**Программирование. Вопросы к экзамену**

**Содержание**

[**Вопрос 1.** Язык программирования C история. 2](#_Toc74752505)

[**Вопрос 2.** Сравнить си и питон 3](#_Toc74752506)

[**Вопрос 3.** Синтаксис и семантика 3](#_Toc74752507)

[**Вопрос 4.** Переменные и простейшие операции 7](#_Toc74752508)

[**Вопрос 5.** printf 8](#_Toc74752509)

[**Вопрос 6.** scanf 9](#_Toc74752510)

[**Вопрос 7.** Операции и выражения 10](#_Toc74752511)

[**Вопрос 8.** Операции присваивания, побочный эффект 12](#_Toc74752512)

[**Вопрос 9.** Логические операции, побитовые 15](#_Toc74752513)

[**Вопрос 10.** Оператор-выражение составной оператор и оператор вывода 18](#_Toc74752514)

[**Вопрос 11.** While, do-while, break, continue 22](#_Toc74752515)

[**Вопрос 12.** for, while, пустой оператор (;) 25](#_Toc74752516)

[**Вопрос 13.** for, do-while 29](#_Toc74752517)

[**Вопрос 14.** Тип данных, простые типы 31](#_Toc74752518)

[**Вопрос 15.** Явное и неявное преобразование типов 37](#_Toc74752519)

[**Вопрос 16.** Подпрограмма 39](#_Toc74752520)

[**Вопрос 17.** Описание функций на языке Си 41](#_Toc74752521)

[**Вопрос 18.** Функция передачи параметров 45](#_Toc74752522)

[**Вопрос 19.** Рекурсии 46](#_Toc74752523)

[**Вопрос 20.** Одномерный статический массив 48](#_Toc74752524)

[**Вопрос 21.** Указатели, объявления … 50](#_Toc74752525)

[**Вопрос 22.** Адресная арифметика 51](#_Toc74752526)

[**Вопрос 23.** Многомерный статический массив 52](#_Toc74752527)

[**Вопрос 24.** Многомерный массив особенности использования 58](#_Toc74752528)

[**Вопрос 25.** Строка 63](#_Toc74752529)

[**Вопрос 26.** Строка, функции стандартной библиотеки 66](#_Toc74752530)

[**Вопрос 27.** Строки, способы хранения слов, функция strtok 74](#_Toc74752531)

[**Вопрос 28.** Структура 80](#_Toc74752532)

[**Вопрос 29.** Объединения, перечисляемый тип 84](#_Toc74752533)

[**Вопрос 30.** Текстовые файлы 88](#_Toc74752534)

[**Вопрос 31.** Двоичные файлы 93](#_Toc74752535)

[**Вопрос 32.** Регулярные структуры над двоичным файлом 97](#_Toc74752536)

[**Вопрос 33.** Этапы получения исполняемого файла 104](#_Toc74752537)

[**Вопрос 34.** Многофайловая организация проекта 106](#_Toc74752538)

# Вопрос 1. Язык программирования C история.

Пройтись по стандартам, отличия, области применения, основные характеристики

**Ответ**

***Стандарты***

Язык программирования Си разрабатывался в период с 1969 по 1973 годы в лабораториях Bell Labs. Название языка стало логическим продолжением старого языка "Би", многие особенности которого были положены в основу.

В 1978 году Брайан Керниган и Деннис Ритчи опубликовали первую редакцию книги "Язык программирования Си". Эта книга, известная среди программистов как "K&R", служила многие годы неформальной спецификацией языка. Версию языка Си, описанную в ней, часто называют "K&R C".

По мере развития язык сначала стандартизировали как ANSI C (также C89), а затем этот стандарт был принят комитетом по международной стандартизации ISO как ISO C (также C90). В стандарте С99 язык получил новые возможности, такие как новые типы данных, массивы переменной длины и встраиваемые функции, а в стандарте C11 в язык добавили улучшенную поддержку Юникода, реализацию потоков и поддержку атомарных типов. Однако с тех пор язык развивается медленно, и в стандарт С18 попали лишь исправления ошибок стандарта C11.

Язык Си разрабатывался как язык системного программирования, для которого можно создать однопроходный компилятор. Стандартная библиотека также невелика. Как следствие данных факторов - компиляторы разрабатываются сравнительно легко. Поэтому данный язык доступен на самых различных платформах. К тому же, несмотря на свою низкоуровневую природу, язык ориентирован на переносимость. Программы, соответствующие стандарту языка, могут компилироваться под различные архитектуры компьютеров.

Целью языка было облегчение написания больших программ с минимизацией ошибок по сравнению с ассемблером, следуя принципам процедурного программирования, но избегая всего, что может привести к дополнительным накладным расходам, специфичным для языков высокого уровня.

***Отличия***

Основные концепции языка:

* сравнительно низкий уровень абстракции;
* простота реализации компилятора;
* ответственность за действия программиста лежит на программисте.

***Области применения***

На сегодняшний день Си используется в таких областях, как:

* разработка операционных, встраиваемых систем;
* разработка высокопроизводительных приложений;
* компиляторы, интерпретаторы;
* ПО с открытым исходным кодом.

***Основные характеристики***

Из данных утверждений вытекают следующие особенности Си:

* простая языковая база, из которой в стандартную библиотеку вынесены многие существенные возможности, вроде математических функций или функций работы с файлами;
* ориентация на процедурное программирование;
* статическая система типов, предохраняющая от бессмысленных операций;
* использование препроцессора для абстрагирования однотипных операций;
* доступ к памяти через использование указателей;
* небольшое число ключевых слов;
* передача параметров в функцию по значению, а не по ссылке (передача по ссылке эмулируется с помощью указателей);
* наличие указателей на функции и статические переменные;
* области видимости имён;
* структуры и объединения - определяемые пользователем собирательные типы данных, которыми можно манипулировать как одним целым.

В то же время в Си отсутствуют:

* вложенные функции;
* прямое возвращение нескольких значений из функций;
* сопрограммы;
* средства автоматического управления памятью;
* встроенные средства объектно-ориентированного программирования;
* средства функционального программирования.

# Вопрос 2. Сравнить си и питон

Си - компилируемый, питон - интерпретируемый, недостатки каждого подхода, типизация, сильные слабые стороны, Си-ручное управление ресурсами

Статическая и динамическая типизация

Интерпретируемый, компилируемый

**Ответ**

***Отличия языка C от языка Python***

Python - интерпретируемый язык, Си - компилируемый. Как следствие, программы на Python более переносимые, но исполняются гораздо медленнее.

Python - язык с динамической типизацией, Си - со статической. Динамически типизированный код быстрее писать, но сложнее отлаживать и сопровождать, так как проверка корректности типов происходит во время исполнения. Корректность типов в статически типизированном коде происходит до исполнения, что позволяет выявлять многие ошибки и, в случае компиляции, генерировать более оптимальный машинный код.

За управление памятью в Python отвечает сборщик мусора. В Си же ответственность за это лежит на программисте. Такой подход усложняет написание кода, но позволяет его оптимизировать.

# Вопрос 3. Синтаксис и семантика

Определения, способы описания синтаксиса и семантики, определение алфавита, лексемы, ключевого имени, зарезервированного

Правило выделения лексем

**Ответ**

***Алфавит, лексемы***

Всякий язык программирования имеет три основные составляющие: алфавит, синтаксис и семантику.

*Алфавит* - набор символов, допустимых для записи программ.

Алфавит языка Си состоит из

* латинских букв (A-Z, a-z);
* цифр (0-9);
* 29 графических символов (! " # % & ' ( ) \* + , - и т.д.);
* пробельных символов.

Из символов алфавита образуются лексемы.

*Лексема* - это минимальная смысловая единица языка.

К лексемам в языке Си относятся

* идентификаторы;
* ключевые слова;
* константы;
* строковые литералы;
* пунктуаторы.

Правило выделения лексем в языке Си:

Каждый очередной непробельный символ должен добавляться к считываемой лексеме до тех пор, пока она остаётся корректной.

Таким образом, например, можно точно сказать, что выражение a+++b тождественно a++ + b.

*Идентификатор* - строка символов, используемая для идентификации некоторой сущности в программе.

В языке Си идентификатор может содержать буквы, цифры и символ подчеркивания. Идентификатор не может начинаться с цифры. Регистр символов имеет значение.

| **Правильные идентификаторы** | **Неправильные идентификаторы** |
| --- | --- |
| lowercase | 123\_digits\_first |
| UPPERCASE | kebab-case |
| snake\_case | crazy~invalid~characters |
| lowerCamelCase |  |
| UpperCamelCase |  |
| snake\_with\_123\_digits |  |
| lowerCamelWith123Digits |  |
| UpperCamelWith123Digits |  |
| \_prefixed\_snake |  |
| \_prefixedLowerCamel |  |
| \_123dEfInItElY\_MYCaSe |  |

*Ключевое слово* - это идентификатор, смысл которого зафиксирован правилами языка программирования и который используется для распознавания предложений в программе.

Ключевые слова языка Си:

* auto
* break
* case
* char
* const
* continue
* default
* do
* double
* else
* enum
* extern
* float
* for
* goto
* if
* inline
* int
* long
* register
* restrict
* return
* short
* signed
* sizeof
* static
* struct
* switch
* typedef
* union
* unsigned
* void
* volatile
* while
* \_Bool
* \_Complex
* \_Imaginary

*Зарезервированное слово* - идентификатор, который синтаксис языка запрещает к использованию в качестве идентификатора, определяемого программистом.

Все ключевые слова в языке Си являются зарезервированными идентификаторами.

*Предопределённый идентификатор* - идентификатор, определённый в стандартной библиотеке и потому, как и зарезервированное слово, запрещённый к использованию в качестве идентификатора, определяемого программистом.

Такими идентификаторами являются константы ***EOF*** и ***NULL***, типы ***size\_t*** и ***ptrdiff\_t***, файловые переменные ***stdin***, ***stdout***, ***stderr*** и прочие сущности стандартной библиотеки.

*Константа* - величина, значение которой остаётся неизменным во время выполнения программы.

Типы констант:

* целые - 42, -273, 0xFF, 1L;
* вещественные - 3.14, -2.718281828459045, 1e-8, 12.345f;
* символьные - 'A', 'b', '\n';
* строковые - "Hello world!\n".

*Пунктуатор* - символ, который имеет независимое синтаксическое и семантическое значение.

В зависимости от контекста пунктуаторы определяют выполняемые операции.

Пунктуатор : один из

[ ] ( ) { } . ->

++ -- & \* + - ~ !

/ % << >> < > <= >= == != ^ | && ||

? : ; ... = \*= /= %= > += -= <<= >>= &=

^= |= , # ## <: :> <% %> %: %:%:

*Синтаксис* - набор правил, описывающий комбинации символов алфавита, считающиеся правильно структурированной программой (документом) или её фрагментом.

Способы описания синтаксиса:

* формальные грамматики;
* формы Бэкуса-Наура и их расширения;
* диаграммы Вирта.

*Семантика* - это правила придания смысла синтаксически корректным программам. В конечном счёте эти правила определяют ту последовательность действий вычислительной машины, которую она должна выполнить, работая по данной программе.

# Вопрос 4. Переменные и простейшие операции

Что такое переменная, атрибуты, простые операции с переменными, побочный эффект, арифметические операции

Операция присваивания

См. методичку

**Ответ**

*Переменная* - абстракция ячейки памяти компьютера или совокупности таких ячеек.

Каждую переменную характеризует следующий набор атрибутов:

* идентификатор;
* адрес;
* тип;
* значение.

*Идентификатор* - строка символов, используемая для идентификации переменной в программе.

*Адрес* - это первая ячейка памяти, с которой связана данная переменная.

*Тип* определяет набор значений, которые может принимать переменная, и набор допустимых над ней операций.

*Значение* - это содержимое ячеек, связанных с переменной.

Прежде чем к переменной можно будет обращаться в программе, она должна быть связана с типом данных.

*Определение переменной* - это оператор программы, перечисляющий имена переменных и устанавливающий, что они имеют определенный тип.

Например, мы можем определить переменные *width* и *speed* следующим образом:

int width;

float speed;

Первое определение говорит, что *width* - переменная типа *int*, то есть может хранить целые числа. Второе определение говорит, что *speed* - переменная типа *float*, то есть может хранить числа с дробной частью.

Если несколько переменных имеют один и тот же тип, их определения можно объединять:

int width, length;

float speed, acceleration;

Переменные в программе на языке Си не инициализируются по умолчанию нулями. После определения переменной в ней хранится "мусор" - случайное значение, которое осталось в той области памяти, которая была выделена под переменную.

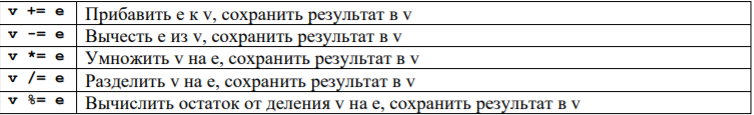
Некоторые операции имеют ***побочный эффект***. При выполнении этих операций, кроме основного эффекта - вычисления значения - происходят изменения объектов или файлов. Таковы, например, операции присваивания (***Операция присваивания*** – это правоассоциативная операция, которая используется для присвоения значения переменной). В операции присваивания «старое» значение переменной часто используется для вычисления «нового» значения этой же переменной. Значением выражения a = b будет значение переменной b и в качестве побочного эффекта это значение будет присвоено переменной a.

Рассмотрим алгоритм работы данной операции более подробно:

* Вычислить l-значение первого операнда (v) выражения.
* Вычислить r-значение второго операнда (e) выражения.
* Присвоить вычисленное r-значение вычисленному l-значению объекта данных;
* Возвратить вычисленное r-значение как результат выполнения операции.

Если v и e имеют разный тип, значение e преобразуется к типу v.!

Говоря о присваивании, существует составное присваивание. Они обладают теми же свойствами, что и обычное присваивание, например, правоассоциативностью. Т.е i+=j+=k это тоже самое, что и i+=(j+=k)



# Вопрос 5. printf

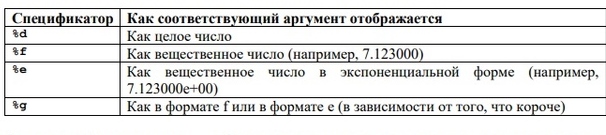
\*назначение, роль строки формата, спецификатора, основные ошибки

Примеры работы

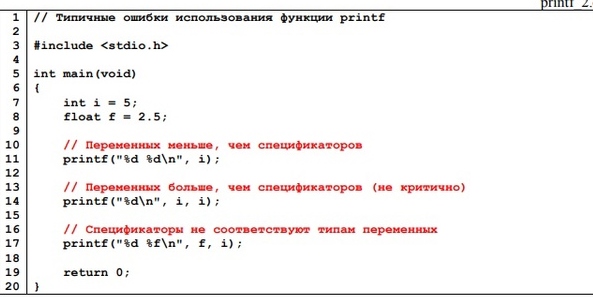
**Ответ**

Функция printf предназначена для отображения содержимого строки, которая называется строкой форматирования. В примере выше это строка "My favorite numbers are %d and %f\n". Символы этой строки выводятся на экран компьютера точно так, как они записаны в строке, за исключением двух случаев. Первый случай связана с заменой так называемых спецификаторов (в примере выше это %d и %f) на значения, указанные в качестве других аргументов при вызове функции printf (в примере выше это переменные i и f). Второй случай связан с заменой так называемых esc-последовательностей (в примере выше это \n), которые используются для вывода непечатных или «трудно» отображаемых символов.

Спецификаторы начинаются с символа “%”. В строке форматирования спецификатор обозначает место, в которое будет выведено соответствующее значение во время отображения строки. Информация, которая указывает за символом “%”, показывает каким образом конвертируется значение из внутреннего представления в отображение.



Основные ошибки при использовании функции printf связаны либо с несоответствием количества спецификаторов и количества переменных, либо с несоответствием спецификатора и типа переменной. Рассмотрим следующий пример.



# Вопрос 6. scanf

спецификаторы, основные ошибки, возвращаемое значение и прочее

Примеры работы

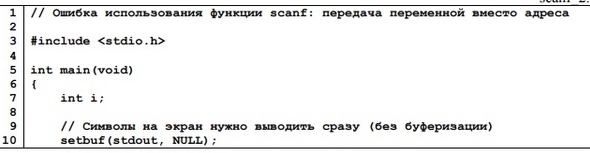
**Ответ**

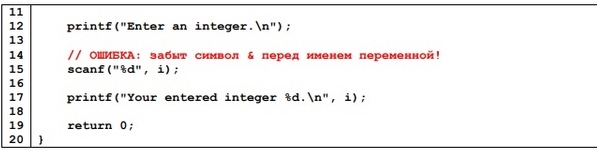
Работа функции scanf (как и функции printf) управляется строкой форматирования. В примере выше это строка "%d%f". Строка форматирования этой функции может содержать как обычные символы, так и спецификаторы. В большинстве случаев строка форматирования содержит только спецификаторы. Это самый надежный способ использования функции scanf.

В отличие от функции printf функции scanf нужно изменить значение переменной (поместить в нее значение, введенное пользователем). Поэтому функция scanf в качестве аргумента должна получать адрес переменной. Для получения адреса переменной перед ее именем необходимо указать символ “&”, который в языке обозначает операцию взятия адреса.

После вызова функция scanf начинает анализировать строку форматирования слева направо. Для каждого спецификатора scanf пытается выделить данные соответствующего типа во входных данных, пропуская пробельные символы. scanf читает выделенное значение, останавливаясь на символе, который не относится к очередному вводимому значению. Если значение было успешно прочитано, scanf продолжает обрабатывать оставшуюся часть строки. Если же значение не соответствует указанному типу, scanf немедленно прекращает работу без анализа оставшейся части строки.

При использовании функции scanf, как правило, допускаются те же самые ошибки, что и при использовании функции printf. Новая разновидность ошибок, специфичная именно для функции scanf, связана с тем, что аргументами функции scanf являются адреса переменных, а не сами переменные.





# Вопрос 7. Операции и выражения

\*куча определений, классификация операций в си, ассоциативность, приоритет

Что такое операции (особенно вычисления) классификация операций, приоритет, сами операции приводить не нужно, побочный эффект

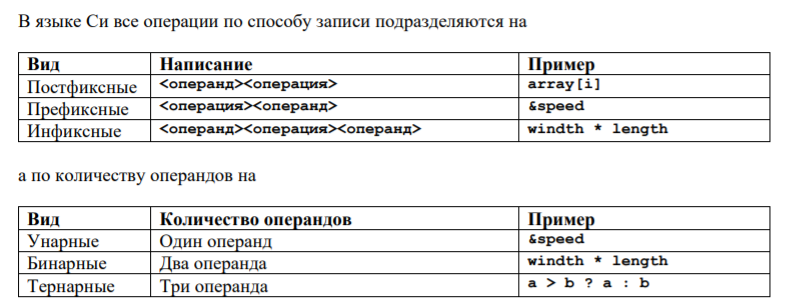
**Ответ**

***Операция*** - конструкция в языках программирования, аналогичная по записи математическим операциям, то есть специальный способ записи некоторых действий. Элементы данных, к которым применяется операция, называют операндами.

Некоторые операции имеют ***побочный эффект***. При выполнении этих операций, кроме основного эффекта - вычисления значения - происходят изменения объектов или файлов. Таковы, например, операции присваивания. Значением выражения a = b будет значение переменной b и в качестве побочного эффекта это значение будет присвоено переменной a.

По способу записи все операции подразделяются на классы:

* постфиксные: 〈операнд〉 〈операция〉 (например, X++)
* префиксные: 〈операция〉 〈операнд〉 (например, ++X)
* инфиксные: 〈операнд〉 〈операция〉 〈операнд〉 (например, X + Y)



***Приоритет операций*** — очерёдность выполнения операций в выражении, при условии, что в выражении нет явного указания порядка следования выполнения операций (с помощью круглых скобок).

Величина, определяющая преимущественное право на выполнение той или иной операции, называется приоритетом. Порядок выполнения операций может регулироваться с помощью круглых скобок.

***Выражение*** – простейшее средство описания действий. Состоит из констант, переменных, а также знаков операций и скобок.

