C FUNDAMENTALS - CONSTANTS

CSC100 Introduction to programming in C/C++ Spring 2025

Marcus Birkenkrahe

February 1, 2025

Contents

1 README

- ☐ Constants with macros or types (`#define`, `const`)
- ☐ Library definitions (`#include`)
- ☐ Reading input from the keyboard

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

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 preprocessor 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);
```

```
PI is 3.141593
```

- Can you see what mistake I made in the next code block?[1]

```
#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!

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

```
#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.

```
#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 (only on the command-line):

```
  wert.c: In function ‘main’:
wert.c:12: warning: "WERT" redefined
   12 | #define WERT 2.0
      |
wert.c:9: note: this is the location of the previous definition
    9 | #define WERT 1.0
      |
```

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:

```
#include <stdio.h>
#include <math.h>    // math functions and definitions
int main(void) {
```

```
  printf("PI is %f\n", M_PI);
  printf("PI is %.16f\n", M_PI);
  return 0;
}

PI is 3.141593
PI is 3.1415926535897931
```

- Do you remember what happens if your precision `p` is greater than the precision delivered by the computer?[2]
- You can redefine the value of any constant using `#define`:

```
#include <stdio.h>
#include <math.h>
#define apple_pie M_PI   // from now on, M_PI is called ‘pie‘
int main(void) {
  printf("PI is %f\n", apple_pie);
  printf("PI is %.16f\n", apple_pie);
  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 (then you don't have to include it explicitly in your code block):

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

PI is 3.141593
PI is 3.1415926535897931
```

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

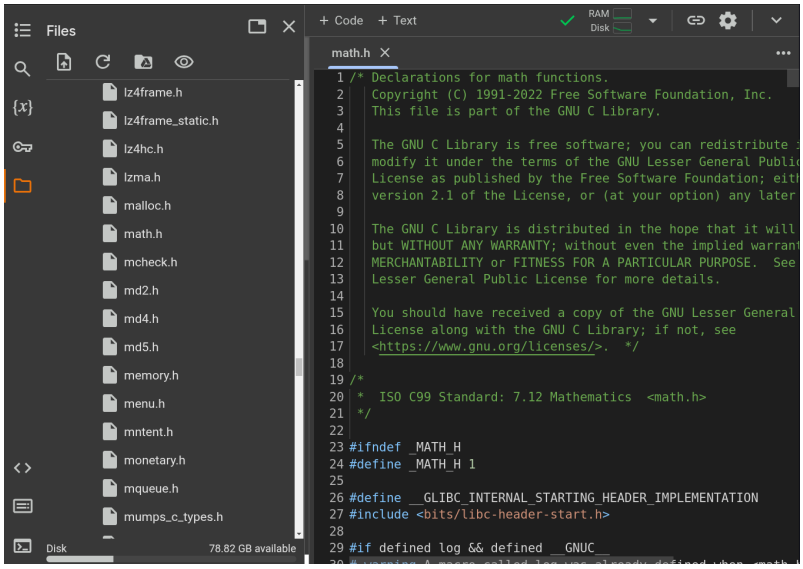
[2] If the formatting precision that you ask for is greater than the precision of the stored constant, the computer will simply make digits up (which is not good).

```
/* 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[3]? Where in MacOS? Find the file, open and look at it in Emacs (the file is read-only).

- In online IDEs like `onecompiler.com`, you can typically not look at header files unless you have access to the command line or the file hierarchy - it does work in Google Colaboratory.



- In the file, look for `M_PI`. You also find the definition of the Euler number e there[4].

- Use it in a `#define` statement to define `e` and print `e` with 16-digit precision, with 15 decimal places:

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

2.718281828459045
```

[3] 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/`. If you have MSYS2, look in `C:\msys64\usr\include`.

[4] Want to know more about this peculiar number `e` that occurs in beautiful formulas like "Euler's identity" ($e^{i\pi} + 1 = 0$? See 3Blue1Brown (2017). I added it to our class YouTube channel.

- It may be that you can do better than that on your computer (mine begins to make numbers up after that even though the constant is defined to a higher accuracy)[5].

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 by adding `TAXRATE_CONST = 0.2f` before the `tax` is computed, 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);
```

[5] This is due to inherent limitations of floating-point representation (IEEE 754 standard): double precision numbes use 64 bits of storage, with 52 bits for the fraction (mantissa), 11 bits for the exponent, and 1 bit for the sign - this allows for 15 to 17 bits of precision.

6 PRACTICE Constants

1. Create a NEW C program and call it `constants.c`.
2. Define the Arkansas sales tax rate (6.5%) as `SALES_TAX_AR` using the `#define` macro.
3. Define the Euler number as `EULER` using `M_E` from `math.h` using `#define`.
4. Define the speed of light as `SPEED_OF_LIGHT` using `const`.
5. Print all three definitions to get the following output using the constants you just defined.

   ```
   The Euler number is: e = 2.7182818285
   The Arkansas sales tax is: 6.5%
   The speed of light is: c = 299792458 m/s
   ```

   Tip: the `%` character is reserved for format specification. To escape it, use `%%` in `printf`.

Program template:

