C Basics

CSC100 Introduction to programming in C/C++ Summer 2022

Table of Contents

- 1. README
- 2. Program structure
- 3. Hello world program
- 4. Compiler workflow
- 5. Shell execution
- 6. Syntax highlighting
- 7. Comments
- 8. Let's practice!
- 9. Variable types and declarations
- 10. Variable assignment
- 11. Formatting printout
- 12. Putting it all together (extended example)
- 13. Let's practice!
- 14. Constants
- 15. Reading input
- 16. Naming identifiers
- 17. Program Layout
- 18. Summary
- 19. Code summary
- 20. Glossary
- 21. References

1 README

- This script summarizes and adds to the treatment by King (2008), chapter 2, C Fundamentals - see also slides (GDrive).

- There is a separate Org-mode file available for practice. Download practice.org using `wget` from GitHub as `practice3.org`:

```
wget tinyurl.com/yckuhh2f -O practice3.org -o log
```

2 Program structure

- All C program statements must be included in a `main` function
- The `main` function has a body delimited by `{...}`
- There can be *preprocessor directives* - `#include` or `#define`.

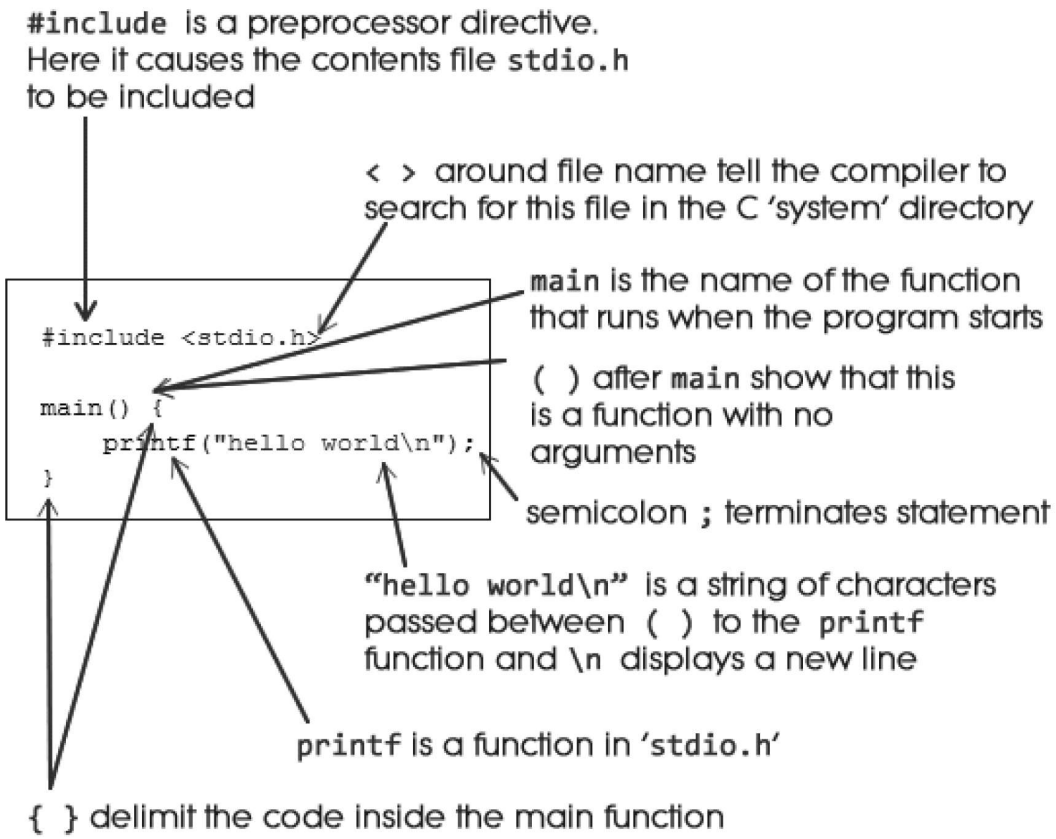


Figure 1: main function structure (Collingbourne, 2017)

