КОНСТРУИРОВАНИЕ ПРОГРАММ

Лекция № 06.1 Объявления. Спецификаторы.

Преподаватель: Поденок Леонид Петрович, 505a-5

+375 17 293 8039 (505a-5)

+375 17 320 7402 (ОИПИ НАНБ)

prep@lsi.bas-net.by

ftp://student:2ok*uK2@Rwox@lsi.bas-net.by/

Кафедра ЭВМ, 2021

2028.09.29-14:40-15:20

Оглавление

Объявления (описания, декларации)

Синтаксис

объявления:
    спецификаторы-объявления [список-деклараторов[1]-с-инициализацией] **;**
спецификаторы-объявления:
    спецификатор-класса-памяти [спецификаторы-объявления]
    спецификатор-типа [спецификаторы-объявления]
    квалификатор-типа [спецификаторы-объявления]
    спецификатор-функции [спецификаторы-объявления]
    спецификатор-выравнивания [спецификаторы-объявления]
список-деклараторов-с-инициализацией:
    декларатор-с-инициализацией
    список-деклараторов-с-инициализацией **,** *декларатор-с-инициализацией*
декларатор-с-инициализацией:
    декларатор
    декларатор **=** *инициализатор*

1 декларатор¸описатель, объявитель — содержит идентификаторы, подлежащие объявлению

Семантика

Объявление идентификатора определяет интерпретацию и атрибуты набора идентификаторов.

Определение идентификатора — это объявление этого же идентификатора, которое:

- для объекта выделяет для него область памяти;
- для функции включает тело функции;
- для константы перечисления является (единственным) объявлением идентификатора

```
enum color {
    red;
    green;
    blue;
};
```

- для имени типа (typedef-имени) является первым (или единственным) объявлением идентификатора

```
typedef unsigned int uint32_t;                                  // <stdint.h>

typedef void (*sighandler_t)(int);                              // <signal.h>
sighandler_t signal(int signum, sighandler_t handler);
```

Общий синтаксис объявления

спецификаторы-объявления [список-деклараторов-с-инициализацией] ;

Декларатоpы содержат идентификаторы (если таковые имеются), которые подлежат объявлению.

Спецификаторы объявления состоят из последовательности спецификаторов, которые указывают тип связывания, продолжительность хранения и *часть типа* сущностей (объекты, функции), на которые эти декларатоpы указывают.

```
extern double gold_constant = 1.6180339887498948482045 9;
extern int errno;                                       // tclUnixPort.h

// linux
extern int *__errno_location(void) __THROW __attribute_const__;
# define errno (*__errno_location())
```

Список деклараторов-с-инициализацией — это разделенная запятыми последовательность деклараторов, каждый из которых может иметь *дополнительную информацию о типе*, или инициализатор, или оба.

Пример 1. decl-test.c

```
#include <stdio.h>
#include <errno.h>

extern int magic_num;
const double M_SQRT2;
double M_PI = 3.14159265358979323846;
static long double gold_constant = 1.6180339887498948482045 9L;
int val, foo(int, int), arr[5] = {0, 1, 2, 3, 4}, **ipp = NULL;

int main (int argc, char *argv[]) {

    printf("helloworld!!!\n");
    return(errno);
}

$ nm decl-test.o
0000000000000020 D arr                // in the initialized data section
                 U __errno_location   // is undefined
0000000000000010 d gold_constant      // in the initialized data section
0000000000000008 B ipp                // in the BSS data section
0000000000000000 T main               // in the text (code) section
0000000000000000 D M_PI               // in the initialized data section
0000000000000000 R M_SQRT2            // in a read only data section
                 U puts               // is undefined
0000000000000000 B val                // in the BSS data section
```

Пример 2

```
#include <stdio.h>
#include <errno.h>

extern int magic_num;
const double M_SQRT2;
double M_PI = 3.14159265358979323846;
static long double gold_constant = 1.6180339887498948482045 9L;
int val, foo(int, int), arr[5] = {0, 1, 2, 3, 4}, **ipp = NULL;

int main (int argc, char *argv[]) {

    printf("magic num: %d\n", magic_num); // ссылка на extern int magic_num
    return(errno);
}

$ nm decl-test.o
0000000000000020 D arr
                 U __errno_location
0000000000000010 d gold_constant
0000000000000008 B ipp
                 U magic_num
0000000000000000 T main
0000000000000000 D M_PI
0000000000000000 R M_SQRT2
                 U printf
0000000000000000 B val
```

Спецификаторы класса памяти (класса хранения)

спецификатор-класса-памяти:

typedef
extern
static
_Thread_local
auto
register

Спецификатор **typedef** называется «спецификатором класса памяти» только для синтаксического удобства

register

Объявление идентификатора для объекта со спецификатором класса хранения **register** предполагает, что доступ к объекту должен быть максимально быстрым.

Степень эффективности указания класса памяти **register** определяется реализацией. Конкретная реализация компилятора может трактовать любое объявление класса памяти **register** просто как **auto**.

Однако, независимо от того, действительно ли используется адресуемая память, адрес любой части объявленного объекта со спецификатором класса памяти **register** не может быть вычислен ни явно с помощью унарного оператора **&**, ни неявно, путем преобразования имени массива в указатель.

Таким образом, единственный оператор, который может быть применен к массиву, объявленному с помощью спецификатора класса памяти **register**, это **sizeof**.

Ограничения

В спецификаторах объявлений может быть задано не больше одного спецификатора класса памяти. Исключение составляет **_Thread_local**, который может появляться с **extern** или **static**.

При объявлении объекта с областью действия блока, если спецификаторы объявления включают **_Thread_local**, они также должны включать в себя либо **static**, либо **extern**.

Если **_Thread_local** присутствует в каком-либо объявлении объекта, он должен присутствовать в каждом объявлении этого объекта.

Спецификаторы типа

Синтаксис

спецификатор-типа:

один из:

void, char, short, int, long, signed,
unsigned, float, double, _Bool, _Complex

спецификатор-атомарного-типа

struct-или-union-спецификатор

спецификатор-перечисления

имя-типа[2]

Семантика

В спецификаторах каждого объявления, а также в списке спецификаторов типа в каждом объявлении структуры и имени типа должен быть указан, по крайней мере, один спецификатор типа.

