Объявления и определения функций

До настоящего момента мы всегда вызывали функции, которые в тексте программы были определены выше.

Хорошо	Ошибка, не компилируется

`void f() { }`
 	
`void g() { f(); }`
	`void g() { f(); }`
	
	`void f() { }`

Исторически, компиляторы C писались так, чтобы компилировать программу в один проход. Это быстрее и требует меньше памяти. Поэтому когда мы идём по строчкам программы сверху вниз и встречаем вызов функции, о которой не было никаких упоминаний, компилятор не может связать этот вызов с функцией, которая определена позднее.

Сейчас программы как правило компилируются за очень много проходов, так как компилятору необходимо произвести оптимизации, а программа "по пути" претерпевает множество трансформаций. Неоптимизированные программы очень медленно работают, даже будучи написанными на C.

Пока это не вызывало проблем, потому что функции можно было объявлять в удобном порядке. Но что, если две функции друг друга вызывают?

```
void f() { g(); }

void g() { f(); }
```

В каком бы порядке мы ни определили функции `f` и `g`, одна из них будет вызвана ещё до своего определения. Чтобы компилятор знал, что это функция, её можно *объявить* ещё до того, как мы определим её тело полностью.

```
void g();  // Объявление функции выглядит как и определение, только тела нет

           // Объявление также называют "прототип"



void f() { g(); }

void g() { f(); }
```

Если у функции есть аргументы, в объявлении можно опустить их имена и оставить только типы.

```
int f(int x, int, int);  // опустили имена для второго и третьего аргументов
```

Похожий механизм с объявлениями и определениями мы увидим при создании составных типов.

Опишите прототип для функции `sum`, которая принимает два указателя на целые числа и одно число, и ничего не возвращает.

1task – program

Вообще объявления функций существуют чтобы показать компилятору, что функция существует, даже если её определение недоступно. Также это нужно, чтобы определить функцию в одном файле, а вызвать из другого.

Пусть в программе есть два файла: `lib.c` и `main.c`. Функция `sum` определена в файле `lib.c`, мы хотим вызвать её из файла `main.c`. Для этого мы *объявляем* функцию `sum` (или, как ещё говорят, пишем её *прототип*) в файле `main.c`. Теперь мы можем просто вызвать её.

lib.c

```
/* Здесь описана логика работы функции */
int sum( int x, int y ) { return x + y; }

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

main.c

```
/* объявление функции, определённой
   в другом файле, позволяет её
   использовать */
int sum( int x, int y );

/* функция mul нам здесь не нужна */

int main() {

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

   return 0;
```

lib.c	main.c
	`}`

Писать всю программу в одном файле, как правило, очень плохая идея. Это очень быстро приводит к тому, что в программе становится сложно разобраться. Чтобы бороться со сложностью программ есть два важных приёма: *модульность* и *абстракция*; о них – на следующих слайдах

Модульность

Программу лучше разбить на небольшие файлы; каждый из таких файлов с кодом, имеющих расширение `.c`, называется *модулем* и может компилироваться отдельно от других файлов с кодом.

Хорошо, когда мы чётко определяем, какую функцию выполняет тот или иной модуль, и помещаем туда только соответствующий код. Если мы тщательно продумываем этот вопрос, модули начинают помогать нам в поиске ошибок.

Вот как это происходит. Представим программу, которая считывает уравнение, решает его и отправляет решение по сети. Скорее всего в такой программе три модуля:

- один принимает уравнение от пользователя;
- другой занимается подсчётом решения;
- третий отправляет готовое решение по сети.

Допустим, мы обнаружили ошибку: по сети передаётся неправильное решение уравнения.

Из этих трёх модулей скорее всего ошибка находится в модуле, читающем уравнение от пользователя, или в модуле, решающем его. Их можно протестировать отдельно и локализовать ошибку с точностью до модуля. Затем нам останется просмотреть код одного модуля в поисках ошибки.

С другой стороны, если бы вся программа была написана в одном файле-модуле, нам пришлось бы искать ошибку в одном большом файле, а это гораздо сложнее.

Это, конечно, игрушечный пример. Реальные программы порой занимают сотни и тысячи файлов, и там без разбиения на независимые по функциональности модули работать вообще невозможно – слишком высока сложность системы.

Абстракция

Абстракция это принцип проектирования систем который позволяет нам забыть об устройстве части системы и использовать её как чёрный ящик. Для нас важно, как эта часть себя ведёт, а не какие детали внутреннего устройства обуславливают это поведение. Такой частью не обязательно является целый файл, это может быть, например, функция: функции тоже позволяют абстрагироваться от того, как именно получен результат вычислений. Достаточно запустить функцию с нужными аргументами, а затем работать со значением, которое она вернула.

Чаще всего об абстракции в C думают на уровне файлов с кодом. Для каждого файла, скажем, `lib.c`, в котором находятся функции, которые нужно вызывать из других файлов, пишут так называемый *заголовочный* файл с таким же именем, но расширением `.h`.

Возьмём для примера следующие три файла:

lib.c

