Introduction to Programming - Variables and Types

CSC 100 - Lyon College - Spring 2025

Marcus Birkenkrahe

January 6, 2025

What are you going to learn?

The secret of the success of the machine is the separation of **value** and **variable**, or of numbers and storage place. When we **operate** on numbers, we can keep the original operands and the result in different places. For the computer to understand what we want to do, it needs to see reserved words, or **keywords**, that have a special meaning. One of the advantages of C is that there are very few keywords. To handle numbers with its limited memory, the computer needs to know what kind or **type** of numbers they are (for example, large or small, whole or decimal). Printing numbers of a certain type requires us to specify their **format**.

Topics:

☐ Printing (`printf` and `puts`)

☐ Format (`%s`, `%i`, `%f`, `%c`)

☐ Values (`1`, `1.0`)

In-class and home assignments

- **Reading**: Chapter 2 (**Think C**) - pp. 13-22.
- **Practice exercises** (in-class):

  For in-class practice, we use OneCompiler (see below). You upload the link to your practice file to Canvas.

1. Print different values with the wrong format specifiers
2. Printing challenge

- **Programming assignments (home)**:
  For home assignments, you can choose your own method.

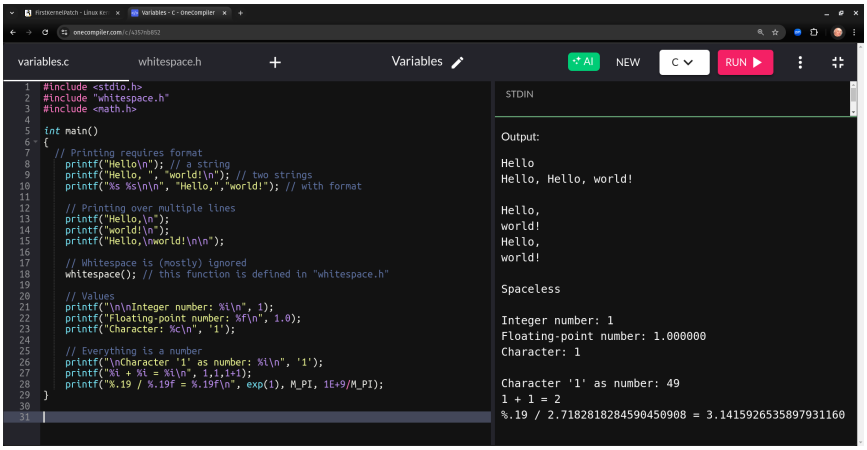
  1.

Code along in OneCompiler

To code along in OneCompiler:

1. Open `onecompiler.com/c`
2. Open `settings` and change `Title` to `"Printing values"`
3. In `Editor Settings`, `Disable Code Autocomplete/Suggestions`
4. Change file name from `Main.c` to `main.c` (there can be only one)
5. Delete the body of the code (single Hello World print statement)
6. Now write any code from the lecture in the body of `main`
7. Execute with `RUN`
8. You find the complete lecture in GitHub as `2_variables.org`
9. Equip your code along files with comments liberally!
10. You can find the complete code along file here: tinyurl.com/2ab5sjle

Each code along session will fill about one screen:



Program structure

- All C program statements must be included in a `main` function
- The `main` function has a body delimited by `{...}`
- There can be *pre-processor directives* - `#include` or `#define`.
- `main()` is similar to `f(x)` in mathematics - () means "no argument"
- `printf()` prints its argument: `"hello world"` which is a 'string'
- `\n` means "go to the next line" - 'escape character'
- `;` ends every command - the computer waits for the next one!
- The computer (aka compiler) ignores "white [empty] space"

Printing requires format

- The `printf` function can be called as many times as you like.
- The function's argument (between parentheses) can be a string:

```
printf("Hello");
```

- But two strings cannot be printed (you'll get a warning):

