

C Basics

CSC100 Introduction to programming in C/C++

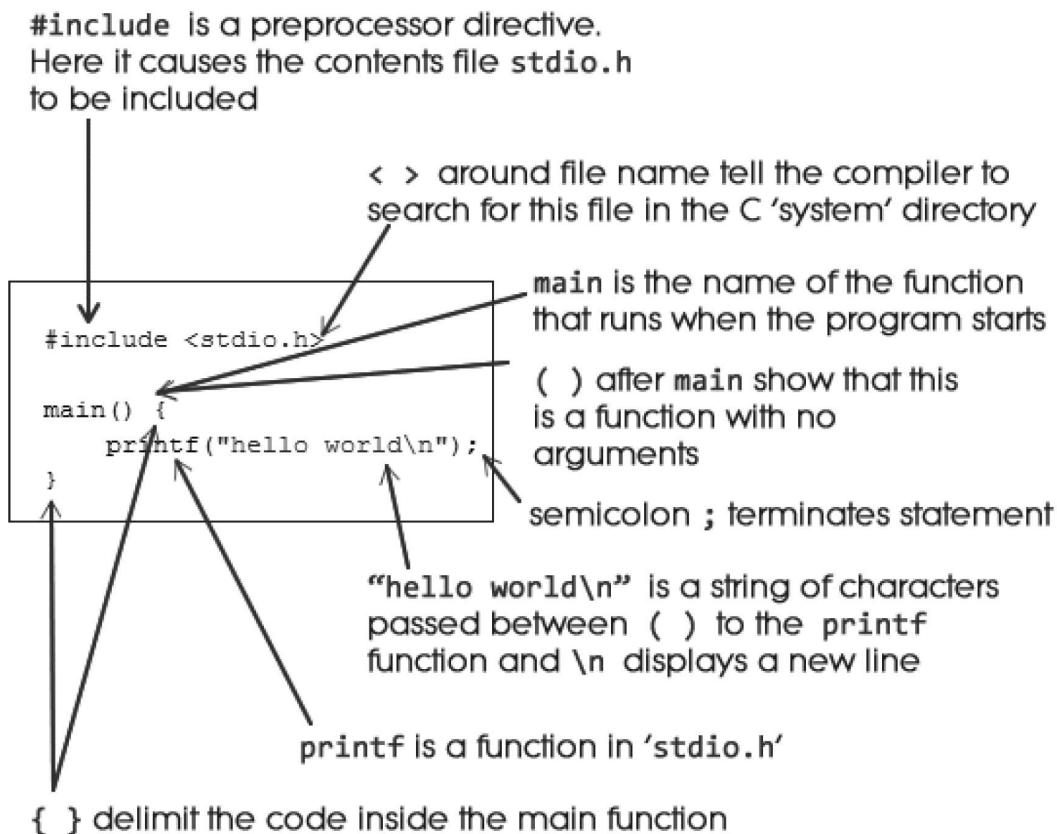
README

- This script summarizes and adds to the treatment by King (2008), chapter 2, C Fundamentals - see also [slides \(GDrive\)](#):
- You should type along in your own Org-mode file if you can. To make this easier, use the auto expansion (`<s>`), add the following two lines at the top of your file, and activate each line with `c-c c-c` (you should get a confirmation in the minibuffer):

```
#+PROPERTY: header-args:C :main yes
#+PROPERTY: header-args:C :includes <stdio.h>
```

- Remember that `C-M-\` inside a code block indents syntactically
- You will find more class content in the `agenda.org` and `notes.org` files.

Program structure



(Image source: Collingbourne, 2017)