Определяет ли спецификатор **int** тот же тип, что и **signed int**, или тот же тип, что и **unsigned int**, определяется реализацией.

Реализация (компилятор) может не поддерживает комплексные типы, в этом случае спецификатор типа ***_Complex*** *использоваться не должен.*

2 typedef-имя

Спецификаторы структур и объединений

Структура — это тип, состоящий из последовательности элементов, которые размещаются в памяти в упорядоченной последовательности.

Объединение — это тип, состоящий из последовательности элементов, чье размещение перекрывается.

Спецификаторы структуры и объединения имеют одинаковую форму.

Ключевые слова **`struct`** и **`union`** указывают, что указываемый тип является, соответственно, типом структуры или типом объединения.

Синтаксис

спецификатор-структуры-или-объединения:

1 *структура-или-объединение идентификатор* **{** *список-объявлений-структуры* **}**

2 *структура-или-объединение* **{** *список-объявлений-структуры* **}**

3 *структура-или-объединение идентификатор*

структура-или-объединение:

4 **`struct`**

5 **`union`**

список-объявлений-структуры:

6 *объявление-структуры*

7 *список-объявлений-структуры объявление-структуры*

объявление-структуры:

8 *список-спецификаторов-квалификаторов список-деклараторов-структуры* ;

9 *список-спецификаторов-квалификаторов* ;

10 *static_assert-declaration*

список-спецификаторов-квалификаторов:

11 *спецификатор-типа список-спецификаторов-квалификаторов*

12 *спецификатор-типа*

13 *квалификатор-типа список-спецификаторов-квалификаторов*

14 *квалификатор-типа*

список-деклараторов-структуры:

15 *декларатор-структуры*

16 *список-деклараторов-структуры* , *декларатор-структуры*

декларатор-структуры:

17 *декларатор*

18 *декларатор* : *константное-выражение*

19 : *константное-выражение*

Объявления структур

```
struct id;

struct id {
```

```
    <список объявлений структуры>
}

struct {                                    // анонимная структура/объединение
    <список объявлений структуры>
}
```

Если «*список-объявлений-структуры*» в «*спецификаторе-структуры-или-объединения*» присутствует, это означает, что объявляется новый тип. Объявление имеет силу в рамках данной единицы трансляции (исходного файла).

«*список-объявлений-структуры*» представляет собой последовательность объявлений для членов структуры или объединения.

Если *список-объявлений-структуры* не содержит именованных членов, либо напрямую, либо через анонимную структуру или анонимное объединение, поведение не определено.

Тип является неполным до тех пор, пока не встретится **}**, которая завершает список.

Член структуры или объединения может иметь любой полный тип объекта, отличный от изменяемого типа (variable modified)[3].

3 Структура или объединение не могут содержать члена с изменяемым типом, поскольку имена членов не являются обычными идентификаторами

Битовые поля

Член может быть объявлен состоящим из указанного числа битов (включая знаковый бит, если таковой имеется). Такой элемент называется *битовым полем*. Его ширине предшествует двоеточие '**:**'.

[декларатор] **:** *константное-выражение*

К объекту битового поля не может быть применен унарный оператор & (address-of), таким образом, у объектов битового поля нет указателей, а также из них нельзя сконструировать массив.

Битовое поле интерпретируется как имеющее целочисленный тип со знаком или без знака, состоящий из указанного числа битов. Если в битовом поле ненулевой ширины типа **`_Bool`** сохраняются значения 0 или 1, значение битового поля в операциях сравнения интерпретируется, как равное сохраненному значению. Битовое поле **`_Bool`** имеет семантику **`_Bool`**.

Компилятор может выделить любую адресуемую единицу памяти достаточной величины, чтобы содержать битовое поле.

Если остается достаточно места, битовое поле, которое следует сразу за другим битовым полем в структуре, упаковывается в смежные биты той же единицы памяти.

Если места остается недостаточно, будет ли битовое поле, которое не умещается, целиком помещено в следующем блоке, или будет частично располагаться в не полностью заполненном блоке и смежном с ним, определяется реализацией.

Порядок распределения битовых полей в блоке

Порядок распределения битовых полей в блоке (от старшего к младшему или от младшего к старшему) опять же определяется реализацией (Most/Least Significant).

Выравнивание адресуемой единицы памяти не определено.

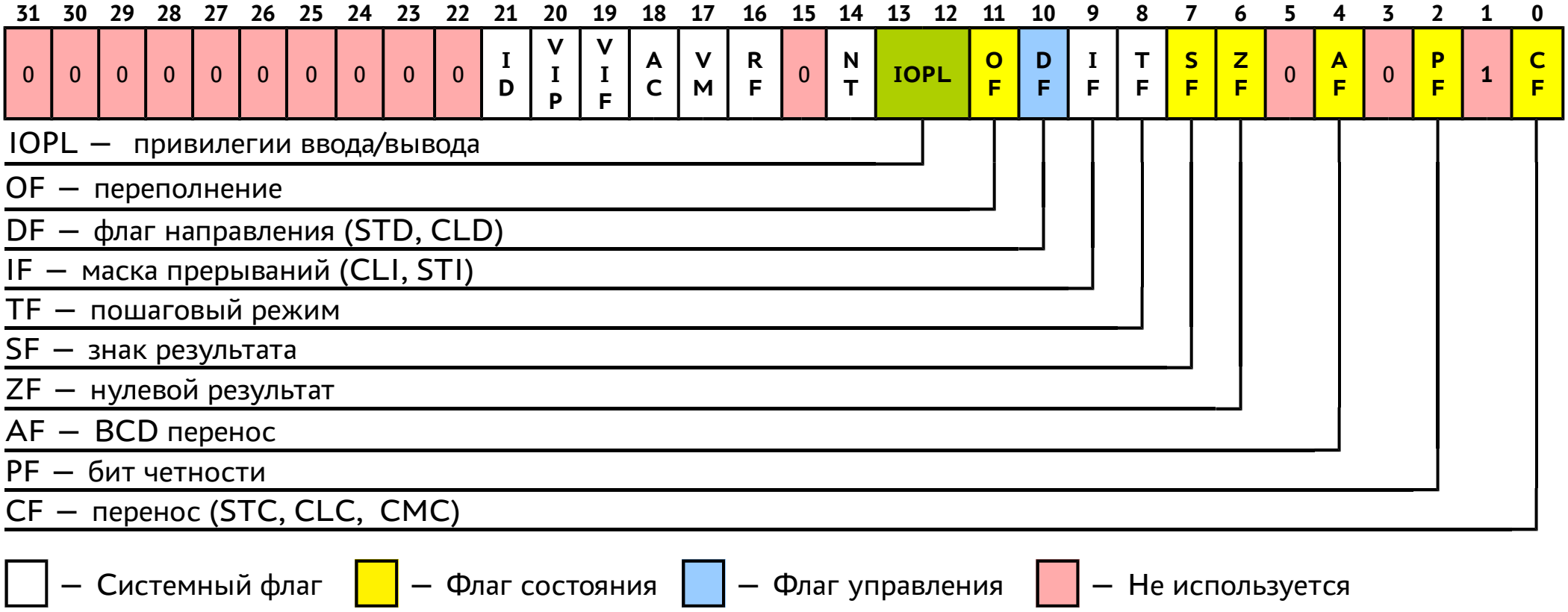
Формат сегментного регистра x86_64 (селектора)

