В первом модуле мы говорили о том, что такое модульность и абстракция. Напомним, что при построении программ (и вообще любых систем):

- модульность означает, что большая система собирается из более маленьких, как из конструктора;
- абстракция означает, что мы упрощаем описание модуля до его поведения. Внутренняя структура модуля, объясняющая, как он достигает своего поведения, нам не важна.

Эти два принципа позволяют нам строить большие и сложные программы, которые не слишком сложно изменять, развивать и исправлять в них ошибки.

В контексте C модулями обычно называются исходные файлы с расширением .c. Однако с точки зрения построения систем, модулями-кирпичиками программы являются не только исходные файлы, но и функции.

Этот урок мы посвятим некоторым приёмам разделения программы на изолированные части и скрытия внутреннего устройства этих частей.

Объявления и определения

Для некоторых сущностей, которые появляются в программе, разделяются их *объявления* и *определения*.

Объявление ставит компилятор в известность о том, что сущность с таким именем имеется. Например:

```
// Объявление функции (прототип)


char f( int64_t n, int64_t );


// Объявление глобальной переменной


int64_t g;
```

Определение задаёт соответствие между именем сущности и её содержанием.

```
// Определение функции


char f( int64_t n, int64_t z ) { return 0; }
```

```
// Объявление глобальной переменной


int64_t g = 1337;
```

В программе можно иметь множество объявлений для одной и той же сущности, но только одно определение.

Объявить можно и структурный тип:

```
// Объявление структурного типа -- мы такого ещё не видели

// Пока мы не можем создавать экземпляры этого типа

struct list;
```

У этого есть два важных применения.

1. Объявление типов данных, ссылающихся друг на друга. Например, если мы захотим описать типы "пользователь" и "группа пользователей" такие, что "пользователь" хранит ссылку на "группу", а "группа" хранит ссылку на "пользователя"-владельца группы, нам придётся использовать объявления:

```
2. struct user;
3.
4. struct group {
5.     struct user* owner;
6.     struct user* users;
7.     size_t       users_count;
8. };
9.
10.
11. struct user {
12.     const char*    name;
13.     struct group* in_group;
    };
```

14. Более общо, пример показывает, что с объявлением структуры можно создавать указатели на неё даже если компилятор не знает её определения (и может и не узнать

в текущем файле). Это используется для создания *непрозрачных типов*, о чём мы тоже узнаем в этом уроке.

Ключевое слово static

Модульная система в C

В терминах языка C модулем называется файл с расширением .c; модуль это *единица трансляции,* то есть компилятор работает с каждым модулем по-отдельности, а затем компонует из скомпилированных модулей один результирующий файл.

Напомним, как объявления и определения используются в программах из множества файлов.

Если в файле `lib.c` функция `mul` определена, а в другом файле `main.c` она используется, то в `main.c` необходимо написать объявление функции. Это объявление обычно попадает туда из заголовочного файла, включённого с помощью директивы `#include`, например:

lib.c

```
/* Определения */

int mul( int x, int y
) {
  return x * y;
}

int div( int x, int y
) {
  return x / y;
}
```

lib.h

```
/* Объявления */

int mul( int x, int y
);
```

main.c

```
#include "lib.h"

/* lib.h:
int mul( int x, int y );
*/

int main() {

   printf("%d\n", mul( 40, 2 )
);

   return 0;
}
```

Функции и глобальные переменные выставляются наружу из каждого файла. Другие файлы могут к ним "подсоединиться", объявив переменные с теми же именами и типами или функции с теми же именами и типами аргументов.

Модули в C недостаточно изолированы

Взаимодействие с модулем обычно происходит через небольшое количество выделенных функций, для которых фиксируются типы и смысл их аргументов; эти функции составляют *интерфейс* модуля. Преимущества выделения интерфейса такие:

- Часть модуля, не входящая в интерфейс, "прячется" от программиста и не загружает его мозг лишней информацией. Достаточно знать интерфейс модуля и представлять его поведение, а про его внутреннюю структуру можно не думать.
- Другие модули не вызывают напрямую функции, не входящие в интерфейс модуля, поэтому их можно легко изменять.
- Часть модуля, которую невозможно использовать из других модулей, всегда будет неиспользованной напрямую, а значит легко переписываемой.