```
/* Определения */

int sum( int x, int y )
{
  return x + y;
}


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


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

lib.h

```
/* Объявления */

int sum( int x, int y
);


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

main.c

```
/* На место этой строчки
   автоматически вставляется
   содержимое файла lib.h */


#include "lib.h"



int main() {


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


   return 0;
}
```

lib.c	lib.h	main.c

В файле `lib.h` мы собрали объявления всех функций из файла `lib.c`. Это показывает, какие функции нам действительно нужно будет использовать в других файлах, а какие мы ввели для нашего собственного удобства. Затем эту пачку прототипов мы разом включаем в состав файлов с кодом с помощью строчки `#include "lib.h"`

В данном случае функции `sum`, `mul` мы делаем доступными (их прототипы включены в `lib.h`), а `div` мы вызывать из файла `main.c` не можем. Впрочем, мы всё равно можем включить прототип `div` в файл `main.c`, отдельно от остальных, и таки вызвать эту функцию:

```
#include "lib.h"


/* отдельный прототип для div,

   которого не было в lib.h */


int div( int, int );


int main() {


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

```
    return 0;
}
```

Рассмотрим реализацию простейшего стека из чисел размера 3. В него можно положить число и забрать его с вершины стека. При этом мы кодируем проверки: нельзя забрать число из пустого стека или положить его в заполненный полностью стек.

stack.c

```
/* define это команда сделать текстовую замену
 * Каждое слово STACK_SIZE в этом файле
 * будет заменено на 3 */

#define STACK_SIZE 3
int stack_contents[STACK_SIZE] = {0};
/* позиция сразу за верхушкой */
int stack_position = 0;

int stack_full() { return stack_position == STACK_SIZE; }
int stack_empty() { return stack_position == 0; }

int stack_push( int value ) {
  if (! stack_full() ) {
    stack_contents[stack_position] = value;
    stack_position = stack_position + 1;
    return 1;
```

main.c

```
#include "stack.h"

void stack_try_push( int i ) {
    if ( stack_push( i ) ) {
        printf( "Pushed %d\n", i);
    } else { printf( "Can't push %d\n", i); }
}

void stack_try_pop() {
    int i = 0;
    if ( stack_pop( &i ) ) {
        printf( "Popped %d\n", i);
    } else { printf( "Can't pop\n" ); }
}

int main() {
    stack_try_push( 42 );
```

stack.c

```
  }
  else { return 0; }
}


int stack_pop( int* value ) {
  if ( ! stack_empty())  {
    stack_position = stack_position - 1;
    *value = stack_contents[stack_position];
    return 1;
  }
  else { return 0; }
}
```

main.c

```
    stack_try_push( 44 );
    stack_try_push( 48 );
    stack_try_pop();
    stack_try_pop();
    stack_try_pop();


    return 0;
}
```

Напишите содержимое файла `stack.h` который позволит использовать в файле `main.c` функции `stack_push`, `stack_pop`, но не функцию `stack_full` или `stack_empty` из файла `stack.c`.

