Chapter 7

Program Organization

What is in This Chapter ?

This chapter discusses a couple of things related to keeping our code organized nicely. It begins with an explanation of **scope** as it pertains to variables. Variables can be accessed from other files, and scoping helps us understand what we can and cannot do. The separation between **src** and **header** files is explained a bit. Finally, we discuss how to create our own **libraries**.

7.1 Variable Details and Scope

You have been using variables for a long time now. This section discusses a little bit of the finer details in regard to how/where variables are stored as well as how they are accessed and where they are accessible from (i.e., within our program or outside of it). Here is an example of a variable declaration:

```
static const unsigned int     myVar = 780;
```

There are 4 keywords being used here. In general, a variable declaration has this format:

<storage class> <qualifier> <modifier> <data type> **<name>** [= <initial value>]**;**

Here is what each of these mean:

<data type>
- indicates the number of bytes and bit model to be used
  e.g., `int`, `float`, `char` or `double`

<modifier>
- modifies # of bytes specified by the data type (defines how the bits are used)
  e.g., `signed`, `unsigned`, `short`, `long`

<qualifier>
- specifies information to allow optimized compilation
  e.g., `const`, `volatile`

<storage class>
- specifies area of memory where variable is to be stored, maybe in another file
  e.g., `static`, `extern`, `register`

Although we discussed data types and modifiers already, let us be a bit clearer now on the options. Here is a complete list of the primitive data type possibilities:

Variable Declaration		Minimum Value	Maximum Value
`int`	`i1;`	-2,147,483,648	2,147,483,647
`short int`	`i2;`	-32,768	32,767
`long int`	`i3;`	-9,223,372,036,854,775,808	9,223,372,036,854,775,807
`signed int`	`i4;`	-2,147,483,648	2,147,483,647
`signed short int`	`i5;`	-32,768	32,767
`signed long int`	`i6;`	-9,223,372,036,854,775,808	9,223,372,036,854,775,807
`unsigned int`	`i7;`	0	4,294,967,295
`unsigned short int`	`i8;`	0	65,535
`unsigned long int`	`i9;`	0	18,446,744,073,709,551,615
`char`	`c1;`	-128	127
`signed char`	`c2;`	-128	127
`unsigned char`	`c3;`	0	255
`float`	`f1;`	1.175494351e-38	3.402823466e+38
`double`	`d1;`	2.2250738585072014e-308	1.7976931348623158e+308

Notice that it does NOT make sense to have any of the following:

```
short double     signed short double     unsigned short double
```

```
short char       signed short char       unsigned short char
long  char       signed long  char       unsigned long  char
short float      signed short float      unsigned short float
long  float      signed long  float      unsigned long  float
short double     signed short double     unsigned short double
long  double     signed long  double     unsigned long  double
```

Now what about the qualifiers? These are used by the compiler to improve program efficiency.

When we use const, we are telling the compiler that the value will not be changed in the program. Consider the following code:

```
const int  age = 18;

age = 21;
printf("age: %d\n", age);
```

The above code will not compile since we will not be allowed to assign a value to **age**. The compiler will give this error:

```
error: assignment of read-only variable 'age'
```

So, we can set the value of **age** only once, when we declare it. But it cannot be altered for the remainder of the program. What then is the difference between these two?

```
#define    AGE 18

const int  age = 18;
```

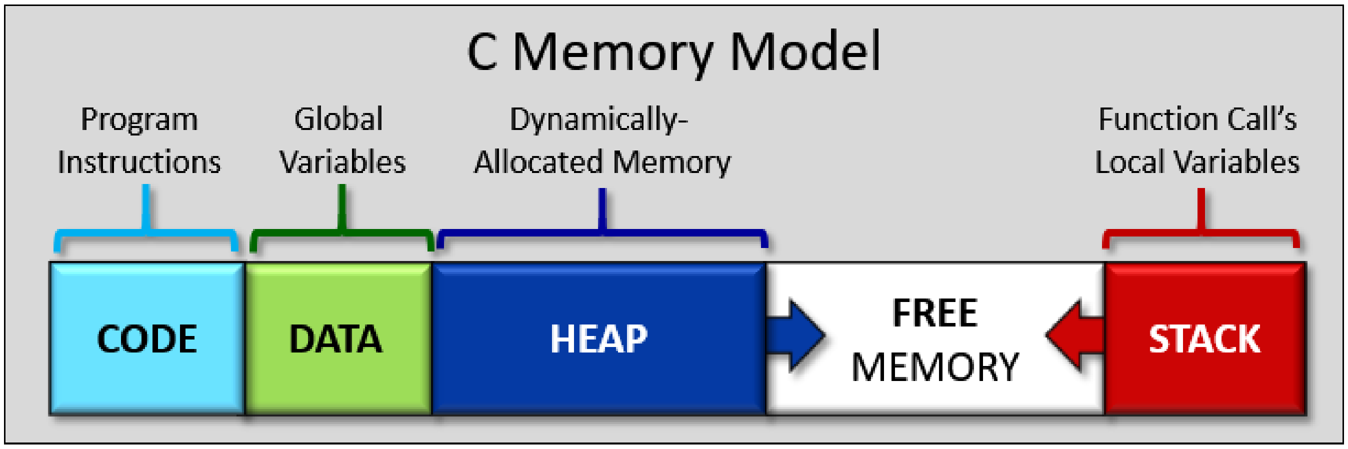
Well, the **#define** is a directive that is done before compiling. It simply goes through your code and replaces the word **AGE** (wherever it is found) by the number **18**. Then the code is compiled. When using **const int age**, however, this actually declares and stores the variable with the value of **18** in your program. This allows you to pass the variable around in your program and it will be type-checked. So, **const** variables basically allow for type-checked constants, whereas **#define** does not check types. The storage location of **const** variables will depend on the compiler.

