24.02.2025

Библиотека C. Макросы и препроцессор. static, extern. Прогрессия.

Филиппов Михаил Витальевич

m.filippov@g.nsu.ru

89232283872

Императивное программирование, 2024-2025

Новосибирский государственный университет

*НАСТОЯЩАЯ НАУКА

Давайте познакомимся

Филиппов Михаил Витальевич

- Окончил магистратуру ФФ НГУ
- Окончил аспирантуру ИТ СО РАН
- Являюсь м.н.с. ИТ СО РАН
- 7+ лет опыт в программировании C/C++

Адженда

Библиотека C	Макросы и препроцессор	Static, extern	Прогрессия
20 минут	45 минут	10 минут	15 минут

Адженда

Библиотека C	Макросы и препроцессор	Static, extern	Прогрессия
15 минут	45 минут	15 минут	15 минут

Введение

Первоначально официальной библиотеки C не существовало. Позже возник стандарт де-факто, основанный на реализации C для Unix. Комитет ANSI C, в свою очередь, разработал официальную стандартную библиотеку, которая в значительной степени базировалась на этом стандарте де-факто. Учитывая распространение языка C по всему миру, комитет затем решил переопределить библиотеку, чтобы она могла быть реализована в широком разнообразии систем.

Получение доступа к библиотеке C:

- Автоматический доступ
- Включение файлов
- Включение библиотек

Использование описаний библиотеки

Документацию по функциям можно найти в нескольких местах. Система может иметь онлайновое руководство, а IDE-среда часто располагает онлайновой справкой. Поставщики компиляторов C иногда предоставляют руководства пользователя в печатном виде и на сайте, которые содержат описание библиотечных функций. Многие издательства выпустили справочные пособия по функциям библиотеки C. Одни из них имеют общую природу, а другие ориентированы на определенные реализации языка.

Библиотека математических функций

Библиотека математических функций содержит множество удобных функций такого рода. Их объявления или прототипы содержатся в заголовочном файле math.h. Обратите внимание, что все углы измеряются в радианах (один радиан составляет 180/π = 57.296 градуса).
Примеры интересных математических функций:

Прототип	Описание
double acos(double x)	Возвращает угол (от 0 до π радиан), косинус которого равен x
double atan2(double y, double x)	Возвращает угол (от - π до π радиан), тангенс которого равен y/x
double cbrt (double x)	Возвращает кубический корень x
double pow(double x, double y)	Возвращает x в степени y
double log(double x)	Возвращает натуральный логарифм x
double sqrt(double x)	Возвращает квадратный корень x
double ceil(double x)	Возвращает наименьшее целое, которое не меньше x
double floor(double x)	Возвращает наибольшее целое, которое не больше x

Библиотека математических функций – важно!

Если в процессе компиляции будет выдано сообщение, такое как
Undefined: _sqrt
Не определено: _sqrt
или
'sqrt': unresolved external
'sqrt': нераспознанный внешний идентификатор
либо нечто подобное, значит, компилятор-компоновщик не смог найти библиотеку математических функций. В системах Unix может потребоваться указать компоновщику на необходимость поиска библиотеки математических функций с помощью флага **- lm**:

cc main.c - lm

Обратите внимание, что флаг **- lm** находится в конце команды. Причина в том, что компоновщик вступает в игру после того, как компилятор скомпилирует файл С. Компилятор GCC в системе Linux может вести себя в такой же манере:

gcc main.c -lm

Варианты типов

```c
#include <stdio.h>
#include <math.h>
#define RAD_TO_DEG (180 / (4 * atanl(1)))
// обобщенная функция извлечения квадратного корня
#define SQRT(X) _Generic((X), \
    long double: sqrtl,       \
    default: sqrt,            \
    float: sqrtf)(X)
// обобщенная функция вычисления синуса угла, заданного в градусах
#define SIN(X) _Generic((X),                \
    long double: sinl((X) / RAD_TO_DEG), \
    default: sin((X) / RAD_TO_DEG),      \
    float: sinf((X) / RAD_TO_DEG))
int main(void) {
    float x = 45.0f;
    double xx = 45.0;
    long double xxx = 45.0L, yy = SQRT(xx);
    long double y = SQRT(x), yyy = SQRT(xxx);
    printf("%.17Lf\n", y);   // соответствует float
    printf("%.17Lf\n", yy);  // соответствует default
    printf("%.17Lf\n", yyy); // соответствует long double
    int i = 45;
    yy = SQRT(i);
    printf("%.17Lf\n", yy); // соответствует default
    yyy = SIN(xxx);
    printf("%.17Lf\n", yyy); // соответствует long double
    return 0;
}
```

В стандарте C предоставляются версии стандартных функций типа float и типа long double, имеющие в имени суффикс f или l (строчная буква “L”). Таким образом, sqrtf() — это версия типа float функции sqrt(), а sqrtl() — версия типа long double функции sqrt().

```
// 6.7082037925720214 8
// 6.7082039324993694 2
// 6.7082039324993690 9
// 6.7082039324993694 2
// 0.7071067811865475 2
```

Библиотека tgmath.h (C99)

Стандарт C99 предлагает заголовочный файл tgmath.h, в котором определены макросы обобщенного типа, по своему действию похожие на те, что были показаны в примере. Если какая-то функция math.h определена для каждого из трех типов float, double и long double, то файл tgmath.h создает макрос обобщенного типа с тем же именем, что и у версии для double. Например, он определяет макрос sqrt() , который разворачивается в функцию sqrtf(), sqrt() или sqrtl() в зависимости от типа предоставленного аргумента. Другими словами, макрос sqrt() ведет себя подобно макросу SQRT ().

Если компилятор поддерживает арифметику комплексных чисел, то в нем доступен файл complex.h, в котором объявлены комплексные аналоги математических функций.

Пример:

```
#include <tgmath.h>
float x = 44.0;
double y;
y = sqrt(x); // вызов макроса, следовательно sqrtf(x)
y = (sqrt)(x); // вызов функции sqrt()
```

Средство выражений _Generic, добавленное в стандарт C11, является простым способом реализации макросов tgmath.h, не прибегая к механизмам, которые выходят за рамки стандарта C.

Библиотека утилит общего назначения

Библиотека утилит общего назначения содержит множество функций, включая генератор случайных чисел, функции для поиска и сортировки, функции для преобразования и функции для управления памятью. В ANSI C прототипы функций rand(), srand(), malloc () и free () находятся в заголовочном файле stdlib.h.
Следующие слады о ней..

Использование функции atexit()

Все-таки есть функция, которая принимает указатели на функции! Чтобы использовать ее, просто передайте адрес функции, которая должна быть вызвана при выходе.

Поскольку имя функции действует как адрес, когда применяется в качестве аргумента функции, для аргумента можно указывать имя. Затем atexit() регистрирует эту функцию в списке функций, предназначенных для выполнения при вызове exit(). Стандарт ANSI гарантирует, что список может вмещать не менее 32 функций. Каждая из них добавляется с помощью отдельного вызова atexit(). Когда функция exit(), в конце концов, вызывается, она выполняет эти функции, причем первой выполняется функция, добавленная в список последней.

