

Программирование на языке C

С.Ю. Скоробогатов

Осень 2015

Список литературы по модулю

Базовые сведения

Введение

Выполнение программ в ОС Linux

Модель данных

Идентификаторы

Литералы

Объявления

Ввод/вывод

Операции

Операторы

Деклараторы

Строки

Структуры, объединения и перечисления

Препроцессор

1. Слайды лекций.
<http://db.tt/NTMYIZ9F> [pdf]
2. Б. Керниган, Д. Ритчи. Язык программирования C, 2-е издание. – М.: Издательский дом «Вильямс», 2009.
<http://db.tt/ewtTQxv7> [djvu]
3. Г. Уоррен. Алгоритмические трюки для программистов. – М.: Издательский дом «Вильямс», 2003.
<http://db.tt/dG52W3ir> [djvu]
<http://db.tt/1JWRQJDH> [pdf]
4. Э. Рэймонд. Как стать хакером.
www.linux.org.ru/books/HOWTO/Hacker-HOWTO.html

Базовые сведения

Введение

Выполнение программ в ОС Linux

Модель данных

Идентификаторы

Литералы

Объявления

Ввод/вывод

Операции

Операторы

Деклараторы

Строки

Структуры, объединения и перечисления

Препроцессор

В отличие от языка Pascal, язык C не рассчитан на поэтапное изучение начинающими программистами, поэтому даже самая примитивная программа на этом языке использует языковые конструкции, которые целесообразно рассматривать не ранее середины курса изучения языка.

Эту проблему можно обойти с помощью «шаблона» программы, который даётся без объяснений и содержит место, куда нужно вписывать свой код:

```
1  #include <stdio.h>
2
3  int main(int argc, char **argv)
4  {
5      /* Место, куда нужно вписать свой код. */
6      return 0;
7  }
```

Комментарии в программах, написанных на языке C, располагаются между «/*» и «*/».

Программы на языке C записываются в текстовых файлах, имеющих расширение «с».

Напишем программу hello.c, выводящую в стандартный поток вывода (по умолчанию – в окно терминала) сообщение «Hello, world!»:

```
1  #include <stdio.h>
2
3  int main(int argc, char **argv)
4  {
5      printf("Hello, World!\n");
6      return 0;
7  }
```

Содержимое файла hello.c называется *исходным текстом* программы и не предназначено для непосредственного выполнения. Для запуска нашей программы её нужно откомпилировать.

Компиляция и запуск программ

Базовые сведения

Введение

Выполнение программ в ОС Linux

Модель данных

Идентификаторы

Литералы

Объявления

Ввод/вывод

Операции

Операторы

Деклараторы

Строки

Структуры, объединения и перечисления

Преппроцессор

Компилятор языка C – это программа, выполняющая перевод исходного текста программы с языка C в машинный код. В нашем курсе мы будем использовать компилятор gcc. Файл, содержащий готовый к выполнению машинный код, мы будем называть *исполняемым файлом*. Тем самым, компилятор языка C по файлу, содержащему исходный код программы, строит эквивалентный ему исполняемый файл. Для компиляции программы hello.c нужно в окне терминала выполнить команду:

```
gcc -o hello hello.c
```

Опция «-o» задаёт имя исполняемого файла (в нашем случае он называется hello). Если эта опция не указана, исполняемый файл будет называться a.out.

Запуск программы hello осуществляется командой

```
./hello
```

Некоторые опции компилятора gcc

Базовые сведения

Введение

Выполнение программ в ОС Linux

Модель данных
Идентификаторы

Литералы

Объявления

Ввод/вывод

Операции

Операторы

Деклараторы

Строки

Структуры, объединения и перечисления

Препроцессор

Компилятор gcc поддерживает огромное количество опций. Нам потребуются некоторые из них:

-Wall -Wextra

разрешает компилятору выводить все возможные предупреждения о сомнительных местах в программе;

-g

говорит компилятору о том, что нужно добавить в исполняемый файл информацию о соответствии участков машинного кода и строчек исходного текста программы (нужно для работы отладчиков);

-O3

включает режим максимальной оптимизации кода по скорости;

-lm

подключает к исполняемому файлу библиотеку математических функций (извлечение квадратного корня, логарифмы, тригонометрия и т.п.).

Пример: опция «-lm» компилятора gcc

Базовые сведения

Введение

Выполнение программ в ОС Linux

Модель данных

Идентификаторы

Литералы

Объявления

Ввод/вывод

Операции

Операторы

Деклараторы

Строки

Структуры, объединения и перечисления

Препроцессор

Программа sq_root.c вычисляет квадратный корень:

```
1  #include <stdio.h>
2  #include <math.h>
3  int main(int argc, char **argv)
4  {
5      double x;
6      scanf("%lf", &x);
7      printf("%lf\n", sqrt(x));
8      return 0;
9  }
```

Попытка её компиляции без опции «-lm» даёт ошибку:

```
/tmp/ccDG6zji.o: In function 'main':
sq_root.c:(.text+0x33): undefined reference to 'sqrt'
collect2: error: ld returned 1 exit status
```

Правильный запуск компилятора gcc:

```
gcc -o sq_root -lm sq_root.c
```

Необходимые сведения о выполнении программ в ОС Linux

Базовые сведения

Введение

Выполнение программ в ОС Linux

Модель данных

Идентификаторы

Литералы

Объявления

Ввод/вывод

Операции

Операторы

Деклараторы

Строки

Структуры, объединения и перечисления

Преппроцессор

Невозможно грамотно программировать на языке C, не имея достаточного представления об аппаратной архитектуре и о некоторых особенностях операционной системы, под управлением которой предполагается запускать написанные на языке C программы.

Поэтому перед изучением языка мы кратко рассмотрим следующие вопросы:

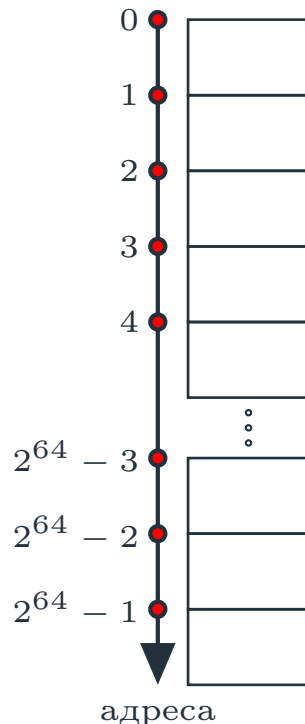
- модель памяти, в которой хранится программа и с которой она работает;
- переменные, их классификация по времени жизни;
- декомпозиция программы на функции и механизм вызова функций.

В нашем курсе мы будем ориентироваться на компьютеры, оснащённые 64-разрядными процессорами Intel и работающие под управлением операционной системы Linux.

Виртуальное адресное пространство

Базовые сведения

Введение
Выполнение программ в ОС Linux
Модель данных
Идентификаторы
Литералы
Объявления
Ввод/вывод
Операции
Операторы
Деклараторы
Строки
Структуры, объединения и перечисления
Преппроцессор



С точки зрения прикладного программиста данные и код программы размещаются в массиве, состоящем из 2^{64} байтов. Номер байта в этом массиве называется *адресом*, а совокупность всех 2^{64} адресов – *виртуальным адресным пространством* программы. В реальности лишь ничтожное количество адресов соответствуют ячейкам физической памяти. Попытка чтения или записи по адресу, за которым не закреплена физическая память, приводит к аварийному завершению программы («Segmentation fault»).

Следует отметить, что Linux на современных 64-разрядных процессорах Intel оставляет прикладной программе только адреса из диапазона $[0, 2^{48})$. Остальные адреса либо недоступны из-за ограничений реализации процессора, либо зарезервированы для операционной системы.

Базовые сведения

Введение

Выполнение программ в ОС Linux

Модель данных

Идентификаторы

Литералы

Объявления

Ввод/вывод

Операции

Операторы

Деклараторы

Строки

Структуры, объединения и перечисления

Препроцессор

Машинный код представляет собой последовательность элементарных команд процессора (*инструкций*), каждая из которых занимает один или несколько подряд идущих байтов. (В языке ассемблера инструкции имеют более удобную мнемоническую запись.)

При запуске программы загрузчик ОС Linux размещает машинный код программы в её виртуальном адресном пространстве, что обеспечивает возможность обращаться к инструкциям по адресам. При этом *адресом инструкции* считается адрес её первого байта.

Можно считать, что в каждый момент времени выполняется одна инструкция машинного кода. Её адрес записан в специальной ячейке памяти внутри процессора, называемой *регистром IP* (аббревиатура IP расшифровывается как «Instruction Pointer»).

Изменение содержимого регистра IP означает *передачу управления* к другой инструкции.

Смысл содержимого памяти

Базовые сведения

Введение

Выполнение программ в ОС Linux

Модель данных

Идентификаторы

Литералы

Объявления

Ввод/вывод

Операции

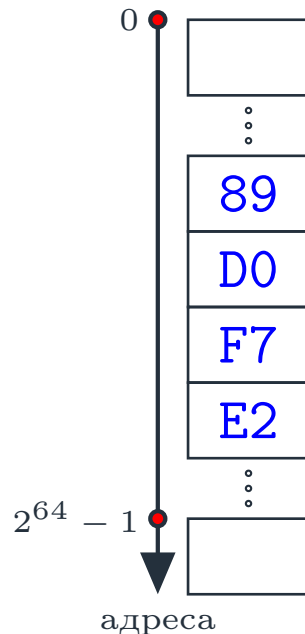
Операторы

Деклараторы

Строки

Структуры, объединения и перечисления

Преппроцессор



Смысл значений, хранящихся в памяти, определяется операциями, которые применяются для их обработки.

Например, 4 подряд идущих байта в памяти, содержащие значения 89, D0, F7, E2 (в 16-ричной системе), могут представлять:

- число с плавающей точкой $-2.29 \cdot 10^{21}$;
- целое число со знаком $-487\,075\,703$;
- целое число без знака $3\,807\,891\,593$;
- IP-адрес 137.208.247.226;
- машинные инструкции
`mov %edx,%eax; mul %edx.`

Если эти байты используются для хранения целочисленного значения, к ним будут применяться целочисленные операции. А если они содержат машинный код, то в какой-то момент на них может быть передано управление.

Базовые сведения

Введение

Выполнение программ в ОС Linux

Модель данных

Идентификаторы

Литералы

Объявления

Ввод/вывод

Операции

Операторы

Деклараторы

Строки

Структуры, объединения и перечисления

Препроцессор

Функция – это фрагмент программы, выполняющий определённые вычисления на основе передаваемого ему набора параметров и возвращающий результат вычислений.

Примеры:

- функция `sqrt` принимает в качестве параметра число x , вычисляет и возвращает в качестве результата \sqrt{x} ;
- функция `printf` принимает строку текста и набор значений, вставляет эти значения в нужные места строки, выводит получившийся текст в стандартный поток вывода и возвращает общее число выведенных символов;
- функция `main` – главная функция, из которой вызываются все остальные функции программы; ей передаются аргументы командной строки, а возвращает она код завершения программы.

Адрес функции – это адрес машинной инструкции, с которой должно начинаться её выполнение.

Базовые сведения

Введение

Выполнение программ в ОС Linux

Модель данных

Идентификаторы

Литералы

Объявления

Ввод/вывод

Операции

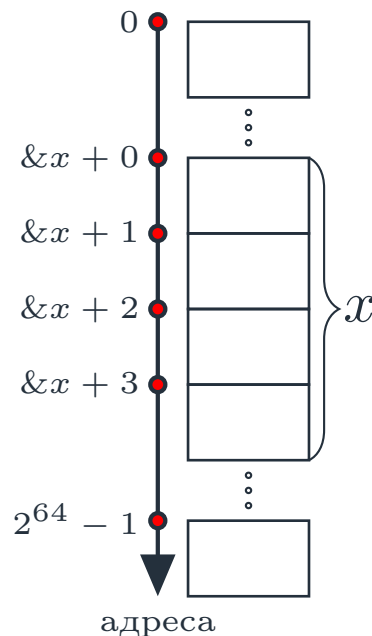
Операторы

Деклараторы

Строки

Структуры, объединения и перечисления

Препроцессор



Переменная – это совокупность байтов, занимающая непрерывный участок виртуального адресного пространства и используемая для хранения некоторого значения.

Размещение переменных в виртуальном адресном пространстве – прерогатива компилятора и менеджера памяти языка C, а также загрузчика операционной системы Linux. Адрес первого байта участка адресного пространства, занимаемого переменной, называется *адресом переменной*. Все переменные имеют разные адреса.

Обращение к переменной – это чтение или запись её значения. Чтобы обратиться к переменной, нужно знать её адрес. Переменные создаются и уничтожаются во время работы программы. *Время жизни переменной* – это интервал времени от создания до уничтожения переменной.

Классификация переменных по времени жизни

Базовые сведения

Введение

Выполнение программ в ОС Linux

Модель данных

Идентификаторы

Литералы

Объявления

Ввод/вывод

Операции

Операторы

Деклараторы

Строки

Структуры, объединения и перечисления

Препроцессор

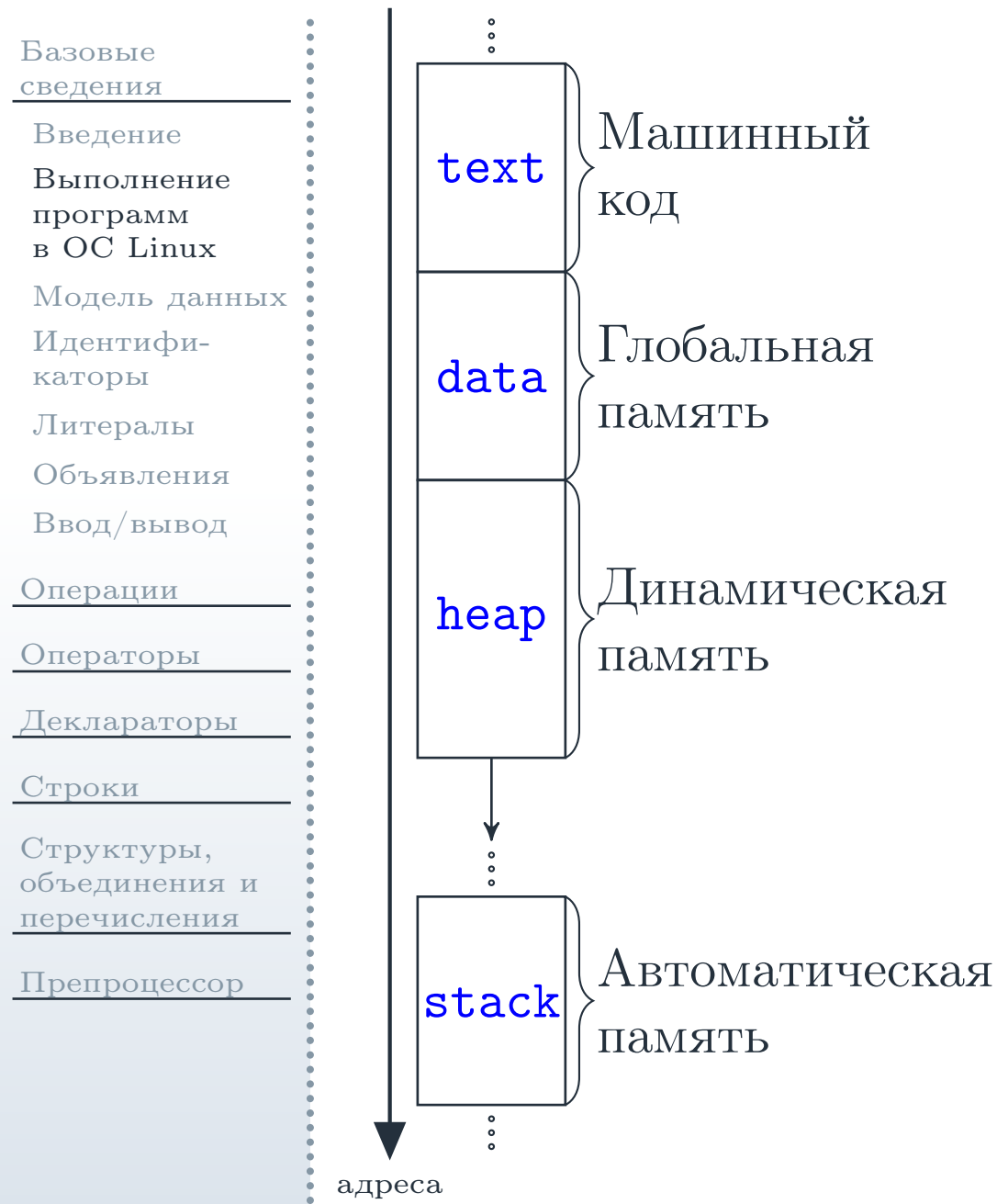
Глобальные переменные – это переменные, время жизни которых длится от момента запуска программы до момента завершения работы программы. К глобальным переменным потенциально можно обращаться из любой функции программы.

Локальные переменные – это переменные, которые автоматически создаются при вызове функции и уничтожаются при завершении работы функции. В локальных переменных хранятся промежуточные результаты вычислений. Как правило, обращения к локальным переменным происходят в функции, которой эти локальные переменные принадлежат.

Формальные параметры функции – это локальные переменные, в которые записываются передаваемые при вызове функции значения.

Динамические переменные – это переменные, создание и уничтожение которых осуществляется во время работы программы путём обращения к менеджеру памяти.

Карта виртуального адресного пространства



Виртуальное адресное пространство делится на 4 области:

- машинный код всех функций размещается в области «**text**»;
- область «**data**» используется для хранения глобальных переменных и строковых констант;
- динамические переменные создаются в области «**heap**», которая может расти во время выполнения программы;
- локальные переменные размещаются в области «**stack**», размер которой составляет по умолчанию 8 Мб.

Фреймы функций в автоматической памяти

Базовые сведения

Введение

Выполнение программ в ОС Linux

Модель данных

Идентификаторы

Литералы

Объявления

Ввод/вывод

Операции

Операторы

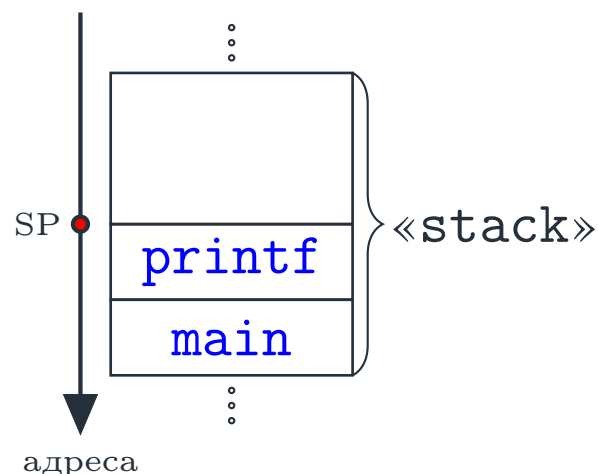
Деклараторы

Строки

Структуры, объединения и перечисления

Препроцессор

Фрейм функции – это участок автоматической памяти, в котором хранятся локальные переменные (включая формальные параметры) и адрес, на который должно быть передано управление после завершения функции (*адрес возврата*).



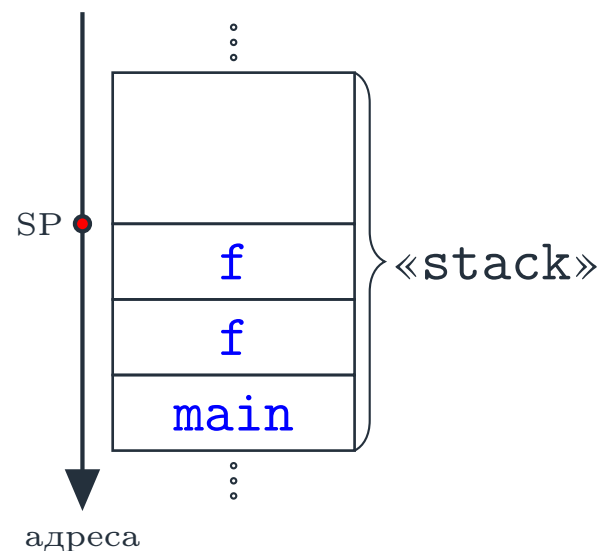
При запуске программы в старших адресах области «**stack**» создаётся фрейм функции **main** и управление передаётся по её адресу.

Если, например, из функции **main** вызывается функция **printf**, то перед фреймом **main** размещается

фрейм **printf**. При этом в качестве адреса возврата в этот фрейм помещается адрес инструкции функции **main**, которая следует за инструкцией вызова функции **printf**.

При завершении функции **printf** управление передаётся по записанному в её фрейме адресу возврата, а сам фрейм уничтожается.

Хранение локальных переменных функции во фрейме, который создаётся в момент вызова функции, обеспечивает возможность *рекурсии*, т.е. вызова функцией самой себя.



