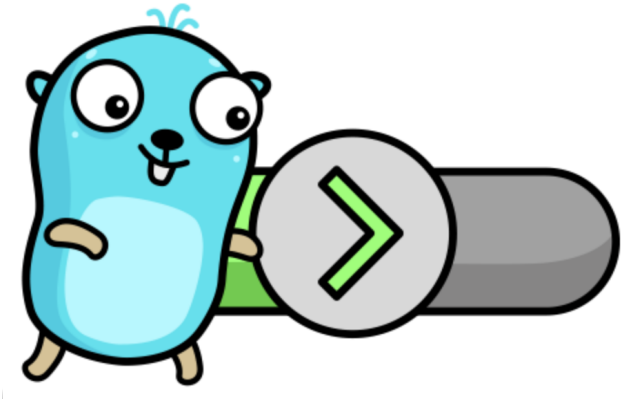
Организация "своих" ошибок

В этом уроке и прокачаем идею `errno_t` и научимся организовывать свои типы для ошибок.

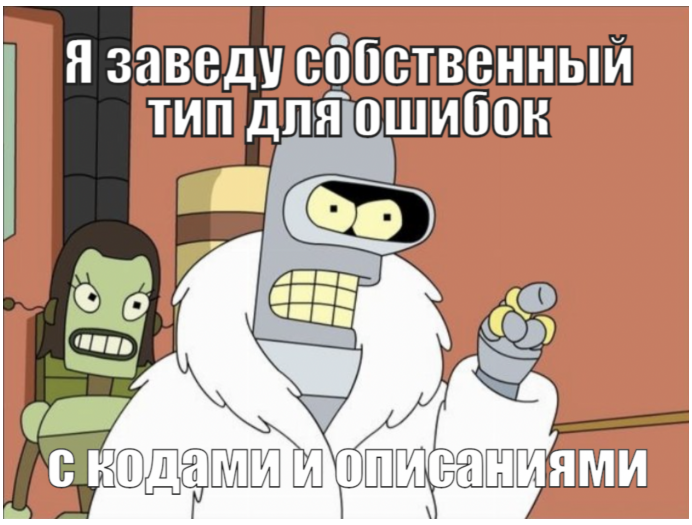


Свой тип для ошибок

Мы поняли, что возвращать ошибки явно, а не выставлять их через глобальный `errno` – достаточно приятная практика.

Но существующий тип `errno_t` не совсем подходит для этих целей, так как он ограничивает нас в использовании собственных кодов ошибок.

Так почему бы нам не завести собственный тип для ошибок и не использовать его?



Ярким примером использования своих кодов ошибок является POSIX функция `gettaddrinfo`:

```
int getaddrinfo(const char *node,
                const char *service,
                const struct addrinfo *hints,

                struct addrinfo **res);
```

Результат передаётся через аргумент `**res`, а возвращаемым значением является одна из документированных ошибок:

```
// ...
EAI_FAIL
       The name server returned a permanent failure indication.

EAI_FAMILY
       The requested address family is not supported.

EAI_MEMORY
       Out of memory.

EAI_NODATA
       The specified network host exists, but does not have any
network addresses defined.

// ...
```

Получить описание ошибки можно с помощью функции `gai_strerror`.

Попробуем написать функцию в такой же концепции только в более строгом виде: вместо `int` определим собственный тип для ошибки.

validation_error_t

В качестве примера напишем функцию `validate`, валидирующую структуру пользователя (например, при регистрации):

```
const int PASS_MIN_LEN = 10;
```

```
typedef struct {
    char *username;
    char *email;
    char *password;
} user_t;

validation_error_t validate(user_t u)
{
    if (strlen(u.password) < PASS_MIN_LEN) {
        return VALID_ERR_WEAK_PASSWORD;
    }
    // ...
    return 0;

}
```

Мы видим, что функция возвращает `validation_error_t` – наш собственный тип, заменяющий `errno_t`:

```
typedef enum {
    VALID_ERR_OK = 0,
    VALID_ERR_INVALID_USERNAME,
    VALID_ERR_INVALID_EMAIL,
    VALID_ERR_WEAK_PASSWORD,
    VALID_ERR_COUNT, // Служебное поле для определения размера enum.

} validation_error_t;
```

При этом из-за того, что `enum` определяет целочисленные значения, мы можем использовать их в качестве индекса массива, что поможет нам при реализации собственного аналога `perror`:

```
const char* const VALIDATION_ERROR_STRS[] =
{
    "Everything is OK",
    "Invalid username",
    "Invalid email",
    "Too weak password",
};

const char* validation_error_str(validation_error_t err)
{
    if (VALID_ERR_OK <= err && err < VALID_ERR_COUNT) {
        return VALIDATION_ERROR_STRS[err];
    }
```

```
    return "Unknown";

}
```