Функции, регистрируемые atexit() должны иметь тип void и не принимать аргументов. Обычно они решают вспомогательные задачи, такие как обновление файла мониторинга программы или переустановка переменных среды.

Использование функции exit()

Когда exit() выполняет функции, указанные с помощью atexit(), она предпринимает собственные шаги по очистке. Функция exit() сбрасывает все потоки вывода, закрывает все открытые потоки и закрывает временные файлы, созданные в результате обращений к стандартной функции ввода-вывода tmpfile(). Затем exit() возвращает управление размещаемой среде и по возможности сообщает среде состояние завершения. Традиционно программы для Unix применяли 0, чтобы указать на успешное завершение, и ненулевое значение для сообщения об отказе. Коды возврата Unix не обязательно работают со всеми системами, поэтому в ANSI C определен макрос по имени EXIT_FAILURE, который может использоваться переносимым образом для обозначения отказа. Аналогично, в ANSI C определен макрос EXIT_SUCCESS для указания на успешное завершение, но exit() также принимает для этой цели значение 0. В рамках стандарта ANSI C применение exit() в не рекурсивной функции main() эквивалентно использованию ключевого слова return. Тем не менее, exit() также завершает программу, когда применяется в функциях, отличных от main().

Функции exit() и atexit()

```c
#include <stdio.h>
#include <stdlib.h>
void sign_off(void);
void too_bad(void);
int main(void) {
    int n;
    atexit(sign_off); /* регистрация функции sigu_off() */
    puts("Введите целое число:");
    if (scanf("%d", &n) != 1) {
        puts("Это не целое число!");
        atexit(too_bad); /* регистрация функции too_bad() */
        exit(EXIT_FAILURE);
    }
    printf("%d является %s.\n", n, (n % 2 == 0) ? "четным" : "нечетным");
    return 0;
}
void sign_off(void) {
    puts("Завершение работы очередной замечательной программы от");
    puts("SeeSaw Software!");
}
void too_bad(void) {
    puts("SeeSaw Software приносит искренние соболезнования");
    puts("в связи с отказом программы.");
}
```

```
Введите целое число:
212
212 является четным.
Завершение работы очередной
замечательной программы от
SeeSaw Software!
```

```
Введите целое число:
.
Это не целое число!
SeeSaw Software приносит
искренние соболезнования
в связи с отказом программы.
Завершение работы очередной
замечательной программы от
SeeSaw Software!
```

Библиотека утверждений

Библиотека утверждений, поддерживаемая заголовочным файлом assert.h — это небольшая библиотека, предназначенная для оказания содействия при отладке программы. Она состоит из макроса по имени assert(). Макрос принимает в качестве аргумента целочисленное выражение. Если выражение оценивается как ложное (ненулевое), макрос assert() выводит в стандартный поток ошибок (stderr) сообщение об ошибке и вызывает функцию abort(), которая прекращает выполнение программы. (Прототип функции abort() находится в заголовочном файле stdlib.h.) Идея состоит в том, чтобы идентифицировать критические места в программе, где должны быть истинными определенные условия, и с помощью оператора assert () завершать программу, если одно из указанных условий нарушается. Обычно аргументом служит выражение отношения или логическое выражение. Когда assert() прекращает выполнение программы, сначала отображается не прошедший проверку тест, имя файла, содержащего этот тест, и номер строки, где находится тест.

Использование assert()

```c
#include <stdio.h>
#include <math.h>
#include <assert.h>
int main()
{
    double x, y, z;
    puts("Введите пару чисел (0 0 для завершения):");
    while (scanf("%lf%lf", &x, &y) == 2 && (x != 0 || y != 0))
    {
        z = x * x - y * y; /*должно быть + */
        assert(z >= 0);
        printf("результатом является %f\n", sqrt(z));
        puts("Введите следующую пару чисел: ");
    }
    puts("Программа завершена.");
    return 0;
}
```

```
Введите пару чисел (0 0 для завершения):
4 3
результатом является 2.645751
Введите следующую пару чисел:
5 3
результатом является 4.000000
Введите следующую пару чисел:
3 5
main: main.c:11: main: Assertion `z >= 0' failed.
Aborted
```

Использование assert()

```
if (z < 0) {
    puts("z меньше 0");
    abort();
}
```

Подход с assert() обладает рядом преимуществ.

- Он автоматически идентифицирует файл и номер строки, где возникла проблема.
- Существует механизм включения и отключения макроса assert() без необходимости в изменении кода.

Если вы считаете, что устранили ошибки в программе, поместите следующее определение макроса

#define NDEBUG

перед местом включения файла assert.h, повторно скомпилируйте программу, и компилятор отключит в файле все операторы assert().

_Static_assert(C11)

Выражение assert() проверяется во время выполнения. В C11 появилось новое средство в форме объявления _Static_assert, которое осуществляет проверку на этапе компиляции. Таким образом, assert() Может привести к прерыванию выполняющейся программы, тогда как _Static_assert() может стать причиной того, что программа не компилируется. Объявление _Static_assert принимает два аргумента. Первым из них является целочисленное константное выражение, а вторым - строка.

```c
#include <stdio.h>
#include <limits.h>
_Static_assert(CHAR_BIT == 16, "Incorrectly assumes 16-bit char type");
int main(void)
{
    puts("Тип char имеет 16 битов.");
    return 0;
}
```

```
clang  main.c -o main -std=c11
main.c:3:1: error: static_assert failed due to requirement '8 == 16' "Incorrectly assumes 16-bit char type"
_Static_assert(CHAR_BIT == 16, "Incorrectly assumes 16-bit char type");
^              ~~~~~~~~~~~~~~
1 error generated.
```

Функции memcpy () и memmove() из библиотеки string.h

Присваивать один массив другому нельзя, поэтому в таких случаях мы применяли циклы для поэлементного копирования одного массива в другой. Единственное исключение состоит в том, что для символьных массивов мы использовали функции strcpy() и strncpy(). Функции memcpy() и memmove() предлагают почти такие же услуги для других видов массивов. Рассмотрим прототипы этих функций:

void *memcpy(void * restrict s1, const void * restrict s2, size_t n);

void *memmove (void *s1, const void *s2, size_t n) ;

Обе функции копируют n байтов из области, на которую указывает аргумент s2, в область, указанную аргументом s1, и обе они возвращают значение s1. Различие между этими двумя функциями, как указывает ключевое слово restrict, связано с тем, что memcpy() разрешено полагать, что две области памяти нигде не перекрываются друг с другом. Функция memmove() не делает такого предположения, поэтому копирование происходит так, как будто все байты сначала помещаются во временный буфер и только затем копируются в область назначения. Что произойдет, если применить memcpy() к перекрывающимся областям? В этом случае поведение функции не определено, т.е. она может как работать, так и не работать.

