C Basics

CSC100 Introduction to programming in C/C++

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.
- `[ ]`

  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);
```

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);
```

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

- `[ ]` Where is `math.h` in Windows? 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);
```

- `[ ]`

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.

```c
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!

TODO Practice: which of these names are not allowed?

Author: Marcus Birkenkrahe
Created: 2022-02-18 Fri 09:50
Validate