Проверим (исходник примера):

```
// validation.c

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

// Код из выкладок выше.
// ...

int main()
{
    validation_error_t err = validate((user_t){"bob",
"bob@gmail.com", "bob123"});
    if (err != 0) {
        printf("user validation err: %s",
validation_error_str(err));
    }

    return 0;

}
```

```
$ gcc validation.c -o main
$ ./main

user validation err: Too weak password
```

Оборачивание "чужих" ошибок в свои

Если внутри функции есть работа с переменной `errno`, то мы можем "обернуть её" в свою ошибку:

```
#include <errno.h>

extern int errno;

typedef enum {
    // ...
    VALID_ERR_SYSTEM_ERR,
```

```
    // ...
} validation_error_t;

const char* const VALIDATION_ERROR_STRS[] = {
    // ...
    "System err", // Чтобы не нарушить индексацию. Использоваться не
будет.
    // ...
};

const char* validation_error_str(validation_error_t err)
{
    if (err == VALID_ERR_SYSTEM_ERR) {
        return strerror(errno);
    }

    if (VALID_ERR_OK <= err && err < VALID_ERR_COUNT) {
         return VALIDATION_ERROR_STRS[err];
    }
    return "Unknown";
}

validation_error_t validate(user_t u) {
    errno = 0;
    somestdlibfunc(u.email);
    if (errno != 0) {
        return VALID_ERR_SYSTEM_ERR;
    }
    // ...
    return 0;
}
```

Но этот способ имеет "дыры", например, предполагается, что `validation_error_str` будет вызвана сразу же после получения `VALID_ERR_SYSTEM_ERR`, иначе `errno` может перетереться другими вызовами библиотечных функций:

```
const char* validation_error_str(validation_error_t err)
{
    if (err == VALID_ERR_SYSTEM_ERR) {
        return strerror(errno); // Надеемся, что успеем вывести
актуальную errno.
    }
```

```
    // ...
    return "Unknown";

}
```

В принципе так можно оборачивать не только `errno`, но и возвращаемые функциями собственные коды ошибок.

Выше представлен лишь один из возможных вариантов работы с "чужими" ошибками, напоминающий **врапинг ошибок в Go** (запомните это понятие, о нём мы будем говорить позже).

Не хватает только добавления к "чужой" ошибке собственного текста, но средствами Си это сделать не так просто, так как получится микс строк на стеке и в куче, и последние могут привести к утечкам памяти.

KISS

Keep it simple, stupid.

Если же функция предполагает более простое API и достаточно, например, понимать, завершилась она успешно или нет, то можно оставить возвращаемым типом `int` и в случае ошибки возвращать отрицательное значение (в соответствии с общепринятыми практиками в Си).

Главное, чтобы способ индикации был унифицирован на весь проект:

```
int is_valid_user(user_t u)
{
    if (validate(u) == VALID_ERR_OK) {
          return 0;
    }
    return -1;

}
```

Задача "Будни разработчика"

Ссылка на заготовку.

Цель данной задачи – прочувствовать всю боль работы с ошибками в Си, особенно в плане освобождения ресурсов. Мы лишний раз увидим, как далеко Go шагнул вперёд благодаря наличию ключевого слова `defer` и встроенного типа `error`.

Вам необходимо реализовать `get_user_handler` – функцию получения данных о пользователе по запросу:

```
http_error_t get_user_handler(char *request_data, char
**response_data);
```

Обработчик должен состоять из последовательного выполнения трёх операций:

1) Десериализация запроса с помощью `unmarshal_request` (функция уже реализована и доступна вам).

```
typedef struct {
    int user_id;
} request_t;


int unmarshal_request(char *request_data, request_t **request);
```

2) Получение пользователя из базы данных с помощью `db_get_user_by_id` (функция уже реализована и доступна вам).

```
typedef struct {
    int  id;
    char *email;
} user_t;

typedef enum {
    DB_ERR_OK        = 0,
    DB_ERR_INTERNAL  = 1,
```

```
    DB_ERR_NOT_FOUND = 2,
} db_error_t;


db_error_t db_get_user_by_id(int uid, user_t **user);
```

3) Сериализация ответа с помощью `marshal_response` (функция уже реализована и доступна вам).

```
typedef struct {
    user_t *user;
} response_t;


int marshal_response(response_t response, char **response_data);
```

Обращаем внимание, что конечная функция возвращает числовой код ошибки типа `http_error_t`.