What about the **volatile** keyword ?

```
volatile int    counter = 0;
```

Recall, when we discussed threads, that we should declare a variable as **volatile** whenever there is a chance that another thread/process may come along and change it at any time unexpectedly. It informs the compiler not to do any optimization with this variable. What kind of optimizations is the compiler trying to do anyway? That is a topic for a compiler construction course. For now, just know that clever compilers may try to do things such as caching variables in registers and changing the order that assignments are evaluated in.

The final variable parameter that needs discussion is the *storage class*. Recall the C memory model that is used to store program code, global/local variables and data within allocated memory:



The storage class parameter allows us to indicate whether the variable is to be stored in the **data segment** or the function call **stack**. If we do not indicate any storage class, the default is to store the variable on the function call **stack**. This is the usual case for local variables and function parameters.

If we use the **static** keyword, then the variable is stored in the **data segment** (i.e., global memory). That means that the variable stays around throughout the entire program. It keeps its value until the program ends. However, it is only visible within the block of code that it is declared in. Can you guess what the output will be for this program?

Code from **static.c**

```
#include <stdio.h>

void function1() {
  static int i = 0;

  i = i + 2;
  printf("%d ", i);
}
void function2() {
  int i = 0;

  i = i + 2;
  printf("%d ", i);
}
int main() {
  function1();
  function1();
  function1();
  printf("\n");
  function2();
  function2();
  function2();
  printf("\n");
}
```

Here it is:

```
2 4 6
2 2 2
```

Do you understand why? The only difference between the functions is the **static** keyword. In **function2**, the variable **i** is initialized each time that the function is called. However, in **function1**, the **static** variable **i** is NOT initialized each time the function is called. It keeps its previous value. So, whenever the compiler finds a **static** variable in the code that is being initialized, it does not re-evaluate the initialization code again.

When we use the **extern** keyword, this indicates that a variable is global and actually declared in another file.

```
extern float    interestRate;
```

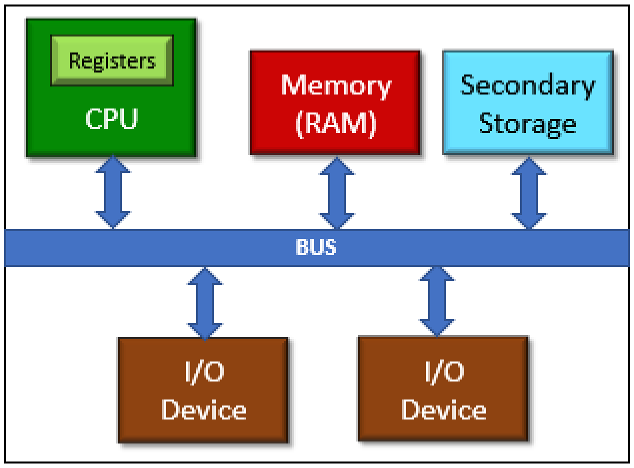
It is similar to the notion in JAVA where we access static/class variables from other classes:

```
Bank.InterestRate
BankAccount.LatestAccountNumber
```

The **register** keyword allows us to tell the compiler to store the variable in a register.

```
register float  counter;
```

