Chapter 3
Variables, Data Types, & Arithmetic Expressions

3.1 Chapter Overview

This chapter introduces variables and data types. Also, arithmetic operations are demonstrated, including the modulus operator, as well as the type cast operator, assignment operator, sizeof operator, and the increment and decrement operators. Also the functions `printf()` and `fprintf()`, and also `scanf()` and `fscanf()` functions are discussed. Different numbering systems, and ASCII code are explained. Lastly, stdin, stdout, stderr are discussed as well.

3.2 Data Types

C provides the following data types: `int`, `float`, `double`, `char`, and `_Bool`. These types indicate what kind of information is being represented.

1. `int` is for integral numbers, or integers - - numbers without decimal places
   - no spaces or commas allowed
   - -10 0 158 are all `ints`.
   - use `%i` or `%d` for the format string used with printf and scanf
   - **octal** numbers are represented with a leading 0, so 052 is octal 52 which equals 42 in decimal (use `%o` for the format string, or `%#o` to print the leading 0)
   - **hexadecimal** numbers are represented with a leading 0x, so 0x02A is hex 02A, which equals 42 in decimal (and 52 in octal) (the format string is `%x` or `%#x` to display the value with the leading 0x; if you use uppercase X, such as `%X` or `%#X`, the hexadecimal digits that follow will be uppercase letters)
   - Binary, octal, and hexadecimal numbers are discussed in Section 3.6 on page 8 of these notes.

2. `float` is for storing floating-point numbers, i.e. values containing decimal places
   - 3.0 125.8 -.001 are all floats
   - use `%f` with printf and scanf
   - **scientific notation** 1.7e4 is a floating point value = 1.7×10^{-4} (use `%e` with printf; `%g` lets the system choose which format is best to use)

3. `double` is the same as float but with roughly twice the precision; all float constants are taken as doubles by the C compiler (use `%f`, `%e`, or `%g` – same as float – for printf)

4. `char` is used to store a single character (a character can be a letter, a digit, or a type of punctuation such as a comma)
   - the character is enclosed in single quotes, such as `'a'`
   - `'\n'` is the newline character; it is treated as a single character (other special characters in Appendix A, pages 430-431)
   - use `%c` with printf
   - … more about how characters are represented in Section 3.7 of these notes.

5. `_Bool` is used just to store the values 1 or 0
   - used to indicate an on/off, yes/no, or true/false situation
   - you can use `<stdbool.h>` in your programs, which defines `bool`, `true`, & `false`, to make it easier working with this type (program 5.10A in book has example)

In C, a number, character, or character string that is given or provided is known as a **constant,** or a **literal**. With the following statements,

```
int a = 3;
char charVar = 'W';
printf("Programming is fun!");
```

the `3` the `W` are *constants*, and the string `Programming is fun!` is a *constant character string*, or *string literal*.

Expressions consisting entirely of constant values are called **constant expressions**.

`128 + 7 - 4` is a *constant expression*

`128 + 7 - y` is not

Table 3.2 on pages 28-29 summarizes the data types, showing constant examples for each, and their corresponding formatting strings.

The number of bits that a data type uses depends on the system. For example, integers are guaranteed to be at least 32 bits big, but sometimes may be 64 bits big. On our system, they take up 32 bits.

There are also optional ***modifiers*** that can be used with integers: `long`, `long long`, `short`, `unsigned`, and `signed`, explained in Chapter 3. The modifier `long` can also be used with doubles.

Type	Size
`short`	16 bits (guaranteed to be at least 16 bits)
`int`	32 bits (guaranteed to be at least 32 bits; sometimes they are 64 bits)
`long int`	64 bits
`long long int`	64 bits
`float`	32 bits
`double`	usually 64 bits (64 on our system)
`long double`	128 bits (which is 16 bytes)
`char`	8 bits
`_Bool / bool`	8 bits

An integer, by default, is a signed value. For an integer to be a signed value, you can declare it as `signed int` or just simply as `int`.

Every value, whether it's a character, integer, or float, has a range of values associated with it. For example, a signed integer that takes up 32 bits allows for values that fall in the range of -2,147,483,648 to +2,147, 483, 647.

