Linux Users' Group of Davis
C Programming Basics, Part II

Speaker: Mark K. Kim

December 6, 1999

Contents

List of Tables

1 Recap

We learned to write simple C programs in our last discussion. More specifically, we learned to:

- compile C programs,
- print text and numbers to the screen,
- read numbers from the keyboard,
- repeat a block of codes many times,
  (using the `while` keyword,)
- branch off to different parts of the program depending on the program state,
  (using the `if` keyword,)
- and write customized functions.

In addition, we also learned to use various C functions.

Now, we're ready to learn various features of C (a.k.a. the "details.") We'll also learn to use several useful C functions not mentioned in our last discussion.

Data type	Range	Description
`void`	N/A	"No type" type
`char`	-2^7 to 2^7-1	Character
`int`	-2^{31} to $2^{31}-1$	Integer
`float`	$\approx$ 6 digits of precision	Real number (decimal point numbers)
`short`	-2^{15} to $2^{15}-1$	Integer
`long`	-2^{31} to $2^{31}-1$	Integer
`double`	$\approx$ 10 digits of precision	Real number

Table 1: Commonly used data types and their sizes in memory.

Qualifier	Description
`unsigned`	Doubles the capacity of integers and real numbers, but you lose the ability to store negative numbers.
`signed`	Undoes the effect of `unsigned`.
`const`	Prevents the variable from changing.

Table 2: Simple qualifiers.

2 Variables—Places to store values

We often need to store things in memory; perhaps a string of text or numbers.

In C, you generally do not access the memory directly. Instead you tell C what you want to store in memory, and what you want to call that part of the memory. Then you can access the memory using that name.

These names you use to access the memory are called *variables*. *What* you want to store in the memory—text or number or whatever—are called *data types*.

variables
data types

2.1 Characters, integers, and real numbers—and many more

You can store various types of variables in memory. The most common types of data types are listed in table 1.

In addition to the data types, you can also add *qualifiers* to slightly modify the behavior of the data types. Some of the simple qualifiers are listed in table 2.

qualifiers

2.2 Creating many, many variables at once—arrays

Sometimes you want to create many variables at once. Like when you create a database. Or if you are dealing with a vector of numbers or something.

In those cases, it's tedious to create many variables one by one:

```
int age_of_lugod_member1;
int age_of_lugod_member2;
int age_of_lugod_member3;
  ⋮
int age_of_lugod_memberN;
```

That's a lot of work!

Instead, C can create many variables for you. It's called an *array* and it looks like: *array*

```
int age_of_lugod_member[N];
```

Then you can read from or write to any of the *elements* of the array by changing the *index* number inside the brackets, like this: *elements* *index*

```
age_of_lugod_member[0] = 18;
age_of_lugod_member[1] = 21;
age_of_lugod_member[2] = 19;
 ⋮
age_of_lugod_member[N-1] = 20;
```

Notice the first index starts at 0 and the last index is one less than the size of the array.

The arrays are most useful when you are trying to initialize the entire element or print the entire element. For example,

```
int i = 0;

/* Set everyone's default age to 20 */
while(i < N) {
   age_of_lugod_member[i] = 20;
   i = i + 1;
}

/* I know I'm 21 */
age_of_lugod_member[7] = 21;

/* Print everyone's ages */
i = 0;
while(i < N) {
   printf("Member %i is %i years old.\n", i, age_of_lugod_member[i]);
   i = i + 1;
}
```

You do have to be careful when you use arrays. Because C trusts the programmer to know what s/he is doing, it lets the programmer access elements beyond the maximum array length; the effect is usually either corrupted memory or program crash (ever heard of "general protection fault?") But this technique can be used to your advantage if you are careful at using it.

2.3 String of text

If you have an array of characters—it's just like having a string of text. In C, you treat character arrays as strings of text (strings of text are often referred to as *strings*.) *strings*

So, you can do:

```
char s[80] = "Hello, LUGOD!";
printf("%s\n", s);
```

(%s can be used to print strings with printf.)

Notice a few things:

- Because strings are actually arrays, you generally cannot copy strings by using the = operator, and you cannot compare two strings by using the == (or any other test-condition) operator. Instead, you usually have to copy them character-by-character (and there are bunch of functions you can use for that—we'll look at those later.)

  There is one exception, however—at the very beginning, when you create the variable, like we did above.

- With the above program, the string s cannot be more than 80 characters in length. If you do, you'll corrupt the memory, and possibly crash the program.