2task – program

Множественные зависимости между модулями

Заголовочные файлы часто включают друг в друга. Например, если файлу `A.h` необходимо использовать код файла `B.h`, то заголовочный файл `B.h` включается в заголовочный файл `A.h`, а `A.h` включается в A.c:

```
/* B.c */

void f() { printf("Hello!"); }


/* B.h */
```

```
void f();


/* A.h */

#include "B.h"


...



/* A.c */

#include "A.h"


int g() {

   f();

   ...

}
```

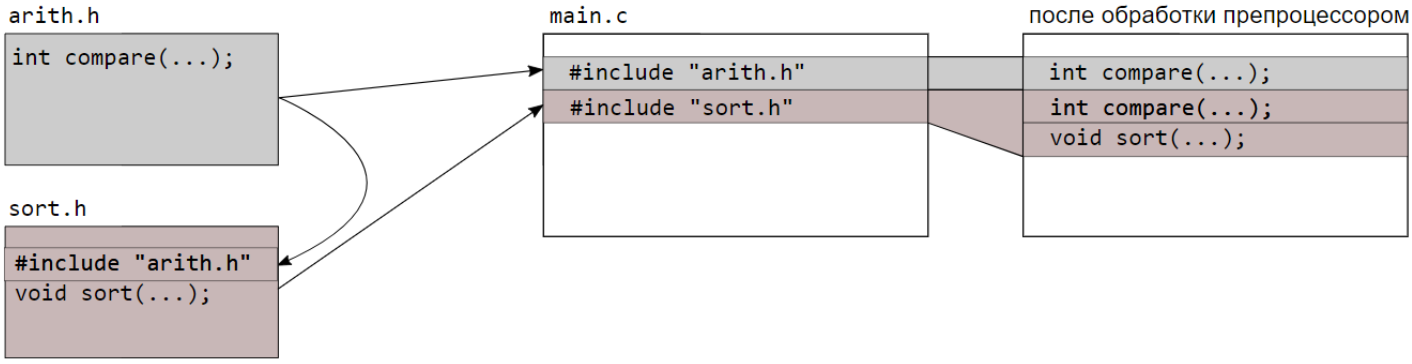
Глобальные переменные и новые типы данных (их мы ещё не умеем создавать) тоже определяются в заголовочных файлах и тоже могут иметь зависимости друг от друга. Такая схема позволяет программисту всегда предоставить компилятору всю необходимую ему информацию.

Теперь представим, что нам нужно реализовать несколько модулей, которые друг от друга зависят:

- модуль `arith.c` реализует длинную арифметику, т.е. действия над числами произвольного размера и их сравнение между собой.
- модуль `sort.c` позволяет отсортировать массивы длинных чисел.
- главный модуль `main.c` считывает массивы чисел от пользователя и сортирует их.

Здесь есть зависимости: главный модуль зависит от `arith.c` и `sort.c`, и в то же время `sort.c` нуждается в `arith.c`. Первым двум модулям соответствуют заголовочные файлы `arith.h` и `sort.h`, и для удовлетворения зависимости мы включим `arith.h` в `sort.h`.

В таком случае, включение в `main.c` файлов `sort.h` и `arith.h` приведёт к тому, что файл `arith.h` включится дважды, и всё его содержимое дублируется:



Следует избегать ситуации с дублирующимися включениями заголовочных файлов, потому что они почти всегда приводят к ошибкам компиляции. Для борьбы с ними придумана техника под названием Include guard. Чтобы освоить её, нам потребуется узнать, что такое препроцессор.

Препроцессор

Первым этапом компиляции программ является препроцессинг. Это чисто текстовая предобработка, своеобразная умная замена одних строчек на другие, которой занимается специальная программа-*препроцессор*. Мы управляем препроцессингом с помощью *директив препроцессора* прямо в тексте программы. Две директивы мы уже встречали:

- `#include` включает содержимое другого файла (и пропускает его через препроцессор);
- `#define` определяет новый **символ препроцессора**. Символом называют некий идентификатор, которому препроцессор ставит строчку в соответствие. Написав `#define X hello` мы далее будем все идентификаторы `X` заменять на `hello`.

```
#define X Y



X(42) ; // замена:  Y(42);

// X не должен являться частью строчки, ключевого слова, имени


int Xtreme; // int Xtreme, нет замены
```

Чтобы решить проблему с многократным включением заголовочных файлов, нам потребуются ещё несколько директив:

```
// можно поставить в соответствие X пустую строчку

#define X

// Главное, что теперь X определен
```

```
#ifdef X

/* Если до этого символ X  был определён (да),
   то включить текст до #endif.
   Этот блок включен.
 */

#endif


#ifndef X

/* Если до этого символ X не был определён (нет),
   то включить текст до #endif.
   Этот блок пропущен.
 */

#endif
```