Как видите, в наших интересах:

- минимизировать интерфейс модуля
- запретить использовать извне все функции кроме интерфейса.

Для описания интерфейса принято использовать заголовочные файлы. В них объявляются только те функции, которые нужны другим модулям. Однако по описанию модульной системы видно, что имеющимися средствами мы не можем закрыть модуль полностью. Программист всегда может вписать прототип функции, имеющейся в модуле, но не включённой в заголовочный файл, и получить к ней доступ:

lib.c	**lib.h**	**main.c**

`/* Определения */`
 	
`int mul( int x, int y ) {`
`  return x * y;`
`}`
	`/* Объявления */`
	
	`int mul( int x, int y );`
		`#include "lib.h"`
		
		`/* Хотя div отсутствует в заголовочном файле,`
		`мы можем её вызывать */`

lib.c	lib.h	main.c

`int div( int x, int y ) {`
 	
`  return x / y;`
 	
`}`
		`int div( int x, int y );`
		
		`int main() {`
		
		`  printf("%d\n", div( 40, 2 ) );`
		
		`  return 0;`
		
		`}`

Поэтому наши усилия по продумыванию и выделению интерфейса могу оказаться бессмысленными, если другие программисты смогут легко вытащить любую "скрытую" функцию из модуля.

Улучшение изоляции

К счастью, в C есть ключевое слово `static`. Оно полностью закрывает доступ к функции или глобальной переменной из других модулей.

lib.c	lib.h	main.c

`/* Определения */`
 	
`int mul( int x, int y ) {`
 	
`  return x * y;`
 	
`}`
	`/* Объявления */`
	
	`int mul( int x, int y );`
		`#include "lib.h"`
		
		`/* Это объявление "новой" функциии div`
		
		`не связыванной с функцией div в lib.c */`

lib.c	lib.h	main.c

`static int div( int x, int y ) {`
 	
`  return x / y;`
 	
`}`
		`int div( int x, int y );`
		
		`int main() {`
		
		`// ошибка: функция div не определена`
		
		`  printf("%d\n", div( 40, 2 ) );`
		
		`  return 0;`
		
		`}`

Две полезные привычки

Благодаря этим привычкам вы получаете массу преимуществ:

1. **Все, все, все функции, кроме вызываемых в других файлах, должны быть помечены `static`.**
   - Функции, помеченные `static`, не могут вызываться из других файлов. Компиляторы могут встраивать любые функции в момент вызова, но `static`-функции они могут ещё и **удалить из файлов с кодом вообще**, если везде, где они вызывались, они встроились. С обычными функциями так не сделать: они всегда могут вызываться "где-то в неизвестном нам ещё файле".
   - Функции, помеченные `static`, не видны в других файлах. Значит разные IDE с автоподстановкой точно не будут вам предлагать использовать static-функции в других файлах, и автоподстановкой будет удобнее пользоваться.
   - В большом проекте программисты никогда не могут охватить умом весь объём кода; чтобы использовать функциональность уже написанного кода они его изучают, просматривают функции в поисках подходящей для них. Часто мы пишем некрасивый код внутри модулей на скорую руку чтобы потом сделать "как надо" -- всё равно эти функции, по нашей задумке, извне модуля никем не будут вызываться. Однако если их **возможно** вызвать значит **вероятно, что кто-то из команды их всё-таки станет использовать.** Как только это происходит, исправить код становится гораздо труднее, так как от него уже зависит другой модуль, и вскоре код превращается в кашу (см. спагетти-код).
   - Можно иметь static-функции с одинаковым именем в разных файлах, так как каждая видна только в своём файле.

2. **Все, все, все переменные, кроме тех, которые вы точно изменяете, должны быть помечены `const`.**
    - Помогает компилятору чаще обнаруживать ошибки в программах
    - Даёт компилятору больше информации для оптимизаций, чтобы ваша программа работала быстрее безо всяких усилий с вашей стороны.

