# Chapter 2 - Variables and types

## More output

As I mentioned in the last chapter, you can put as many statements as you want in main(). For example, to output more than one line:

#include <stdio.h>

#include <stdlib.h>

/\* main: generate some simple output \*/

int main (void)

{

printf ("Hello World.\n"); /\* output one line \*/

printf ("How are you?\n"); /\* output another line \*/

return (EXIT\_SUCCESS);

}

As you can see, it is legal to put comments at the end of a line, as well as on a line by themselves.

## Values

Computer programs operate on values stored in computer memory. A value is one of the fundamental things that a program manipulates. The only values we have manipulated so far are the strings we have been outputting, like "Hello, world." You (and the compiler) can identify these string values because they are enclosed in quotation marks.

There are different kinds of values, including integers and characters. It is important for the program to know exactly what kind of value is manipulated because not all manipulations will make sense on all values. We therefore distinguish between different types of values.

## Variables

One of the most powerful features of a programming language is the ability to manipulate values through the use of variables. So far the values that we have used in our statements where fixed to what was written in the statement. Now we will use a variable as a named location that stores a value.

Just as there are different types of values (integer, character, etc.), there are different types of variables. When you create a new variable, you have to declare what type it is. For example, the character type in C is called char. The following statement creates a new variable named fred that has type char:

char fred;

This kind of statement is called a declaration.

## Assignment

Now that we have created some variables, we would like to store values in them. We do that with an assignment statement.

first\_letter = 'a'; /\* give first\_letter the value 'a' \*/

hour = 11; /\* assign the value 11 to hour \*/

minute = 59; /\* set minute to 59 \*/

This example shows three assignments, and the comments show three different ways people sometimes talk about assignment statements. The vocabulary can be confusing here, but the idea is straightforward:

* When you declare a variable, you create a named storage location.
* When you make an assignment to a variable, you give it a value.

## Outputting variables

You can output the value of a variable using the same commands we used to output simple values.

int hour, minute;

char colon;

hour = 11;

minute = 59;

colon = ':';

printf ("The current time is ");

printf ("%i", hour);

printf ("%c", colon);

printf ("%i", minute);

printf ("\n");

This program creates two integer variables named hour and minute, and a character variable named colon. It assigns appropriate values to each of the variables and then uses a series of output statements to generate the following:

The current time is 11:59

When we talk about "outputting a variable," we mean outputting the value of the variable. The name of a variable only has significance for the programmer. The compiled program no longer contains a human readable reference to the variable name in your program.

The printf() command is capable of outputting several variables in a single statement. To do this, we need to put placeholders in the so called format string, that indicate the position where the variable value will be put. The variables will be inserted in the order of their appearance in the statement. It is important to observe the right order and type for the variables.

By using a single output statement, we can make the previous program more concise:

int hour, minute;

char colon;

hour = 11;

minute = 59;

colon = ':';

printf ("The current time is %i%c%i\n", hour, colon, minute);

On one line, this program outputs a string, two integers and a character. Very impressive!

## Keywords

A few sections ago, I said that you can make up any name you want for your variables, but that's not quite true. There are certain words that are reserved in C because they are used by the compiler to parse the structure of your program, and if you use them as variable names, it will get confused. These words, called **keywords**, include int, char, void and many more.

* auto
* break
* case
* char
* const
* continue
* default
* do
* double
* else
* enum
* extern
* float
* for
* goto
* if
* inline
* int
* long
* register
* restrict
* return
* short
* signed
* sizeof
* static
* struct
* switch
* typedef
* union
* unsigned
* void
* volatile
* while
* \_Bool
* \_Complex
* \_Imaginary

The complete list of keywords is included in the C Standard, which is the official language definition adopted by the the International Organization for Standardization (ISO) on September 1, 1998.

Rather than memorize the list, I would suggest that you take advantage of a feature provided in many development environments: code highlighting. As you type, different parts of your program should appear in different colors. For example, keywords might be blue, strings red, and other code black. If you type a variable name and it turns blue, watch out! You might get some strange behavior from the compiler.

## Operators

**Operators** are special symbols that are used to represent simple computations like addition and multiplication. Most of the operators in C do exactly what you would expect them to do, because they are common mathematical symbols. For example, the operator for adding two integers is +.