If you want an unsigned integer, you would declare it as `unsigned int` which shifts the range to positive values from 0 to +4,294,967,295.

3.3 Variables

We have seen in our first program how to display a message on the screen. Clearly, we can write more complex programs. To be able to do that, we need variables to do calculations and save information. ***Variables are symbolic names*** *for storing program computations and results; symbolic names* ***referring to a memory location*** *where values are stored.*

Rules for forming variable names are explained on page 29 in chapter 3. Variable names have to start with an upper or lower case letter or an underscore (_), then can be followed by upper or lower case letters or underscores (_) or digits (0 – 9). Variables must be declared before they are used in a program. Usually, all the variables in a function are declared in a group at the top of the function. Also, variable names are **case sensitive**, so `SUM` is not the same as `sum`. It is common to capitalize variable names for constants.

To declare a variable you need to indicate its type, and sometimes an initial value, such as:

```
int sum1, sum2;
float _average = 0.0;
bool true;
const double PI = 3.141592654;
```

Variable names **cannot be reserved words** (Table 1.2.2 page 428 in Appendix A). Reserved words, or keywords, are words that have special meaning to the compiler. Variable names also cannot be predefined function names either. For example, `int`, `char`, `main`, and `printf` are all invalid variable names . Other examples of invalid variable names are shown at the bottom of page 29 in chapter 3.

When deciding on variable names, try to pick **meaningful names** that reflect the intended use of the variable. This helps to dramatically increase the readability of the program; and also helps with debugging and documenting your code.

Lastly, variable names that are comprised of two or more words are usually in one of two forms: either separated with an underscore, or uses "camel case". For example: `last_name` or `lastName` would both be common ways of forming that variable name.

3.4 Arithmetic Expressions and Other Operators

We can perform different **arithmetic operations** on variables and/or constants. `+ - * / %` represent addition, subtraction, multiplication, division, and modulus. The modulus operator (`%`), or modulo, returns the remainder of dividing two numbers (e.g. 7 % 3 = 1, 10 % 5 = 0). All of these operations are **binary operations** (i.e., they take two operands). A statement involving an arithmetic operation can be the following: `a + 5` or `a * b` or `a / d`.

You can include arithmetic operations with the **assignment operater** (=) (e.g., `a = b + c;`). This adds `b` and `c` and stores the sum in variable `a`. You can also do the following `a = a * b;`. This multiplies the original value of `a` by `b` and stores the result back in `a`, thus updating the variable `a`.

You can use `+ - * / %` as **unary operators**. For example, `a += 2;` adds two to the value of `a` and stores the new value in `a`. Accordingly, this is equivalent to `a = a + 2;`.

Arithmetic operations have **precedence** that dictates the order in which they are evaluated. For example, `a + b * c` will be evaluated as `(b * c)` then that result will be added to `a`. This is because multiplication has a higher precedence than addition so it is evaluated first. `* / %` and `%` have equal precedence and it is higher than `+` and `-` which have the same precedence. Operators of the same precedence are evaluated from left to right. For example, `a - b + c` will be evaluated as `a - b` then that result will be added to `c`. Table 5.1 in Appendix A, pages 443-444 summarizes various operators in C in order of decreasing precedence.

If we want to evaluate operations of lower precedence first, we put the expression in parentheses. If we have multiple nested parentheses, then the inner most is evaluated first, then the next outer one, and so on. If parentheses have the same nested level, they are evaluated from left to right. Examples:

1. `a * (b + c)` will add `b` and `c` then multiply that result by `a`
2. `a - ((b + c) * d)` will add `b` and `c`, then multiply the result by `d`, then subtract that result from `a`
3. `(a + b) * (c + d)` will add `a` and `b`, then add `c` and `d`, then multiply the two results. Therefore, parentheses have higher precedence than `* / %`.
4. `a /= b + c` will add `b` and `c` first, then `a` will be divided by that sum and the result stored back into `a`. This is because the addition operator has a higher precedence than the assignment operator; in fact all the operators except the comma operator, have a higher precedence than the assignment operator.

