2)*(2);

b = SQUARE(i+1);                  b = (i+1)*(i+1);

c = SQUARE(i++);                  c = (i++)*(i++);   ✗
```

Macro is hard to debug, avoid it if you can