2.4 Pointers

Pointers are a little difficult to explain so I'd like to use an analogy.

Let's say you and your friend are sitting on the couch, watching the television. Then your friend asks you where the remote control is. You tell your friend where it is by pointing to it. Your friend stands up, walks up to the remote control, grabs it, then comes back to the couch. She/he then changes the channel on the television.

Now, you could have fetched the remote control for your friend but you didn't. It's so much easier for you to point than to actually fetch it. If there were no such thing as "pointing," you could probably live without it, but you wouldn't want to because it makes your life so much easier.

The same applies to C. The pointer in C exists to point to other variables. And then it allows you to access other variables through a single pointer.

Pointers are not necessary to write programs, but they makes your life easier from time to time....

Anyway, the pointer uses two operators—the *pointer to* operator (*), and the *address of* operator (&). Here is an example program:

pointer to
address of

```
int i;   /* Integer */
int* ip; /* Integer pointer */

i = 5;   /* Assign value to i */
ip = &i; /* Point to i */

printf("ip points to %i\n", *ip);
```

First, you create an integer pointer ip (int* is equivalent to 'pointer-to-int.') It can point to any integer variable.

Then ip needs something to point to, like the memory address of a integer variable.

So you find out the address of the i variable using the address-of operator (&) and assign that address to ip. Now ip contains the memory address of i.

Now, to access the variable i via ip, like printing the integer ip points to, you need to find out the value of the integer it points to. So you use the *pointer-to* operator (use it like *ip) to find out what ip points to, then you can print that, assign new values to it, read it, do whatever.

You can also have pointer to functions. We won't go into that topic since it's an advanced topic.

2.5 Mixing pointers and arrays

You can also mix pointers with arrays. You can do this by pointing to the beginning of the array, then accessing the array via the pointer.

With this technique, you can access any portion of the array.

To make a pointer point to the beginning of the array, you do something like:

```
int iarray[10];  /* Array */
int* ip;         /* Pointer */

ip = iarray;     /* Point to the array */
```

Then to access different parts of the array, type:

```
*ip = 0;     /* same as iarray[0] = 0 */
*(ip+1) = 1; /* same as iarray[1] = 1 */
*(ip+2) = 2; /* same as iarray[2] = 2 */
  ⋮
```

3 Operators

C has several types of operators. There are the ones used to do math operations (`+`, `-`, `*`, `/`, `%`,) then there are the ones used to do test conditions (`==`, `!=`, `<`, `<=`, `>`, `>=`,) both of which we've already looked at during our last talk.

Then there are what's known as the *logical operators*, that are used for compound test conditions (we've looked at a couple of them already—`&&` and `||`); and there is also what's known as the *bitwise operators*, that are used to manipulate the memory bit-by-bit.

logical operators

bitwise operators

We'll look at these two different operators. We will also look at some shortcut operators.

3.1 Logical operators

Logical operators are used in compound test conditions. We've already looked at `&&` and `||` operators. We'll now look at how logical operation is done in C, then look at the `!` operator

3.1.1 How logical operations work

When we perform test conditions like "`flag == 1`," we produce a result value. The resulting value is an integer—1 if the test condition is true (if `flag` is indeed 1,) or 0 if the test condition is false (if `flag` is not 1.)

So you can do things like this:

```
int result;
result = (0 == 2);   /* result = 0 */
result = (1 == 2);   /* result = 0 */
result = (2 == 2);   /* result = 1 */
result = (3 == 2);   /* result = 0 */
  ⋮
```

So if you really wanted to, you can do things like:

```
int flag = (2 == 2);
if(flag) {
   printf("This always gets printed.\n");
}
if(1) {
   printf("Me too!\n");
}
if(0) {
   printf("But not me :(\n");
}
```

In addition, C treats any number that *isn't* 0 to be true. So you can use any number in place of 1:

```
int flag = -3;
if(flag) {
   printf("This always gets printed.\n");
}
if(10) {
   printf("Me too!\n");
}
if(0) {
   printf("But not me :(\n");
}
```

But notice you can't do things like:

```
int test1 = (2 == 2);
int test2 = 3;
if(test1 == test2) {
   printf("This doesn't get printed.\n");
}
```

Although you'll probably never write any code like that, it's been known to happen in a complex program.

So if you ever want to compare two test conditions, you have to use something other than test conditions to test the test conditions. You use logical operators.