```
// include header files
...
// define constants
...
/* main program */
int main(void)
{
  // print constants
  ...
  return 0;
}
```

Upload your result to Canvas (`In-class practice 4: Constants`)!

6.1 Solution

Onecompiler: `https://onecompiler.com/c/437ukkdbb`

```
/*****************************************************
 * constants.c: print constant values.
 * Input: None
 * Output: Euler number, AR sales tax, speed of light
 * Author: Marcus Birkenkrahe
 * Date: 02/01/2025
 *****************************************************/
#include <stdio.h>
#include <math.h>

// constant definitions
#define SALES_TAX_AR 6.5
#define EULER M_E
const int c = 299792458;

int main()
{
  // print constants
  printf("The Euler number is: e = %.10f\n", EULER);
  printf("The Arkansas sales tax is: %.1f%%\n", SALES_TAX_AR);
  printf("The speed of light is: c = %i m/s\n", c);

  return 0;
}
```

```
The Euler number is: e = 2.7182818285
The Arkansas sales tax is: 6.5%
The speed of light is: c = 299792458 m/s
```

7 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`.

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

Enter an integer!
You entered 0
```

- Test suite:

```
gcc ../src/iscan.c -o iscan
echo 100 | ./iscan

Enter an integer!
You entered 100
```

- 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.

8 PRACTICE Reading input

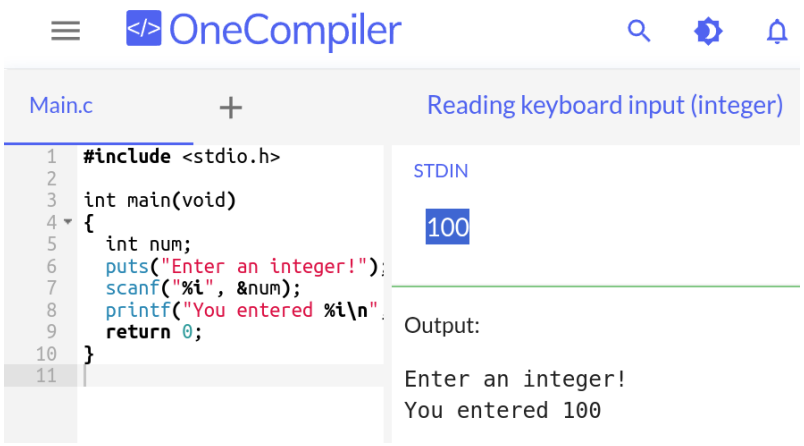
1. Copy the code in your `main` template:

```
#include <stdio.h>

int main(void)
{
  int num;
  puts("Enter an integer!");
  scanf("%i", &num);
  printf("You entered %i\n", num);
  return 0;
}

Enter an integer!
You entered 0
```

2. Run it with an integer input:



3. Modify the program so that it reads a floating-point value instead of an integer. You must make changes on three lines!

4. Test the program with the input (`STDIN`): 3.141593

8.1 Solution

```
#include <stdio.h>

int main(void)
{
  float num;
  puts("Enter a floating-point number!");
  scanf("%f", &num);
  printf("You entered %f\n", num);
  return 0;
}
```

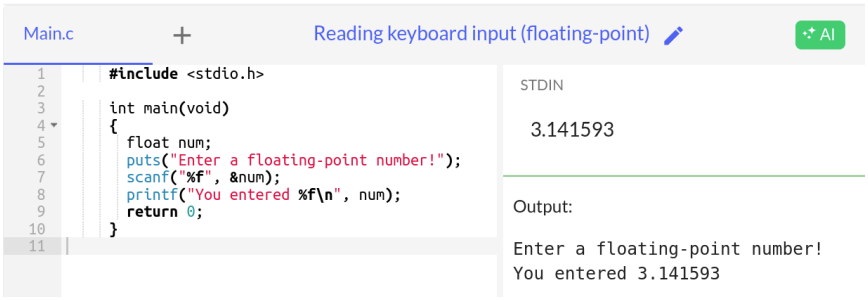
```
Enter a floating-point number!
You entered 0.000000
```

Test suite:

```
gcc ../src/fscan.c -o fscan
echo 3.141593 | ./fscan
```

```
Enter a floating-point number!
You entered 3.141593
```

OneCompiler.com:



9 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

```
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;

// myfunc: [no idea what this does]
// Returns: t (int)
```

```
// Params: z (int)
int myfunc(int z) {
  int t;
  t = z + v;
  return t;
}

// calculate_grand_total
// Returns: grand_total (int)
// Params: subtotal (int)
int calculate_grand_total(int subtotal) {
  int grand_total;
  grand_total = subtotal + SERVICE_CHARGE;
  return grand_total;
}
```