***Ассоциативность*** - свойство операций, позволяющее восстановить последовательность их выполнения при отсутствии явных указаний на очерёдность при равном приоритете; при этом различается левая ассоциативность, при которой вычисление выражения происходит слева направо, и правая ассоциативность — справа налево. Соответствующие операторы называют левоассоциативными и правоассоциативными.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Лексемы | Операция | Класс | Приоритет | Ассоциативность |
| имена, литералы | простые лексемы | первичный | 16 | нет |
| a[k] | индексы | постфиксный | 16 | слева направо |
| f(…) | вызов функции | постфиксный | 16 | слева направо |
| . | прямой выбор | постфиксный | 16 | слева направо |
| -> | опосредованный выбор | постфиксный | 16 | слева направо |
| ++ -- | положительное и отрицательное приращение | постфиксный | 16 | слева направо |
| ( имя типа ) {init} | составной литерал (C99) | постфиксный | 16 | слева направо |
| ++ -- | положительное и отрицательное приращение | префиксный | 15 | справа налево |
| sizeof | размер | унарный | 15 | справа налево |
| ~ | побитовое НЕ | унарный | 15 | справа налево |
| ! | логическое НЕ | унарный | 15 | справа налево |
| - + | изменение знака, плюс | унарный | 15 | справа налево |
| & | адрес | унарный | 15 | справа налево |
| \* | опосредование (разыменование) | унарный | 15 | справа налево |
| ( имя типа ) | приведение типа | унарный | 14 | справа налево |
| \* / % | мультипликативные операции | бинарный | 13 | слева направо |
| + - | аддитивные операции | бинарный | 12 | слева направо |
| << >> | сдвиг влево и вправо | бинарный | 11 | слева направо |
| < > <= >= | отношения | бинарный | 10 | слева направо |
| == != | равенство/неравенство | бинарный | 9 | слева направо |
| & | побитовое И | бинарный | 8 | слева направо |
| ^ | побитовое исключающее ИЛИ | бинарный | 7 | слева направо |
| | | побитовое ИЛИ | бинарный | 6 | слева направо |
| && | логическое И | бинарный | 5 | слева направо |
| || | логическое ИЛИ | бинарный | 4 | слева направо |
| ? : | условие | тернарный | 3 | справа налево |
| = += -= \*= /= %= <<= >>= &= ^= |= | присваивание | бинарный | 2 | справа налево |
| , | последовательная оценка | бинарный | 1 | слева на право |

# Вопрос 8. Операции присваивания, побочный эффект

Обычное и составное присваивание, арифметика, операции сравнения, побочный эффект (на что влияет)

Арифметическое присваивание, декремент, инкремент

Неопределенное поведение

**Ответ**

***Обычное присваивание***

После вычисления значения выражения, как правило, необходимо сохранить результат в переменной. Для этих целей в языке Си используется операция присваивания.

v = e

Формально алгоритм работы этой операции можно описать следующим образом:

1. Вычислить left-значение первого операнда (v) выражения.
2. Вычислить right-значение второго операнда (e) выражения.
3. Присвоить вычисленное right-значение вычисленному left-значению объекта данных;
4. Возвратить вычисленное right-значение как результат выполнения операции.

Если v и e имеют разный тип, значение e преобразуется к типу v (неявное преобразование).

int a;

double b = 3.5;

a = b;

(a = 3)

В языке Си широко используются обе возможности операции присваивания: и как механизма изменения значения объекта данных (через его left-значение), и как функции, возвращающей значение (через right-значение объекта данных). Поэтому оператор присваивания можно использовать в выражениях.

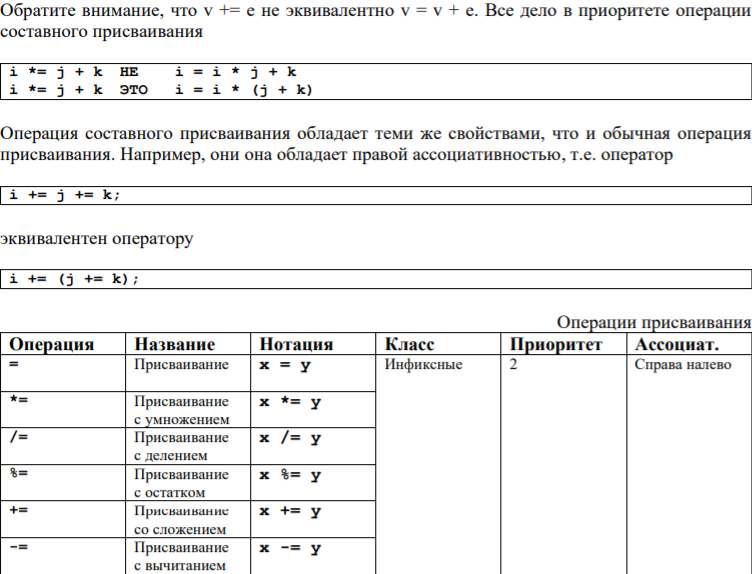
while(rc = scanf(…))

или

i = j = k = 0

***Cоставное присваивание***

Составное присваивание (compound assignment) В операции присваивания *«старое»* значение переменной часто используется для вычисления «нового» значения этой же переменной:

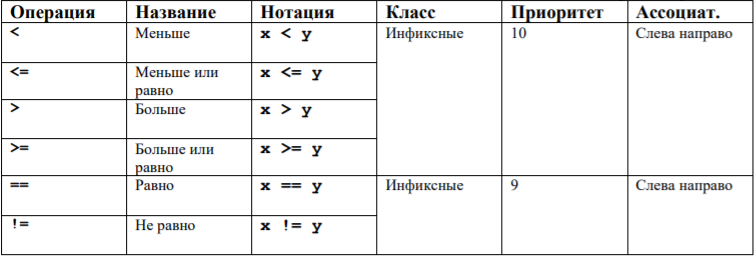
Составное присваивание (compound assignment) В операции присваивания «старое» значение переменной часто используется для вычисления «нового» значения этой же переменной: 

Инкремент/декремент

Часто в программе можно встретить операцию «инкремента» (т.е. увеличение на единицу) и «декремента» (т.е. уменьшение на единицу) переменной.

В языке Си существуют операции, которые позволяют еще более сократить запись. Это операции инкремента “++” и декремента “--”.

Их можно использовать в постфиксной и префиксной форме. Постфиксная (i++) форма сначала возвращает значение переменной, а затем производится арифметическая операция. При записи в префиксной форме (++i) происходит тоже самое, но она сначала совершает арифметическую операцию, а после возвращает значение.

**Операции сравнения**

Операции отношения эквивалентны соответствующим математическим операциям за одним исключением. В выражении они возвращают целое значение 0, когда операция ложна, или целое значение 1, в противном случае.

Как и арифметические операции операции отношения могут использоваться для сравнения целых и вещественных чисел (вещественные лучше не сравнивать через ==, нужно посчитать их дельту и проверить меньше ли она определённого eps, если меньше, то можно считать их равными), причем тип операндов может перемешиваться.

Приоритет операций отношения меньше приоритета арифметических операций, поэтому выражение i + j < k – 1 ==== (i + j) < (k - 1)

А выражение i < j < k вполне допустимо в языке Си, но имеет значение отличного от аналогичного математического выражения. Поскольку операции отношения левоассоциативны, это выражение эквивалентно следующему (i < j) < k

Побочный эффект

Побочный эффект выражается в неявном изменении значения переменной в процессе вычисления выражения

Присваивание, декремент, инкремент имеют побочный эффект.

оператор присваивания имеет побочный эффект, который проявляется в изменении значения переменной х. Оператор ++ имеет побочный эффект инкремента переменной х. Вывод х имеет побочный эффект внесения изменений в консольное окно.

Неопределённое поведение

Оно может проявиться в этом коде

int i = 5;

i = ++i + ++i;

Неопределенность в языках C и C++ связана с тем, что согласно стандартам С и С++ побочные эффекты (то есть инкремент в данном случае) могут быть применены в любой удобный для компилятора момент между двумя [точками следования](https://ru.wikipedia.org/wiki/%D0%A2%D0%BE%D1%87%D0%BA%D0%B0_%D1%81%D0%BB%D0%B5%D0%B4%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F).

Точка следования – момент времени, когда «пыль улеглась», и все встреченные побочные эффекты гарантированно завершены и остались позади.

# Вопрос 9. Логические операции, побитовые

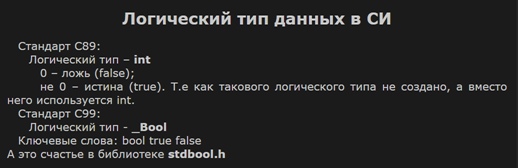
\*сравнение, логические, особенности логических операций, побитовые, сравнить побитовые и логические

Побитовые операции, особенности логического типа, про Bool можно не рассказывать

Про сравнения, про схемы вычисления логический операций, побитовые операции, сравнить логические и побитовые

**Ответ**

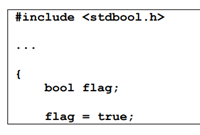
Логический тип



\_Bool – это целочисленный тип (точнее беззнаковый целочисленный тип). В отличие от целочисленного типа переменные типа \_Bool могут принимать только значения 0 и 1. Попытка присвоить отличное от нуля значение переменной типа \_Bool приведет к тому, что значение этой переменной станет равным 1.

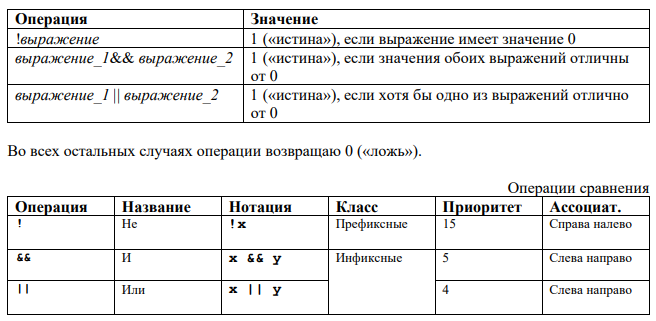
Переменные типа \_Bool могут использоваться в арифметических выражениях (но это не желательно).

Стандарт c99 предоставляет заголовочный файл stdbool.h, который облегчает использование «нового» логического типа. Это заголовочный файл содержит макрос bool для обозначения \_Bool, а также макросы true и false.



Логические операции

Логические операции «И» (&&), «ИЛИ» (||) и «НЕТ» (!) используются для формирования сложных выражения из операций отношений и сравнений. Значениями логических операция являются целые числа 0 («ложь») и 1 («истина»). Операнды этих операций часто имеют такие же значения, но это не обязательно: логические операции интерпретируют любое число, отличное от нуля, как «истину», а 0 – как «ложь»



Вычисление логических выражений

Логические выражения в языке Си всегда вычисляются слева направо

При этом используется так называемая краткая схема вычисления логического выражения: вычисления прекращаются как только становится понятным значение всего выражения.

Например, expression\_1 && expression\_2

если подвыражение expression\_1 ложно, то нет никакой необходимости вычислять значение подвыражения expression\_2, потому что значение всего выражения ложно.

Побитовые операции

Битовые операции позволяют оперировать отдельными битами или группами битов, из которых формируются байты (и машинные слова). Язык Си поддерживает все битовые операции, к которым относятся:

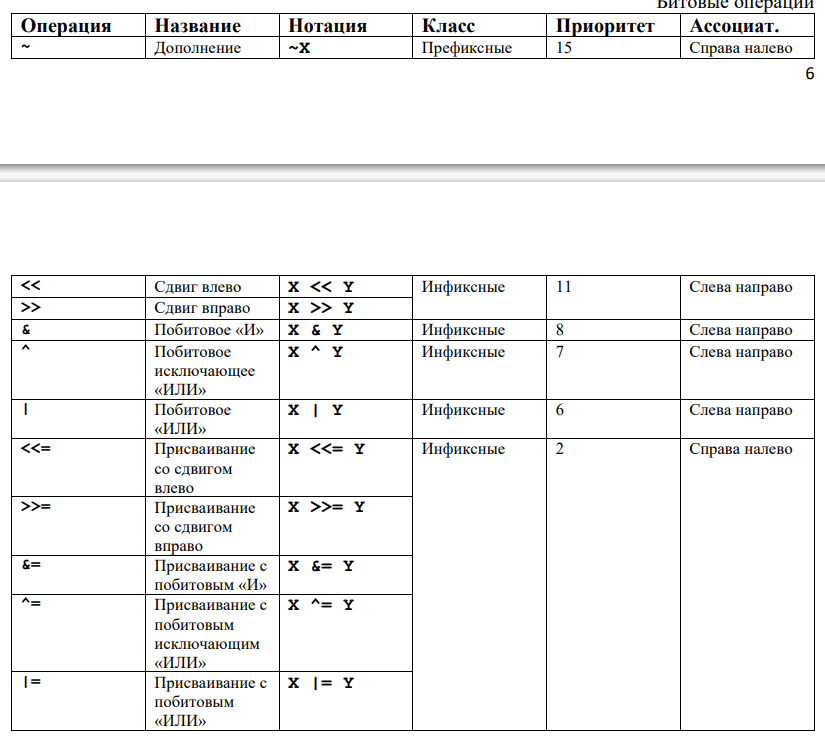
операции сдвига o сдвиг влево (<>);

побитовые операции o дополнение (~);

и (&); o исключающее или (^);

или (|).

Битовые операции применимы только к целочисленным переменным. Обычно эти операции стараются выполнять над беззнаковыми целыми, чтобы • повысить переносимость программы; • не было путаницы с битом, который отвечает за знак.



Операция «битовое дополнение» является унарной, остальные битовые операции – бинарные. Операции ~, &, ^ и | выполняют соответствующие булевы операции над всеми битами своих операндов.

Приоритет побитовых операций меньше приоритета операций сравнения и равенства. Поэтому, например, if (status & 0x4000 != 0) означает if (status & (0x4000 != 0)).

Операции составного присваивания &=, ^=, |= соответствуют битовым операциям &, ^, |

# Вопрос 10. Оператор-выражение составной оператор и оператор выбора

\*не забыть про тернарную и брейк

Оператор выбора, оператор break. Условный оператор, тернарный оператор

**Ответ**

В языке Си любое выражение можно «превратить» в оператор, добавив к этому выражению точку с запятой:

++i;

Когда этот оператор выполняется, переменная i сначала увеличивается на единицу, затем возвращается значение операции префиксного инкремента. Если бы выражение ++i было бы частью другого выражения, это значение там использовалось бы. Однако в нашем случае оно просто отбрасывается, и управление переходит к следующему оператору. Поскольку результат выражения отбрасывается в нем не было бы никакого толку, если бы не побочный эффект.

***Составной оператор***

Составной оператор (или блок) группирует несколько операторов и объявлений путем их размещения в фигурных скобках в единый оператор

{

[список объявлений и операторов]

}

В отличие от оператора выражения составные операторы не завершаются точками с запятой. Составной оператор используется везде, где синтаксис предусматривает несколько операторов, но назначение программы требует их несколько

***Условный оператор***

Условный оператор позволяет программе сделать выбор между двумя альтернативами, проверив значение выражения. Формальный синтаксис оператора таков

if (выражение)

оператор\_1

else

оператор\_2

Скобки вокруг выражения обязательны, они являются частью самого условного оператора. Часть else не является обязательной.

Выполнение условного оператора начинается с вычисления значение выражения в скобках. Если оно истинно (т.е. отлично от нуля), выполняется оператор\_1. Если оно ложно (т.е. имеет нулевое значение) и присутствует часть else, то выполняется оператор\_2. Если часть else отсутствует, управление передается оператору, следующему за условным оператором.

Поскольку в выражении условного оператора анализируется числовое значение этого выражения, отдельные конструкции можно упростить. Например, вместо

if (выражение != 0) ...

часто пишут

if (выражение) ...

Авторы языка считают, что такая запись является более естественной и понятной.

Нет каких-либо ограничений на то, какие операторы могут появляться внутри условного оператора. Нет ничего необычного, когда один условный оператор «вкладывается» внутрь другого. Например, необходимо найти максимальное из трех чисел:

Поскольку часть else условного оператора может отсутствовать, в случае вложенных условных операторов это может приводить к путанице. Например, к какому условному оператору нужно отнести часть else в следующем фрагменте

В языке Си проблема решается благодаря тому, что else всегда связывается с ближайшим предыдущим оператором if без else. В этом примере else относится к внутреннему условному оператору, так что корректно выровненный фрагмент выглядит следующим образом

Для реализации выбора из нескольких альтернатив можно использовать следующую конструкцию (так называемый каскадный условный оператор)

if (выражение\_1)

оператор\_1

else if (выражение\_2)

оператор\_2

...

else if (выражение\_n)

оператор\_n

else

оператор

Эта последовательность условных операторов представляет собой общий способ записи принятия многовариантного решения. Перечисленные выражения вычисляются по порядку, если какое-нибудь из них оказывается истинным, то выполняется соответствующий оператор, и цепочка проверки условий прерывается.

Последний фрагмент данной конструкции (else без условия) обрабатывает случай «ничего из перечисленного». Иногда этот вариант не нужен и его можно опустить.

***Условная операция***

Условная операция состоит из двух символов “?” и “:”, которые используются вместе следующим образом

выражение\_1 ? выражение\_2 : выражение\_3

выражение\_1, выражение\_2 и выражение\_3 могут быть выражениями любого типа. Условная операция требует трех операндов. Сначала вычисляется выражение выражение\_1. Если оно отлично от нуля, то вычисляется выражение выражение\_2, и его значение становится значением условной операции. Если значение выражение выражение\_1 равно нулю, то значением условной операции становится значение выражения выражение\_3.

max = a > b ? a : b;

***Оператор выбора***

Часто при написании программы приходится принимать многовариантное решение, для чего обычно используется каскадный условный оператор.

if (выражение\_1)

оператор\_1

else if (выражение\_2)

оператор\_2

...

else if (выражение\_n)

оператор\_n

else

оператор

Язык Си предоставляет оператор switch как альтернативу такому каскадному условному оператору

switch (mark)

{

case 5: printf("Excellent\n");

break;

case 4: printf("Good\n");

break;

case 3: printf("Average\n");

break;

case 2: printf("Poor\n");

break;

default: printf("Illegal mark\n");

break;

}

Считается, что оператор switch легче читается. Кроме того, switch часто быстрее, чем каскадный условный оператор.

Особенности оператора switch

* Управляющее выражение, которое располагается за ключевым словом switch, обязательно должно быть целочисленным (не вещественным, не строкой).
* Константное выражение – это обычное выражение, но оно не может содержать переменных и вызовов функций
* После каждой case-метки может располагаться любое число операторов. Никакие скобки не требуются. Последним оператором в группе таких операторов обычно бывает break.
* case-метки не могут быть одинаковыми.
* Порядок саse-меток (даже метки default) не важен.
* Только одно константное выражение может располагаться в case-метке, но несколько caseметок могут предшествовать одной и той же группе операторов
* case-метка default не является обязательной

Выполнение оператора break «внутри» оператора switch передает управление за оператор switch, т.е. на оператор, который располагается за оператором switch. Если бы оператор break отсутствовал, то стали бы выполняться операторы, расположенные в следующих case-метках

Пропуск оператора break является частой ошибкой при использовании оператора switch.

**Оператор break**

Оператор ***break*** используется для принудительного выхода из циклов ***while***, ***do while***, ***for***. Выход выполняется из ближайшего цикла или оператора ***switch***.

Пример

// Определение "простоты" числа "в лоб"

#include <stdio.h>

int main(void)

{

int d, n = 17;

for (d = 2; d < n; d++)

if (n % d == 0)

break;

if (n < d)

printf("………………");

else

printf("………………");

return 0;

}

# Вопрос 11. While, do-while, break, continue

\*выразить один цикл через другой(в вопросе всегда два цикла), возможно реализовать цикл через goto

**Ответ**

**Оператор while**

В языке Си цикл с предусловием реализуется с помощью оператора while. В общей форме этот оператор записывается следующим образом

**while (выражение) оператор**

Скобки вокруг выражения являются обязательными. Оператор после скобок представляет собой тело цикла.Выполнение оператора while начинается с вычисления значение выражения. Если оно отлично от нуля, выполняется тело цикла, после чего значение выражения вычисляется еще раз. Процесс продолжается в подобной манере до тех пор, пока значение выражения не станет равным 0. Таким образом, возможно ситуация, когда тело такого цикла не выполнится ни разу.

При работе над алгоритмом следите, чтобы условие в теле цикла менялось, чтобы переменным, которые в нем задействованы, присваивались новые значения. Например, если условие цикла у вас (x< 0), а в теле цикла переменной x вообще нет, такой цикл если и начнет выполняться, никогда не закончится.

Пример правильной реализации цикла ***while***

// Сумма первых n натуральных чисел

#include <stdio.h>

int main(void)

{

int sum, i, n = 5;

i = 1;

sum = 0;

while (i <= n)

{

sum += i;

i++;

}

return 0;

}

Пример неправильной реализации цикла ***while***

// Сумма первых n натуральных чисел

#include <stdio.h>

int main(void)

{

int sum, i, n = 5;

i = 1;

sum = 0;

while (i <= n)

{

sum += i;

}

return 0;

}

**Оператор do while**

В языке Си цикл с постусловием реализуется с помощью оператора while. В общей форме этот оператор записывается следующим образом

**do оператор while (выражение)**

Скобки вокруг выражения являются также обязательными.

Выполнение оператора ***do while*** начинается с тела цикла (оператора), после которого вычисляется выражение. Если оно отлично от нуля, тело цикла выполняется еще раз и так далее. Выполнение оператора ***do-while*** заканчивается, когда значение этого выражения станет равным нулю.

Пример работы

// Вычисление количества цифр в числе

#include <stdio.h>

int main(void)

{

int digits = 0, n = 1024;

do

{

digits++;

n /= 10;

}

while (n != 0);

return 0;

}

**Оператор break**

Оператор ***break*** используется для принудительного выхода из циклов ***while***, ***do while***, ***for***. Выход выполняется из ближайшего цикла или оператора ***switch***.

Пример

// Определение "простоты" числа "в лоб"

#include <stdio.h>

int main(void)

{

int d, n = 17;

for (d = 2; d < n; d++)

if (n % d == 0)

break;

if (n < d)

printf("………………");

else

printf("………………");

return 0;

}

**Оператор continue**

Оператор ***continue*** принудительно переходит в конец оператора цикла(дальнейшее тело цикла пропускается). Оператор ***continue*** можно использовать только в теле цикла.

Пример

// Вычисление суммы десяти положительных чисел

#include<stdio.h>

int main(void)

{

int n, i = 0, sum = 0;

while (i < 10)

{

scanf("%d", &n);

if (n <= 0)

continue;

sum += n; i++;

}

return 0;

}

**Выражение одного цикла через другой**

вообще хрен выразишь один через другой, потому что в do while тело выполняется хоть раз, а в while – не факт.