Непрозрачные типы

В примере с объявлениями структурных типов мы использовали указатель на структурный тип, чьё определение мы не знали:

```
struct user;


struct group {
   struct user* owner;    // мы не знаем, что внутри struct user
   struct user* users;
   size_t       users_count;
};
```

Мы предоставляли определение `struct user` дальше в программе. Если не предоставлять его вообще, мы сможем работать с данными типа `struct user` только по указателю:

```
struct user* puser;  // OK
struct user  myuser; // ошибка


void f(struct user* puser) { ... };  // OK
void f(struct user  user) { ... };  // ошибка
```

Более того, мы не сможем напрямую обращаться к полям `struct user`, потому что мы не знаем, какие они. Указатель на такие данные можно передавать между функциями, но нельзя создавать переменные такого типа.

Для примера опишем непрозрачный тип-массив.

Файл с кодом:

```
/* array.c */

#include <malloc.h>
```

```c
#include "array.h"

// в .c файле нужно указать полное описание структуры
struct array_int {
  int64_t *data;
  size_t count;
};


// Нужно уметь создавать и инициализировать этот тип
// Самый простой вариант -- выделять его в куче

struct array_int* array_int_create(size_t count) {
  struct array_int* a = malloc(sizeof(struct array_int));
  a->data = malloc(sizeof(int64_t) * sz);
  a->count = count;
  return a;
}

void array_int_destroy(struct array_int *a) {
  free( a-> data );
  free( a );
}

// указатель на элемент массива
int64_t* array_int_at(struct array_int* a, size_t i) {
```

```
  if (i >= a->count) { return NULL; }
  return a->data + i;
}

// Доступ к полям напрямую возможен только в этом файле
// Для других файлов нужно сделать функции для доступа к полям.
// Не обязательно давать доступ ко всем полям
size_t array_count(struct array_int const* a) {
  return a-> count;
}
```

Заголовочный файл:

```
/* array.h */
#include <malloc.h>
#include <stdio.h>
#include <stdint.h>

#include "array.h"

// в .h файле нужно только объявить структуру но не определять её
struct array_int;

// функции, указанные в .h файле,
// должны принимать struct array_int строго по указателю

struct array_int* array_int_create(size_t);
```

```
void array_int_destroy(struct array_int *a);


int64_t* array_int_at(struct array_int* a, size_t i);


size_t array_count(struct array_int const* a);
```

Преимущества непрозрачных типов

- Всё взаимодействие с данными, включая доступ к полям, теперь происходит через функции, которые мы явно предоставили. Другое взаимодействие невозможно.
- Внутреннее устройство типа скрыто, поэтому реализацию можно свободно менять, и это не отразится на остальной части программы.

Например, мы хотим изменить структуру так, чтобы она хранила размер массива не в элементах, а в байтах. Имея непрозрачный тип мы можем поменять `array_size` так, чтобы пользователь не заметил разницы:

```
/* array.c */


//...


// теперь size хранит размер в байтах
struct array_int {
  int64_t *data;
  size_t size;
};


//...


// array_count теперь не просто возвращает значение поля, он делает расчёт.
// Для тех, кто использует структуру array_int извне, не изменилось ничего.
size_t array_count(struct array_int const *a) {
```

```
  return a-> size / sizeof( int64_t );
}
```

Если бы другие части программы могли обратиться к полю `count` (ставшему `size`) напрямую, то после изменения его значения они бы тихо сломались.