Например, пусть из функции `main` вызывается функция `f`, а из неё ещё раз вызывается функция `f`. В результате в области «`stack`» появятся два фрейма функции `f`, каждый со своим набором локальных переменных. Естественно, локальные переменные (в которые входят и формальные параметры) в разных фреймах функции `f` никак не пересекаются и могут иметь разные значения. Бывают более сложные виды рекурсии, например, когда функция `f` вызывает `g`, а та, в свою очередь, вызывает `f`. Отметим, что использование рекурсии может привести к исчерпанию свободного места в области «`stack`».

Базовые сведения

Введение

Выполнение программ в ОС Linux

Модель данных

Идентификаторы

Литералы

Объявления

Ввод/вывод

Операции

Операторы

Деклараторы

Строки

Структуры, объединения и перечисления

Препроцессор

Модель данных определяет способы отображения информации об объектах предметной области, с которой работает программа, в её виртуальное адресное пространство. Модель данных языка C во многом обусловлена аппаратной архитектурой.

Язык C – статически типизированный. Это означает, что каждой переменной в программе соответствует определённый в её исходном тексте и, тем самым, известный во время компиляции тип данных.

Тип переменной определяет размер участка памяти, занимаемого переменной, и набор допустимых операций.

Зная тип переменной, компилятор языка C порождает нужные команды для выполнения операций (например, разные команды для сложения целых чисел и чисел с плавающей точкой) или сообщает о недопустимых операциях (например, операцию вычисления остатка от деления запрещено применять к числам с плавающей точкой).

Базовые типы данных

Базовые сведения

Введение

Выполнение программ в ОС Linux

Модель данных

Идентификаторы

Литералы

Объявления

Ввод/вывод

Операции

Операторы

Деклараторы

Строки

Структуры, объединения и перечисления

Препроцессор

Базовые типы данных языка C представляют целые числа и числа с плавающей точкой, операции с которыми реализованы на аппаратном уровне.

Имя типа	Знак	Размер (битов)	Мин. знач.	Макс. знач.
char	есть	8	-128	127
unsigned char	нет	8	0	255
short	есть	16	-32768	32767
unsigned short	нет	16	0	65535
int	есть	32	-2^{31}	$2^{31} - 1$
unsigned int	нет	32	0	$2^{32} - 1$
long	есть	64	-2^{63}	$2^{63} - 1$
unsigned long	нет	64	0	$2^{64} - 1$
float	есть	32	$\pm 3.4 \cdot 10^{\pm 38}$	
double	есть	64	$\pm 1.79 \cdot 10^{\pm 308}$	
long double	есть	128	$\pm 1.19 \cdot 10^{\pm 4932}$	

Представление целых чисел

Базовые сведения

Введение

Выполнение программ в ОС Linux

Модель данных

Идентификаторы

Литералы

Объявления

Ввод/вывод

Операции

Операторы

Деклараторы

Строки

Структуры, объединения и перечисления

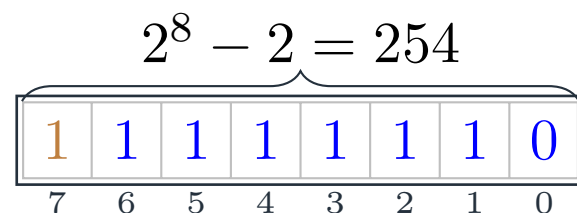
Препроцессор

Целое значение в памяти представляется своим *образом* – последовательностью из n битов, где n – размер его типа. Образ беззнакового значения содержит просто запись значения в двоичной системе счисления. При этом в n битов помещается число от 0 до $2^n - 1$.

Знаковое целочисленное значение x , находящееся в диапазоне от -2^{n-1} до $2^{n-1} - 1$, представляется в образе в *дополнительном коде*:

- если $x \geq 0$, то образ содержит двоичную запись x ;
- если $x < 0$, то в образе записывается число $2^n + x$.

Например, 11111110_2 – образ числа -2 типа `char`.



Нетрудно догадаться, что старший бит образа указывает на знак значения: 0 – неотрицательное, 1 – отрицательное. Этот бит называется *знаковым*.

Сложение и умножение целых чисел в дополнительном коде

Базовые сведения

Введение

Выполнение программ в ОС Linux

Модель данных

Идентификаторы

Литералы

Объявления

Ввод/вывод

Операции

Операторы

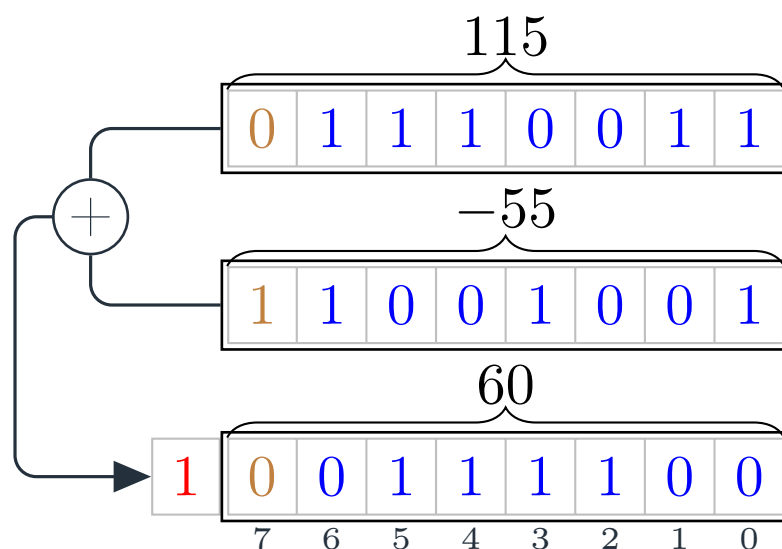
Деклараторы

Строки

Структуры, объединения и перечисления

Препроцессор

Сложение (а также умножение) целых чисел можно выполнять путём сложения (умножения) их образов вне зависимости от того, знаковые это числа или беззнаковые. Это достигается за счёт отбрасывания битов суммы (произведения), не поместившихся в заданный размер образа.



Например, выполним сложение 8-битовых образов чисел 115 и -55 . Образ числа -55 имеет значение 201. Число $316 = 115 + 201 - 9$ -битовое. Отбросив непоместившийся в 8-битовую разрядную сетку старший бит, получим число $60 = 115 - 55$.

Отметим, что приведённый рисунок также иллюстрирует переполнение суммы беззнаковых чисел 115 и 201.

Сужение образов целых чисел

Базовые сведения

Введение

Выполнение программ в ОС Linux

Модель данных

Идентификаторы

Литералы

Объявления

Ввод/вывод

Операции

Операторы

Деклараторы

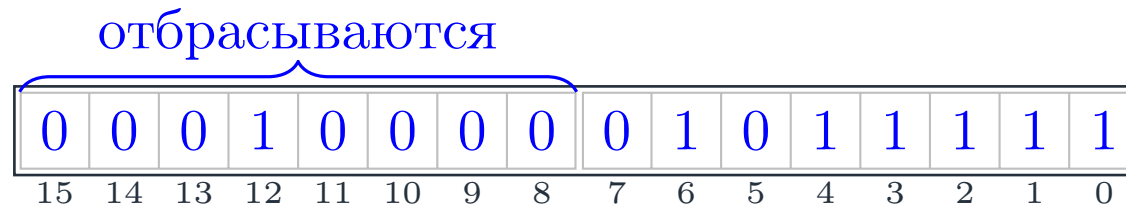
Строки

Структуры, объединения и перечисления

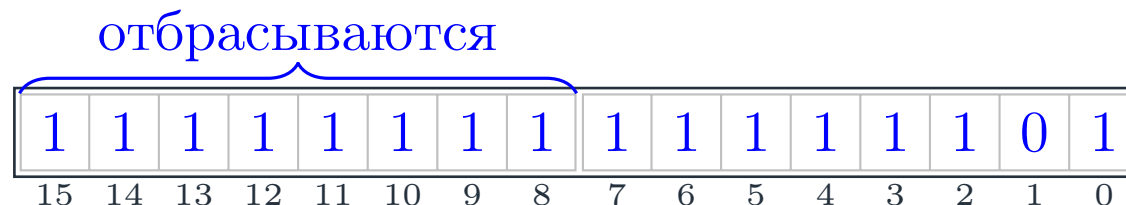
Препроцессор

Сужение образа целочисленного значения – это уменьшение его размера. При сужении старшие биты образа, не помещающиеся в заданный размер, отбрасываются.

Если число не помещается в нужное число битов, то в результате сужения оно может поменяться. Например, сужение значения 4191 типа `short` до `char` даёт число 95:



Сужение сохраняет значение числа (как знакового, так и беззнакового), если оно помещается в нужное количество битов. Например, сужение значения `-3` типа `short` до `char` даёт `-3`:



Расширение образов целых чисел

Базовые сведения

Введение

Выполнение программ в ОС Linux

Модель данных

Идентификаторы

Литералы

Объявления

Ввод/вывод

Операции

Операторы

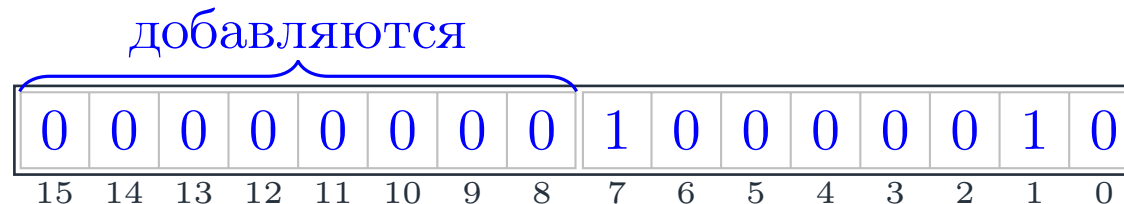
Деклараторы

Строки

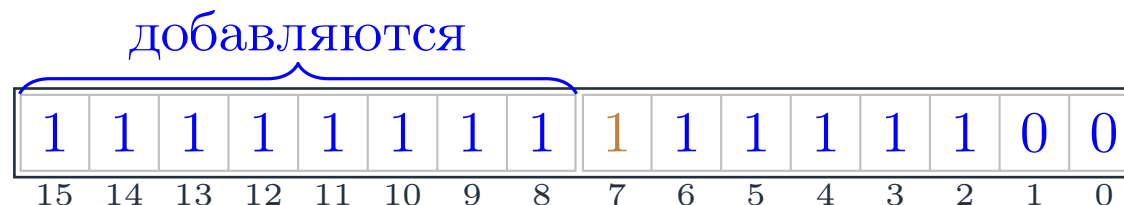
Структуры, объединения и перечисления

Препроцессор

Расширение образа – это, наоборот, увеличение его размера. Оно никогда не меняет значение, представляемое образом. Расширение образа беззнакового значения означает добавление к нему слева нужного количества нулевых битов. Например, расширение значения 130 типа `unsigned char` до `unsigned short`:



Расширение образа знакового значения выполняется путём «размножения» знакового бита: слева к значению добавляется нужное количество битов, равных знаковому биту. Например, расширение значения -4 типа `char` до `short`:



Порядок байтов образа целого числа в памяти

Базовые сведения

Введение

Выполнение программ в ОС Linux

Модель данных

Идентификаторы

Литералы

Объявления

Ввод/вывод

Операции

Операторы

Деклараторы

Строки

Структуры, объединения и перечисления

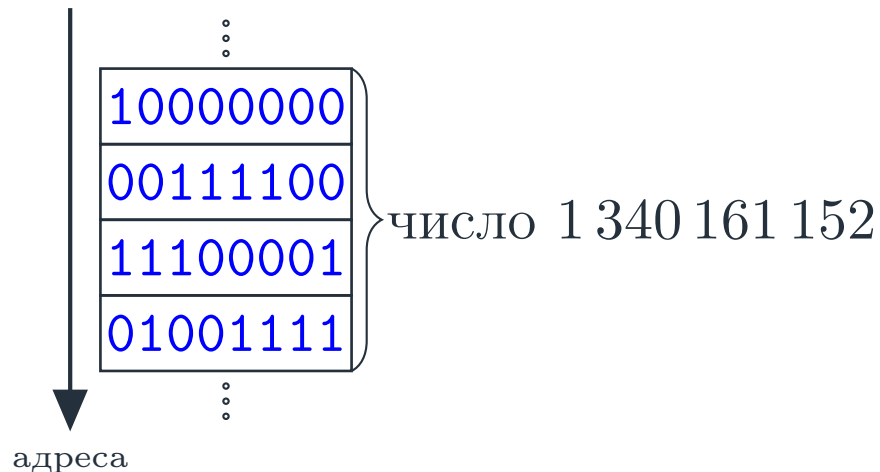
Преппроцессор

Образ целого числа в памяти занимает 1, 2, 4 или 8 байтов. Если каждый байт считать «цифрой» от 0 до 255, то можно сказать, что образ представлен в системе счисления по основанию 256.

Рассмотрим 32-разрядный образ числа 1 340 161 152:

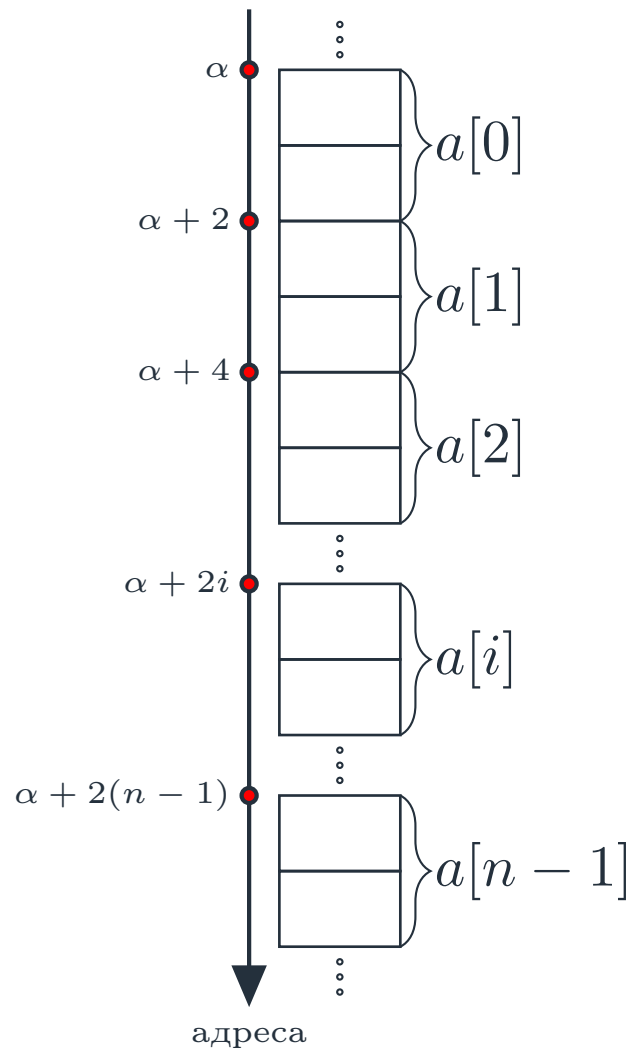
$$01001111_2 \cdot 256^3 + 11100001_2 \cdot 256^2 + 00111100_2 \cdot 256 + \underbrace{10000000_2}_{\text{младший байт}}.$$

Архитектура процессоров Intel подразумевает, что байты образа расположены в обратном порядке, т.е. младший байт идёт первым. Такой порядок байтов называется *little-endian*.



Базовые сведения

Введение
Выполнение программ в ОС Linux
Модель данных
Идентификаторы
Литералы
Объявления
Ввод/вывод
Операции
Операторы
Деклараторы
Строки
Структуры, объединения и перечисления
Препроцессор



Массив – это переменная, представляющая конечную последовательность значений одного типа. Каждое такое значение называется *элементом* массива, а общее количество элементов – *размером* массива.

Элементы массива располагаются в памяти друг за другом без промежутков и нумеруются, начиная с нуля. Порядковый номер элемента называется его *индексом*.

Если n – размер массива, то его элементы имеют индексы от 0 до $n - 1$. Пусть α – адрес массива, а w – размер его элемента, тогда адрес i -того элемента очевидно вычисляется как $\alpha + w \cdot i$.

Многомерные массивы

Базовые сведения

Введение

Выполнение программ в ОС Linux

Модель данных

Идентификаторы

Литералы

Объявления

Ввод/вывод

Операции

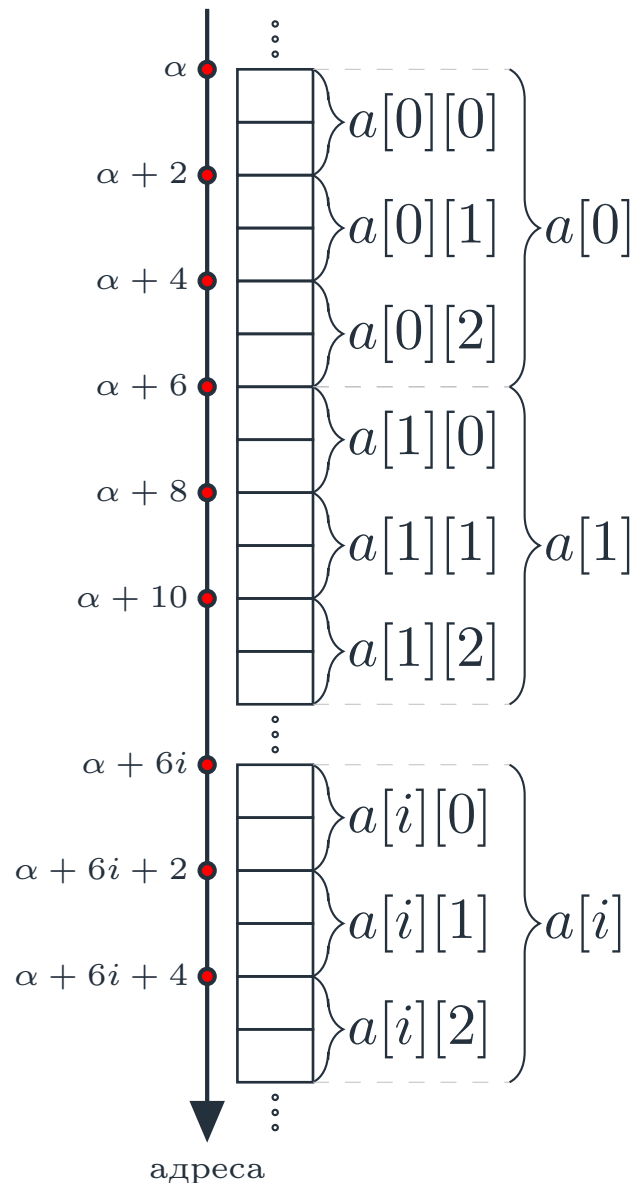
Операторы

Деклараторы

Строки

Структуры, объединения и перечисления

Препроцессор



Многомерный массив – это массив, элементами которого являются массивы.

Размерность массива – это число, задающее глубину вложенности многомерного массива. Размерность простого массива равна 1, размерность многомерного массива на 1 больше размерности его элемента.

Массив размерности 2 называется *двумерным массивом*.

Если размер двумерного массива равен m , а каждый его элемент имеет размер n , то о таком массиве говорят как о *матрице*, имеющей m строк и n столбцов.

Базовые сведения

Введение

Выполнение программ в ОС Linux

Модель данных

Идентификаторы

Литералы

Объявления

Ввод/вывод

Операции

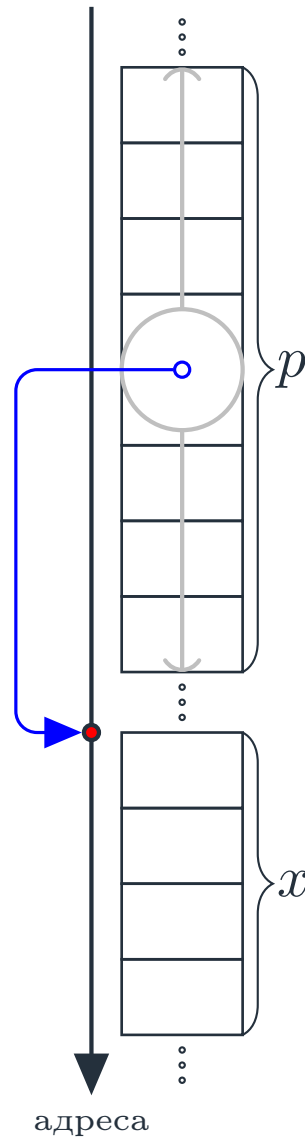
Операторы

Деклараторы

Строки

Структуры, объединения и перечисления

Препроцессор



Указатель – это переменная, в которой хранится адрес. Она занимает в памяти 8 байт, т.к. виртуальное адресное пространство содержит 2^{64} адресов. Как и образ любого целочисленного значения, образ указателя имеет порядок байт little-endian.

Говорят, что указатель *указывает* на некоторую переменную, если он содержит её адрес. *Разыменование* указателя – это чтение или запись по адресу, который хранится в указателе. Если в указателе содержится адрес 0, то считается, что указатель ни на что в памяти не указывает, т.к. гарантируется, что ни одна переменная по нулевому адресу не расположена. Разыменование такого указателя даёт ошибку «Segmentation fault».