**do while через while (из примеров)**

// Вычисление количества цифр в числе

#include <stdio.h>

int main(void)

{

int digits = 0, n = 1024;

while (n != 0)

{

digits++;

n /= 10;

}

return 0;

}

**while через do while (из примеров)**

// Сумма первых n натуральных чисел (n > 0)

#include <stdio.h>

int main(void)

{

int sum, i, n = 5;

i = 1;

sum = 0;

do

{

sum += i;

i++;

}

while (i <= n);

return 0;

}

# Вопрос 12. for, while, пустой оператор (;)

\*обычно пустой оператор с for, рассказать оператор запятая

**Ответ**

**Оператор while**

В языке Си цикл с предусловием реализуется с помощью оператора while. В общей форме этот оператор записывается следующим образом

**while (выражение) оператор**

Скобки вокруг выражения являются обязательными. Оператор после скобок представляет собой тело цикла.Выполнение оператора while начинается с вычисления значение выражения. Если оно отлично от нуля, выполняется тело цикла, после чего значение выражения вычисляется еще раз. Процесс продолжается в подобной манере до тех пор, пока значение выражения не станет равным 0. Таким образом, возможно ситуация, когда тело такого цикла не выполнится ни разу.

При работе над алгоритмом следите, чтобы условие в теле цикла менялось, чтобы переменным, которые в нем задействованы, присваивались новые значения. Например, если условие цикла у вас (x< 0), а в теле цикла переменной x вообще нет, такой цикл если и начнет выполняться, никогда не закончится.

Пример правильной реализации цикла ***while***

// Сумма первых n натуральных чисел

#include <stdio.h>

int main(void)

{

int sum, i, n = 5;

i = 1;

sum = 0;

while (i <= n)

{

sum += i;

i++;

}

return 0;

}

Пример неправильной реализации цикла ***while***

// Сумма первых n натуральных чисел

#include <stdio.h>

int main(void)

{

int sum, i, n = 5;

i = 1;

sum = 0;

while (i <= n)

{

sum += i;

}

return 0;

}

**Оператор for**

Оператор forобычно используют для реализации цикла со счетчиком. В общей форме этот оператор записывается следующим образом

**for** **(выражение\_1; выражение\_2; выражение\_3) оператор**

Оператор цикла ***for*** может быть заменен (за исключением редких случаев) оператором ***while***

выражение\_1;

while (выражение\_2)

{

оператор

выражение\_3;

}

Выражение\_1 – шаг инициализации, который выполняется только один раз до того, как цикл начнет выполняться.

Выражение\_2 – выражение отношения или логическое выражение. Управляет завершением цикла (тело цикла выполняется пока выражение выражение\_2 отлично от нуля).

Выражение\_3 – выражение, которое выполняется в конце каждой итерации цикла.

#include <stdio.h>

int main(void)

{

int sum = 0, i = 1, n = 5;

while (i <= n)

{

sum += i;

i++;

}…

}

#include <stdio.h>

int main(void)

{

int sum = 0, i, n = 5;

for (i = 1; i <= n; i++)

sum += i;

…

}

Любое из трех выражений можно опустить, но точки с запятой должны остаться на своих местах. Если опустить выражение\_1 или выражение\_3, то соответствующие действия выполняться не будут.

i = 1;

for ( ; i <= n; )

sum += i++;

Если же опустить проверку условия выражение\_2, то по умолчанию считается, что условие продолжения цикла всегда истинно.

for ( ; ; )

printf("Infinity loop\n");

В С99 первое выражение выражение\_1 в цикле for может быть заменено определением. Эта особенность позволяет определять переменные для использования в цикле

for (int i = 0; i < n; i++)

Переменную i не нужно объявлять до оператора for. При этом переменная i может быть доступна только из тела цикла.

**Операция запятая**

Иногда бывает необходимо написать оператор for с двумя или более выражениями инициализации или изменить несколько переменных в конце цикла.

Это можно сделать с помощью операции запятая.

Операция запятая выглядит следующим образом

**выражение\_1, выражение\_2**

Эта операция выполняется следующим образом: сначала вычисляется выражение\_1 и его значение отбрасывается, затем вычисляется выражение\_2. Значение этого выражения является результатом операции всей операции.выражение\_1 всегда должно содержать побочный эффект. В противном случае от этого выражения не будет никакого толка.

for (sum = 0, i = 1, n = 5; i <= n; sum += i, i++)

...

**Пустой оператор**

Пустой оператор состоит только из символа “;”. Основная «специализация» пустого оператора –реализация циклов с пустым телом:

for (d = 2; d < n; d++)

if (n % d == 0)

break;

for (d = 2; n % d != 0 && d < n; d++)

;

(одно и то же типа)

Пустой оператор легко может стать источником ошибки

if (d == 0);

printf("ERROR: division by zero!\n");

будет ошибка.

# Вопрос 13. for, do-while

\*-//-

**Ответ**

**Оператор do while**

В языке Си цикл с постусловием реализуется с помощью оператора while. В общей форме этот оператор записывается следующим образом

**do оператор while (выражение)**

Скобки вокруг выражения являются также обязательными.

Выполнение оператора ***do while*** начинается с тела цикла (оператора), после которого вычисляется выражение. Если оно отлично от нуля, тело цикла выполняется еще раз и так далее. Выполнение оператора do-while заканчивается, когда значение этого выражения станет равным нулю.

Пример работы

// Вычисление количества цифр в числе

#include <stdio.h>

int main(void)

{

int digits = 0, n = 1024;

do

{

digits++;

n /= 10;

}

while (n != 0);

return 0;

}

**Оператор for**

Оператор forобычно используют для реализации цикла со счетчиком. В общей форме этот оператор записывается следующим образом

**for (выражение\_1; выражение\_2; выражение\_3) оператор**

Оператор цикла ***for*** может быть заменен (за исключением редких случаев) оператором ***dwhile***

выражение\_1;

while (выражение\_2)

{

оператор

выражение\_3;

}

Выражение\_1 – шаг инициализации, который выполняется только один раз до того, как цикл начнет выполняться.

Выражение\_2 – выражение отношения или логическое выражение. Управляет завершением цикла (тело цикла выполняется пока выражение выражение\_2 отлично от нуля).

Выражение\_3 – выражение, которое выполняется в конце каждой итерации цикла.

#include <stdio.h>

int main(void)

{

int sum = 0, i = 1, n = 5;

while (i <= n)

{

sum += i;

i++;

}…

}

#include <stdio.h>

int main(void)

{

int sum = 0, i, n = 5;

for (i = 1; i <= n; i++)

sum += i;

…

}

Любое из трех выражений можно опустить, но точки с запятой должны остаться на своих местах. Если опустить выражение\_1 или выражение\_3, то соответствующие действия выполняться не будут.

i = 1;

for ( ; i <= n; )

sum += i++;

Если же опустить проверку условия выражение\_2, то по умолчанию считается, что условие продолжения цикла всегда истинно.

for ( ; ; )

printf("Infinity loop\n");

В С99 первое выражение выражение\_1 в цикле for может быть заменено определением. Эта особенность позволяет определять переменные для использования в цикле

for (int i = 0; i < n; i++)

Переменную i не нужно объявлять до оператора for. При этом переменная i может быть доступна только из тела цикла.

**Выражение одного цикла через другой**

**for через do while**

выражение\_1;

do

{

оператор

выражение\_3;

}

while (выражение\_2);

**do while через for**

for (;;)

{

оператор

if (!выражение)

break;

}

# Вопрос 14. Тип данных, простые типы

\*какая бывает типизация(строгая, нестрогая, динамическая, статическая), про простые типы (int, float …), можно про перечисляемый, как инициализировать переменную типа, ввод/вывод, typedef, sizeof

Классификация типов (перечислить)

Простые типы, целые, вещественные, символьные, перечисляемые и логические (но не задерживаться на них)

Рассказ про тип сопровождается как определить переменную с ним, как сделать вывод с ним, особенности

Про то что вещественный тип обычными способами не сравнить

Определение пользовательских типов, конструкция type def, size of

**Ответ**

***Типизированные языки разделяются на несколько категорий:***

*Статическая / динамическая типизация:*

При ***статической типизации*** конечные типы данных должны быть известы уже перед этапом компиляции, \*\*при динамической\*\* - тип данных определется на этапе исполнения программы.

**Строгая / нестрогая (или сильная / слабая) типизация:**

При строгой типизации не разрешается смешивать в одном выражении различные типы данных (например целое и вещественное), а так же выполнять автоматические неявные преобразования типов. При ***несторгой типизации*** выполняются неявные преобразования, даже они эти преобразнания неодозначны или если может произойти потеря точности вычислений.

Так, язык Си, к примеру, относится к языкам с нестрогой статической типизацией.

Тип в языка программирования

Тип переменной определяет множество значений, которые может принимать переменная,

набор операций, которые можно применять к таким значениям, и способ хранения значений и

выполнения операций.

Обычно различают *простые* и *составные* типы. Переменные простого типа могут содержать

только одно значение, а переменные составного типа – несколько однотипных или

разнотипных значений.

Простые типы

В языке Си выделяются следующие простые типы:

1) целые \*int\*

2) вещественные \*float\*и др.

3) символьные \*char\*

4) перечислимый тип

5) void

6) указатели

Составными типы

В языке Си выделяются следующие простые типы:

1) массив

2) структура

3) объединение

***1) Целый тип***

Переменные

У данного типа есть модификатокы (\*signed\*, \*unsigned\*, \*long\*, \*short\*), применение которых позволяет несколько типовелых чисел, отличающихся друг от друга по объему памяти и возможности присваивания положительного / отрицательного числа.

Всего можно выделить 6 (начиная со стандарта С99 8, в силу необходимости поддержки больших чисел на процессорах 64-х) различных типов данных:

short int

short unsigned int

int

unsigned int

long int

long unsigned int

*Начиная со стандарта С99*

long long int

long long unsigned int

Все остальные комбинации будут синонимами к типам выше.

Так например:

short int и short - синонимы, так как допускается опускать слово int

signed int и int - синонимы, так как изначально целые типы знаковые

Константы

Целочисленные костанты можно записать в \*десятичной\*, \*восьмиричной\* или \*шестнадцатиричной\* системе счисления.

16 200 // Десятичная

020 0310 // Восьмиричная (0 в начале обязателен)

0x1 0xc8 // Шестнадцатиричная (0x в начале обязателен)

Чтобы заставить компилятор интерпретировать константу как long int необходимо добавить

суффикс L или l к значению (в случае типа long long int – LL или ll)

16L 020l 0x1L

Чтобы показать, что константа беззнаковая, необходимо добавить суффикс U и u к значению (Суффиксы L (или LL) и U могут использоваться вместе.)

16U 0310LLU 0xc8LU

*Чтение и вывод на экран :*

Для десятичной

1) Тип int. Используется спецификатор d

printf("Тип int: ");

int n = 0;

scanf("%d", &n);

printf("n = %d\n", n);

2) Тип short int. Для вывод необходимо к спецификатору добавить h

printf("Тип short: ");

short s = 0;

scanf("%hd", &s);

printf("s = %hd\n", s);

3) Для использования типов long int и long long int необходимо добавить спецификаторы l и ll соответственно

printf("Типы long и long long: ");

long ln = 0;

long long lln = 0;

scanf("%ld %lld", &ln, &lln);

printf("ln = %ld;\n lln = %lld", ln, lln);

4) Чтобы распечатать беззнаковый целочисленный тип необходимо заменить спецификатор d на u

printf("Любой беззнаковый целочисленный тип(в данном случае long): ");

unsigned long uln = 0;

scanf("%lu", &uln);

printf("uln = %lu", uln);

5) У восьмиричной и шестнадцатиричной логика таже, только необходимо для восьмиричной поменять спецификатор u на o, а для шестнадцатиричной - u на x

printf("Восьмиричная СС: ");

unsigned long lo = 0;

scanf("%lо", &lo);

printf("lo = %lo", lo);

printf("Шестнадцатиричная СС: ");

unsigned x = 0;

scanf("%x", &x);

printf("x = %x", x);

***2) Вещественный тип***

Переменные

Стандарт С99 предоставляет следующие вещественные типы

float

double

long double

Друг от друга эти типы отличаются количетством знаков после запятой(то бишь точностью). Так \*float\* это наимение точная величина, а \*long double\* - наиболее

В силу особенностей хранения вещественный чисел в памяти (разбиение на мантису, знак и экспененту) могут возникать погрешности, из-за чего лучше избегать прямого сравнения == и пользоваться сравнением через епсилон:

float f1 = 0.01, f2 = 0.02, eps = 1e-7;

if (fabs(f1 - f2) < eps)

printf("Equal");

else

printf("Not equal");

Константы

Имеют место следующие записи:

57.0 57. 5.7e1 57E0 .57

Вещественная константа должна содержать десятичную точку и/или экспоненту.

По умолчанию вещественные константы имеют тип \*double\*. Чтобы заставить компилятор интерпретировать константу как \*float\* необходимо добавить F (или f), а тобы как \*long double\* - L (или l)

57.0F 57.f

5.7e1L 57E0l

*Чтение и запись*

1) Для чтения или записи типа \*float\* можно использовать спецификаторы f, e или g

printf("Тип float");

float f = .0, e = 0., g = 0.0;

scanf("%f %e %g", &f, &e, &g);

printf("f = %f, e = %e, g = %g", f, e, g);

2) Для чтения \*double\* необходимо добавить с спецификатору l слева, а вот для вывода этого лучше не делать, поскольку стандарт С99 позволяет пользоваться обычными спецификаторами и для этого типа

printf("Тип double");

double lf = .0, le = 0., lg = 0.0;

scanf("%lf %le %lg", &lf, &le, &lg);

printf("lf = %f, le = %e, lg = %g", lf, le, lg);

3) Для чтения и записи \*long double\* необходимо добавить к спеуификатору L слева

printf("Тип long double");

long double llf = .0, lle = 0., llg = 0.0;

scanf("%Lf %Le %Lg", &llf, &lle, &llg);

printf("llf = %Lf, lle = %Le, llg = %Lg", llf, lle, llg);

***3) Символьный тип***

Для работы с символами предназначен тип \*char\*. При этом под набором символов, как правило, понимается набор символов ASCII.

Переменной типа char может быть присвоено значение любого ASCII символа:

char ch = 'H';

ch = '3';

ch = '!';

ch = 'p';

ch = ' ';

Под значение типа char отводится один байт и интерпретируется как \*short\*, те в переменную типа \*char\* помещается не сам символ, а его код из таблицы ASCII.

Стандарт не определяет «знаковость» этого типа. При необходимости это можно явно задать при помощи модификаторов \*signed\* / \*unsigned char\*

Чтение и запись

Для чтения и записи используется спецификатор c

printf("Тип char");

char c = 'W';

scanf("%c", &c);

printf("c = %c или c = %d", c, c);

***4) Перечисляемый тип***

В языке Си предусмотрен тип, разработанный для переменных, которые принимают небольшое количество значений. Это перечисляемый тип, значения которого

перечисляются программистом, и интерпритируются «константы» как целые. По умолчанию компилятор назначает неотрицательные целые значения (по возрастанию, те 0, 1 и тд) «константам» в каждом конкретном перечислении.

enum season\_e {WINTER, SPRING, SUMMER, AUTUMN};

Поскольку значения перечисляемого типа — это целые числа, позволяется их перемешивать с обычными целыми.

int i;

enum {WINTER, SPRING, SUMMER, AUTUMN} season;

i = WINTER; // i === 0

season = 0; // season === WINTER

season++; // season === SPRING

***5) Логические тип***

В стандарте С99 был добален тип \*\_Bool\*. Это беззнаковый целочисленный тип, с допустимыми значениями 0 и 1, те если записать любое отличное от 0 значение, то переменная будет равна 1.

Этот тип можно использовать в арифметических выражениях, но не нужно

Определение пользовательского типа

Конструкция typedef позволяет задавать собственное имя для типа данных. Здесь "тип" - любой разрешенный тип данных и "имя" - любой разрешенный идентификатор.

typedef unsigned int length; // тип - unsigned int, название - length

Использование конструкции typedef позволяет улучшить читаемость программы сопровождение программы

Операция sizeof

Унарная операция sizeof позволяет определить какое количество памяти требуется для хранения величины определенного типа. Значением выражения является беззнаковое целое, представляющее собой количество байт необходимых для хранения величины типа type\_name.

sizeof(type\_name)

Когда операция sizeof применяется к выражению скобки могут быть опущены.

printf("%lu", sizeof( 1 / 1)); // 4

printf("%lu", sizeof((double) 1 / 1)); // 8

# Вопрос 15. Явное и неявное преобразование типов

Для чего это нужно, идеи лежащие в основе, проблемы преобразования типов

Проблемы с неявным преобразованием типа, примеры

Операция приведения, особенности использования явного определения типа

**Ответ**

***Для чего это нужно***

Си позволяет использовать в выражениях операнды разных типов. Например, вы можете записать выражение, которое одновременно содержит целые, вещественные и символьные операнды. Однако компилятор во время трансляции программы в машинный код такой гибкости лишен, потому что машинные команды, как правило, обрабатывают операнды одного размера и одного типа. Компилятор вынужден автоматически преобразовывать типы операндов так, чтобы подобрать подходящую машинную команду. Такие преобразования без участия программиста называются *неявными* (С99 6.3). Си также позволяет программисту при необходимости выполнить *явное* преобразование типа (С99 6.5.4), используя операцию преобразования типа.

***Идеи, лежащие в основе***

Правила, согласно которым выполняется неявное преобразование типа, достаточно сложны, потому что в Си много различных арифметических типов. Поэтому познакомимся лишь с основными идеями, которые лежат в основе этих правил.

Неявное преобразование выполняется в следующих ситуациях:

* когда операнды в арифметическом или логическом выражении имеют разные типы (это преобразование даже получило специальное название – обычное арифметическое преобразование;
* когда тип выражения с правой стороны операции присваивания не совпадает с типом выражения с левой стороны;
* когда тип аргументов в вызове функции не соответствует типу параметра в ее заголовке;
* когда тип выражения в операторе return не соответствует типу функции.

***Обычное арифметическое преобразование*** - стратегия, которая лежит в основе арифметического преобразования, очень проста – преобразовать операнды к самому «широкому» типу, который может содержать значения обоих операндов. Такое преобразование иногда называют расширением типа.

***Преобразования во время присваивания*** - обычное арифметическое преобразование не работает, когда речь идет об операции присваивания. Си в этом случае поступает очень просто: тип выражения с правой стороны преобразуется к типу переменной с левой стороны. Если тип переменной такой же «широкий», как и тип выражения, это не вызовет никаких затруднений

***Преобразование типов аргументов при вызове функции*** - Си разрешает вызывать функции, когда типы аргументов не соответствуют типам параметров. Правила, по которым выполняется такое преобразование, зависят от того, известен ли компилятору прототип функции.

1. Прототип известен. Значение каждого аргумента неявно преобразуется к типу соответствующего параметра как при присваивании.
2. Прототип неизвестен. Компилятор выполняет расширение по умолчанию:
3. float преобразуется в double,
4. выполняется целочисленное расширение.

***Преобразование типов при выходе из функции*** - если тип выражения в операторе ***return*** не соответствует типу функции, выражения будет неявно преобразовано к типу функции (как при присваивании).

***Явное преобразование***

Иногда преобразование типа нужно выполнить явно. Операция преобразования типа имеет вид

(имя типа) выражение

где *«имя типа»* указывает тип, к которому будет преобразовано выражение.

Следующий пример показывает, как вычислить дробную часть вещественного числа

float f = 25.154, fract\_part;

frac\_part = f – (int) f;

Операцию явного преобразования типа можно использовать для *«документирования»* преобразования типа, которое и так имеет место

i = (int) f;

***Проблемы преобразования типов***

***Обычное арифметическое преобразование*** - когда знаковый операнд используется вместе с беззнаковым, знаковый операнд преобразуется в беззнаковое значение. Это преобразование заключается в прибавлении величины UMAX + 1, где UMAX представляет наибольшее значение представимое в беззнаковом типе.

Иногда это может приводить к странным ошибкам

#include

int main(void)

{

int si = -1;

unsigned int ui = 1;

printf("%d\n", si < ui);

return 0;

}

При сравнении переменных si и ui с помощью операции меньше, естественно ожидать, что результатом сравнения будет значение «истина». Но до выполнения операции сравнения si 3 преобразуется в тип unsigned int и значение переменной si становится равным 4294967295. Таким образом, сравнение si < ui дает значение «ложь».

***Преобразования во время присваивания*** - если тип переменной такой же «широкий», как и тип выражения, это не вызовет никаких затруднений

В противном случае произойдет «сужение» типа и результат может не соответствовать ожидаемому:

char c = 'a';

int i = 2020;

float f;

double d;

c = 1500; // NOT OK (-36)

i = 1.0e20; // NOT OK (2147483647)

f = 1.0e100; // NOT OK (1.#INF00)

В этом случае компилятор обычно выдает предупреждение (например, *“overflow in implicit constant conversion”* или *“floating constant exceeds range of 'float'”*).

# Вопрос 16. Подпрограмма

\*какие преимущества при использовании, типы подпрограмм, структура, параметры, рассказать как именно в си

Подпрограммы, преимущества, формы подпрограмм, процедуры, функции, общее описание, заголовок и тело, вызовы - отдельная операция, параметры, способы передачи параметров, чем отличаются параметры от аргументов

Пример на языке Си

**Ответ**

***Подпрограмма***

***Подпрограмма*** – именованная часть программы, содержащая описание определённого набора действий. Подпрограмма может быть многократно вызвана из разных частей программы

***Преимущества***

Преимущества использования подпрограмм:

Уменьшение сложности программирования за счет декомпозиции большой задачи на несколько подзадач меньшего размера, которые легче понимать.