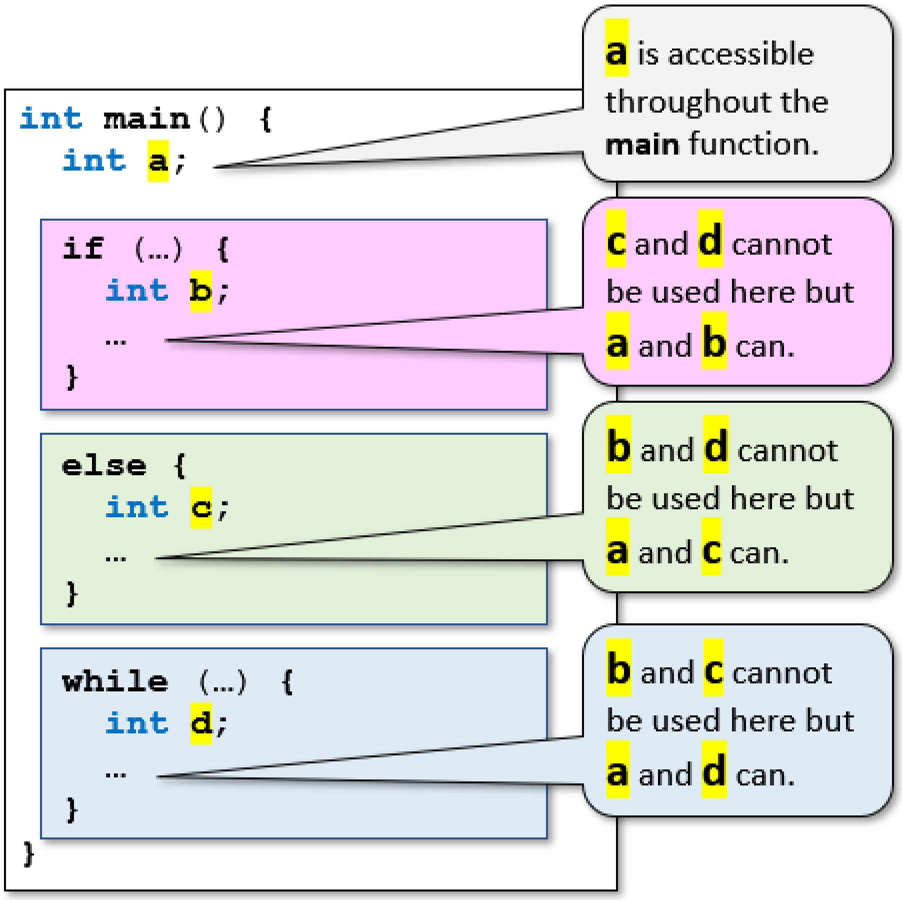
Registers are inside the CPU. Therefore, when we perform operations using data within registers, it is much faster, since we do not have to first go access the variable from the memory, which is much slower. When doing assembly coding, fast code hinges on the proper use of registers … knowing just when and how long to keep data in a register. So, doing this can speed up your program … however … it is often better to rely on the compiler and assembler to optimize such things. Some compilers may actually ignore the request to store the value in a register as opposed to regular memory.



It is important to fully understand the scope of variables:

*The **scope** of a variable indicates where the variable can be used in the program.*

Typically, a variable is declared and used in what is called a ***block scope***. In C, a ***block*** is any sequence of statements between a pair of braces. Consider this code template that defines 4 variables. Each variable is defined within its own colored block and is therefore only visible within that block and hence inaccessible outside that block:


int main() {
int a;
if (…) {
int b;
…
}
else {
int c;
…
}
while (…) {
int d;
…
}
}
a is accessible throughout the main function.
c and d cannot be used here but a and b can.
b and d cannot be used here but a and c can.
b and c cannot be used here but a and d can.

This block scoping should be somewhat intuitive to us all by now. At the closing of a brace **}** , all variables defined in that block no longer exist in the program. So, in the above code, we could have renamed both variables **c** and **d** to **b**, and there would be no confusion since in each of the inner blocks, the other two variables do not exist.

It can be a bit tricky sometimes if a variable name is used more than once when dealing with nested loops. For example, consider this code that defines variable **b** twice:

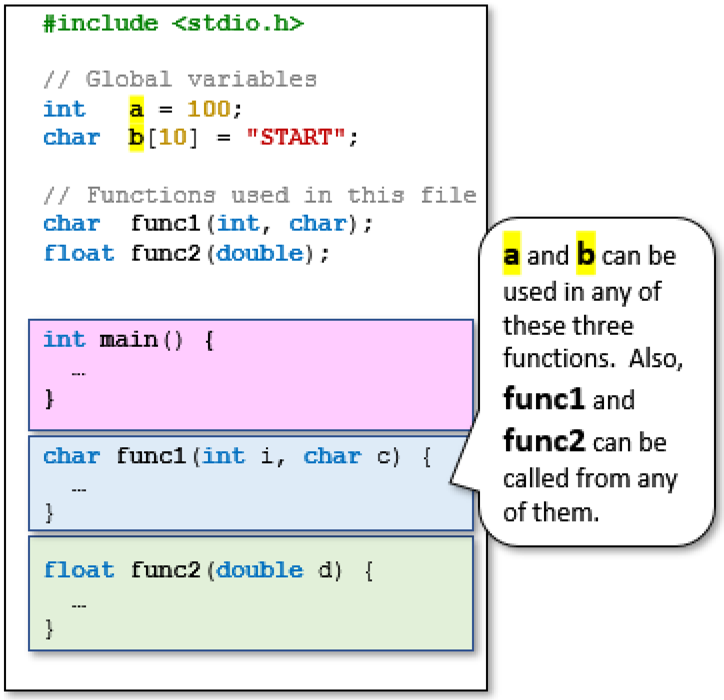
```
int a = 0;

while (a < 2) {
  int b = 8;
  while (a < 2) {
    int b;
    b = a++;
    printf("INNER: a=%d, b=%d\n", a, b);
  }
  printf("OUTER: a=%d, b=%d\n", a, b);
}
```