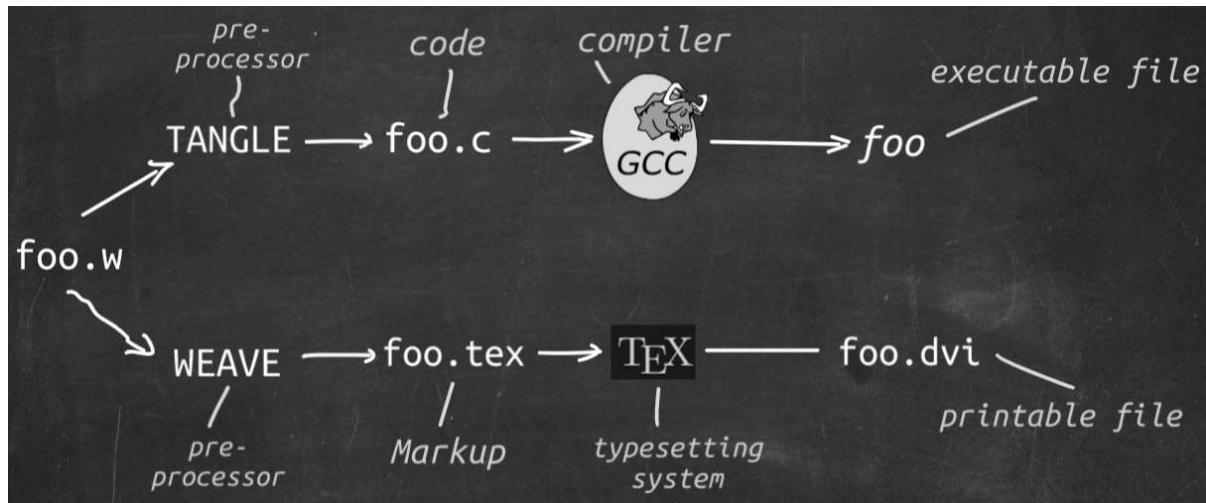
Hello world program

"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 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.](#)

Hello World Version 1

- What happens here:
 - 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`

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

Hello world

Hello World Version 2

The program could also have been written much simpler:

- This function `main` is missing the `void` argument, and the `int` (indicating the type of variable returned - an integer).
- Alas, in Org-mode, this program will not compile, though outside of Org-mode, it will (with a warning). Try it!

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

Hello world

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

Compiler workflow

The machine cannot process `pun.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`. 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
```

Challenge: are these executables portable?²

Shell execution

- You can also save the code in a program `pun.c`
- Compile it on the Windows command line or in the Emacs shell:

COMMAND	ACTION
C-x C-f <code>pun.c</code>	Create C file <code>pun.c</code>
	Copy block or write code anew in <code>pun.c</code>
C-x C-s	Save <code>pun.c</code>
M-x eshell	start a command line shell in an Emacs buffer
<code>gcc -o pun pun.c</code>	compile program and create executable
<code>ls</code>	list files - you should see <code>pun.exe</code>
<code>pun</code>	execute program
<ul style="list-style-type: none"> ◦ The shell is an Emacs Lisp simulation of a Linux shell (<code>bash</code>) ◦ Windows PowerShell would also work (run with <code>./pun[.exe]</code>) 	

Syntax highlighting

- Notice the slight syntax highlighting difference to `repl.it`

```

1 #include <stdio.h>
2
3 int main(void) {
4     printf("Hello World\n");
5     return 0;
6 }
```

- There is no highlighting standard - you should experiment with different themes³
- Display line numbers with `display-line-numbers-mode`, and highlight lines with `hl-line-mode`⁴:

```

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 }
```

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

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
- The block with **declarations** comes before the statements⁵

Variable assignment

- A variable gets its value through **assignment**
- In 1, the variable `height` gets the value `8`. It is called a **constant** because it cannot change.

```
height = 8;
```

- [X] If you tried to run 1, you got an error. Can you see why?⁶
- [X]

However, 1 throws another error. What's wrong this time?⁷

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

- Code block 1 works.

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

- A constant 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` and vice versa is possible but not safe.
- Variables with values can be used to compute other values, as shown in 1.

```
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**

Formatting printout

`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`⁸.

Formatting integer numbers

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

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

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

Formatting floating point numbers

- In 1, `%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
```

Change floating point precision

- By default, %f displays the result with six digits. 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 errors

- What happens 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("%.2f\n", foo); // formatted float as float
printf("%d\n", foo); // formatted float as int

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

Putting it all together (C program)

- 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"⁹.
- [X]

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 run?

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

- [X]

Fixed the errors in 1. The compiler no longer complains, but we don't see anything. How can we print the result?

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

- [X]

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 166¹⁰ 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
```

- [X]

Now for the final program 1. This time, we allow for tangling the program as `dweight.c`.

```
#include <stdio.h>

int main(void)
{
    // 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);

    return 0;
}
```

Constants

Macro definition with `#define`

- If I don't want a value to change, I 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);
```

- So if I mistype my definition, I get errors. Take 1 and introduce an error: in 1, `= 3.141593` is substituted for `PI` everywhere - the program will not compile.

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

- [X]

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. Constants declared with `#define` can be redefined.
- [X]

Write a program that demonstrates how a constant declared with `#define` can be redefined later with a second `#define` declaration. Print out each constant after defining it.