Сокрытие деталей реализации от пользователей подпрограммы.

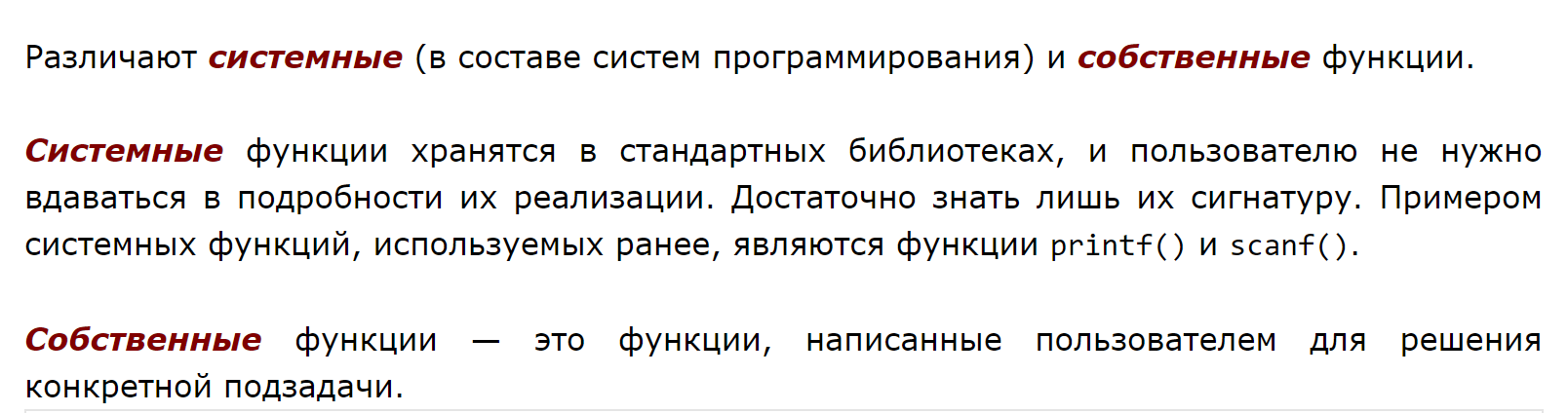
Уменьшение дублированного кода.

Возможность повторного использования кода в других программах.

***Виды подпрограмм***

Функция – это подпрограмма специального вида, которая всегда должна возвращать результат. Вызов функции является, с точки зрения языка программирования, выражением.

Процедура – это независимая именованная часть программы, которую после однократного описания можно многократно вызвать по имени из последующих частей программы для выполнения определенных действий



***Что такое заголовок подпрограммы?***

В языках высокого уровня описание подпрограммы обычно состоит из двух частей: заголовка и тела.

Заголовок подпрограммы описывает её имя и параметры, т.е. информацию, необходимую для вызова подпрограммы.

Заголовок зависит от вида подпрограммы. Он несет в себе следующую информацию:

- вид подпрограммы (процедура или функция);

- имя подпрограммы;

- список формальных параметров;

- тип результата (только для функций).

***Что такое тело подпрограммы?***

Тело подпрограммы – набор операторов, который будет выполнен всякий раз, когда подпрограмма будет вызвана.

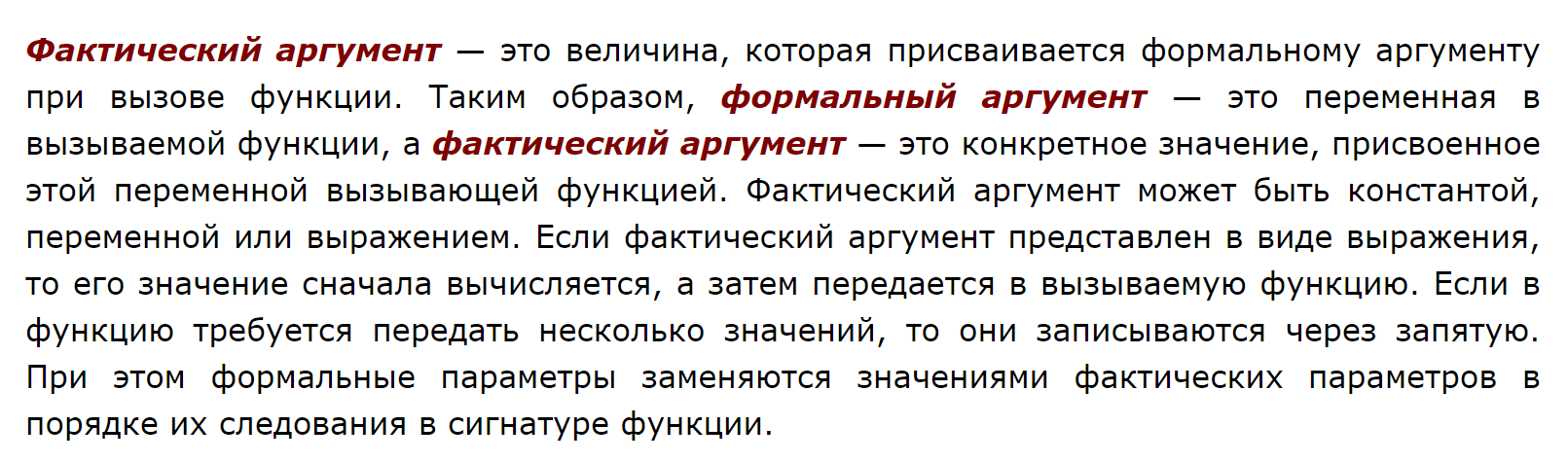
***Вызов подпрограммы. Формальные и фактические параметры.***

Вызов подпрограммы выполняется с помощью команды вызова, включающей в себя имя подпрограммы, за которым следуют параметры

Чтобы отличать параметры подпрограммы, описанные в её заголовке и теле, от параметров, указываемых при вызове подпрограммы, используются формальные и фактические параметры.

Формальные параметры указываются при описании подпрограммы, а фактические – непосредственно при её вызове.

Фактические параметры часто называют аргументами.



***Способы передачи параметров в подпрограмму. Реализация этих способов применительно к языку Си.***

Существуют две концептуальные модели передачи данных при передаче параметров:

либо фактическое значение физически перемещается (передача по значению, англ. pass by value),

либо передается путь доступа к нему (передача по ссылке, англ. pass by reference).

Передача параметров по значению

Когда параметр передается по значению, значение фактического параметра используется для инициализации соответствующего формального параметра, который в дальнейшем действует как локальная переменная.

Передача параметров по ссылке

Метод передачи по ссылке передает путь доступа к данным (обычно – просто адрес) в вызываемую подпрограмму. Это открывает доступ к ячейке памяти, хранящий фактический параметр. Таким образом, вызываемая подпрограмма может получить доступ к фактическому параметру.

В Си аргументы передаются в функцию «по значению».

«Благодаря этому свойству обычно удается написать более компактную программу, содержащую меньшее число посторонних переменных, поскольку параметры можно рассматривать как должным образом инициализированные локальные переменные.» K&R

Из-за такого способа передачи параметров возникают трудности при реализации функций, которые должны возвратить несколько параметров одновременно.

# Вопрос 17. Описание функций на языке Си

\*общее описание функций(заголовок, тело), return, вызов функции, объявление функции, передача параметров(по значению)

Составные части заголовка, тело функции (особенности), оператор return, как вызывается функция, что такое объявление, особенности передачи параметров (по значению и не только) и сильные и слабые стороны)

**Ответ**

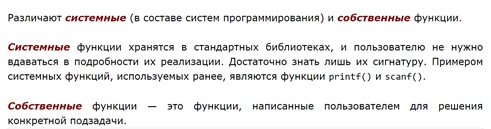
***Описание функции на языке Си.***

***Функция*** - это совокупность объявлений и операторов, обычно предназначенная для решения определенной задачи.

Функция — это самостоятельная единица программы, которая спроектирована для реализации конкретной подзадачи.Функция является подпрограммой, которая может содержаться в основной программе, а может быть создана отдельно (в библиотеке). Каждая функция выполняет в программе определенные действия.

***Сигнатура*** функции определяет правила использования функции. Обычно сигнатура представляет собой описание функции, включающее имя функции, перечень формальных параметров с их типами и тип возвращаемого значения.

***Семантика*** функции определяет способ реализации функции. Обычно представляет собой тело функции.



***Особенности описания функций без параметров в языке Си.***

void foo(void);

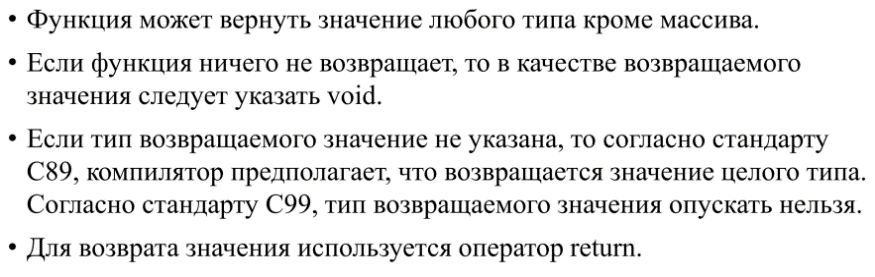
Это объявление означает, что у функции нет ни одного параметра.

void foo();

Это объявление означает, что у функции могут быть, а могут и не быть параметры. Если параметры есть, мы не знаем ни их количество, ни их тип.

***Заголовок функции языке Си.***

ТипВозвращаемогоЗначения ИмяФункции(СписокФормальныхАргументов)



Заголовок подпрограммы описывает её имя и параметры, т.е. информацию, необходимую для вызова подпрограммы.

Заголовок функции состоит из трех частей. Сначала нужно указать тип значения которое возвращает функция. После типа возвращаемого значения идет имя функции. Следом за именем функции в скобках, пишут типы и количество аргументов (параметров) функции. Т.е. значения которые передаются в функцию.

***Тело функции на языке Си.***

После заголовка функции в фигурных скобках пишется тело функции.

Здесь описано как должна эта функции работать. Если функция должна возвращать какое-то значение то в теле функции обязательно должна быть инструкция return.

У каждой функции есть исполнимая часть, которая называется телом функции и заключена в фигурные скобки.

Тело функции может содержать как описания переменных, так и операторы. Переменные, описанные в теле функции, «принадлежат» только этой функции и не могут быть ни получены, ни изменены другой функцией.

Тело функции не может содержать в себе определения других функций (т.е. в языке Си нет вложенных функций).

Если функция ничего не возвращает, ее тело может быть пустым.

***Оператор return: назначение, использование.***

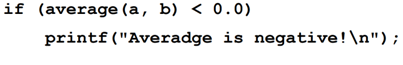
Оператор return прерывает выполнение функции и возвращает вычисленное значение и управление в ту часть программы, из которой эта функция была вызвана.

Оператор return может использоваться в функциях, которые ничего не возвращают. В этом случае он записывается без указания выражения. Помещать оператор return в конец функции, которая ничего не возвращает можно, но не имеет никакого смысла.

***Вызов функции на языке Си.***

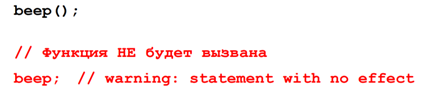
Для вызова функции необходимо указать ее имя, за которым в круглых скобках через запятую перечислить аргументы.

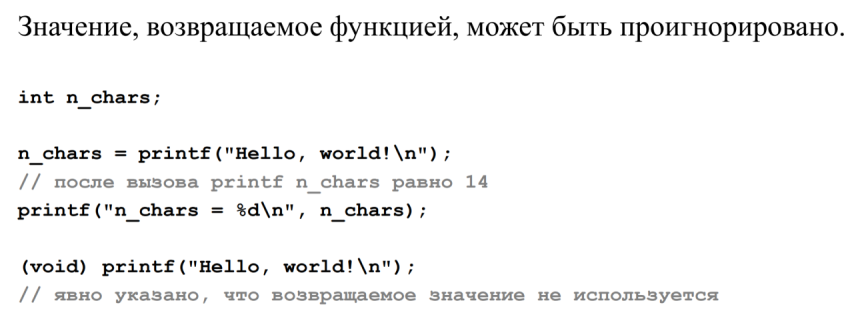




Если функция возвращает значение, ее можно использовать в выражениях.

Указывать скобки при вызове функции необходимо, даже если у этой функции нет параметров.





***Выполнение вызова функции.***

Выделяется область памяти доступная только вызываемой функции (power).

В этой области памяти создаются переменные-параметры (base, n) и локальные переменные (res).

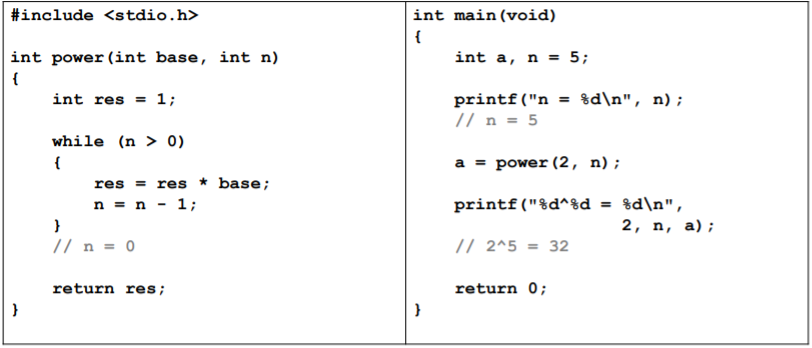
Переменным-параметрам присваиваются начальные значения, в качестве которых выступают аргументы из точки вызова (base = 2, n = 5).

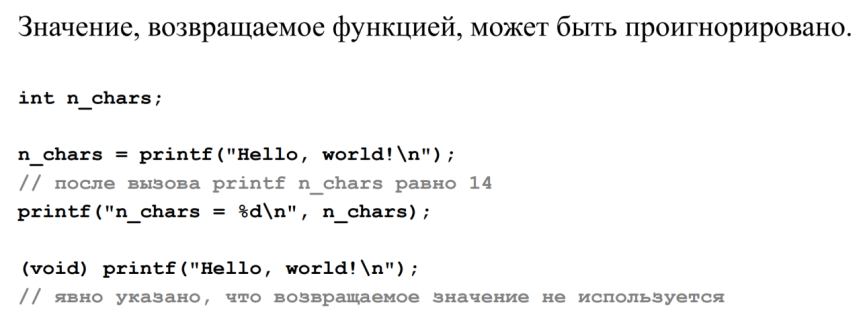
Инициализируются локальные переменные (res = 1). В случае отсутствия такой инициализации локальные переменные содержат «мусор»

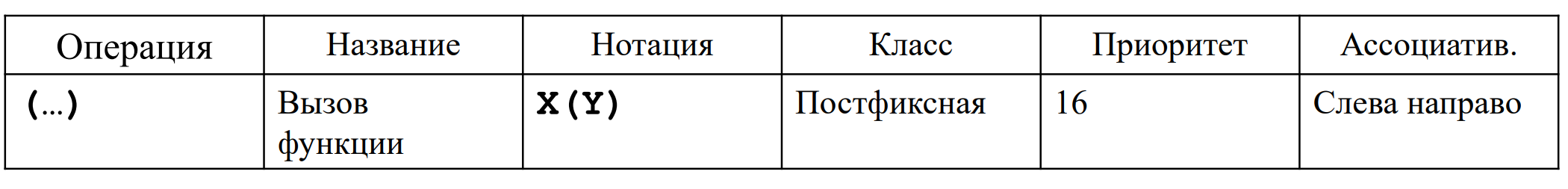
Выполняется тело функции.

Вычисленное значение возвращается из функции (в нашем случае попадает в переменную a).

Выделенная область памяти разрушается







***Объявление функции на языке Си.***

Объявление функции состоит из заголовка функции

тип-результата имя-функции(список формальных параметров с их типами);

Объявление функции должно соответствовать ее определению.

Объявление функции может не содержать имен параметров. Однако их обычно оставляют для большей наглядности.

***Определение функции на языке Си.***

тип имя\_функции(параметры)

{

    инструкции

}

Каждая функция в языке Си должна быть определена, то есть должны быть указаны:

тип возвращаемого значения;

имя функции;

информация о формальных аргументах;

тело функции.

Определение функции имеет следующий синтаксис:

ТипВозвращаемогоЗначения ИмяФункции(СписокФормальныхАргументов)

{

   ТелоФункции;

  ...

return(ВозвращаемоеЗначение);

}

***Способ передачи параметров в функцию на языке Си.***

Передача параметров по значению

Когда параметр передается по значению, значение фактического параметра используется для инициализации соответствующего формального параметра, который в дальнейшем действует как локальная переменная.

Передача параметров по ссылке

Метод передачи по ссылке передает путь доступа к данным (обычно – просто адрес) в вызываемую подпрограмму. Это открывает доступ к ячейке памяти, хранящий фактический параметр. Таким образом, вызываемая подпрограмма может получить доступ к фактическому параметру.

Любая функция может принимать параметры.

Перед именем каждого параметра указывается его тип. Тип указывается перед каждым параметром, даже если несколько параметров имеют один и тот же тип!

Описания параметров в списке разделяются запятыми.

Если у функции нет параметров, вместо списка параметров указывается ключевое слово void.

# Вопрос 18. Функция передачи параметров

\*в си-по значению, что происходит при вызове функций, что делать, если надо вернуть несколько значений(указатели)

Особенности передачи параметров (по значению и по адресу) действия которые происходят при вызове функции

Что нужно сделать, когда нужно вернуть несколько параметров.

Определение указателя, базовые функции для работы с ним, использование указателя для передачи параметра

**Ответ**

В языке C в функцию может передаваться от 0 до произвольного числа параметров.

int function(par1, par2, par3)

Есть разница между передачей в функцию значения переменной и её адреса.

В первом случае значение переменной просто копируется в функцию.

Во втором передаётся её адрес, что даёт возможность изменять её значение.

В случае, если необходимо изменить значение какой-либо внешней переменной, передаётся указатель на неё

void make\_zero (int \*num)

{

\*num = 0;

}

int num;

make\_zero(&num);

// теперь num == 0

Кроме того, указатели можно использовать, чтобы вернуть несколько значений из функции. В таком случае в функцию передаётся несколько указателей на переменные, а в самой функции меняется их значение.

void increase (int \*one, int \*two)

{

\*one += 1;

\* two += 1;

}

***Указатель*** — переменная, содержащая адрес объекта. Указатель не несет информации о содержимом объекта, а содержит сведения о том, где размещен объект.

***Тип указателя*** — это тип переменной, адрес которой он содержит.

Для работы с указателями в Си определены две операции:

* операция \* (звездочка) — позволяет получить значение объекта по его адресу — определяет значение переменной, которое содержится по адресу, содержащемуся в указателе;
* операция & (амперсанд) — позволяет определить адрес переменной.

# Вопрос 19. Рекурсии

Определение, преимущества/недостатки, хвостовая рекурсия

Рекурсия, определение, пример, особенности выполнения. Преимущества и недостатки. Хвостовая рекурсия - отдельный вид рекурсии, чем он хорош

**Ответ**

***Определение рекурсии***

Функция называется *рекурсивной*, если она вызывает саму себя.

***Пример рекурсии***

Например, следующая функция рекурсивно вычисляет факториал, используя формулу  
n! = n \* (n - 1)!, 0!=1

Отобразим последовательность запусков при вызове функции fact(3):

fact(3) = 3 \* fact(2) // приостановка выполнения fact(3)

fact(2) = 2 \* fact(1) // приостановка выполнения fact(2)

fact(1) = 1 \* fact(0) // приостановка выполнения fact(1)

fact(0) = 1

fact(1) = 1 // возобновление выполнения fact(1)

fact(2) = 2 // возобновление выполнения fact(2)

fact(3) = 6 // возобновление выполнения fact(3)

Следующий хороший пример рекурсии — это быстрая сортировка, предложенная Ч. А. Р. Хоаром в 1962 г. Для заданного массива выбирается один элемент, который разбивает остальные элементы на два подмножества — те, что меньше, и те, что не меньше него. Та же процедура рекурсивно применяется и к двум полученным подмножествам. Если в подмножестве менее двух элементов, то сортировать нечего, и рекурсия завершается.

***Преимущества и недостатки***

Преимущества:

* Рекурсивная форма может быть структурно проще и нагляднее, в особенности, когда сам реализуемый алгоритм рекурсивен.

Недостатки:

* Рекурсивный вызов использует больше памяти, поскольку создает свой набор переменных.
* Рекурсия выполняется медленней, поскольку на каждый вызов функции требуется определенное время.

Рекурсивная программа не обеспечивает ни экономии памяти, поскольку требуется где-то поддерживать стек значений, подлежащих обработке, ни быстродействия; но по сравнению со своим нерекурсивным эквивалентом она часто короче, а часто намного легче для написания и понимания. Такого рода программы особенно удобны для обработки рекурсивно определяемых структур данных вроде деревьев.

***Хвостовая рекурсия***

*Хвостовая рекурсия* - частный случай рекурсии, при котором любой рекурсивный вызов является последней операцией перед возвратом из функции.

Подобный вид рекурсии может быть легко заменён на итерацию путём формальной и гарантированно корректной перестройки кода функции. Оптимизация хвостовой рекурсии путём преобразования её в итерацию реализована во многих оптимизирующих компиляторах.

*Итерация* в программировании — организация обработки данных, при которой действия повторяются многократно, не приводя при этом к вызовам самих себя (в отличие от рекурсии).