3.1.2 The AND and OR operators

The logical operators are called that because it fits our logical concepts. Let's use an analogy that uses logical concepts to illustrate.

Say I own one dog but no cat. If a guy named Linus says "Mark owns one dog *and* one cat" he would be telling a *lie* because only the first part of the statement is true (I own one dog) but not the first part *and* the second part (I don't own one cat.) But if he says "Mark owns one dog *and* zero cats" then he would be telling the *truth* because both the first part of the statement (I own a dog) *and* the second part (I own zero cats) are true. He can also say "Mark owns zero dogs and one cat" and "Mark owns zero dogs and zero cats" and the result would be both false because for "and," both the first part and the second part of the statement must be true.

	Mark has no dog... **(false statement)**	**Mark has one dog...** **(true statement)**
...and one cat **(false statement)**	false	false
...and no cat **(true statement)**	false	true

	Mark has no dog... **(false statement)**	**Mark has one dog...** **(true statement)**
...or one cat **(false statement)**	false	true
...or no cat **(true statement)**	true	true

Table 3: AND and OR truth tables for the dog/cat analogy.

`&&`	**0**	**1**
0	0	0
1	0	1

`||`	**0**	**1**
0	0	1
1	1	1

Table 4: AND and OR truth tables for C

You can also ask the same questions for "or": If Linus says "Mark owns one dog *or* one cat," he'd be telling the *truth* because for "or," only one of the statements need to be true (that I own a dog.) You can ask three more questions and you'd end up with two true's and one false.

You can make a table out of this. It's called the *truth table* and it looks like table 3 *truth table*

Using 1's and 0's (as truths are represented in C,) you get the table that looks like table 4 Note that AND operator results in true if and only if both statements are true; and the OR operator results in true if either statements are true.

So you can use the table to look-up what the result of a `&&` or `||` operation is.

3.1.3 The NOT operator

You can also negate the results of a logical operation. For example, "not true" is false, and "not false" is true.

The operator that presents the concept of "not" logic is the `!` operator. Here's an example:

```
int flag = 0;  /* flag is false */
if(!flag) {
   printf("This always gets printed!\n");
}
```

3.2 The bitwise operators

Bitwise operators are similar to logical operators—it has the AND operator (`&`,) the OR operator (`|`,) the NOT operator (`~`,) and a few more operators that have no logical operator equivalents.

The difference is that the truth-table operations are done bit-by-bit.

Say you have an integer 12 (1100 in binary[1]) and you AND it with 10 (1010 in binary.) The result is 8 (1000 in binary) because the 8's digits in 12 and 10 are both 1's (remember true *and* true is true for the *and* operation—see table 4); the 4's digits in 12 and 10 are 1 and 0, respectively (1 *and* 0 is 0;) the 2's digits in 12 and 10 are 0 and 1, respectively (0 *and* 1 is 0;) and the 1's digits in 12 and 10 are 0 and 0, respectively (0 *and* 0 is 0.)

If you summarize this to a math-arithmetic-looking form, it looks like this:

```
     1100
&    1010
---------
     1000
```

Similarly, with the OR operator:

```
     1100
|    1010
---------
     1110
```

And the one's-complement operator (I previously called this the "bitwise not operator"):

```
~    ...00001010
----------------
     ...11110101
```

(Be careful since it negates all the preceding zeros.)

Your scientific calculator should have the bitwise AND, OR, and NOT operator keys.

3.2.1 Shift operators

In addition to the above bitwise operators, there are also shift operators that "shift" the bits either to the left or right. The result is equivalent to multiplying or dividing the number by 2 with a few exceptions.

To shift bits to the left, you use the `<<` operator; For example, to shift the number '12' 3 bits to the left, you type `12 << 3`. The result is 96, equivalent to 12×2^3.

```
         1100
<<          3
-------------
      1100000
```

If the bits shift too far to the left, you will lose any bits that goes over the maximum limit. For example, integers are 32-bits on Linux systems, so you can only have up to 32-bits using integers; beyond that, you will lose the bits.

To shift bits to the right, you use the `>>` operator. If the bits shift too far to the right, you'll also lose those bits as well.

By the way, because the left-most bit represents the negative-ness of the number, you'll get weird numbers if you left-shift 1's into the left-most bit. You'll also get an unpredictable result if you right-shift a negative number because the way negative numbers shift to the right is undefined by the ANSI C standard. The best way to shift numbers, therefore, is to use positive numbers only by making use of the `unsigned` keyword.