Базовые сведения

Введение

Выполнение программ в ОС Linux

Модель данных

Идентификаторы

Литералы

Объявления

Ввод/вывод

Операции

Операторы

Деклараторы

Строки

Структуры, объединения и перечисления

Препроцессор

Имя переменной – это символическое обозначение адреса переменной в исходном тексте программы.

Не каждая переменная имеет имя: например, обращение к динамической переменной, созданной во время выполнения программы, осуществляется через указатель, содержащий адрес этой динамической переменной.

В машинном коде имена переменных отсутствуют: вместо них компилятор языка C вставляет адреса переменных.

Имя переменной записывается в виде *идентификатора* – последовательности латинских букв, цифр и знаков подчёркивания, не начинающейся с цифры.

Следует иметь в виду, что в идентификаторах учитывается регистр букв.

Примеры идентификаторов:

```
i k2 ALPHA item_num _count_
```

Целочисленные литералы

Базовые сведения

Введение

Выполнение программ в ОС Linux

Модель данных

Идентификаторы

Литералы

Объявления

Ввод/вывод

Операции

Операторы

Деклараторы

Строки

Структуры, объединения и перечисления

Преппроцессор

Литерал – это изображение некоторого значения (целого числа, числа с плавающей точкой, строки и т.п.) в исходном тексте программы.

Язык C поддерживает целочисленные литералы, записанные в десятичной, восьмеричной и шестнадцатеричной системах счисления:

```
65      /* Десятичная, т.к. первая цифра - не 0 */
0101    /* Восьмеричная, т.к. первая цифра - 0 */
0x41    /* Шестнадцатеричная: начинается с 0x */
0       /* Ноль в любой системе - ноль :- ) */
```

Кроме того, в качестве целочисленных литералов могут выступать записанные в апострофах символы таблицы ASCII:

```
'A'     /* число 65 - код буквы A в таблице ASCII */
```

Для справки: *таблица ASCII* задаёт отображение символов (латинские буквы, цифры, знаки препинания и т.п.) в целые числа от 0 до 127.

Escape-последовательности

Базовые сведения

Введение

Выполнение программ в ОС Linux

Модель данных

Идентификаторы

Литералы

Объявления

Ввод/вывод

Операции

Операторы

Деклараторы

Строки

Структуры, объединения и перечисления

Препроцессор

Escape-последовательность – это последовательность символов, начинающаяся с обратной косой черты и обозначающая один символ таблицы ASCII, который по той или иной причине не может быть другим способом введён с клавиатуры.

Как правило, Escape-последовательности используются для изображения управляющих символов:

```
'\n'  /* Перевод строки (код 10) */
'\t'  /* Горизонтальная табуляция (код 9) */
'\b'  /* Возврат на одну позицию влево (код 8) */
'\r'  /* Возврат в начало строки (код 13) */
```

Кроме того, часть Escape-последовательностей позволяет «бороться» с ограничениями синтаксиса языка C:

```
'\\'  /* Обратная косая черта */
'\''  /* Апостроф */
'\\"'  /* Кавычка (нужна в строковых литералах) */
```

Литералы с плавающей точкой

Базовые сведения

Введение

Выполнение программ в ОС Linux

Модель данных

Идентификаторы

Литералы

Объявления

Ввод/вывод

Операции

Операторы

Деклараторы

Строки

Структуры, объединения и перечисления

Препроцессор

Литералы с плавающей точкой используются для записи вещественных чисел:

```
3.14159265359 /* Обычная запись вещ. числа */  
45.32e-20      /* "Научная" запись вещ. числа */
```

Отметим, что внутри литерала с плавающей точкой не должно быть пробелов:

```
45.32 e -20    /* Ошибка! */
```

По умолчанию числа, представленные литералом с плавающей точкой, имеют тип `double`. Если добавить к литералу суффикс «`f`», то получится литерал типа `float` (точность – 7 цифр):

```
3.14159265359f /* В реальности будет 3.141593 */
```

Чтобы записать литерал типа `long double`, надо добавить к литералу суффикс «`L`».

Базовые сведения

Введение

Выполнение программ
в ОС Linux

Модель данных

Идентификаторы

Литералы

Объявления

Ввод/вывод

Операции

Операторы

Деклараторы

Строки

Структуры,
объединения и
перечисления

Препроцессор

Строковый литерал представляет собой последовательность символов и Escape-последовательностей, заключённую в кавычки:

```
"alphabet "
```

```
"Hello , \World!\n"
```

```
"Richard \\"rms\" \Stallman "
```

Внутри кавычек можно записывать не только символы ASCII, но и, например, русские буквы.

```
"Мама \мыла \раму "
```

Строковый литерал можно разбить на несколько записанных подряд строковых литералов. Этот приём бывает полезен, если, например, строковый литерал не помещается на одной строке.

```
"alpha" "bet" /* То же самое, что и "alphabet" */
```


Объявление переменных базовых типов

Базовые сведения

Введение

Выполнение программ в ОС Linux

Модель данных

Идентификаторы

Литералы

Объявления

Ввод/вывод

Операции

Операторы

Деклараторы

Строки

Структуры, объединения и перечисления

Препроцессор

Объявление переменной сообщает компилятору имя и тип переменной. Оно играет двойную роль:

- во время компиляции программы оно определяет смысл операций, применяемых к переменной;
- во время выполнения программы объявление обеспечивает резервирование участка виртуального адресного пространства для хранения значения переменной.

Объявление переменных базовых типов в сочетании с присвоением им начальных значений записывается как

```
тип имя1 = значение1, ..., имяN = значениеN;
```

Например,

```
long i = 0, j = 1, alpha = 20 + 4*5;
```

Синтаксис объявления переменных сложных типов (массивов, указателей и т.д.) мы рассмотрим позже.

Место объявлений переменных в «шаблоне» программы

Базовые сведения

Введение

Выполнение программ в ОС Linux

Модель данных

Идентификаторы

Литералы

Объявления

Ввод/вывод

Операции

Операторы

Деклараторы

Строки

Структуры, объединения и перечисления

Препроцессор

Объявление переменной должно быть записано до первого оператора, в котором эта переменная используется.

Локальные переменные объявляются внутри фигурных скобок, ограничивающих тело функции. Одним из способов объявить глобальную переменную является вынесение её объявления на верхний уровень программы (другой способ – объявление с ключевым словом `static`).

```
1  #include <stdio.h>
2
3  int count = 5;
4
5  int main(int argc, char **argv)
6  {
7      double k = 2.5 * count, x = 12.5e10;
8      return 0;
9  }
```

В приведённом примере переменная `count` – глобальная, а `k` и `x` – локальные переменные функции `main`.

Неинициализированные переменные

Базовые сведения

Введение

Выполнение программ в ОС Linux

Модель данных

Идентификаторы

Литералы

Объявления

Ввод/вывод

Операции

Операторы

Деклараторы

Строки

Структуры, объединения и перечисления

Препроцессор

При объявлении переменной можно не указывать её начального значения. Например, объявление

```
char a = 'A', b;
```

означает, что переменная **b** осталась без начального значения (т.е. **b** – *неинициализированная переменная*).

Неинициализированные глобальные переменные по умолчанию получают нулевое значение. Что касается неинициализированных локальных переменных, то при создании такой переменной в память ничего не пишется, и значением переменной становится «мусор», который случайно оказался в выделенном для неё участке памяти.

Распространённая ошибка при программировании на C – это неявное предположение, что неинициализированная локальная переменная имеет нулевое значение. Она особенно коварна тем, что проявляется не при каждом запуске программы.

Стандартные потоки ввода/вывода

Базовые сведения

Введение

Выполнение программ в ОС Linux

Модель данных

Идентификаторы

Литералы

Объявления

Ввод/вывод

Операции

Операторы

Деклараторы

Строки

Структуры, объединения и перечисления

Препроцессор

При запуске любой программы с ней ассоциируются три так называемых *потока*: стандартный поток ввода `stdin`, стандартный поток вывода `stdout` и стандартный поток вывода ошибок `stderr`.

Каждый поток соединяет программу либо с терминалом (по умолчанию), либо с файлом, либо с другой программой.

Перенаправление `stdout` программы `hello` в файл `world.txt`:

```
./hello >world.txt
```

Связывание `stdin` программы `sq_root` с файлом `input.txt`, а `stdout` – с файлом `output.txt`:

```
./sq_root <input.txt >output.txt
```

Перенаправление `stderr` компилятора `gcc` в файл `errors.txt`:

```
gcc -o hello hello.c 2>errors.txt
```

В стандартной библиотеке языка C существует функция `printf`, позволяющая выводить данные в `stdout`:

```
printf(форматная строка, значение1, значение2, ...);
```

Форматная строка задаёт текст для вывода, в котором предусмотрены места для вставки выводимых значений:

```
1  #include <stdio.h>
2
3  int main(int argc, char **argv)
4  {
5      printf("Values: %d, %f, %c, %s\n",
6             5+5, 3.14*2, 'A', "hello");
7      return 0;
8  }
```

Вывод:

```
Values: 10, 6.280000, A, hello
```

Форматные спецификаторы

Необязательные поля

%	флаги	ширина	. точность	размер	тип
---	-------	--------	------------	--------	-----

Место выводимого значения в форматной строке задаётся *форматным спецификатором*, начинающимся со знака «%». Количество форматных спецификаторов должно совпадать с количеством значений, передаваемых в функцию `printf`. Для вывода текста без вставки значений используется форматная строка, не содержащая спецификаторов. Если текст содержит знак «%», его необходимо удвоить, чтобы не спутать с форматным спецификатором:

```
printf("100%% ready!\n");
```

Вывод:

```
100% ready!
```

Базовые сведения

Введение

Выполнение программ в ОС Linux

Модель данных

Идентификаторы

Литералы

Объявления

Ввод/вывод

Операции

Операторы

Деклараторы

Строки

Структуры, объединения и перечисления

Препроцессор

Тип выводимого значения

Необязательные поля

%	флаги	ширина	. точность	размер	тип
---	-------	--------	------------	--------	-----

Функция `printf` получает выводимые значения в виде последовательностей байт, лишённых информации о типах.

Типы значений нужно указывать в спецификаторах.

Поле «тип» одновременно задаёт тип вставляемого значения и его желаемое представление при выводе:

d, i	десятичное знаковое число;
u	десятичное беззнаковое число;
x, X	шестнадцатеричное беззнаковое число;
f	число с плавающей точкой в обычной записи;
e	число с плавающей точкой в «научной» записи;
c	символ ASCII;
s	строка;
p	указатель.

Базовые сведения

Введение

Выполнение программ в ОС Linux

Модель данных

Идентификаторы

Литералы

Объявления

Ввод/вывод

Операции

Операторы

Деклараторы

Строки

Структуры, объединения и перечисления

Препроцессор

Пример: представление целого числа при выводе

Базовые сведения

Введение

Выполнение программ в ОС Linux

Модель данных

Идентификаторы

Литералы

Объявления

Ввод/вывод

Операции

Операторы

Деклараторы

Строки

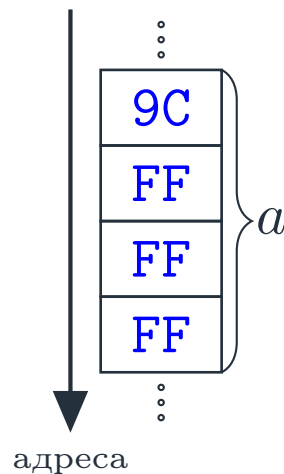
Структуры, объединения и перечисления

Преппроцессор

```
1  #include <stdio.h>
2  int main(int argc, char **argv)
3  {
4      int a = -100;
5      printf("%d, %u, %x, %X\n", a, a, a, a);
6      return 0;
7  }
```

Вывод:

-100, 4294967196, ffffffff9c, FFFFFFFF9C



Записанное в переменной `a` 32-разрядное целое знаковое число `-100` представлено в памяти байтами `9C, FF, FF, FF`. (Обратите внимание: порядок байт числа в памяти – `little-endian`.)
Функция `printf` по-разному выводит значение переменной `a` в зависимости от «типов» в форматных спецификаторах.

Пример: неправильный размер целого числа

Базовые сведения

Введение

Выполнение программ в ОС Linux

Модель данных

Идентификаторы

Литералы

Объявления

Ввод/вывод

Операции

Операторы

Деклараторы

Строки

Структуры, объединения и перечисления

Препроцессор

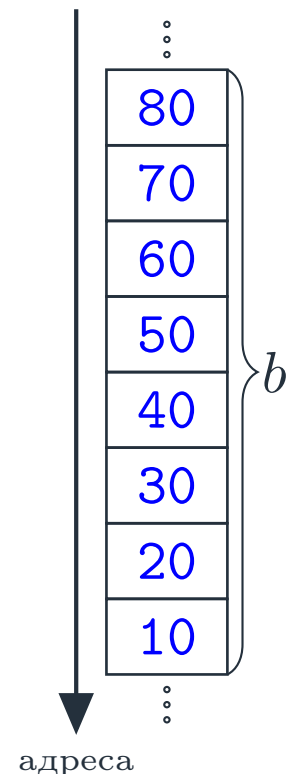
По умолчанию функция `printf` предполагает, что передаваемые ей целочисленные значения имеют размер 32 бита.

```
1  #include <stdio.h>
2
3  int main(int argc, char **argv)
4  {
5      long b = 0x1020304050607080;
6      printf("%x\n", b);
7      return 0;
8  }
```

Вывод:

50607080

В приведённом примере функция `printf` «увидела» только первые 4 байта числа, записанного в переменной `b`.



Размер выводимого целого значения

Базовые сведения

Введение
Выполнение программ в ОС Linux
Модель данных
Идентификаторы
Литералы
Объявления
Ввод/вывод

Операции

Операторы

Деклараторы

Строки

Структуры, объединения и перечисления

Препроцессор

Необязательные поля

%	флаги	ширина	. точность	размер	тип
---	-------	--------	------------	--------	-----

Возможные значения поля «размер»:

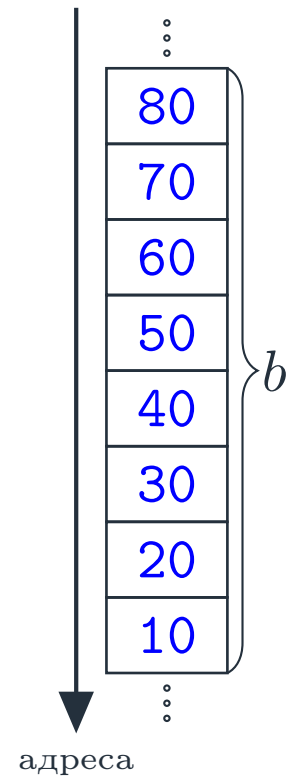
hh 1-байтовое целое;

h 2-байтовое целое;

ll, l 8-байтовое целое.

```
1  #include <stdio.h>
2  int main(int argc, char **argv)
3  {
4      long b = 0x1020304050607080;
5      printf("%hhx_%hx_%llx\n", b,b,b);
6      return 0;
7  }
```

80 7080 1020304050607080



Ширина выводимого значения

Необязательные поля

%	флаги	ширина	. точность	размер	тип
---	-------	--------	------------	--------	-----

Поле «ширина» задаёт минимальное количество знаков, которое должно быть использовано для вывода значения. Если выводимое значение короче указанной ширины, оно дополняется пробелами или нулями в зависимости от указанных «флагов».

```
1  #include <stdio.h>
2  int main(int argc, char **argv)
3  {
4      printf("%2d\n%6d\n", -100, -100);
5      return 0;
6  }
```

```
-100
  -100
```

Точность выводимого значения

Базовые сведения

Введение
Выполнение программ в ОС Linux
Модель данных
Идентификаторы
Литералы
Объявления
Ввод/вывод

Операции

Операторы

Деклараторы

Строки

Структуры, объединения и перечисления

Препроцессор

Необязательные поля

%	флаги	ширина	. точность	размер	тип
---	-------	--------	-------------------	--------	-----

Число, указанное в поле «точность», отделяется от поля «ширина» точкой и имеет разный смысл в зависимости от значения поля «тип»:

d, i, u, x, X

минимальное количество выводимых цифр (короткие числа дополняются нулями);

f, e

количество цифр, которое нужно вывести после десятичной точки;

s

максимальное количество букв строки, которое будет напечатано.

Пример: использование поля «ТОЧНОСТЬ»

Базовые сведения
Введение
Выполнение программ в ОС Linux
Модель данных
Идентификаторы
Литералы
Объявления
Ввод/вывод
Операции
Операторы
Деклараторы
Строки
Структуры, объединения и перечисления
Препроцессор

```
1  #include <stdio.h>
2
3  int main(int argc, char **argv)
4  {
5      printf("%8.6d\n", -100);
6      printf("%.4x\n", 0x41);
7      printf("%10.3f\n", 3.1415);
8      printf("%10.2e\n", 3.1415);
9      printf("%.5s\n", "abcdefghij");
10     return 0;
11 }
```

Вывод:

```
-000100
0041
      3.142
    3.14e+00
abcde
```

Флаги в форматном спецификаторе

Необязательные поля

%	флаги	ширина	. точность	размер	тип
---	-------	--------	------------	--------	-----

В форматном спецификаторе можно указывать сразу несколько флагов:

—

выравнивать значение по левому краю (используется в комбинации с полем «ширина»);

#

добавлять префикс «0x» или «0X» к ненулевому значению (поле «тип» должно содержать «x» или «X», соответственно);

0

дополнять значение слева нулями, а не пробелами (используется в комбинации с полем «ширина»).

Базовые сведения

Введение

Выполнение программ в ОС Linux

Модель данных

Идентификаторы

Литералы

Объявления

Ввод/вывод

Операции

Операторы

Деклараторы

Строки

Структуры, объединения и перечисления

Препроцессор

Для ввода данных из `stdin` в стандартной библиотеке языка C предусмотрена функция `scanf`:

```
scanf(форматная строка, адрес1, адрес2, ...);
```

Форматная строка задаёт количество и типы вводимых значений. Значения сохраняются по указанным адресам.

```
1  #include <stdio.h>
2
3  int main(int argc, char **argv)
4  {
5      int x, y;
6      scanf("%d%d", &x, &y);
7      printf("x= %d, y= %d\n", x, y);
8      return 0;
9  }
```

Обратите внимание на операцию «&» в строке 6 – она возвращает адрес переменной.

Особенности работы функции `scanf`

Базовые сведения

Введение

Выполнение программ в ОС Linux

Модель данных

Идентификаторы

Литералы

Объявления

Ввод/вывод

Операции

Операторы

Деклараторы

Строки

Структуры, объединения и перечисления

Препроцессор

Мы не будем рассматривать детали работы функции `scanf`, т.к. она почти не применяется в индустрии (из-за отсутствия средств проверки правильности ввода). Остановимся на ключевых особенностях её работы:

- синтаксис форматных спецификаторов – почти такой же, как и у `printf`;
- считываемые из `stdin` значения могут разделяться любым количеством *пробельных символов* – пробелов, символов перевода строки и горизонтальной табуляции;
- спецификатор «`%s`» подразумевает считывание строки до первого пробельного символа, т.е. `scanf` нельзя использовать для ввода строк, содержащих пробелы;
- спецификатор «`%c`» означает считывание ровно одного символа, причём этот символ может быть пробельным;
- пробельный символ в форматной строке означает пропускание всех пробельных символов до первого непробельного.

Базовые
сведения

Операции

Выражения

Арифметика

Сравнения

Поразрядные
операции

Логические
операции

Присваивание

Остальные
операции

Приоритеты

Операторы

Деклараторы

Строки

Структуры,
объединения и
перечисления

Препроцессор

Выражение в языке C – это запись формулы, описывающей вычисление некоторого значения.

Выражение состоит из *операций*, обозначающих выполняемые действия, и *операндов*, задающих значения, над которыми выполняются операции.

В зависимости от количества операндов операции бывают:

унарные – 1 операнд;

бинарные – 2 операнда;

тернарные – 3 операнда.

В качестве операндов могут выступать:

- литералы;
- идентификаторы (имена переменных, функций, полей структур);
- подвыражения.

Ещё выражения могут содержать круглые скобки.

Арифметические операции

Базовые сведения
Операции
Выражения
Арифметика
Сравнения
Поразрядные операции
Логические операции
Присваивание
Остальные операции
Приоритеты
Операторы
Деклараторы
Строки
Структуры, объединения и перечисления
Препроцессор

$-a$ унарный минус;
 $a + b$ сложение;
 $a - b$ вычитание;
 $a * b$ умножение;
 a / b деление;
 $a \% b$ остаток от деления.

Операнды арифметических операций могут быть любых числовых типов. Исключение составляет операция «%», которую можно применять только к целочисленным операндам.

Кроме того, типы операндов бинарной операции могут различаться.

Отметим, что деление на ноль приводит к аварийному завершению программы с выдачей, как это ни странно, сообщения «Floating point exception».

