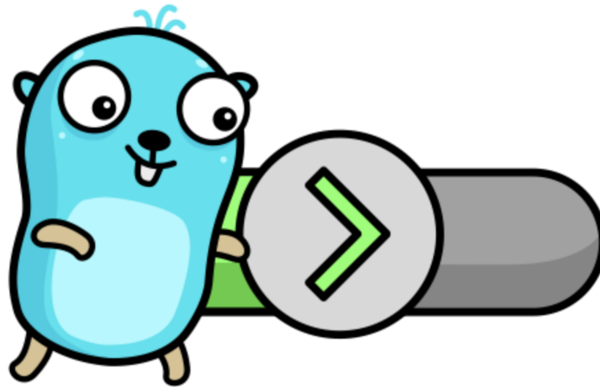
Знакомство с errno_t

В этом уроке мы поговорим о специальном типе `errno_t` и где он может быть полезен.



C11 Annex K: errno_t

errno_t – специальный тип, введённый в стандарте C11 в приложении K и предназначенный для хранения значений (кодов ошибок), которые может принимать переменная `errno`:

```
As a matter of programming style, errno_t may be used as the type of
something
that deals only with the values that might  be found in errno. For
example, a
```

```
function which returns the value of errno might be declared as having
the return type errno_t.
```

Не нужно путать с `errno`, которая является переменной, а не типом!

Это же приложение стандарта предлагает варианты "безопасных" (`_s`) функций, например:

```
errno_t fopen_s(FILE *streamptr, const char *filename, const char
*mode);
```

Откуда мы и можем подсмотреть приёмы построения наших функций, а именно:

- результат функции передаётся через указатель (`NULL` при ошибке);
- ошибка передаётся как возвращаемое значение (`0` при успешной работе);

Допустимым (но менее естественным) является и обратный вариант:

- результат функции передаётся как возвращаемое значение;
- ошибка передаётся через указатель-аргумент функции.

Убивая двух зайцев

Лёгким движением руки мы можем поменять сигнатуру аллокатора из предыдущего урока и убить двух зайцев разом:

- избавиться и от глобальной ошибки;
- избавиться от смеси ошибки и валидного результата в возвращаемом значении:

```
errno_t allocate(int user_id, size_t size, void **mem);
```

Более того, теперь мы можем писать более предсказуемый, читаемый и поддерживаемый код вида

```
errno_t major_func() // Сигнатура функции даёт понять, что возвращается ошибка.
{
    errno_t err = 0;

    if (err = minor_func1())
        return err;

    int val = 0;
    if (err = minor_func2(&val))
```

```
        return err;

    if (err = minor_func3(val))
        return err;

    return 0;
}
```

Недостатки (до боли знакомые нам в Go), с которыми придётся смириться:

- количество кода увеличивается на 30-40%;
- возвращаемую ошибку легко проигнорировать: у функции нет возможности принуждать своих пользователей проверять её;
- если функция аллоцирует ресурсы, то она должна гарантировать, что при возврате ошибки они будут освобождены.

Задача "Аллокатор через errno_t"

Ссылка на заготовку.

Системный программист решил зарефакторить свой аллокатор и переписать его через `errno_t`.

Алгоритм работы функции остаётся прежним:

- функция принимает ID пользователя и количество требуемых байт памяти;
- если пользователь не является администратором (ID не совпадает с `ADMIN`), то функция возвращает ошибку "Operation not permitted".
- если пользователь запрашивает менее чем 1024 байта, то функция возвращает ошибку "Numerical argument out of domain".
- в остальных случаях функция выставляет в принимаемый аргумент указатель на требуемого размера память, выделенную стандартным

способом (`malloc` / `calloc`); при этом если во время выделения памяти произошла ошибка, то следует вернуть её.

Не нужно писать функции ввода/вывода, ваша заготовка будет вставлена в подготовленную функцию `main`.

