# Директивы препроцессора

Препроцессор – это специальная программа, часть компилятора языка **C**, которая позволяет включать в текст программы файлы, вводить макроопределения и выполнять условную компиляцию. По сути осуществляет обработку исходного кода перед передачей его на следующий шаг компиляции. Важно помнить что препроцессор работает на уровне текста программы и **не проверяет** никаких синтаксических и семантических правил языка.

Работа препроцессора осуществляется с помощью специальных директив, начинающихся знаком **#**. По окончании строк директив, точку с запятой ставить не нужно.

# Директива #include

Директива **#include** позволяет включать в текст программы содержимое указанного файла:

*#include <имя_файла>*

Если файл является заголовочным стандартной библиотеки и находится в папке компилятора, он заключается в угловые скобки **<>**. Если файл находится в текущем каталоге проекта, он указывается в кавычках "". Для файла, находящегося в другом каталоге необходимо в кавычках указать полный путь.

```
#include <stdio.h>
#include "part.c"
#include "C:\headers\funcs.h"
```

# Директива #define

Директива **#define** позволяет вводить в текст программы макроопределения:

*#define <идентификатор> <замена>*

Директива указывает компилятору, что нужно подставить строку, определенную как **Замена**, вместо каждого аргумента **Идентификатор** в исходном файле. **Замена** не действует для строковых констант или на части более длинного идентификатора. Имена макроопределений принято писать БОЛЬШИМИ_БУКВАМИ.

```
#define A 3
printf("%d + %d = %d", A, A, A+A); // 3 + 3 = 6
```

В зависимости от значения константы компилятор присваивает ей тот или иной тип. С помощью суффиксов можно переопределить тип константы:

- U или u представляет целую константу в беззнаковой форме (unsigned);
- F (или f) позволяет описать вещественную константу типа float;
- L (или l) позволяет выделить целой константе 8 байт (long long int);

```
#define A 280u    // unsigned int
#define B 28.0f   // float
#define C 28.0    // double
```

Вторая форма синтаксиса **#define** определяет макрос, подобный функции, с параметрами. Эта форма допускает использование необязательного списка параметров, которые должны находиться в скобках:

```
#define <имя>(<аргумент1>[, ..., <аргументN>]) <замена>
```

После определения макроса каждое последующее вхождение замещается версией аргумента **замена**, в которой вместо формальных аргументов подставлены фактические аргументы.

```c
#define PI 3.14159265
#define SIN(x) sin(PI*x/180.0)

int main()
{
  int c;

  printf("Angle in deg: ");
  scanf("%d", &c);
  printf("sin(%d)=%f", c, SIN(c));

  return 0;
}
```

Однако при использовании таких макроопределений следует соблюдать осторожность:

```c
#define SUM_BAD(A,B) A+B
#define SUM_GOOD(A,B) (A+B)
int main()
{
  int a=3, b=5, c=2;

  printf("%d\n", SUM_BAD(a,b)*c); // 13!
  printf("%d\n", SUM_GOOD(a,b)*c); // 16

  return 0;
}
```

Оператор **##** в директиве **#define** берет два отдельных токена и вставляет их вместе в один токен. Полученный токен может быть именем переменной, именем типа или любым другим идентификатором:

```
//объявляем переменную заданного типа, с заданным именем типа var_***
//и начальным значением
#define DECLARE_AND_SET(type, varname, value) type var_##varname = value;

int main()
{
    DECLARE_AND_SET(int, num1, 4 * 2);

    printf("%d", var_num1); //8
}
```

По умолчанию текст макроопределения должен размещаться на одной строке.
Если требуется перенести текст макроопределения на новую строку, то в конце текущей строки ставится символ обратный слеш ****:

```
#define SUM(A,B) (A+ \
                  B)
```

# Директива #undef

Директива **#undef** отменяет предыдущее макроопределение:

```c
#define VAR 1

int main()
{
    printf("%d",VAR);

    #undef VAR

    printf("%d",VAR); // ОШИБКА КОМПИЛЯЦИИ!
}
```

# Условная компиляция

Директивы **#if** или **#ifdef / #ifndef** вместе с директивами **#elif**, **#else** и **#endif** управляют компиляцией исходного файла.

Если указанное выражение после **#if** имеет ненулевое значение, в записи преобразования сохраняется группа строк, следующая сразу за директивой **#if**. Синтаксис условной директивы следующий:

*#if <константное выражение>*
*  <группа операций>*
*#elif <константное выражение>*
*  <группа операций>*
*#else*
*  <группа операций>*
*#endif*

Отличие директив **#ifdef** и **#ifndef** от **#if** заключается в том, что константное выражение может быть задано только с помощью **#define**. У каждой директивы **#if** должна быть соответствующая закрывающая директива **#endif**.