3 Hello world program

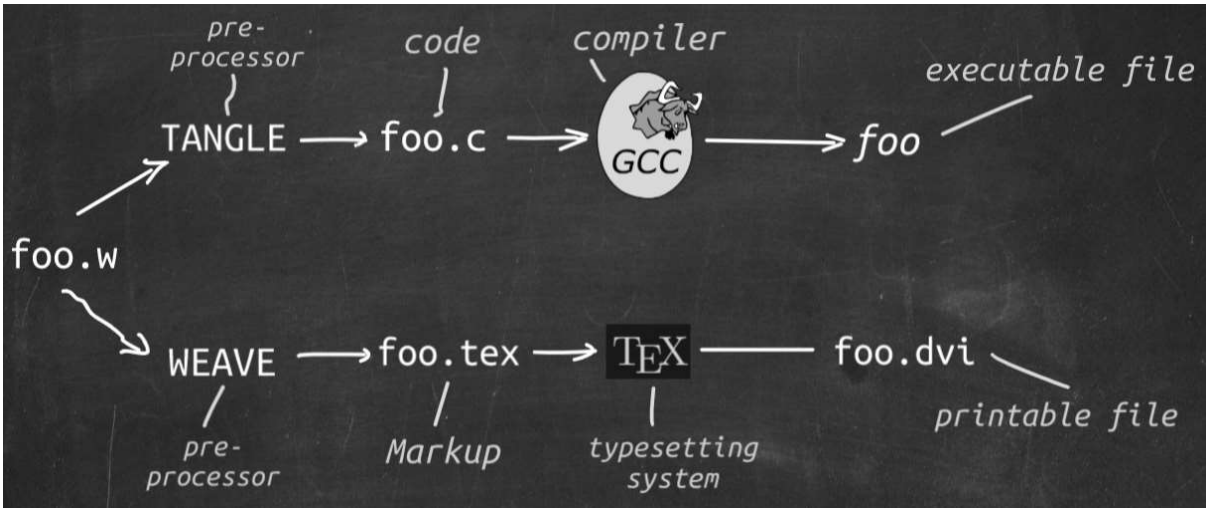
3.1 "What a Tangled Web We Weave…"

> "Oh, what a tangled web we weave, when first we practice to deceive!" (Sir Walter Scott, 1808)

In this section, we're once again running code blocks from within Org-mode - with a few new literate programming features:

- To distinguish (and reference) code blocks, we will name them (`#+NAME:`). The name can can then be referenced anywhere
- To turn the code block into a source code C file (`.c`), we will add a `:tangle FILENAME` statement to the header
- To create the tangled (source code) file from a block, use the keys `C-c C-v t` (`org-babel-tangle`)
- To create the tangled (source code) from a file (all blocks), use the keys `C-c C-v f` (`org-babel-tangle-file`)
- Since source code files should have comments, we add the header argument `:comments both`: now, the most recent org block is used as a comment

- The workflow of "tangling" and "weaving"[1] looks like this:



Learn more about extracting source code from Org files.

3.2 Hello World Version 1

```
#include <stdio.h>
int main(void)
{
  printf("Hello world\n");
  return 0;
}
```

```
Hello world
```

What happens in 1:

- A header file (`stdio.h`) is included for input/output
- A function (`main`) without arguments (`void`) is defined
- The function returns integer data (`int`)
- A string (`"..."`) is printed out
- A new-line is added at the end (`\n`)
- If successful, the program returns the value `0`

3.3 Hello World Version 2

The program could also have been written much simpler:

- In 1,the function `main` is missing the `void` argument, and the `int` (indicating the type of variable returned - an integer).

```
#include <stdio.h>
main()
{
  printf("Hello world\n");
}
```

```
Hello world
```

3.4 Hello World Version 3

The program could also have been written more complicated:

- `int argc` is an integer, or single number - the number of arguments that were passed to `main`
- `char **argv` (or `char *argv[]`) is a *pointer* that refers to an array of characters - a more complicated data structure.

```
#include <stdio.h>

int main(int argc, char **argv)
{
  printf("hello world\n");
  return 0;
}
```

```
hello world
```

4 Compiler workflow

The machine cannot process `hello.c` without help. It must

Preprocess	**The preprocessor acts on lines beginning with `#`**
Compile	The compiler translates instructions into object code
Link	The linker combines object code and functions like `printf()`
Run	The final `*.exe` program is a binary (machine) program
Debug	The debugger controls rule violations along the way