The compiler allows **b** to be defined in the inner **while** loop as well as the outer. The output is:

```
INNER: a=1, b=0
INNER: a=2, b=1
OUTER: a=2, b=8
```

Notice that the outer **while** loop's variable **b** is unusable from within the inner **while** loop because the redeclaration of **b** essentially hides the outer declaration. It is not a good idea to re-use a variable name like this because the code becomes unintuitive.

Another kind of scoping that we use regularly is ***file scope***. Any variable that is declared outside any block is usable anywhere within the program file. This kind of scoping is used for **global variables** (i.e., accessible from all code in the file), **functions** (we want to be able to call them from anywhere in the code) and **function prototypes/signatures** (tells the compiler that the function is to be found somewhere in the file). An example is shown here on the right.



```
#include <stdio.h>

// Global variables
int   a = 100;
char  b[10] = "START";

// Functions used in this file
char  func1(int, char);
float func2(double);


int main() {
  …
}

char func1(int i, char c) {
  …
}

float func2(double d) {
  …
}
```

a and **b** can be used in any of these three functions. Also, **func1** and **func2** can be called from any of them.

Interestingly, through use of the **extern** keyword, these global variables **a** and **b**, as well as the functions **func1** and **func2**, can be accessed from other files.

Notice, in the above example, that **func1** and **func2** define some incoming parameters. These parameters have what is called ***function prototype scope***. That means, the parameters are essentially just variables that exist for as long as the function is running and cannot be used outside of the function.

A final type of scoping is that of ***function scope***. Within a function, we can declare ***labels*** in our code. A **label** identifies a place in the code that we can jump to using the **goto** statement. Here is an example that attempts to identify whether or not there is a student who received **3** or more zero grades, in which case they failed. It repeatedly asks for student grades, one student at a time. A value of **-1** is entered to go to the next student. If at any time a student has been found who has **3** zero grades, then the checking is complete, since we have found a failed student. The code jumps out of both **while** loops by using a **goto** statement to go to the code specified by the **done** label.

```
while (1) {
  int num=0, zeroCount = 0;
  printf("Enter a student's grades:\n");
  while (num != -1) {
    scanf("%d", &num);
    if (num == 0)
      zeroCount++;
    if (zeroCount > 2)
      goto done;
  }
}
done: printf("Found failed student\n");
```

The code works. However, the use of **goto** statements is nearly always a bad idea … unless somehow you are forced to do it. In the above code, it is simply better to use a stopping *flag* as follows:

```
char stillLooking = 1;
while (stillLooking) {
  int num=0, zeroCount = 0;
  printf("Enter a student's grades:\n");
  while (num != -1) {
    scanf("%d", &num);
    if (num == 0)
      zeroCount++;
    if (zeroCount > 2) {
      stillLooking = 0;
      num = -1;
    }
  }
}
printf("Found failed student\n");
```

There may be some very tricky cases where simple solutions like this are not easy to figure out, but that usually means that some poor coding design choices have been made along the way. Please … do your best to NEVER use **labels** and **goto** statements.

See if you can figure out the result of the following program which is split across two files called **scope.c** and **util.c**:

Code from **scope.c**

```
#include <stdio.h>

extern void simpleFunc(int x);   // This is defined in "util.c"

int x = 10;                      // This will be accessed from "util.c"

int main() {
  x = x + 10;
  printf("main:   x = %d\n", x);

  int x = 5;
  x = x + 10;
  printf("main:   x = %d\n", x);

  {
    int x = 8;
    x = x + 10;
    printf("inner:  x = %d\n", x);
  }

  printf("main:   x = %d\n", x);
  simpleFunc(x);
  printf("main:   x = %d\n", x);
}
```

Code from **util.c**

```
#include <stdio.h>

extern int   x;              // This is defined in "scope.c"

void simpleFunc(int z) {
  x = x + 10;
  printf("func:   x = %d\n", x);
}
```

Of course, since there are two files to compile, we will need to include both of them in the **gcc** command line. Here is the compiling command and the output (with highlighted results):

```
student@COMPBase:~$ gcc -o scope scope.c util.c
student@COMPBase:~$ ./scope
main:    x = 20
main:    x = 15
inner:   x = 18
main:    x = 15
func:    x = 30
main:    x = 15
student@COMPBase:~$
```

There are a couple of things that you should notice. First, the **x** defined within the inner block (defined by braces) of the **main** function doesn't alter the **x** defined earlier in the **main** function nor does it affect the global variable. Second, the **x** used in the **simpleFunc()** function is the global **x** from the **scope.c** file … not any of the **x** variables defined in the **main** function.