Использование assert()

```c
#define SIZE 10
void show_array(const int ar[], int n);
_Static_assert(sizeof(double) == 2 * sizeof(int), "double не имеет дв
int main() {
    int values[SIZE] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}, target[SIZE];
    double curious[SIZE / 2] = {2.0, 2.0e5, 2.0e10, 2.0e20, 5.0e30};
    puts("Использование memcpy():");
    puts("значения (исходные данные): ");
    show_array(values, SIZE);
    memcpy(target, values, SIZE * sizeof(int));
    puts("целевые данные (копия значений):");
    show_array(target, SIZE);
    puts("\nИспользование memmove() с перекрывающимися областями:");
    memmove(values + 2, values, 5 * sizeof(int));
    puts("значения -- элементы 0-5 скопированы в элементы 2-7:");
    show_array(values, SIZE);
    puts("\nИспользование memcpy() для копирования double в int:");
    memcpy(target, curious, (SIZE / 2) * sizeof(double));
    puts("целевые данные -- 5 значений double в 10 позициях int: ");
    show_array(target, SIZE / 2);
    show_array(target + 5, SIZE / 2);
    return 0;
}
void show_array(const int ar[], int n) {
    for (int i = 0; i < n; i++)
        printf("%d ", ar[i]);
    putchar('\n');
}
```

```
Использование memcpy():
значения (исходные данные):
1 2 3 4 5 6 7 8 9 10
целевые данные (копия значений):
1 2 3 4 5 6 7 8 9 10

Использование memmove() с
перекрывающимися областями:
значения -- элементы 0-5
скопированы в элементы 2-7:
1 2 1 2 3 4 5 8 9 10

Использование memcpy() для
копирования double в int:
целевые данные -- 5 значений double
в 10 позициях int:
0 1073741824 0 1091070464
536870912
1108516959 2025163840 1143320349
-2012696540 1179618799
```

Переменное число аргументов: файл stdarg.h

Заголовочный файл stdarg.h предоставляет возможность функций с переменным числом аргументов. Однако использовать ее сложно. Вы должны выполнить следующие действия.

1. Подготовить прототип функции, в котором применяется троеточие.
2. Создать в определении функции переменную типа va_list.
3. Использовать макрос для инициализации этой переменной списком аргументов.
4. Применить макрос для доступа к списку аргументов.
5. Использовать макрос для очистки.

Пример:

```
void f1 (int n, ...); // допустимо
int f2(const char * s, int k, ...); // допустимо
char f3(char cl, ..., char c2); // недопустимо, троеточие не в конце
double f3 (...); // недопустимо, параметры отсутствуют
```

Прототипированную функцию f1() можно вызывать следующим образом:

```
f1 (2, 200, 400); // 2 дополнительных аргумента
f1 (4, 13, 117, 18, 23); // 4 дополнительных аргумента
```

Переменное число аргументов: файл stdarg.h

Тип **va_list**, объявленный в заголовочном файле stdargs.h, представляет объект данных, применяемый для хранения параметров, которые соответствуют разделу троеточия в списке параметров. **Пример:**

```
double sum(int lim,...)
{
    va_list ap; // объявление объекта для хранения аргументов
```

В этом примере **lim** является параметром parmN и **указывает количество аргументов** в списке переменных-аргументов.

Затем функция будет использовать макрос va_start(), также определенный в stdargs.h, для копирования списка аргументов в переменную va_list. Макрос принимает два аргумента: переменную va_list и параметр parmN.

```
va_start(ap, lim); // инициализация ap списком аргументов
```

Переменное число аргументов: файл stdarg.h

На следующем этапе производится ***доступ к содержимому списка аргументов***. Это предусматривает применение еще одного макроса, **va_arg()**, который принимает два аргумента: переменную типа va_list и имя типа. При первом вызове он возвращает первый элемент списка, при следующем вызове — следующий элемент списка и т.д.

Например, если первым аргументом в списке был double, а вторым — int, вы могли бы поступить так:

```
double tic;
int toc;
...
tic = va_arg(ap, double); // извлечение первого аргумента
toc = va_arg(ap, int); // извлечение второго аргумента
```

Но будьте внимательны. Тип аргумента на самом деле должен соответствовать спецификации. Автоматическое преобразование типов, предпринимаемое для операции присваивания***, здесь не происходит.***

Наконец, вы должны провести очистку с помощью макроса va_end(). Например, может понадобиться освободить память, динамически выделенную для хранения аргументов. Этот макрос принимает в качестве аргумента переменную va_list:

```
va_end(ap); // очистка
```

Переменное число аргументов: файл stdarg.h

Поскольку макрос va_arg() не обеспечивает копирование предыдущих аргументов для их возможного восстановления, может оказаться целесообразным сохранение копии переменной va_list. Для этой цели в стандарте C99 предусмотрен макрос по имени va_copy(). Он принимает два аргумента типа va_list и копирует второй аргумент в первый:

```
va_list ap;
va_list apcopy;
double double tic;
int toc;
...
va_start(ap, lim); // инициализация ap списком аргументов
va_copy(apcopy, ap); // делает apcopy копией ap
tic = va_arg(ap, double); // извлечение первого аргумента
toc = va_arg(ap, int); // извлечение второго аргумента
```

На данном этапе по-прежнему можно извлечь первые два элемента из apcopy несмотря на то, что они были удалены из ap.

stdarg.h пример

```c
#include <stdio.h>
#include <stdarg.h>
double sum(int, ...);
int main(void) {
    double s, t;
    s = sum(3, 1.1, 2.5, 13.3);
    t = sum(6, 1.1, 2.1, 13.1, 4.1, 5.1, 6.1);
    printf("возвращаемое значение sum(3, 1.1, 2.5, 13.3): %g\n", s);
    printf("возвращаемое значение sum(6, 1.1, 2.1, 13.1, 4.1, 5.1, 6.1): %g\n", t);
    return 0;
}
double sum(int lim, ...) {
    va_list ap; // объявление объекта для хранения аргументов
    double tot = 0;
    va_start(ap, lim); // инициализация ap списком аргументов
    for (int i = 0; i < lim; i++)
        tot += va_arg(ap, double); // доступ к каждому элементу в списке аргументов
    va_end(ap);                    // очистка
    return tot;
}
```

```
возвращаемое значение sum(3, 1.1, 2.5, 13.3): 16.9
возвращаемое значение sum(6, 1.1, 2.1, 13.1, 4.1, 5.1, 6.1): 31.6
```

Адженда

Библиотека C	Макросы и препроцессор	Static, extern	Прогрессия
20 минут	45 минут	10 минут	15 минут

Первые шаги в трансляции программы

До передачи управления препроцессору компилятор должен провести программу через ряд этапов трансляции. Компилятор начинает свою, устанавливая соответствие символов исходного кода с исходным набором символов. При этом обрабатываются много байтные символы и триграфы — расширения символов, которые обеспечивают интернациональное применение языка С.

Пример:

printf("Это было вели\
колепно!\n”);

преобразуются в одному логическую строку.

printf("Это было великолепно!\n");

Пример 2:

int/* это не похоже на пробел */fox;

превращается в

int fox;

Кроме того, в рамках реализации компилятора возможна замена каждой последовательность пробельных символов одиночным пробелом. Наконец, программа готова для этапа предварительной обработки, и препроцессор начинает поиск своих потенциальных директив, обозначаемых символом # в начале строки.