Между ними может располагаться любое кол-во директив **#elif**, но не более одной директивы **#else**, которая должна быть последней.

```
#define DEBUG
#define VERSION 2
int main()
{

    #if VERSION<3
        printf("Old version! Update, please!");
    #endif

    #ifdef DEBUG
        printf("Debug version!");
    #else
        printf("Release version!");
    #endif

  return 0;
}
```

# Модульное программирование

**Модульное программирование** — это организация программы как совокупности небольших независимых блоков, называемых модулями.

**Модуль** — функционально законченный отдельно-компилируемый фрагмент программы, оформленный в виде отдельного файла с исходным кодом.

Модуль в языке **C** состоит из интерфейса (заголовочого файла **.h**) и реализации (файла **.c**).

Код, подключающий модуль, на этапе компиляции нуждается только в интерфейсе модуля, поэтому на этапе препроцессинга заголовочный файл копируется в код директивой **#include "module.h"**

Реализация модуля должна полностью реализовывать указанный интерфейс, поэтому она также включает свой заголовочный файл.

main.c | testlib.h | testlib.c

```c
#include <stdio.h>

//main.c
#include <stdlib.h>
#include "testlib.h"

int main()
{
    sayHello();
    return 0;
}
```

main.c | testlib.h | testlib.c

```c
//testlib.h
#ifndef TESTLIB_H
#define TESTLIB_H

void sayHello();

#endif //TESTLIB_H
```