This was just an example, but you should not be creating so many variables with the same name. It is just bad programming practice. Also, it is usually not a very good idea to be using a lot of global variables defined in other files, as this makes it hard to understand (and keep track of) code in a large system. Nevertheless, it is sometimes very convenient to have access to a global variable throughout may program files. But the convenience comes with a cost of having spaghetti-like code that is not modular.

As you can guess, the **extern** keyword can be useful since typically our programs are spread across multiple files. Typically, a program may have many source files as well as some header files. It is good to split your code across multiple files because this makes your code more readable, easier to modify and modular.

Header files do not contain code but typically contain:

- **global constant** declarations
- global **type definitions**
- forward declarations of **function prototypes**

Source files contain:

- function implementations (i.e., our actual code)
- **global variable** declarations
- **constants**, **types**, function **prototypes** needed only in that file

Regarding the constants, types and prototypes … if there are few, declare them in the source file. If there are many, make a separate header file for them.

How do we know what goes into a header file ? Typically, we group together all the declarations that are needed by multiple source files. If something is only needed by a single file, then it does not belong in a header file.

How do we know what goes into a single source file ? Typically, we group together all functions that are related to a purpose of some sort. Here is a list of files that I used when doing my PhD work on shortest path algorithms. From the comments, notice what kind of code was in the c **source** files and what definitions were in the **header** files.

spmain.c	- the **main program** to start the code
sp.c	- code that implements the **approximate shortest path algorithm**
schemes.c	- code to generate **approximation schemes**
chenhan.c	- code pertaining to a particular **shortest path algorithm by Chen & Han**
spgui.c	- code pertaining to the **GUI for displaying** shortest paths
funnel.c	- code for creating and manipulating a **funnel** data structure
graph.c	- code for creating and manipulating a **graph** data structure
sleeve.c	- code for creating and manipulating a **sleeve** data structure
spheap.c	- code for creating and manipulating a **heap** data structure
tin.c	- code for creating and manipulating a **tin** data structure
graphgen.c	- code to generate **graphs for testing**
vdmhostlib.c	- code allowing **networked communications** to the display manager
easyMotif.c	- code for managing **GUI windows**

chenhan.h	- definitions pertaining to the **Chen and Han algorithm**
colors.h	- definitions needed by the **GUI** for color display purposes
graph.h	- definitions needed when using the **graph** data structure
sp.h	- definitions needed when using the **shortest path** data structure
tin.h	- definitions needed when using the **tin** data structure
vdmlib.h	- definitions needed when **communicating** with the display manager
easyMotif.h	- definitions needed to open and manipulate **windows**

7.2 Libraries

When programming, it is often the case that we make use of existing library functions:

A ***library*** *is simply a set of commonly used functions that are reusable and can be used by many different programs.*

The purpose of a library is to prevent programmers from having to re-write code that has already been written adequately by someone else. There are advantages to using them:

- saves time ... no need to “reinvent the wheel”.
- it will also help to keep errors to a minimum since library functions are usually properly tested and debugged.
- code written by experts who have spent a lot of time on it.
- code usually written painstakingly in order to provide a correct implementation.
- code optimized for peak performance.
- it promotes reusability and portability.

A library is NOT an executable file. It does NOT contain a **main** function. So, it is NOT a program. It consists of header file(s) as well as object files (which are all archived into one file with a **.a** extension).

So how do we use a library ?

There will be a header file that corresponds to that library. We should include that in our program. We have already been doing this to include some standard library functions:

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

These headers, by default, are in the location of **/usr/include** in the file system. You can navigate there as shown in this example shell output:

```
student@COMPBase:~$ cd /home
student@COMPBase:/home$ ls
student
student@COMPBase:/home$ cd ..
student@COMPBase:/$ ls
bin    dev   initrd.img      lost+found  opt   run   srv  usr      vmlinuz.old
boot   etc   initrd.img.old  media       proc  sbin  sys  var
core   home  lib             mnt         root  snap  tmp  vmlinuz
student@COMPBase:/ $ cd usr
student@COMPBase:/usr$ ls
bin  games  include  lib  local  locale  sbin  share  src
```

```
student@COMPBase:/usr$ cd include
student@COMPBase:/usr/include$ ls
aio.h          fcntl.h         math.h         pwd.h          term_entry.h
aliases.h      features.h      mcheck.h       python3.5m     term.h
alloca.h       fenv.h          memory.h       rdma           termio.h
argp.h         fmtmsg.h        menu.h         re_comp.h      termios.h
argz.h         fnmatch.h       misc           regex.h        tgmath.h
ar.h           form.h          mntent.h       regexp.h       thread_db.h
arpa           fstab.h         monetary.h     reglib         tic.h
asm-generic    fts.h           mqueue.h       resolv.h       time.h
assert.h       ftw.h           mtd            rpc            ttyent.h
autosprintf.h  _G_config.h     nc_tparm.h     rpcsvc         uapi
byteswap.h     gconv.h         ncurses_dll.h  sched.h        uchar.h
c++            getopt.h        ncurses.h      scsi           ucontext.h
complex.h      gettext-po.h    net            search.h       ulimit.h
cpio.h         GL              netash         semaphore.h    unctrl.h
crypt.h        glob.h          netatalk       setjmp.h       unistd.h
ctype.h        gnumake.h       netax25        sgtty.h        ustat.h
cursesapp.h    gnu-versions.h  netdb.h        shadow.h       utime.h
cursesf.h      grp.h           neteconet      signal.h       utmp.h
curses.h       gshadow.h       netinet        sound          utmpx.h
cursesm.h      i386-linux-gnu  netipx         spawn.h        valgrind
cursesp.h      iconv.h         netiucv        stab.h         values.h
cursesw.h      ifaddrs.h       netpacket      stdc-predef.h  video
cursslk.h      inttypes.h      netrom         stdint.h       wait.h
dirent.h       langinfo.h      netrose        stdio_ext.h    wchar.h
dlfcn.h        lastlog.h       nfs            stdio.h        wctype.h
drm            libdrm          nl_types.h     stdlib.h       wordexp.h
elf.h          libgen.h        nss.h          string.h       X11
endian.h       libintl.h       obstack.h      strings.h      xcb
envz.h         libio.h         panel.h        stropts.h      xen
err.h          libsync.h       paths.h        sudo_plugin.h  xf86drm.h
errno.h        limits.h        poll.h         syscall.h      xf86drmMode.h
error.h        link.h          printf.h       sysexits.h     xlocale.h
eti.h          linux           protocols      syslog.h       xorg
etip.h         locale.h        pthread.h      tar.h          zconf.h
execinfo.h     malloc.h        pty.h          termcap.h      zlib.h
student@COMPBase:/usr/include$
```

As you can see, there are many header files. I have highlighted (in yellow) a few familiar ones that we have used already.

Once we include the header file in our code, we then need to compile and link to the library. Where are the libraries located ? Well, it depends on the installation. In our system, it is installed in **/usr/lib/x86_64-linux-gnu**. The standard library functions are located in the **libc.a** library in that directory. As it turns out, library files have a **.a** extension.

But we don't have to link to the standard libraries because this is done by default when we compile. Recall, however, that we had to include **`<math.h>`** when we wanted to use trigonometry functions and then we had to include the library using **`-lm`** when we compiled:

```
student@COMPBase:~$ gcc -o trig trig.c -lm
student@COMPBase:~$
```

The **-l** (that's an "L", not a "1") here indicates that we want to use some functions that are in a particular library. In this case, it was the math library, which has the name **libm.a** and is located here: **/usr/lib/x86_64-linux-gnu**.

Now, let's see how we can create our own library and link to it from one of our programs …

Consider the following structure that defines a first and last name for a person:

```
#define MAX_STR 32

typedef struct {
  char first[MAX_STR];
  char last[MAX_STR];
} NameType;
```

Now consider a function that gets a first and last name from the user:

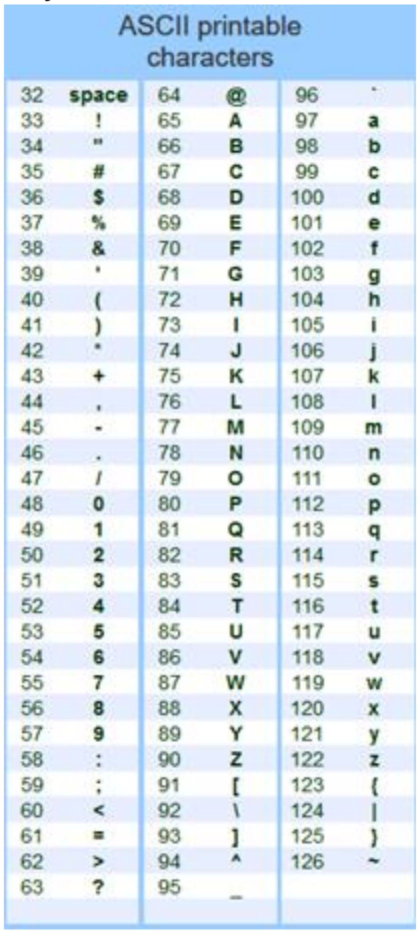
```
void enterName(NameType *name) {
  printf("\nEnter a name (e.g., John Doe): ");
  scanf("%s %s", name->first, name->last);
}
```