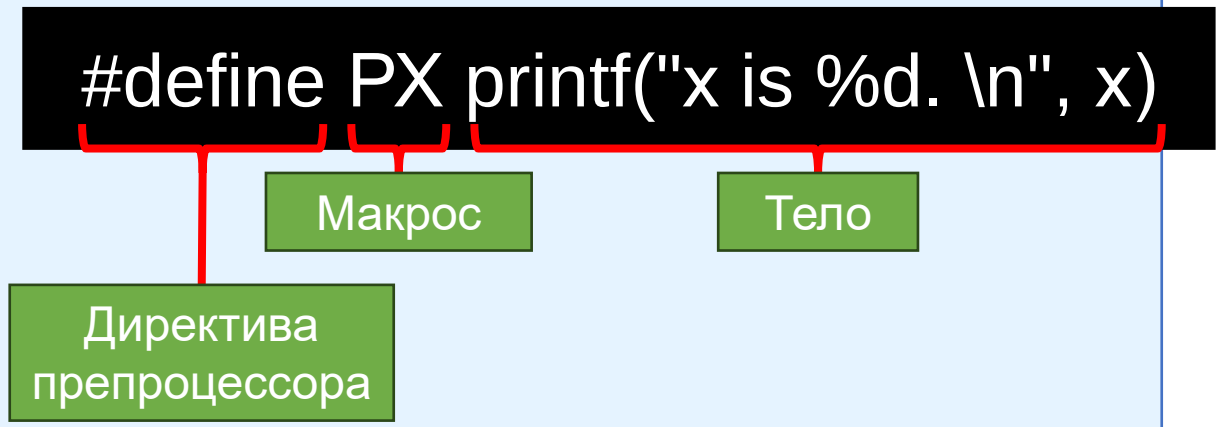
Символические константы: #define

Подобно всем директивам препроцессора, директива #define начинается с символа # в начале строки. Стандарт ANSI и последующие стандарты разрешают предварение символа # пробелами или табуляциями, а также наличие пробела между # и остальной частью директивы. Однако в прежних версиях C обычно требовалось, чтобы директива начиналась в крайней левой позиции в строке, а пробелы между символом # и остальной частью директивы не допускались. Директива может находиться в любом месте файла исходного кода, и ее определение распространяется от этого места до конца файла.

Директива препроцессора простирается до тех пор, пока не встретится первый символ новой строки после знака #. Другими словами, длина директивы ограничена одной строкой. Однако, как уже упоминалось ранее, комбинация обратной косой черты и символа новой строки удаляются до начала работы препроцессора, поэтому директиву можно распространить на несколько физических строк. Тем не менее, эти строки образуют одну логическую строку.

Символические константы: #define

```c
#include <stdio.h>
#define TWO 2                    /* при желании можно использовать комментарии */
#define OW "Логика - последнее убежище лишенных\
воображения. - Оскар Уайльд" /* обратная косая черта переносит определение */
                                 /* на следующую строку                        */
#define FOUR TWO *TWO
#define PX printf("X = %d.\n", x)
#define FMT "X = %d.\n"
int main(void)
{
    int x = TWO;
    PX;
    x = FOUR;
    printf(FMT, x);
    printf("%s\n", OW);
    printf("TWO: OW\n");
    return 0;
}
```



Символические константы: #define

Когда должны использоваться символические константы? Вы должны их применять для большинства числовых констант. Если число представляет собой некоторую константу, участвующую в вычислениях, то символическое имя сделает ее назначение более понятным. Если число является размером массива, то символическое имя упростит его будущее изменение и корректировку границ выполнения циклов. Если число представляет собой системный код, такой как EOF, то символическое имя увеличит степень переносимости программы; понадобится изменять только одно определение EOF. Мнемоническое значения, изменяемость и переносимость — все эти характеристики делают символические константы заслуживающими внимания.

Однако ключевое слово const, которое теперь поддерживается в C, обеспечивает более гибкий способ создания констант. С помощью const можно создавать глобальные и локальные константы, числовые константы, константы в форме массивов и константы в виде структур. С другой стороны, константы-макросы могут использоваться для указания размеров стандартных массивов, а также инициализирующих значений для величин const.

```
 #define LIMIT 20
const int LIM = 50;
static int data1[LIMIT]; // допустимо
static int data2[LIM]; // не обязательно должно быть допустимым
const int LIM2 = 2 * LIMIT; // допустимо
const int LIM3 = 2 * LIM; // не обязательно должно быть допустили
```

Лексемы

Формально тело макроса должно быть строкой лексем, а не строкой символов. Лексемы препроцессора С — это отдельные “слова” в теле определения макроса. Они отделяются друг от друга пробельными символами. Например, определение
#define FOUR 2*2
имеет одну лексему — последовательность 2 * 2, но определение
#define SIX 2 * 3
содержит три лексемы: 2, * и 3.
Строки символов и строки лексем отличаются в том, как трактуются последовательности из множества пробелов. Рассмотрим следующее определение:
#define EIGHT 4 * 8
Препроцессор, который интерпретирует тело макроса как строку символов, вместо EIGHT подставляет 4 * 8. То есть дополнительные пробелы будут частью замены, но препроцессор, который интерпретирует тело как строку лексем, заменит EIGHT тремя лексемами, разделенными одиночными пробелами: 4 * 8. Другими словами, интерпретация в виде строки символов трактует пробелы как часть тела, а интерпретация в виде строки лексем считает пробелы разделителями между лексемами внутри тела. На практике не которые компиляторы С рассматривают тела макросов как строки, а не как лексемы. Это различие имеет практическое значение только дня более сложных случаев использования по сравнению с приведенными здесь.

Переопределение констант

Предположим, что вы определили константу LIMIT как имеющую значение 20, и затем в том же файле определили ее снова, но уже со значением 25. Такой процесс называется переопределением константы. Политика переопределения зависит от реализации компилятора. Одни реализации считают переопределение ошибкой, если только новое определение не совпадает со старым. Другие разрешают переопределение, возможно, выдавая предупреждение. В стандарте ANSI принят первый вариант, разрешающий переопределение, только если новое определение дублирует предыдущее.

Совпадение определений означает, что их тела должны иметь одни и те же лексемы в том же самом порядке. Поэтому приведенные ниже определения эквивалентны:

```
#define SIX 2 * 3
#define SIX 2   *   3
```

Оба определения содержат те же самые три лексемы, а избыточные пробелы не являются частью тела. Следующее определение рассматривается как отличающееся:

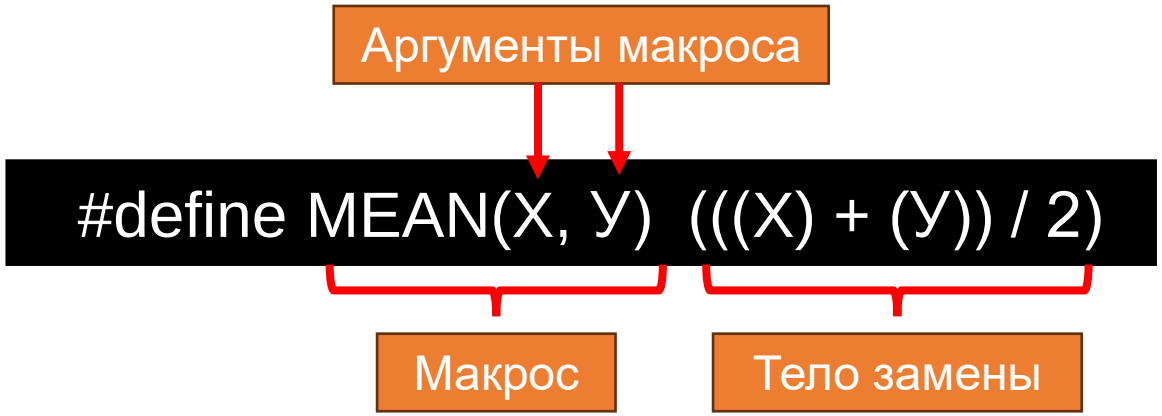
```
 #define SIX 2*3
```

Оно содержит только одну лексему, а не три, поэтому не совпадает с предыдущими определениями. Если вы хотите переопределить макрос, применяйте директиву #undef.

