

Раздел «Язык Си» . CoffeExtern :

- Проект
 - Сборка проекта
 - Функция из другого файла
 - Зачем делать функции static
 - Переменная из другого файла
 - И extern, и объявление
 - static переменная
 - Где определять макросы?
 - Структуры и typedef
 - Итого что где определяем
 - Заголовочные файлы
 - Условная компиляция

Проект

Когда задача слишком большая, чтобы ее код было удобно писать в 1 файле, разбивают этот файл на несколько и организуют проект.

Для организации проекта в разных системах разработки нужно делать разные действия. Мы предполагаем, что пишем файлы в текстовом редакторе и компилируем из командной строки компилятором gcc.

Сборка проекта

Пусть наш проект состоит из файлов a.c и b.c. Мы хотим собрать его в hello.exe.

Для сборки проекта с помощью gcc из командной строки нужно:

```
gcc -Wall -o hello.exe a.c b.c
```

Для запуска собранной программы нужно запустить hello.exe.

Или, если нас устраивает имя исполняемого модуля по умолчанию (a.out), то собираем проект и запускаем программу командами:

```
gcc -Wall a.c b.c  
./a.out
```

Функция из другого файла

Пусть в файле a.c определена функция void foo(int x). В файле a.c, если хотим вызвать foo(7) до реализации функции, нужно написать прототип функции.

Если хотим вызвать эту функцию в другом файле, то в нем тоже нужно сначала написать прототип этой функции.

```
// a.c  
#include <stdio.h>  
  
void foo(int x);    // прототип функции  
  
void foo(int x) {    // реализация функции  
    printf("a.c foo: x=%d\n", x);  
}
```

и другой файл, где используем эту функцию:

```
// b.c  
#include <stdio.h>  
  
void foo(int x);    // прототип функции, определенной в другом файле
```

Поиск

 Поиск

Раздел «Язык Си»

[Главная](#)
[Зачем учить C?](#)
[Определения](#)

Инструменты:

[Поиск](#)
[Изменения](#)
[Index](#)
[Статистика](#)

Разделы

[Информация](#)

```
int main() {
    foo(7);           // вызов функции, определенной в другом файле
    return 0;
}
```

Соберем и запустим программу. Получим в консоли:

```
a.c foo: x=7
```

static – функция только для этого файла

Если определить функцию с ключевым словом **static**, то она будет видна только в том файле, где ее определили. Увидеть из другого файла ее нельзя:

```
// a.c
#include <stdio.h>

void foo(int x);           // прототип функции foo
static void bzz();         // прототип функции bzz - только для этого файла

void foo(int x) {           // реализация функции foo
    printf("a.c foo: x=%d\n", x);
    bzz();                 // вызов функции bzz
}

static void bzz() {         // реализация функции bzz
    printf("bzz\n");
}
```

и другой файл, где используем эту функцию:

```
// b.c
#include <stdio.h>

void foo(int x);           // прототип функции, определенной в другом файле
static void bzz();         // тут мы сказали, что будет РЕАЛИЗОВАНА в этом же файле функция bzz
                             // warning: 'bzz' declared 'static' but never defined

int main() {
    foo(7);                 // вызов функции, определенной в другом файле

    // bzz();              ОШИБКА: bzz не реализована
    return 0;
}
```

```
$ gcc -Wall a.c b.c
b.c:17:13: warning: 'bzz' used but never defined
    static void bzz();
               ^~~
/tmp/ccHK2wjF.o:b.c:(.text+0x7e): undefined reference to `bzz'
collect2: error: ld returned 1 exit status
```

Если мы закоментируем вызов bzz в main (и заодно прототип этой функции), то программа соберется и при запуске получим:

```
a.c foo: x=7
bzz
```

Зачем делать функции static

Пусть в программе есть два несвязанных куска – работа с сетью и рисование графического интерфейса. Они написаны в двух разных файлах: net.c и gui.c. Над программой работают 2 разных программиста. Один пишет работу с сетью, второй программирует графический интерфейс.

Они оба захотели написать функцию void dump(), которая у одного печатает состояние сетевого соединения, а у другого – текущее положение мыши, фокуса, введенного текста с клавиатуры и тп.

Если они сделают две функции `void dump()`, то при сборке в общую программу возникнет ошибка – линковщик не будет знать при вызове функции `dump()` какую из этих двух функций вызвать. Заметим, что функция нужна только в файле `net.c` или в файле `gui.c`.

Если обе функции будут описаны `static void dump()`, то при вызове из того же файла, линковщик будет знать, какую именно функцию `dump` вызывать (определенную в этом же файле!).

Поэтому функции, которые нужны в одном конкретном файле и не будут нужны вне этого файла принято писать `static`.