This is a useful function that may be nice to have in a library, as there may be many applications that require the input of people's names. The name may actually be entered erroneously … but we won't check for this. Assume that the name has been entered properly. However, what if we want to make sure that we have consistency in capitalization for all the names. If, for example, the application is at a kiosk, then people may enter various forms of capitalization of the name such as:

`mark lanthier`	(e.g., user ignores capitals)
`Mark Lanthier`	(e.g., this is what we'd like)
`MARK LANTHIER`	(e.g., caps lock is on)
`mARK lANTHIER`	(e.g., user shifts to capitalize, not realizing caps lock is on)

How can we make the capitalization consistent and matching the format of the 2nd input above? We'd have to alter the characters. Assume that we do one name at a time. How do we ensure that we get "Mark" regardless of whether they enter "mark", "MARK", "mARK" or any other combination of caps?

Well, recall that chars are just numbers that correspond to ASCII codes as shown here on the right. Notice that there is a difference between ASCII codes for **'A'** and **'a'** … which is **97** - **65** = **32**. This is the same difference between codes for **'B'** and **'b'**, **'C'** and **'c'** … **'Z'** and **'z'**. So, we can just check to see if the first character is between **'a'** and **'z'** (i.e., between **97** and **122**) and then subtract **32** from that value to make it between (**65** and **90**). For all other letters, if we find that they are capitalized, we just add **32**.



ASCII printable characters

32	space	64	@	96	`
33	!	65	A	97	a
34	"	66	B	98	b
35	#	67	C	99	c
36	$	68	D	100	d
37	%	69	E	101	e
38	&	70	F	102	f
39	'	71	G	103	g
40	(	72	H	104	h
41	)	73	I	105	i
42	*	74	J	106	j
43	+	75	K	107	k
44	,	76	L	108	l
45	-	77	M	109	m
46	.	78	N	110	n
47	/	79	O	111	o
48	0	80	P	112	p
49	1	81	Q	113	q
50	2	82	R	114	r
51	3	83	S	115	s
52	4	84	T	116	t
53	5	85	U	117	u
54	6	86	V	118	v
55	7	87	W	119	w
56	8	88	X	120	x
57	9	89	Y	121	y
58	:	90	Z	122	z
59	;	91	[	123	{
60	<	92	\	124	|
61	=	93	]	125	}
62	>	94	^	126	~
63	?	95	_		

Here is a function to do this:

```
void capFix(char *str) {
  // Check the first letter and capitalize it
  if (str[0] >= 'a' && str[0] <= 'z')
    str[0] = str[0] - 32; // 32 is ASCII diff between 'A' and 'a'

  // Uncapitalize the remaining letters
  for (int i=1; i<strlen(str); i++)
    if (str[i] >= 'A' && str[i] <= 'Z')
      str[i] = str[i] + 32; // 32 is ASCII diff between 'A' and 'a'
}
```

We will place these two functions into the library that we will create. Of course, when we want to use this library we will need to include a header file with the function prototypes in it, as well as the structure definition. So, we will start with this **"names.h"** header file:

Code from **names.h**

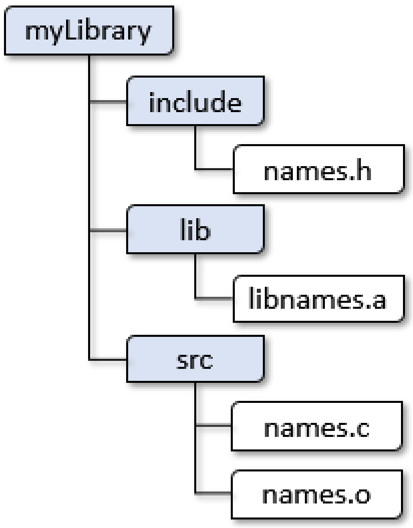
```
#define MAX_STR 32

typedef struct {
  char first[MAX_STR];
  char last[MAX_STR];
} NameType;

extern void enterName(NameType *);
extern void capFix(char *);
```

The library source code will be in the following **names.c** file:

Code from **names.c**

```
#include <stdio.h>
#include <string.h>
#include "names.h"   // this is our own header, hence double quotes

// Get a first and last name from stdin
void enterName(NameType *name) {
  printf("\n");
  printf("Enter a name: ");
  scanf("%s %s", name->first, name->last);
}

// Fix name by capitalizing it properly
void capFix(char *str) {
  // Check the first letter and capitalize it
  if (str[0] >= 'a' && str[0] <= 'z')
    str[0] = str[0] - 32; // 32 is ASCII diff between 'A' and 'a'

  // Uncapitalize the remaining letters
  for (int i=1; i<strlen(str); i++)
    if (str[i] >= 'A' && str[i] <= 'Z')
      str[i] = str[i] + 32; // 32 is ASCII diff between 'A' and 'a'
}
```

In order to stay organized, we will set up the folder structure shown here, called **myLibrary**. It will have some subfolders to hold the **include** file(s), **lib** file and **src** files(s).