Упражнение. Допишите предложенный заголовочный файл для модуля, реализующего стек поверх массива:

```
/* stack.c */


struct stack_int {
  item *items;
  size_t maxcount;
  size_t count;
};


static const size_t STACK_LIMIT = 128;


struct stack_int *stack_int_create() {
  struct stack_int *const result = malloc(sizeof(struct stack_int));
  *result =
      (struct stack_int){malloc(STACK_LIMIT * sizeof(item)), STACK_LIMIT, 0};
  return result;
}


void stack_int_destroy(struct stack_int *s) { free(s->items); free(s); }


bool stack_int_empty(struct stack_int const *s) { return s->count == 0; }
```

```
bool stack_int_full(struct stack_int const *s) {
  return s->maxcount == s->count;
}

bool stack_int_push(struct stack_int *s, item i) {
  if (stack_int_full(s)) {
    return false;
  }
  s->count = s->count + 1;
  s->items[s->count] = i;
  return true;
}

struct maybe_item stack_int_pop(struct stack_int *s) {
  if (stack_int_empty(s)) {
    return none_int;
  }
  const struct maybe_item result = some_int(s->items[s->count].value);
  s->count = s->count - 1;
  return result;
}

static void stack_int_foreach(struct stack_int const* s, void (f) (item)) {
  for (size_t i = 0; i < s->count; i = i + 1) {
    f(s->items[i]);
  }
```

```
}

static void print_int64(item i) { printf("%" STACK_ITEM_PRI "\n", i.value); }


void stack_int_print(struct stack_int const* s) {

  stack_int_foreach( s, print_int64 );

}
```

1task – program

В предыдущем задании вы описали интерфейс модуля с помощью заголовочного файла к нему, скрыв детали реализации главного типа данных – стека.

```
// Почему тут используется typedef:

// https://stepik.org/lesson/499140/step/12

typedef struct {

  int64_t value;

} item;


#define STACK_ITEM_PRI PRId64


struct maybe_item {

  bool valid;

  item value;

};


static const struct maybe_item none_int = {0, {0}};


static struct maybe_item some_int(int64_t value) {

  return (struct maybe_item){true, {value}};
```

```
}


struct stack_int;


struct stack_int *stack_int_create();


void stack_int_destroy(struct stack_int *s);


bool stack_int_empty(struct stack_int const *s);

bool stack_int_full(struct stack_int const *s);


bool               stack_int_push(struct stack_int *s, item i);

struct maybe_item stack_int_pop(struct stack_int *s);


void stack_int_print(struct stack_int const *s);
```

С таким описанием интерфейс стека **полностью отделён от его реализации**. Мы можем изменить реализацию стека на любую другую.

В одном из прошлых уроков мы реализовывали стек поверх массива ограниченной вместимости; в этот раз мы сделаем альтернативный вариант, поверх связного списка. При этом добавление в стек = добавление в начало связного списка; удаление из стека = удаление первого элемента из списка. Такой стек не переполняется, но как правило работает медленнее массива.

2task – program

Интерфейсы

Интерфейс модуля может включать в себя три типа сущностей:

- Функции, доступные извне (не помеченные `static`);
- Глобальные переменные, доступные извне (не помеченные `static`);
- Типы данных.

Язык С не предоставляет языковых конструкций для полного описания интерфейса модуля. Однако с помощью структур и указателей можно описать функции, входящие в интерфейс модуля, явно. Опишем, например, интерфейс простейшего модуля для работы с массивом, внутри заголовочного файла:

```
typedef struct { int64_t value;} item;


struct array_int;


struct array_int* array_int_create( size_t sz         );
void              array_int_destroy(struct array_int*);


item*             array_int_at(struct array_int* a, size_t i);


// экземпляр структуры типа array_interface совмещённый
// с определением типа


static struct array_interface {


  struct array_int* (*create)  (size_t);
  item*             (*at)      (struct array_int*, size_t i);
  void              (*destroy)(struct array_int*);


} const array = {
 array_int_create,
 array_int_at,
 array_int_destroy
};
```

Важно, что структура `array` помечена `static` и `const`. Это означает, что в каждом файле, подключившим этот заголовок, будет своя независимая копия структуры. Это позволит компилятору оптимизировать вызовы через `array`:

```
// Благодаря оптимизациям эти строчки приведут

// к генерации одинаковых машинных инструкций



struct array_int* a = array.create( 10 );

struct array_int* a = array_create( 10 );
```

С помощью такого описания можно ограниченно эмулировать пространства имён. Можно делать и вложенные пространства имён:

```
static struct array_interface {

  struct array_int_interface {

  struct array_int *(*create)(size_t);

  item *(*at)(struct array_int *, size_t i);

  void (*print)(struct array_int *);

  void (*destroy)(struct array_int *);

  } int64;

} const array_int = {

  {

    array_int_create,

    array_int_at,

    array_int_print,

    array_int_destroy

  }

};
```

```
...
```

```
array.int64.print( ... )
```

