C FUNDAMENTALS - VARIABLES

CSC100 Introduction to programming in C/C++ Spring 2024

Marcus Birkenkrahe

January 30, 2024

Contents

1 README

- This script summarizes and adds to the treatment by King (2008), chapter 2, C Fundamentals - see also slides (GDrive).
- To **code along during the lecture** using Emacs and Org-mode, open the EWW browser inside Emacs (`M-x eww`) and download the file from tinyurl.com/4-variables-codealong-org, then save it (`C-x C-w`) as an Org-mode (`.org`) file in your home directory `~/`.
- There is a separate Org-mode file available for **practice after the lecture**, at tinyurl.com/4-variables-codealong-org. Download and complete it for bonus points.

2 VARIABLE TYPES AND DECLARATIONS

- C computes using placeholders, or **variables**, to manage memory

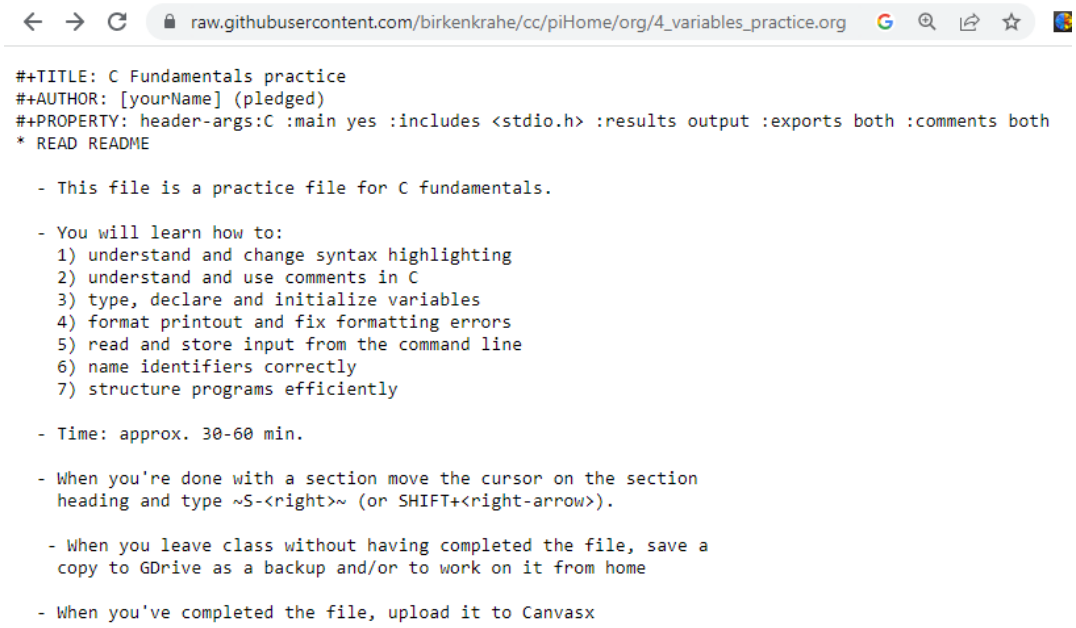

Figure 1: Raw file to code along in GitHub

- Each variable must have a **type** to specify the data it can hold
- E.g. `int` (integer), `float` (floating point), `char` (character)
- Variables must be **declared** before they can be used:

```
int height;
float profit;
char name;
```

- Several variables of the same type can be declared **together**:

```
int height, length, width, volume;
float profit, loss;
char first_name, last_name;
```

- Variable type declarations **must precede statements** that use the variables[1] - you must tell the computer first, how much **memory** you'll need for the job.

[1] In the C99 standard, declarations don't have to come before statements.

3 VARIABLE ASSIGNMENT

- A variable gets its value through **assignment**
- In the code block below, the variable `height` gets the value `8`. `8` is called a **"string literal"** because it cannot change.

```
height = 8;
```

☐ If you would try to run the code above, you would get an error. Can you see why?[2]

☐ The example below would throw another error. What's wrong now?[3]

```
height = 8;
int height;
```

☐ Phew! The next block finally works, that is, it compiles and runs. But what does the code actually do?[4]

```
int height;
height = 8;
```

- A *string literal* assigned to a `float` variable contains a decimal point and the letter `f` to indicate its "floatiness":

```
float profit;
profit = 2150.48f;
```

- Assigning a `float` to an `int` and vice versa is possible (but not safe as we will see):

```
/* ASSIGNING A float TO AN int */
float profit;
int iProfit;
iProfit = 2150.48;  // Don't do this!
```

[2] Assignment is variable use. Variable types must be declared before they can be used.