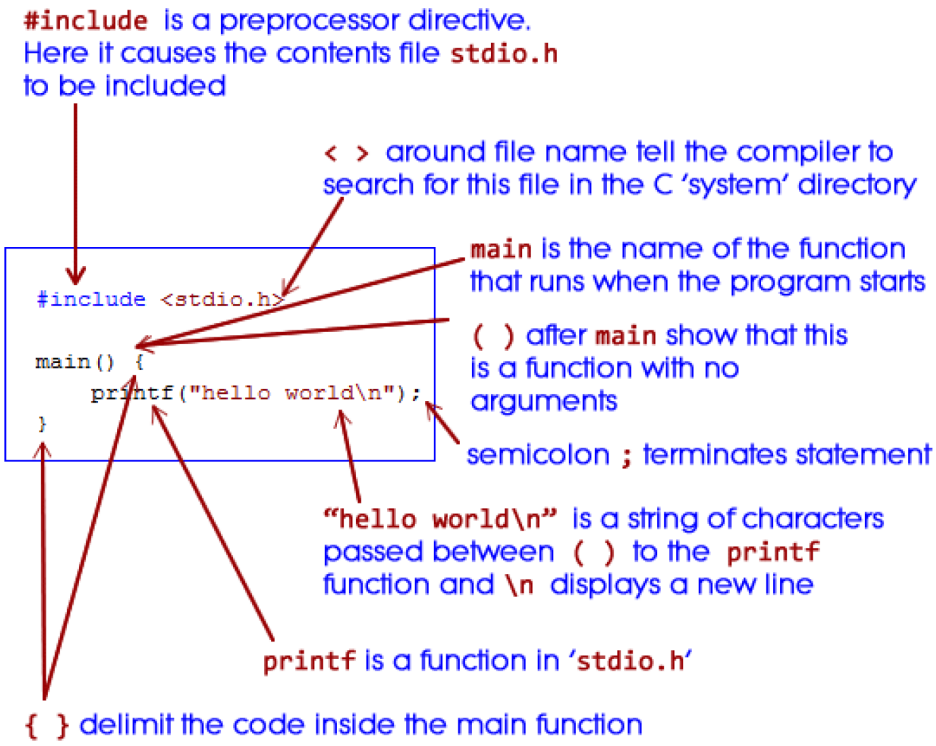


Figure 1: main function structure (Collingbourne, 2017)

```
printf("Hello, ", "world!");
```

- In Python, this would be possible:

```
print("Hello,", "world!") # print two strings
print("Hello," + "world!") # concatenate two strings

Hello, world!
Hello,world!
```

- In C, we can do this only if we tell the compiler what's coming:

```
printf("%s %s\n", "Hello,", "world!"); // print two strings
```

- The command `printf` is called a *function* that takes two arguments: A list of arguments and a string that contains formatting information.
- In the example:
  1. `"%s %s\n"` is the format string: "Print 2 strings and a new line".
  2. `"Hello,","world!"` is a list of two strings to be printed.
- Much of programming (not just in C) boils down to picking the right representations for numbers that you want the computer to work with.
- This is not unlike college algebra where the way we write numbers helps us see relationships and simplify operations.
- Example: Factoring $x^2 + 5x + 6 = (x + 2)(x + 3)$ - this representation helps us realize that the roots of the polynomial (where it equals 0) are $x = -2$ and $x = -3$.

Printing over multiple lines

- To print "Hello, world!" over two lines, you can use one or two function calls:

```
printf("Hello,\n");
printf("world!\n");

printf("Hello,\nworld!\n");
```

Whitespace is (mostly) ignored

- As soon as a delimiter `;` is reached, all space ('whitespace') until the next command is ignored.
- Here is a version of the complete "Hello, world!" program that ignores whitespace at the expense of readability:

```
int main(){printf("Hello, world!\n");}
```

- The only required space comes after reserved keywords (syntax highlighting reveals that because keywords are color-coded - though not in Colab).
- The *preprocessor* directive `#include <stdio.h>` also must be on a line of its own.

Values

- The machine only knows numbers:
  1. `1` is a whole or **integer** number:

     ```
     printf("Integer number: %i\n", 1);
     ```

  2. `1.0` is a decimal or **floating-point** number.

     ```
     printf("Floating-point number: %.1f\n", 1.0);
     ```

  3. `'1'` is a character (also stored as a number).

     ```
     printf("Character: %c\n", '1');
     ```

Everything is a number

- To show that characters are stored as numbers, print it as a number:

```
printf("Character '1' as number: %i\n", '1');
```

- Though we'll mostly work with these three, C has many different ways of representing numbers.