***Пример хвостовой рекурсии***

int fac\_times (int n, int acc) {

**return** (n==0) ? acc : fac\_times(n - 1, acc \* n);

}

int factorial (int n) {

**return** fac\_times (n, 1);

}

***Пример нехвостовой рекурсии***

Стоит отметить, что другой, более естественный рекурсивный способ вычисления [факториала](https://dic.academic.ru/dic.nsf/ruwiki/8486), приведённый ниже, не является хвостово-рекурсивным, так как в каждом вызове функции после рекурсивного вызова производятся дополнительные операции, а именно умножение на n.

int factorial (int n) {

**return** (n == 0) ? 1 : n \* factorial(n - 1);

}

***Преимущества хвостовой рекурсии***

Хвостовая рекурсия не вызывает проблем с глубиной стека, связанных с обычной рекурсией. Поскольку хвостовые рекурсии непосредственно возвращают результат родительскому объекту, кадр стека отбрасывается при следующем рекурсивном вызове. Это позволяет проводить гораздо более глубокую (бесконечную) рекурсию. Каждый внутренний шаг рекурсивного вызова по существу заменяется новым вызовом.

Нетрудно видеть, что необходимость расширения стека при рекурсивных вызовах диктуется требованием восстановления состояния вызывающего экземпляра функции (то есть её параметров, локальных данных и адреса возврата) после возврата из рекурсивного вызова. Но если рекурсивный вызов является последней операцией перед выходом из вызывающей функции и результатом вызывающей функции должен стать результат, который вернёт рекурсивный вызов, сохранение контекста уже не имеет значения — ни параметры, ни локальные переменные уже использоваться не будут, а адрес возврата уже находится в стеке. Поэтому в такой ситуации вместо полноценного рекурсивного вызова функции можно просто заменить значения параметров в стеке и передать управление на точку входа. До тех пор, пока исполнение будет идти по этой рекурсивной ветви, будет, фактически, выполняться обычный цикл. Когда рекурсия завершится (то есть исполнение пройдёт по терминальной ветви и достигнет команды возврата из функции) возврат произойдёт сразу в исходную точку, откуда произошёл вызов рекурсивной функции. Таким образом, при любой глубине рекурсии стек переполнен не будет. (wiki)

# Вопрос 20. Одномерный статический массив

Как описывается, инициализация, передача в функцию, указатели, индексация

Одномерный статический массив, как описывается, операция индексации, особенности передачи в функцию, связь массива и указателя, адрес в массиве

При решении задач с большим количеством данных одинакового типа использование переменных с различными именами, не упорядоченных по адресам памяти, затрудняет программирование. В подобных случаях в языке Си используют объекты, называемые массивами.

***Массив*** — это непрерывный участок памяти, содержащий последовательность объектов одинакового типа, обозначаемый одним именем.

Массив характеризуется следующими основными понятиями:

***Элемент массива*** (значение элемента массива) – значение, хранящееся в определенной ячейке памяти, расположенной в пределах массива, а также адрес этой ячейки памяти.

Каждый элемент массива характеризуется тремя величинами:

* адресом элемента — адресом начальной ячейки памяти, в которой расположен этот элемент;
* индексом элемента (порядковым номером элемента в массиве);
* значением элемента.

***Адрес массива*** – адрес начального элемента массива.

***Имя массива*** – идентификатор, используемый для обращения к элементам массива.

***Размер массива*** – количество элементов массива

***Размер элемента*** – количество байт, занимаемых одним элементом массива.

***Объявление и инициализация массивов***

Для объявления массива в языке Си используется следующий синтаксис:

**тип имя[размерность]={инициализация};**

***Инициализация*** представляет собой набор начальных значений элементов массива, указанных в фигурных скобках, и разделенных запятыми.



Если количество инициализирующих значений, указанных в фигурных скобках, меньше, чем количество элементов массива, указанное в квадратных скобках, то все оставшиеся элементы в массиве (для которых не хватило инициализирующих значений) будут равны нулю. Это свойство удобно использовать для задания нулевых значений всем элементам массива.

Если массив проинициализирован при объявлении, то константные начальные значения его элементов указываются через запятую в фигурных скобках. В этом случае количество элементов в квадратных скобках может быть опущено.

При обращении к элементам массива индекс требуемого элемента указывается в квадратных скобках [].

***Передача массива в функцию***

Обработку массивов удобно организовывать с помощью специальных функций. Для обработки массива в качестве аргументов функции необходимо передать

* адрес массива,
* размер массива.

Исключение составляют функции обработки строк, в которые достаточно передать только адрес.

При передаче переменные в качестве аргументов функции данные передаются как копии. Это означает, что если внутри функции произойдет изменение значения параметра, то это никак не повлияет на его значение внутри вызывающей функции.

Если в функцию передается адрес переменной (или адрес массива), то все операции, выполняемые в функции с данными, находящимися в пределах видимости указанного адреса, производятся над оригиналом данных, поэтому исходный массив (или значение переменной) может быть изменено вызываемой функцией.

Пусть есть массив

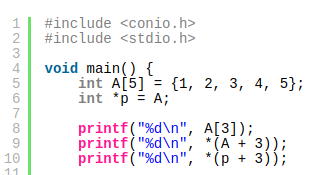
int A[5] = {1, 2, 3, 4, 5};

массив хранит адрес, откуда начинаются его элементы. Используя указатель можно также получить доступ до элементов массива

int \*p = A;

тогда вызов A[3] эквивалентен вызову \*(p + 3)

На самом деле оператор [ ] является синтаксическим сахаром – он выполняет точно такую же работу. То есть вызов A[3] также эквивалентен вызову \*(A + 3)



# Вопрос 21. Указатели, объявления …

\*организация памяти, зачем в си указатели, инициализация, базовые операции, const

Указатели, объявления, базовые операции

Организация памяти, определения, для чего в языке Си, инициализация указателя, оператор const

Указатель — переменная, содержащая адрес объекта. Указатель не несет информации о содержимом объекта, а содержит сведения о том, где размещен объект.

Указатель, как и любая переменная, должен быть объявлен.

Общая форма объявления указателя.



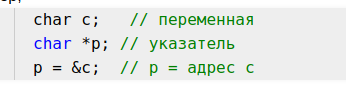
Тип указателя— это тип переменной, адрес которой он содержит.

Для работы с указателями в Си определены две операции:

операция \* (звездочка) — позволяет получить значение объекта по его адресу — определяет значение переменной, которое содержится по адресу, содержащемуся в указателе;

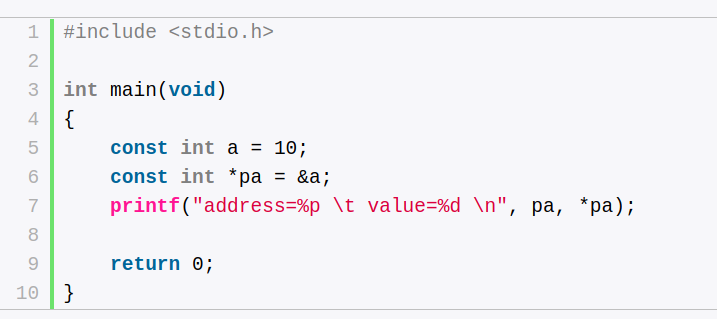
операция & (амперсанд) — позволяет определить адрес переменной.

Например,



Указатели нужны, чтобы напрямую обращаться к памяти компютера и иметь возможность оперировать ей без посредников.

Кроме переменных в программе на Си для хранения данных могут использоваться константы, которые предваряются ключевым словом const, и указатели также могут указывать на константы, но в этом случае перед определением указателя также ставится слово const:



# Вопрос 22. Адресная арифметика

\*все про указатели

**Ответ**

Указатели в си играют важную роль.

***Указатель*** – Переменная, содержащая адрес другой переменной. Для различных типов данных, надо использовать различные указатели.

Объявление указателя похоже на инициализацию обычной переменной, отличительной особенностью является наличие пунктуатора «\*».

Пример:

Int a = 4;

Int \*p = &a;

Указатель p теперь указывает на адрес, в котором размещено значение переменной a.

Теперь мы можем изменить значение переменной a, используя указатель p.

\*p = 6; // Теперь значение a == 6

При работе с указателями важно знать два унарных оператора

<\*> - оператор разыменования(извлечение значения по адресу)

<&>- оператор извлечения адреса переменной.

С помощью этих операторов мы можем изменять значения переменных, на которых ссылается указатель, а также изменять место, на которое указатель “указывает”.

При использовании массивов, мы также используем указатели:

int array[5] = {6, 7, 8, 9, 10};

array[2] = 12;

Запись array[2] === \*(array + 2). Следует помнить, что название массива распадается в указатель, на его первый элемент. А прибавление/вычитание к/из массиву целого числа имеет следующий смысл.

array + 2 === array + sizeof (int) \* 2 (т.к элементы массива расположены последовательно, от первого к последнего, сдвинув наш указатель от 0 на два вправо, получим третий элемент).

Указатели на указатель.

Так как возможно инициализировать указатель, то под него выделяется память. Если память выделяется, то на эту память можно указать. Именно таким смыслом обладает указатель на указатель. Он указывает на область памяти, в которой находится указатель:

int digit = 4;

int \*p = &digit;

int \*\*point\_to\_p = &p;

Теперь point\_to\_p указывает на адрес, в котором расположен указатель p, для получения значения digit, можно поступить следующим образом:

printf(“%d”,\*\*point\_to\_p);

\*point\_to\_p == p, следовательно \*\*point\_to\_p === \*p === digit.

Указатели на указатель используются при объявлении многомерного массива.

int arr[2][2] = {1, 2, 3, 4};

arr[1] в данном случае – указатель на два элемента типа int.

Его можно заменить на выражение \*(arr + 1)

Соответственно для получения значения 4 (arrr[1][1]), можно использовать:

\*(\*(arr + 1) + 1)

Используя массив 3 на n получим указатель на указатель на указатель и т.д.

# Вопрос 23. Многомерный статический массив

\*описание, инициализация, индексация, компоненты многомерного массива, в си - нет отдельного типа, по сути - это одномерный массив

Многомерный статический массив

Индексация, компоненты многомерного массива, в си нет отдельного типа, это по сути одномерный массив

**Ответ**

***«Концепция» многомерного массива в языке Си.***

В языке Си нет специального типа для описания многомерных массивов. Массив размерности “n” рассматривается как одномерный массив, элементами которого являются массивы размерности “n – 1”. Количество размерностей многомерного массива практически не ограничено (оно определяется реализацией компилятора).

***Описание многомерного массива на языке Си. Особенности расположения в памяти.***

№1Память под статические массивы выделяется во время компиляции. Именно поэтому

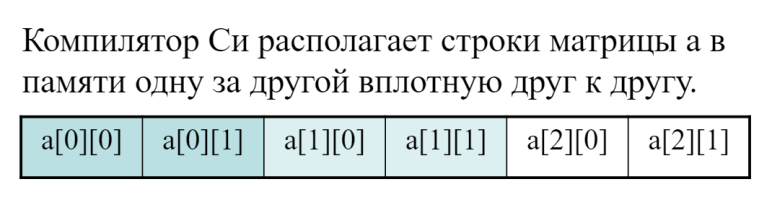
1) для задания статического массива нельзя использовать переменные, даже константные (значение переменных становится «известно» во время выполнения программы);

2) размер статического массива не может быть изменен во время работы программы.

№2Рассуждать можно следующим образом:

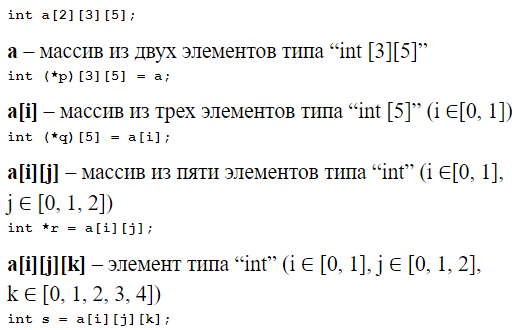
1) матрица – это одномерный массив, элементами которого являются одномерные массивы (строки);

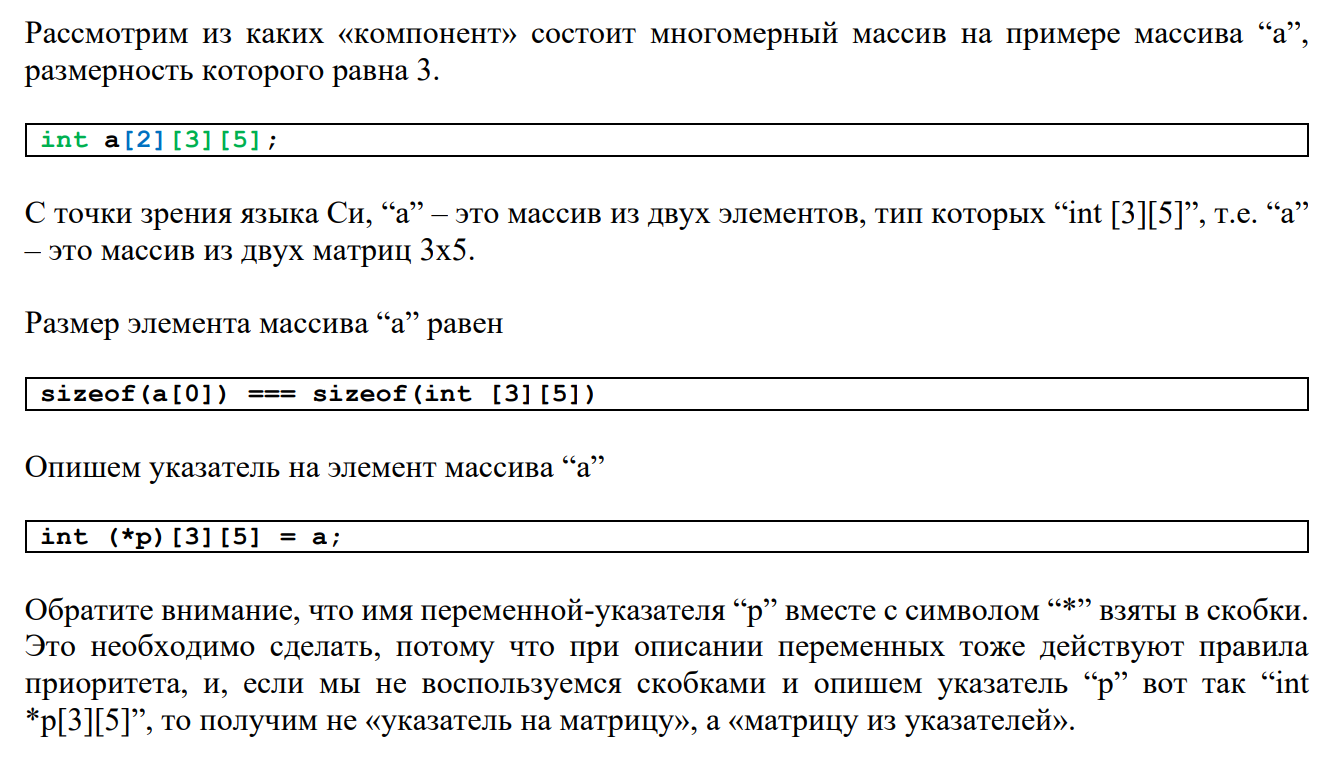
2) массив – последовательность элементов одного и того же типа, расположенных в памяти друг за другом. Что и требовалось получить :)



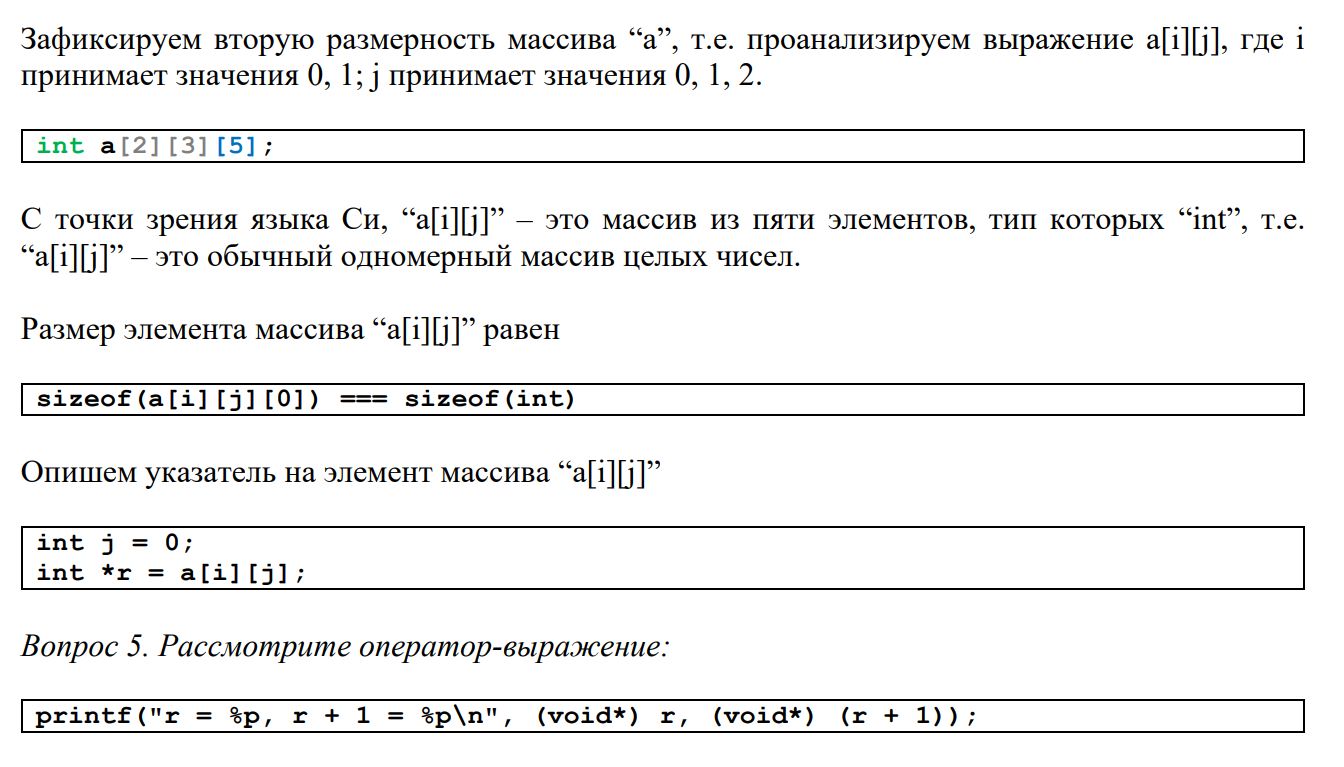
На слайде приведено определение двумерного массива “a” (или как мы привыкли называть такой массив - матрицы). При описании многомерного массива каждая размерность указывается в отдельных квадратных скобках. Значение размерности, как и в случае одномерного статического массива, задается константным выражением. Это выражение компилятор должен уметь вычислить во время компиляции (вопрос 1: почему?). В нашем случае описана матрица из 3-х строк и 2-х столбцов. В памяти массив “a” будет располагаться по строкам (вопрос 2: почему?): сначала первая строка, затем вторая (сразу же за первой) и, наконец, третья (сразу же за второй)

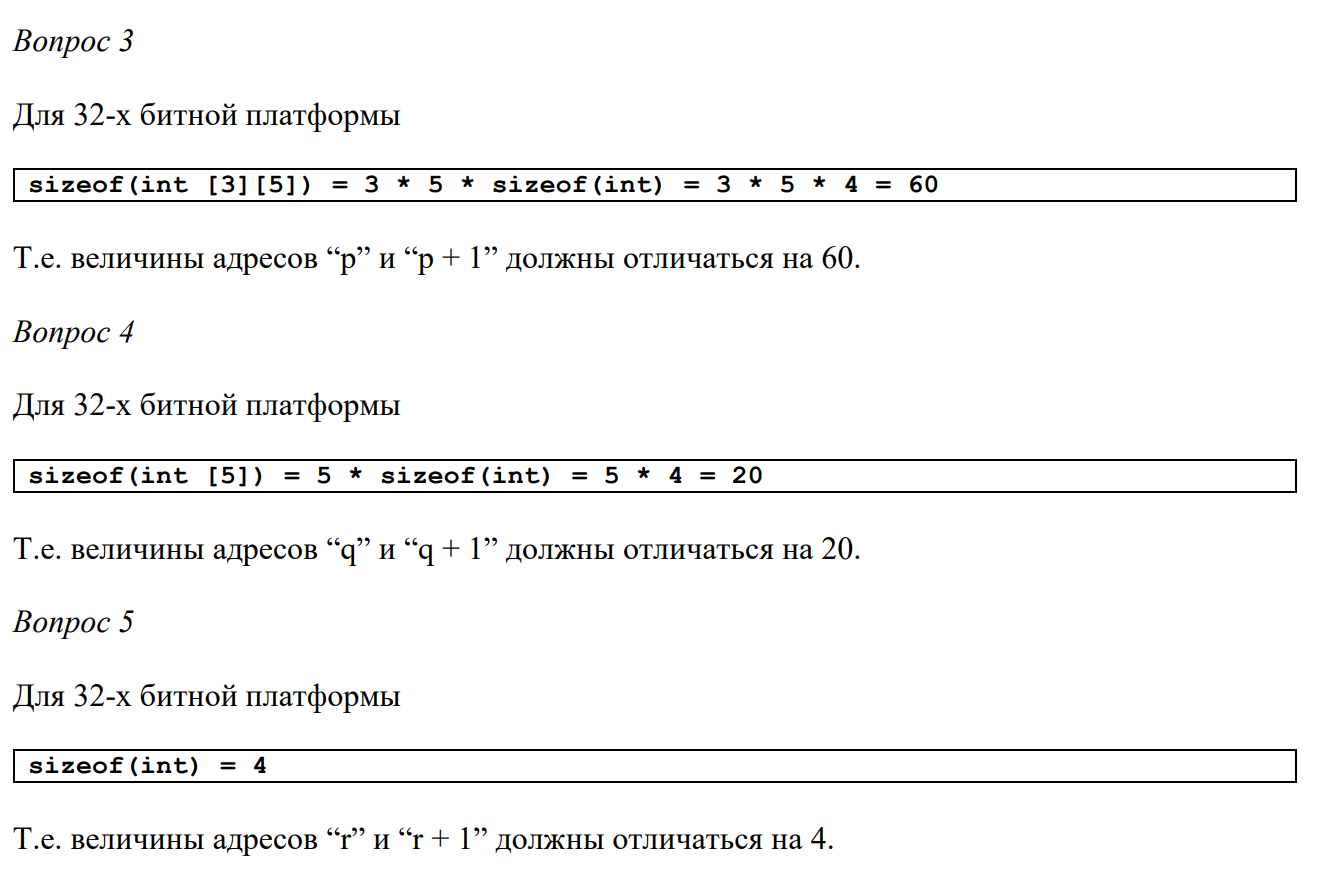
***«Слои» многомерного массива в языке Си.***



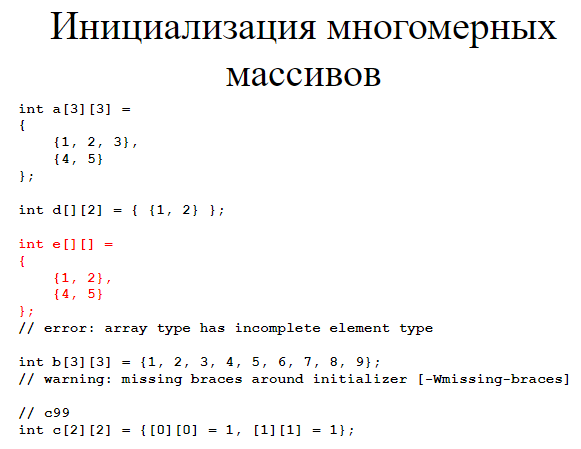








***Инициализация многомерных массивов на языке Си.***



Многомерные массивы можно инициализировать точно так же, как и одномерные. При инициализации многомерных массивов нужно указывать не один список инициализации, а несколько (см. первый пример на слайде).