Описание возвращаемой ошибки получается тестами в `main` с помощью `http_error_str`.

В следующих ситуациях ожидаются следующие ошибки:

- `unmarshal_request` завершился с ошибкой (`-1`) – ***400 Bad Request***
- `user_id` в запросе меньше или равен нулю – ***422 Unprocessable Entity***
- `db_get_user_by_id` завершился с ошибкой "не найдено в базе" (`DB_ERR_NOT_FOUND`) – ***404 Not Found***
- `db_get_user_by_id` завершился с внутренней ошибкой (`DB_ERR_INTERNAL`) – ***500 Internal Server Error***
- `marshal_response` завершился с ошибкой (`-1`) – ***500 Internal Server Error***

В случае успешного выполнения функции ожидается ***200 OK*** и не `NULL` данные ответа `response_data`.

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
```

Не забывайте при ошибке освобождать полученные ресурсы:

- При этом функции гарантируют отсутствие мусора (в принимаемых аргументах), если они завершаются с ошибкой (т.е. не нужно лишний раз вызывать `free` от незаполненных `NULL`-аргументов).

- Помните, что если структура содержит поля-указатели (и по ним была выделена память), то сначала следует освободить их, а затем уже саму структуру:

```
free(user->email);
free(user);
```

Подсказки:

```
request_t *req = NULL;
```

```
unmarshal_request(request_data, &req);

// ...

user_t *user = NULL;
db_get_user_by_id(req->user_id, &user);

// ...


marshal_response((response_t){user}, response_data);
```

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

```
$ make check
200 OK
{"user": {"id": "4224", "email": "bob@gmail.com"}
422 Unprocessable Entity
400 Bad Request
500 Internal Server Error
404 Not Found
422 Unprocessable Entity
```

Sample Input 1:

```
{"user_id": 4224}
```

Sample Output 1:

```
200 OK
{"user": {"id": "4224", "email": "bob@gmail.com"}
```

Sample Input 2:

```
{"user_id": -100}
```

Sample Output 2:

```
422 Unprocessable Entity
```

Sample Input 3:

```
{"user_id"}
```

Sample Output 3:

```
400 Bad Request
```

Sample Input 4:

```
{"user_id": 10001}
```

Sample Output 4:

```
500 Internal Server Error
```

Sample Input 5:

```
{"user_id": 777}
```

Sample Output 5:

```
404 Not Found
```

Sample Input 6:

```
{"user_id": 0}
```

Sample Output 6:

```
422 Unprocessable Entity
```

Напишите программу. Тестируется через stdin → stdout

Time Limit: 5 секунд

Memory Limit: 256 MB

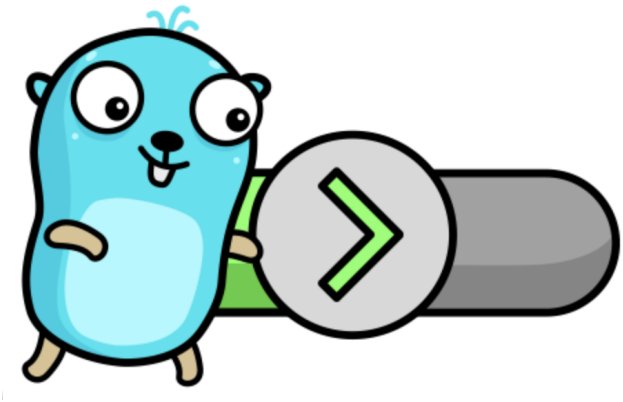
Подведём итоги

За этот модуль мы пробежались по организации ошибок в Си и проделали путь от `errno` до собственного типа для ошибок.

Среди прочего были выявлены следующие проблемы:

- отсутствие единого подхода к работе с ошибками в стандартной библиотеке и прикладном коде;
- отсутствие широкоиспользуемого типа (аналога `errno_t`), показывающего, что функция возвращает ошибку;
- отсутствие удобного способа получить человекочитаемое описание любой ошибки;
- отсутствие удобного способа обогатить ошибку дополнительной информацией;
- отсутствие удобного способа "завернуть" в свою ошибку "чужую";
- проблема освобождения ресурсов при возврате ошибок - вынужденный переход к `goto` или копипаста;
- отсутствие возможности возвращать из функции сразу и результат и индикатор ошибки (как следствие, ещё большее увеличение количества кода).

Запомним пункты выше и посмотрим, какими средствами Golang постарался обойти эти проблемы и попытался стать лучше!



P.S. После выполнения задачи "**Будни разработчика**" программирование на Go вообще должно показаться сказкой :)