- Which representation (for example, long or short) is the best depends on the problem.
- Example:

  1. Adding 1 + 1 and showing the result is short:

     ```
     printf("%i + %i = %i\n", 1, 1, 1 + 1);
     ```

  2. Dividing Euler's number (`exp(1)` by Pi (`3.14159...`) is long:

     ```
     printf("%.19f / %.19f = %.19f\n", exp(1), M_PI, 1E+9 / M_PI);
     ```

PRACTICE About classroom practice sessions

- These and all of the following `PRACTCE` exercises are for you to complete in class. If you're not able, you should complete them at home using the available Zoom recordings.
- How you do this depends on you:

  1. If you have Emacs + Org-mode, you can do this effortlessly (but you have to know your way around Emacs + Org-mode). You can create new code blocks (like in Google Colab) and run them without having to enter `#include ...  return 0` every time.
  2. If you only have a browser, you have choices as explained in the orientation lecture. `onecompiler.com` is the easiest:
     - Open onecompiler.com/c in the browser if you haven't done it yet.
     - Create a new project with a `main.c` file
     - Give it a suitable title like `"Variables - Practice"`, write a short description, like:

       Project for practicing variables in C - CSC 100 class.

       and tag it:

       `practice, variable, csc100`

       Make sure the visibility is `Public (visible to everyone)` so that you can post the link to Canvas if requested.
     - In the `editor settings`, check `Disable Code Autocomplete/Suggestions`
     - Delete the "hello world" printing line and off you go!

PRACTICE Everything is a number

- What do you get when you get the format wrong? The results will surprise you.
- Print these values using the requested format in a `printf` call:

	VALUE	FORMAT	SPECIFIER
1	`3.14`	integer	`%i`
2	`3.14`	character	`%c`
3	`3`	floating-point	`%f`
4	`3`	character	`%c`
5	`'3'`	integer	`%i`
6	`'3'`	floating-point	`%f`

- Solution & Explanation:

  1. **Printing a floating-point number as an integer:** The output is an integer representation of the bits read by `printf` - it changes because the mismatch between the format specifier and the value argument causes **undefined behavior**.

     ```
     printf("%i\n", 3.14);
     ```

     ```
     -663820216
     ```

  2. **Printing a floating-point number as character**: Prints a garbage character (not recognized as an ASCII character).

     ```
     printf("%c\n", 3.14);
     ```

     ```
     H
     ```

  3. **Printing an integer as floating-point number**: Undefined behavior again. This result could also be another value.

     ```
     printf("%f\n", 3);
     ```

  4. **Printing an integer as character**: `3` is interpreted as the character with the ASCII code 3.

     ```
     printf("%c\n", 3);
     ```

5. **Printing a character as an integer**: Results in printing the ASCII code of the character.

   ```
   printf("%i\n", '3');
   ```

6. **Printing a character as a floating-point number**: Undefined behavior.

   ```
   printf("%f\n", '3');
   ```

PRACTICE Print challenge

Print `3.14 - 3 = .14` using `printf` and the values `3.14`, `3`, and `.14`

```
printf("%f - %i = %f\n", 3.14, 3, .14);
printf("%.2f - %i = %.2f\n", 3.14, 3, .14);
```

Glossary: Printing Values

Term	Definition
Value	Data that can be operated on, such as a number or character.
Variable	A named storage location that holds a value.
Type	A classification of values, e.g. integer (`int`)
Format	How a value should be printed using specifiers like `%i`, `%f`, or `%s`.
Keyword	Reserved word with a special meaning in C, such as `return`, `void`, or `main`.
Function	A reusable block of code that performs a specific task, such as `printf()`.

Summary: Printing Values in C

1. **Values** are data items like numbers or characters that the computer processes.
2. **Variables** store values and are referenced by unique names.
3. **Types** define what kind of values variables can hold (e.g., `int` for whole numbers, `float` for decimals).
4. **Format specifiers** like `%i` (integer), `%f` (float), and `%s` (string) control how values are printed.

5. **Keywords** are reserved words in C that the compiler understands as commands.

6. **Functions** like `printf()` are essential to printing values in C.