В частности, можно сделать временный псевдоним для более удобного доступа к функциям.

```
struct array_int_interface const arr = array.int64;
```

```
struct array_int* a = arr.create( 10 );
```

```
arr.at(a, 4)-> value = 42;
arr.print( a );
arr.destroy( a );
```

Напоминаем, как выглядит интерфейс для модуля, реализующего стек.

```
typedef struct {
  int64_t value;
} item;
```

```
#define STACK_ITEM_PRI PRId64
```

```
struct maybe_item {
  bool valid;
  item value;
};
```

```
static const struct maybe_item none_int = {0, {0}};
```

```
static struct maybe_item some_int(int64_t value) {
  return (struct maybe_item){true, {value}};
}

struct stack_int;

struct stack_int *stack_int_create();

void stack_int_destroy(struct stack_int *s);

bool stack_int_empty(struct stack_int const *s);
bool stack_int_full(struct stack_int const *s);

bool              stack_int_push(struct stack_int *s, item i);
struct maybe_item stack_int_pop(struct stack_int *s);

void stack_int_print(struct stack_int const *s);
```

Определите структуру (пометив её const и static), описывающую интерфейс этого модуля и позволяющую писать, например, `stack.int64.full( ... )`.

3task – program

Программисты не так часто прибегают к описанию интерфейса модуля внутри структуры для эмуляции пространст имён. К сожалению, типы таким образом не описать, только функции. Поэтому для типов всё равно приходится всегда писать длинные префиксы.

С помощью `define` и других директив препроцессора можно немного улучшить этот механизм и внедрить поддержку пространст имён и для типов, при этом код будет легче читать, но очень сложно отлаживать и поддерживать. В дальнейшем мы покажем некоторые полезные трюки.

Ваши коллеги, скорее всего, не поймут вашу особую, препроцессорную магию.

Старайтесь писать так, чтобы вы сами через полгода, в три часа ночи, с больной головой могли понять свой код.

Другим применением для структур, содержащих указатели на функции, является организация полиморфизма, а именно подтипов. В C нет классов и механизма наследования, но можно организовать структуры так, чтобы можно было передавать экземпляр дочернего класса вместо экземпляра класса-родителя; при этом "методы" класса-потомка могут перегружать методы класса-родителя.

Для этого в структуры придётся добавить указатель на метаданные, содержащие таблицы виртуальных функций. Благодаря им во время выполнения программы можно будет всегда узнать тип структуры и вызвать правильную версию метода.

Мы посвятим достижению полиморфизма в C отдельный урок.

Блоки кода и локальные переменные

Сейчас для нас все переменные являются или глобальными, или локальными для функции. Однако на деле локальные переменные принадлежат не функции, а блоку кода.

Определение

Блоком кода называется набор statement'ов, заключённый в фигурные скобки:

```
void f() {

   char x;

   { // начало блока



       printf("Hello");



   }  // конец блока




} // конец функции
```

Сам блок тоже является statement'ом. Тело функции всегда является блоком.

Видимость переменных

Переменные, объявленные внутри блока, не видны вне блока.

```
void f() {

    {
        int64_t x = 42;


        printf("%" PRId64 , x );
    }


// ошибка: x не виден после конца блока, в котором он объявлен
    printf("%" PRId64 , x );


}
```

Привычные нам локальные переменные, объявленные в блоке тела функции, являются частным случаем этого правила.