[1]If you don't know what a binary number is, you probably won't use bitwise operators. But you can easily convert to/from binary numbers using your scientific calculator—punch in your number in "normal" decimal integer, then press the "BIN" key (3rd-$\times$ on my TI-36X.) To convert back, press the "DEC" key (3rd-EE on my TI-36X.)

3.3 Shortcut operators

Because programmers are such lazy group of people, we have shortcut operators for the most commonly done tasks.

3.3.1 Increment/decrement operators

To increment or decrement a variable by one, you use the `++` and `--` operator, respectively:

```
int c = 1;

++c;  /* Now 'c' is 2 */
--c;  /* Now 'c' is back to 1 */
c++;  /* Now 'c' is 2 again */
c--;  /* Now 'c' is 1 again */
```

When you place `++` *before* the name of a variable (this operator is called the *pre-increment operator*,) the value increments; and then the whole expression becomes the incremented value. When you place `++` *after* the name of a variable (this operator is called the *post-increment operator*,) the expression becomes that value then increment after. Compare the differences:

pre-increment operator

post-increment operator

```
int c=1, d=1;
int a, b;

a = ++c;  /* c=2, a=2 */
b = d++;  /* d=2, b=1 */
```

The same idea applies to the decrement operator.

3.3.2 Self modifying operators

In C, a code like

```
int c = 1;

c = c + 5;
```

can be shortened to

```
int c = 1;

c += 5;
```

You can also use the same idea for the following operators: `-`, `*`, `/`, `%`, `<<`, `>>`, `&`, `|`. (You can also use it for the `^` operator, which we didn't talk about and won't talk about because it's not used very much.)

4 More branch statements (alternatives to `if`)

When you want to *branch* to different parts of the program, you can use the `if` statement. *branch*

But there is another statement that allows you to branch. It's the `switch` statement. *switch*

It's just like the `if` statements except the syntax is different. `switch` can't do everything `if` can (if you ever get stuck using switch, you'll know what I mean,) but you can use `switch` to make your code look better.

The syntax is:

```
switch(value-or-test-condition) {
 case case1:
   statements
   break;
 case case2:
   statements
   break;
 ⋮
 case caseN:
   statements
   break;
 default: /* default action */
   statements
}
```

The *`value-or-test-condition`* is compared with each *`caseN`*, then the statements are executed if they are equivalent.

You can use `switch` to make your program look more organized. Also, sometimes the compiler can optimize `switch` statements better than it can optimize `if`.

5 More loops (alternatives to `while`)

There are also a few alternatives to the `while` statement. The two are `do` and `for` statements. *do* *for*

5.1 `do`

The `do` loop does exactly what the `while` loop does, except it runs the code inside the loop once before checking the test condition.

Its syntax looks like:

```
do {
   statements
} while(test-condition);
```

We won't go into the details since I know of very few program that use the `do` loop. But *I* like it `:)`

5.2 `for`

The `for` loop is designed to simplify `while` loop that looks like this:

```
statement1
while(test-condition) {
   statements
```

```
    statement2
}
```

into this:

```
for(statement1; test-condition; statement2) {
    statements
}
```

It may seem redundant to have the `for` loop, but it simplifies things. Think of it as a shortcut.

The most common `for` loop looks like:

```
for(i = 0; i < some-number; i++) {
    statements
}
```

which makes things look nice because all the terms involving `i` is on the first line of the loop.

6 Grouping variables—`structs`

Sometimes it's good to group several variables together to give an appearance of "togetherness." Like if you want to make a database of LUGOD members, you'd want to group all information for a single person together.

The keyword used for this purpose is `struct`. First, you tell C what the grouping looks like (you place this code before `main`):

```
struct Group-name {
    variable-type1 variable-name1
    variable-type2 variable-name2
    ⋮
    variable-typeN variable-nameN
};
```

and then you can create new variables based on this structure like this:

```
struct Group-name variable-name;
```

Member variables are accessed via the *member-of* operator, the period (`.`). If you have a pointer to the structure, you can use the operator `->` *member-of*

Please refer to `struct.c` for an example.

7 Reading options and arguments

You sometimes give arguments to a program. Like the *-a* option you give the `ls` program.

Your C program can read these options and arguments. To do that, you modify `main` to look like this:

```
int main(int argc, char* argv[]) {
   ...
}
```

When your program runs, `argc` contains the number of arguments on the command line; and `argv`, a string array, contains the argument.