Если требуется переопределить некоторые константы, то для достижения цели может быть проще использовать ключевое слово const и правила области действия.

Использование аргументов в директиве #define

С помощью аргументов можно создавать функциональные макросы, которые выглядят и действуют во многом подобно обычным функциям. Макрос с аргументами очень похож на функцию, потому что аргументы заключаются в круглые скобки. Определения функциональных макросов имеют один или более аргументов в скобках, и эти аргументы затем присутствуют в выражении замены



Пример:
#define SQUARE(X) X*X
Оно может применяться в программе следующим образом:
z = SQUARE (2);

Пример #define

```c
#include <stdio.h>
#define SQUARE(X) X*X
#define PR(X) printf("Результат: %d.\n", X)
int main(void) {
    int x = 5;
    int z;
    printf("x = %d\n", x);
    z = SQUARE(x);
    printf("Вычисление SQUARE(x): ");
    PR(z);
    z = SQUARE(2);
    printf("Вычисление SQUARE(2): ");
    PR(z);
    printf("Вычисление SQUARE(x+2): ");
    PR(SQUARE(x + 2));
    printf("Вычисление 100/SQUARE(2): ");
    PR(100 / SQUARE(2));
    printf("x = %d.\n", x);
    printf("Вычисление SQUARE(++x): ");
    PR(SQUARE(++x));
    printf("После инкрементирования x = %x.\n", x);
    return 0;
}
```

```
// x = 5
// Вычисление SQUARE(x): Результат: 25.
// Вычисление SQUARE(2): Результат: 4.
// Вычисление SQUARE(x+2): Результат: 17.
// Вычисление 100/SQUARE(2): Результат: 100.
// x = 5.
// Вычисление SQUARE(++x): Результат: 49.
// После инкрементирования x = 7.
```

Создание строк из аргументов макроса: операция

Представим, что вы хотите поместить аргумент макроса в строку. Язык C позволяет сделать это. Внутри заменяющей части функционального макроса символ # становится операцией препроцессора, которая преобразует лексемы в строки.

```c
#include <stdio.h>
#define PSQR(x) printf("Квадрат " #x " равен %d.\n", ((x) * (x)))
int main(void)
{
    int y = 5;
    PSQR(y);
    PSQR(2 + 4);
    return 0;
}
```

```
Квадрат y равен 25.
Квадрат 2 + 4 равен 36.
```

Средство слияния препроцессора: операция

Подобно #, операция ## Может применяться в заменяющей части функционального макроса. Вдобавок она может использоваться в заменяющей части объектного макроса. Операция ## объединяет две лексемы в одну.

Пример:

#define XNAME(n) x ## n

Тогда макрос XNAME(4) будет расширен: x4

```c
#include <stdio.h>
#define XNAME(n) x##n
#define PRINT_XN(n) printf("x" #n " = %d\n", x##n);
int main(void)
{
    int XNAME(1) = 14; // превращается в int xl = 14;
    int XNAME(2) = 20; // превращается в int x2 = 20;
    int x3 = 30;
    PRINT_XN(1); // превращается в printf("xl = %d\n", xl) ;
    PRINT_XN(2); // превращается в printf("x2 = %d\n", x2);
    PRINT_XN(3); // превращается в print f ( "x3 = %d\n", x3) ;
    return 0;
}
```

```
// x1 = 14
// x2 = 20
// x3 = 30
```

Средство слияния препроцессора: операция

Подобно #, операция ## Может применяться в заменяющей части функционального макроса. Вдобавок она может использоваться в заменяющей части объектного макроса. Операция ## объединяет две лексемы в одну.

Пример:

#define XNAME(n) x ## n

Тогда макрос XNAME(4) будет расширен: x4

```
#include <stdio.h>
#define XNAME(n) x##n
#define PRINT_XN(n) printf("x" #n " = %d\n", x##n);
int main(void)
{
    int XNAME(1) = 14; // превращается в int x1 = 14;
    int XNAME(2) = 20; // превращается в int x2 = 20;
    int x3 = 30;
    PRINT_XN(1); // превращается в printf("x1 = %d\n", xl) ;
    PRINT_XN(2); // превращается в printf("x2 = %d\n", x2);
    PRINT_XN(3); // превращается в print f ( "x3 = %d\n", x3) ;
    return 0;
}
```

```
// x1 = 14
// x2 = 20
// x3 = 30
```

Макросы с переменным числом аргументов: . . . И __VA_ARGS__

Некоторые функции, скажем, printf(), принимают переменное количество аргументов. В C99/C11 тоже самое сделано и для макросов.

Идея заключается в том, что последний аргумент в списке аргументов для определения макроса может быть троеточием. Если это так, то в заменяющей части может применяться предопределенный макрос __VA_ARGS__, который будет подставлен вместо троеточия.

Не забывайте, что троеточие должно быть последним аргументом макроса; следующее определение является ошибочным:

#define WRONG (X, ..., У) #X #__VA_ARGS__ #y // не работает

```c
#include <stdio.h>
#include <math.h>
#define PR(X, ...) printf("Сообщение " #X ": " __VA_ARGS__)
int main(void) {
    double x = 48;
    double y;
    y = sqrt(x);
    PR(1, "x = %g\n", x);
    PR(2, "x = %.2f, y = %.4f\n", x, y);
    return 0;
}
```

```
// Сообщение 1: x = 48
// Сообщение 2: x = 48.00, y = 6.9282
```

Выбор между макросом и функцией

- Помните, что имя макроса не должно содержать пробелов, но пробелы допускаются в замещающей строке. В ANSI C разрешены пробелы в списке аргументов.
- Заключайте в скобки каждый аргумент и определение в целом. Это гарантирует корректное группирование элементов в выражении следующего рода:

```
forks = 2 * MAX (guests + 3, last);
```

- Используйте прописные буквы для имен функциональных макросов. Данное соглашение не так широко распространено, как применение прописных букв в именах константных макросов. Тем не менее, одна из веских причин использования прописных букв связана с тем, что это напоминает вам о возможных побочных эффектах макросов.
- Если вы намерены применять макрос вместо функции главным образом для ускорения работы программы, сначала попытайтесь выяснить, обеспечит ли это заметный выигрыш. Макрос, который используется в программе один раз, не приведет к значительному улучшению скорости ее выполнения. Макрос, находящийся внутри вложенного цикла, является намного лучшим кандидатом для ускорения работы программы.