Неявные преобразования операндов арифметических операций

Базовые сведения

Операции

Выражения

Арифметика

Сравнения

Поразрядные операции

Логические операции

Присваивание

Остальные операции

Приоритеты

Операторы

Деклараторы

Строки

Структуры, объединения и перечисления

Препроцессор

Перед выполнением операции образы коротких операндов (8- и 16-битовые) расширяются до 32 битов (знаково или беззнаково в зависимости от типа операнда).

Если после расширения образов коротких операндов типы операндов бинарной операции остаются различными, то происходит неявное преобразование одного из операндов:

- если оба операнда – значения с плавающей точкой, то операнд, имеющий меньшую точность, преобразуется к типу операнда, имеющего большую точность;
- если один из операндов – значение с плавающей точкой, а другой операнд – целый, то целый операнд преобразуется к типу операнда с плавающей точкой;
- если операнды – 64-битовое и 32-битовое целые числа, то 32-битовое число расширяется до 64 бит;
- если один из операндов – беззнаковое целое число, а другой – знаковое целое число, то последний трактуется как беззнаковое число.

Пример: расширение коротких операндов

Базовые сведения
Операции
Выражения
Арифметика
Сравнения
Поразрядные операции
Логические операции
Присваивание
Остальные операции
Приоритеты
Операторы
Деклараторы
Строки
Структуры, объединения и перечисления
Препроцессор

```
1  #include <stdio.h>
2
3  int main(int argc, char **argv)
4  {
5      char x = -128;
6      int y = -x;
7      printf("%d\n", y);
8      return 0;
9  }
```

Вывод:

128

Если бы перед применением унарного «-» значение переменной `x` не было бы расширено до `int`, мы бы не получили 128, так как это число не помещается в диапазон значений типа `char`: $-128 \leq x \leq 127$.

Пример: неявное преобразование знакового числа в беззнаковое

Базовые сведения
Операции
Выражения
Арифметика
Сравнения
Поразрядные операции
Логические операции
Присваивание
Остальные операции
Приоритеты
Операторы
Деклараторы
Строки
Структуры, объединения и перечисления
Препроцессор

```
1  #include <stdio.h>
2
3  int main(int argc, char **argv)
4  {
5      int x = -100;
6      unsigned int y = 5;
7      printf("%d\n", x/y);
8      return 0;
9  }
```

Вывод:

858993439

Так как тип переменной y – беззнаковый, то значение x трактуется как беззнаковое число $4\,294\,967\,196 = 2^{32} - 100$. Деление этого числа на 5 как раз и даёт 858 993 439.

Пример: переполнение результата операции

Базовые сведения
Операции
Выражения
Арифметика
Сравнения
Поразрядные операции
Логические операции
Присваивание
Остальные операции
Приоритеты
Операторы
Деклараторы
Строки
Структуры, объединения и перечисления
Препроцессор

```
1  #include <stdio.h>
2
3  int main(int argc, char **argv)
4  {
5      unsigned int x = 3000000000, y = 2000000000;
6      unsigned long z = x + y;
7      printf("%lld\n", z);
8      return 0;
9  }
```

Вывод:

705032704

Оба операнда операции «+» – 32-битовые целые числа, поэтому перед выполнением сложения они не расширяются. Сумма x и y – это число 5 000 000 000. Оно не помещается в диапазон значений типа `unsigned int`, т.к. его размер – 33 бита. Поэтому старший единичный бит отбрасывается, и в результате мы получаем 705 032 704.

Округление результата деления

Базовые
сведения

Операции

Выражения

Арифметика

Сравнения

Поразрядные
операции

Логические
операции

Присваивание

Остальные
операции

Приоритеты

Операторы

Деклараторы

Строки

Структуры,
объединения и
перечисления

Препроцессор

При делении двух целых чисел дробная часть результата отбрасывается. Тем самым, происходит округление¹ результата в меньшую сторону: $\left\lfloor \frac{a}{b} \right\rfloor$.

Для округления результата в большую сторону следует воспользоваться формулой

$$\left\lceil \frac{a}{b} \right\rceil = \left\lfloor \frac{a + b - 1}{b} \right\rfloor.$$

Действительно,

$$\left\lceil \frac{6}{3} \right\rceil = \left\lfloor \frac{6 + 3 - 1}{3} \right\rfloor = 2,$$

$$\left\lceil \frac{7}{3} \right\rceil = \left\lfloor \frac{7 + 3 - 1}{3} \right\rfloor = 3.$$

¹Мы будем обозначать округление числа x в меньшую и большую сторону как $\lfloor x \rfloor$ и $\lceil x \rceil$, соответственно.

Инкремент и декремент

Базовые сведения

Операции

Выражения

Арифметика

Сравнения

Поразрядные операции

Логические операции

Присваивание

Остальные операции

Приоритеты

Операторы

Деклараторы

Строки

Структуры, объединения и перечисления

Препроцессор

++a префиксный инкремент: увеличивает **a** на единицу и возвращает полученное значение;

--a префиксный декремент: уменьшает **a** на единицу и возвращает полученное значение;

a++ постфиксный инкремент: увеличивает **a** на единицу и возвращает значение, которое было до увеличения;

a-- постфиксный декремент: уменьшает **a** на единицу и возвращает значение, которое было до увеличения.

Операнд операций инкремента и декремента должен быть *левым значением*, т.е. обозначать некоторую переменную в памяти.

Не рекомендуется в одном выражении применять эти операции к одной и той же переменной более одного раза.

Пример: постфиксный и префиксный инкремент

Базовые сведения
Операции
Выражения
Арифметика
Сравнения
Поразрядные операции
Логические операции
Присваивание
Остальные операции
Приоритеты
Операторы
Деклараторы
Строки
Структуры, объединения и перечисления
Препроцессор

```
1  #include <stdio.h>
2
3  int main(int argc, char **argv)
4  {
5      int a = 10, b = 10;
6      printf("%d\n", a++);
7      printf("%d\n", a);
8      printf("%d\n", ++b);
9      printf("%d\n", b);
10     return 0;
11 }
```

Вывод:

```
10
11
11
11
```

Постфиксный инкремент в строке 6 увеличил переменную `a`, но вернул её старое значение.

Пример: ошибки, связанные с неправильным использованием инкремента и декремента

Базовые сведения
Операции
Выражения
Арифметика
Сравнения
Поразрядные операции
Логические операции
Присваивание
Остальные операции
Приоритеты
Операторы
Деклараторы
Строки
Структуры, объединения и перечисления
Препроцессор

```
1  #include <stdio.h>
2
3  int main(int argc, char **argv)
4  {
5      int a = 10;
6      a++ ++;    /* Ошибка! */
7      5--;       /* Ошибка! */
8      ++(a*a);   /* Ошибка! */
9      return 0;
10 }
```

Сообщения компилятора:

```
ts.c: In function 'main':
ts.c:6:6: error: lvalue required as increment operand
ts.c:7:3: error: lvalue required as increment operand
ts.c:8:2: error: lvalue required as increment operand
```

Сообщения об ошибках говорят о том, что операнд инкремента должен являться левым значением (lvalue).

Базовые
сведения

Операции

Выражения

Арифметика

Сравнения

Поразрядные
операции

Логические
операции

Присваивание

Остальные
операции

Приоритеты

Операторы

Деклараторы

Строки

Структуры,
объединения и
перечисления

Препроцессор

$a < b$ меньше;
 $a > b$ больше;
 $a \leq b$ меньше или равно;
 $a \geq b$ больше или равно;
 $a == b$ равно;
 $a != b$ не равно.

В языке C отсутствует специальный тип данных для булевских значений. Вместо него используется тип `int`: значение 0 обозначает «ложь», все остальные значения – «истина».

Операции сравнения возвращают значения 0 и 1.

Операнды операций сравнения подвергаются тем же неявным преобразованиям, что и операнды арифметических операций.

Операции «`==`» и «`!=`» следует с осторожностью применять к операндам с плавающей точкой. Из-за возможной потери точности два числа, которые по логике решаемой задачи должны быть равны, могут оказаться не равны.

Пример: сравнение знакового и беззнакового чисел

Базовые сведения

Операции

Выражения

Арифметика

Сравнения

Поразрядные операции

Логические операции

Присваивание

Остальные операции

Приоритеты

Операторы

Деклараторы

Строки

Структуры, объединения и перечисления

Препроцессор

```
1  #include <stdio.h>
2
3  int main(int argc, char **argv)
4  {
5      int x = -100;
6      unsigned int y = 100;
7      printf("%d\n", x < y);
8      return 0;
9  }
```

Предупреждение компилятора (указана опция «-Wextra»):

```
ts.c: In function 'main':
ts.c:7:19: warning: comparison between signed and
unsigned integer expressions [-Wsign-compare]
```

Значение переменной `x` трактуется как беззнаковое число $2^{32} - 100$, которое больше 100. Поэтому операция «`<`» возвратит 0, а не 1, хотя по логике вещей $-100 < 100$.

Поразрядные логические операции

Базовые сведения

Операции

Выражения

Арифметика

Сравнения

Поразрядные операции

Логические операции

Присваивание

Остальные операции

Приоритеты

Операторы

Деклараторы

Строки

Структуры, объединения и перечисления

Препроцессор

$\sim a$ поразрядное НЕ;
 $a \& b$ поразрядное И;
 $a | b$ поразрядное ИЛИ;
 $a \wedge b$ поразрядное ИСКЛЮЧАЮЩЕЕ ИЛИ.

Операнды поразрядных операций – целые числа.

Операция « \sim » применяет логическое НЕ к каждому биту операнда и возвращает получаемое значение.

Результат операций « $\&$ », « $|$ » и « \wedge » формируется путём применения соответствующей логической операции ко всем парам битов, которые стоят на одинаковых позициях.

a_i	b_i	$(a \& b)_i$
0	0	0
0	1	0
1	0	0
1	1	1

a_i	b_i	$(a b)_i$
0	0	0
0	1	1
1	0	1
1	1	1

a_i	b_i	$(a \wedge b)_i$
0	0	0
0	1	1
1	0	1
1	1	0

Примеры: поразрядные логические операции

Базовые сведения

Операции

Выражения

Арифметика

Сравнения

Поразрядные операции

Логические операции

Присваивание

Остальные операции

Приоритеты

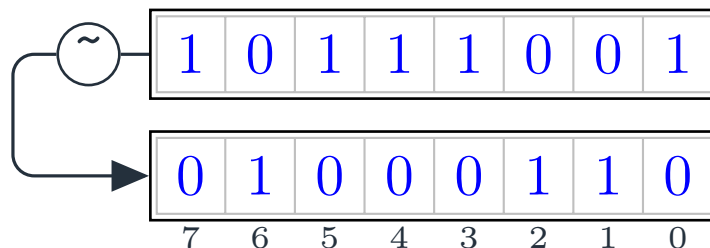
Операторы

Деклараторы

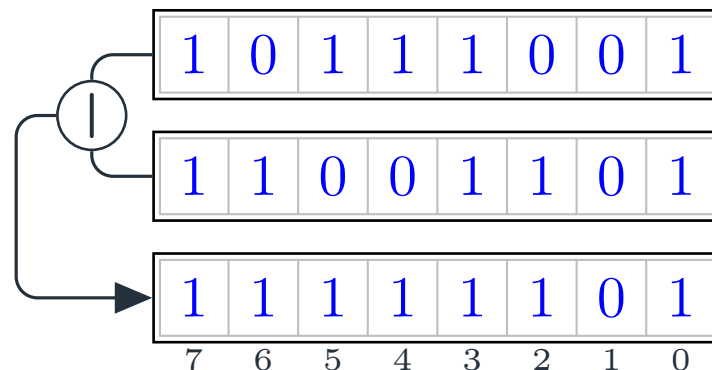
Строки

Структуры, объединения и перечисления

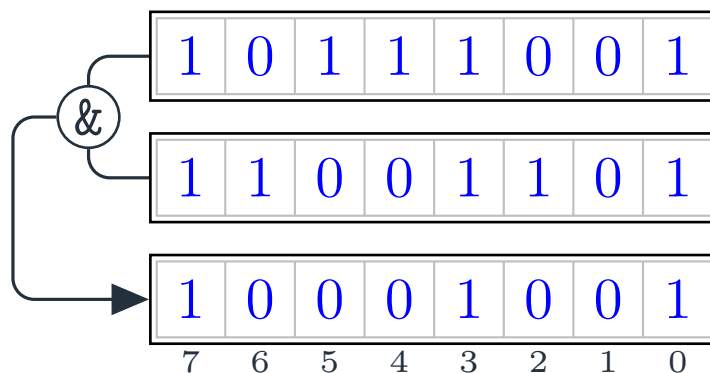
Препроцессор



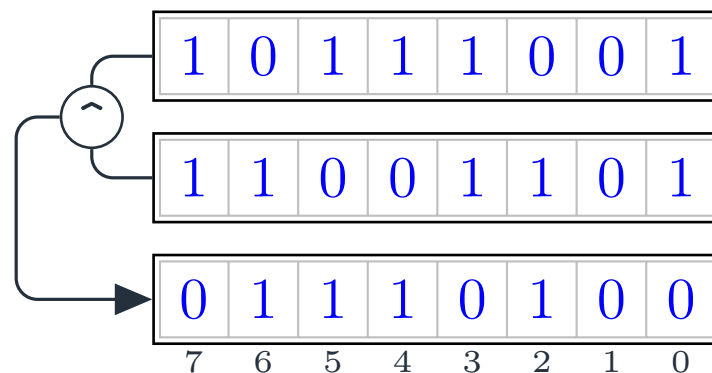
Поразрядное НЕ



Поразрядное ИЛИ



Поразрядное И



Поразрядное
ИСКЛЮЧАЮЩЕЕ ИЛИ

Операции поразрядного сдвига

Базовые сведения

Операции

Выражения

Арифметика

Сравнения

Поразрядные операции

Логические операции

Присваивание

Остальные операции

Приоритеты

Операторы

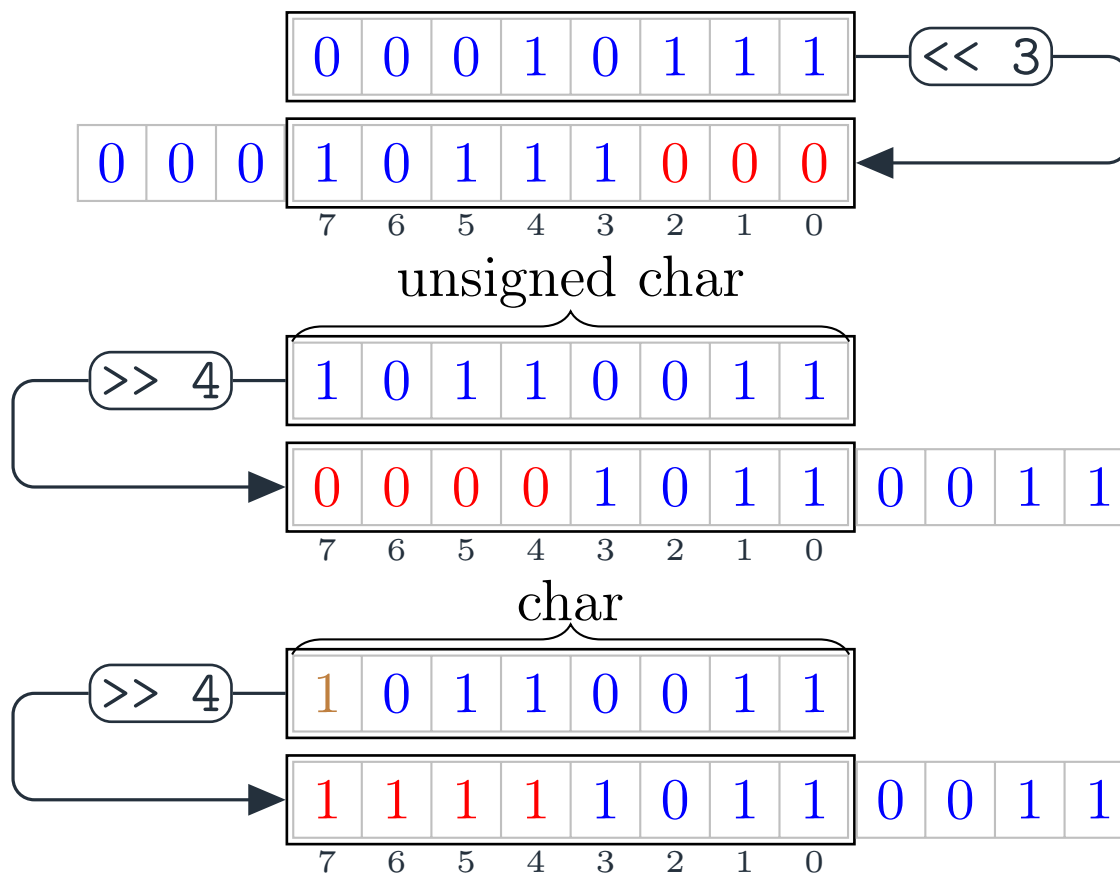
Деклараторы

Строки

Структуры, объединения и перечисления

Преппроцессор

$a \ll b$ сдвиг целого a влево на b битов: $a \cdot 2^b$;
 $a \gg b$ сдвиг целого a вправо на b битов: $a / 2^b$.



При сдвиге знакового числа вправо освободившиеся старшие биты заполняются значением знакового бита.

Базовые
сведения

Операции

Выражения

Арифметика

Сравнения

Поразрядные
операции

Логические
операции

Присваивание

Остальные
операции

Приоритеты

Операторы

Деклараторы

Строки

Структуры,
объединения и
перечисления

Препроцессор

`!a` логическое НЕ;
`a && b` логическое И;
`a || b` логическое ИЛИ.

Операция «`!`» превращает «истину» в «ложь», и наоборот.
Операция «`&&`» возвращает «истину», если оба её операнда равны «истине». В противном случае она возвращает «ЛОЖЬ».

Операция «`||`» возвращает «ложь», если оба её операнда равны «лжи». В противном случае она возвращает «истину».

Особенностью операций «`&&`» и «`||`» является то, что они не всегда вычисляют второй операнд. Операция «`&&`» вычисляет второй операнд только в случае, если первый операнд равен «истине». Операция «`||`» вычисляет второй операнд только в случае, если первый операнд равен «лжи».

Пример: логические операции не всегда вычисляют второй операнд

Базовые сведения

Операции

Выражения

Арифметика

Сравнения

Поразрядные операции

Логические операции

Присваивание

Остальные операции

Приоритеты

Операторы

Деклараторы

Строки

Структуры, объединения и перечисления

Препроцессор

```
1  #include <stdio.h>
2
3  int main(int argc, char **argv)
4  {
5      int a = 10, b = 10;
6      printf("%d\n", (a == b) || 5/0);
7      printf("%d\n", (a != b) && 5/0);
8      return 0;
9  }
```

Вывод:

1
0

Если бы второй операнд всегда вычислялся, программа бы завершилась аварийно по делению на ноль.

Операция присваивания

Базовые сведения

Операции

Выражения

Арифметика

Сравнения

Поразрядные операции

Логические операции

Присваивание

Остальные операции

Приоритеты

Операторы

Деклараторы

Строки

Структуры, объединения и перечисления

Препроцессор

$a = b$ присвоить переменной a значение b .

Первый операнд операции присваивания должен являться левым значением.

Операция присваивания возвращает присвоенное значение, и оно может выступать в роли операнда другой операции:

```
x = 2 * (y = i + j);
```

Операция присваивания *правоассоциативна*, то есть в выражении вида $a_1 = a_2 = \dots = a_n$ операции присваивания выполняются справа налево. Это, в частности, даёт возможность компактно записать присваивание одного значения нескольким переменным:

```
i = j = 0;
```

Операция присваивания может выполнять неявное преобразование значения, чтобы оно соответствовало типу переменной в левой части.

Операции составного присваивания

Базовые
сведения

Операции

Выражения

Арифметика

Сравнения

Поразрядные
операции

Логические
операции

Присваивание

Остальные
операции

Приоритеты

Операторы

Деклараторы

Строки

Структуры,
объединения и
перечисления

Преппроцессор

$a \text{ op} = b$ эквивалентно $a = a \text{ op} b$.

Операции «+», «-», «*», «/», «%», «&», «^», «|», «<<» и «>>» допускают сочетание с операцией присваивания.

```
1  #include <stdio.h>
2
3  int main(int argc, char **argv)
4  {
5      int a = 10, b = 20;
6      printf("%d, ", a += b *= 15);
7      printf("a=%d, b=%d\n", a, b);
8      return 0;
9  }
```

Вывод:

```
310, a = 310, b = 300
```

Тернарная условная операция

Базовые сведения

Операции

Выражения

Арифметика

Сравнения

Поразрядные операции

Логические операции

Присваивание

Остальные операции

Приоритеты

Операторы

Деклараторы

Строки

Структуры, объединения и перечисления

Препроцессор

`a ? b : c` возвращает `b` или `c` в зависимости от `a`.

Если `a` – «истина», тернарная операция вычисляет и возвращает значение `b`. В противном случае она вычисляет и возвращает значение `c`.

```
1  #include <stdio.h>
2
3  int main(int argc, char **argv)
4  {
5      printf("%d\n", (10 == 10) ? 2+3 : 5/0);
6      return 0;
7  }
```

Вывод:

5

Если бы третий операнд тернарной операции всегда вычислялся, программа бы завершилась аварийно по делению на ноль.