Include guard

Чтобы предотвратить многократное включение файла, почти всегда пользуются трюком под названием Include Guard. Его цель в том, чтобы, используя препроцессор, ограничить возможность включения одного файла в другой одним разом. То есть, для каждого модуля любой заголовочный файл можно включить в него только один раз, а повторные включения ни к чему не приводят. Напротив, один заголовочный файл по прежнему можно включить в любое количество модулей.

Опишем Include Guard для произвольного файла file.h

```
/* Так теперь выглядит сам файл file.h */

#ifndef FILE_H

#define FILE_H


/* содержимое заголовочного файла:

   объявления функций, переменных и т.д. */


#endif
```

Включив `file.h` в первый раз, мы определим символ препроцессора `FILE_H.` При каждом дальнейшем включении `file.h` этот символ будет уже определён, и директива `#ifndef` пропустит текстовый блок до `#endif`, то есть повторное включение файла `file.h` вставит пустую строку, чего мы и добивались.

```
#include "file.h" // включит содержимое файла

// теперь  FILE_H определён

#include "file.h" // заменится на пустую строчку

#include "file.h" // заменится на пустую строчку

...
```

Мы выбрали имя для символа `FILE_H` так, чтобы оно было похоже на имя файла. Имя символа должно быть, очевидно, уникально, но будет ли это `FILE_H`, `__FILE__H` или `H_FILE` – неважно, просто делайте это одинаково для всех заголовочных файлов.

Напишите содержимое заголовочного файла с Include Guard. В качестве имени символа препроцессора используйте ARITH_H.
Внутри объявите функцию `sum`, возвращающую `int` и принимающую массив целых чисел и целое число.

3task – program

#include и стандартная библиотека

Как мы узнали, `#include` включает один файл в другой "как есть", в виде текста. В состав стандартной библиотеки языка C, той самой, которая предоставляет нам функции `printf` и `scanf`, входят несколько заголовочных файлов. Они не содержат определений функций, только их объявления. Функции `printf` и `scanf` на самом деле становятся нам доступны если включить заголовочный файл `stdio.h`:

```
#include <stdio.h>


int main() {
    /* Сейчас мы корректно используем функцию printf */
    printf("Hello!");
    return 0;
}
```

Чтобы показать, как выглядит такой файл после препроцессинга, пропустим его через компилятор gcc с флагом -E:

```
/* Запишем предыдущий пример в file.c и запустим:
    gcc -E file.c
    чтобы получить следующий текст:
*/


/* stdio.h */
/* ... */
/* вот прототип функции printf.
    Вы пока не знаете про все ключевые слова в нём */
extern int printf (const char *__restrict __format, ...);
/* Но видно, что определения тела функции printf тут нет */


/* ... */
```

```c
int main() {
    /* Сейчас мы корректно используем функцию printf */
    printf("Hello!");
    return 0;
}
```

Заметим, что мы написали имя файла в угловых скобках. Разница с кавычками в следующем:

- директива `#include <file.h>` ищет файл в стандартных директориях, на которые настроен компилятор (их можно доопределять)
- `#include "file.h"` ищет файл сначала в текущей директории, затем в стандартных директориях из прошлого пункта.

Как правило, файлы библиотек (как стандарной, так и сторонних) пишутся в угловых скобках, а файлы из самого проекта – в кавычках.

Отлично! Мы прошли подготовительную часть и изучили основные концепции языка и правила структурирования кода. Вы уже можете написать простую программу из нескольких файлов-модулей, работать с указателями, массивами, производить ввод и вывод, знаете про препроцессор.

Надеемся, что для вас язык стал немного привычен; благодаря этому мы в следующем модуле вернёмся ко многим уже изученным темам на более глубоком уровне.

Мы начнём с системы типов, узнаем про константные типы, разные числовые типы, типы `size_t`, функциональные типы, структуры, перечисления и объединения. В будущем мы научимся скрывать определения типов данных и узнаем про то, какие бывают типы полиморфизма в C и какие бывают вообще.