Иерархия блоков

Блоки можно вкладывать один в другой; при этом переменные с одинаковым именем начинают *перекрывать* друг друга:

```
void f() {

    int64_t x = 100;
    {
        // До конца этого блока по имени `x` мы будем обращаться к этой переменной:
        int64_t x = 42;
```

```
        // выведет 42
        printf("%" PRId64 , x );
    }



    // выведет 100
    printf("%" PRId64 , x );



}
```

Минимизация видимости

Прячьте переменные так глубоко в блоки, как только можете. Чем в меньшем количестве строчек кода знают о том, что переменная существует, тем лучше. Например, в таком коде лучше переместить переменную `a` внутрь тела цикла, ведь вне цикла она не используется; также не нужно сохранять её состояние между итерациями.

```
void g( int64_t* x ) { ... }
bool condition() { ... }

void f() {

    int64_t a;

    if (condition()) {
```

```
void g( int64_t* x ) { ... }
bool condition() { ... }

void f() {




    if (condition()) {
        int64_t a;
```

`g( &a );`
`printf("%" PRId64 "\n", a );`
`}`
`}`
	`g( &a );`
	`printf("%" PRId64 "\n", a );`
	`}`
	`}`

Это помогает в поиске ошибок, ведь если в строчке нашёлся баг, то нужно изучить все переменные, которые эта строчка может использовать. Эта задача не так проста, ведь в строчке могут вызываться функции, которые вызывают ещё сотни других функций:

```
// глобальная переменная


int64_t x;


...

/* не посмотрев на код f, g и всех функций, которые они вызывают, мы не знаем, меняется ли в этой строчке переменная x */

printf( "%" PRId64 , f() + g() );
```

Глобальные переменные видны отовсюду, поэтому при ошибке почти в любой строчке они попадают под подозрение: их может изменить любая часть программы.

Когда мы пишем маленькие функции, в них, как правило, немного вложенных блоков. Если уровней вложенности много (скажем, три вложенных цикла), это почти всегда значит, что функцию можно разбить на несколько функций меньшего размера.

Когда можно опускать фигурные скобки

Когда блок состоит из одного statement'а, скобки можно опустить. Это касается прежде всего веток if'а и тела цикла.

Со скобками	**Без скобок**

`void array_int_print(`
`int64_t* array,`
`size_t size) {`
	`void array_int_print(`
	`int64_t* array,`
	`size_t size) {`

Со скобками	Без скобок
`for( size_t i = 0; i < size; i = i + 1 ) { printf("%" PRId64 "\n", array[i] ); } }`	`for( size_t i = 0; i < size; i = i + ) printf("%" PRId64 "\n", array[i] ) }`

Тело функции всегда заключается в фигурные скобки.

Автор курса любит опускать скобки, потому что это делает код визуально легче. Однако у этого есть несколько негативных последствий:

- При добавлении statement'ов в тело цикла мы вынуждены будем добавить и фигурные скобки тоже. При разработке программ мы пользуемся системами контроля версий (например, Git), которые показывают историю изменения файла. С точки зрения того, как изменения накапливаются и как их можно комбинировать, лучше иметь тело цикла внутри блока.
  Тогда при изменении тела цикла скобки не будут затронуты.
- Мозг склонен думать, что визуальная структура кода определяет его значение; если несколько строчек выровнены одинаково, мозг объединяет их в одно целое. Однако для программы тело цикла это всегда один statement или блок в фигурных скобках.

```
void array_int_print(
   int64_t* array,
   size_t size) {

   for( size_t i = 0; i < size; i = i + 1 )
      printf("%" PRId64 "\n", array[i] );
      printf("end of loop");                  // эта строчка выполнится один раз, *после* цикла


}
```

- Код выглядит единообразно, когда все тела циклов и веток if находятся внутри скобок, даже состоящие из одного statement'а.

"В индустрии" у каждого проекта есть соглашения оформления кода. Они отвечают на такие вопросы, как "стоит ли переносить строку перед открывающей фигурной скобкой {". Многие из этих правил скорее вопрос вкуса и привычки, но важно, чтобы весь проект был оформлен в одном стиле, иначе читать код становится тяжело (а программисты читают код гораздо чаще и больше, чем пишут).

Упражнение. Эта функция по задумке должна:

- для положительных чисел вывести делители числа от 1 до его самого, затем однократно вывести символ "$".
- для остальных чисел вывести `No`.

Исправьте ошибку в функции.

Sample Input:

```
4 8 9
```

Sample Output:

```
1 2 4 $1 2 4 8 $1 3 9 $
```

4task – program