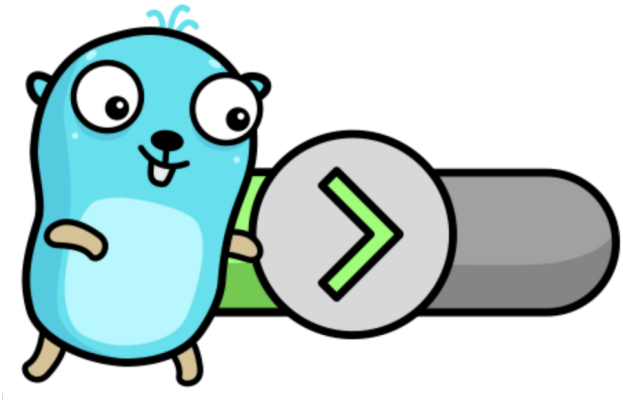
Знакомство с errno

В этом уроке мы познакомимся с глобальным индикатором ошибки, используемым в Си.



Немного истории

Разработка языка Go началась в Google в сентябре 2007 года такими "мастодонтами" как Роберт Гризмер, Роб Пайк и Кен Томпсон. Последний был замечен в создании языка Си и операционной системы UNIX.

Неудивительно, что Golang унаследовал некоторые черты, присущие Си:

- компилируемый тип исполнения;
- статическую строгую типизацию;
- Си-подобный синтаксис (условные конструкции, фигурные скобки, комментарии и пр.):

It must be familiar, roughly C-like. Programmers working at Google are early in their careers and are most familiar with procedural

languages, particularly from the C family. The need to get programmers productive quickly in a new language means that the language cannot be too radical.

В целом Golang может рассматриваться как попытка создать замену языкам Си и C++.

Исходя из этого давайте посмотрим на концепцию обработок ошибок в Си, чтобы почувствовать её эволюцию при переходе к Go.

errno

`errno` – это пример индикатора глобальной ошибки. В каждом потоке существует своя локальная версия `errno`, перед запуском программы равная нулю.

Например, в компиляторе gcc мы видим, что `errno` реализована в виде макроса, разворачивающегося в вызов функции, возвращающей указатель на целочисленный буфер:

```
// ...

/* The error code set by various library functions.  */
extern int *__errno_location (void) __THROW __attribute_const__;
# define errno (*__errno_location ())


// ...
```

Перед вызовом функции из стандартной библиотеки Си (или системного вызова) следует занулить `errno`, а после вызова – проверить её значение: в случае ошибки функция изменит `errno`.

Давайте попробуем открыть несуществующий файл (исходник примера):

```
// unexistent_file.c

#include <stdio.h>
#include <errno.h>

extern int errno;

int main()
{
    FILE *fp;
    fp = fopen("unexistent_file.txt", "r");

    printf("value of errno: %d\n", errno); // value of errno: 2
    return 0;

}
```

```
$ gcc unexistent_file.c -o main
$ ./main
value of errno: 2
```

В `errno` записалось целочисленное значение `2`, но что оно значит?

Значения errno

Стандарт ISO C (7.5 Errors <errorno.h>) определяет следующие ошибки:

- **EDOM** – (**E**rror **dom**ain) ошибка области определения.
- **EILSEQ** – (**E**rror **i**nval**i**d **seq**uence) ошибочная последовательность байтов.
- **ERANGE** – (**E**rror **range**) результат слишком велик.

Негусто, правда?

Прочие коды ошибок определены в стандарте POSIX. Кроме того, в спецификациях стандартных функций обычно указываются используемые ими коды ошибок и их описание.

Давайте подсмотрим в Linux:

```
// uapi/asm-generic/errno-base.h
...
#define    EPERM        1    /* Operation not permitted */
#define    ENOENT       2    /* No such file or directory */
...
#define    EDOM        33    /* Math argument out of domain of func */
...

// uapi/asm-generic/errno.h
...
#include <asm-generic/errno-base.h>

#define    EILSEQ      84    /* Illegal byte sequence */
...
```

Мы видим, что двойке соответствует макрос **ENOENT** "No such file or directory", и именно эту ошибку вернула нам `fopen` при попытке открыть несуществующий файл.

strerror & perror

Для удобства отладки в стандартной библиотеке Си существует функция `strerror`, возвращающая строку, описывающую переданный функции код ошибки:

```
// The strerror() function returns a pointer to a string that
// describes the error code passed in the argument errnum.

char* strerror(int errnum);
```

(Интересно, что сама `strerror` может в процессе своей работы выставить в `errno` код `EINVAL` или `ERANGE`).

