Enum and typedef

Raul P. Pelaez

October 14, 2025

Contents

1 Introduction

A few lessons before we learned about `struct` and `union`, which are two ways of creating custom data types in C. In this lesson we will learn about two more ways of interacting with types in C: `enum` and `typedef`.

2 Enumerations: Named constants

Info

`enum`
An enumeration is a user-defined type consisting of a set of named integer constants, known as enumerators.

Enumerations are defined using the `enum` keyword, followed by the name of the enumeration and a list of enumerators enclosed in curly braces.

Here is an example of defining an enumeration for the days of the week:

```c
#include <stdbool.h>
enum Day {
    SUNDAY,
    MONDAY,
    TUESDAY,
    WEDNESDAY,
    THURSDAY,
    FRIDAY,
    SATURDAY
};

bool isItMonday(enum Day day) {
    return day == MONDAY;
}
```

Similar as with `struct` and `union`, we can avoid writing `enum Day` every time we want to use the type by using `typedef`:

```
typedef enum {
    SUNDAY,
    MONDAY,
    TUESDAY,
    WEDNESDAY,
    THURSDAY,
    FRIDAY,
    SATURDAY
} Day;

bool isItMonday(Day day) {
    return day == MONDAY;
}
```

In a way, an `enum` is a type that can only take a limited set of (named) values.

2.1 Enums are integers (kinda)

By default, the first enumerator is assigned the value 0, and each subsequent enumerator is assigned a value that is one greater than the previous enumerator. So in the example above, `SUNDAY` is 0, `MONDAY` is 1, and so on. That does not mean that enumerators are of type `int`, but they can be used in any context where an integer is expected.

Here is an example of using enumerators in arithmetic operations:

```
#include <stdio.h>
typedef enum {
  CIRCLE,
  LINE,
  TRIANGLE,
  SQUARE
} Shape;
int main() {
  Shape s = TRIANGLE;
  printf("Shape value: %d\n", s); // Output: Shape value: 2
  printf("Next shape value: %d\n", s + 1); // Output: Next shape value: 3
  return 0;
}
```

Output

```
Shape value: 2
Next shape value: 3
```

We can influence these values slightly by explicitly assigning values to some or all of the enumerators:

```
typedef enum {
    CIRCLE = 0,
    LINE = 1,
    TRIANGLE = 3,
    SQUARE, // 4
    PENTAGON // 5
} Shape;
int main() {
  printf("Number of sides in a triangle: %d\n", TRIANGLE);
  printf("Number of sides in a square: %d\n", SQUARE);
  printf("Number of sides in a pentagon: %d\n", PENTAGON);
  return 0;
}
```

Output

```
Number of sides in a triangle: 3
Number of sides in a square: 4
Number of sides in a pentagon: 5
```

Note how unspecified enumerators continue counting from the last specified value.

Advanced

The underlying type of an enumeration is an implementation-defined integer type that can represent all the enumerator values. This means that the size of an enumeration may vary between different compilers or platforms. However, it is guaranteed that the size of an enumeration is at least as large as the size of an int.

Advice

Usages of enums

The named constant functionality of enumerations makes them useful in several scenarios:

- Encoding states or modes in a program, e.g. the current state of a videogame (`MAINMENU`, `PLAYING`, `PAUSED`, `GAMEOVER`).
- Error codes
- Choices or options, e.g. for a menu, file permissions, etc.
- Bit flags (using powers of two as values) to represent combinations of options.

One thinks of an integer when the value can be any number between -2^{31} and $2^{31} - 1$ (on a typical 32-bit system). One thinks of an enum when the value can only be one of a small set of named values. We could encode the days of the week using an integer (from 0 to 6), but doing so would require us to remember what each number means, and writing a lot of error handling code to ensure that the value is always between 0 and 6. Using an enum makes the code more readable and less error-prone.

2.2 Enums and switch statements

Enums are often used in conjunction with `switch` statements to handle different cases based on the value of an enumeration variable. Here is an example:

```c
#include <stdio.h>
typedef enum {
  RED,
  GREEN,
  BLUE
} Color;
void printColor(Color color) {
  switch (color) {
  case RED:
    printf("Color is Red\n");
    break;
  case GREEN:
    printf("Color is Green\n");
    break;
  case BLUE:
    printf("Color is Blue\n");
    break;
  default:
    printf("Unknown color\n");
    break;
  }
}
int main() {
```

```
  Color myColor = GREEN;
  printColor(myColor);
  return 0;
}
```

Output

```
Color is Green
```

Insight

Enums in the standard library
Despite enums being such a fundamental C feature, the standard library barely uses them. The reason for this is the strong focus on backward compatibility in C. Enumerations where introduced in the C89 standard (called ANSI C), but C had been around since around 1972 without them. Before C89 (pre-ANSI), macro defines were used to define named constants (e.g. errno, file modes, etc). Changing these to enums would have broken a lot of existing code, so the standard library kept using macros for named constants. Nowadays, new libraries and APIs often use enums for named constants, but the C standard library remains mostly unchanged.
The backwards compatibility of C means that code written in the 70s and 80s can still be compiled and run today, which is quite impressive. On the other hand, it also means that the standard library must maintain some outdated practices for the sake of compatibility. Writing in C often means dealing with these legacy issues, keep it in mind during your learning experience and learn to separate the good practices from the "historical baggage".