I compiled the `hello.c` program on a Linux box - the executable is called `hello.out`. The other binary is `hello.exe` compiled on Windows. Compare the two executables - what do you notice?

```
-rwxrwxrwx 1 marcus marcus 48432 Dec 29 08:38 hello.exe
-rwxrwxrwx 1 marcus marcus 16696 Dec 29 12:28 hello.out
```

Question: are these executables portable?[2]

5 Shell execution

- You can also save the code in a C source code file `hello.c`
- Instead of Emacs, you could use the simple `nano` editor (*try it now*)

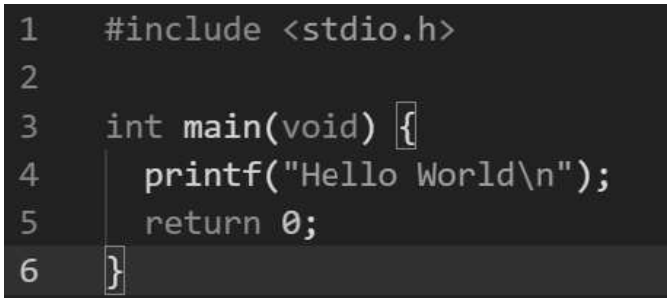
- On Windows: compile it on the Windows command line or in the Emacs shell:

COMMAND	ACTION
`C-x C-f hello.c`	Create C file `hello.c`
	Copy block or write code anew in `hello.c`
`C-x C-s`	Save `hello.c`
`M-x eshell`	start a command line shell in an Emacs buffer
`M-x shell`	start a command line shell in an Emacs buffer
`gcc hello.c -o hello`	compile program and create executable
`ls -l hello*`	list files - you should see `hello`, `hello.c`
`/.hello`	execute program

- The *eshell* is an Emacs Lisp simulation of a Linux shell (`bash`)
- On Windows, `PowerShell` would also work (run with `./pun[.exe]`)

6 Syntax highlighting

- Notice the slight syntax highlighting difference to an online REPL `repl.it`[3]:



- There is no highlighting standard - you should experiment with different themes[4]

- Display line numbers with `display-line-numbers-mode`, and highlight lines with `hl-line-mode`[5] - you can toggle these, and you can go through the minibuffer history with `M-x M-p` and `M-n`:

```
1
2   #include <stdio.h>
3
4   int main(void)
5   {
6     printf("To C, or not to C: that is the question.\n");
7     return 0;
8   }
```

7 Comments

Forgetting to terminate a *comment* may cause the compiler to ignore part of your program - but both syntax highlighting and auto-indent in the editor will tip you off:

```
printf("My "); /* forgot to close this comment ...
                  printf("cat ");
                  printf("has ");  /* so it ends here */
printf("fleas");
```

```
My fleas
```

Let's fix this:

```
printf("My "); /* forgot to close this comment */
printf("cat ");
printf("has ");  /* so it ends here */
printf("fleas");
```

```
My cat has fleas
```

8 Let's practice!

Go to the Org-mode practice file and complete the first few exercises:

1. understand and change syntax highlighting
2. understanding and using comments in C

9 Variable types and declarations

- C computes using placeholders, or **variables**
- 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, see 1:

```
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[6]: you must tell the computer first, how much memory you'll need.

10 Variable assignment

- A variable gets its value through **assignment**

- In 1, 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 1, you would get an error. Can you see why?[7]
- [ ]

Example 1 would throw another error. What's wrong?[8]

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

- [ ]

  Code block 1 finally works, that is, it compiles and runs.

  What does the code actually do?

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

- A *string literal* assigned to a `float` variable contains a decimal point and the letter `f`, as shown in 1.

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

- Assigning a `float` to an `int` (as in 1) and vice versa (1) is possible (but not safe as we will see).

```
float profit;
int iProfit;
iProfit = 2150.48;
```

```
float profit;
int iProfit;
profit = 2150;
```

- [ ]

  Variables with values can be used to compute other values, as shown in 1

  How many things does this little program have to do[9]?

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

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

- You can also initiate and declare variables at once. In 1, the `volume` from before is computed inside `printf`.

