C FUNDAMENTALS - CONSTANTS / INPUT / NAMING / LAYOUT - LECTURE

CSC100 Introduction to programming in C/C++ Spring 2023

Marcus Birkenkrahe

Time-stamp: *<2023-02-18 Sat 14:05>*

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, download **the raw file** from GitHub and open it in Emacs with:

  ```
  emacs --file 5_constants_codealong.org
  ```

- There is a separate Org-mode file available for **practice after the lecture**. Download **the raw file** from GitHub and open it in Emacs

  ```
  emacs --file 5_constants_practice.org
  ```

2 Constants

- Constants are values that do not change (ever?)

- In C, you can define them with: macros, libraries, or as `const` type
- They have different degrees of permanency

3 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 below 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**[1].

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

```
PI is 3.141593
```

- Can you see what mistake I made in the next code block?[2]

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

- Can you see what went wrong in the next code block? 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 next program demonstrates how a constant declared with `#define` can be redefined later with a second `#define` declaration.

[1] As an aside, "Emacs" was originally named EMACS as an akronym for "Editor MACroS" because of its extensibility through macros - the word comes from the Greek meaning "large" or "prominent", as in "macroscopic" or "macro economy".

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

```
#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
```

- However, `gcc` is warning us about it!

4 Library definitions with `#include`

- Since mathematical constants are so important in scientific computing, there is a library that contains them, `math.h`.
- Below, it is included at the start to give us the value of Pi as the constant `M_PI` with much greater precision than before[3]:

```
#include <stdio.h>
#include <math.h>
#define Donna M_PI   // from now on, M_PI is called Donna
int main(void) {
  printf("PI is %f\n", Donna);
  printf("PI is %.16f\n",Donna);
  return 0;
}
```

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

- Inside Emacs with Org-mode, you can include the math header file `math.h` as a code block header argument:

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

[3]In the tangled `.C` file, you can see that this `#include` statement is inside the `main` bracketed area!

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

- Here is more information on C header files and on how `#include` works. This online tutorial isn't half bad by the way, if you can ignore the flood of ads.
- 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 1: Mathematical constants in /usr/include/math.h

- Where is `math.h` in Windows[4]? Where in MacOS? Find the file, open and look at it in Emacs (the file is read-only).
- In the file, look for `M_PI` (using the incremental search `C-s`). You also find the definition of the Euler number e there. Use it in a `#define` statement to define `e` and print `e` with 16-digit precision.

[4]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`. If you have Cygwin, you'll find it in `c:/Cygwin/usr/include/`.

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

```
2.7182818284590451
```

5 Type definition with `const`

- Modern C has the `const` identifier to protect constants. In the code, `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);
```

```
Tax on revenue 200.00 is 35.00
```

- What happens if you try to redefine the constant `TAXRATE_CONST` after the type declaration? Modify the previous code block 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);
```

6 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`. The input comes from the file `./data/input` [5].

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

  ```
  Enter an integer!
  You entered 1
  ```

- 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.
- To see how input works on the command line, **tangle** the code above as `scanf.c` (add `:tangle scan.c` in the code block header), and run the file on the command line: `C-u C-c C-v t`

  ```
  gcc scan.c -o iscan   ## compiles source code to executable
  iscan < input  ## feed input to the executable
  ```

7 Naming conventions

- Use upper case letters for CONSTANTS

  ```
  const double TAXRATE;
  ```

- Use lower case letters for variables

  ```
  int tax;
  ```

- Use lower case letters for function names

[5] Alas, 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.

```
hello();
```

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

```
hello_world();
helloWorld();  // this is C++ style "camelCase"
```

- Name according to function! In the next code block, 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;
}
```

8 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  // allowed crazy German identifier
```

- 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!

```
int void = 1;
float float = 3.14;
```

- If Windows complains about the app, close the screen dialog to see the debugger:

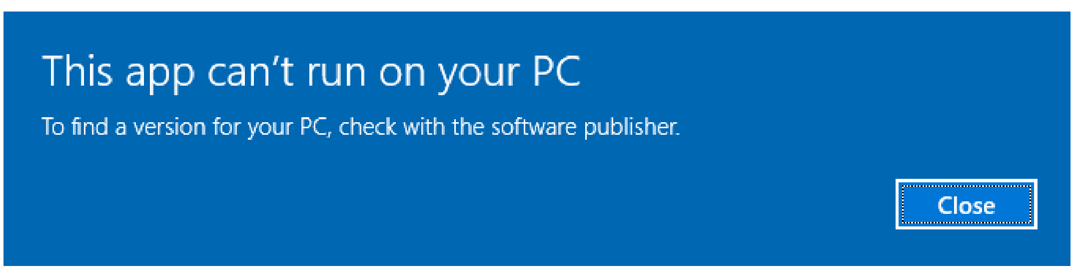

Figure 2: Windows screen dialog

```
▮/Users/BIRKEN~1/AppData/Local/Temp/babel-vrOuI0/C-src-6B45nn.c: In function 'main':
c:/Users/BIRKEN~1/AppData/Local/Temp/babel-vrOuI0/C-src-6B45nn.c:9:5: error: two or more data types in declaration specifiers
    9 | int void = 1;
      |     ^~~~
c:/Users/BIRKEN~1/AppData/Local/Temp/babel-vrOuI0/C-src-6B45nn.c:9:10: error: expected identifier or '(' before '=' token
    9 | int void = 1;
      |          ^
c:/Users/BIRKEN~1/AppData/Local/Temp/babel-vrOuI0/C-src-6B45nn.c:10:7: error: two or more data types in declaration specifiers
   10 | float float = 3.14;
      |       ^~~~~
c:/Users/BIRKEN~1/AppData/Local/Temp/babel-vrOuI0/C-src-6B45nn.c:10:13: error: expected identifier or '(' before '=' token
   10 | float float = 3.14;
      |             ^
[ Babel evaluation exited with code 1 ]
Access is denied.
[ Babel evaluation exited with code 1 ]
2 U\**-  *Org-Babel Error Output*   All L1     (Compilation)
```

Figure 3: Org-babel error output buffer

9 Program Layout

- You can think of a program statement as a series of tokens[6]:

```
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
7	punctuation	statement closure

- You can have any amount of white (empty) space between program tokens (this is not so for all programming languages[7]).
- 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?

[6] 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), and bots like ChatGPT.

[7] 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.

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

```
Dimensions: 12x10x8
Volume (cubic inches): 960
Dimensional weight (pounds): 6
```

- Another exception are the preprocessor directives (beginning with #): they need to be on a line of their own[8].

```
#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)
;
```

```
height 5
```

- But this does not:

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

[8]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 "..".

1. The variable `height` is not declared
2. The `printf` function is not recognized
3. 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 then run it:

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

              var2
              );
```

```
Variable1=1,variable2=100
```

10 Let's practice!

Download the raw Org-mode practice file, complete the second batch of exercises, then upload the completed file to Canvas:

1. Defining constants
2. Standard math library
3. Reading input with `scanf`
4. Naming identifiers
5. Program layout

../img/3_practice1.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. Norton. URL: knking.com.