Note that with `int` arithmetic, if the result is a decimal value, all decimal portions are lost. For example, `25 / 2` would normally be 12.5, but if you run this in a program using `int`s, it would give you `12`, because it **truncates** the value – it drops everything after the decimal point, even if `%f` is used with printf. However, if either one of the values is a `float`, then the result will be printed as a `float` (using `%f` with printf). So if it was `25.0 / 2` or `25 / 2.0`, you would get `12.500000` (using `%f` with printf).

Whenever a floating-point value is assigned to an integer variable, it is **truncated**, dropping all the numbers after the decimal point.

You can use a **type cast operator** to convert the value of the variable or literal to the type that you cast to it.

1. With the above example, `(float) 25 / 2` would give you the answer of `12.500000` (using `%f` with the printf or storing the answer in a variable of type `float`). Floats, by default, print the decimal values to 6 places. **To limit the decimal places to two places of precision**, you would use `%.2f` with printf; to limit it to three places, use `%.3f` with printf.
2. Another example of truncation: `(int) 29.55 + (int) 21.99` is evaluated in C as `29 + 21`.

What would be the answer to this expression? 11 / 2 / 2.0 / 2 = _____

The **sizeof** operator can be used to show the number of bytes of memory taken up by a data type or a variable. For example, `sizeof(int)` used in a printf statement will return 4 on our system (4 bytes, which equals 32 bits). With this example: `sizeof(sum)` if `sum` was declared as a `double`, 8 would be returned on our system (8 bytes, which equals 64 bits).

Increment and decrement operators:

`result++;` is the same as `result = result + 1;` and also the same as `result += 1;`

`result--;` is the same as `result = result - 1;` and also the same as `result -= 1;`

The increment and decrement operators can be pre-fix or post-fix. That is, either `++result` or `result++`. If used in an expression, there is a difference.

```
int n = 2;
int result;
result = 2 * (n++);
printf("%d\n", result);
printf("%d\n", n);
```

will produce the output:

```
4
3
```

whereas

```
int n = 2;
int result;
result = 2 * (++n);
printf("%d\n", result);
printf("%d\n", n);
```

will produce the output:

```
6
3
```

In the first code example, post-fix increment is used, which means multiply the current value of `n` by `2` first, then increment `n`.

The second code example, which uses pre-fix increment, says to increment `n` first, then multiply that new value of `n` by `2`.

3.5 `printf( )` *and* `scanf( )` *functions*

`printf()`

Terminal output is achieved with the `printf()` function, as we have already seen. The `printf()` function sends output to "standard output", which is normally the screen. The `printf()` function has one or more arguments, as seen in the following examples:

```
printf("\nProgramming is fun. \n");    // has 1 argument, the string in blue
printf ("\nThe sum of 50 and 25 is %i. \n\n", sum);    // has 2 arguments
printf ("\nThe sum of %i and %i is %i. \n\n", value1, 20, value1 + 20);  // 4 args
```

The format strings used in all of the above examples is `%i` which is used for integer values. `%d` can also be used for integer values. The format strings are placeholders for each variable, literal, or expression to be printed. In other words, the format strings that appear in the first argument (the one in quotes) are replaced by the value(s) of the variable(s) or literal(s) or expression(s) that follow the quotes. As was mentioned previously, Table 3.2 on pages 28-29 summarizes the data types, and shows their corresponding formatting strings.

The `printf()` function returns the number of characters that it printed, though we usually don't need or care about that, so without capturing that value into a variable, it is just discarded. If we wanted capture the return value, we could do something like the following:

```
int num1 = 55;
int num2 = 30;
int sum = num1 + num2;
int printCount;
printCount = printf("%d + %d = %d\n", num1, num2, sum);
```

The value of `printCount` after that `printf()` statement executes should be 13.
`55 + 30 = 85` is 12 characters plus the newline character equals 13.