Порядок полей в слове при компиляции gcc -1.1.0 — LSB (`rpl:ti:index`).

```
union {
    struct {
        unsigned int index : 13;  // индекс в таблице дескрипторов
        _Bool        ti    : 1;   // LDT или  GDT
        unsigned int rpl   : 2;   // запрашиваемый уровень привилегий
    } sreg;
    uint16_t         as_reg16;    // в виде 16 битного слова
} selector;
```

Битового поле, объявленное *без декларатора*, только лишь с двоеточием и шириной, означает *неименованное битовое поле*. Такой элемент структуры неименованного битового поля полезен в качестве заполнителя, чтобы структура битового поля соответствовала требованиям, установленным вне контекста программы.

Регистр флагов (EFLAGS Register) x86_64



```
union {
    uint32_t     as_reg32;  // в виде 32-разрядного слова
    struct {
        uint32_t CF   :  1; // перенос
        uint32_t      :  1; // зарезервировано = 1
        uint32_t PF   :  1; // четность
        uint32_t      :  1; // зарезервировано = 0
        uint32_t AF   :  1; // дополнительный перенос
        uint32_t      :  1; // зарезервировано = 0
        uint32_t ZF   :  1; // ноль
        uint32_t SF   :  1; // знак
        uint32_t TF   :  1; // пошаговый режим
        uint32_t IF   :  1; // cli/sti
        uint32_t DF   :  1; // направление
        uint32_t OF   :  1; // переполнение
        uint32_t IOPL :  2; // привиелгии воода/вывода
        uint32_t NT   :  1; // system
        uint32_t      :  1; // зарезервировано = 0
        uint32_t RF   :  1; // system
        uint32_t VM   :  1; // system
        uint32_t AC   :  1; // system
        uint32_t VIF  :  1; // system
        uint32_t VIP  :  1; // system
        uint32_t ID   :  1; // system
        uint32_t      : 10; // зарезервировано = 0
    } eflags;
} EFLAGS = {0x0000000000000002};
```

В качестве специального случая элемент структуры битового поля шириной 0 указывает, что никакое дополнительное битовое поле не должно упаковываться в блок, в который было помещено предыдущее битовое поле, если оно есть.

Анонимные структура и объединение

Непоименованный член, спецификатор типа которого является спецификатором структуры без тега, называется *анонимной структурой*.

Непоименованный член, спецификатор типа которого является спецификатором объединения без тега, называется *анонимным объединением*.

Члены анонимной структуры или объединения считаются членами содержащей их структуры или объединения.

Это применяется рекурсивно, если содержащая структура или объединение также являются анонимными.

Каждый член небитового поля структуры или объекта объединения выравнивается определенным реализацией способом, в соответствии с его типом.

Внутри объекта структуры члены небитовых полей и блоки, в которых находятся битовые поля, имеют адреса, которые увеличиваются в порядке их объявления.

```
struct {              // анонимная структура
    double x;
    double y;
}
```

Размер объединения достаточен, чтобы вместить самый большой из его членов.

В объекте объединения в одно и то же время может быть сохранено значение не более одного из членов.

Указатель на объект объединения, соответствующим образом преобразованный, указывает на любой из его элементов (или, если элемент является битовым полем, то на элемент, в котором он находится), и наоборот.

В конце структуры или объединения может быть безымянный отступ.

```
struct geo_point {
    double lat;         // широта (φ)
    double lon;         // долгота (λ)
    union {
        double chi;     // угол между ортодромиями (χ)
        double hgt;     // высота
    }
};
```

```
struct geo_point gp1, gp2; // две точки
...
gp2.lat = gp1.lat;
gp2.lon = gp1.lon;
gp2.chi =   0.0;
gp1.hgt = 123.0;
```

Пластичный массив

Как особый случай, последний элемент структуры, содержащей более чем один именованный элемент может иметь незавершенный тип массива — этот элемент называется *членом пластичного массива (свободный или висящий член)*.

В большинстве случаев член пластичного массива игнорируется.

В частности, размер структуры остается такой, как если бы элемент пластичного массива отсутствовал, за исключением того, что такой член может иметь большее завершающее заполнение, чем подразумевает его отсутствие.

Формат IFF — Interchange File Format.


```
struct chunk {
    uint16_t  type_id;
    uint16_t  data_sz;
    char       data[];
}
```

Ограничения

«объявление-структуры», которое не описывает анонимую структуру или анонимное объединение, должно содержать «список-деклараторов-структуры».

Структура или объединение не должны содержать членов с неполным типом или типом функции, следовательно, структура не должна содержать своего экземпляра, однако может содержать указатель на свой экземпляр.

```
struct s {
    struct s a;  // недопустимо — незавершенный тип
    struct s *a; // нормально — размер указателя всегда известен
}
```

Последний член структуры с более чем одним именованным членом может иметь незавершенный тип массива.

```
struct s {
    int a;     // именованый член
    int arr[]; // незавершенный тип массива
}
```

Такая структура (и любое объединение, содержащее, возможно, рекурсивно член, являющийся такой структурой) не должно быть членом структуры или элементом массива.

Выражение, которое определяет ширину битового поля, должно быть целочисленным константным выражением с неотрицательным значением, которое не превышает ширину объекта того типа, который был бы указан, если бы двоеточие и выражение отсутствовали[4].