10 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  // Crazy German
  ```

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

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

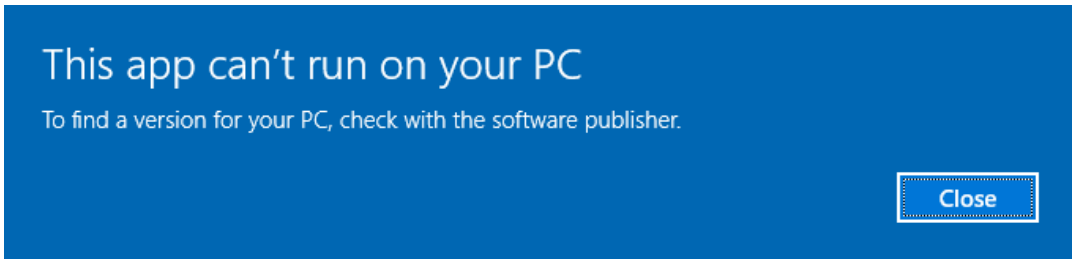


Figure 2: Windows screen dialog

11 PRACTICE Naming identifiers

1. Create a `NEW` file.
2. Copy the code from tinyurl.com/cpp-naming-practice into the `main` program:

```
// integer constant for the speed of light
const int ... = 299792458;
```

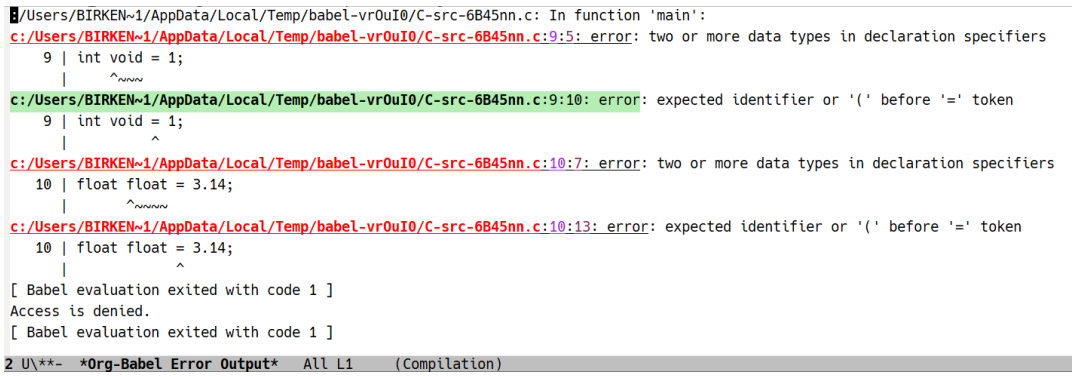


Figure 3: Org-babel error output buffer

```
// floating-point constant for pi
#define ... 3.141593f

// integer variable for volume computations
int ...

// character variable for last names
char ...

// function that adds two integers i and j
int ... (int i,int j) {
  return i + j;
}

// variable whose name contains "my", "next", and "birthday"
int ...
```

3. Complete the code according to the naming rules so that it compiles:

   - Upper case letters for constants
   - Lower case letters for variables and function names
   - Separate names with underscore or insert capital letters
   - Name according to function

Solution in OneCompiler: onecompiler.com/c/437uq28e8

```
// integer constant for the speed of light
  const int c = 299792458;

  // floating-point constant for pi
  #define PI 3.141593f

  // integer variable for volume computations
  int volume;

  // character variable for last names
  char last_name;

  // function that adds two integers i and j
  int add_integers(int i,int j) {
    return i + j;
  }

  // variable whose name contains "my", "next", and "birthday"
  int my_next_birthday;
```

12 Glossary

TERM	EXPLANATION
Constant	Value that does not change during program execution.
Macro definition	Defining constants using `#define` (text substitution).
`#define`	Preprocessor directive to define constants (can be redefined later).
Library constants	Constants provided by standard libraries such as `math.h` (e.g., `M_PI`).
`const`	Keyword in C that defines constants with enforced immutability.
`math.h`	A C standard library header that includes math constants/functions.
`M_PI`	Predefined constant for in `math.h` with high precision.
Redefinition	Attempting to assign a new value to a constant (not with `const`)
`scanf`	Function used to take user input, requiring format specifiers.
Naming conventions	Best practices for naming variables, constants, and functions.
Identifier	A name assigned to variables, constants, or functions in a program.
`printf`	Function used to print formatted output to the console.
`scanf`	Function used to read formatted input from the user.

13 Summary

- Constants in C are values that do not change during program execution.
- They can be defined using `#define` (macro definition), library constants from `math.h`, or the `const` keyword.
- `#define` replaces occurrences of a constant name with a literal value but does not provide type safety and can be redefined.
- Library constants like `M_PI` from `math.h` offer high precision and are predefined in standard headers.
- The `const` keyword ensures immutability and provides type safety.
- Naming best practices:
  - Use **uppercase** for constants.
  - Use **lowercase** for variables and function names.
  - Separate words with underscores (`_`) or use camelCase.
- Identifiers **cannot start with a number** or contain special characters.
- Reserved keywords like `int`, `void`, and `return` **cannot be used as variable names**.
- Constants are essential for input/output operations using `printf` and `scanf`, which require format specifiers.
- Using constants improves **code clarity**, prevents accidental value modifications, and enhances **program stability**.

14 References

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