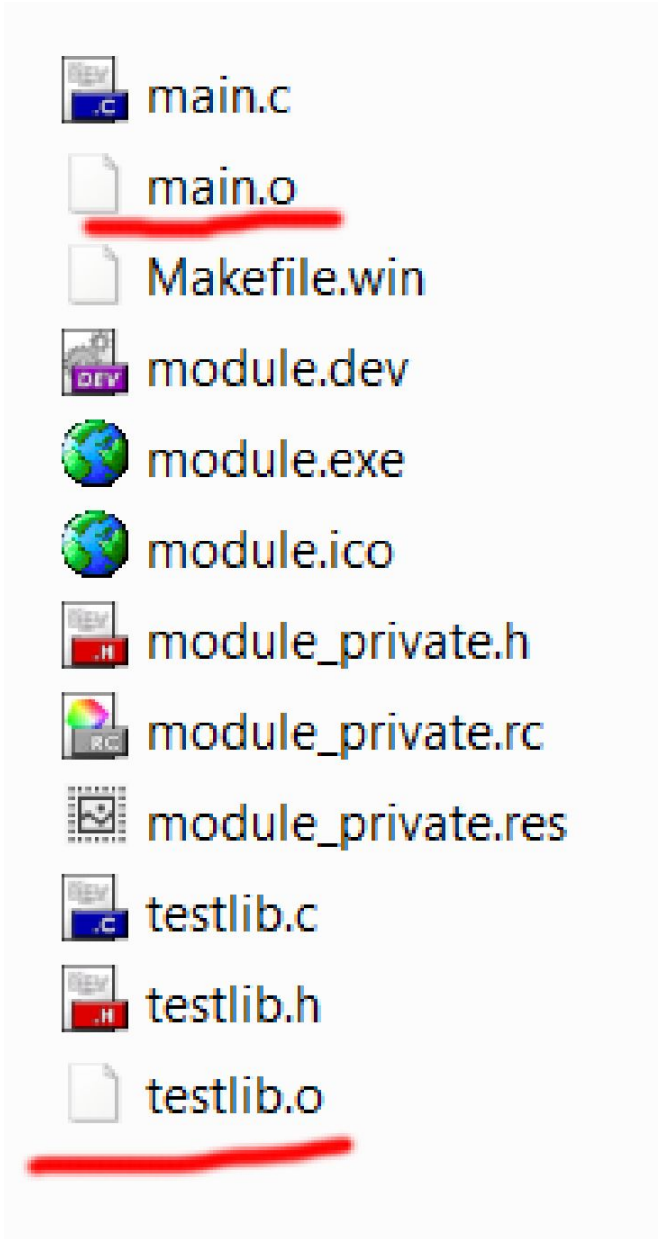
main.c | testlib.h | testlib.c

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

void sayHello()
{
    printf("Hello, module!\n");
}
```

В данном примере в файле **main.c** не понадобилось подключать **stdio.h**, хотя он и используется в модуле **testlib.c**, т.к. никакие типы из **stdio.h** не нужны для корректной обработки интерфейса **testlib.h**, оказывающегося в **main.c** на этапе компиляции.

Если бы это было не так, то библиотеки должны были бы быть включены не в **testlib.c**, а в **testlib.h**, чтобы во всех местах, где подключается модуль **testlib**, и все нужные заголовочные файлы включались автоматически.




После компиляции мы получили два объектных файла модулей ***.o**, которые представляют собой блоки машинного кода и данных с неопределенными адресами ссылок на данные и функции в других объектных модулях, а также список своих функций и данных.

При компиляции **компоновщик** собирает код и данные каждого объектного файла модуля в итоговую программу (***.exe**), вычисляет и заполняет адреса перекрестных ссылок между модулями.

Соответственно при внесении изменений в один модуль, нет необходимости перекомпилировать все остальные модули, а значит компиляция будет осуществляться гораздо быстрее.

# Глобальные переменные

Поскольку **C** позволяет выполнять раздельную компиляцию модулей для большой программы в целях ускорения компиляции и помощи управлению большими проектами, должны быть способы передачи информации о глобальных переменных файлам программы. Решение заключается в объявлении глобальных переменных в одном файле и использовании при объявлении в других файлах ключевого слова **extern**:

```
// File 1

int x, y;
char ch;

main(void)
{
    ...
}
```

```
//File 2

extern int x, y;
extern char ch;

void func1(void)
{
    x = y / 10.0;
}
```

Спецификатор **extern** сообщает компилятору, что следующие за ним типы и имена переменных объявляются где-то в другом месте. **Extern** позволяет компилятору знать о типах и именах глобальных переменных без действительного создания этих переменных. Когда два модуля объединяются, все ссылки на внешние переменные пересматриваются.

Если при объявлении выделяется память под переменную, то процесс называется определением. Использование **extern** приводит к объявлению, но не к определению. Оно просто говорит компилятору, что определение происходит где-то в другом месте программы.

# Статические переменные

Статические переменные являются долговременными переменными, существующими на протяжении функции или файла. Они отличаются от глобальных переменных, поскольку не известны за пределами функции или файла, но могут хранить свои значения между вызовами.

Различают **статические локальные** и **статические глобальные** переменные.

Когда модификатор **static** применяется к локальной переменной, это приводит к тому, что компилятор создает долговременную область для хранения переменной почти также, как это делается для глобальной переменной.

Ключевое различие между статической локальной и глобальной переменными заключается в том, что статическая локальная переменная остается известной только в том блоке, в котором она была объявлена. Т. е. статическая локальная переменная - это локальная переменная, сохраняющая свое значение между вызовами функций.

```c
//получать при каждом вызове уникальный номер
//каждый раз на единицу больше предыдущего
unsigned int getUniqID()
{
    //статическая переменная хранит свое значение
    //между вызовами функции
    static unsigned int id = 1;
    return id++;
}

int main()
{
    printf("%d", getUniqID()); // 1
    printf("%d", getUniqID()); // 2
    printf("%d", getUniqID()); // 3

    return 0;
}
```

Когда спецификатор **static** применяется к глобальной переменной, он сообщает компилятору о необходимости создания глобальной переменной, которая будет известна только в файле, где статическая глобальная переменная объявлена.

Это означает, что, даже если переменная является глобальной, другие подпрограммы в других файлах не будут знать о ней. Таким образом, не возникает повода для побочных эффектов.

**Файл main.c:**

```c
static unsigned int id = 255;

int main()
{
    prn();
    return 0;
}
```

**Файл func.c:**

```c
#include <stdio.h>

extern unsigned int id;

void prn()
{
    printf("%d", id); //ошибка! undefined reference to 'id'
}
```

# Объявление типа

Язык **C** позволяет определять имена новых типов данных с помощью ключевого слова **typedef**. На самом деле здесь не создается новый тип данных, а определяется новое имя существующему типу. Он позволяет облегчить создание машинно-независимых программ. Единственное, что потребуется при переходе на другую платформу, - это изменить оператор **typedef**. Он также может помочь документировать код, позволяя назначать содержательные имена стандартным типам данных:

```
typedef <тип> <имя>;
```

где тип — это любой существующий тип данных, а имя - это новое имя для данного типа. Новое имя определяется в дополнение к существующему имени типа, а не замещает его. Использование **typedef** может помочь при создании более легкого для чтения и более переносимого кода.

```c
//объявляем новое имя типа
typedef unsigned int UInt;

//теперь его можно использовать как любой другой тип
UInt a = 5 + 2;
printf("%d", a); // 7

//можно определить имя типа для указателя
typedef int* IntPtr;
IntPtr ptr; //и создавать указатели данного типа

//можно упростить объявление типа для указателей на функции
typedef float (*MathFunc)(float);
MathFunc funcPtr; //указатель на функцию

//или определить имя типа для массива
typedef int IntArr[6];
IntArr arr = {1,2,3,4,5,6};
```