Если значение равно ноль, объявление не должна иметь декларатора.

Битовое поле должно иметь тип, который является квалифицированной или неквалифицированной версией **_Bool**, **signed int**, **unsigned int** или какого-либо другого типа, определяемого реализацией. Разрешены ли атомарные типы, определяется реализацией.

4 Хотя число битов в объекте _Bool составляет не менее CHAR_BIT, ширина (количество битов знака и значения) _Bool может составлять всего 1 бит.

Примеры структур/объединений

спецификатор-структуры-или-объединения:

1 *структура-или-объединение идентификатор* **{** *список-объявлений-структуры* **}**

2 *структура-или-объединение* **{** *список-объявлений-структуры* **}**

3 *структура-или-объединение идентификатор*

структура-или-объединение:

4 **struct**

5 **union**

```
struct some_struct

struct some_struct { // some_struct – имя типа структуры
    int       n;     // декларатор: член структуры с типом int
    const int m;     // декларатор: член структуры с типом const int
    double    x;     // декларатор: член структуры с типом double
}

или с деклараторами:

struct some_struct { // some_struct – имя типа структуры
    int       n;     // декларатор: член структуры с типом int
    const int m;     // декларатор: член структуры с типом const int
    double    x;     // декларатор: член структуры с типом double
} sobject, other, s_array[5]; // список деклараторов с уточнением типа
```

Пример 1.
Иллюстрация анонимных структур и объединений

```
struct v {
    union {             // анонимное объединение
        struct {        // анонимная структура
            int i, j;
        };
        struct {
            long k, l;
        } w;
    };
    int m;
} v1;
v1.i   = 2; // годится
v1.k   = 3; // недействительно: внутренняя структура не анонимная
v1.w.k = 5; // годится
```

Пример 2.
После объявления:

```
struct s {
    int    n;
    double d[]; // свободный/висящий член
};
```

структура **`struct s`** имеет свободный член **`d`** с типом массива элементов типа **`double`**.

Типичный способ использовать такую конструкцию состоит в следующем:

```
int m       = // размер массива
struct s *p = malloc(sizeof (struct s) + sizeof (double[m]));
```

и, предполагая, что вызов **`malloc()`** был успешен, объект, на который указывает **`p`**, ведет себя в большинстве случаев так, как если бы **`p`** был объявлен как:

```
struct {
    int    n;
    double d[m];
} *p;
```

Существуют обстоятельства, при которых эта эквивалентность может нарушаться, в частности, смещения члена **`d`** в обоих случаях могут не совпадать.

После вышеупомянутой декларации:

```
struct s {
    int    n;
    double d[]; // свободный/висящий член
};
struct s t1 = { 0 };          // нормально
struct s t2 = { 1, { 4.2 }}; // недопустимо
t1.n        = 4;              // нормально
t1.d[0]     = 4.2;            // неопределенное поведение
```

Инициализация **t2** недопустима (нарушает ограничение), потому что **struct s** обрабатывается так, как если бы она не содержала члена **d**.

Назначение для **t1.d[0]**, скорее всего, является неопределенным поведением, но возможно, что **sizeof(struct s) >= offsetof(struct s, d) + sizeof(double)**, и в этом случае назначение будет допустимым. Тем не менее, такое не должно появляться в коде, строго соответствующем стандарту.

После дальнейшего объявления:

```
struct ss { int n; };
```

выражения:

```
sizeof (struct s) >= sizeof (struct ss)
sizeof (struct s) >= offsetof (struct s, d)
```

всегда равны 1.

Пример *: Если **sizeof (double)** равен 8, то после выполнения следующего кода:

```
struct s *s1;
struct s *s2;
s1 = malloc(sizeof (struct s) + 64);
s2 = malloc(sizeof (struct s) + 46);
```

и, предполагая, что вызовы **malloc()** были успешны, объекты, на которые указывают **s1** и **s2**, ведут себя так, как если бы идентификаторы были объявлены как:

```
struct { int n; double d[8]; } *s1;
struct { int n; double d[5]; } *s2;
```

Если следующие два присваивания были выполнены успешно:

```
s1 = malloc(sizeof (struct s) + 10);
s2 = malloc(sizeof (struct s) + 6);
```

то они будут вести себя так, как будто вместо них были объявления:

```
struct { int n; double d[1]; } *s1, *s2;
```

и:

```
double *dp;
 dp = &(s1->d[0]); // годится
*dp = 42;          // годится
 dp = &(s2->d[0]); // годится
```

Присваивание:

```
*s1 = *s2;
```

копирует только член **n**. При этом, если какой-либо из элементов массива находится в пределах первых **sizeof (struct s)** байт структуры, они могут быть скопированы или просто перезаписаны неопределенными значениями.

Пример 3.

Поскольку члены анонимных структур и объединений считаются членами содержащей их структуры или объединения, **struct s** имеет более одного именованного члена, поэтому использование элемента «болтающегося» массива допустимо:

```
struct s {
    struct {
        int i;
    };
    int a[];
};
```

Спецификаторы перечислений

Синтаксис

спецификатор-перечисления:

enum *[идентификатор]* **{** *список-перечислителей* **}**

enum *[идентификатор]* **{** *список-перечислителей* **, }**

enum *идентификатор*

список-перечислителей:

перечислитель

список-перечислителей **,** *перечислитель*

перечислитель:

константа-перечисления

константа-перечисления **=** *константное-выражение*

Ограничения

Выражение, которое определяет значение константы перечисления, должно быть целочисленным константным выражением, значение которого представлено в виде **int**.

Семантика

Идентификаторы в *списке-перечислителей* объявляются как константы, имеющие тип **int**, и могут появляться везде, где разрешены константы типа **int**. Поэтому, идентификаторы *констант-перечисления*, объявленные в одной и той же области видимости, должны отличаться друг от друга и от других идентификаторов, которые объявлены в обычных объявлениях.