Включение файлов: директива #inciude

Когда препроцессор встречает директиву #include, он ищет файл с указанным в директиве именем и включает его содержимое в текущий файл. Директива #include в файле исходного кода заменяется текстом включаемого файла. Это аналогично вводу содержимого включаемого файла в той же позиции внутри исходного файла. Существуют две разновидности #include:

#include <stdio.h> <- Имя файла указано в угловых скобках
#include "mystuff.h" <- Имя файла указано в двойных кавычках

В системе Unix угловые скобки сообщают препроцессору о необходимости поиска файла в одном или большем числе стандартных системных каталогов. Двойные кавычки говорят о том, что сначала следует просмотреть текущий каталог (или другой каталог, который указан вместе с именем файла), а затем искать в стандартных каталогах:

#include <stdio.h> <- Поиск в системных каталогах
#include "hot.h" <- Поиск в текущем рабочем каталоге
#include "/usr/biff/p.h" <- Поиск в каталоге /usr/biff

Суффикс .h традиционно применяется для заголовочных файлов — файлов с информацией, которая помещается в начале программы. Включение крупного заголовочного файла не обязательно приводит к значительному увеличению размера программы. Содержимое заголовочных файлов по большей части является информацией, которая используется компилятором для генерации окончательного кода, а не материалом, добавляемым к этому коду.

Случаи применения заголовочных файлов

- **Символические константы.** В типичном файле stdio.h, к примеру, определены константы EOF, NULL и BUFSIZ (размер стандартного буфера ввода-вывода).
- **Функциональные макросы.** Например, функция getchar() обычно определена как getc(stdin), а getc() — в форме довольно сложного макроса. Заголовочный файл ctype.h, как правило, содержит определения макросов для функций ctype.
- Объявления функций. Заголовочный файл string.h (strings.h в некоторых более старых системах), например, содержит объявления для семейства функций обработки строк. Согласно ANSI C и последующим стандартам, эти объявления представлены в виде прототипов функций.
- **Определения шаблонов структур.** Стандартные функции ввода-вывода используют структуру FILE, содержащую информацию о файле и связанном с ним буфере. Объявление этой структуры находится в файле stdio.h.
- **Определения типов.** Вы можете вспомнить, что стандартные функции ввода-вывода применяют аргумент типа указателя на FILE. Обычно в файле stdio.h используется #define или typedef для того, чтобы имя FILE представляло указатель на структуру. Аналогично, в заголовочных файлах определены типы size_t и time_t.

Директива #undef

Директива #undef отменяет заданное определение #define.
Пример:
#define LIMIT 400
Тогда директива
#undef LIMIT
удалит это определение. Затем LIMIT можно переопределить, назначив новое значение. Отмена определения LIMIT допустима даже в случае, если предварительное определение не делалось. Если вы хотите использовать некоторое имя, но не уверены в том, что оно не было определено ранее, на всякий случай его определение можно отменить.

Определение с точки зрения препроцессора

В отношении того, что считать идентификатором, препроцессор следует таким же правилам, как и язык C: идентификатор может состоять только из букв верхнего и нижнего регистра, цифр и символа подчеркивания, а первым символом не может быть цифра. Когда препроцессор встречает в какой-то директиве идентификатор, он считает его определенным или неопределенным. При этом определённый означает, что идентификатор определен препроцессором. Если идентификатор является именем макроса, созданного ранее директивой #define в том же файле, и он не отменялся посредством #undef, то идентификатор определен. Если идентификатор — не макрос, а, скажем, переменная с областью действия на уровне файла, то с точки зрения препроцессора он не определен.

Определенным может быть объектный макрос, включая пустой макрос, или функциональный макрос:

```
#define LIMIT 1000          // идентификатор LIMIT определен
#define GOOD                // идентификатор GOOD определен
#define A(X) ((-(X)) * (X)) // идентификатор A определен
int q;                      // идентификатор q - не макрос, поэтому не определен
#undef GOOD                 // идентификатор GOOD не определен
```

Несколько предопределенных макросов, таких как __DATE__ и __FILE__, всегда считаются определенными, причем их определение не может быть отменено.

Условная компиляция

Директивы #ifdef, #else и #endif

```
#ifdef MAVIS
#include "horse.h" // выполняется, если идентификатор MAVIS определен
#define STABLES 5
#else
#include "cow.h" // выполняется, если идентификатор MAVIS не определен
#define STABLES 15
#endif
```

Директива **#ifdef** говорит о том, что, если следующий за ней идентификатор (MAVIS) был определен препроцессором, необходимо обработать все директивы и скомпилировать весь код C до следующей директивы **#else** или **#endif** в зависимости от того, что встретится раньше. Если предусмотрена директива **#else**, то должен быть обработан весь код между **#else** и **#endif**, когда идентификатор не определен.

Форма **#ifdef #else** во многом подобна оператору if else языка C. Основное отличие в том, что препроцессор не распознает фигурные скобки ({}) как метод обозначения блока, поэтому для пометки блоков директив используются директивы **#else** (если есть) и **#endif** (должна присутствовать). Такие условные структуры могут быть вложенными.

Директивы #ifdef, #else и #endif

```c
#include <stdio.h>
#define JUST_CHECKING
#define LIMIT 4
int main(void)
{
    int i;
    int total = 0;
    for (i = 1; i <= LIMIT; i++)
    {
        total += 2 * i * i + 1;
#ifdef JUST_CHECKING
        printf("i=%d, промежуточная сумма = %d\n", i, total);
#endif
    }
    printf("Итоговая сумма = %d\n", total);
    return 0;
}
```

```
// i=1, промежуточная сумма = 3
// i=2, промежуточная сумма = 12
// i=3, промежуточная сумма = 31
// i=4, промежуточная сумма = 64
// Итоговая сумма = 64
```

Директива #ifndef

Директива #ifndef может использоваться совместно с директивами #else и #endif тем же самым способом, что и #ifdef. Директива #ifndef выясняет, не определен ли следующий за ней идентификатор; она представляет собой инверсию директивы #ifdef. Эта директива часто применяется для определения константы, если она еще не была определена.