```
int height = 8, length = 12, width = 10;

printf("Volume: %d", height * length * width);
```

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

11 Formatting printout

11.1 `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`[10].
- the standard input/output library `stdio.h` also contains `putchar()`, which prints a character to the screen.

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

11.2 Formatting integer numbers

- In the code 1, `%d` is a format specifier for an `int`:

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

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

11.3 Formatting floating-point numbers

- In 1, the format specifier `%f` is used to print a `float`.

```
float profit; // type declaration profit = 2150.48f; // variable
profit = 2150.f;

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

11.4 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

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

- 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!

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

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

```
The profit is: $2150.479980
```

11.5 Formatting errors

- Bad things happen when you get the formatting wrong.
- In 1, we print a `float` first correctly, then with the wrong format identifier, and then the other way around.

```
float foo;  // defined float
foo = 3.14f;   // assigned float
printf("float as float: %.2f\n",foo);  // formatted float as float
printf("float as int: %d\n",foo);  // formatted float as int

int bar;  // defined int
bar = 314;   // assigned int
printf("int as int: %d\n",bar);  // formatted int as int
printf("int as float: %.2f\n",bar);  // formatted int as float
```

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

12 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"[11].
- `[ ]`

  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 ". What does 1 need to compile?

```
volume = 12 * 10 * 8
weight = volume / 166
```

- `[ ]`

  Fixed the errors in the block 1 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;
```

- `[ ]`

  The code in 1 prints the result of the computation.

```
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
```

- 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[12] as shown in 1.

```
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
```

- [ ]

  Now for the final program 1.

```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: %dx%dx%d\n", length, width, height);
printf("Volume (cubic inches): %d\n", volume);
printf("Dimensional weight (pounds): %d\n", weight);
```

13 Let's practice!

Go to the Org-mode practice file and complete the second batch of exercises:

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

14 Constants

14.1 Macro definition with `#define`

- If you don't want a value to change, you can define a `constant`. There are different ways of doing that. The code in 1 shows a declarative constant definition for the pre-processor that blindly substitutes the value everywhere in the program. This is also called a **macro definition**.

```
#define PI 3.141593
printf("PI is %f\n",PI);
```

- `[ ]`

  Can you see what mistake I made in 1?

```
#define PI = 3.141593
printf("PI is %f\n", PI);
```

  Instead of "`3.141593`", the expression "`= 3.141593`" is substituted for `PI` everywhere - the program will not compile.

- `[ ]`

  Can you see what went wrong in 1 ? If you don't see it at once, check the compiler error output!

```
#define PI 3.141593;
printf("PI is %f\n", PI);
```

- It's easy to make mistakes with user-defined constants. For one thing, "constants" declared with `#define` can be redefined (so they aren't really constant at all).
- `[ ]`

  The program 1 demonstrates how a constant declared with `#define` can be redefined later with a second `#define` declaration.

```
#define WERT 1.0
printf("Constant is %.2f\n", WERT);

#define WERT 2.0
printf("Constant is %.2f\n", WERT);
```

```
Constant is 1.00
Constant is 2.00
```

14.2 Library definitions with `#include`

- Since mathematical constants are so important in scientific computing, there is a library that contains them, `math.h`. In 1, it is included at the start to give us the value of Pi as the constant `M_PI` with much greater precision than before.

```
printf("PI is %f\n",M_PI);
printf("PI is %.16f\n",M_PI);
```

```
PI is 3.141593
PI is 3.1415926535897931
```

- If you write source code outside of Emacs Org-mode, you have to include this library file explicitly like this:

```
#include <math.h>
```

- Here is more information on C header files and on how `#include` works.

- In Linux, `math.h` and the other header files sit in `/usr/include/`. The screenshot shows the math constant section of `math.h`.