```
#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
```

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 %.18f\n",M_PI);
```

```
PI is 3.141593
PI is 3.141592653589793100
```

- 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 6: Mathematical constants in /usr/include/math.h

- [X] Where is `math.h` in Windows¹¹? Where in MacOS? Find the file, open and look at it in Emacs (the file is read-only).

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);

```

- [X]

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);

```

Naming identifiers

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;
}
```

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!

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`. To run it, you have to tangle the code block (with `c-u c-c c-v t`), compile and run it on the command line - not on the simulated eshell or the Emacs shell, but on the Windows CMD terminal command line¹².

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

- [X] Class assignment:
 - modify the program 1 to read a `float` instead of an `int`
 - tangle it as `fscan.c`
 - compile and run it in the terminal: `gcc -o fscan fscan.c`
- SOLUTION: change `int` to `float` in the declaration, and `%d` to `%f` in the `scanf` function.

```
float i;
puts("Enter a floating point number!");
scanf("%f", &i);
printf("You entered %f\n", i);
```

Program Layout

- You can think of a program statement as a series of tokens¹³:

```
printf ( "Height: %d\n" , height ) ;
```

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
7 punctuation	statement closure

- You can have any amount of white (empty) space between program tokens (this is not so for all programming languages¹⁴).
- [X]

As an example, here is a version of `dweight` 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*width;
```

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

```
#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
- [X]

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
);

```

Formatted I/O

- Interactive notebooks [io_printf.org](#) and [io_scanf.org](#) for self study in GDrive.

Glossary of terms and commands

TERM	EXPLANATION
Variable	Placeholder for a value, e.g. a number
Type	Tells the computer to reserve memory, e.g. <code>int</code> for integer numbers
Type declaration	Combination of type and variable name - e.g. <code>int height;</code>
<code>int</code>	C type for integer numbers, e.g. 2
<code>float</code>	C type for floating point numbers, e.g. 3.14
<code>char</code>	C type for characters, like "joey"
Formatting	Tells the computer how to print, e.g. <code>%d</code> for <code>int</code> types
<code>%d</code>	Format for integers

TERM	EXPLANATION
%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 and functions
stdio.h	Input/Output library, enables printf and scanf
const	Constant identifier, e.g. const double PI = 3.14;

Concept 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

Code summary

CODE	EXPLANATION
#include	directive to include other programs
stdio.h	standard input/output header file (more)
main(void)	main function without argument
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

Jargon

CONCEPT	EXPLANATION
Compiler	translates source code to object code
Linker	translates object code to machine code
Syntax	language rules

CONCEPT	EXPLANATION
Debugger	checks syntax
Directive	starts with #, one line only, no delimiter
Preprocessor	processes directives
Statement	command to be executed, e.g. <code>return</code>
Delimiter	ends a statement (in C: semicolon - ;)
Function	a rule to compute something with arguments

References

- Collingbourne (2019). The Little Book of C (Rev. 1.2). Dark Neon.
- King (2008). C Programming - A Modern Approach. Norton. [Online: knking.com](http://www.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 You can find [themes for GNU Emacs](#) here, and install them using `M-x package-list-packages`.

⁴ 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)
```

⁵ In the C99 standard, declarations don't have to come before statements.

⁶ Assignment is variable use. Variable types must be declared before they can be used.

⁷ The declaration must precede the use of the variable.

⁸ [See here](#) for a comparison of `printf()` vs. `puts()`.

⁹

"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)

10 165/166 is 0.9939759, so we've just messed with the actual volume.

11 Look for it in the MinGW directory - there's an /include subdirectory that contains many header/library files .h.

12 You could try and run it in Emacs. Can you explain the result?

13 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).

14 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.

Author: Marcus Birkenkrahe

Created: 2022-03-11 Fri 09:00

Validate