До настоящего момента мы имели дело с однофайловыми проектами. В таких проектах текст программы, т.е. функция main и другие функции, предназначенные для реализации алгоритма решения задачи, располагаются целиком в одном файле. Обычно со временем функционал любой программы разрастается и в случае однофайловой организации проекта это приводит к следующим недостаткам:

- ориентироваться в тексте программы становится тяжело (например, из-за того, что файл большой, приходится пользоваться поиском, чтобы найти нужное место);
- одновременная работа нескольких программистов над таким проектом становится неэффективной из-за частых конфликтов (один программист исправил функцию, другой по какой-то причине «откатил» эти изменения);
- даже при небольшом изменении приходится перекомпилировать всю программу.

Эти недостатки призвана исправить многофайловая организация проекта, к достоинствам которой относят:

- улучшение структурированности программы (обычно в один файл выносят функции, которые логически связаны друг с другом, это, в свою очередь, повышает «читаемость» программы, способствует повторному использованию функций, выделенных в отдельный файл, и т.п.);
- упрощение организации работы нескольких программистов над проектом (например, каждый может вносить изменения в отдельный файл);
- отсутствие необходимости перекомпилировать файлы, в которые изменения не вносились.

1. Многофайловый проект, раздельная компиляция

Предположим, есть однофайловый проект для программы “Hello, world!”.

hello.c

```
 1. #include <stdio.h>
 2.
 3. void hello(void)
 4. {
 5.     printf("Hello, world!\n");
 6. }
 7.
 8. int main(void)
 9. {
10.     hello();
11.
12.     return 0;
13. }
```

Для его компиляции достаточно команды

```
gcc -std=c99 -Wall -Werror -o hello.exe hello.c
```

Разделим исходный код программы на два файла: в файле main.c расположим функцию main, в файле hello.c – функцю hello.

hello.c

```
 1. #include <stdio.h>
 2.
 3. void hello(void)
 4. {
 5.     printf("Hello, world!\n");
```

```
6.  }
```

main.c

```
1.  int main(void)
2.  {
3.      hello();
4.
5.      return 0;
6.  }
```

Попытаемся собрать исполняемый файл только на основе одного из этих файлов.

```
gcc -std=c99 -Wall -Werror -o hello.exe main.c

... implicit declaration of function 'hello' [-Werror=implicit-function-declaration]
...
cc1.exe: all warnings being treated as errors
```

Файл main.c не может быть скомпилирован, потому что компилятору не знакома функция hello.

```
gcc -std=c99 -Wall -Werror -o hello.exe hello.c

... undefined reference to `WinMain'
collect2.exe: error: ld returned 1 exit status
```

Исполняемый файл на основе hello.c не может быть получен по другой причине: при компиляции этого файла ошибок не возникает, но на этапе компоновки не может быть найдена функция main.

Для исправления ошибки компиляции файла main.c необходимо объяснить компилятору прототип функции hello. Для этого достаточно добавить заголовок функции hello в файл main.c и расположить его до вызова этой функции.

main.c (исправленная версия)

```
1.  void hello(void);
2.
3.  int main(void)
4.  {
5.      hello();
6.
7.      return 0;
8.  }
```

Если теперь скомпилировать файл main.c, то компиляция пройдет без ошибки, но ошибка возникнет на этапе компоновки – теперь не хватает функции hello.

```
gcc -std=c99 -Wall -Werror -o hello.exe main.c

... undefined reference to `hello'
collect2.exe: error: ld returned 1 exit status
```

Таким образом для получения исполняемого файла на основе многофайлового проекта нужны все файлы, составляющие этот проект. При этом для функций, которые определяются в одном файле, а вызываются в другом, в последний нужно поместить объявление таких функций.