[3] The declaration must precede the use of the variable.

[4] `int height;` reserves memory for an integer variable, and `height=8;` puts the *numeric integer* value `8` into the memory cell. From now on, whenever you use `height`, the computer will substitute `8` for it.

```
/* ASSIGNING AN int TO AN float */
float profit;
int iProfit;
profit = 2150;  // Don't do this!
```

☒ Variables with values can be used to compute other values:

```
int height, length, width, volume;

height = 8;
length = 12;
width = 10;
volume = height * length * width;
```

☒ How many things does this last little program have to do[5]?

- You can also initiate and declare several variables at once. Here, the `volume` from before is computed inside `printf`:

```
int height = 8, length = 12, width = 10;
printf("Volume: %d", height * length * width);
```

```
Volume: 960
```

- To print these variables, we need to learn **formatting** identifiers, expressions like `%d` that you've seen before.

4 FORMATTING WITH `printf` VS. `puts`

- We use the built-in (via `stdio.h`) function `printf` to print.
- We also used `puts` in the past, which includes the newline character `\n` that we need to add for `printf`[6].

[5] Answer: 11 things! (1) memory allocation for four integer variables; (2) assignments for four variables; (3) multiplication of three integers.

[6] Python e.g. is white-space sensitive: the indentation level is significant, it denotes code blocks, and needs to be consistent. The same goes for Org-mode markdown and code blocks.

- the standard input/output library `stdio.h` also contains `putchar()`, which prints a character to the screen.

```
char c = 'A';
putchar(c);

A
```

5 FORMATTING INTEGER NUMBERS WITH %d

- The *format specifier* `%d` is used to print an `int`:

```
int height;  // type declaration
height = 8;  // variable assignment

printf("The height is: %d\n", height); // formatted printout

The height is: 8
```

6 FORMATTING FLOATING-POINT NUMBERS WITH %f

- The format specifier `%f` is used to print a `float`:

```
float profit; // type declaration
profit = 2150.48f; // variable assignment

printf("The profit is: $%f\n", profit); // formatted printout

The profit is: $2150.479980
```

7 CHANGE FLOATING POINT PRECISION

- By default, `%f` displays the result with **six digits** after the decimal point. To change it to `p` digits, put `.p` between `%` and `f`. E.g. to print it with 2 digits, `p=2`:

```
float profit;       // type declaration
profit = 2150.48f;  // variable assignment

// formatted printout: 2 digits after decimal point
printf("The profit is: $%.2f\n", profit);

The profit is: $2150.48
```

- Formatting instructions need to be precise: if you don't specify `p=2`, the computer simply makes digits up! The output below is `$2150.479980`, which can be rounded to the correct result, but it is strictly not the same number! See for yourself:

```
float profit;       // type declaration
profit = 2150.48f;  // variable assignment

printf("The profit is: $%.8f\n", profit); // formatted printout

The profit is: $2150.47998047
```

8 FORMATTING ERRORS

- Bad things happen when you get the formatting wrong.
- Below, we print a `float` first correctly, then with the *wrong* format specifier, and then the other way around.

```
float foo;  // declare float
foo = 3.14f;   // assigned float

// formatting float as float
printf("float as float: %.2f\n", foo);

// formating float as int
printf("float as int: %d\n", foo);

int bar;  // defined int
bar = 314;   // assigned int
```

```
// formatting int as int
printf("int as int: %d\n", bar);

// formatting int as float
printf("int as float: %.2f\n", bar);  // int as float

float as float: 3.14
float as int: 1610612736
int as int: 314
int as float: 0.00
```

- When you print an `integer` as a floating point number or vice versa, the results are in general unpredictable!

9 PUTTING IT ALL TOGETHER (EXTENDED EXAMPLE)

- Shipping fees are based on volume instead of weight. For the conversion, the volume is divided by 166. If the result exceeds the actual weight, the shipping fee is based on the "dimensional weight"[7].