Перечислитель с оператором **=** определяет свою константу перечисления как значение константного выражения. Если первый перечислитель не имеет оператора **=**, значение его константы перечисления равно 0. Каждый последующий перечислитель, не имеющий оператора **=** определяет свою константу перечисления, как значение константного выражения, полученного путем добавления 1 к значению предыдущей константы перечисления. (Использование перечислителей с оператором **=** может создавать константы перечисления со значениями, которые дублируют другие значения в том же перечислении.) Перечислители перечисления также известны как его *члены*.

```
enum car {volvo, honda, toyota, lada = 1};
enum car {
    toyota,   //
    honda,    //
    volvo,    //
    lada = 1, //
};
```

Перечисляемый тип является неполным до тех пор, пока не встретится **}**, которая завершает объявления списка перечислителей, и становится полным после **}**.

Пример

Следующий фрагмент:

```
enum hue {
    chartreuse, // Шартрез (желтый или зеленый ликер)
    burgundy,   // Бургундское (темное красное/краснокоричневое)
    claret=20,  // Кларет (сухое красное, бордо)
    chernila    // Плодово-ягодное
};
enum hue col, *cp;
col = claret;
cp = &col;
if (*cp != burgundy)
/* ... */
```

делает **hue** тегом перечисления, после чего объявляет **col** в качестве объекта с этим типом и **cp** как указатель на объект с этим типом.

Значения перечисления находятся в наборе **{0, 1, 20, 21}**.

Теги (самостоятельно)

Семантика

Все объявления структур, объединений или перечислимых типов, которые имеют одинаковую область видимости и используют один и тот же *тег*, объявляют один и тот же тип.

Независимо от того, есть ли тег или какие другие объявления типа находятся в той же единице трансляции, тип является неполным вплоть до закрывающей скобки списка, определяющего содержимое, и становится полной (завершенной) после нее.

Неполный тип может использоваться только тогда, когда размер объекта этого типа не нужен. Размер объекта не требуется, например, когда объявляется **`typedef`** имя в качестве спецификатора для структуры или объединения, или когда объявляется указатель или функция, возвращающая структуру или объединение. Спецификация должна быть завершена до того, как такая функция может быть вызвана или определена.

Два объявления структур, объединений или перечислимых типов, которые находятся в разных областях видимости или используют разные теги, объявляют разные типы. Каждое объявление структуры, объединения или перечислимого типа, которое не включает тег, объявляет отдельный тип.

Спецификатор типа в форме

struct *[идентификатор]* **{** *список-объявлений-структуры* **}**
union *[идентификатор]* **{** *список-объявлений-структуры* **}**
или
enum *[идентификатор]* **{** *список-перечислителей [,]***}**

объявляет структуру, объединение или перечислимый тип. Список определяет *содержимое структуры*, *содержимое объединения* или *содержимое перечисления*.

Если указан идентификатор, спецификатор типа одновременно объявляет идентификатор тегом данного типа.

Если объявление имеет **typedef**-имя, последующие объявления могут использовать это **typedef**-имя для объявления объектов, имеющих указанный тип структуры, тип объединения или перечислимый тип.

Объявления в форме

struct/union *идентификатор* **;**

указывает тип структуры и тип объединения, а также объявляют идентификатор в качестве тега данного типа. Подобной конструкции с **enum** (без списка) не существует.

Пример 1 Объявление структуры, которая ссылается сама на себя:

```
struct tnode {
    int count;
    struct tnode *left;
    struct tnode *right;
};
```

Данный код определяет структуру, которая содержит целое число и два указателя на объекты того же типа, что и сама структура.

Как только выдано это объявление, объявление

```
struct tnode s, *sp;
```

объявляет **s** как объект данного типа, а **sp** как указатель на объект данного типа.

После этих объявлений выражение **sp->left** является указателем на левую структуру относительно объекта, на который указывает **sp**, а выражение **s.right->count** обозначает член **count** правой структуры с тегом **tnode**, на которую указывает **s**.

Следующая альтернативная формулировка использует механизм **typedef**:

```
typedef struct tnode TNODE;
struct tnode {
    int count;
    TNODE *left, *right;
};
TNODE s, *sp;
```

Пример 2.

Чтобы проиллюстрировать использование предварительного объявления тега для указания пары взаимно ссылочных структур, годятся объявления

```
struct s1 {                  // D1
    struct s2 *s2p; /* ... */
};

struct s2 {                  // D2
    struct s1 *s1p; /* ... */
};
```

Они указывают пару структур, которые содержат указатели друг на друга. Однако...

Однако следует обратить внимание, что если **s2** уже был объявлен как тег во внешней области, объявление **D1** будет ссылаться на него, а не на тег **s2**, объявленный в **D2**.

```
struct s2;
...
{
    struct s1 {                 // D1
        struct s2 *s2p; /* ... */
    };

    struct s2 {                 // D2
        struct s1 *s1p; /* ... */
    };
}
```

Чтобы устранить эту контекстную чувствительность, можно вставить перед **D1** объявление

```
struct s2;
```

Она объявляет новый тег **s2** во внутренней области видимости, а объявление **D2** после него завершает спецификацию нового типа.

Ограничения

Конкретный тип должен иметь свое содержимое, *определенное* не более одного раза.

Если два объявления, использующие один и тот же *тег*, объявляют один и тот же тип, они оба должны использовать один и тот же выбор **`struct`**, **`union`** или **`enum`**.

Спецификатор типа в форме

```
enum идентификатор
```

без списка перечислителей должен появляться только после того, как указанный тип будет завершен (полным).

Спецификаторы атомарного типа

Синтаксис

спецификатор-атомарного-типа:

_Atomic (*имя-типа* **)**

Ограничения

Если компилятор не поддерживает атомарные типы, спецификаторы атомарных типов не будут доступны.

Имя типа в спецификаторе атомарного типа не должно ссылаться на тип массива, тип функции, атомарный тип или квалифицированный тип.

Семантика

Объекты атомарных типов являются единственными объектами, свободными от *гонок данных*, то есть они могут быть изменены двумя потоками одновременно или изменены одним и прочитаны другим.

Свойства, связанные с атомарными типами, имеют смысл только для выражений, которые являются **lvalues**.

Если за ключевым словом **_Atomic** сразу следует левая скобка, оно интерпретируется как спецификатор типа (с именем типа), а не как квалификатор типа.