Элементы, которым «не досталось» инициализирующего значения, инициализируются значением 0 (в первом примере элемент матрицы a[1][2] и вся ее третья строка будут проинициализированы значением 0).

При инициализации одномерного массива можно вообще не указывать количество элементов в нем (Компилятор сам посчитает количество элементов в списке инициализации и создаст массив ровно на такое количество элементов). В случае многомерных массивов можно не указывать значение только самой первой (старшей) размерности (см. второй пример на слайде).

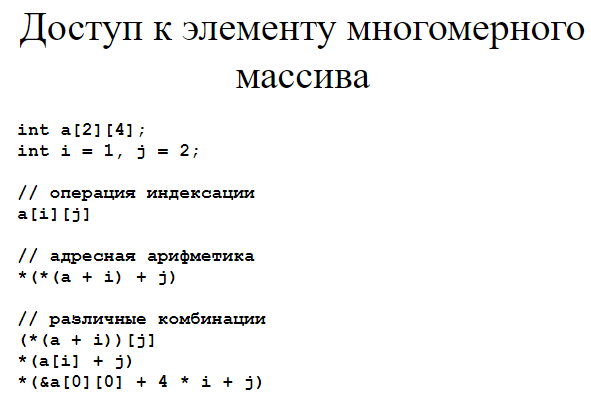
Если не указать значения всех размерностей, возникнет ошибка компиляции (см. третий пример на слайде): в этом случае компилятор не может ни выделить память под массив, ни сгенерировать код для доступа к элементу массива.

Инициализирующие значения для многомерного массива можно указать в одном списке инициализации (см. четвертый пример). Правда, в этом случае компилятор выдает предупреждение, которое из-за ключа “-Werror” случае превратится в ошибку компиляции. ПОМНИТЬ, ЧТО МАТРИЦА ЭТО ОДНОМЕРНЫЙ МАССИВ В ПАМЯТИ

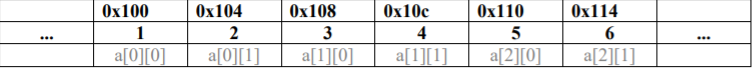
При использовании стандарта c99 можно использовать выделенные инициализаторы (см. пятый пример на слайде): в этом случае необходимо указать значения всех индексов инициализируемого элемента.

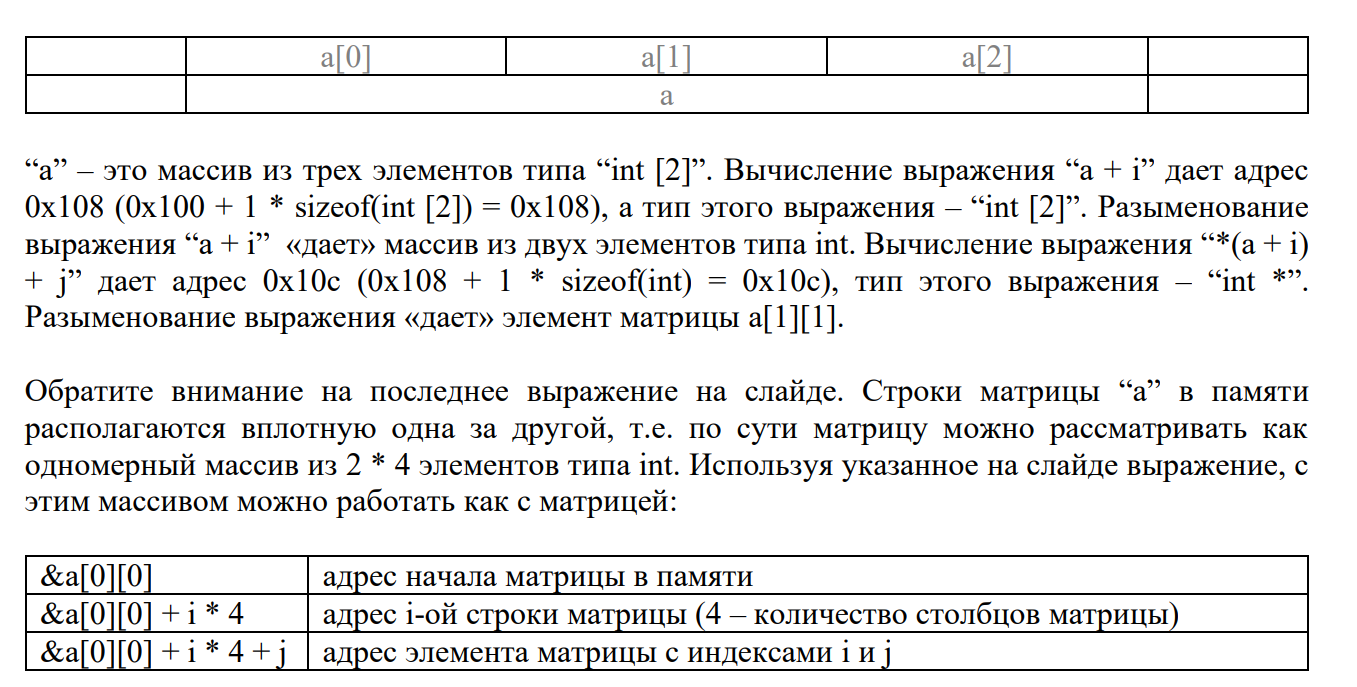
Для int b[3][3] 

***Доступ к элементу многомерного массива в языке Си.***



a[3][2] = {{1, 2}, {3, 4}, {5, 6}}





“a” – это массив из трех элементов типа “int [2]”.

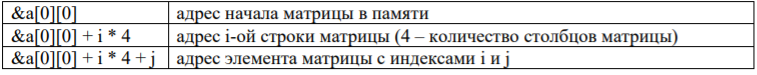
Вычисление выражения “a + i” дает адрес 0x108 (0x100 + 1 \* sizeof(int [2]) = 0x108), а тип этого выражения – “int [2]”.

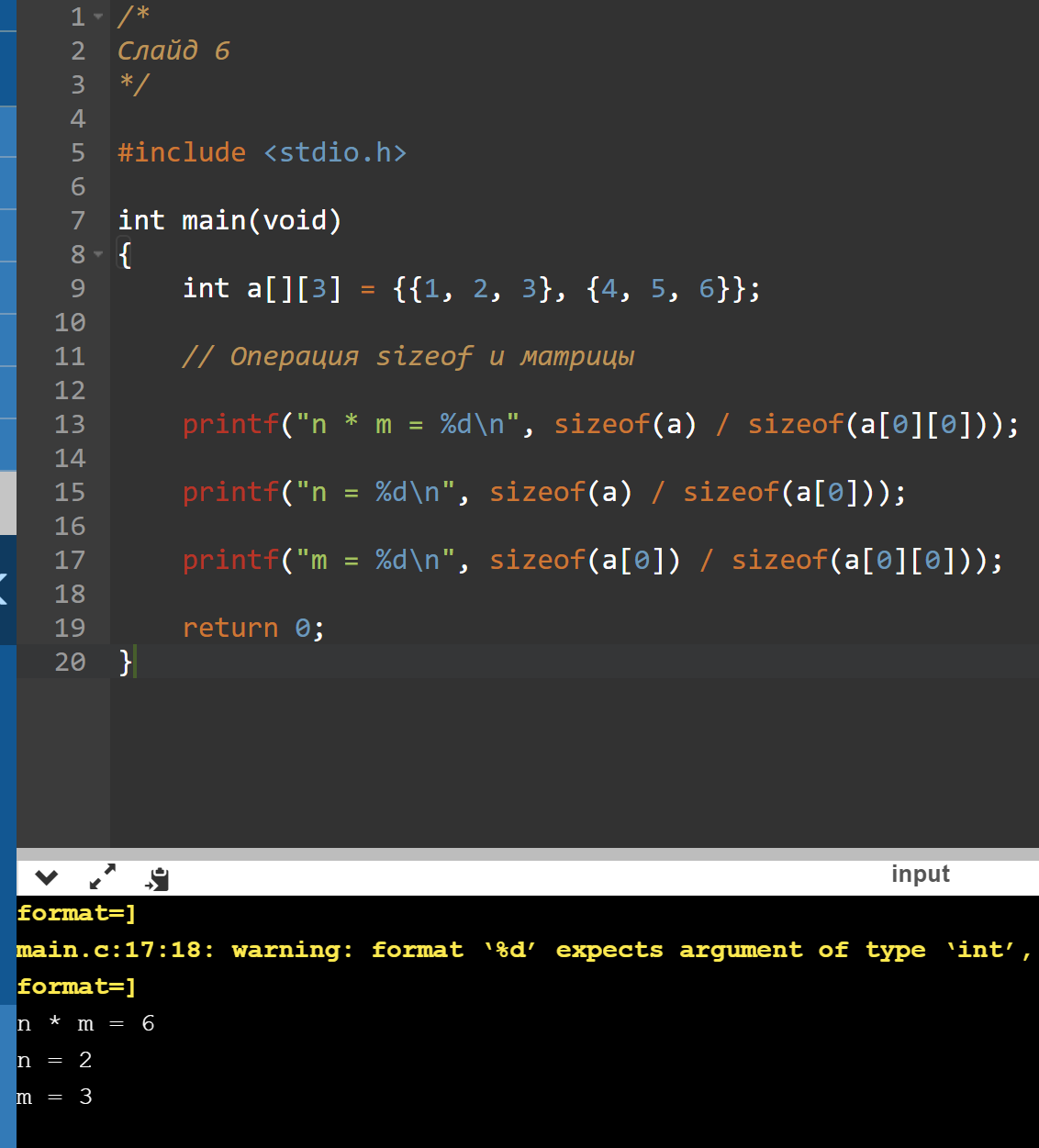
Разыменование выражения “a + i” «дает» массив из двух элементов типа int.

Вычисление выражения “\*(a + i) + j” дает адрес 0x10c (0x108 + 1 \* sizeof(int) = 0x10c), тип этого выражения – “int \*”.

Разыменование выражения «дает» элемент матрицы a[1][1].

Обратите внимание на последнее выражение на слайде. Строки матрицы “a” в памяти располагаются вплотную одна за другой, т.е. по сути матрицу можно рассматривать как одномерный массив из 2 \* 4 элементов типа int. Используя указанное на слайде выражение, с этим массивом можно работать как с матрицей:





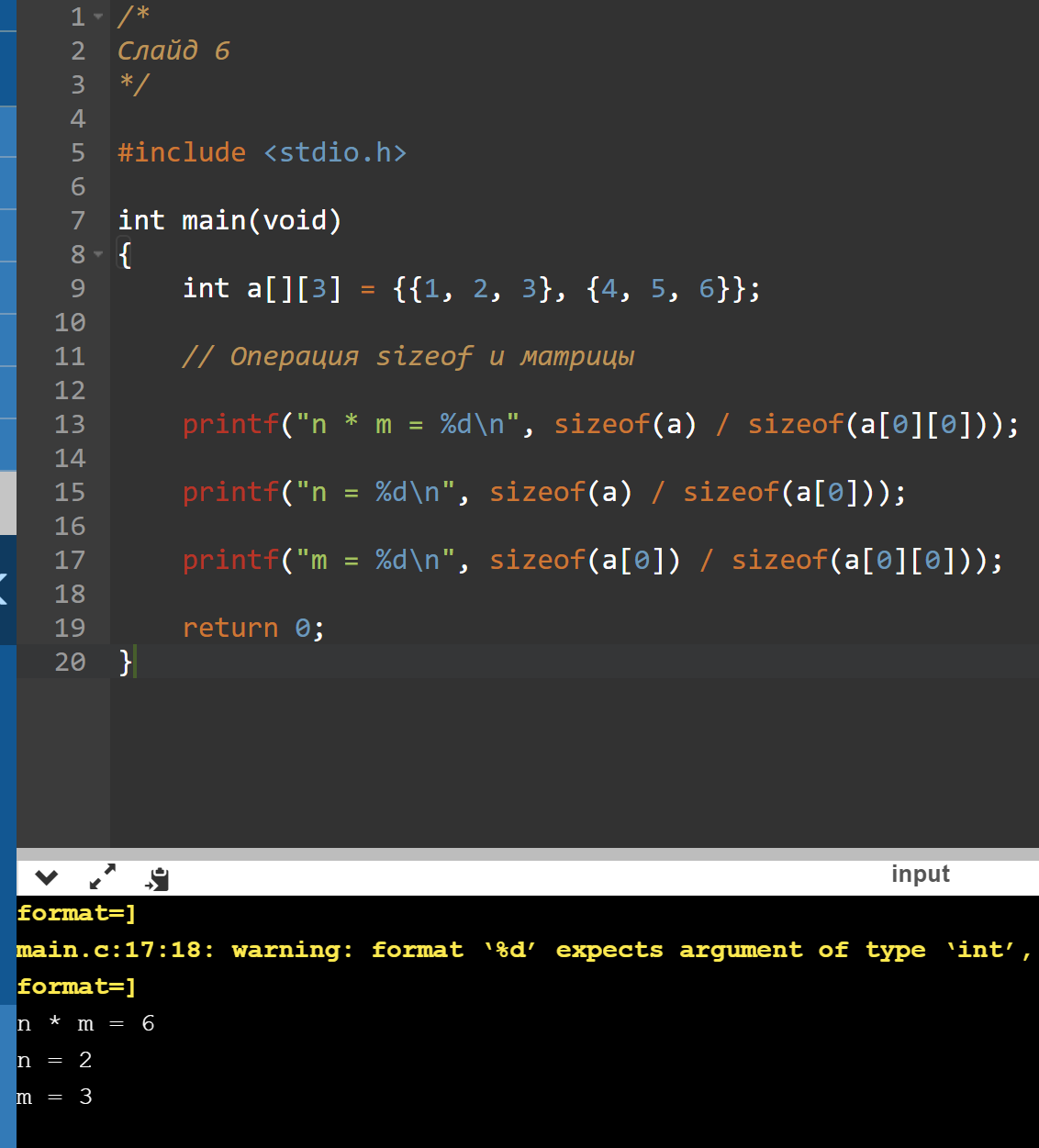
# Вопрос 24. Многомерный массив особенности использования

\*sizeof, как обработать с помощью указателей, передача в функцию, const

size of, как обработать с указателями, особенности передачи в функцию. Модификаторы conts для массивов, преобразователь типа - почему запрещено использовать в массиве

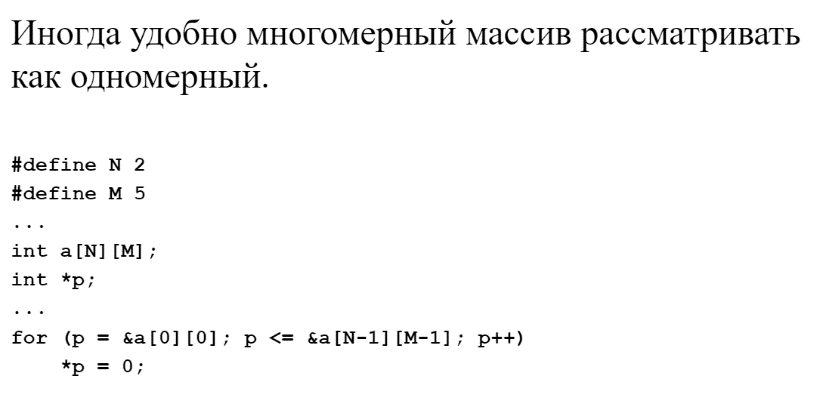
**Ответ**

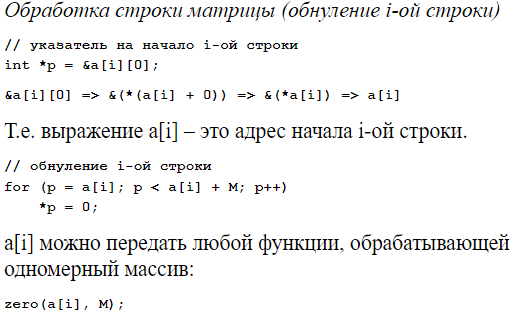
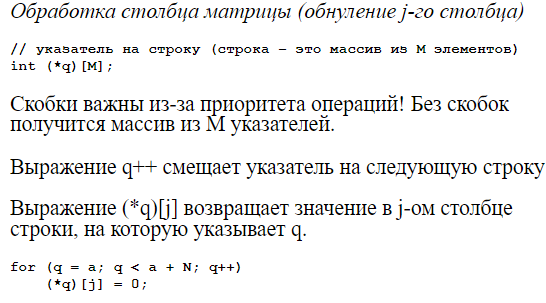
***Работа с sizeof***