Stepic компилирует задачи на Си следующей командой:

```
# C (gcc 6.3.0, C99 Standard)

$ gcc -std=c99 -pipe -O2 -static -o main
```

Вывод команды `make check` (из заготовки задачи) для авторского решения:

```
$ make check
allocation was successful for 1024 bytes
allocation error: Operation not permitted
allocation error: Numerical argument out of domain
allocation was successful for 1025 bytes
allocation error: Numerical argument out of domain
allocation was successful for 10000 bytes
allocation error: Numerical result out of range

allocation error: Operation not permitted
```

Sample Input 1:

```
777 1024
```

Sample Output 1:

```
allocation was successful for 1024 bytes
```

Sample Input 2:

```
1 100000000000
```

Sample Output 2:

```
allocation error: Operation not permitted
```

Sample Input 3:

```
777 100
```

Sample Output 3:

```
allocation error: Numerical argument out of domain
```

Sample Input 4:

```
777 1025
```

Sample Output 4:

```
allocation was successful for 1025 bytes
```

Sample Input 5:

```
777 0
```

Sample Output 5:

```
allocation error: Numerical argument out of domain
```

Sample Input 6:

```
777 10000
```

Sample Output 6:

```
allocation was successful for 10000 bytes
```

Sample Input 7:

```
777 1000000000000000000000000000000000
```

Sample Output 7:

```
allocation error: Cannot allocate memory
```

Sample Input 8:

```
1 1
```

Sample Output 8:

```
allocation error: Operation not permitted
```

Напишите программу. Тестируется через stdin → stdout

Time Limit: 5 секунд

Memory Limit: 256 MB

Шаг вперёд и два назад

Использование `errno_t` тоже имеет ряд недостатков.

Во-первых, так как **Приложение К** (C11 Annex K) необязательное, то данный тип может быть не реализован компиляторами в принципе или реализован, но закрыт `define`'ом:

```
#ifndef __STDC_LIB_EXT1__

    typedef int errno_t;
```

```
#endif
```

Мы столкнулись с этим в предыдущей задаче и, чтобы компиляция в Stepik проходила, пришлось объявить тип явно.

Более того, у сообщества есть ряд вопросов к **Приложению K** и оно не является широко поддерживаемым. Мы закрываем на это глаза, так как нам важно обратить внимание на описываемые подходы, определить их применимость для себя и уловить эволюцию относительно Go, а не собрать свод лучших практик по разработке на Си.

Во-вторых, из описания типа следует, что мы можем хранить в нём только ошибки из известного списка (ISO, POSIX)

(исходник примера):

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <errno.h>

#define __STDC_WANT_LIB_EXT1__ 1
// Если define выше не работает для нашего компилятора, то определяем тип руками:
// typedef int errno_t;

const errno_t ESOMETHINGREALLYBAD = 4242;
```

```
errno_t g()
{
    // ...
    int something_really_bad_happens = 1;

    // ...
    if (something_really_bad_happens == 1) {
        return ESOMETHINGREALLYBAD;
    }

    // ...
    return 0;
}

int main()
{
    errno_t err = g();
    if (err != 0) {
        puts(strerror(err)); // Unknown error: 4242
        return EXIT_FAILURE;
    }

    return EXIT_SUCCESS;
}
```

```
$ gcc custom_err.c -o main
```

```
$ ./main
```

```
Unknown error: 4242
```

Код, представленный выше, вызывает следующие замечания:

- мы теряем поддержку `perror` (логично, ведь в `errno` ничего не пишем) и `strerror` ("Unknown error" вместо нормального описания);

- разработчику нужно знать существующие коды `errno`, чтобы своей ошибкой не перетереть системную - соответственно, нужен удобный способ объединять множества своих ошибок и системных.

Более того, возникают вопросы:

- можем ли мы в принципе пользоваться механизмом, предназначенным для нашей операционной системы, а не прикладного кода?
- можем ли мы из своей функции возвращать существующие POSIX-ошибки, хотя пользователь может ожидать, что их источником будет системный вызов, драйвер и т.п.?

Промежуточные итоги

Исходя из предыдущих шагов, можно сделать вывод, что использование `errno_t` подойдёт для обёрток над функциями стандартной библиотеки, системными функциями и другим непользовательским кодом, уже использующим `errno`.

А как решить проблемы, обозначенные ранее, без использования данного типа, мы увидим в следующем уроке.