Переменная из другого файла

В файле `a.c` определена глобальная переменная `int x`.

Как обратиться к ней в файле `b.c`?

Нужно добавить "прототип" переменной. Т.е. написать объявление переменной с ключевым словом **`extern`**.

```
// a.c
int x = 7;
```

```
// b.c
#include <stdio.h>

extern int x;

int main() {
    printf("x = %d\n", x);
    return 0;
}
```

В чем отличие `int x` и `extern int x`?

`int x`; – это объявление переменной. Создается переменная `x` типа `int`. Выделяется память под эту переменную. В нашем случае `int x = 7`; сразу происходит явная инициализация.

`extern int x`; – НЕ создает переменной. Это обещание компилятору, что переменная `x` объявлена в каком-то файле. Если такая переменная нигде не будет создана, то во время связывания, возникнет ошибка.

И `extern`, и объявление

В файле можно и определить переменную как `extern`, и объявить ее.

```
// a.c
extern int x;

int x = 7;
```

static переменная

Переменная со словом `static`, объявленная вне всяких блоков и функций – НЕ глобальная. Она видна только в данном файле.

Нельзя написать `extern static int x`;

```
// a.c
extern int x;

int x = 7;

static int y = 9;    // 0 by default

void foo() {
    x = 10;
    y = 11;
}
```

```
// b.c
#include <stdio.h>

extern int x;
extern static int y; // ОШИБКА!

int main() {
    printf("x = %d\n", x);
    y = 666;          // ОШИБКА! НИКАК НЕ ПОЛУЧИТСЯ доступить к y
    return 0;
}
```

Где определять макросы?

Если нужно использовать макрос в обоих файлах, то нужно определить его во всех файлах, где собираетесь использовать

```
// a.c
#define N 10
int x;
int a[N];
```

```
// b.c
#define N 10
extern int x;
extern int a[N];
```

Структуры и typedef

Все, что определяет новый тип или новый псевдоним существующего типа, должно быть указано каждом файле.

```
// a.c
struct _D {
    int k;
    char c;
};
typedef struct _D D;

void foo(D d) {
    d.k = 1;
}
```

В b.c недостаточно декларировать будущее определение структуры struct _D. Ее нужно полностью определять, как и typedef:

```
// b.c

// struct _D; - недостаточно!

struct _D {
    int k;
    char c;
};
typedef struct _D D;

void foo(D d); // теперь можно определять прототип функции с D.
```

Итого что где определяем

- **Функция** – нужно написать прототип функции, можно написать прототип в том же файле, где реализуем функцию.
- **Переменная** – объявляем в одном файле, во всех других пишем **extern**, можно написать extern до объявления переменной в том же файле.
- **Макросы** – дублируем везде, где используем.
- **определение структуры, typedef** – дублируем везде, где используем.

Заголовочные файлы

Как удобно дублировать макросы и структуры, описывать прототипы функции и внешние переменные?

Используйте для этого заголовочный файл `a.h` (`h` – от слова `header`).

```
// a.h:
#define N 10
#define prn printf("%s %s %d\n", __FILE__, __FUNCTION__, __LINE__)

struct _D {
    int k;
    char c;
};
typedef struct _D D;

extern int x;
extern int a[10];

void foo(D);
```

Заголовочный файл вставляется с помощью команды препроцессора `#include`

```
// a.c
#include <stdio.h>
#include "a.h"

int main() {
    D d = {1, 2};
    foo(d);
    x = 33;

    return 0;
}
```

Условная компиляция

В `a.h` мы тоже можем написать директивы `#include`. Например, `#include <stdio.h>`

Вдруг мы напишем `b.h`, который включает `a.h`, который включает `b.h`, который включает `a.h`,... Как разорвать круг рекурсивных вставок?

Или файлы `b.h` и `c.h` оба содержат `a.h` и включаются в файл `main.c`. Если в файле `a.h` определена структура или макрос, то будет ошибка о двойном определении.

Нам поможет условная компиляция. Напишите файл `a.h` в таком виде:

```
#ifndef _A_H
#define _A_H
// тут то содержимое файла a.h, которое мы хотели
#endif
```

При первом включении `a.h` макрос `_A_H` не определен, значит код между `#ifndef...` `#endif` будет включен. При выполнении этого кода определяется `_A_H`.

При последующем включении `a.h` `#ifndef _A_H` будет ложно (мы уже определили `_A_H` при первом включении) и код от `#ifndef` до `#endif` будет выброшен.

Т.е. повторного включения не будет.

-- TatyanaDerbysheva - 29 Apr 2018

(с) Материалы раздела "Язык Си" публикуются под лицензией GNU Free Documentation License.