2. Printing, variables and user input

Herman Kamper

The previous lecture ended with a program that printed a fixed message on the screen. Useful as a starting point, but not very interesting. Real programs need to store values, compute with them and respond to input from the user. This lecture introduces the tools that make that possible: escape sequences for formatted output, variables for storing data, arithmetic for computing with the variables, and `scanf` for reading from the keyboard.

Consider a concrete example. A user types in a resistance R and a current I, and the program computes and prints the voltage $V = RI$. That is a real program solving a real engineering problem. Recall the four components of a computer from the previous lecture: this program has input (R and I from the keyboard), processing (multiply them together) and output (print V on the screen). To write it, we need three things: a way to store the values R, I and V inside the program, a way to compute $V = R \times I$, and a way to read R and I from the keyboard. This lecture builds each of those pieces and ends by putting them all together into the Ohm's law program.

Formatted output

In the previous lecture `printf` printed a fixed string. We can do more than that with *escape sequences*, special codes inside the string that control formatting. Each one starts with a backslash. The most useful ones are:

Escape sequence	Effect
`\n`	moves to a new line
`\t`	inserts a tab stop
`\\`	prints a literal backslash
`\"`	prints a literal double quote

The last two are needed because the backslash and the double quote already have special meanings inside a string: a lone `"` would end the string early, and a lone `\` would be read as the start of an escape sequence. The following program uses several of these codes:

```c
#include <stdio.h>

int main()
{
    printf("\tHello\nwonderful\nworld!\n");
    printf("I \\ need coffee\n");

    return 0;
}
```

The output is:

```
    Hello
wonderful
world!
I \ need coffee
```

The `\t` at the start of the first `printf` indents "Hello" with a tab; the three `\n` codes split the output across three lines. In the second `printf`, `\\` produces a single backslash.

Variables: programming vs mathematics

In mathematics, a variable is a symbol that stands for a value. If you write $x = 5$, $y = 3$ and $z = x - y$, then $z = 2$. Now change x to 6 and set $z = x + y$. What is z now? The trick is to track each variable as a box that holds a value, updating the boxes as each step executes:

Variable	Box
x	$5 \to 6$
y	3
z	$2 \to 9$

So z is 9 after the last step. Notice that y is not 3 in any permanent sense; it is a box that currently contains 3. When we reassigned z, the old value 2 was thrown away and replaced by 9.

Variables in programming work in almost the same way,[1] and they are exactly this: named boxes that hold values. You already understand the idea from school mathematics. The next step is to see how it works in C.

[1] The main difference is that a program updates a variable only when you explicitly assign to it; the value does not change on its own. Sometimes in mathematics we do think of variables automatically updating based on other variables. If this confuses you, just ignore this comment.

Variables in C

A variable in C has four properties: a name, a value, a type and an address in memory. The name and value are familiar from mathematics; the type and the address are new. Here is a program that creates three variables and adds two of them:

```c
#include <stdio.h>

int main()
{
    int integer1;  // declares the box
    int integer2;
    int sum;

    integer1 = 45;
    integer2 = 72;

    sum = integer1 + integer2;

    printf("Sum is %d\n", sum);
    return 0;
}
```

Before a variable can be used it must be *declared*. The declaration

```c
int integer1;
```

asks the compiler to reserve a box in memory, name it `integer1`, and treat whatever goes into it as an integer (the `int` keyword). A declaration does not store a value; it just creates the box. The assignment `integer1 = 45;` then puts the value 45 into that box. After all three assignments, the memory looks as follows:

Type	Name	Address (bytes)	Value
`int`	`integer1`	3000	45
`int`	`integer2`	3004	72
`int`	`sum`	9004	117

The address is the position in the computer's memory where the box physically lives, as ones and zeros on the chip.[2] You never set it yourself: C places each variable somewhere in memory and keeps track of the address for you. When the program evaluates `integer1 + integer2`, it goes to the address of each box, reads the value stored there, adds the two, and writes the result to the address of `sum`.

[2]One byte is eight bits. This becomes important in just a bit when we look at the sizes of the different boxes. For instance in this table, `integer2` is 4 bytes after `integer1`, which means it is 32 bits after it. The address blocks doesn't always follow on each other exactly like this; the operating system decided to put `sum` in a completely different part of the memory.

The %d inside the `printf` string is a *format specifier*: a placeholder that gets replaced with the value of the variable when the program runs. The `d` stands for decimal integer, so %d goes with `int` variables. You can print several values in one call by using several specifiers:

```
printf("%d + %d = %d\n", integer1, integer2, sum);
```

Here the first %d is replaced by `integer1`, the second by `integer2` and the third by `sum`, giving the output `45 + 72 = 117`.

You can also initialise a variable on the same line as its declaration:

```
int integer1 = 45;
```

Always initialise a variable before you use it. A declared but uninitialised variable contains whatever happened to be at that memory location beforehand. Reading it back gives meaningless numbers, and using one by accident is a common source of subtle bugs. By convention, declare all of a function's variables together at the top of the block, before the statements that use them.

Types and memory

Why must you specify a type at all? Because the computer needs to know how big to make the box, that is, how many ones and zeros to set aside, and how to interpret the bits stored inside it. The four types you will use most in this course are:

Category	C type	Size	Range
Characters	`char`	8 bits	-128 to 127
Integers	`int`	32 bits	$-2\,147\,483\,648$ to $2\,147\,483\,647$
Decimals	`float`	32 bits	$\approx \pm 3.4 \cdot 10^{38}$
Decimals (more precise)	`double`	64 bits	$\approx \pm 1.7 \cdot 10^{308}$

The size explains the addresses in the previous table. An `int` occupies 4 bytes (32 bits), so `integer2` starts 4 bytes after `integer1`: address 3000 becomes 3004. A `char` needs only 1 byte, since a single character can be stored with far fewer ones and zeros than a whole number.