Output Formatting Placeholder

```
%[flags][width][.precision][length]type
```

flags:

- `-` left-justify
- `+` generate a plus sign for positive values
- `#` puts a leading 0 on an octal value and 0x on a hex value
- `0` pad a number with leading zeros

width:

- minimum number of characters to generate
- numbers are padded with leading spaces unless the width has a leading zero flag specifying leading zeros
- if the number is larger than the number of digits specified, the number will be printed with the wider width

.precision: round a float number to the specified number of decimal places

length: a length character of `l` may be specified for printing out long numeric types

Examples:

```
printf("[%5d] [%+05d] [%#5o] [%#7x]\n",
       123, 123, 123, 123);
printf("[%f] [%5.2f] [%5.0f%%]\n",
       123.456, 123.456, 123.456);
```

```
[  123] [+0123] [ 0173] [   0x7b]
[123.456000] [123.46] [  123%]
```

scanf()

Input from the keyboard is achieved with the use of the `scanf()` function. The `scanf()` function captures input from "standard input", which is normally the keyboard, and stores it in the variable specified in the second argument to the function. In the following example, you can see a variable called `input` declared, which will hold the value of an integer that will be entered by the user. Then the user is prompted to enter an integer. Following that is the call to `scanf()` which will capture that input and store it in the variable:

```
int input;
printf("Enter an integer: ");
scanf("%d", &input);
```

A single call to the `scanf()` function can be used to read in more than one value entered by the user. The following shows an example of this, and also shows the use of two different formatting strings for two different data types:

```
int age;
float gpa;
printf("Enter your age (as an integer) followed by \
      your gpa (as a floating-point value):  ");
scanf("%d%f", &age, &gpa);
```

The `scanf()` function returns the number of items read in. Unless the return value is saved into a variable, it is discarded and ignored, as with `printf()`. The following example shows a variable called `count`, which is used to capture the number of items read in from the keyboard.

```
int count;
int age;
float gpa;
printf("Enter your age (as an integer) followed by \
      your gpa (as a floating-point value:");
count = scanf("%d%f", &age, &gpa);

print("You entered %d items at the keyboard:  age of %d, gpa of %f\n",
      count, age, gpa);
```

Format Placeholder for Input

- `%d` – for integers, no octal or hexadecimal
- `%i` – for integers allowing octal and hexadecimal
- `%f` – for float
- `%lf` – for double (the letter l, not the number 1)

Do not specify width, use newline characters, or other special `printf` options.

3.6 Numbering Systems

Decimal and Binary Numbers

In our daily lives we use the decimal number system, where we count from 0 to 9, then we go to the 2-digit number 10, and so on. With decimal numbers, there are 10 number choices for any given digit (0 through 9).

Computers use binary numbers because everything boils down to "switches" in the hardware. The switches are either off or on (i.e., 0 or 1). There are only two number choices allowed in this system: 0 and 1. So, we count from 0 to 1, then we go to the 2-digit number 10, which is different from the decimal number 10 (i.e., 10 in binary is equivalent to 2 in decimal).

Decimal numbers follow base 10, while binary numbers follow base 2.

For example, 101 is one hundred one in decimal which is $(1 * 10^2) + (0 * 10^1) + (1 * 10^0)$.

In binary, this number is $(1 * 2^2) + (0 * 2^1) + (1 * 2^0)$ which is the same as 5 in decimal.

See if you can fill in the answers below:

11101_2 = _____$_{10}$ 1011011_2 = _____$_{10}$ 110111_2 = _____$_{10}$

72_{10} = _____$_2$ 26_{10} = _____$_2$ 47_{10} = _____$_2$

Other Number Systems

Other number systems include octal (base 8), and hexadecimal (base 16). For octal, there are 8 number choices for any given digit: 0 - 7. For hexadecimal numbers, digits go from 0 - 15. But to differentiate between 15 (fifteen) and 15 (one followed by a five), we use 0 to 9, then A, B, C, D, E, F to represent 10, 11, 12, 13, 14, and 15 respectively.

More practice:

101001110000_2 = _____$_8$ $100101100111001001100111_2$ = _____$_8$

101001110000_2 = _____$_{16}$ $110101100111001001100111_2$ = _____$_{16}$

$2AF3_{16}$ = _____$_2$ $B38C6_{16}$ = _____$_2$

72632_8 = _____$_2$

112_8 = _____$_{10}$

112_{16} = _____$_{10}$

00101010111110011_2 = _____$_{10}$

3.7 ASCII Code

The data type for a single character is `char`. Each character is represented in 1 byte (8 bits) by a specific binary value. On most computers today, that value is defined by the **American Standard Code for Information Interchange**, or **ASCII**.