Пример

```
#include <stdio.h>

unsigned long           a1; // неатомарный тип
_Atomic (unsigned long) a4; // атомарный тип введен спецификатором
_Atomic unsigned long   a2; // атомарный тип введен квалификатором
unsigned long _Atomic   a3; // атомарный тип введен квалификатором

int main (int argc, char *argv[]) {

    a1 = 0x0123456789ABCDEF;
    a2 = 0x123456789ABCDEF0;
    a3 = 0x23456789ABCDEF01;

    a4 = a1;
    a4 = a2;
    a4 = a3;
    a1 = a4;

    printf("%lux %lux %lux %lux\n", a1, a2, a3, a4);

}
```

```
a1 = 0x0123456789ABCDEF;
        movabs  rax, 81985529216486895
        mov     QWORD PTR a1[rip], rax

a2 = 0x123456789ABCDEF0;
        movabs  rax, 1311768467463790320
        mov     QWORD PTR a2[rip], rax
        mfence

a3 = 0x23456789ABCDEF01;
        movabs  rax, 2541551405711093505
        mov     QWORD PTR a3[rip], rax
        mfence
```

MFENCE выполняет операцию сериализации для всех инструкций загрузки из памяти и сохранения в память, которые были выданы до инструкции MFENCE. Она гарантирует, что каждая инструкция загрузки и сохранения, которая предшествует инструкции MFENCE в программном порядке, становится глобально видимой перед любой инструкцией загрузки или сохранения, следующей за инструкцией MFENCE (инструкция загрузки считается глобально видимой, когда значение, которое должно быть загружено в ее регистр назначения является определенным).

Компилятор gcc 11.2 выдает:

```
 a1 = 0x0123456789ABCDEF;
        movabs  rax, 81985529216486895
        mov     QWORD PTR a1[rip], rax
a2 = 0x123456789ABCDEF0;
        movabs  rax, 1311768467463790320
        xchg    rax, QWORD PTR a2[rip]
 a3 = 0x23456789ABCDEF01;
        movabs  rax, 2541551405711093505
        xchg    rax, QWORD PTR a3[rip]
  a4 = a1;
        mov     rax, QWORD PTR a1[rip]
        xchg    rax, QWORD PTR a4[rip]
  a4 = a2;
        mov     rax, QWORD PTR a2[rip]
        xchg    rax, QWORD PTR a4[rip]
  a4 = a3;
        mov     rax, QWORD PTR a3[rip]
        xchg    rax, QWORD PTR a4[rip]
  a1 = a4;
        mov     rax, QWORD PTR a4[rip]
        mov     QWORD PTR a1[rip], rax
```

Квалификаторы типа

Квалификаторы типа модифицируют тип, введенный спецификаторами типа.

квалификатор-типа:

```
const
restrict          (применимо только для указателей)
volatile
_Atomic
```

Общее для всех

Свойства, связанные с квалифицированными типами, имеют смысл только для выражений, которые являются **lvalues**.

Если другие квалификаторы появляются вместе с квалификатором **_Atomic** в *списке-спецификторов-квалификаторов*, то результирующий тип — это так называемый атомарный тип.

Если спецификация типа массива включает какие-либо квалификаторы типов, то этими квалификаторами модифицируется тип элементов, а не тип массива.

Если какие-либо квалификаторы типа включает в себя спецификация типа функции, поведение не определено.

const

Если предпринята попытка изменить объект, определенный с использованием **const**-квалификатора, с помощью **lvalue**, не квалифицированного, как **const**, поведение не определено.

volatile

Объект, имеющий **volatile**-квалифицированный тип, может быть изменен способами, неизвестными компилятору (операции ввода-вывода), или иметь какие-нибудь неизвестные побочные эффекты.

volatile объявление может использоваться для описания объекта, соответствующего отображенному на память *порту ввода/вывода*, или объекта, к которому обращается функция асинхронного прерывания.

Действия над объектами, объявленными таким образом, не «оптимизируются» компилятором и не переупорядочиваются, за исключением случаев, разрешенных правилами для вычисления выражений.

`restrict`

Квалификатор **`restrict`** требует, чтобы все обращения к этому объекту выполнялись прямо или косвенно через значение *только этого конкретного* указателя.

Например, оператор, который присваивает значение, возвращаемое **`malloc()`**, единственному указателю, устанавливает такую связь между выделенным объектом и указателем.

Предполагается, что использование квалификатора **`restrict`** (подобно использованию класса памяти **`register`**) должно способствовать исключительно оптимизации — удаление всех экземпляров квалификатора **`restrict`** из всех исходных *единиц трансляции*, подаваемых на вход препроцессора, не изменит наблюдаемое поведение программы.

Квалификатор **`restrict`** позволяет сообщить компилятору, что объявляемый указатель указывает на объект, на который не указывает никакой другой указатель.

Гарантию того, что на один объект не будет указывать более одного указателя,

даёт программист и только он.

При этом оптимизирующий компилятор может генерировать более эффективный код. Использование квалификатора **`restrict`** при объявлении объектов, не являющихся указателями, не допускается.

При использовании ключевого слова **`restrict`** программа, написанная на C, может сравниться по скорости с программой, написанной на Fortran.

В языке C++ нет ключевого слова **`restrict`** (не описано в стандарте), но разработчики разных компиляторов C++ добавили аналогичные по назначению ключевые слова, например:

- **`__restrict`** и **`__restrict__`** у GNU Compiler Collection;
- **`__restrict`** и **`__declspec(restrict)`** у Visual C++;
- **`__restrict__`** у Clang.

Указатель с модификатором **restrict** на указатель с модификатором **restrict** может быть определён только во вложенном блоке. Пример:

```
struct T {
    int i;
};
struct T s;

int main() {
   struct T * restrict p = &s;
   int * restrict pp     = &p->i; // undefined behavior
   {
      int * restrict ppp  = &p->i; // допустимо
   }
   return 0;
}
```

Определение указателя **pp** — неопределённое поведение, так как **pp** находится в одном блоке с **p**.

Определение **ppp** находится во вложенном блоке и поэтому допустимо.

Примеры

Пример 1

Объект **real_time_clock**, объявленный как

```
extern const volatile int real_time_clock;
```

может изменяться аппаратным обеспечением, но не может быть увеличен или уменьшен, а так же ему не может быть присвоено значение.