Операция приведения типа

Базовые
сведения

Операции

Выражения

Арифметика

Сравнения

Поразрядные
операции

Логические
операции

Присваивание

Остальные
операции

Приоритеты

Операторы

Деклараторы

Строки

Структуры,
объединения и
перечисления

Препроцессор

(*тип*)*a* преобразует значение *a* к указанному типу.

Эта унарная операция создаёт новое значение нужного типа на основе значения операнда. Если при этом необходимо выполнить сужающее преобразование (например, приведение значения типа `int` к `char`), то часть битов операнда может быть отброшена.

С помощью операции приведения типа можно попытаться решить проблему со сравнением знакового и беззнакового чисел:

```
1  #include <stdio.h>
2
3  int main(int argc, char **argv)
4  {
5      int x = -100;
6      unsigned int y = 100;
7      printf("%d\n", x < (int)y);
8      return 0;
9  }
```

Операция вычисления размера

Базовые сведения

Операции

Выражения

Арифметика

Сравнения

Поразрядные операции

Логические операции

Присваивание

Остальные операции

Приоритеты

Операторы

Деклараторы

Строки

Структуры, объединения и перечисления

Препроцессор

`sizeof(mun)` возвращает размер значения указанного типа в байтах;

`sizeof a` возвращает размер переменной в байтах.

Эта операция нужна для разработки переносимых программ. Дело в том, что типы данных в разных компиляторах языка C на разных аппаратных платформах могут иметь разные размеры.

```
1  #include <stdio.h>
2
3  int main(int argc, char **argv)
4  {
5      int x;
6      printf("%u_ %u\n", sizeof x, sizeof(long double));
7      return 0;
8  }
```

4 16

Операция «запятая»

Базовые сведения

Операции

Выражения

Арифметика

Сравнения

Поразрядные операции

Логические операции

Присваивание

Остальные операции

Приоритеты

Операторы

Деклараторы

Строки

Структуры, объединения и перечисления

Препроцессор

`a, b` вычисляет `a` и `b` и возвращает значение `b`.

Эта операция играет вспомогательную роль. Забегая вперёд, отметим, что она бывает полезна в заголовке цикла `for` и при составлении макросов.

```
1  #include <stdio.h>
2
3  int main(int argc, char **argv)
4  {
5      int x = 10;
6      int y = (x + 2, x*10);
7      printf("%d\n", y);
8      return 0;
9  }
```

Вывод:

100

Приоритет и ассоциативность операций

Базовые
сведения

Операции

Выражения

Арифметика

Сравнения

Поразрядные
операции

Логические
операции

Присваивание

Остальные
операции

Приоритеты

Операторы

Деклараторы

Строки

Структуры,
объединения и
перечисления

Препроцессор

Рассмотрим ситуацию, когда в выражении две операции op_1 и op_2 имеют общий операнд b :

$$a \ op_1 \ b \ op_2 \ c$$

В языке С операции упорядочены по *приоритету*. «Спорный» операнд b будет принадлежать той операции, у которой выше приоритет.

Если приоритеты операций op_1 и op_2 оказываются равны, то решение принимается на основе соответствующей их уровню приоритета *ассоциативности*:

- b принадлежит op_1 в случае левой ассоциативности;
- b принадлежит op_2 в случае правой ассоциативности.

С помощью этих правил и таблицы приоритетов можно определить порядок выполнения операций в выражении.

Таблица приоритетов операций

Категория	Номер	Символы	Ассоциативность	Направление
Базовые сведения	1	[] . -> ++ -- ()	постфикс. унар.	лев.
	2	! ~ + - ++ --	префикс. унар.	прав.
Выражения		sizeof & * ()		
	3	* / %	бинарные	лев.
Арифметика	4	+ -	бинарные	лев.
Сравнения	5	<< >>	бинарные	лев.
Поразрядные операции	6	< <= > >=	бинарные	лев.
Логические операции	7	== !=	бинарные	лев.
Присваивание	8	&	бинарная	лев.
Остальные операции	9	~	бинарная	лев.
Приоритеты	10		бинарная	лев.
	11	&&	бинарная	лев.
Операторы	12		бинарная	лев.
Деклараторы	13	? :	тернарная	прав.
Строки	14	= += -= *= /= %=	бинарные	прав.
		&= ^= = <<= >>=		
Структуры, объединения и перечисления	15	,	бинарные	лев.
Препроцессор				

индексация

вызов функции

приведение типа

разыменование

Уменьшение приоритета

73 / 164

Проблемы с приоритетами операций

Базовые сведения

Операции

Выражения

Арифметика

Сравнения

Поразрядные операции

Логические операции

Присваивание

Остальные операции

Приоритеты

Операторы

Деклараторы

Строки

Структуры, объединения и перечисления

Препроцессор

Исторически так сложилось, что приоритеты операций «&», «^» и «|» в языке С выбраны неудачно. Это служит источником трудноуловимых ошибок.

Например, пусть требуется проверить, что два младших бита целого числа x – единичные. Для этого совершенно естественно было бы записать выражение

$$x \& 3 == 3$$

Однако, приоритет операции «==» выше приоритета операции «&». Поэтому выражение будет воспринято компилятором языка С как

$$x \& (3 == 3)$$

При этом компилятор не выдаст никаких сообщений.

Для исправления ситуации нужно явно поставить круглые скобки в правильном месте:

$$(x \& 3) == 3$$

Простые операторы

Базовые сведения

Операции

Операторы

Простые операторы

Блок

if...else

while

for

do...while

break

continue

goto

switch

Оформление кода

Деклараторы

Строки

Структуры, объединения и перечисления

Препроцессор

Оператор – это синтаксическая конструкция, которая описывает некоторое действие, совершаемое программой, и при этом, в отличие от выражения, не возвращает значения.

Операторы бывают *простыми* и *составными*. К простым операторам относятся *пустой оператор*

;

и *оператор-выражение*

выражение ;

Пустой оператор ничего не делает, а оператор-выражение вычисляет выражение. Примеры операторов-выражений:

```
v = a * b * c;  
printf ("%d\n", v);
```

(Напомним, что функция `printf` возвращает количество выведенных символов, но в данном операторе её возвращаемое значение не используется.)

Простейшим составным оператором является *блок*, представляющий последовательность операторов:

```
{  
    оператор1  
    оператор2  
    ...  
    операторN  
}
```

Выполнение блока означает последовательный запуск операторов, из которых он состоит.

Тело функции `main` и любой другой функции представляет собой не что иное, как блок.

В стандарте ANSI языка C определено, что объявления переменных располагаются строго в начале блока до первого оператора. Диалект C, реализованный в компиляторе gcc, ослабляет это ограничение и разрешает размещать объявления переменных в любом месте блока.

Оператор выбора if ... else

Базовые сведения

Операции

Операторы

Простые операторы

Блок

if...else

while

for

do...while

break

continue

goto

switch

Оформление кода

Деклараторы

Строки

Структуры, объединения и перечисления

Препроцессор

Оператор *выбора* задаёт развилку в коде и имеет в языке C две формы, отличающиеся количеством ветвей:

заголовок
if (условие) оператор
положительная ветвь

а также

положительная ветвь
if (условие) оператор
else оператор
отрицательная ветвь

Здесь условие – это некоторое выражение. Если оно принимает ненулевое значение, то выполняется положительная ветвь оператора `if`. Если значение условия равно нулю, то выполняется отрицательная ветвь (если она существует).

Пример: определение високосности года

Базовые сведения
Операции
Операторы
Простые операторы
Блок
if...else
while
for
do...while
break
continue
goto
switch
Оформление кода
Деклараторы
Строки
Структуры, объединения и перечисления
Препроцессор

```
1  #include <stdio.h>
2
3  int main(int argc, char **argv)
4  {
5      int y;
6      scanf("%d", &y);
7
8      if (y%4) printf("невисокосный");
9      else {
10         if (y%100 || !(y%400)) printf("високосный");
11         else printf("невисокосный");
12     }
13     return 0;
14 }
```

Diagram illustrating the logic of the leap year calculation:

- `y%4 != 0` points to the first `if` condition.
- `y%100 != 0` points to the first part of the `||` condition.
- `y%400 == 0` points to the `!(y%400)` part of the `||` condition.

По григорианскому календарю год – високосный, если он делится на 4, но при этом либо не делится на 100, либо делится на 400.

То есть 1900 и 2013 – невисокосные годы, а 2000 и 2012 – високосные.

Вложенные операторы if

Базовые сведения

Операции

Операторы

Простые операторы

Блок

if...else

while

for

do...while

break

continue

goto

switch

Оформление кода

Деклараторы

Строки

Структуры, объединения и перечисления

Преппроцессор

Тонким моментом синтаксиса языка С является конструкция вида

```
if (условие1) if (условие2) оператор1 else оператор2
                вложенный оператор if
```

В этой конструкции отрицательная ветвь относится ко второму (вложенному) оператору if. Если нужно, чтобы она относилась к первому (внешнему) оператору if, то вложенный if необходимо заключить в блок:

```
if (условие1) {
    if (условие2) оператор1
} else оператор2
```

Вообще, вложенные операторы if считаются плохой практикой программирования. Если есть возможность их избежать, этой возможностью нужно воспользоваться.

Цикл с предусловием while

Цикл с предусловием – это цикл, перед каждой итерацией которого проверяется некоторое условие. Цикл продолжается до тех пор, пока условие истинно.

В языке С предусмотрены две конструкции цикла с пред-
условием: `while` и `for`. Цикл `while` записывается как

$$\overbrace{\text{while (условие)}}^{\text{заголовок}} \underbrace{\text{оператор}}_{\text{тело}}$$

Оператор, являющийся телом цикла **while**, выполняется многократно до тех пор, пока условие не станет равно 0.

```
i = 0;
while (i < 10) printf("%d_", i++);
```

0 1 2 3 4 5 6 7 8 9

Тело цикла может не выполняться ни разу, если условие равно 0 сразу перед выполнением цикла.

Цикл с предусловием for

Базовые сведения

Операции

Операторы

Простые операторы

Блок

if...else

while

for

do...while

break

continue

goto

switch

Оформление кода

Деклараторы

Строки

Структуры, объединения и перечисления

Препроцессор

Цикл с предусловием **for** имеет следующий синтаксис:

заголовок **тело**

for (инициализация; условие; завершение) оператор

Здесь «инициализация», «условие» и «завершение» – некоторые выражения. По сути цикл **for** – это сокращённая запись следующего фрагмента кода:

```
инициализация;  
while (условие) {  
    оператор  
    завершение;  
}
```

Т.е. «инициализация» выполняется один раз перед выполнением цикла, «условие» проверяется перед каждой итерацией, а «завершение» выполняется после каждой итерации.

```
for (i = 0; i < 10; i++) printf("%d_", i);
```

Цикл с постусловием do ... while

Базовые сведения

Операции

Операторы

Простые операторы

Блок

if...else

while

for

do...while

break

continue

goto

switch

Оформление кода

Деклараторы

Строки

Структуры, объединения и перечисления

Препроцессор

Цикл с постусловием – это цикл, после каждой итерацией которого проверяется некоторое условие. Цикл продолжается до тех пор, пока условие истинно.

В языке C цикл с постусловием имеет вид

```
do оператор заголовокwhile (условие);  
           тело
```

Оператор, являющийся телом цикла `do ... while`, выполняется многократно до тех пор, пока условие не станет равно 0. Тело цикла выполняется хотя бы один раз, даже если условие равно 0 сразу перед выполнением цикла.

```
i = 0;  
do printf("%d_", i++); while (i < 10);
```

Цикл с постусловием широко применяется в случае, если значения переменных, входящих в условие, до выполнения первой итерации цикла не определены.

Цикл без условия и оператор break

Базовые сведения

Операции

Операторы

Простые операторы

Блок

if...else

while

for

do...while

break

continue

goto

switch

Оформление кода

Деклараторы

Строки

Структуры, объединения и перечисления

Препроцессор

Цикл без условия – это цикл, выход из которого либо вообще не предусмотрен, либо осуществляется внутри его тела. Учитывая, что любое из трёх выражений в конструкции цикла `for` может быть не указано, канонической формой цикла без условия в языке C является конструкция

```
for (;;) оператор
```

Для выхода из любого цикла (не только из цикла без условия) в языке C предусмотрен оператор

```
break;
```

Пример:

```
i = 0;
for (;;) {
    if (i == 10) break;
    printf("%d_", i++);
}
```

Для того чтобы прервать текущую итерацию цикла и перейти к проверке условия и, возможно, следующей итерации, существует оператор

```
continue;
```

Пример:

```
for (i = 0; i < 10; i++) {  
    if (i%2 == 1) continue;  
    printf("%d ", i);  
}
```

Вывод:

```
0 2 4 6 8
```

Оператор `continue` (как, впрочем, и `break`) позволяет в ряде случаев упростить организацию тела цикла, уменьшив в нём вложенность синтаксических конструкций.

Метки и оператор goto

Базовые сведения

Операции

Операторы

Простые операторы

Блок

if...else

while

for

do...while

break

continue

goto

switch

Оформление кода

Деклараторы

Строки

Структуры, объединения и перечисления

Препроцессор

Любой оператор может быть помечен:

метка: оператор

Здесь «метка» – это произвольный идентификатор.


Объявление переменных не является оператором и помечено быть не может.

На любой помеченный оператор можно передать управление с помощью оператора **goto**:

goto метка;

Пример:

```
i = 0;
repeat: printf("%d", i++);
if (i < 10) goto repeat;
```



Забегая вперёд, отметим, что оператор **goto** можно использовать для передачи управления строго в рамках тела функции, которой он принадлежит.

Оператор множественного выбора **switch** выполняет переход на ту или иную метку в зависимости от значения выражения, которое должно быть целым числом:

заголовок тело
switch (выражение) оператор

В пределах тела оператора **switch** могут использоваться специальные метки:

```
case целочисленный_литерал
default
```

Оператор **switch** передаёт управление на метку, литерал которой совпадает со значением выражения. Если такой метки нет, но есть метка **default**, то управление передаётся на неё. Если же и метки **default** нет, то управление передаётся на оператор, следующий за оператором **switch**.

Для выхода из середины тела оператора **switch** можно использовать оператор **break**.

Пример: программа, вычисляющая простейшие арифметические выражения вида «a op b»

Базовые сведения	1	<code>#include <stdio.h></code>
	2	
Операции	3	<code>int main(int argc, char **argv)</code>
Операторы	4	<code>{</code>
Простые операторы	5	<code>int a, b;</code>
Блок	6	<code>char op;</code>
if...else	7	<code>printf("Введите выражение вида a op b: ");</code>
while	8	<code>scanf("%d%c%d", &a, &op, &b);</code>
for	9	<code>switch (op) {</code>
do...while	10	<code>case '+':</code>
break	11	<code>printf("%d\n", a + b);</code>
continue	12	<code>break;</code>
goto	13	<code>case '-':</code>
switch	14	<code>printf("%d\n", a - b);</code>
Оформление кода	15	<code>break;</code>
	16	<code>... ← Аналогично для '*' и '/'</code>
Деклараторы	17	<code>default:</code>
Строки	18	<code>printf("неправильная операция\n");</code>
Структуры, объединения и перечисления	19	<code>}</code>
	20	<code>return 0;</code>
Препроцессор	21	<code>}</code>

Вложенность и отступы

Базовые сведения

Операции

Операторы

Простые операторы

Блок

if...else

while

for

do...while

break

continue

goto

switch

Оформление кода

Деклараторы

Строки

Структуры, объединения и перечисления

Препроцессор

Вложенность оператора или объявления – это количество блоков, в которые оператор или объявление входит.

Отступ размера n – это последовательность из n символов табуляции в начале строки программы.

```
int main(int argc, char **argv)
{
```

```
    ...
    (←таб)объявление←Вложенность: 1
```

```
    {
```

```
        ...
        {
```

```
            ...
            (←таб)(←таб)(←таб)оператор←Вложенность: 3
```

```
            ...
```

```
        }
```

```
    ...
```

```
    }
```

```
    ...
```

```
}
```


Правила расположения оператора, не являющегося блоком

Базовые сведения

Операции

Операторы

Простые операторы

Блок

if...else

while

for

do...while

break

continue

goto

switch

Оформление кода

Деклараторы

Строки

Структуры, объединения и перечисления

Препроцессор

1. Если оператор не является телом другого оператора, то он располагается в отдельной строчке программы с отступом, который равен его вложенности.

```
(←таб→)...(←таб→)оператор
```

2. В случае, когда оператор – тело цикла do ... while, он располагается справа от do и слева от while.

```
... do оператор while (условие);
```

3. Если оператор – тело другого оператора или положительная ветвь if, то он размещается справа от заголовка.

```
... while (условие) оператор
```

4. Если оператор – отрицательная ветвь if, то он размещается справа от else.

```
... else оператор
```

Правила расположения блока

Базовые сведения

Операции

Операторы

Простые операторы

Блок

if...else

while

for

do...while

break

continue

goto

switch

Оформление кода

Деклараторы

Строки

Структуры, объединения и перечисления

Препроцессор

1. Открывающая фигурная скобка блока всегда стоит в конце строки программы.

Закрывающая фигурная скобка блока всегда стоит в начале строки программы и предваряется отступом, равным вложенности блока.

```
... {  
    ...  
    (←таб→)...(←таб→)} ...
```

2. Если блок – тело цикла `do ... while`, то его открывающая скобка располагается справа от `do`, а закрывающая скобка – слева от `while`.

```
... do {  
    ...  
} while (условие);
```

Правила расположения блока (продолжение)

Базовые сведения

Операции

Операторы

Простые операторы

Блок

if...else

while

for

do...while

break

continue

goto

switch

Оформление кода

Деклараторы

Строки

Структуры, объединения и перечисления

Препроцессор

3. Если блок – тело `for`, `while` или `switch`, то его открывающая скобка располагается справа от заголовка.

```
... while (условие) {  
    ...  
}
```

4. Если блок – положительная ветвь `if`, его открывающая скобка помещается справа от заголовка, а закрывающая – слева от `else`, если есть отрицательная ветвь.

```
... if (условие) {  
    ...  
} else ...
```

5. Если блок – отрицательная ветвь `if`, его открывающая скобка помещается справа от `else`.

```
... else {  
    ...  
}
```

Другие правила оформления кода

Базовые сведения

Операции

Операторы

Простые операторы

Блок

if...else

while

for

do...while

break

continue

goto

switch

Оформление кода

Деклараторы

Строки

Структуры, объединения и перечисления

Препроцессор

1. Объявление располагается в отдельной строке программы с отступом, который равен его вложенности.

```
(←таб→)...(←таб→)объявление
```

2. Если положительная ветвь `if` – не блок, то `else` стоит в начале строки программы и предваряется отступом, который равен вложенности оператора `if`.

```
... if (условие) ...  
(←таб→)...(←таб→)else ...
```

3. Метки `case` и `default` оператора `switch` размещаются на отдельных строках и предваряются отступом, который равен вложенности оператора `switch`.

```
... switch (выражение) ... {  
    ...  
(←таб→)...(←таб→)case литерал:  
    ...  
}
```

Объявление массива: декларатор []

Базовые сведения

Операции

Операторы

Деклараторы

Массивы

Указатели

Сложные объявления

Типовые литералы

Функции

Valgrind

Псевдонимы

Динамическая память

Строки

Структуры, объединения и перечисления

Препроцессор

Для объявления переменных составных типов данных к имени переменной в объявлении применяется специальная операция, называемая *декларатором*. Следует понимать, что декларатор не порождает исполняемого кода, а является всего лишь инструкцией компилятору.

`a[размер]` декларатор, сообщающий компилятору, что `a` является массивом указанного размера.

Например, объявим массив из 10 элементов типа `int`:

```
int arr[10];
```

Если бы не декларатор «[]», переменная `arr` имела бы тип `int`, но декларатор модифицировал её тип, и она стала массивом `int`'ов.

Применение ещё одного декларатора превращает `arr` в двумерный массив (матрица 10×20):

```
int arr[10][20];
```

Операция индексации массива

Базовые
сведения

Операции

Операторы

Деклараторы

Массивы

Указатели

Сложные
объявления

Типовые лите-
ралы

Функции

Valgrind

Псевдонимы

Динамическая
память

Строки

Структуры,
объединения и
перечисления

Препроцессор

`a[b]` возвращает левое значение, соответствующее элементу массива `a`, имеющему индекс `b`.

Для иллюстрации использования операции «`[]`» приведём программу, осуществляющую ввод и вывод массива из 12 `float`'ов:

```
1  #include <stdio.h>
2
3  int main(int argc, char **argv)
4  {
5      int i
6      float a[12];
7      for (i = 0; i < 12; i++)
8          scanf("%f", &a[i]);
9      for (i = 0; i < 12; i++)
10         printf("%f ", a[i]);
11     return 0;
12 }
```

В программе приходится использовать циклы, так как функции `scanf` и `printf` массивов не поддерживают.