Алгоритм получения исполняемого файла в нашем случае выглядит следующим образом:

- получим объектный файл на основе main.c,
- получим объектный файл на основе hello.c,
- из полученный объектных файлов получим исполняемый файл.

```
gcc -std=c99 -Wall -Werror -c main.c
gcc -std=c99 -Wall -Werror -c hello.c
gcc -o hello.exe hello.o main.o
```

2. Заголовочные файлы

В нашем примере всего два си-файла, поэтому добавить объявление функции hello в нужное место программы несложно. Эта задача становится трудно решаемой, когда программа состоит из большого количества файлов и большого количества функций. В этом случае заголовки функций из си-файла выносят в так называемый заголовочный файл, который подключается в те си-файлы, в которых требуются эти функции.

hello.h

```
1.  void hello(void);
```

hello.c

```
1.  #include <stdio.h>
2.  #include "hello.h"
3.
4.  void hello(void)
5.  {
6.      printf("Hello, world!\n");
7.  }
```

main.c

```
1.  #include "hello.h"
2.
3.  int main(void)
4.  {
5.      hello();
6.
7.      return 0;
8.  }
```

Появление заголовочного файла никак не изменяет алгоритм получения исполняемого файла.

Обратите внимание, что заголовочный файл hello.h подключается не только в файл main.c, но и в файл hello.c. Это сделано специально. Оба файла содержат дублирующуюся информацию – заголовок функции hello. Если по какой-то причине эти заголовки в файлах hello.c и hello.h начнут расходится (например, из-за опечатки или потому что заголовок функции был изменен при внесении каких-то изменений в определение этой функции), это может привести к получению неработающей программы. Чтобы автоматизировать такую проверку и переложить ее на компилятор, необходимо включить заголовочный файл hello.h в соответствующий си-файл (т.е. hello.c).

Другой положительный «эффект» от включения заголовочного файла в си-файл заключается в том, что расположение определений функций в си-файле может быть произвольным.

Имена стандартных заголовочных файлов обычно указываются в угловых скобках. Благодаря этому, компилятор ищет эти файлы по специальным путям, по которым он разместил стандартные заголовочные файлы при своей установке. Имена заголовочных файлов вашей

программы обычно указываются в двойных кавычках. В этом случае компилятор ищет их в текущей директории.

Замечание
С помощью ключа **"-I"** можно указать компилятору дополнительные пути, по которым нужно искать заголовочные файлы.

В заголовочный файл кроме объявлений функций помещают определения типов и макросов. Здесь мы можем столкнуться со следующей ситуацией (идея примера заимствована из https://ru.wikipedia.org/wiki/Include_guard).

grandfather.h

```
1.  struct foo
2.  {
3.      int foo;
4.  };
```

father.h

```
1.  #include "grandfather.h"
```

child.c

```
1.  #include "grandfather.h"
2.  #include "father.h"
```

После обработки файла child.c препроцессором структура foo будет определена в нем два раза (один раз при включении grandfather.h, второй при включении father.h), что приведет к возникновению ошибки компиляции.

Для того чтобы защититься от повторного включения используется конструкция, которая называется «include guard».

grandfather.h (правильное оформление заголовочного файла)

```
1.  #ifndef _GRANDFATHER_H_
2.
3.  #define _GRANDFATHER_H_
4.
5.  struct foo
6.  {
7.      int foo;
8.  };
9.
10. #endif
```

Рассмотрим как работает эта конструкция при обработке файла child.c препроцессором. Сначала обрабатывается файл grandfather.h.

```
1.  #ifndef _GRANDFATHER_H_
```

Имя **_GRANDFATHER_H_** определено? Нет, продолжаем.

```
3.  #define _GRANDFATHER_H_
```

Теперь имя определено. Продолжаем.

```
5.  struct foo
6.  {
```

```
 7.      int foo;
 8. };
```

В файл child.c помещается определение структуры foo. Продолжаем.

```
10. #endif
```

Директива #if закончилась. Переходим к обработке father.h.

```
 1. #ifndef _GRANDFATHER_H_
```

Имя **_GRANDFATHER_H_** определено? Да, пропускаем все до соответствующей директивы #endif.

Для правильной работы «include guard» необходимо позаботится о том, чтобы имена, которые используются в этой конструкции были уникальны.

Замечание
В качестве не стандартного решения можно предложить использовать прагму once (#pragma once), которая должна располагаться в начале заголовочного файла.