3 Typedef

Info

`typedef`: type aliases
The `typedef` keyword is used to create a new name (alias) for an existing type.
The syntax is:

```
typedef existing_type new_type_name;
```

For instance, we can create an alias for an unsigned integer type:

```
typedef unsigned int uint;
uint add(uint a, uint b) {
    return a + b;
}
```

Insight

`typedef` is the reason why the "trick" of omitting the name of a struct or enum works. When we do:

```
typedef struct {
    int x;
    int y;
} Point;
```

we are creating an anonymous struct and then using `typedef` to create an alias for it called `Point`.

We can also use `typedef` to create aliases for more complex types, such as pointers or function pointers. Here is an example of creating a type alias for a pointer to a function that takes two integers and returns an integer:

```
#include <stdio.h>

typedef int (*Operation)(int, int);

int add(int a, int b) {
```

```c
    return a + b;
}

int multiply(int a, int b) {
    return a * b;
}

int execute(Operation op, int a, int b) {
    return op(a, b);
}

int main() {
    int sum = execute(add, 5, 3);
    int product = execute(multiply, 5, 3);
    printf("Sum: %d\n", sum);          // Output: Sum: 8
    printf("Product: %d\n", product); // Output: Product: 15
    return 0;
}
```

Output

```
Sum: 8
Product: 15
```

In this example, both `add` and `multiply` are functions that match the signature defined by the `Operation` type alias. The `execute` function takes an `Operation` as a parameter and calls it with the provided integers.

4 Exercises

Goal

Improving the stdlib

There standard library function `strerror` (documentation here) takes `errno` as input and returns a pointer to a string describing the error. The function is defined as:

```c
char *strerror(int errnum);
```

Due to historical reasons `errno` is an integer, and the different error codes are defined as macros (e.g. `#define ENOENT 2`). This means that the user of `strerror` must remember what each error code means, and also that the function can be called with any integer, even if it does not correspond to a valid error code.

Let us modernize this API using enums and typedefs. Your task is to:

1. Define an enumeration called `ErrorCode` that includes at least the following error codes:
   - `MACCES` (Permission denied)
   - `MNOMEM` (Out of memory)
   - `MINVAL` (Invalid argument)

   The `M` prefix is to avoid name clashes with the existing macros.
2. Write a function called `getErrorMessage` that takes an `ErrorCode` as input and returns a pointer to a string describing the error. Use a switch statement to return the appropriate message for each error code.
3. Write a main function that demonstrates the usage of the `ErrorCode` enum and the `getErrorMessage` function by printing the error messages for the defined error codes.

Milestone

Answer the following questions (you may do so by writing comments in your code):

1. You do not need to handle invalid error codes in your `getErrorMessage` function. Why?
2. This is an usage example of `strerror`:

```c
#include <stdio.h>
#include <string.h>
#include <errno.h>
int main() {
    FILE *file = fopen("nonexistent.txt", "r");
    if (!file) {
        printf("Error opening file: %s\n", strerror(errno));
        return 1;
    }
    fclose(file);
    return 0;
}
```

Output

```
Error opening file: No such file or directory
```

Can you explain why, despite `strerror` returning a `char *`, we do not need to free the returned string? The pointer is not marked as `const`, so it seems that we could modify the string, but that would be a bad idea. Why?

hint

- Look up the documentation for `strerror` to see how it manages memory for the returned string. Try using `man` in the terminal, or go to cppref.

Goal

Custom logging

Imagine that a component (say a function) of your program needs to deal with logging, but you want to keep the logging implementation separate from the rest of the code. One solution is to take as argument a function pointer that will be called to log messages. This way, the component does not need to know how logging is implemented, it just calls the provided function, and the user of the component can provide any logging implementation they want.

The signature of the logging function is:

```c
void logMessage(const char *message, LogLevel level);
```

The signature of the component function should be:

```c
void performTask(Logger logger);
```

1. Write an enumeration called `LogLevel` with the levels: `DEBUG`, `INFO`, `WARNING`, and `ERROR`.
2. Define a type alias called `Logger` for a pointer to a function that matches the signature of the logging function.
3. Implement the `performTask` function that takes a `Logger` as argument and calls it to log a message (e.g. "Task started", "Task completed", etc).
4. Write two different logging functions:
   - `consoleLogger`: logs messages to the console using `printf`.
   - `fileLogger`: logs messages to a file called "log.txt". Make sure to open the file in append mode and close it after writing the message.
5. Write a main function that demonstrates the usage of `performTask` with both logging implementations.
   Both loggers should include the log level in the output (e.g. "[INFO] Task started").

Advanced milestone

Write a new logger function called `coloredTimestampedConsoleLogger` that prints messages to the console with the following features:

- Each log level should be printed in a different color.
- Each message should be prefixed with a timestamp in the format "[YYYY-MM-DD HH:MM:SS]".

hint

- You can use the <time.h> library to get the current time and format it.
- For coloring the output, you can use ANSI escape codes. For example, to print text in red, you can use the escape code "\033[31m" before the text and "\033[0m" after the text to reset the color.