Массивы переменного размера

Базовые сведения
Операции
Операторы
Деклараторы
Массивы
Указатели
Сложные объявления
Типовые литералы
Функции
Valgrind
Псевдонимы
Динамическая память
Строки
Структуры, объединения и перечисления
Препроцессор

Допускается использовать переменную в качестве размера массива, размещаемого в автоматической памяти. Продемонстрируем это на примере программы, вычисляющей сумму элементов массива, размер которого становится известен только во время выполнения программы:

```
1  #include <stdio.h>
2
3  int main(int argc, char **argv)
4  {
5      long n, i;
6      scanf("%ld", &n);
7      int a[n];
8      for (i = 0; i < n; i++) scanf("%d", &a[i]);
9
10     int sum = 0;
11     for (i = 0; i < n; i++) sum += a[i];
12     printf("%d\n", sum);
13     return 0;
14 }
15
```

переменный размер

При объявлении массива можно инициализировать его элементы с помощью *массивового литерала*, имеющего вид

`{ значение1, значение2, ..., значениеN }`

При этом размер массива в деклараторе «`[]`» можно не указывать – компилятор сам сосчитает значения, перечисленные в литерале. Например:

```
int arr[] = { 10, 20, 30, 40, 50 };
```

Если же указать размер массива, но перечислить значения не всех элементов, то неперечисленные элементы получат нулевые значения. Поэтому самый простой способ объявления массива, забитого нулями, выглядит как

```
int arr[10] = { 0 };
```

Массивовые литералы в некоторых случаях можно использовать не только при объявлении массива. Однако, эти применения не будут рассматриваться в нашем курсе.

Многомерные массивовые литералы

Базовые сведения

Операции

Операторы

Деклараторы

Массивы

Указатели

Сложные объявления

Типовые литералы

Функции

Valgrind

Псевдонимы

Динамическая память

Строки

Структуры, объединения и перечисления

Препроцессор

В случае многомерных массивов используются вложенные массивовые литералы. Например:

```
int matrix[2][3] = { { 1, 2, 3 }, { 4, 5, 6 } };
int matrix3d[2][3][4] = {
    { {1,2,3,4}, {1,3,5,7}, {4,5,7,8} },
    { {8,3,5,2}, {2,5,6,4}, {8,5,3,5} }
};
```

Отметим, что компилятор языка C не умеет «считать» в литерале количество элементов вложенных массивов, поэтому размер можно не указывать только в самом левом деклараторе «[]»:

```
int matrix[][3] = { { 1, 2, 3 }, { 4, 5, 6 } };
```

Нетрудно догадаться, что объявление многомерного массива, у которого все элементы равны нулю, записывается как

```
int matrix[2][3] = { { 0 } };
```

Объявление указателя: декларатор *

Базовые сведения

Операции

Операторы

Деклараторы

Массивы

Указатели

Сложные объявления

Типовые литералы

Функции

Valgrind

Псевдонимы

Динамическая память

Строки

Структуры, объединения и перечисления

Препроцессор

***a** декларатор, сообщаящий компилятору, что **a** является указателем.

При объявлении *типизированного* указателя задаётся тип переменной, адрес которой в нём хранится. Например, указатель **p** может содержать адрес **int**'овой переменной:

```
int *p;
```

Для хранения адреса переменной произвольного типа используются *нетипизированные* указатели, в объявлении которых имя типа заменено ключевым словом **void**:

```
void *ptr;
```

Нетипизированный указатель невозможно разыменовывать, потому что хранящийся в нём адрес переменной ничего не говорит ни о размере этой переменной, ни об операциях, которые к ней можно применять.

Операции получения адреса и разыменовывания

Базовые сведения

Операции

Операторы

Деклараторы

Массивы

Указатели

Сложные объявления

Типовые литералы

Функции

Valgrind

Псевдонимы

Динамическая память

Строки

Структуры, объединения и перечисления

Препроцессор

`&a` возвращает адрес операнда;

`*p` разыменовывает указатель `p`.

Операнд операции «`&`» должен быть левым значением, а операнд операции «`*`» – типизированным указателем.

Пример:

```
1  #include <stdio.h>
2
3  int main(int argc, char **argv)
4  {
5      int a = 10, b = 20;
6      int *pa = &a, *pb = &b;
7      *pa *= *pb;
8      printf("%d\n", a);
9      return 0;
10 }
```

200

Указатели высших порядков

Базовые сведения

Операции

Операторы

Деклараторы

Массивы

Указатели

Сложные объявления

Типовые литералы

Функции

Valgrind

Псевдонимы

Динамическая память

Строки

Структуры, объединения и перечисления

Препроцессор

Так как указатель – это переменная, то у него тоже есть адрес. Соответственно, можно объявлять указатель на указатель, а также указатели более высоких порядков.

Указатели высших порядков получаются путём многократного применения декларатора «*» к имени переменной.

Пример:

```
1  #include <stdio.h>
2
3  int main(int argc, char **argv)
4  {
5      int a = 10;
6      int *pa = &a, **ppa = &pa, ***pppa = &ppa;
7      printf("%d, %d, %d\n", *pa, **ppa, ***pppa);
8      return 0;
9  }
```

10, 10, 10

Введём следующие обозначения:

- T – некоторый тип;
- p и q – указатели, типизированные типом T и содержащие адреса p и q , соответственно;
- i – целое число.

Операции сложения и вычитания применимы к указателям, но при этом отмасштабированы по `sizeof(T)`:

$p++$ и $++p$ увеличивает указатель на `sizeof(T)`;

$p--$ и $--p$ уменьшает указатель на `sizeof(T)`;

$p + i$ возвращает адрес $p + i \cdot \text{sizeof}(T)$;

$p - i$ возвращает адрес $p - i \cdot \text{sizeof}(T)$;

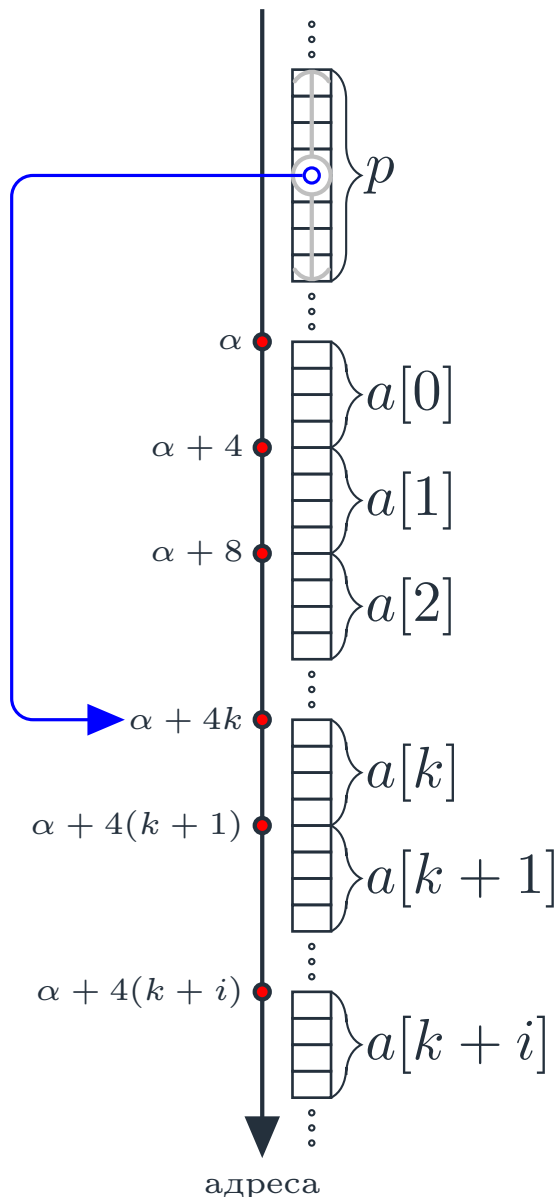
$q - p$ возвращает число $(q - p) / \text{sizeof}(T)$.

Естественно, операции инкремента и декремента возвращают значения согласно принятой для них схеме.

Отметим, что операции сложения двух указателей и вычитания указателя из числа не разрешены.

Смысл прибавления числа к указателю

Базовые сведения
Операции
Операторы
Деклараторы
Массивы
Указатели
Сложные объявления
Типовые литералы
Функции
Valgrind
Псевдонимы
Динамическая память
Строки
Структуры, объединения и перечисления
Препроцессор



Смысл арифметики указателей становится понятен в контексте работы с массивами.

Предположим, что указатель p , типизированный типом `int`, содержит адрес k -того элемента массива a :

$$p = \alpha + 4k.$$

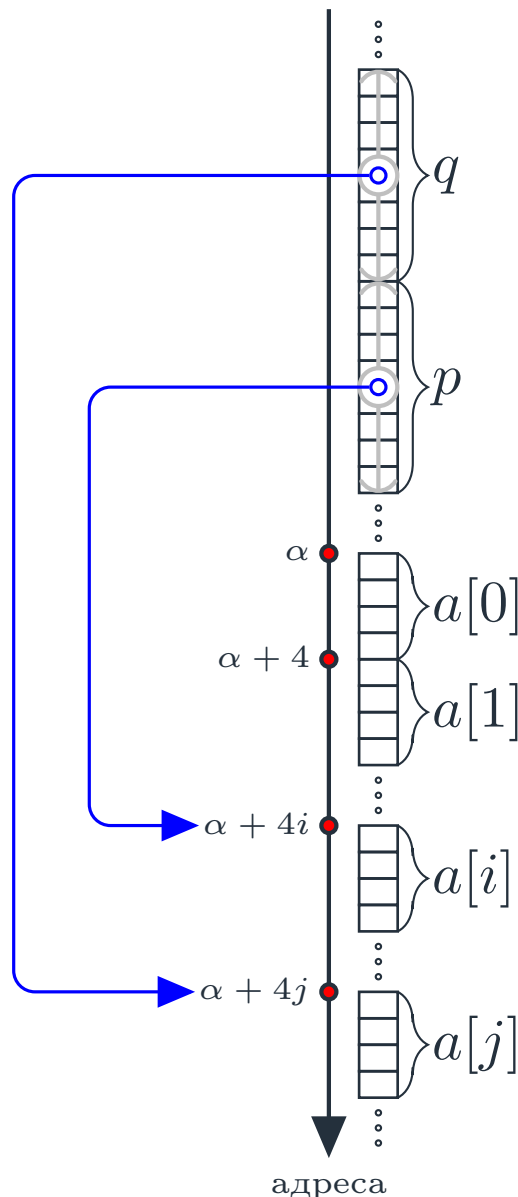
Тогда прибавление к указателю единицы даст адрес $(k + 1)$ -го элемента массива:

$$p + 1 = \alpha + 4k + 4 = \alpha + 4(k + 1).$$

Аналогично, прибавив к p число i , мы получим адрес $(k + i)$ -го элемента массива.

Смысл разности указателей

Базовые сведения
Операции
Операторы
Деклараторы
Массивы
Указатели
Сложные объявления
Типовые литералы
Функции
Valgrind
Псевдонимы
Динамическая память
Строки
Структуры, объединения и перечисления
Препроцессор



Теперь пусть даны два указателя p и q , типизированные типом `int` и содержащие адреса i -того и j -того элементов массива a , соответственно:

$$p = \alpha + 4i,$$

$$q = \alpha + 4j.$$

Тогда разность $q - p$ — это «расстояние» между j -тым и i -тым элементами массива, измеренное в элементах:

$$q - p = \frac{(\alpha + 4j) - (\alpha + 4i)}{4} = j - i.$$

Естественно, разность указателей может получиться отрицательной.

Неявное преобразование массива к указателю

Базовые сведения

Операции

Операторы

Деклараторы

Массивы

Указатели

Сложные
объявления

Типовые лите-
ралы

Функции

Valgrind

Псевдонимы

Динамическая
память

Строки

Структуры,
объединения и
перечисления

Препроцессор

Исключительно важный аспект языка C: если в некотором выражении фигурирует массив, и этот массив не является операндом операций «&» и «sizeof», то он неявно преобразуется к указателю на нулевой элемент массива. Следовательно, имя массива `a` в выражении означает тот же самый адрес, что и `&a[0]` и `&a`.

В качестве примера приведём программу суммирования элементов массива, не использующую операцию индексации:

```
1  #include <stdio.h>
2
3  int main(int argc, char **argv)
4  {
5      int a[10], *p, sum;
6      for (p = a; p - a < 10; p++) scanf("%d", p);
7
8      for (p = a, sum = 0; p - a < 10; p++) sum += *p;
9      printf("%d\n", sum);
10     return 0;
11 }
```


Индексация указателей

Базовые сведения
Операции
Операторы
Деклараторы
Массивы
Указатели
Сложные объявления
Типовые литералы
Функции
Valgrind
Псевдонимы
Динамическая память
Строки
Структуры, объединения и перечисления
Препроцессор

На самом деле операндом операции «[]» является не массив, а типизированный указатель, и запись `a[i]` является лишь более наглядной формой выражения `*(a + i)`.
Пример (переворачивание массива нечётного размера):

```
1  #include <stdio.h>
2
3  int main(int argc, char **argv)
4  {
5      int a[15], i, *p;
6      for (i = 0; i < 15; i++) scanf("%d", a + i);
7
8      for (p = a + 7, i = 1; i < 8; i++) {
9          int t = p[i];
10         p[i] = p[-i];
11         p[-i] = t;
12     }
13
14     for (i = 0; i < 15; i++) printf("%d_", a[i]);
15     return 0;
16 }
```

Приоритеты деклараторов

Базовые сведения

Операции

Операторы

Деклараторы

Массивы

Указатели

Сложные объявления

Типовые литералы

Функции

Valgrind

Псевдонимы

Динамическая память

Строки

Структуры, объединения и перечисления

Препроцессор

К имени переменной можно применять несколько деклараторов:

```
int m[5][4]; /* матрица 5x4 */
float **p; /* указатель на указатель на float */
```

Как и в случае обычных операций, при использовании деклараторов возникают спорные случаи вида

тип $decl_1$ а $decl_2$;

В языке C постфиксные деклараторы имеют больший приоритет, чем префиксные. Например, объявление

```
int *a[10];
```

делает переменную **a** массивом указателей на **int**.

Изменить порядок применения деклараторов позволяют круглые скобки. Например, чтобы **a** стала указателем на массив **int**'ов размера 10, нужно записать

```
int (*a)[10];
```

Пояснение смысла указателя на массив

Базовые сведения

Операции

Операторы

Деклараторы

Массивы

Указатели

Сложные
объявления

Типовые литералы

Функции

Valgrind

Псевдонимы

Динамическая
память

Строки

Структуры,
объединения и
перечисления

Преппроцессор

Разберёмся, зачем может понадобиться рассмотренный нами на предыдущем слайде указатель на массив.

Рассмотрим фрагмент программы:

```
int arr[20];  
int *p = arr;  
int (*q)[20] = &arr;
```

Если распечатать значения `p` и `q`, можно убедиться, что они содержат одинаковые адреса, совпадающие с адресом нулевого элемента массива `arr`.

Однако, арифметические операции с указателями `p` и `q` будут работать по-разному. Действительно, `++p` увеличит адрес, хранящийся в `p`, на 4, тогда как `++q` приведёт к увеличению `q` на 80 (размер массива из 20 `int`'ов).

Эта особенность указателя на массив позволяет использовать его для путешествия по строкам матрицы.

Пример: сумма строк матрицы

Базовые сведения	1	<code>#include <stdio.h></code>
	2	
Операции	3	<code>int main(int argc, char **argv)</code>
Операторы	4	<code>{</code>
Деклараторы	5	<code>int m[5][4], i, j;</code>
Массивы	6	<code>for (i = 0; i < 5; i++) {</code>
Указатели	7	<code>for (j = 0; j < 4; j++) {</code>
Сложные объявления	8	<code>scanf("%d", &m[i][j]);</code>
Типовые литералы	9	<code>}</code>
Функции	10	<code>}</code>
Valgrind	11	
Псевдонимы	12	<code>int (*p)[4], v[4] = { 0 };</code>
Динамическая память	13	<code>for (p = &m[0]; p - &m[0] < 5; p++) {</code>
	14	<code>for (j = 0; j < 4; j++) {</code>
	15	<code>v[j] += (*p)[j];</code>
Строки	16	<code>}</code>
Структуры, объединения и перечисления	17	<code>}</code>
	18	
Препроцессор	19	<code>for (j = 0; j < 4; j++) printf("%d_", v[j]);</code>
	20	<code>return 0;</code>
	21	<code>}</code>

Запись типового литерала

Базовые сведения

Операции

Операторы

Деклараторы

Массивы

Указатели

Сложные
объявления

Типовые литералы

Функции

Valgrind

Псевдонимы

Динамическая
память

Строки

Структуры,
объединения и
перечисления

Препроцессор

Типовый литерал – это изображение типа данных в исходном тексте программы.

Типовые литералы используются в операции приведения типа, в операции `sizeof` и в некоторых других конструкциях языка C.

Типовой литерал для нужного нам типа можно получить из объявления переменной этого типа путём удаления имени переменной. Например, пусть нам нужен типовый литерал для массива из 10 указателей на `int`-овые массивы размера 16. Сначала запишем объявление переменной этого типа (без инициализации и точки с запятой):

```
int (*a[10])[16]
```

Удалив из объявления имя переменной, получим требуемый типовый литерал:

```
int (*[10])[16]
```

Приведение типов указателей

Базовые сведения

Операции

Операторы

Деклараторы

Массивы

Указатели

Сложные
объявления

Типовые литералы

Функции

Valgrind

Псевдонимы

Динамическая
память

Строки

Структуры,
объединения и
перечисления

Препроцессор

В языке C можно присваивать значения `void`-указателей типизированным указателям, и наоборот:

```
int a = 10;
void *p = &a;    /* Неявное приведение. */
int *q = p;      /* Неявное приведение. */
```

Однако хорошим тоном считается использование операции приведения:

```
int a = 10;
void *p = (void*)&a;
int *q = (int*)p;
```

Компилятор выдаёт предупреждение при попытке присвоить значение указателя, типизированного одним типом, указателю, типизированному другим типом. Например,

```
int x, *p = &x;
float *q = p;    /* warning: initialization from
                  incompatible pointer type */
```

Объявление функции: декларатор ()

Базовые сведения

Операции

Операторы

Деклараторы

Массивы

Указатели

Сложные объявления

Типовые литералы

Функции

Valgrind

Псевдонимы

Динамическая память

Строки

Структуры, объединения и перечисления

Препроцессор

f (параметры) декларатор, сообщающий компилятору, что **f** является функцией с указанным списком формальных параметров.

Объявления формальных параметров перечисляются внутри декларатора «()» через запятую. При этом объявление каждого параметра записывается как объявление неинициализированной переменной. Например,

```
... max(int x, int y) ...
```

Функция типизируется типом возвращаемого значения. Например, функция **sin** возвращает значение типа **double**:

```
double sin(double x);
```

Если функция ничего не возвращает, вместо типа указывается ключевое слово **void**. Например,

```
void minmax(int *arr, long size, int *min, int *max);
```

Функция не может возвращать массив.

Прототипы и определения функций

Базовые сведения

Операции

Операторы

Деклараторы

Массивы

Указатели

Сложные
объявления

Типовые литералы

Функции

Valgrind

Псевдонимы

Динамическая
память

Строки

Структуры,
объединения и
перечисления

Препроцессор

Тело функции – это оператор-блок, определяющий алгоритм работы функции.

Объявление функции, не содержащее тела, называется *прототипом*. Более полное объявление функции, содержащее тело, называется *определением* функции.

Список формальных параметров прототипа может не содержать имён параметров, т.е. состоять из типовых литералов. Например,

```
void minmax(int*, long, int*, int*);
```

Объявление прототипа функции даёт возможность вызывать функции, определения которых расположены в других файлах или в той части текущего файла, которую компилятор ещё не обработал.

Главным отличием определения функции от прототипа является наличие оператора-блока после декларатора «()»:

ЗАГОЛОВОК **ТЕЛО**
тип имя (параметры) блок

Пример: определение функции minmax

Базовые сведения

Операции

Операторы

Деклараторы

Массивы

Указатели

Сложные объявления

Типовые литералы

Функции

Valgrind

Псевдонимы

Динамическая память

Строки

Структуры, объединения и перечисления

Препроцессор

Функция `minmax` находит минимальное и максимальное целое число в массиве `arr` размера `size`:

```
1 void minmax(int *arr, long size, int *min, int *max)
2 {
3     int i;
4     *min = *max = arr[0];
5     for (i = 1; i < size; i++) {
6         if (arr[i] < *min) *min = arr[i];
7         else if (arr[i] > *max) *max = arr[i];
8     }
9 }
```

Этой функции нужно возвращать сразу два значения, поэтому она получает в качестве параметров указатели `min` и `max`. Функция записывает минимальное число по адресу, содержащемуся в `min`, а максимальное число – по адресу, содержащемуся в `max`.