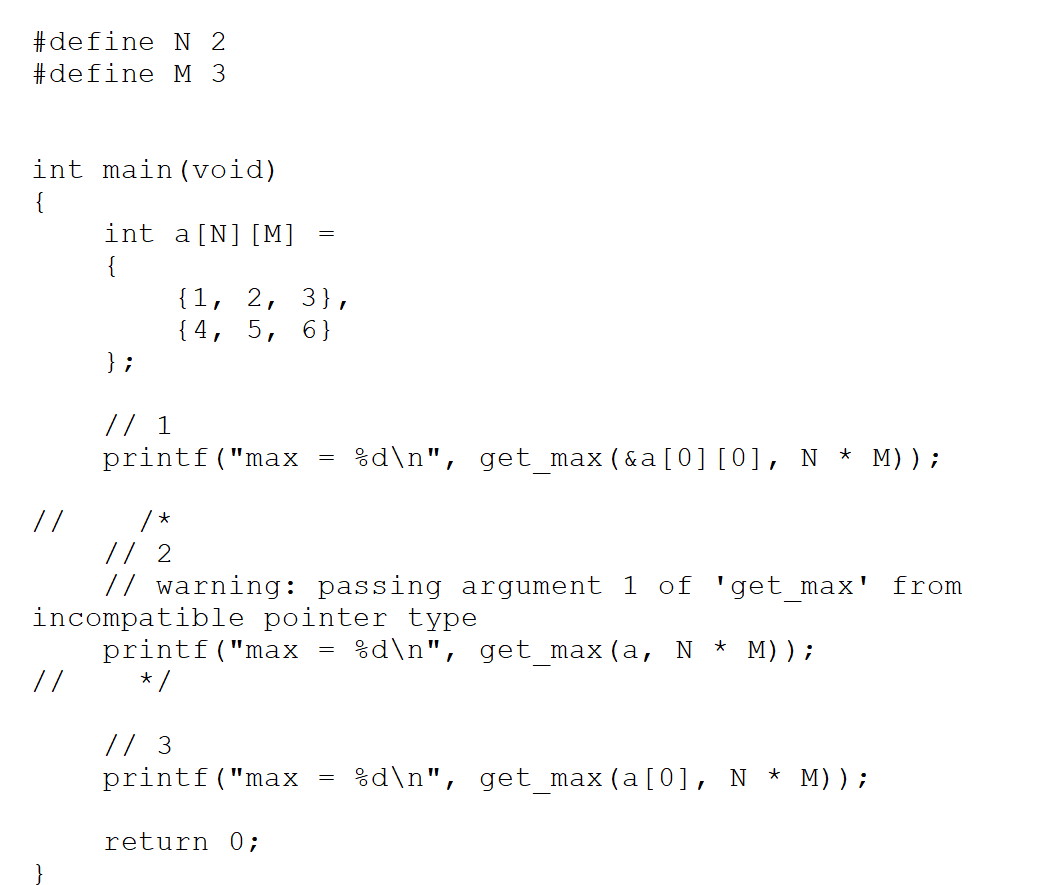
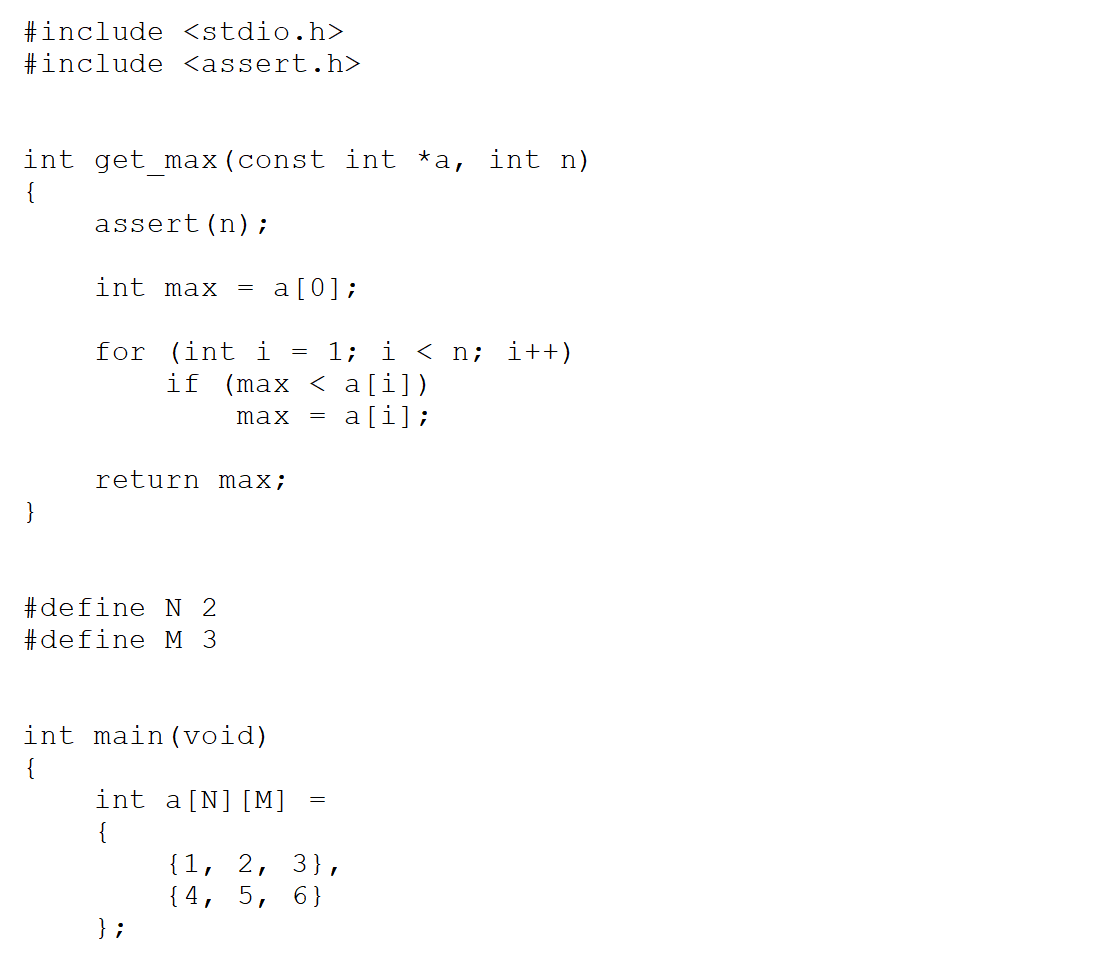
***Обработка многомерных массивов с помощью указателей.***

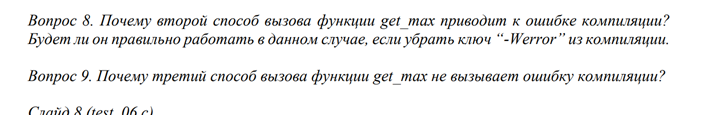
Как уже говорилось, статическую матрицу “a” можно рассматривать как одномерный массив. Этот факт удобно использовать, если нужно обработать все элементы матрицы (например, найти максимальный элемент во всей матрице) и уже есть функция, которая делает это для одномерного массива.

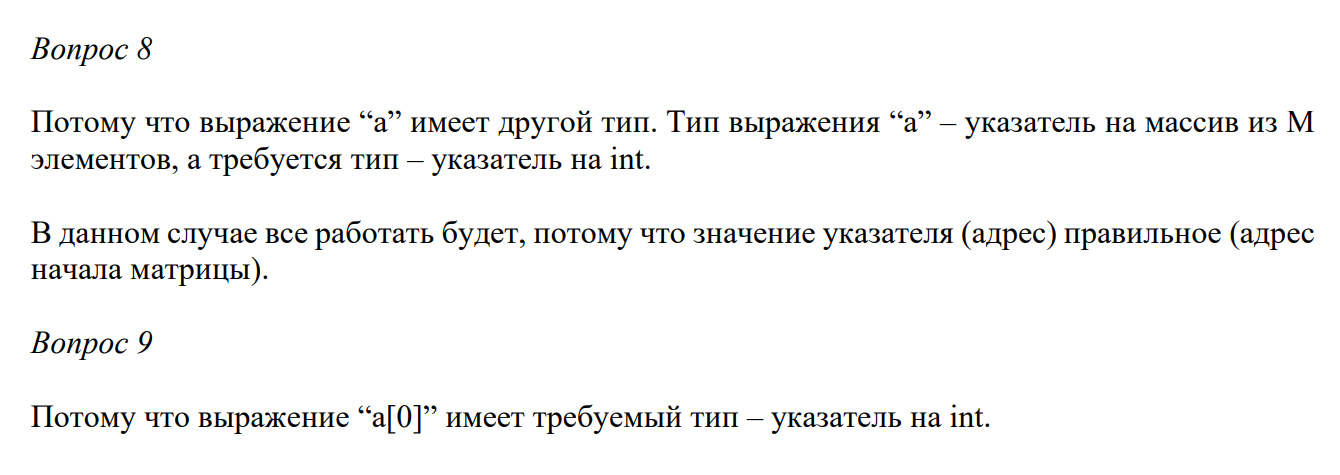


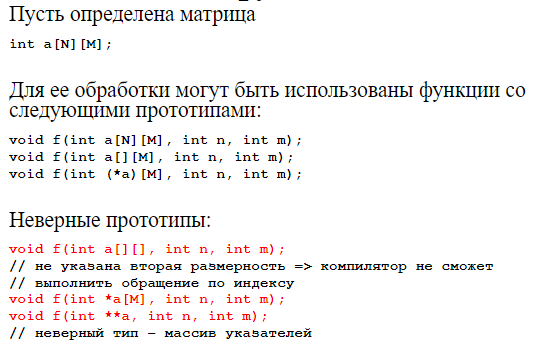
Для обработки столбца Сначала необходимо определиться с типом указателя. В отличие от предыдущего случае элементы, которые необходимо обнулить, расположены не подряд, в разных строках. Нам требуется указатель, при прибавлении 1 к которому, мы бы перешли к следующей строке, т.е. нам нужен указатель на строку.

Применив операцию разыменования к такому указателю “\*q”, мы получим строку, т.е. одномерный массив. Примениd к этому массиву операцию индексации “(\*q)[j]” мы получим элемент j-го столбца, который располагается в строке, на которую указыва





***Передача многомерного массива в функцию.***



Чтобы понять как многомерный массив передать в функцию, нужно вспомнить, что

1. многомерный массив – это одномерный массив, элементами которого являются массивы меньшей размерности;
2. . одномерный массив передается в функцию по указателю.

***Особенности использования const и многомерных массивов в языке Си.***

Массивы любой размерности передаются в функцию по указателю. Это означает, что функция может изменить массив. В случае одномерного массива, чтобы подстраховаться от случайного изменения массива, мы использовали модификатор const. Например,

// функция вывода целочисленного массива

void arr\_print(const int \*arr, int n);

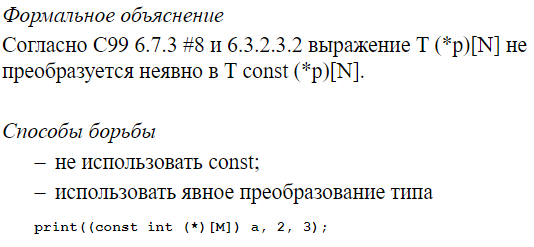
// Вариант 1 (ОШИБКА компиляции)

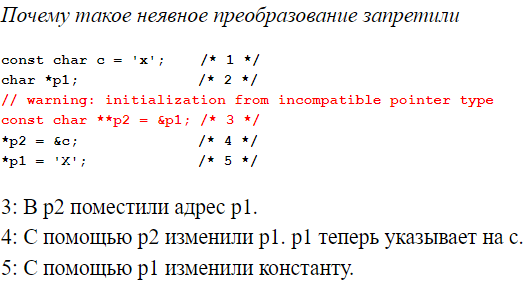
// warning: expected 'const int (\*)[5]' but argument is of type 'int (\*)[5]'

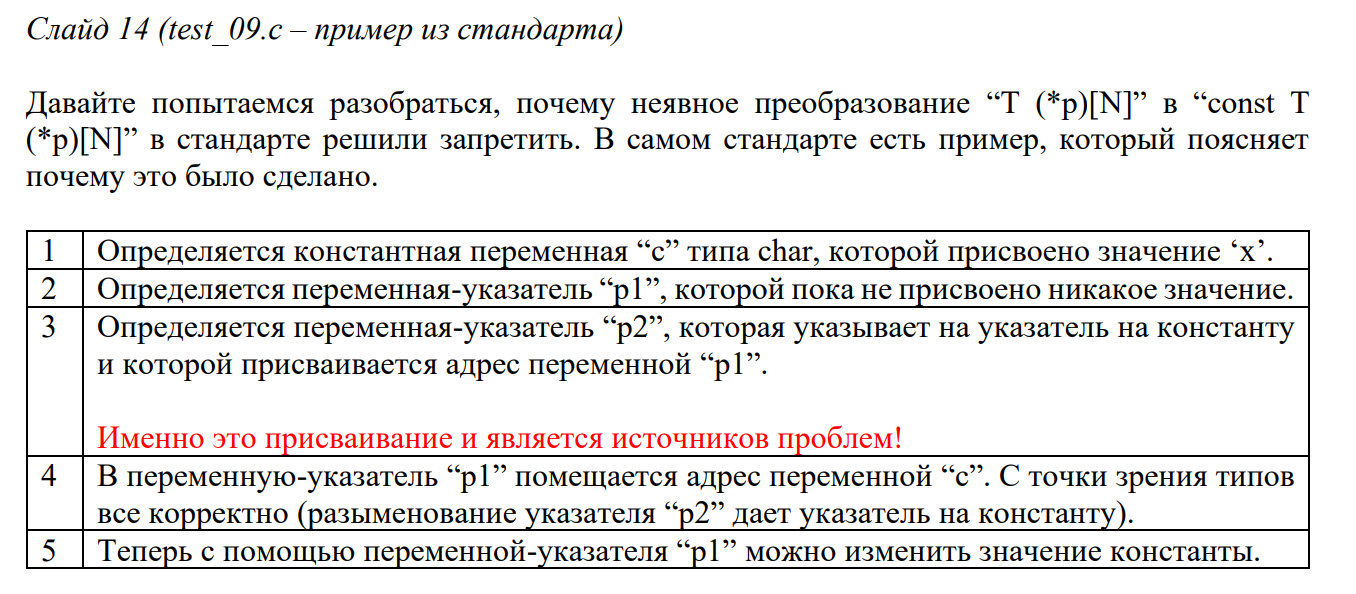
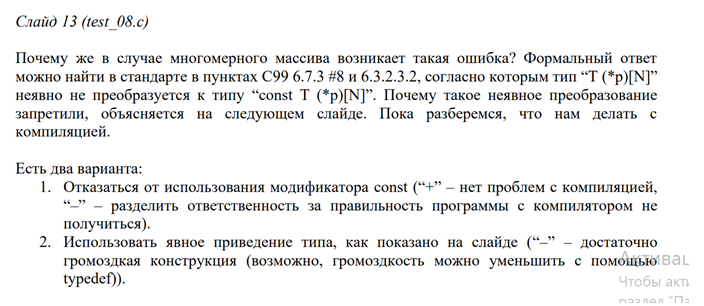
print(a, 2, 3);

// Вариант 2 (использование явного преобразования типа для успешной компиляции)

print((const int (\*)[M]) a, 2, 3);



НЕВОЗМОЖНО НЕЯВНОЕ ПРЕОБРАЗОВАНИЕ ИЗ-ЗА  




# Вопрос 25. Строка

\*объявление, инициализация, строковые литералы, ввод/вывод, gets-не юзабельная, аналогично-scanf, какие траблы у них, fgets-збс

Строка, что такое, показать как описывается, строковые литералы, инициализация строк, ввод/вывод строк, почему gets не используется (проблема переполнения буфера), scanf аналогично. Какие есть особенности при вводе строк, как обезопасить при вводе

Описывать безопасный способ через fgets

Написать функцию самостоятельного ввода строки (см. лекцию)

**Ответ**

***Строка*** – это последовательность символов, заканчивающаяся и включающая первый нулевой символ (англ., null character ‘\0’ (символ с кодом 0)).

* Преимущества подхода
  + Простота.
* Недостатки подхода
  + Отсутствие быстрого способа определения длины строки.
  + Тщательность при работе с нулевым символом.

Для представления символов в языке Си используется тип char, а для представления последовательности символов – массив, тип элементов которого char.

Т.е. строка в языке Си – это массив, тип элементов которого char, и который обязательно содержит нулевой символ.

***Определение(объявление)*** переменной-строки, которая может содержать до 80 символов обычно выглядит следующим образом:

* #define STR\_LEN 80
* ...
* char str[STR\_LEN+1]; // !

Поскольку строка – массив символов, для доступа к элементу строки может использоваться операция индексации:

* int count\_spaces(const char \*s)
* {
* int count = 0;
* for (int i = 0; s[i] != '\0'; i++)
* if (s[i] == ' ')
* count++;
* return count;
* }

***Строковый литерал*** – последовательность символов, заключенных в двойные кавычки. [c99 6.4.5]

* char str[] = "String for test";
* printf("Max is %d\n", max);

/\* Строковый литерал рассматривается компилятором как массив элементов типа char. Когда компилятор встречает строковый литерал из n символов, он выделяет n+1 байт памяти, которые заполняет символами строкового литерала и завершает нулевым символом.

Массив, который содержит строковый литерал, существует в течение всего времени выполнения программы.

В стандарте сказано, что поведение программы не определено при попытке изменить строковый литерал.

Обычно строковые литералы хранятся в read only секции. \*/

***Инициализация строковых переменных***

* char str\_1[] = {‘J’,‘u’,‘n’, ‘e’, ‘\0’};
* char str\_2[] = "June";
* char str\_3[5] = "June";
* char str\_4[3] = "June";
* // error: initializer-string for array of chars is too long
* char str\_5[4] = "June";
* // str\_5 не строка!

Массив символов и указатель на строковый литерал

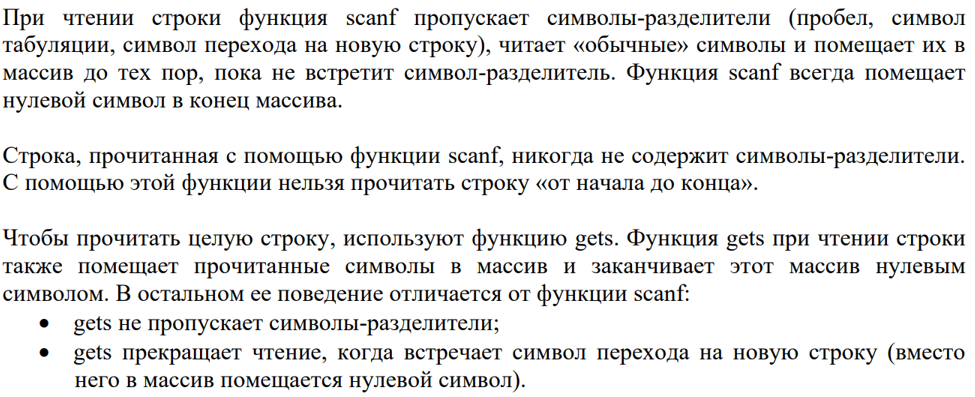
* // массив символов
* char str\_arr[] = "June";
* // указатель на строковый литерал
* char \*str\_ptr = "June";
* void process(const char \*str);
* str\_arr[0] = 'j'; // ок
* str\_ptr[0] = 'j'; // ошибка времени выполнения

***Вывод строк***

* #include <stdio.h>
* ...
* char str[] = "Hello, world!";
* printf("%s\n", str);
* puts(str);

***Ввод строк***

* #include <stdio.h>
* ...
* char str[10];
* scanf("%s", str);
* // Через scanf нельзя ввести строку с пробелами!
* gets(str);



Функции scanf и gets ***небезопасны и недостаточно*** гибки. Указанные функции не контролируют размер переданного массива и могут легко ***переполнить*** его. Иногда приходится реализовать свою собственную функцию для ввода строки, в основе которой лежит посимвольное чтение вводимой строки с помощью функции getchar.

***«Правильный» ввод строк :*** стандартная библиотека

* char \*fgets(char \*s, int size, FILE \*stream);

Прекращает ввод когда (любое из)

* прочитан символ ‘\n’;
* достигнут конец файл;
* прочитано size-1 символов.

Введенная строка всегда заканчивается нулем.

* fgets(str, sizeof(str), stdin);

Иногда приходится реализовать свою ***собственную функцию*** для ввода строки, в основе которой лежит посимвольное чтение вводимой строки с помощью функции getchar.

* #include <stdio.h>
* int getchar(void);

О чем следует задуматься:

* Должна ли функция пропускать «разделители»?
* Какой символ должен приводить к окончанию ввода? Этот символ должен включаться в строку?
* Что делать если строка слишком длинная?

Пример реализации (НЕ СПИСЫВАТЬ!!!!):

* int read\_line(char \*s, int n)
* {
* int ch, i = 0;
* while ((ch = getchar()) != '\n' && ch != EOF)
* if (i < n - 1)
* s[i++] = ch;
* s[i] = '\0';
* return i;
* }
* Параметры функции: s – массив, в котором сохраняются символы, n – размер этого массива.
* Функция возвращает количество символов, сохраненных в массиве.
* Символы, которые не помещаются в массив, игнорируются.

# Вопрос 26. Строка, функции стандартной библиотеки

\*про все основные функции, кроме strtok, определение строки

Стандартные библиотеки для работы со строками (кроме strtok)

25 вопрос + почему работа со строками выделено в библиотеку

**Ответ**

Строка – это последовательность символов, заканчивающаяся и включающая первый нулевой символ (англ., null character ‘\0’ (символ с кодом 0)).

· Преимущества подхода

o Простота.

· Недостатки подхода

o Отсутствие быстрого способа определения длины строки.

o Тщательность при работе с нулевым символом.

Для представления символов в языке Си используется тип char, а для представления последовательности символов – массив, тип элементов которого char.

Т.е. строка в языке Си – это массив, тип элементов которого char, и который обязательно содержит нулевой символ.

Определение(объявление) переменной-строки, которая может содержать до 80 символов обычно выглядит следующим образом:

· #define STR\_LEN 80

· ...

· char str[STR\_LEN+1]; // !

Поскольку строка – массив символов, для доступа к элементу строки может использоваться операция индексации:

· int count\_spaces(const char \*s)

· {

· int count = 0;

· for (int i = 0; s[i] != '\0'; i++)

· if (s[i] == ' ')

· count++;

· return count;

· }

Строковый литерал – последовательность символов, заключенных в двойные кавычки. [c99 6.4.5]

· char str[] = "String for test";

·

· printf("Max is %d\n", max);

/\* Строковый литерал рассматривается компилятором как массив элементов типа char. Когда компилятор встречает строковый литерал из n символов, он выделяет n+1 байт памяти, которые заполняет символами строкового литерала и завершает нулевым символом.

Массив, который содержит строковый литерал, существует в течение всего времени выполнения программы.

В стандарте сказано, что поведение программы не определено при попытке изменить строковый литерал.

Обычно строковые литералы хранятся в read only секции. \*/

Инициализация строковых переменных

· char str\_1[] = {‘J’,‘u’,‘n’, ‘e’, ‘\0’};

·

· char str\_2[] = "June";

·

· char str\_3[5] = "June";

· char str\_4[3] = "June";

· // error: initializer-string for array of chars is too long

· char str\_5[4] = "June";

· // str\_5 не строка!

Массив символов и указатель на строковый литерал

· // массив символов

· char str\_arr[] = "June";

· // указатель на строковый литерал

· char \*str\_ptr = "June";

·

· void process(const char \*str);

·

· str\_arr[0] = 'j'; // ок

· str\_ptr[0] = 'j'; // ошибка времени выполнения

Вывод строк

· #include <stdio.h>

· ...

· char str[] = "Hello, world!";

·

· printf("%s\n", str);

· puts(str);

Ввод строк

· #include <stdio.h>

· ...

· char str[10];

·

· scanf("%s", str);

· // Через scanf нельзя ввести строку с пробелами!

· gets(str);

Функции scanf и gets небезопасны и недостаточно гибки. Указанные функции не контролируют размер переданного массива и могут легко переполнить его. Иногда приходится реализовать

свою собственную функцию для ввода строки, в основе которой лежит посимвольное чтение вводимой строки с помощью функции getchar.