The following are all legal C expressions whose meaning is more or less obvious:

1+1 hour-1 hour\*60+minute minute/60

**Expressions** can contain both variables names and values. In each case the name of the variable is replaced with its value before the computation is performed.

## Division and Integer Division

Addition, subtraction and multiplication all do what you expect, but you might be surprised by division. For example, the following program:

int hour, minute;

hour = 11;

minute = 59;

printf ("Number of minutes since midnight: %i\n", hour\*60 + minute);

printf ("Fraction of the hour that has passed: %i\n", minute/60);

would generate the following output:

Number of minutes since midnight: 719

Fraction of the hour that has passed: 0

The first line is what we expected, but the second line is odd. The value of the variable minute is 59, and 59 divided by 60 is 0.98333, not 0. The reason for the discrepancy is that C is performing **integer division**.

When both of the **operands** are integers (operands are the things operators operate on), the result must also be an integer, and by definition integer division always rounds down, even in cases like this where the next integer is so close.

A possible alternative in this case is to calculate a percentage rather than a fraction:

printf ("Percentage of the hour that has passed: ");

printf ("%i\n", minute\*100/60);

The result is:

Percentage of the hour that has passed: 98

Again the result is rounded down, but at least now the answer is approximately correct. In order to get an even more accurate answer, we could use a different type of variable, called floating-point, that is capable of storing fractional values. We'll get to that in the next chapter.

## Order of operations

When more than one operator appears in an expression the order of evaluation depends on the rules of **precedence**. A complete explanation of precedence can get complicated, but just to get you started:

* Multiplication and division happen before addition and subtraction. So 2\*3-1 yields 5, not 4, and 2/3-1 yields -1, not 1.
* If the operators have the same precedence they are evaluated from left to right. So in the expression minute\*100/60, the multiplication happens first, yielding 5900/60, which in turn yields 98. If the operations had gone from right to left, the result would be 59\*1 which is 59, which is wrong.
* Any time you want to override the rules of precedence (or you are not sure what they are) you can use parentheses. Expressions in parentheses are evaluated first, so 2\*(3-1) is 4. You can also use parentheses to make an expression easier to read, as in (minute\*100)/60, even though it doesn't change the result.

## Operators for characters

Interestingly, the same mathematical operations that work on integers also work on characters. For example,

char letter;

letter = 'a' + 1;

printf ("%c\n", letter);

outputs the letter b. Although it is syntactically legal to multiply characters, it is almost never useful to do it.

Earlier I said that you can only assign integer values to integer variables and character values to character variables, but that is not completely true. In some cases, C converts automatically between types. For example, the following is legal.

int number;

number = 'a';

printf ("%i\n", number);

The result is 97, which is the number that is used internally by C to represent the letter 'a'. However, it is generally a good idea to treat characters as characters, and integers as integers, and only convert from one to the other if there is a good reason.

Automatic type conversion is an example of a common problem in designing a programming language, which is that there is a conflict between **formalism**, which is the requirement that formal languages should have simple rules with few exceptions, and **convenience**, which is the requirement that programming languages be easy to use in practice.

More often than not, convenience wins, which is usually good for expert programmers, who are spared from rigorous but unwieldy formalism, but bad for beginning programmers, who are often baffled by the complexity of the rules and the number of exceptions. In this book I have tried to simplify things by emphasizing the rules and omitting many of the exceptions.

## Composition

So far we have looked at the elements of a programming language---variables, expressions, and statements---in isolation, without talking about how to combine them.

One of the most useful features of programming languages is their ability to take small building blocks and **compose** them. For example, we know how to multiply integers and we know how to output values; it turns out we can do both at the same time:

printf ("%i\n", 17 \* 3);

Actually, I shouldn't say "at the same time," since in reality the multiplication has to happen before the output, but the point is that any expression, involving numbers, characters, and variables, can be used inside an output statement. We've already seen one example:

printf ("%i\n", hour \* 60 + minute);

You can also put arbitrary expressions on the right-hand side of an assignment statement:

int percentage;

percentage = (minute \* 100) / 60;

This ability may not seem so impressive now, but we will see other examples where composition makes it possible to express complex computations neatly and concisely.