Операция вызова функции

Базовые сведения

Операции

Операторы

Деклараторы

Массивы

Указатели

Сложные объявления

Типовые литералы

Функции

Valgrind

Псевдонимы

Динамическая память

Строки

Структуры, объединения и перечисления

Препроцессор

f (параметры) вызывает функцию **f**, передавая ей указанные фактические параметры, возвращает значение функции.

Выражения, значения которых передаются функции в качестве фактических параметров, записываются внутри круглых скобок через запятую. Например:

```
min(a[i] + a[j], 2*x)
```

Значения фактических параметров присваиваются формальным параметрам функции непосредственно перед передачей управления её телу.

Количество и типы фактических параметров должны соответствовать количеству и типам формальных параметров. Несовпадение типов фактического и формального параметров допускается в том случае, если возможно осуществить неявное преобразование значения фактического параметра к типу формального параметра.

Выполнение функции завершается либо после выполнения последнего оператора её тела, либо с помощью оператора завершения функции. В любом случае управление передаётся в точку программы, где функция была вызвана.

Если функция не возвращает значения, оператор завершения функции имеет форму

```
return ;
```

Если функция возвращает значение, оно указывается в операторе:

```
return значение ;
```

В теле функции может быть несколько операторов `return`.

Примеры:

```
1  int min(int x, int y)
2  {
3      if (x < y) return x;
4      return y;
5  }
```

```
1  int min(int x, int y)
2  {
3      return x < y ? x : y;
4  }
```

Пример: сумма элементов массива

Базовые сведения	1	<code>#include <stdio.h></code>
	2	
Операции	3	<code>int sum(int *arr, long size)</code>
Операторы	4	<code>{</code>
Деклараторы	5	<code>int i, s = 0;</code>
Массивы	6	<code>for (i = 0; i < size; i++) s += arr[i];</code>
Указатели	7	<code>return s;</code>
Сложные объявления	8	<code>}</code>
Типовые литералы	9	
Функции	10	<code>int main(int argc, char **argv)</code>
Valgrind	11	<code>{</code>
Псевдонимы	12	<code>long n, i;</code>
Динамическая память	13	<code>scanf("%ld", &n);</code>
	14	
Строки	15	<code>int a[n];</code>
Структуры, объединения и перечисления	16	<code>for (i = 0; i < n; i++) scanf("%d", &a[i]);</code>
	17	
	18	<code>printf("%d\n", <u>sum(a, n)</u>);</code>
Препроцессор	19	<code>return 0;</code> вызов функции
	20	<code>}</code>

Применение прототипов функций

Базовые сведения

Операции

Операторы

Деклараторы

Массивы

Указатели

Сложные
объявления

Типовые литералы

Функции

Valgrind

Псевдонимы

Динамическая
память

Строки

Структуры,
объединения и
перечисления

Препроцессор

Одной из ситуаций, в которых необходимо использовать прототипы, является определение взаиморекурсивных функций. Например, пусть функция `f` вызывает функцию `g`, которая, в свою очередь, вызывает функцию `f`. Если поместить определение функции `f` до определения функции `g`, то из тела функции `f` будет невозможно вызвать функцию `g`. Проблема решается путём объявления прототипа функции `g` до определения функции `f`:

```
тип g(параметры); ← прототип g
```

```
тип f(параметры) ← определение f
{
    ... g(...); ...
}
```

```
тип g(параметры) ← определение g
{
    ... f(...); ...
}
```

В языке C разрешено использовать указатели на функции. Указатель на функцию содержит адрес, с которого начинается тело функции, представленное в машинном коде. Для объявления указателя на функцию используется декларатор «*». Например:

```
/* f -- указатель на функцию, принимающую и  
    возвращающую int */  
int (*f)(int x);  
/* a -- массив из 10 указателей на функции,  
    принимающие в качестве параметра float и  
    возвращающие указатель на char */  
char *(*a[10])(float x);
```

Имя функции обозначает её адрес. Его можно присвоить указателю, если типы формальных параметров и возвращаемого значения у функции и указателя совпадают. Применение операции «()» к указателю на функцию означает вызов функции, адрес которой хранится в указателе.

Пример: указатели на функции

Базовые сведения	1	<code>#include <stdio.h></code>
Операции	2	
Операторы	3	<code>int for_each(int *a, int n, int (*f)(int prev, int x))</code>
Деклараторы	4	<code>{</code>
Массивы	5	<code>int i, prev;</code>
Указатели	6	<code>for (i = 0, prev = 0; i < n; i++)</code>
Сложные объявления	7	<code>prev = f(prev, a[i]);</code>
Типовые литералы	8	<code>return prev;</code>
Функции	9	<code>}</code>
Valgrind	10	
Псевдонимы	11	<code>int sum_positive(int p, int x) {</code>
Динамическая память	12	<code>return x > 0 ? p+x : p;</code>
Строки	13	<code>}</code>
Структуры, объединения и перечисления	14	<code>int max(int p, int x) { return p > x ? p : x; }</code>
Препроцессор	15	
	16	<code>int main(int argc, char **argv)</code>
	17	<code>{</code>
	18	<code>int A[10] = { -1, 4, 8, -5, 3, 10, 0, 7, 1 };</code>
	19	<code>printf("%d□", for_each(A, 10, sum_positive));</code>
	20	<code>printf("%d\n", for_each(A, 10, max));</code>
	21	<code>return 0;</code>
	22	<code>}</code>

Стандартная библиотека языка C содержит функцию `qsort`, сортирующую любые массивы по возрастанию:

```
void qsort(void *base, size_t nel, size_t width,  
           int (*compare)(const void *a, const void *b));
```

Параметры функции `qsort`:

- `base` – адрес начала массива;
- `nel` – размер массива;
- `width` – размер одного элемента массива в байтах;
- `compare` – указатель на функцию сравнения двух элементов (возвращает 0 в случае $a = b$, отрицательное значение в случае $a < b$ и положительное значение в случае $a > b$).

Для работы с этой функцией нужно подключить заголовочный файл `stdlib.h`:

```
#include <stdlib.h>
```


Пример: сортировка строк матрицы

Базовые сведения
Операции
Операторы
Деклараторы
Массивы
Указатели
Сложные объявления
Типовые литералы
Функции
Valgrind
Псевдонимы
Динамическая память
Строки
Структуры, объединения и перечисления
Препроцессор

Напишем программу сортировки строк матрицы по возрастанию сумм их элементов.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int sum(int *a, int n)
5  {
6      int i, s;
7      for (i = 0, s = 0; i < n; i++)
8          s += a[i];
9      return s;
10 }
11
12 int comp_rows(const void *a, const void *b)
13 {
14     return sum((int*)a, 4) - sum((int*)b, 4);
15 }
16
17 ...
```

Пример: сортировка строк матрицы (продолжение)

Базовые сведения	15	...
Операции	16	
Операторы	17	<code>int main(int argc, char **argv)</code>
Деклараторы	18	<code>{</code>
Массивы	19	<code>int i, j, M[4][4] = {</code>
Указатели	20	<code>{ 1, 2, 3, 4 },</code>
Сложные объявления	21	<code>{ 1, -2, 3, -4 },</code>
Типовые литералы	22	<code>{ -1, 2, -3, 4 },</code>
Функции	23	<code>{ 1, 2, -3, -4 },</code>
Valgrind	24	<code>};</code>
Псевдонимы	25	
Динамическая память	26	<code>qsort(M, 4, sizeof(int[4]), comp_rows);</code>
Строки	27	
Структуры, объединения и перечисления	28	<code>for (i = 0; i < 4; i++) {</code>
Препроцессор	29	<code>for (j = 0; j < 4; j++)</code>
	30	<code>printf("%d_", M[i][j]);</code>
	31	<code>printf("\n");</code>
	32	<code>}</code>
	33	<code>return 0;</code>
	34	<code>}</code>

Базовые
сведения

Операции

Операторы

Деклараторы

Массивы

Указатели

Сложные
объявления

Типовые лите-
ралы

Функции

Valgrind

Псевдонимы

Динамическая
память

Строки

Структуры,
объединения и
перечисления

Препроцессор

Программирование на языке С связано с трудоёмким устранением ошибок, связанных с неправильным применением указателей, выходами за границы массива и использованием неинициализированных переменных.

Отладчик valgrind – это интерпретатор машинного кода, ключевой особенностью которого является аннотирование виртуального адресного пространства интерпретируемой программы, а также содержимого отдельных ячеек памяти:

- valgrind хранит информацию, инициализирован или нет каждый бит любой ячейки памяти;
- для каждого адреса динамической памяти (области «heap») valgrind в любой момент времени знает, принадлежит ли он динамической переменной, и более того, помнит, в каком месте кода эта динамическая переменная была создана.

Запуск отладчика valgrind

Базовые сведения

Операции

Операторы

Деклараторы

Массивы

Указатели

Сложные
объявления

Типовые литералы

Функции

Valgrind

Псевдонимы

Динамическая
память

Строки

Структуры,
объединения и
перечисления

Преппроцессор

Пусть требуется отладить программу test.c. Чтобы в сообщениях отладчика valgrind были указаны номера строк программы test.c, при её компиляции должна быть указана опция «-g»:

```
gcc -g -o test test.c
```

Запуск исполняемого файла test под управлением отладчика valgrind осуществляется командой

```
valgrind -q ./test
```

Мы указываем опцию «-q» при вызове valgrind, чтобы он выводил только самую важную информацию об ошибках в программе test.

Если ни одной ошибки не обнаружено, valgrind в этом режиме вообще ничего не выводит.

Пример: использование неинициализированной переменной в условии оператора if

```
1  #include <stdio.h>
2
3  int main(int argc, char **argv)
4  {
5      int z;
6      if (z) printf("z != 0\n");
7      return 0;
8  }
```

Ошибка, обнаруженная valgrind'ом:

```
Conditional jump or move depends on uninitialised
value(s)
   at 0x40054F: main (test.c:6)
```

Обратите внимание на то, что valgrind не покажет ошибку, если изменить тело функции main следующим образом:

```
3      int z;
4      z |= 1; /* устанавливаем в 1 младший бит z */
5      if (z & 1) printf("z != 0\n");
```

Пример: разыменование неинициализированного указателя

Базовые
сведения

Операции

Операторы

Деклараторы

Массивы

Указатели

Сложные
объявления

Типовые лите-
ралы

Функции

Valgrind

Псевдонимы

Динамическая
память

Строки

Структуры,
объединения и
перечисления

Препроцессор

```
1  #include <stdio.h>
2
3  int main(int argc, char **argv)
4  {
5      int *p;
6      *p = 10;
7      return 0;
8  }
```

Ошибки, обнаруженные valgrind'ом:

```
Use of uninitialised value of size 8
   at 0x4004F4: main (test.c:6)
```

```
Invalid write of size 4
```

```
   at 0x4004F4: main (test.c:6)
```

```
Address 0x0 is not stack'd, malloc'd or (recently)
free'd
```

То есть, во-первых, указатель `p` не инициализирован, а во-вторых, мы обращаемся к памяти по адресу `0x0`, который в нём «случайно» оказался записан.

Пример: неправильный вызов scanf

```
1  #include <stdio.h>
2
3  int main(int argc, char **argv)
4  {
5      int x;
6      scanf("%d", x); /* надо передавать адрес x */
7      printf("%d\n", x);
8      return 0;
9  }
```

Ошибки, обнаруженные valgrind'ом:

Use of uninitialised value of size 8

...

by 0x4005CE: main (test.c:6)

Invalid write of size 4

...

by 0x4005CE: main (test.c:6)

Address 0x0 is not stack'd, malloc'd or (recently) free'd

Псевдонимы типовых литералов

Базовые сведения

Операции

Операторы

Деклараторы

Массивы

Указатели

Сложные объявления

Типовые литералы

Функции

Valgrind

Псевдонимы

Динамическая память

Строки

Структуры, объединения и перечисления

Препроцессор

Для упрощения объявлений в языке C предусмотрена синтаксическая конструкция для объявления псевдонимов типовых литералов:

```
typedef объявление;
```

Например,

```
typedef unsigned long size_t;
typedef int int_array[10];
typedef int (*compare_t)(const void*, const void*);
```

Идентификатор, фигурирующий в таком объявлении, становится псевдонимом соответствующего типового литерала и может быть затем использован в других объявлениях. Например,

```
size_t dimensions[3];
int_array a;
void qsort(void*, size_t, size_t, compare_t);
```


Выделение динамической памяти

Базовые сведения

Операции

Операторы

Деклараторы

Массивы

Указатели

Сложные объявления

Типовые литералы

Функции

Valgrind

Псевдонимы

Динамическая память

Строки

Структуры, объединения и перечисления

Препроцессор

Функция `malloc` создаёт динамическую переменную:

```
void *malloc(size_t size);
```

Эта функция выделяет в области «heap» непрерывный участок размера `size` байтов и возвращает адрес начала этого участка. Если участок запрошенного размера не может быть выделен, `malloc` возвращает нулевой адрес.

Чтобы воспользоваться функцией `malloc`, необходимо в начале программы подключить заголовочный файл `stdlib.h`:

```
#include <stdlib.h>
```

Пример выделения памяти для массива из 10 `int`'ов:

```
int *a = (int*)malloc(10 * sizeof(int));
if (a == NULL) {
    printf("Не хватает памяти\n");
    return -1;
}
```

`NULL` – это обозначение нулевого адреса.

Освобождение динамической памяти

Базовые сведения

Операции

Операторы

Деклараторы

Массивы

Указатели

Сложные объявления

Типовые литералы

Функции

Valgrind

Псевдонимы

Динамическая память

Строки

Структуры, объединения и перечисления

Препроцессор

Когда динамическая переменная становится больше не нужна, занимаемый ею участок памяти нужно освободить с помощью вызова функции `free`:

```
void free(void *ptr);
```

Функция `free` получает в качестве параметра адрес переменной и помечает участок памяти, занимаемый этой переменной, как свободный. Этот участок затем может быть выделен для хранения другой переменной.

Например,

```
char *s = (char*)malloc(1024 * sizeof(char));  
... /* Использование динамической переменной */ ...  
free(s);
```

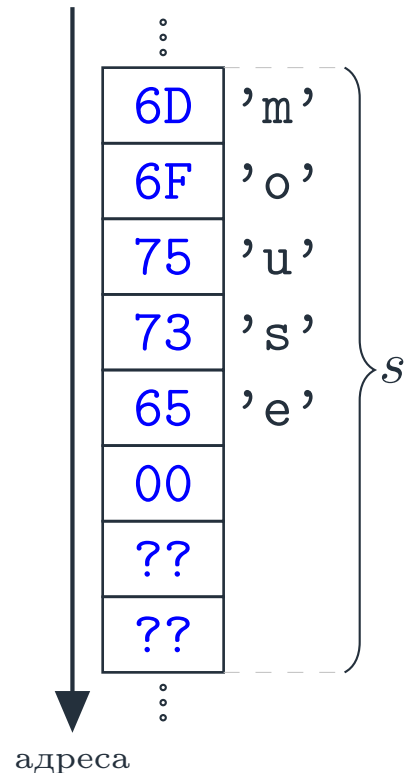
Отметим, что «забывание» адреса динамической переменной приводит к тому, что эту переменную уже невозможно будет уничтожить. Это явление считается грубой ошибкой программирования и называется *утечкой памяти*.

Пример: многомерные динамические массивы

Базовые сведения	1	<code>#include <stdio.h></code>
	2	<code>#include <stdlib.h></code>
Операции	3	
Операторы	4	<code>int main(int argc, char **argv)</code>
Деклараторы	5	<code>{</code>
Массивы	6	<code> const int M = 5, N = 4;</code>
Указатели	7	<code> int **a, i, j;</code>
Сложные объявления	8	
Типовые литералы	9	<code> a = (int**)malloc(M*sizeof(int*));</code>
Функции	10	<code> if (a == NULL) return -1;</code>
Valgrind	11	<code> for (i = 0; i < M; i++) {</code>
Псевдонимы	12	<code> a[i] = (int*)malloc(N*sizeof(int));</code>
Динамическая память	13	<code> if (a[i] == NULL) return -1;</code>
	14	<code> }</code>
Строки	15	
	16	<code> a[3][2] = 666;</code>
Структуры, объединения и перечисления	17	
	18	<code> for (i = 0; i < M; i++) free(a[i]);</code>
Препроцессор	19	<code> free(a);</code>
	20	
	21	<code> return 0;</code>
	22	<code>}</code>

Представление строк в памяти

Базовые сведения
Операции
Операторы
Деклараторы
Строки
ASCIIZ-строки
gets
strlen
strcmp
strncpy
strncat
atoi
argc, argv
Структуры, объединения и перечисления
Препроцессор



Строка в языке C представляет собой массив `char`'ов, элементы которого содержат ASCII-коды символов, причём конец строки обозначается нулевым кодом (так называемые *ASCIIZ-строки*).

Размер массива, отводимого для хранения строки, должен превышать длину строки хотя бы на единицу (место для нулевого элемента).

Допускается применять строковый литерал для задания начального значения при объявлении строки, т.е., например, следующие два объявления эквивалентны:

```
char s1[8] = { 'm', 'o', 'u', 's', 'e', 0 };  
char s2[8] = "mouse";
```

Ввод строковых данных

Базовые
сведения

Операции

Операторы

Деклараторы

Строки

ASCIIZ-строки

gets

strlen

strcmp

strncpy

strncat

atoi

argc, argv

Структуры,
объединения и
перечисления

Препроцессор

С помощью функции `scanf` можно осуществлять ввод строк, не содержащих пробелов.

Если предполагается, что текст в стандартном потоке ввода может содержать пробелы, используется функция

```
char *gets(char *s);
```

Здесь `s` – указатель на массив `char`'ов, в который будет помещен введенный текст. Отметим, что если размер массива окажется недостаточен, произойдет выход за его границы.

Функция `gets` считывает текст из стандартного потока ввода до первого символа `'\n'`. Она возвращает указатель `s`.

Когда `gets` вызывается после `scanf`, имеет смысл «попросить» `scanf` считать из потока ввода все пробельные символы. Например,

```
int i;  
scanf("%d_", &i);  
char s[100];  
gets(s);
```

Смысл строковых литералов

Базовые сведения

Операции

Операторы

Деклараторы

Строки

ASCIIZ-строки

gets

strlen

strcmp

strncpy

strncat

atoi

argc, argv

Структуры, объединения и перечисления

Препроцессор

Строковый литерал в выражении означает указатель на нулевой элемент массива `char`'ов, содержащего записанную в литерале строку. Этот массив размещается компилятором в защищённом от записи участке области «`data`».

Пример:

```
1  #include <stdio.h>
2
3  int main()
4  {
5      char *s = "mouse";
6      printf("%p, %p\n", "mouse", s, "mouse"[0]);
7      return 0;
8  }
```

Вывод:

```
0x40063c , 0x40063c , m
```

Обратите внимание на то, что одинаковые строковые литералы указывают на один и тот же массив.

Вычисление длины строки

Базовые
сведения

Операции

Операторы

Деклараторы

Строки

ASCIIZ-строки

gets

strlen

strcmp

strncmp

strncat

atoi

argc, argv

Структуры,
объединения и
перечисления

Преппроцессор

Заголовочный файл `string.h` содержит прототипы библиотечных функций для работы со строками. Мы рассмотрим некоторые из этих функций и приведём их возможные реализации на языке C.

Для вычисления длины строки служит функция

```
size_t strlen(const char *s, size_t n);
```

Здесь `s` – указатель на начало строки, а `n` – размер массива, в котором расположена строка.

Функцию `strlen` можно было бы написать следующим образом:

```
size_t strlen(const char *s, size_t n)
{
    size_t len;
    for (len = 0; len < n && s[len]; len++);
    return len;
}
```

Сравнение строк с помощью операций «==» и «!=» почти не имеет смысла, так как сравниваться будет не содержимое строк, а только указатели. Поэтому для сравнения строк используется функция

```
int strcmp(const char *s1, const char *s2);
```

Она возвращает 0 в случае равенства строк, отрицательное число – если первая строка меньше второй, положительное число – если первая строка больше второй.

Определение функции `strcmp` могло бы выглядеть как

```
int strcmp(const char *s1, const char *s2)
{
    while (*s1 && *s2 && *s1 == *s2) {
        s1++;
        s2++;
    }
    return *s1 - *s2;
}
```


Пример: сортировка массива указателей на строки

Базовые сведения	1	<code>#include <stdio.h></code>
Операции	2	<code>#include <stdlib.h></code>
Операторы	3	<code>#include <string.h></code>
Деклараторы	4	
Строки	5	<code>int compare(const void *a, const void *b)</code>
ASCIIZ-строки	6	<code>{</code>
gets	7	<code>return strcmp(*(char**)a, *(char**)b);</code>
strlen	8	<code>}</code>
strcmp	9	
strncpy	10	<code>int main(int argc, char **argv)</code>
strncat	11	<code>{</code>
atoi	12	<code>char *a[] = { "red", "blue", "green",</code>
argc, argv	13	<code>"yellow", "black" };</code>
Структуры, объединения и перечисления	14	<code>int count = sizeof a/sizeof(char*);</code>
Препроцессор	15	<code>qsort(a, count, sizeof(char*), compare);</code>
	16	
	17	<code>int i;</code>
	18	<code>for (i = 0; i < count; i++) printf("%s\n", a[i]);</code>
	19	<code>return 0;</code>
	20	<code>}</code>

Копирование строк осуществляется функцией