Пример:

```
#ifndef SIZE
#define SIZE 100
#endif
```

Обычно такая конструкция используется для предотвращения множественных определений одного и того же макроса при включении нескольких заголовочных файлов, каждый из которых может содержать определение. В этом случае определение в первом заголовочном файле становится активным, а последующие определения в других заголовочных файлах игнорируются.

```
#define SIZE 10
#include "arrays.h"
```

Директива #ifndef – пример 2 файла

names.h

```
#ifndef NAMES_H_
#define NAMES_H_
// константы
#define SLEN 32
// объявления структур
struct names_st
{
    char first[SLEN];
    char last[SLEN];
};
// определения типов
typedef struct names_st names;
// прототипы функций
void get_names(names *);
void show_names(const names *);
char *s_gets(char *st, int n);
#endif
```

main.c

```
#include <stdio.h>
#include "names.h"
#include "names.h" // непреднамеренное второе включение
int main()
{
    names winner = {"Иван", "Иванов"};
    printf("Победителем стал %s %s.\n", winner.first,
           winner.last);
    return 0;
}
```

Директивы #if и #elif

Директива **#if** во многом похожа обычный оператор if языка C. За **#if** следует константное целочисленное выражение, которое считается истинным, когда оно имеет ненулевое значение. Для расширения комбинация **#if - #else** можно использовать директиву **#elif** (в некоторых старых реализациях она недоступна).

```
#if SYS == 1
    #include "ibmpc.h"
#elif SYS == 2
    #include "vax.h"
#elif SYS == 3
    #include "mac.h"
#else
    #include "general.h"
#endif
```

В более новых реализациях предлагается второй способ проверки, определено ли имя. Вместо строки **#ifdef VAX** можно применять следующую форму записи: **#if defined (VAX)**

```
#if defined (IBMPC)
    #include "ibmpc.h"
#elif defined (VAX)
#include "vax.h"
#elif defined (MAC)
    #include "mac.h"
#else
    #include "general.h"
#endif
```

Предопределенные макросы

Макрос	Описание
__DATE__	Строка символов в форме "Ммм дд гггг", представляющая дату обработки препроцессором, например, Aug 24 2014
__FILE__	Строка символов, представляющая имя текущего файла исходного кода
__LINE__	Целочисленная константа, представляющая номер строки в текущем файле исходного кода
__STDC__	Установлен в 1 для указания, что реализация соответствует стандарту C
__STDC_HOSTED__	Установлен в 1 для размещаемой среды; в противном случае — 0
__STDC_VERSION__	Для C99 установлен в 199901L; для C11 установлен в 201112L
__TIME__	Время трансляции в форме "чч:мм:сс"

***Следует отметить**, что стандарт C99 предоставляет предопределенный идентификатор __func__ , который расширяется до строкового представления имени содержащей его функции. По этой причине данный идентификатор должен иметь область действия в пределах функции, в то время как макросы, по существу, располагают областью действия на уровне файла.

Предопределенные макросы

```c
#include <stdio.h>
void why_me();
int main() {
    printf("Файл: %s.\n", __FILE__);
    printf("Дата: %s.\n", __DATE__);
    printf("Время: %s.\n", __TIME__);
    printf("Версия: is %ld.\n", __STDC_VERSION__);
    printf("Это строка %d.\n", __LINE__);
    printf("Это функция %s\n", __func__);
    why_me();
    return 0;
}
void why_me() {
    printf("Это функция %s\n", __func__);
    printf("Это строка %d.\n", __LINE__);
}
```

```
gcc main.c -o main -std=c99
```

```
Файл: main.c.
Дата: Jan 22 2025.
Время: 21:29:07.
Версия: is 199901.
Это строка 9.
Это функция main
Это функция why_me
Это строка 17.
```

Директивы #line и #error

Директива **#line** позволяет переустанавливать нумерацию строк и имя файла, выводимые с помощью макросов __LINE__ и __FILE__.

Пример:

```
#line 1000          //переустанавливает текущий номер строки в 1000
#line 10 “cool.c”   //переустанавливает номер строки в 10, а имя файла - в cool. c
```

Директива **#error** заставляет препроцессор выдать сообщение об ошибке, которое включает любой текст, указанный в директиве. Если это возможно, процесс компиляции должен приостановиться.

Пример:

```
#if __STDC_VERSION__ != 201112L
#error Несоответствие C11
#endif
```

После этого попытка компиляции программы могла бы привести к получению следующих результатов:

```
$ gcc main.c
main.c :14:2 : error: #error Несоответствие C11
$ gcc -std=c11 main.c
$
```

Процесс компиляции не проходит, когда компилятор использует более старый стандарт, и завершается успешно, когда применяется стандарт C11.

Директива #pragma

У современных компиляторов существует несколько настроек, которые можно модифицировать с помощью аргументов командной строки или через меню IDE-среды.
Директива #pragma позволяет помещать инструкции для компилятора в исходный код. Например, во время разработки стандарта C99 на него ссылались как на C9X, и в одном из компиляторов использовалась следующая директива для включения поддержки этого стандарта:
#pragrna c9x on
В общем случае каждый компилятор имеет собственный набор указаний. Они могут применяться, например, для управления объемом памяти, выделяемой под автоматические переменные, для установки уровня строгости при проверке ошибок или для включения нестандартных языковых средств. В стандарте C99 предоставляются три стандартных указания технической природы, которые здесь не рассматриваются.

Директива #pragma

Кроме того, стандарт C99 поддерживает операцию препроцессора _Pragma. Она преобразует строку в обычное указание компилятору.

Пример:

_Pragma("nonstandardtreatmenttypeB on") ⇔ #pragma nonstandardtreatmenttypeB on

Поскольку в этой операции не используется символ #, она может выступать в качестве части расширения макроса:

#define PRAGMA(X) _Pragma(#X)

#define LIMRG(X) PRAGMA(STDC CX_LIMITED_RANGE X)

После этого можно применять код вроде показанного ниже:

LIMRG (ON)

Кстати, следующее определение не работает, хотя выглядит вполне корректным:

#define LIMRG (X) _Pragma (STDC CX_LIMITED_RANGE #X)

Проблема в том, что оно полагается на конкатенацию строк, но компилятор не выполняет конкатенацию до тех пор, пока не завершится работа препроцессора.

Оператор _Pragma выполняет всю работу по превращению из строк, т.е. управляющие последовательности в строке преобразуются в представляющие их символы. Таким образом, вызов операции

_Pragma("use_bool \"true \"false")

принимает следующий вид:

#pragma use_bool "true "false

Обобщенный выбор(C11)

Термин ***обобщенное программирование*** относится к коду, который не является специфичным для конкретного типа, но после указания типа может транслироваться в код для этого типа. В C11 появился новый вид выражения, называемого выражением обобщенного выбора, которое можно применять для выбора значения на основе типа выражения, т.е. базируясь на том, является ли типом выражения int, double и т.д. **Пример:**

_Generic(x, int: 0, float: 1, double: 2, default: 3)

Здесь _Generic — новое ключевое слово C11. Круглые скобки после _Generic содержат несколько элементов, разделенных запятыми. Первый элемент представляет собой выражение, а каждый из оставшихся элементов — тип, за которым следует значение, наподобие float: 1. Тип первого элемента соответствует одной из меток, и значением всего выражения будет значение, указанное после давшей совпадение метки.