```
/* Some useful constants.  */
#if defined __USE_MISC || defined __USE_XOPEN
# define M_E            2.7182818284590452354   /* e */
# define M_LOG2E        1.4426950408889634074   /* log_2 e */
# define M_LOG10E       0.43429448190325182765  /* log_10 e */
# define M_LN2          0.69314718055994530942  /* log_e 2 */
# define M_LN10         2.30258509299404568402  /* log_e 10 */
# define M_PI           3.14159265358979323846  /* pi */
# define M_PI_2         1.57079632679489661923  /* pi/2 */
# define M_PI_4         0.78539816339744830962  /* pi/4 */
# define M_1_PI         0.31830988618379067154  /* 1/pi */
# define M_2_PI         0.63661977236758134308  /* 2/pi */
# define M_2_SQRTPI     1.12837916709551257390  /* 2/sqrt(pi) */
# define M_SQRT2        1.41421356237309504880  /* sqrt(2) */
# define M_SQRT1_2      0.70710678118654752440  /* 1/sqrt(2) */
#endif
```

Figure 8: Mathematical constants in /usr/include/math.h

- [ ]

  Where is `math.h` in Windows[13]? Where in MacOS? Find the file, open and look at it in Emacs (the file is read-only).

```
#include <math.h>
#define e M_E
printf("%.16f\n", e);
```

14.3 Type definition with `const`

- Modern C has the `const` identifier to protect constants. 1 shows an example. Here, `double` is a higher precision floating point number type.

```
const double TAXRATE_CONST = 0.175f;
double revenue = 200.0f;
double tax;

tax = revenue * TAXRATE_CONST;

printf("Tax on revenue %.2f is %.2f", revenue, tax);
```

- [ ]

  What happens if you try to redefine the constant `taxrate` after the type declaration? Modify 1 accordingly and run it.

```
const double TAXRATE_CONST = 0.175f;
double revenue = 200.0f;
double tax;

TAXRATE_CONST = 0.2f;
tax = revenue * TAXRATE_CONST;

printf("Tax on revenue %.2f is %.2f", revenue, tax);
```

15 Reading input

- Before you can print output with `printf`, you need to tell the computer, which format it should prepare for.
- Just like `printf`, the input function `scanf` needs to know what format the input data will come in, otherwise it will print nonsense (or rather, memory fragments from God knows where).

- The following statement reads an `int` value and stores it in the variable `i`.[14]. The input comes from the file `./data/input`.

```
int i;
puts("Enter an integer!");
scanf("%d", &i);
printf("You entered %d\n", i);
```

- To input a floating-point (`float`) variable, you need to specify the format with `%f` both in the `scanf` and in the `printf` statement. We'll learn more about format specifiers soon.

16 Naming identifiers

16.1 Naming conventions

(The code blocks in this section are all silent - will give no output - but because they're only snippets, they will not compile.)

- Use upper case letters for CONSTANTS

```
const double TAXRATE;
```

- Use lower case letters for variables

```
int tax;
```

- Use lower case letters for function names

```
hello();
```

- If names consist of more than one word, separate with _ or insert capital letters:

```
hello_world();
helloWorld();
```

- Name according to function! In 1, both functions are identical from the point of view of the compiler, but one can be understood, the other one cannot.

```
const int SERVICE_CHARGE;
int v;

int myfunc(int z) {
  int t;
  t = z + v;
  return t;
}

int calculate_grand_total(int subtotal) {
  int grand_total;
  grand_total = subtotal + SERVICE_CHARGE;
  return grand_total;
}
```

16.2 Naming rules

- What about rules? The compiler will tell you if one of your names is a mistake! However, why waste the time, and the rules are interesting, too, at least syntactically, to a nerd.

- Names are sensitive towards spelling and capitalization: `helloWorld` is different from `HELLOWORLD` or `Helloworld`. Confusingly, you could use all three in the same program, and the compiler would distinguish them.

- Names cannot begin with a number, and they may not contain dashes/minus signs. These are all illegal:

```
10times  get-net-char
```

  These are good:

```
times10    get_next_char
```

- There is no limit to the length of an identifier, so this name, presumably by a German programmer, is okay:

```
Voreingenommenheit_bedeutet_bias_auf_Deutsch
```

- The keywords in the table have special significance to the compiler and cannot be used as identifiers:

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

- Your turn: name some illegal identifiers and see what the compiler says!

17 Program Layout

- You can think of a program statement as a series of tokens[15]:

```
printf ( "Height: %d\n"   ,   height )  ;
  1    2        3         2     5    6  7
```

	TOKEN	MEANING
1	identifier	protected C keyword (function)
2	punctuation	function call begins
3	string literal	text + formatting + escape character
4	punctuation	separator
5	identifier	integer variable
6	punctuation	function call ends

	TOKEN	MEANING
7	punctuation	statement closure

- You can have any amount of white (empty) space between program tokens (this is not so for all programming languages[16]).
- [ ]

  As an example, here is a version of `dweight.c` that works just as well, on one line, with almost all whitespace deleted. Only in one place, the space is needed. Can you see where?