The choice between `float` and `double` is about precision. Neither can store a value like π exactly, because that would take infinitely many digits, but a `double` uses twice as many bits and so holds many more decimal places. A sensible default is to use `float`, and switch to `double` only when you genuinely need the extra precision.

The following declarations create one variable of three different types:

```
int population = 540000;
char letter = 'a';
double votePercentage = 0.4512;
```

Given that `int` takes 4 bytes and `char` takes 1, the three variables might be laid out in memory like this:

Type	Name	Address (bytes)	Value
`int`	`population`	3000	540 000
`char`	`letter`	3004	`'a'`
`double`	`votePercentage`	3005	0.4512

Note that a single character is enclosed in single quotes (`'a'`), not double quotes. Double quotes are for strings of text; single quotes are for individual characters. We will look at strings and characters in much more detail later.

Arithmetic

C supports the standard arithmetic operators: `+`, `-`, `*` and `/`. You can use them directly inside a `printf`:

```
printf("Product: %d\n", integer1 * integer2);
printf("Average: %d\n", (integer1 + integer2) / 2);
```

Two things are worth knowing.

First, operator precedence works as in mathematics: `*` and `/` bind more tightly than `+` and `-`. Parentheses override the default order, as in the average calculation above, where the addition must happen before the division.

Second, *integer division* truncates the result.[3] When both operands are of type `int`, the division produces an `int`: `45 / 2` gives `22`, not `22.5`. The fractional part is silently discarded. This catches many beginners by surprise. If you need a decimal result, use `float` or `double` variables instead.

[3] Dividing by zero is a different matter: it is not defined, and the program crashes at runtime rather than giving a wrong answer.

Getting input from the user

So far every value has been written directly into the program. To make a program useful, you need to read values from the keyboard while it runs. In C this is done with `scanf`:

```c
/* Getting user input. */
#include <stdio.h>

int main()
{
    int integer1;
    int integer2;
    int sum;

    printf("Enter first integer\n");
    scanf("%d", &integer1);

    printf("Enter second integer\n");
    scanf("%d", &integer2);

    sum = integer1 + integer2;

    printf("Sum is %d\n", sum);
    return 0;
}
```

`scanf` looks similar to `printf`: you give it a format string with a specifier and a variable. The critical difference is the `&` before the variable name. The `&` operator means "the memory address of": it tells `scanf` exactly where in memory to write the value the user types. Without the `&`, `scanf` would be handed the current contents of the box rather than its location, and would have no idea where to put the result. Forgetting the `&` is one of the most common beginner mistakes.

You can print an address directly to see that it is a real location:

```c
printf("Address: %p\n", &integer1);
```

The `%p` specifier prints a memory address. The exact value differs every run, but it confirms that `integer1` lives at a specific place in your computer's memory.

Putting it all together: Ohm's law

We now have every piece. Recall the goal: the user provides a resistance R and a current I, and the program computes and prints the voltage $V = RI$.

Voltages, resistances and currents are not whole numbers, so `int` is the wrong type. We use `float`, which holds decimal values. The format specifier for `float` is `%f`, used for both `scanf` and `printf`:

```c
/* Compute voltage using Ohm's law. */
#include <stdio.h>

int main()
{
    float R;  // resistance in ohms
    float I;  // current in amps
    float V;  // voltage in volts

    printf("Enter resistance (ohms): ");
    scanf("%f", &R);

    printf("Enter current (amps): ");
    scanf("%f", &I);

    V = R * I;

    printf("Voltage:\t%f volts\n", V);
    return 0;
}
```

To print with exactly two decimal places, replace `%f` with `%.2f`. The number between the dot and the `f` controls the precision.

This small program is a complete example of what a computer does: input (R and I from the keyboard), processing ($V = R \times I$) and output (the result on the screen). It is the same pattern at the heart of every program you will write.

Looking forward

This lecture has introduced variables, arithmetic and user input. The next lecture looks at *selection*: how to make a program choose between different actions depending on the value of a variable. That small addition turns a program from a fixed sequence of steps into something that can respond to different situations. It is the first step towards programs that solve real problems.

Exercises

1. What is the output of the following program? Work it out by hand before running it.

```c
#include <stdio.h>

int main()
{
    printf("Name:\tHerman\n");
    printf("Course:\tCP143\n");
    return 0;
}
```

2. Consider the following C code:

```c
int x = 4;
int y = 7;
int z = x + y;
x = 10;
```

   (a) In mathematics, if $x = 4$, $y = 7$ and $z = x + y$, and then x changes to 10, what is z?
   (b) In the C program above, what is the value stored in `z` after the last line? Explain why.

3. What is the value of each expression below, assuming `a = 10` and `b = 3` are both of type `int`?

   (a) `a + b * 2`
   (b) `(a + b) * 2`
   (c) `a / b`
   (d) `a % b` *(hint: look up the modulo operator)*

4. Declare three `int` variables `x`, `y` and `z`. Assign `x = 8` and `y = 3`. Then write the assignment statement that stores the value of `x - y` in `z` and print it with `printf`.

5. Write a complete C program that asks the user for two integers and prints their sum, difference, product and quotient, each on its own line with a descriptive label. Use `%d` throughout.

6. The formula for the area of a triangle is $A = \frac{1}{2}bh$. Write a C program that reads the base b and height h as `float` values and prints the area with two decimal places.

7. Why is the `&` operator required in `scanf` but not in `printf`? What does `&integer1` represent, and what would happen if you omitted the `&`?