- We write a program to compute the dimensional `weight` of a box of given `volume` - we use `/` for division. Let's say the box is 12" x 10" x 8 ". How can we compute this in C?

```
volume = 12 * 10 * 8   // volume = height * width * length
weight = volume / 166  // dimensional weight
```

- Fixed the errors in the block below. The compiler no longer complains, but we don't get any output. How can we print the result?

```
int weight, volume;
volume = 12 * 10 * 8;
weight = volume / 166;
```

- This code prints the result of the computation using the format specifier for integer values:

[7] The tokenization is an important sub-process of natural language processing, a data science discipline that is responsible for language assistants like Siri, robotic calls, autocoding and machine translation (like Google translate).

```
int weight, volume;     // declare variable types
volume = 12 * 10 * 8;   // compute value
weight = volume / 166;  // assign and compute values
printf("The dimensional weight is %d\n",weight); // print result
```

```
The dimensional weight is 5
```

- This is not what we need. When dividing one integer by another, C "truncates" the answer - the result is rounded down, but the shipping company wants us to round up. This can be achieved by adding 165 to the volume before dividing by 166[8] as shown:

```
int weight, volume;     // declare variable types
volume = 12 * 10 * 8;   // compute value
weight = (volume + 165) / 166;  // assign and compute values
printf("The dimensional weight is %d\n",weight); // print result
```

```
The dimensional weight is 6
```

☐ Now for the final program. I have set it up so that this can be tangled as a file `dweight.c`:

```
// declare variable types
int height, length, width, volume, weight;

// variable assignments
height = 8;
length = 12;
width = 10;
volume = height * length * width;
weight = (volume + 165) / 166;

// print results
printf("Dimensions: %d times %d times %d\n", length, width, height);
printf("Volume (cubic inches): %d\n", volume);
printf("Dimensional weight (pounds): %d\n", weight);
```

[8] You cannot enter input in an Org-mode file interactively. You either have to tangle the code and compile/run it on the command line, or redirect the input using the `:cmdline < file` header argument, where `file` contains the input.

```
Dimensions: 12 times 10 times 8
Volume (cubic inches): 960
Dimensional weight (pounds): 6
```

10 LET'S PRACTICE!

Get `4_variables_practice.org` from GitHub and complete it.

1. Typing, declaring and initializing variables
2. Formatting printout and fixing formatting errors

   `../img/practice.gif`

11 SUMMARY

- C programs must be compiled and linked
- Programs consist of directives, functions, and statements
- C directives begin with a hash mark (`#`)
- C statements end with a semicolon (`;`)
- C functions begin and end with parentheses `{` and `}`
- C programs should be readable
- Input and output has to be formatted correctly

12 CODE SUMMARY

CODE	EXPLANATION
`#include`	directive to include other programs
`stdio.h`	standard input/output header file
`main(int argc, char **argv)`	main function with two arguments
`return`	statement (successful completion)
`void`	empty argument - no value
`printf`	printing function
`\n`	escape character (new-line)
`/* ... */ //...`	comments
`scanf`	input pattern function
`main(void)`	main function without argument

13 GLOSSARY

CONCEPT	EXPLANATION
Compiler	translates source code to object code
Linker	translates object code to machine code
Syntax	language rules
Debugger	checks syntax
Directive	starts with `#`, one line only, no delimiter
Preprocessor	processes directives
Statement	command to be executed, e.g. `return`
Delimiter	ends a statement (in C: semicolon - ;)
Function	a rule to compute something with arguments
String	Sequence of *character* values like `hello`
String literal	Unchangeable, like the numbe `8` or the string `hello`
Constant	Set value that is not changed
Variable	A named memory placeholder for a value, e.g. `int i`
Data type	A memory storage instruction like `int` for integer
Comment	Region of code that is not executed
Format specifier	Formatting symbol like `%d%` or `%f%`
Data type	Tells the computer to reserve memory, e.g. `int` for integer numbers
Type declaration	Combination of type and variable name - e.g. `int height;`
`int`	C type for integer numbers, e.g. 2
`float`	C type for floating point numbers, e.g. 3.14
`char`	C type for characters, like "joey"
Formatting	Tells the computer how to print, e.g. `%d` for `int` types
`%d`	Format for integers
`%f` and `%.pf`	Format for floating point numbers (with `p` digits after the point)
`#define`	Define a constant with the preprocessor, e.g. `#define PI 3.14`
`math.h`	Math library, contains mathematical constants & functions
`stdio.h`	Input/Output library, enables `printf` and `scanf`
`const`	Constant identifier, e.g. `const double PI = 3.14;`

14 REFERENCES

- Collingbourne (2019). The Little Book of C (Rev. 1.2). Dark Neon.

- King (2008). C Programming - A Modern Approach. Norton. URL:knking.com.