For example, if you run your program like this:

```
./myprog -la
```

then `argc` is 2, `argv[0]` is "./myprog", and `argv[1]` is "-la". In addition, `argv[2]` becomes `NULL` (`argv[2] == NULL` is true.)[2] *NULL*

On GNU C, there is a command `getopt` that parses the command line for you so you don't have to worry about parsing it yourself. Please read the man page for further information.

8 Interfacing with shell scripts

The `main` function also `return` an integer value to the operating system (i.e. `return 0;`). The same is true when you use the `exit` function to exit the program (i.e. `exit(1);`).

You can read this value, called exit code, from shell scripts to do different things. Please refer to Peter's shell script talk notes for further information.

A note of caution: Most shell scripts assume 0 to be a normal exit code, and other values to be some kind of error. Programs like `make` stops processing when one of the programs it runs returns a non-zero code. So be careful when you exit with non-standard exit codes.

9 Using headers and libraries

There are various headers and libraries you can use with C. There are several standards:

- ANSI/ISO C standard
- POSIX standard

  The standard used to program for UNIX-like operating systems.
- GNU extensions

  GNU includes several extensions to make open-source and cross-platform programming easier. The `getopt` function, for example.
- BSD UNIX extensions
- OpenGroup UNIX extensions
- platform-based standard

  Platforms like DOS includes `dos.h` header file that contains all DOS-based extensions. X Window includes `xlib.h` for X Window programming.

[2] `NULL` is a predefined pointer you can use to indicate that a pointer points nowhere. Usually, it points to memory location `0` so you can do an operation like `if(pointer)` to determine whether a pointer points anywhere valid (somewhere other than 0.)

Due to time constrains, we'll only briefly discuss some of the ANSI/ISO C standard functions.[3]

There are only several commonly-used functions listed here, with just the names and brief descriptions. Please refer to the man page, section 2 or 3 (depends on the usage; check both), for the details.

9.1 Functions for I/O

The I/O functions can be used to print to the screen, read from the keyboard, or print/read to/from a file.

Here are some functions you can use to read from / write to the console:

Console I/O (`stdio.h`)	
`printf`	Used to print to the screen
`scanf`	Used to read from the keyboard

To read from / write to a file, you must first open the file, then close the file after you're done using it.

While you're accessing the file, you don't access the file via its filename. But rather, you access the file via a pointer to a `struct` named FILE, that contains various necessary information about the file. (The pointer to `FILE`, `FILE*` is called the *file handle* because you use the handle to "hold onto" the file while it's open.) *file handle*

Here are the commonly used functions used for file I/O:

File I/O (`stdio.h`)	
`fopen`	Used to open a file for reading/writing
`fclose`	Used to close a file after done reading/writing
`fprintf`	Used to print to a file
`fscanf`	Used to read from a file
`feof`	Used to check whether you've reached the end of file

Because UNIX treats everything as files (including the screen and the keyboard,) you can use the file I/O functions also as screen/keyboard I/O functions. To use file I/O functions with the screen/keyboard, use the file handle `stdin` when you want to read from the keyboard, and use the file handle `stdout` when you want to write to the screen. You can also use the file handle `stderr` if you wish to write to the standard output (also the screen by default.) Please refer to notes on the shell talk for details on how one can use standard error. You need not close these handles because C does it automatically.

9.2 String functions

In C, all string access must be done character-by-character. C makes that chore a little easier by providing several built-in functions:

[3] Most of the information here was carefully extracted from *The C Programming Language* by Brian W. Kernighan and Dennis M. Ritchie, second edition, published by Prentice Hall. ISBN 0-13-110362-8. It is consider to be the bible of the C language and I highly recommend it. You can purchase a copy at the UC Davis bookstore.

String operations (`string.h`)	
`strcpy`	Used to copy strings
`strcat`	Used to append one string to another
`strncpy`	Used to copy strings up to N characters
`strncat`	Used to append strings up to N characters
`strlen`	Returns the length of the string
`strcmp`	Compares two strings
`strchr`	Finds a character in a string
`strstr`	Finds a string within a string

9.3 Math functions

C also provides several basic math functions. In addition to including `math.h`, you also must use the *-lm* option when you compile your program.

All trigonometric functions use the radian coordinate system.