Пример 2:

```
#define MYTYPE(X) _Generic((X),\
int: "int",\
float : "float",\
double: "double",\
default: "other"\
)
```

Обобщенный выбор(C11)

```c
#include <stdio.h>
#define MYTYPE(X) _Generic((X), \
    int: "int",                 \
    float: "float",             \
    double: "double",           \
    default: "другой")
int main(void)
{
    int d = 5;
    printf("%s\n", MYTYPE(d));         // d имеет тип int
    printf("%s\n", MYTYPE(2.0 * d)); // 2.0 * d имеет тип double
    printf("%s\n", MYTYPE(3L));        // 3L имеет тип long
    printf("%s\n", MYTYPE(&d));        // &d имеет тип int*
    return 0;
}
```

```
// int
// double
// другой
// другой
```

Встраиваемые функции (C99)

Обычно с вызовом функции связаны накладные расходы. Это означает, что подготовка вызова, передача аргументов, переход к коду функции и возврат требуют времени на выполнение. В стандарте C99 был позаимствован у C++ (но не во всем точно) другой подход — встраиваемые функции. Преобразование функции во встроенную может привести к тому, что компилятор заменит вызов функции встраиваемым кодом и/или предпримет оптимизации другого рода либо вообще не окажет никакого воздействия.

Существуют разные способы создания определений встраиваемых функций. Простой подход предполагает использование спецификатора функции **inline** наряду со спецификатором класса хранения **static**. Как правило, встраиваемые функции определяются до их первого применения в файле, так что определение действует также и в качестве прототипа.

Пример

```
#include <stdio.h>
inline static void eatline() // встраиваемое определение/прототип
{
    while (getchar() != '\n')
        continue;
}
int main() {
    ...
    eatline(); // вызов функции
    ...
}
```

Функции _Noreturn(C11)

Когда в стандарте C99 появилось ключевое слово inline, оно было единственным примером спецификатора функции. В стандарт C11 был добавлен второй спецификатор функции, _Noreturn, предназначенный для указания функции, которая по завершении не возвращает управление вызывающей функции. Примером функции _Noreturn является exit(); после обращения к ней вызывающая функция никогда не возобновит свое выполнение. Обратите внимание, что это отличается от возвращаемого типа void. Типичная функция void возвращает управление вызывающей функции; она просто не предоставляет какое-либо значение.

Цель _Noreturn заключается в том, чтобы проинформировать пользователя и компилятор, что конкретная функция не возвратит управление вызывающей программе.

Информирование пользователя помогает предотвратить неправильное употребление функции, а указание на такой факт компилятору может сделать возможными некоторые оптимизации кода.

Адженда

Библиотека C	Макросы и препроцессор	Static, extern	Прогрессия
20 минут	45 минут	10 минут	15 минут

Static

Спецификатор **static** создает объект со статической продолжительностью хранения, который появляется после загрузки программы в память и исчезает при завершении программы. Если **static** применяется в объявлении с областью видимости в пределах файла, то область видимости ограничивается одним этим файлом. Если **static** используется в объявлении с областью видимости в пределах блока, то область видимости ограничивается этим блоком. Таким образом, объект существует и сохраняет свое значение на протяжении выполнения программы, но может быть доступен посредством идентификатора, только когда выполняется код внутри его блока.

Статическая переменная с областью видимости в пределах блока не имеет связывания. Статическая переменная с областью видимости в пределах файла имеет внутреннее связывание.

Extern

Спецификатор **extern** указывает, что вы объявляете переменную, которая была определена в каком-то другом месте. Если объявление, содержащее **extern**, имеет область видимости в пределах файла, то переменная, на которую производится ссылка, должна иметь внешнее связывание. Если объявление с **extern** имеет область видимости в пределах блока, то ссылаемая переменная может иметь либо внешнее, либо внутреннее связывание, что зависит от определяющего объявления этой переменной.

Пример 1/2

parta.c

```c
#include <stdio.h>
void report_count();
void accumulate(int k);
int count = 0; // область видимости в пределах файла, внешнее связывание
int main(void) {
    int value;      // автоматическая переменная
    register int i; // регистровая переменная
    printf("Введите положительное целое число (0 для завершения) : ");
    while (scanf("%d", &value) == 1 && value > 0) {
        ++count; // использование переменной с областью видимости в пределах файла
        for (i = value; i >= 0; i--)
            accumulate(i);
        printf("Введите положительное целое число (0 для завершения): ");
    }
    report_count();
    return 0;
}
void report_count() {
    printf("Цикл выполнен %d раз(а)\n", count);
}
```

Пример 2/2

partb.c

```c
#include <stdio.h>
extern int count;              // ссылочное объявление, внешнее связывание
static int total = 0;          // статическое определение, внутреннее связывание
void accumulate(int k);        // прототип
void accumulate(int k)         // k имеет область видимости в пределах блока,
{                              // связывание отсутствует
    static int subtotal = 0; // статическая переменная, связывание отсутствует
    if (k <= 0) {
        printf("итерация цикла: %d\n", count);
        printf("subtotal: %d; total: %d\n", subtotal, total);
        subtotal = 0;
    }
    else {
        subtotal += k;
        total += k;
    }
}
```

```
Введите положительное целое число (0 для завершения) : 5
итерация цикла: 1
subtotal: 15; total: 15
Введите положительное целое число (0 для завершения): 10
итерация цикла: 2
subtotal: 55; total: 70
Введите положительное целое число (0 для завершения): 2
итерация цикла: 3
subtotal: 3; total: 73
Введите положительное целое число (0 для завершения): 0
Цикл выполнен 3 раз(а)
```

Классы хранения и функции

Функции также имеют классы хранения. Функция может быть либо внешней (по умолчанию), либо статической. (В стандарте C99 добавлена третья возможность — встраиваемая функция) Доступ к внешней функции могут получать функции в других файлах, но статическая функция может применяться только внутри файла, где она определена. Рассмотрим, например, файл со следующими прототипами функций:

```
double gamma(double); /* по умолчанию внешняя */
static double beta(int, int);
extern double delta(double, int);
```

Функции gamma () и delta () могут использоваться функциями в других файлах, которые являются частью программы, но beta () — нет. Из-за такого ограничения функции beta () одним файлом в остальных файлах можно применять другие функции с этим же именем. Причина использования класса хранения static связана с созданием функций, закрытых в отношении конкретного модуля, благодаря чему устраняется возможность конфликта имен.

Обычная практика предусматривает применение ключевого слова extern при объявлении функции, определённой в другом файле. Главным образом это касается ясности, т.к. объявление функции предполагается как extern, если только не указано ключевое слово static.

Адженда

Библиотека C	Макросы и препроцессор	Static, extern	Прогрессия
20 минут	45 минут	10 минут	15 минут

Задача

Найти все подстроки, которые содержат разные буквы

Тест

Вход:

“aba”

Выход: 5

“a”

“ab”

“b”

“ba”

“a”

Решение

```c
#include <stdio.h>
int countStr(char *str) {
    int hash_symbol[sizeof(char) * 256] = {0};
    int left = 0, right = 0, sum = 0;
    for (; str[right] != '\0'; ++right) {
        hash_symbol[str[right]]++;
        while (hash_symbol[str[right]] == 2) {
            hash_symbol[str[left]]--;
            sum += right - left;
            left++;
        }
    }
    sum += (2 * (right - left) - (right - left - 1)) * (right - left) / 2;
    return sum;
}
int main(void) {
    const int sizeStr = 10;
    char str[sizeStr + 1];
    scanf("%s", str);
    printf("\ncount = %d \n", countStr(str));
    return 0;
}
```

```
aba

count = 5
```