Также существует функция `perror`, которая, используя `strerror`, печатает описание текущего значения `errno`, добавляя его к пользовательскому сообщению через пробел и двоеточие (исходник примера):

```
// perror.c

#include <stdio.h>
#include <errno.h>

int main()
{
    FILE *fp;
    fp = fopen("unexistent_file.txt", "r");

    perror("cannot open file"); // cannot open file: No such file or
directory
    return 0;

}
```

```
$ gcc perror.c -o main
$ ./main
cannot open file: No such file or directory
```

Забегая вперёд – формат выводимого сообщения ничего не напоминает? :)

Задача "Аллокатор"

Ссылка на заготовку.

Перед системным программистом встала задача написать функцию-аллокатор, работающую по следующему принципу:

- функция принимает ID пользователя и количество требуемых байт памяти;
- если пользователь не является администратором (ID не совпадает с `ADMIN`), то функция возвращает `NULL` и выставляет `errno` в "Operation not permitted".

- если пользователь запрашивает менее чем 1024 байта, то функция возвращает `NULL` и выставляет `errno` в "Numerical argument out of domain".
- в остальных случаях функция возвращает указатель на требуемого размера память, выделенную стандартным способом (`malloc` / `calloc`); при этом если во время выделения памяти произошла ошибка, то следует оставить значение `errno`, установленное функцией стандартной библиотеки.

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 10000000000000000000000000000000000
```

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

Подводные камни

Наш аллокатор унаследовал ряд "проблем", присущих функциям стандартной библиотеки и связанных с:

- использованием глобальной переменной для хранения ошибки (`errno`);
- в ту же очередь использованием **in-band** индикатора ошибки: возвращаемого значения, которое может интерпретироваться как ошибка, а не как валидное значение.

```
void *p = allocate(uid, size);
if (p == NULL) {
   // Обработка ошибки через errno.
}
```

Рассмотрим данные пункты подробнее.

Чем плох глобальный errno?

Тем же, чем плохи глобальные переменные в принципе. Посмотрим на примеры возможных ошибок.

1) Потеря актуальной ошибки:

```
// ...
mytime_t t = get_current_time(); // Здесь произошла ошибка, но мы забыли проверить.
mysize_t s = get_window_size();  // Здесь errno, выставленная выше, возможно была перетёрта.

// ...
```

2) Потеря актуальной ошибки при желании вывести информацию о ней:

```
errno = 0;

FILE *file1 = fopen(name1, "r");
if (file1 == NULL) {
    perror("cannot open first file");
    exit(EXIT_FAILURE);
}

FILE *file2 = fopen(name2, "r"); // Здесь произошла ошибка.
if (file2 == NULL) {
    // Здесь возможно мы её перетёрли.
    fclose(file1);

    // Здесь возможно мы увидим errno от fclose, а не от fopen второго файла.
    perror("cannot open second file");
    exit(EXIT_FAILURE);
}
```

3) Использование `errno` как индикатор того, что функция завершилась успешно/неуспешно:

```
errno = 0;

// Может быть такое, что библиотечная функция внутри вызвала другую библиотечную функцию,
// которая выставила errno, но не повлияла на результат верхнеуровневой функции.
int result = some_lib_func();
if (errno != 0) {
    // В итоге мы получим валидный результат,
    // но программа завершится, так как errno был выставлен.
    perror("cannot some_lib_func");
    exit(EXIT_FAILURE);
}
```

Строго говоря и в Go мы можем выстрелить себе в ногу путём игнорирования ошибок или их случайного перекрытия (о чём мы будем говорить дальше в курсе), но здесь на помощь приходят статические анализаторы кода.

Чем плох "in-band error indicator"?

Это подход, когда мы используем один "канал" и для передачи значения и для передачи управляющей информации (в данном случае – ошибки).

Вы уже могли заметить, что функции в Си обычно возвращают или:

- валидное значение (не `NULL` указатель, положительное целое число, ноль, свой тип и пр.);
- значение - индикатор ошибки (`NULL` указатель, отрицательное целое число, не ноль, свой тип и пр.).

```
int server_fd = 0;
if ((server_fd = socket(AF_INET, SOCK_STREAM, 0)) == 0) {
    // Обработка ошибки.

}

time_t current_time;
if ((current_time = time(NULL)) == ((time_t)-1)) {
    // Обработка ошибки.

}

FILE *fp = NULL;
if ((fp = fopen("test", "rb+")) == NULL) {
    // Обработка ошибки.

}
```

Мы получаем целый зоопарк вариантов, для грамотной работы с которым приходится прибегать к документации той или иной функции, из чего можно заключить, что подобный подход не удовлетворяет очевидным требованиям к хорошему API, а именно:

- простота использования функции даже без документации;
- сложность использования функции неправильно;
- лёгкость чтения и поддержки кода, использующего функцию.

Промежуточные итоги

Мы познакомились с организацией ошибок в Си через глобальную переменную `errno` и **in-band error indication**, а также обозначили недостатки с этим связанные.

В следующем уроке мы посмотрим, как их можно нивелировать.