Math functions (`math.h`)	
`sin`	Used to find the sine of an angle
`cos`	Used to find the cosine of an angle
`tan`	Used to find the tangent of an angle
`asin`	Used to find the inverse sine of a real number
`acos`	Used to find the inverse cosine of a real number
`atan`	Used to find the inverse tangent of a real number
`atan2`	Used to find the inverse tangent of a real number, compensating for the 2nd and 3rd quadrants
`pow`	Used to find the power of some number
`exp`	Used to find e^n
`log`	Used to find the natural log (ln)
`log10`	Used to find $\log_{10}$
`abs`	Takes the absolute value of an integer (This function is in `stdlib.h`)
`labs`	Takes the absolute value of a long integer (This function is in `stdlib.h`)
`fabs`	Takes the absolute value of a real number
`ceil`	Rounds up a real number
`floor`	Rounds down a real number

You can round numbers by adding .5 to the real number, then taking its `floor`.

9.4 Dynamic memory allocation functions

In C, the size of all arrays must be know before you compile the program. If you do not know the size of the array before you compile it, you must dynamically create memory at runtime, then use a pointer to read from and write to the memory.

Here are functions that allows you to do that:

Dynamic memory functions (`stdlib.h`)	
`malloc`	Create new memory of size *N* and returns its address
`realloc`	Resizes an existing memory, either by extending the memory (if possible) or by creating new memory as necessary
`free`	Destroys the memory so you can use it for something better

`malloc` and `realloc` will return `NULL` if you run out of memory, so you should check to make sure these functions return `NULL` or not. Otherwise you could either corrupt memory or crash your program.

9.5 Conversion functions

You can convert between strings to numbers, numbers to strings, a number to a character, a character to a number, etc. Here are the functions to let you do that:

Conversion functions (`stdlib.h`)	
`atoi`	Convert from string to integer
`atof`	Convert from string to real number
`atol`	Convert from string to long
`sprintf`	Convert from integers or real numbers to string (This function is in `stdio.h`)

9.6 Randomization functions

You can produce a series of random-ish numbers in C. Here are the functions to do that:

Randomizing functions (`stdlib.h`)	
`srand`	You must give the randomization function a "seed" with this function. Otherwise you'll get the same series of number every time you use the `rand` function (this is useful if you're running a simulation and want to be able to reproduce the simulation.) Most people use the computer's internal clock to randomize the the seed.
`rand`	This function returns a random-ish integer, between 0 and `RAND_MAX`.

9.7 Date and time functions

Keeping track of the time is useful when you want to slow down your program so it does not run too fast, or if you wish to keep track of how often your program runs.

Here are some functions to help you do this:

Date/time functions (`time.h`)	
`time`	Returns a `struct` containing current date/time; for details on the structure `struct`, please read the section 2 manpage
`difftime`	Returns the difference of two times
`asctime`	Prints the date/time

9.8 Miscellaneous functions

Here are some miscellaneous functions:

Miscellaneous functions (`stdlib.h`)	
`system`	Executes an external program
`qsort`	Built-in sorting function
`bsearch`	Built-in search-tree function

A Using `printf`-family functions

When you use `printf`, `fprintf`, and/or `sprintf`, you can use `%i` to print integers, `%f` to print floats, etc. You can also type `\n` to print the newline character. We'll explore what other symbols you can use to print other variables and symbols.

Here are the symbols used to print variables:

Symbols to print variables for `printf` family functions	
`%c`	Character
`%s`	String
`%d`	Integer (more common)
`%i`	Integer (less common)
`%f`	Real
`%o`	Octal integer
`%x`	Hexadecimal integer

You can also use the above symbols with `scanf`-family functions to read values into variables. Because `scanf` functions need to write variables, it needs to know the address of the variables it writes to; therefore you must use the address-of operator when you use `scanf`, like this:

```
int i;
scanf("%d", &i);
```

The only exceptions are the pointers and the arrays—they are variables that hold addresses to the memory area that `scanf` writes to, so `scanf` does not need to know the address of the pointers or arrays. That means strings, which are arrays, don't need the ampersand.

You can add modifiers to the above symbols to control how you want the characters to be printed.

A common one is placing a number between `%` and the letter to force the output to a certain width. For example:

```
printf("4-digit hex number: %04x\n");
```

The '0' in front of '4' uses 0 as the number padding.

There are numerous other modifiers that cannot be listed all here. Please refer to *The C Programming Language* for details.