Тот же пример, но более «видимый»

```
extern
const
volatile
int real_time_clock;
```

Пример 2. Квалификация агрегатного типа

Следующие объявления и выражения иллюстрируют поведение, когда квалификаторы типов изменяют агрегатный тип:

```
const struct s { // const относится к объекту cs, но не к объявлению s
    int mem;
} cs = { 1 };    // поэтому объект cs не может модифицироваться,
struct s ncs;    // а объект ncs может

const struct ss { // предупреждение: бесполезный квалификатор типа в
    int mem;      // пустой декларации
};                //

typedef int A[2][3];           // тип -- массив массивов элементов типа int
const A a = {{4,5,6},{7,8,9}}; // массив массивов элементов типа const int

int       *pi;
const int *pci;

ncs = cs;        // допустимо
cs  = ncs;       // нарушает ограничение на модификацию lvalue через =

pi  = &ncs.mem;  // допустимо
pi  = &cs.mem;   // нарушает – попытка получить доступ по записи
pci = &cs.mem;   // допустимо
pi = a[0];       // недопустимо: a[0] имеет тип ‘‘const int *’’
```

Пример 3

Объявление

```
_Atomic volatile int *p;
```

указывает, что **p** имеет тип «указатель на **volatile atomic int**», это тип указателя на **volatile**-квалифицированный атомарный тип.

Пример 4 restrict

Объявления в области видимости файла

```
int * restrict a;
int * restrict b;
extern int c[];
```

утверждают, что если доступ к объекту осуществляется с помощью одного из **a**, **b** или **c**, и этот объект изменяется в любом месте программы, то к нему никогда не будет получен доступ с использованием любого из двух других.

Пример 5

Объявления параметров функции в следующем примере

```
void f(int n,
       int * restrict dd, int * restrict ss) {
    while (n-- > 0) {
        *dd++ = *ss++;
}   }
```

утверждают, что во время каждого выполнения функции, если к объекту получают доступ через один из параметров указателя, то к нему также нет доступа через другой.

Преимущество квалификаторов **`restrict`** состоит в том, что они позволяют транслятору сделать эффективный анализ зависимостей для функции **`f()`** без проверки каких-либо вызовов функции **`f()`** в программе.

Программист должен изучить все эти вызовы, чтобы убедиться, что ни один из них не дает неопределенного поведения. Например, второй вызов функции **`f()`** в **`g()`** имеет неопределенное поведение, поскольку доступ к каждому из элементов, начиная с **`d[1]`** по **`d[49]`**, осуществляется как через **`dd`**, так и через **`ss`**.

```
void g(void) {
    extern int d[100];
    f(50, d + 50, d); // valid
    f(50, d + 1, d);  // undefined behavior
}
```

Пример 6

Объявления параметров функции

```
void h(int n,              // параметр 1
       int * restrict p,  // параметр 2
       int * restrict q,  // параметр 3
       int * restrict r)  // параметр 4
{
    int i;
    for (i = 0; i < n; i++) {
        p[i] = q[i] + r[i];
    }
}
```

иллюстрирует, как объект может быть изменен через два **restrict**-квалифицированных указателя.

Например, если **a** и **b** являются непересекающимися массивами, вызов вида

```
    h(100, a, b, b)
```

имеет определенное поведение, поскольку массив **b** не изменяется в функции **h**.

Пример 7

Правило, ограничивающее присваивания между **restrict**-указателями, не различает вызов функции и эквивалентный вложенный блок.

За одним единственным исключением, определенное поведение имеют только присвоения типа «внешние-внутренним» между **restrict**-указателями, объявленными во вложенных блоках.

```
{
    int * restrict p1;
    int * restrict q1;
    p1 = q1; // поведение неопределено
    {
        int * restrict p2 = p1; // допустимо
        int * restrict q2 = q1; // допустимо
        p1 = q2; // поведение неопределено -- вынос указателя во внешний блок
        p2 = q2; // поведение неопределено -- вынос указателя во внешний блок
    }
}
```

Единственное исключение позволяет вынести значение **restrict**-указателя из блока, в котором он (или, точнее, обычный идентификатор, используемый для его обозначения) объявлен — это когда данный блок заканчивает выполнение. Например, это позволяет функции **new_vector()** возвратить **vector**.

```
typedef struct {
    int n;
    float * restrict v;
} vector;

vector new_vector(int n) {

    vector t;  //

    t.n = n;
    t.v = malloc(n * sizeof (float));

    return t;
}
```

typedef — определения типов

Синтаксис

typedef *объявление-типа* **;**

typedef *определение-типа* *идентификатор* **;**

Семантика

В объявлении, спецификатором класса памяти которого является **typedef**, каждый декларатор определяет идентификатор, который следует трактовать, как имя определения типа (typedef-имя), которое обозначает тип, указанный для этого идентификатора обычным способом.

Строго говоря, **typedef**-объявление не вводит новый тип, а только синоним для указанного типа. В следующих объявлениях:

```
typedef T type_ident;
type_ident D;
```

type_ident определяется как имя определения типа с типом, указанным спецификаторами объявления в **T** (тип, известный как **T**), а идентификатор в **D** имеет тип «список-производных-деклараторов-типа **T**», где «список-производных-деклараторов-типа» указывается декларатором **D**.

Пример 1
После

```
typedef int MILES, KLICKSP( );
typedef struct {
    double hi, lo;
} range;
```

конструкции

```
MILES distance;
extern KLICKSP *metricp;
range x;
range z, *zp;
```

все являются действительными определениями.
Тип **distance — int**,
тип **metricp** — «указатель на функцию без параметров, возвращающую **int**»,
типы **x** и **z** — указанная структура;
zp — указатель на такую структуру.
Объект **distance** имеет тип, совместимый с любым другим объектом типа **int**.

Пример 2
После объявления

```
typedef struct s1 {
    int x;
} t1, *tp1;

typedef struct s2 {
    int x;
} t2, *tp2;
```

тип **t1** и тип, на который указывает **tp1**, совместимы.
Тип **t1** также совместим с типом **struct s1**, но не совместим с типами **struct s2**, **t2**, а также типом, на который указывает **tp2** или **int**.