```
int height,length,width,volume,weight;height=8;length=12;width=10;volume=height*length*wid
```

- Another exception are the preprocessor directives - they need to be on a line of their own[17].

```
#include <stdio.h>
#define  CONSTANT 5
```

- You can divide statements over any number of lines as long as you don't divide keywords or tokens. This works:

```
int
height
= 5
  ;
printf
(
 "height %d\n" ,
 height)
;
```

- But this does not:

```
int
hei ght
= 5
  ;
print f
(
 "height
 %d\n" ,
 height)
  ;
```

  - The variable `height` is not declared
  - The `printf` function is not recognized
  - The string literal is not complete
- Good practice:
  - Space between tokens makes identification easier
  - Indentation makes nesting easier to spot

    - Blank lines can divide a program into logical units
- `[ ]`

Practice: improve the layout of this program (get it from GDrive), then run it.

```
int var1=1;int var2;var2=
                    var1
                    *100;
printf (      "Variable1=%d,variable2=%d\n",
              var1,

              var2
              );
```

18 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

19 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

20 Glossary

CONCEPT	EXPLANATION
Compiler	translates source code to object code

CONCEPT	EXPLANATION
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;`

21 References

- Collingbourne (2019). The Little Book of C (Rev. 1.2). Dark Neon.
- King (2008). C Programming - A Modern Approach. Norton. Online: knking.com.

Footnotes:

[1] In our case, instead of weaving TeX files (`.tex`) to print, we weave Markdown files (`.md`), or WORD (`*.odt`) files, or we dispense with the weaving altogether because Org-mode files (equivalent of the `*.w` or "web" files) look fine on GitHub. GitHub.

[2] Executables are the result of compilation for a specific computer architecture and OS. The `.exe` program was compiled for Windows, the `.out` program was compiled for Linux. They will only run on these OS.

[3] replit.com is an online Read-Eval-Print-Loop (REPL) that looks like a Linux installation (in fact, it is a so-called Docker container, an emulated, customized Linux installation). When registering (for free) you can use many different programming languages - here is a link to my container.

[4] You can find different themes for GNU Emacs here, and install them using `M-x package-list-packages`. To see the differences, enter `M-x custom-themes` and pick another theme now. You can save it automatically for future sessions.

[5] If you always want to have line numbers and highlight the line under the cursor, put these lines in your `.emacs` file: and restart Emacs:

```
;; always display line numbers
(global-display-line-numbers-mode)
;; enable global highlighting
(global-hl-line-mode 1)
```

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

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

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

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

[10] See here for a comparison of `printf()` vs. `puts()`.

[11]

> "Cargo space has physical limits based on the volume of the cargo and the weight. The reason why both volume & weight are evaluated can be better understood if you consider the cost of shipping a large object with less weight.
>
> For example, a large box containing styrofoam cups weighs very less, i.e., the dimensional (volume) weight of that box will likely be more than its actual weight. It is for this reason that most airlines and other transport providers evaluate both dimensional weight & actual weight, and then use the greater of the two weights to bill you for the transportation costs. The greater of the two weights is also commonly referred to as ‘chargeable weight’." (UniRelo 2020)

[12] 165/166 is 0.9939759, so we've just messed with the actual volume.

[13] If you installed the MinGW compiler (GCC for Windows), look for it in the MinGW directory - there's an `/include` subdirectory that contains many header/library files `.h`.

[14] 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.

[15] 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, auto-coding and machine translation (like Google translate).

[16] 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.

[17] The `<..>` brackets indicate that the file in between the brackets can be found in the system `PATH`. If a local file is included, use double apostrophes `"..."`.

Date: 2022-05-26 Thu 00:00
Author: Marcus Birkenkrahe
Created: 2022-05-27 Fri 12:41