We will begin the library creation by compiling the **src** file(s). In our case, we only have one **src** file and one **header** file. So, we can go into the **src** file folder and compile as shown below.

Notice that we are using the `-I` option when compiling. This allows us to indicate a path to any *include*/header files that we want to use. In this case, the *include* file is in the **include** folder which is one level up (i.e., `../`) from the **src** folder that we are compiling in. Pay careful attention to the spacing, capitalization and slash characters. We have to get these things just right.

```
student@COMPBase:~/myLibrary/src$ ls
names.c
student@COMPBase:~/myLibrary/src$ gcc -c names.c -I ../include/
student@COMPBase:~/myLibrary/src$ ls
names.c  names.o
student@COMPBase:~/myLibrary/src$
```

To create the library file that we will link to from our code, we need to use the **ar** binary utility for archiving in linux, which is **ar**. It has many options, but we'll be using the **rs** options which means that we are inserting one or more files into the archive. By default, the libraries that we create should be named **libXXX.a**, where **XXX** is of our choosing. If we want to call the library **names**, then we should create a file called **libnames.a**. After we create the library, we will move it to the **lib** folder and remove the **.o** file. Here are the commands

```
student@COMPBase:~/myLibrary/src$ ls
names.c  names.o
student@COMPBase:~/myLibrary/src$ ar rs libnames.a names.o
ar: creating libnames.a
student@COMPBase:~/myLibrary/src$ ls
libnames.a  names.c  names.o
student@COMPBase:~/myLibrary/src$ mv libnames.a ../lib/
student@COMPBase:~/myLibrary/src$ rm names.o
student@COMPBase:~/myLibrary/src$ ls
names.c
student@COMPBase:~/myLibrary/src$ cd ../lib
student@COMPBase:~/myLibrary/lib$ ls
libnames.a
student@COMPBase:~/myLibrary/lib$
```

Now we are ready to try and use the library. We will write a program that will make use of this newly-created library that will simply ask the user to enter a first and last name, and then print out the name with proper capitalization. It will do this repeatedly, until a **"-1 -1"** name has been entered.

Here is the code:

Code from **capitalize.c**

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

int main() {
  NameType    newName;

  while(1) {
    enterName(&newName);

    if (strcmp(newName.first,"-1") == 0 && strcmp(newName.last,"-1") == 0)
      break;

    capFix(newName.first);
    capFix(newName.last);

    printf("My name is %s %s\n", newName.first, newName.last);
  }
}
```

We need to compile and link this code, making use of the header file and including the library that we created. We will back up from the **myLibrary** folder:

```
student@COMPBase:~/myLibrary/lib$ ls
libnames.a
student@COMPBase:~/myLibrary/lib$ cd ..
student@COMPBase:~/myLibrary/$ ls
include  lib  src
student@COMPBase:~/myLibrary/$ cd ..
student@COMPBase:~$ ls
capitalize.c  myLibrary  scope  scope.c  static  static.c  util.c
student@COMPBase:~$
```

To compile the program, we will again need to specify the path to the include file by using the `-I` option in **gcc** which allows us to specify the path to include files.

```
student@COMPBase:~$ ls
capitalize.c  myLibrary  scope  scope.c  static  static.c  util.c
student@COMPBase:~$ gcc -c capitalize.c -I myLibrary/include
student@COMPBase:~$ ls
capitalize.c  capitalize.o  myLibrary  scope  scope.c  static  static.c
util.c
student@COMPBase:~$
```

Now all we need to do is to link the code with the library in order to create the executable. To do this, we also need to specify the path to the library using `-L`. Also, we will use `-lnames` to indicate the library name. Notice that we do NOT say **`-llibnames.a`** since the "**lib**" and "**.a**" portions of the filename are implied:

```
student@COMPBase:~$ ls
capitalize.c  myLibrary  scope  scope.c  static  static.c  util.c
student@COMPBase:~$ gcc -o capitalize capitalize.o -L ./myLibrary/lib/ -lnames
student@COMPBase:~$ ls
capitalize    capitalize.o  scope    static    util.c
capitalize.c  myLibrary     scope.c  static.c
student@COMPBase:~$
```

We can then run the code as a normal program:

```
student@COMPBase:~$ ./capitalize

Enter a name: MARK LANTHIER
My name is Mark Lanthier

Enter a name: mark lanthier
My name is Mark Lanthier

Enter a name: mARK lANTHIER
My name is Mark Lanthier

Enter a name: -1 -1
student@COMPBase:~$
```