**WARNING:** There are limits on where you can use certain expressions; most notably, the left-hand side of an assignment statement has to be a variable name, not an expression. That's because the left side indicates the storage location where the result will go. Expressions do not represent storage locations, only values. So the following is illegal: minute + 1 = hour;.

## Scanning User Input

In all of the examples so far we have assigned variable values before we run our program. We can also ask the user to input values when the program is running. scanf() is a function in the C programming language that is used to read input from the user. It can be used to read various types of data, such as integers, characters, and strings (we will get back to strings in a later chapter)

To read a single character from the user, you can use the %c format specifier. The %c format specifier tells scanf() to read a character and store it in the variable. For example:

char c;

scanf("%c", &c);

In this example, scanf() will read a character from the user's input and store it in the variable c. Note that the & operator is used when we refer to the variable name. This will be explained in a later chapter, but it is important not to forget it here.

To read an integer, you can use the %d or %i format specifier. These specifiers tells scanf() to read an integer and store it in the variable. For example:

int x;

scanf("%d", &x);

In this example, scanf() will read an integer from the user's input and store it in the variable x.

scanf() can also be used to scan strings, but we will get back to strings in a later chapter.

## Prompting for User Input

Prompting for input is a way to gather information from the user and store it in variables for use in the program. When using scanf(), you first need to print a prompt to let the user know what to enter. This can be done using printf().

For example, to prompt the user for an integer, you can use:

int x;

printf("%s", "Enter an integer: \n");

scanf("%d", &x);

In this example, the printf() function is used to print the message "Enter an integer: " with a new line at the end. The user can enter an integer and the scanf() function is used to read the integer from the user and store it in the variable x.

**Caution** It is considered insecure to useprintf() to print a string without the string format specifier.

printf( "Enter an integer: \n"); //works but is considered insecure

printf("%s", "Enter an integer: \n"); // this is considered a secure use

As an alternative to printing strings with printf(), you can use the puts() function to print a string. Unlike printf(), puts() automatically appends a newline character after the string, which can be useful if you want to print multiple strings on separate lines.

For example:

puts("Welcome to the program");

puts("Enter an integer:");

## Mixing Calls to Scan Integers and Character

It is important to note that when mixing calls to scanf() to read different types, newline characters can cause problems. For example, if the user enters an integer followed by a newline character, the newline character will be left in the input buffer and can cause issues when trying to read the next input using scanf().

One way to manage this is to include a space in the quotes before the format specifier - like this " \%d". The space indicates to ignore whitespace characters.

#include <stdio.h>

#include <stdlib.h>

int main (void)

{

int age;

char initial;

puts("enter your age");

scanf("%d", &age);

puts("enter your first intial");

scanf(" %c", &initial); //note the space after " and before %

}

Another way to manage this is to insert a second call to scan the newline.

#include <stdio.h>

#include <stdlib.h>

int main (void)

{

int age;

char initial;

char newline;

puts("enter your age");

scanf("%d", &age);

scanf("%c", &newline); //this will scan the extra new line

puts("enter your first intial");

scanf("%c", &initial);

}

## Glossary

variable:

A named storage location for values. All variables have a type, which determines which values it can store.

value:

A letter, or number, or other thing that can be stored in a variable.

type:

The meaning of values. The types we have seen so far are integers (int in C) and characters (char in C).

keyword:

A reserved word that is used by the compiler to parse programs. Examples we have seen include int, void, and char.

statement:

A line of code that represents a command or action. So far, the statements we have seen are declarations, assignments, and output statements.

declaration:

A statement that creates a new variable and determines its type.

assignment:

A statement that assigns a value to a variable.

expression:

A combination of variables, operators and values that represents a single result value. Expressions also have types, as determined by their operators and operands.

format specifier:

A special character or sequence of characters that tells the printing and scanning functions how to format and interpret data.

operator:

A special symbol that represents a simple computation like addition or multiplication.

operand:

One of the values on which an operator operates.

precedence:

The order in which operations are evaluated.

composition:

The ability to combine simple expressions and statements into compound statements and expressions in order to represent complex computations concisely.