```
char *strncpy(char *dest, const char *src, size_t n);
```

Здесь `dest` – указатель на строку-приёмник, `src` – указатель на строку-источник, `n` – размер массива, отведённого под строку-приёмник.

Функция `strncpy` могла бы быть реализована как

```
char *strncpy(char *dest, const char *src, size_t n)
{
    char *temp = dest;
    while (n-- && (*dest = *src)) {
        dest++;
        src++;
    }
    return temp;
}
```

Как видно из приведённой реализации, функция `strncpy` возвращает указатель на строку-приёмник.

Использование функции strncpy

Особенность работы функции strncpy: если размер строки-приёмника недостаточен, то в результате она остаётся без завершающего нуля. Поэтому грамотное использование этой функции подразумевает принудительную установку нуля:

```
1  #include <stdio.h>
2  #include <string.h>
3
4  int main(int argc, char **argv)
5  {
6      char s1[8], *s2 = "encyclopedia";
7      s1[7] = 0;
8      strncpy(s1, s2, 7);
9      printf("%s\n", s1);
10     return 0;
11 }
```

Вывод:

```
encyclo
```

Базовые
сведения

Операции

Операторы

Деклараторы

Строки

ASCIIZ-строки

gets

strlen

strcmp

strncpy

strncat

atoi

argc, argv

Структуры,
объединения и
перечисления

Препроцессор

Конкатенация строк

Базовые
сведения

Операции

Операторы

Деклараторы

Строки

ASCIIZ-строки

gets

strlen

strcmp

strcpy

strncat

atoi

argc, argv

Структуры,
объединения и
перечисления

Препроцессор

Конкатенация строк осуществляется функцией

```
char *strncat(char *dest, const char *src, size_t n);
```

Здесь `dest` – указатель на строку-приёмник, `src` – указатель на строку-источник, `n` – максимальное количество символов, которое может быть добавлено к строке-приёмнику.

Функция `strncat` дописывает содержимое строки-источника в конец строки-приёмника. Её можно реализовать как

```
char *strncat(char *dest, const char *src, size_t n)
{
    size_t len;
    for (len = 0; dest[len]; len++);
    strcpy(dest + len, src, n);
    return dest;
}
```

Функция `strncat`, как и `strcpy`, возвращает указатель на строку-приёмник.

Перевод строки в число

Базовые
сведения

Операции

Операторы

Деклараторы

Строки

ASCIIZ-строки

gets

strlen

strcmp

strncpy

strncat

atoi

argc, argv

Структуры,
объединения и
перечисления

Препроцессор

Если строка содержит десятичную запись целого числа, то это целое число можно получить с помощью функций

```
int atoi(const char *str);  
long atol(const char *str);
```

Возможная реализация функции atoi:

```
int atoi(const char *s)  
{  
    while (*s == ' ' || *s == '\t' || *s == '\n') s++;  
  
    int sign = 1, res;  
    if (*s == '+') s++;  
    else if (*s == '-') sign = -1, s++;  
  
    for (res = 0; *s >= '0' && *s <= '9'; s++) {  
        res = res*10 + *s - '0';  
    }  
  
    return res * sign;  
}
```

Передача аргументов из командной строки

Базовые
сведения

Операции

Операторы

Деклараторы

Строки

ASCIIZ-строки

gets

strlen

strcmp

strncpy

strncat

atoi

argc, argv

Структуры,
объединения и
перечисления

Преппроцессор

Параметры функции `main`:

- `argc` – количество аргументов командной строки;
- `argv` – указатель на массив указателей на строки, в которых содержатся аргументы командной строки.

Пример (файл `sample.c`):

```
1  int main(int argc, char **argv)
2  {
3      printf("%d %s\n", argc, argv[0]);
4      return 0;
5  }
```

Вывод при запуске `./sample`:

```
1  ./sample
```

Один аргумент передаётся всегда: `argv[0]` – это имя исполняемого файла программы.

Пример: сумма целых чисел

Базовые
сведения

Операции

Операторы

Деклараторы

Строки

ASCIIZ-строки

gets

strlen

strcmp

strncpy

strncat

atoi

argc, argv

Структуры,
объединения и
перечисления

Препроцессор

```
1  #include <stdio.h>
2  #include <string.h>
3
4  int main(int argc, char **argv)
5  {
6      int i, sum;
7      if (argc < 2) {
8          printf("usage: _sum_ x1 _..._ xN\n");
9          return 0;
10     }
11
12     for (i = 1, sum = 0; i < argc; i++) {
13         sum += atoi(argv[i]);
14     }
15
16     printf("%d\n", sum);
17     return 0;
18 }
```

Пример вызова программы sum:

```
./sum 10 20 30 40
```

Структурой называют составной тип данных, значение которого представляет собой последовательность именованных полей разных типов.

Структурные типы могут быть именованными и безымянными. В качестве имён структур выступают их *теги*.

Объявление переменных структурного типа совмещено со связыванием структурного типа с его тегом:

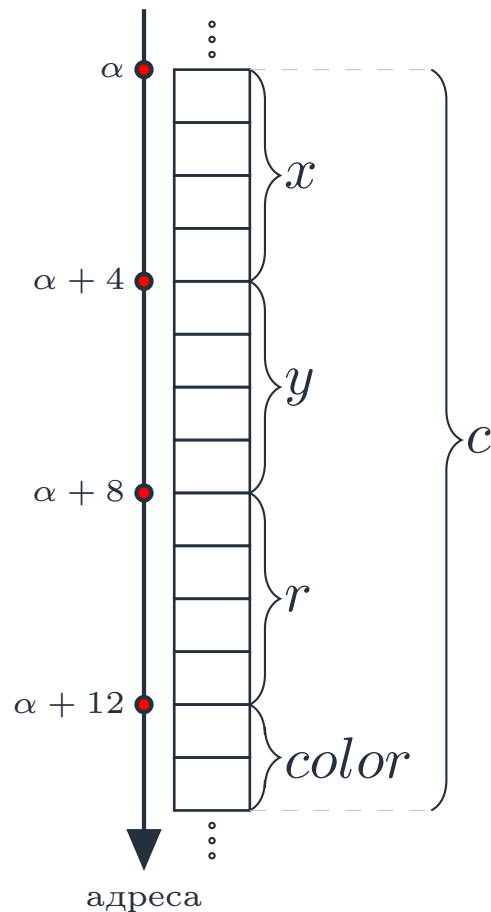
```
struct тег {  
    объявления_полей  
} список_переменных_с_деклараторами;
```

Тем самым, структурный тип в объявлении занимает то место, где мы привыкли помещать базовый тип (`int`, `float`, и т.п.) или `void`.

Синтаксис объявлений полей идентичен объявлениям переменных за тем исключением, что не разрешается указывать начальные значения полей.

Пример: структура, описывающая круг

Базовые сведения
Операции
Операторы
Деклараторы
Строки
Структуры, объединения и перечисления
Структуры
Объединения
Перечисления
Препроцессор



Пусть круг определяется координатами центра, радиусом и цветом. Причём координаты и радиус – вещественные числа, и цвет задаётся целочисленным кодом:

```
struct Circle {  
    float x, y;  
    float radius;  
    short color;  
} c, *pc, ac[100];
```

Одновременно с привязкой структурного типа к тегу `Circle` мы объявили три переменные – `c`, `pc` и `ac`.

Отметим, что либо тег структуры, либо имена переменных в объявлении можно не указывать. В отсутствие тега структура – безымянна. Цель объявления, в котором нет имён переменных, – привязка тега к структурному типу.

Использование тегов структур

Базовые
сведения

Операции

Операторы

Деклараторы

Строки

Структуры,
объединения и
перечисления

Структуры

Объединения

Перечисления

Препроцессор

Коль скоро со структурным типом связан тег, этот тег в сочетании с ключевым словом **struct** является псевдонимом этого структурного типа. Т.е. его, например, можно использовать для объявлений переменных:

```
struct Point {  
    int x, y, z;  
} pt1;
```

```
struct Point pt2, pts[100];
```

Отметим, что теги структурных типов удобно использовать в типовых литералах, например, при создании массива структур в динамической памяти:

```
struct Point *a;  
a = (struct Point*)malloc(sizeof(struct Point) * 10);
```

Операции для работы со структурами

Базовые
сведения

Операции

Операторы

Деклараторы

Строки

Структуры,
объединения и
перечисления

Структуры

Объединения

Перечисления

Препроцессор

a.f возвращает левое значение, соответствующее полю **f** структуры **a**;

p->f возвращает левое значение, соответствующее полю **f** структуры, на которую указывает указатель **p**.

С точки зрения машинного кода, действие обеих рассматриваемых операций заключается в прибавлении к адресу структуры смещения указанного поля относительно начала структуры. При этом операция «->» является всего лишь более удобной записью выражения **(*p).f**.

Пример:

```
struct Point pt;  
p.x = 10;  
p.y = 20;  
p.z = 30;
```

```
struct Point *ppt = &pt;  
printf("(%d, %d, %d)", pt->x, pt->y, pt->z);
```

Структурные литералы

Базовые
сведения

Операции

Операторы

Деклараторы

Строки

Структуры,
объединения и
перечисления

Структуры

Объединения

Перечисления

Преппроцессор

Аналогично массивовым литералам, в языке С существуют структурные литералы, позволяющие инициализировать поля переменных структурных типов при их объявлении. Структурный литерал записывается как

```
{ поле1, поле2, ..., полеN }
```

Например,

```
struct Point pt = { 10, 20, 30 };
```

Структурные литералы могут сочетаться с массивовыми литералами. Например,

```
struct Point pts[] = { {1,2,3}, {4,5,6}, {7,8,9} };
```

```
struct {  
    char *name;  
    int scores[3];  
} ivanov = { "Иванов", {90, 85, 70} },  
  petrov = { "Петров", {83, 90, 98} };
```

Пример: сортировка массива структур

Базовые сведения
Операции
Операторы
Деклараторы
Строки
Структуры, объединения и перечисления
Структуры
Объединения
Перечисления
Препроцессор

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  struct Month { int num; char *name; };
5
6  int compare(const void *a, const void *b)
7  {
8      return ((struct Month*)a)->num -
9             ((struct Month*)b)->num;
10 }
11
12 int main(int argc, char **argv)
13 {
14     struct Month a[] = { {5,"May"}, {2,"February"},
15                          {3,"March"}, {4,"April"}, {1,"January"},
16     };
17     int i, count = sizeof a/sizeof(struct Month);
18     qsort(a, count, sizeof(struct Month), compare);
19     for (i = 0; i < count; i++)
20         printf("%s\n", a[i].name);
21     return 0;
22 }
```

Объединением называют составной тип данных, значение которого представляет собой совокупность занимающих одно и то же место в памяти именованных полей разных типов.

Типы-объединения, как и структурные типы, могут быть именованными и безымянными. В качестве имён структур выступают их теги.

Объявление переменных типа-объединения совмещено со связыванием типа-объединения с его тегом:

```
union тег {  
    объявления_полей  
} список_переменных_с_деклараторами ;
```

Отметим, что либо тег типа-объединения, либо имена переменных в объявлении можно не указывать.

Для обращения к полям объединения используются уже знакомые нам операции «.» и «->».

Пример: IP-адрес

Базовые сведения

Операции

Операторы

Деклараторы

Строки

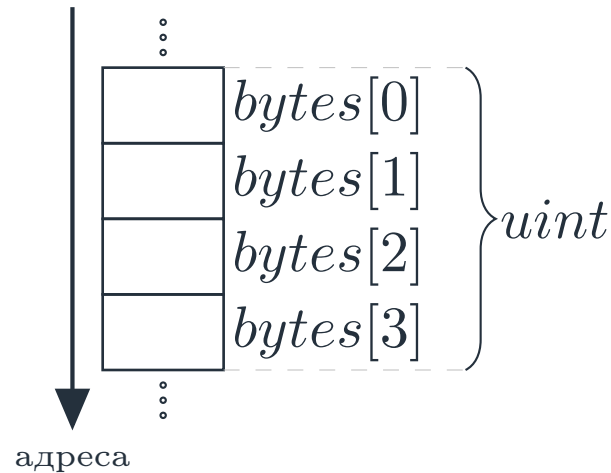
Структуры, объединения и перечисления

Структуры

Объединения

Перечисления

Препроцессор



IP-адрес можно представлять либо как четыре целых числа в диапазоне от 0 до 255, либо как беззнаковое 32-разрядное целое число:

```
union IP {  
    unsigned int uint;  
    unsigned char bytes[4];  
};
```

Для инициализации переменной-объединения применяется литерал вида

```
{ поле1 }
```

В этом литерале можно указывать только значение первого поля. Остальные поля получают значения автоматически, т.к. они совмещены в памяти с первым полем. Например,

```
union IP addr = { 1000000000 };
```

Пример: вывод байтов образа числа

Базовые
сведения

Операции

Операторы

Деклараторы

Строки

Структуры,
объединения и
перечисления

Структуры

Объединения

Перечисления

Препроцессор

```
1  #include <stdio.h>
2
3  union Double {
4      double x;
5      unsigned char bytes[sizeof(double)];
6  };
7
8  int main(int argc, char **argv)
9  {
10     union Double v;
11     scanf("%lf", &v.x);
12
13     int i;
14     for (i = 0; i < sizeof(double); i++) {
15         printf("%02X□", v.bytes[i]);
16     }
17     return 0;
18 }
```


Объявление перечислений

Базовые
сведения

Операции

Операторы

Деклараторы

Строки

Структуры,
объединения и
перечисления

Структуры

Объединения

Перечисления

Препроцессор

Перечислением называют целочисленный тип данных, значения которого задаются именованными константами. Объявление переменных типа-перечисления совмещено со связыванием типа-перечисления с его тегом:

```
enum тег {  
    объявления_констант  
} список_переменных_с_деклараторами;
```

Пример:

```
enum Month {  
    JANUARY = 1,    FEBRUARY = 2,    MARCH = 3,  
    APRIL = 4,      MAY = 5,        JUNE = 6,  
    JULY = 7,       AUGUST = 8,       SEPTEMBER = 9,  
    OCTOBER = 10,   NOVEMBER = 11,   DECEMBER = 12  
} m1, m2, *months;
```

Отметим, что либо тег типа-перечисления, либо имена переменных в объявлении можно не указывать.

Директивы препроцессора

Базовые
сведения

Операции

Операторы

Деклараторы

Строки

Структуры,
объединения и
перечисления

Препроцессор

Директивы

include

define

ifdef

Препроцессор – это программа, выполняющая предварительную обработку программ, написанных на языке С. Работой препроцессора можно управлять с помощью директив препроцессора.

Каждая директива располагается в отдельной строке программы и начинается со знака «#».

```
#include <stdio.h>
#ifndef PI
#define PI 3.14159265359
#endif
#define SUM(a,b) ((a)+(b))
```

Директива `#include`

Базовые
сведения

Операции

Операторы

Деклараторы

Строки

Структуры,
объединения и
перечисления

Препроцессор

Директивы

`include`

`define`

`ifdef`

Директива `#include` вызывает вставку содержимого текстового файла.

```
#include <stdio.h>
#include "file.txt"
```

Угловые скобки говорят о том, что файл нужно искать в «системных» каталогах, а кавычки – о том, что сначала нужно посмотреть в текущий каталог.

Пример: использование директивы #include

Базовые
сведения

Операции

Операторы

Деклараторы

Строки

Структуры,
объединения и
перечисления

Препроцессор

Директивы

include

define

ifdef

Файл point.h:

```
1 struct Point { int x, y; };
```

Файл main.c:

```
1 #include "point.h"
2
3 int main(int argc, char **argv)
4 {
5     struct Point p;
6     return 0;
7 }
```

Препроцессор выдаст компилятору текст программы:

```
struct Point { int x, y; };

int main(int argc, char **argv)
{
    struct Point p;
    return 0;
}
```

Простые макроподстановки

Базовые
сведения

Операции

Операторы

Деклараторы

Строки

Структуры,
объединения и
перечисления

Препроцессор

Директивы

include

define

ifdef

Простые макроподстановки определяются директивой

```
#define имя_макроподстановки значение
```

Например,

```
#define N 100
#define MAIN int main(int argc, char **argv)
#define BEGIN {
#define END }
```

Препроцессор заменяет все вхождения имени макроподстановки в файле на её значение.

Пример: использование директивы #define

Базовые сведения
Операции
Операторы
Деклараторы
Строки
Структуры, объединения и перечисления
Препроцессор
Директивы
include
define
ifdef

```
1  #define MAX 100
2  #define LOOP for (i = 0; i < MAX; i++)
3
4  int main(int argc, char **argv)
5  {
6      int i;
7      LOOP printf("%d_", i);
8      return 0;
9  }
```

Препроцессор выдаст компилятору текст программы:

```
int main(int argc, char **argv)
{
    int i;
    for (i = 0; i < 100; i++) printf("%d_", i);
    return 0;
}
```

Макроподстановки с параметрами

Базовые
сведения

Операции

Операторы

Деклараторы

Строки

Структуры,
объединения и
перечисления

Препроцессор

Директивы

include

define

ifdef

Макроподстановки с параметрами определяются директивой

```
#define имя(парам1, ..., парамN) значение
```

Например,

```
#define PRINT_INT(x) printf("%d", x)
#define MAX(a,b) ((a) > (b) ? (a) : (b))
#define FOR(i,a,b) for (i = a; i < b; i++)
```

Использование макроподстановок с параметрами похоже на вызов функций:

```
int main(int argc, char **argv)
{
    PRINT_INT(MAX(10,20));
    return 0;
}
```

Пример: макроподстановки с параметрами

Базовые сведения	1	<code>#define INPUT_MATRIX(a, m, n, spec) { \</code>
Операции	2	<code>int i, j; \</code>
Операторы	3	<code>for (i = 0; i < (m); i++) { \</code>
Деклараторы	4	<code>for (j = 0; j < (n); j++) { \</code>
Строки	5	<code>scanf(spec, &(a)[i][j]); \</code>
Структуры, объединения и перечисления	6	<code>} \</code>
Препроцессор	7	<code>} \</code>
Директивы	8	<code>}</code>
include	9	
define	10	<code>int main(int argc, char **argv)</code>
ifdef	11	<code>{</code>
	12	<code>int A[4][5];</code>
	13	<code>INPUT_MATRIX(A, 4, 5, "%d");</code>
	14	<code>return 0;</code>
	15	<code>}</code>

Пример: макроподстановки с параметрами (продолжение)

Базовые
сведения

Операции

Операторы

Деклараторы

Строки

Структуры,
объединения и
перечисления

Препроцессор

Директивы

include

define

ifdef

Препроцессор выдаст компилятору текст программы:

```
int main(int argc, char **argv)
{
    int A[4][5];
    {
        int i, j;
        for (i = 0; i < (4); i++) {
            for (j = 0; j < (5); j++) {
                scanf("%d", &(A)[i][j]);
            }
        };
    }
    return 0;
}
```

Круглые скобки в макроподстановках

Базовые сведения

Операции

Операторы

Деклараторы

Строки

Структуры, объединения и перечисления

Преппроцессор

Директивы

include

define

ifdef

При записи значения макроопределения с параметрами целесообразно обрамлять круглыми скобками как само значение, так и вхождения параметров.

Например, раскрытие макроподстановок в программе

```
1  #define SUM(a,b) a+b
2  #define MUL(a,b) a*b
3  int x = SUM(1,2)*5;
4  int y = MUL(5, 1+2);
```

приведёт к тому, что на вход компилятора будет подано:

```
int x = 1+2*5;
int y = 5*1+2;
```

Правильно было бы:

```
#define SUM(a,b) ((a)+(b))
#define MUL(a,b) ((a)*(b))
```

Директивы условной компиляции

Базовые
сведения

Операции

Операторы

Деклараторы

Строки

Структуры,
объединения и
перечисления

Препроцессор

Директивы

include

define

ifdef

Директивы условной компиляции управляют тем, будет или не будет включён в программу некоторый фрагмент текста:

```
#ifdef имя_макроопределения
#ifdef имя_макроопределения
#else
#endif
```

Например,

```
#ifdef DEBUG
void *my_malloc(size_t n)
{
    printf("Allocating %d bytes\n", n);
    return malloc(n);
}
#else
#define my_malloc malloc
#endif
```

Заголовочные файлы и директивы условной компиляции

Базовые сведения

Операции

Операторы

Деклараторы

Строки

Структуры, объединения и перечисления

Препроцессор

Директивы

include

define

ifdef

Одно из полезных применений условной компиляции – приём, гарантирующий, что некоторый заголовочный файл не может быть включён директивой `#include` более одного раза.

Например, рассмотрим заголовочный файл `point.h`:

```
1  #ifndef POINT_H_INCLUDED
2  #define POINT_H_INCLUDED
3
4  struct Point {
5      int x, y;
6  };
7
8  #endif
```