The following ASCII table is from http://www.asciitable.com/. If you look at the chart starting with the lowest characters, you will find numbers, then upper case letters, and then lower case letters. Special characters are sprinkled around these ranges.

The type `char` in C is really a type of integer. For example, if you had the following line of code in C:

```
printf("%c \n", 115);
```

the lower case letter `s` would be printed to the screen because 115 is the decimal value representing that character. 115 corresponds to the binary number `1110011` which, padded out to 8 bits, would be `01110011`. In other words, when we press the `s` key on the keyboard, the processor gets the binary number `01110011` which to us, in decimal, is 115.

See if you can figure out how to print the letter `s` to the screen using the octal and hexadecimal values given in the ASCII table below.

Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040		Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH (start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX (start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX (end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT (end of transmission)	36	24	044	$	$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ (enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK (acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL (bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS (backspace)	40	28	050	(	(	72	48	110	H	H	104	68	150	h	h
9	9	011	TAB (horizontal tab)	41	29	051	)	)	73	49	111	I	I	105	69	151	i	i
10	A	012	LF (NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT (vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF (NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR (carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO (shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI (shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE (data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1 (device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2 (device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3 (device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4 (device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK (negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN (synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB (end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN (cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM (end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB (substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC (escape)	59	3B	073	;	;	91	5B	133	[	[	123	7B	173	{	{
28	1C	034	FS (file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174			|
29	1D	035	GS (group separator)	61	3D	075	=	=	93	5D	135	]	]	125	7D	175	}	}
30	1E	036	RS (record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US (unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177	&#127;	DEL

3.8 stdin, stdout, & stderr

The C system automatically opens up 3 files (that are defined in <stdio.h>) when a program is run:

1. stdin – input from the keyboard
2. stdout – output sent to the terminal window
3. stderr – most error messages produced by the system are sent to an error file, which is also usually associated with the terminal window

stdin

We know that the `scanf()` function gets user input from the keyboard, or **stdin**. The `fscanf()` function is similar to `scanf()` with an additional argument as the first argument which specifies where the input is coming from. If the input is coming from the keyboard, then stdin would be specified as the first argument; if the input is coming from some other file, then that other file will be specified instead.

The following two statements are essentially the same:

```
scanf("%d", &input);
fscanf(stdin, "%d", &input);
```

stdout

We also know that the `printf()` function sends the output automatically to **stdout,** or the terminal window. There is also a function called `fprintf()` which looks pretty similar to `printf()` with an additional argument as the first argument to the function which specifies a file where the output will be sent to. You will see `fprintf()` used frequently with stdout specified as the first argument so that the output will be sent to the screen. If you were going to send output to some other file, you would use `fprintf()` and the first argument would indicate the file that the output is to be sent to.

So the following two statements are essentially the same in that they will send the same message to the terminal window:

```
printf("I love programming!\n");
fprintf(stdout, "I love programming!\n");
```

Output that is sent to the terminal window can be redirected to a file, using the **redirect** operator, which is the "greater than" character. The following command at the Unix command prompt will send all `printf/fprintf to stdout` output to the file called "output.txt":

```
a.out > output.txt
```

stderr

When `fprintf()` uses **stderr** as the first argument, the output will be sent to the terminal window even if a redirect is used. Sometimes, you may want the output sent to a file separated from other output that you want to be shown on the screen. This is common for error statements, or for debugging.

Copy the following program and run it two ways: first by just typing `a.out` (or `./a.out`) and then by typing `a.out > output.txt` and see what happens:

```
#include <stdio.h>

int main(void) {

   printf("\n\n1.  this print statement will show up on the screen \
           unless a redirect is used\n\n");
   fprintf(stdout, "2.  same with this one\n\n");
   fprintf(stderr, "\n\n3.  this one will always go to the screen\n\n");

   return 0;
}
```