Пример 3

Следующие непонятные (маловразумительные) конструкции

```
typedef signed int t;       // имя определения типа с типом signed int
typedef int        plain;   //  имя определения типа с типом int

// структурf с тремя членами – битовыми полями
struct tag {
    unsigned  t:4; // t -- имя члена структуры (unsigned – специф. типа)
    const t:5;     // t –– имя типа, совместимого с signed int
    plain r:5;     // r -- имя члена структуры
};
```

объявляют имя определения типа **t** с типом **signed int**, имя определения типа с типом **int** и структуру с тремя членами битового поля,

- первый с именем **t**, который содержит значения в диапазоне [0, 15];
- непоименованное битовое поле с квалификацией **const**, которое (если к нему можно получить доступ) будет содержать значения в диапазоне [−15, +15] или [−16, +15];
- третий из них с именем **r** будет содержать значения в одном из диапазонов [0, 31], [−15, +15] или [−16, +15].

(Выбор диапазона определяется реализацией.)

Первые два объявления битовых полей отличаются тем, что **unsigned** является спецификатором типа, поэтому **t** является именем члена структуры, тогда как **const** является квалификатором типа (который модифицирует **t**, который здесь виден как имя определения типа).

```
typedef signed int t;
typedef int        plain;
```

Если после этих объявлений во внутренней области встречаются

```
t f(t(t));
long t;
```

тогда первое трактуется, как функция **f**, которая объявляется с типом «функция, возвращающая **signed int**, с одним непоименованным параметром с типом указателя на функцию, возвращающую **signed int**», а второе — как идентификатор **t** с типом **long int**.

Пример 4.

Имена определения типа могут использоваться для улучшения читабельности кода. Все три из следующих объявлений функции **signal** указывают точно один и тот же тип, причем, первый без использования каких-либо имен определения типа.

```
typedef void fv(int), (*pfv)(int);

void (*signal(int, void (*)(int)))(int);
fv *signal(int, fv *);
pfv signal(int, pfv);
```

signal() - работа с сигналами в C

```
#include <signal.h>

typedef void (*sighandler_t)(int);

sighandler_t signal(int signum, sighandler_t handler);
```

```
typedef int (*foo)(int);
int (*foo)(int);

typedef int *foo(int);
int *foo(int);
```

Пример 5

Если имя определения типа обозначает тип массива переменной длины, длина массива фиксируется во время определения typedef-имени, а не каждый раз, когда оно используется:

```
void copyt(int n) {

    typedef int B[n];  //  B -- n int'ов, n вычисляется именно сейчас
    n += 1;
    B a;               // a -- n int'ов, n без учета += 1
    int b[n];          // a и b имеют разные размеры
    for (int i = 1; i < n; i++) {
        a[i-1] = b[i];
    }
}
```

Спецификаторы функций

спецификатор-функции:

`inline`

`_Noreturn`

`inline`

Функция, объявленная со спецификатором **`inline`**, является *встроенной функцией*.

Создание функции, как встроенной, предполагает, что вызовы функции должны быть максимально быстрыми. Это может достигаться с использованием механизма, альтернативного обычному вызова функции, например «встроенная подстановка/замена (inline substitution)».

Степень эффективности указания спецификатора **`inline`** определяется транслятором.

Любая функция с внутренним типом связывания может быть встроенной функцией.

Для функции с внешним типом связывания применяются следующие ограничения:

- если функция объявлена со спецификатором **`inline`**, то она должна быть определена в той же *единице трансляции*.

- если все объявления функции в единице трансляции в области видимости файла включают спецификатор **`inline`** без **`extern`**, то определение функции в этом модуле трансляции является *встроенным определением* (*inline definition*).

Объявление встроенной функции с внешним типом связывания может привести либо к внешнему определению, либо к определению, доступному для использования только внутри единицы трансляции. Объявление в области видимости файла с помощью **extern** создает внешнее определение. В следующем примере показана полностью единица трансляции.

```
inline double fahr(double t) {          // встроенное определение fahr
    return (9.0 * t) / 5.0 + 32.0;
}
inline double cels(double t) {          // встроенное определение cels
    return (5.0 * (t - 32.0)) / 9.0;    // имеет внешний тип связывания
}
extern double fahr(double);             // создает внешнее определение fahr
double convert(int is_fahr, double temp) {
/* Транслятор может выполнить inline подстановку */
    return is_fahr ? cels(temp) : fahr(temp);
}
```

Следует заметить, что определение **fahr()** является внешним определением, поскольку **fahr()** также объявляется с помощью **extern**, но определение **cels()** является встроенным определением.

Поскольку **cels()** имеет внешний тип связывания и на нее ссылаются, внешнее определение должно появиться в другой единице трансляции. Встроенное определение и внешнее определение различны и могут оба использоваться для вызова.

_Noreturn

Функция, объявленная со спецификатором **_Noreturn**, не должна возвращать управление вызывающей стороне.

Компилятор обычно выдает диагностическое сообщение для функции, объявленной с помощью спецификатора функции **_Noreturn**, которая по его мнению способна вернуть управление вызвавшему ее коду.

```
_Noreturn void f() {
    abort(); // ok
}

_Noreturn void g(int i) { // causes undefined behavior if i <= 0
    if (i > 0) {
        abort();
    }
}
```

Спецификаторы выравнивания

спецификатор-выравнивания:

```
_Alignas ( имя-типа )
_Alignas ( константное-выражение )
```

Первая форма эквивалентна форме **`_Alignas (_Alignof (`***имя-типа***`))`**.

Спецификация выравнивания, равная нулю, не имеет никакого эффекта и не влияет на другие спецификации выравнивания в том же объявлении.

Если **определение** объекта имеет спецификатор выравнивания, любое другое **объявление** этого объекта должно либо указывать эквивалентное выравнивание, либо не иметь спецификатора выравнивания.

Если определение объекта не имеет спецификатора выравнивания, любое другое объявление этого объекта также не должно иметь спецификатора выравнивания.

Ограничения

Атрибут выравнивания не должен указываться в объявлении **`typedef`**, или битового поля, или функции, или параметра, или объекта, объявленного с помощью спецификатора класса памяти **`register`**.

Вторая форма

`_Alignas (` *константное-выражение* **`)`**

Константное выражение должно быть целочисленным константным выражением. Оно должен вычислять основное выравнивание или допустимое расширенное выравнивание, поддерживаемое в контексте, в котором оно появляется, или ноль.

Комбинированный эффект всех атрибутов выравнивания в объявлении не должен приводить к выравниванию, которое является менее строгим, чем выравнивание, которое в противном случае потребовалось бы для типа объекта или члена, подлежащих объявлению.