There are also various weird characters you can work with. They are not for `printf` functions but for any string in C in general:

Printable symbols	
`\n`	Newline character (to next line)
`\r`	Carriage return character (to beginning of the same line)
`\t`	Tab character
`\b`	Backspace character (back one character w/o erasing character)
`\a`	Alarm (print it to get a beep)
`\\`	Backslash character
`%%`	Percent character (works only with `printf`-family of functions, and it's only necessary with `printf`-family of functions)

B Representing numbers

You can represent different types of numbers in C. There are decimal integers, reals (floats), octals, and hexadecimals.

You already know how to read and write decimal integers—0, 1, 2, 3,.... And you may already know how to read and write floating numbers (if you know the e-notation.)

These two numbers exist in C because those are the numbers we use in everyday life.

But...computers don't use decimal (base 10) numbers. They use binary (base 2) numbers.

It would be useful to write numbers in binary, though, because it helps us *visualize* what computer "sees," the way computer stores the numbers in its memory.

But large binary numbers are too long to write. So computer scientists created octal (base 8) and hexadecimal (base 16) numbers, that are multiple bases of the binary, to make it easier for programmers to write numbers that represent numbers inside the computer's mind (memory.)

B.1 Floating numbers

In computers, there is something we call the *e-notation*. This is a technique used to represent numbers in the form $a \times 10^b$. *e-notation*

Since printing out `a * 10^b` (`^` is a common notation used to represent "power") or `a * 10**b` (`**` is a notation used to represent "power" in FORTRAN) wastes too much screen space, computer scientists made a shorter notation to represent $a \times 10^b$, and it looks like this:

```
aeb
```

It is very easy to confuse this with e^b. Don't `:)`

So to represent 6.02×10^{23} in C, type `6.02e23`. Compact, saves memory, everyone's happy.

B.2 Octal numbers

Base 8 numbers (called *octal* numbers because the prefix "oct" means "eight," just like the prefix "dec" in "decimal" means "ten.") use only 8 of the 10 numbers in our decimal system. These numbers are 0, 1, 2, 3, 4, 5, 6, and 7. *octal*

Octal numbers are not used very often in C, but it *is* often used with the UNIX program `chmod`. So it is good to know octal numbers. Please read the `chmod` manual for the details on the usage of `chmod`.

Due to time constrains, we will not go into the details of converting between decimal (standard) numbers to octal numbers, but you can use a scientific calculator to convert between decimal and octal numbers.[4] To convert from a decimal number to a octal number, type the number you wish to convert to the octal number, then press the `OCT` button (it's `3rd-)` on my TI-36X calculator). To convert from a decimal number to a octal number, put the calculator in octal mode by pressing the `OCT` button, typing the octal number you wish to convert to decimal, then pressing the `DEC` button (it's `3rd-EE` on my TI-36X.)

Since we use the octal numbers to make it easier to visualize binary numbers, it helps to have a octal-to-binary conversion table, like table 5.

[4]Note: Not all scientific calculators have base conversion keys.

Octal	Binary
0	000
1	001
2	010
3	011
4	100
5	101
6	110
7	111

Table 5: Octal-to-binary conversion table

To represent octal numbers in C, you must precede the number by `0`. So, to write `21` in octal (equivalent to 17 in decimal,) type `021` in C.

B.3 Hexadecimal numbers

Hexadecimal numbers are base 16 numbers that utilizes all 10 numbers in the decimal system, plus 6 more from the alphabet to make up the 6 numbers (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, and F—16 "numbers!")

To represent hexadecimal numbers in C, you precede the number by `0x`. Some examples of hexadecimal numbers are `0x10` (16 in decimal,) `0x2A` (42 in decimal,) `0xFACE` (64206 in decimal,) and `0xDEAD` (57005 in decimal.)

You can also use your calculator to convert between decimal and hexadecimal numbers. Your scientific calculator should have a button labeled `HEX` (`3rd-(` on my calculator.) To type hexadecimal numbers, you can use the keys labeled `A-F` (`3rd` and one of `1/x`, x^2, $\sqrt{x}$, `SIN`, `COS`, `TAN` keys on my calculator.)

A hexadecimal-to-binary table is given in table 6.

Hexadecimal	**Binary**	**Decimal**
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
A	1010	10
B	1011	11
C	1100	12
D	1101	13
E	1110	14
F	1111	15

Table 6: Hexadecimal-to-binary conversion table