«Правильный» ввод строк : стандартная библиотека

· char \*fgets(char \*s, int size, FILE \*stream);

Прекращает ввод когда (любое из)

‒ прочитан символ ‘\n’;

‒ достигнут конец файл;

‒ прочитано size-1 символов.

Введенная строка всегда заканчивается нулем.

· fgets(str, sizeof(str), stdin);

Иногда приходится реализовать свою собственную функцию для ввода строки, в основе которой лежит посимвольное чтение вводимой строки с помощью функции getchar.

· #include <stdio.h>

· int getchar(void);

О чем следует задуматься:

‒ Должна ли функция пропускать «разделители»?

‒ Какой символ должен приводить к окончанию ввода? Этот символ должен включаться в строку?

‒ Что делать если строка слишком длинная?

Пример реализации (НЕ СПИСЫВАТЬ!!!!):

· int read\_line(char \*s, int n)

· {

· int ch, i = 0;

· while ((ch = getchar()) != '\n' && ch != EOF)

· if (i < n - 1)

· s[i++] = ch;

· s[i] = '\0';

· return i;

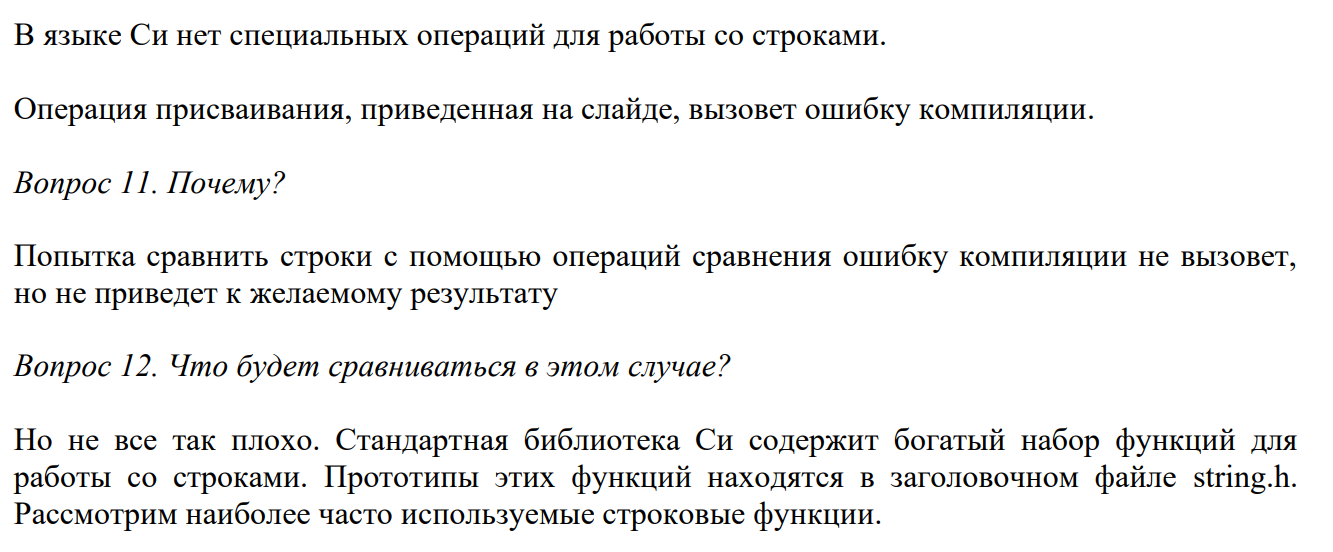
· }

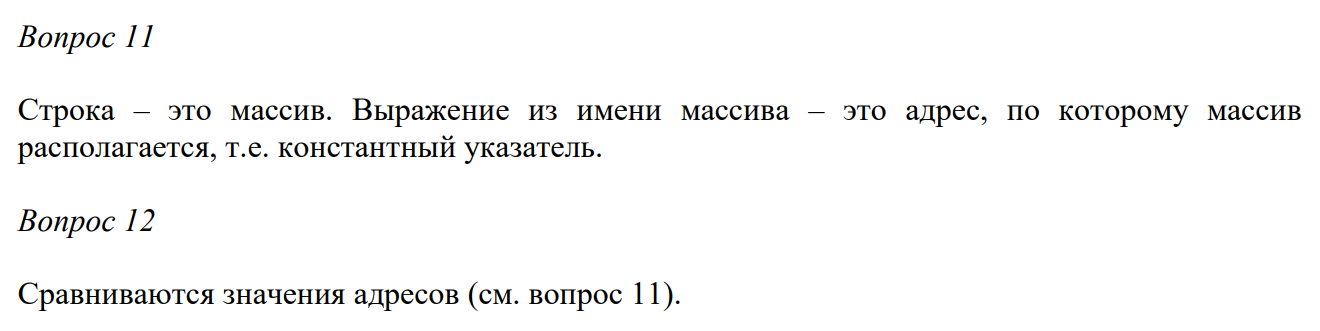
‒ Параметры функции: s – массив, в котором сохраняются символы, n – размер этого массива.

‒ Функция возвращает количество символов, сохраненных в массиве.

‒ Символы, которые не помещаются в массив, игнорируются.

***Обработка строк (strcpy, strcat, strlen, strcmp, snprintf, strtok, перевод строки в число).***





Правильнее было бы сказать «strcpy копирует строку, на которую указывает указатель s2 в массив, на который указывает указатель s1». Функция strcpy заменяет операцию присваивания для строк.

**char\* strcpy(char \*s1, const char \*s2);** // c99 7.21.2.3

**char\* strcpy(char \*s1, char \*s2);**

char src[] = "Hello!";

char dst[20];

strcpy(dst, src);

Недостаток функции strcpy состоит в том, что она не контролирует размеры строки, адресуемой указателем s2. Безопаснее пользоваться функцией strncpy.

**char\* strncpy(char \*s1, const char \*s2, size\_t n);**

strncpy(dst, src, sizeof(dst) - 1);

dst[sizeof(dst) - 1] = '\0';

Однако у функции strncpy есть неприятная особенность: нулевой символ не будет помещен в строку s1, если строка s2 больше или равна размеру массива s1. Поэтому функцию strncpy обычно используют следующим образом

char str\_long[] = "Very very long string...";

char dst[10];

strncpy(dst, src\_long, sizeof(dst) - 1);

dst[sizeof(dst) - 1] = '\0';

***ВНИМАНИЕ***

Верен ли следующий фрагмент программы. Объясните почему.

char \*str\_1 = "abcdef";

char \*str\_2 = "123";

strcpy(str\_1, str\_2);

Функция strcpy изменяет значение строки, на которую указывает str\_1, а str\_1 – это указатель 12 на строковый литерал, который изменять нельзя.

**size\_t strlen(const char \*s);**

**char dst[20];**

**size\_t len;**

**strcpy(dst, "Hello!");**

**len = strlen(dst); // len = 6, а не 20**

Функция strlen возвращает длину строки s, т.е. количество символов в стоке s до нулевого символа. Нулевой символ в это количество не входит.

Обратите внимание, что функция возвращает размер строки, а не размер массива, в котором эта строка располагается.

Функция strcat добавляет содержимое строки s2 в конец строки s1. Она возвращает указатель на строку s1. Обычно это значение не используется.

Функция strcat заменяет операцию конкатенации (сложения) для строк.

**char\* strcat(char \*s1, const char \*s2);** // c99 7.21.3.1

**char\* strcat(char \*s1, const char \*s2);** // c99 7.21.3.1

**char\* strcat(char \*s1, const char \*s2);**

char src[] = ", world.";

char dst[20] = "Hello";

strcat(dst, src);

Как и функция strcpy функция strcat не контролирует размер результирующей строки. Поэтому безопаснее использовать функцию strncat, которая, как и функция strncpy, ограничивает количество обрабатываемых символов. Функция strncat (в отличие от функции strncpy) добавляет нулевой символ в строку (только для него нужно не забыть предусмотреть место!).

**char\* strncat(char \*s1, const char \*s2, size\_t n);**

int len = (sizeof(des)- 1) - strlen(des);

strncat(des, src, len);

**ВНИМАНИЕ**

Верен ли следующий фрагмент программы. Объясните почему.

char str\_1[10];

char \*str\_2 = "123";

strcat(str\_1, str\_2);

Функция strcat «складывает» строки. Переменная str\_1 – это не строка (она не содержит нулевой символ).

Функция strcmp сравнивает строки s1 и s2

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

* значение < 0, если s1 меньше s2
* 0, если s1 равна s2
* значение > 0, если s1 больше s2

Строки сравниваются в лексикографическом порядке (как в словаре).

**int strncmp(const char \*s1, const char \*s2, size\_t n);**

Иногда спрашиваю почему функция strcmp возвращает «нечеткие» значения (например, если строка s1 меньше строки s2 можно было бы вернуть значение -1). На этот вопрос можно получить ответ, рассмотрев реализацию функции strcmp (см. test\_12.c) от создателей языка Си.

**int strcmp(char \*s, char \*t)**

{

    int i;

    for (i = 0; s[i] == t[i]; i++)

        if (s[i] == '\0')

            return 0;

    return s[i] - t[i];

}



// stdio.h, c99 7.19.6.6

int sprintf(char \*s, const char \*format, ...);

Функция sprintf объявлена в заголовочном файле stdio.h, а не в string.h. Она ведет себя подобно функции printf, но при этом выполняет запись в строку, а не выводит ее на экран. В качестве первого аргумента функция sprintf принимает адрес целевой строки. Остальные аргументы аналогичны аргументам функции printf – строка форматирования и список аргументов, которые эта функция должна записать.

// stdio.h, c99 7.19.6.5

int snprintf(char \*s, size\_t n, const char \*format, ...);

Вместо функции sprintf, которая не контролирует размер результирующей строки, лучше использовать функцию snprintf, которая сохраняет в целевой строке максимум n – 1 символов, а по окончании работы помещает в нее нулевой символ.

ВНИМАНИЕ

К сожалению, реализация стандартной библиотеки компилятора gcc в MSYS использует библиотеки времени выполнения от Microsoft, которые не реализую такое поведение (не гарантируют завершение строки нулевым символом).

**char\* strtok(char \*string, const char \*delim);**

char str\_test\_1[] = "   This   is a,,, test string!!!";

char \*pword = strtok(str\_test\_1, "\n ,.!?");

while (pword)

{

    printf("[%s]\n", pword);

    pword = strtok(NULL, "\n ,.!?");

}

Функция strtok выполняет поиск лексем (слов) в строке string. Последовательные вызовы этой функции разбивают строку string на лексемы. Лексема представляет собой последовательность символов, разделенных символами-разделителями, которые указываются в строке delim.

При первом вызове функция принимает строку string в качестве аргумента. В последующие вызовы функция ожидает нулевого

Функция возвращает указатель на найденную лексему. Если лексем не найдено, то возвращается нулевой указатель.

При каждом вызове strtok можно варьировать набор символов-разделителей delim

Довольно часто возникает задача перевода строки в число. Для ее решения в стандартной библиотеке есть два «семейства» функций (оба объявлены в заголовочном файле stdlib.h).

«Семейство atoi» На вход эти функции получают строку, а возвращают значение числа соответствующего типа. Основной недостаток функций этого семейства заключается в том, что в случае возникновения ошибки они возвращают значение 0.

// stdlib.h

// Семейство функций (atoi, atof, atoll), c99 7.20.1.2

**long int atol(const char\* str);**

«Семейство strtol» В отличие от функций «семейства atoi» эти функции не только могут учитывать разные системы счисления (для целых типов), но и возвращают указатель endptr на символ, на котором закончилось выделение числа. Этот указатель позволяет понять было ли выполнено преобразование правильно.

Обратите внимание на тип аргумента endptr. Этот аргумент имеет тип «указатель на указатель», потому что функции strtol нужно изменить значение указателя.

// stdlib.h

// Семейство функций (strtoul, strtoll, ...) ,c99 7.20.1.3

**long int strtol(const char\* string,**  **char\*\* endptr, int basis);**

ВНИМАНИЕ

Как с помощью параметра endptr можно понять, что преобразование строки в число было выполнено правильно?

Универсальный ответ дать нельзя, потому что нужно знать как организованы данные в программе.

Функция isdigit проверяет аргумент, передаваемый через параметр сharacter, является ли он десятичной цифрой. int isdigit( int character );

Значение, отличное от нуля (т.е. истинно), если аргумент функции — это десятичная цифра .

Ноль (т.е. ложь), в противном случае.

# Вопрос 27. Строки, способы хранения слов, функция strtok

\*как в программе хранится массив слов, алгоритм выделения слов, функция strtok

Функция strotok. Способы хранения строк, каким образом можно хранить массив слов (матрица и массив указателей), алгоритм выделения слов

Пример функции + функцию стандартной библиотеки. Особенности использования функции strtok

Алгоритм выделения слов на основе strtok

Пример

**Ответ**

***Strtok описание и алгоритм разбиения***

**char\* strtok(char \*string, const char \*delim);**

char str\_test\_1[] = "   This   is a,,, test string!!!";

char \*pword = strtok(str\_test\_1, "\n ,.!?");

while (pword)

{

    printf("[%s]\n", pword);

    pword = strtok(NULL, "\n ,.!?");

}

Функция strtok выполняет поиск лексем (слов) в строке string. Последовательные вызовы этой функции разбивают строку string на лексемы. Лексема представляет собой последовательность символов, разделенных символами-разделителями, которые указываются в строке delim.

При первом вызове функция принимает строку string в качестве аргумента. В последующие вызовы функция ожидает нулевого

Функция возвращает указатель на найденную лексему. Если лексем не найдено, то возвращается нулевой указатель.

При каждом вызове strtok можно варьировать набор символов-разделителей delim

Довольно часто возникает задача перевода строки в число. Для ее решения в стандартной библиотеке есть два «семейства» функций (оба объявлены в заголовочном файле stdlib.h).

***Способы описания «массива строк» на языке Си.***



Какой способ лучше выбрать для хранения массива строк?

Очевидное решение – создать массив строк, элементами которого будут строки (т.е. фактически описать двумерный массив, элементы которого имеют тип char). Прежде, чем описать такой массив, придется определить максимальный размер строки (количество столбцов матрицы), которая в нем будет хранится. Понятно, что не все строки будут такого максимального размера. Из-за этого часть памяти будет расходоваться впустую

В такой ситуации нам бы помог так называемый “ragged array” – двумерный массив, в которой размерность «строки» изменяется. Однако в языке Си нет такого типа, но есть средство для его имитации – массив указателей на строки. Каждый элемент массива – это указатель на строку, которая содержит только те символы, из которых она состоит.

#include <stdio.h>

int main(void)

{

char arr\_1[][9] = {"January", "February", "March"};

/\*const\*/ char \*arr\_2[] = {"January", "February", "March"};

for (int i = 0; i < 3; i++)

printf("%s\n", arr\_1[i]);

for (int i = 0; i < 3; i++)

printf("%s\n", arr\_2[i]);

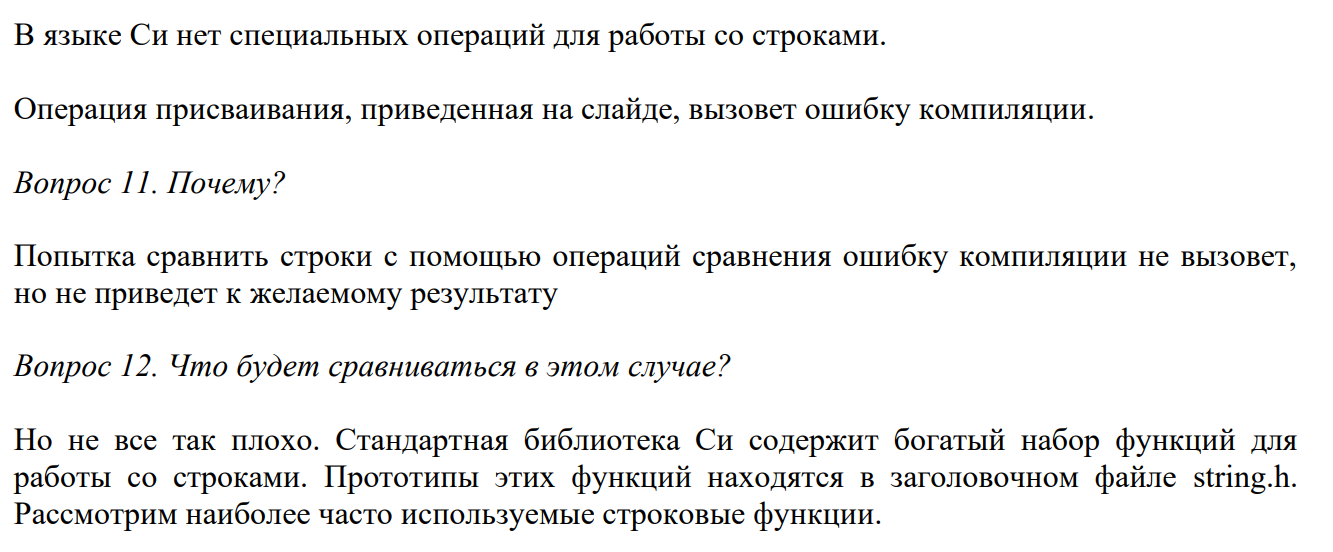
return 0;

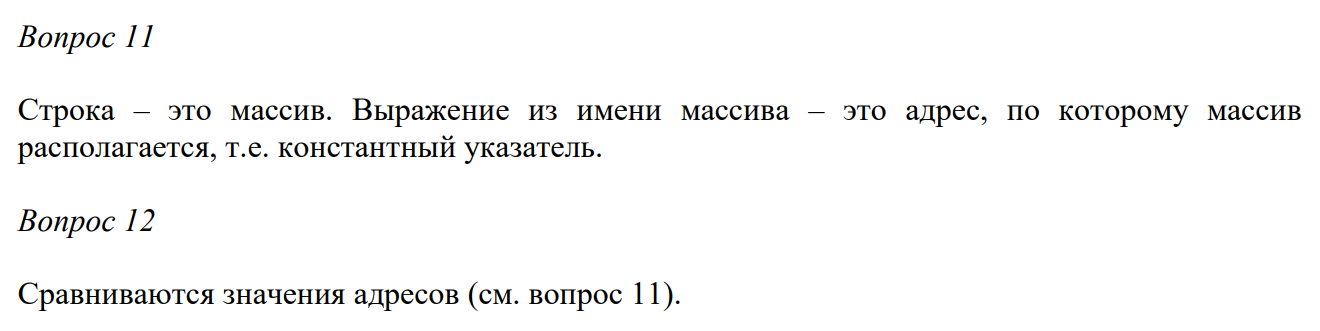
}

Если раскомментировать модификатор const ошибка компиляции не возникнет.

В приведенном примере модификатор const лучше использовать, потому что в массиве arr\_2 хранятся указатели на строковые литералы. Его не нужно использовать, если указатели адресуют «настоящие» строки, которые можно изменять.

***Обработка строк (strcpy, strcat, strlen, strcmp, snprintf, strtok, перевод строки в число).***





Правильнее было бы сказать «strcpy копирует строку, на которую указывает указатель s2 в массив, на который указывает указатель s1». Функция strcpy заменяет операцию присваивания для строк.

**char\* strcpy(char \*s1, const char \*s2);** // c99 7.21.2.3

**char\* strcpy(char \*s1, char \*s2);**

char src[] = "Hello!";

char dst[20];

strcpy(dst, src);

Недостаток функции strcpy состоит в том, что она не контролирует размеры строки, адресуемой указателем s2. Безопаснее пользоваться функцией strncpy.

**char\* strncpy(char \*s1, const char \*s2, size\_t n); // c99**

strncpy(dst, src, sizeof(dst) - 1);

dst[sizeof(dst) - 1] = '\0';

Однако у функции strncpy есть неприятная особенность: нулевой символ не будет помещен в строку s1, если строка s2 больше или равна размеру массива s1. Поэтому функцию strncpy обычно используют следующим образом

char str\_long[] = "Very very long string...";

char dst[10];

strncpy(dst, src\_long, sizeof(dst) - 1);

dst[sizeof(dst) - 1] = '\0';

***ВНИМАНИЕ***

Верен ли следующий фрагмент программы. Объясните почему.

char \*str\_1 = "abcdef";

char \*str\_2 = "123";

strcpy(str\_1, str\_2);

Функция strcpy изменяет значение строки, на которую указывает str\_1, а str\_1 – это указатель 12 на строковый литерал, который изменять нельзя.

**size\_t strlen(const char \*s);**

**char dst[20];**

**size\_t len;**

**strcpy(dst, "Hello!");**

**len = strlen(dst); // len = 6, а не 20**

Функция strlen возвращает длину строки s, т.е. количество символов в стоке s до нулевого символа. Нулевой символ в это количество не входит.

Обратите внимание, что функция возвращает размер строки, а не размер массива, в котором эта строка располагается.

Функция strcat добавляет содержимое строки s2 в конец строки s1. Она возвращает указатель на строку s1. Обычно это значение не используется.

Функция strcat заменяет операцию конкатенации (сложения) для строк.

**char\* strcat(char \*s1, const char \*s2);** // c99 7.21.3.1

**char\* strcat(char \*s1, const char \*s2);** // c99 7.21.3.1

**char\* strcat(char \*s1, const char \*s2);**

char src[] = ", world.";

char dst[20] = "Hello";

strcat(dst, src);

Как и функция strcpy функция strcat не контролирует размер результирующей строки. Поэтому безопаснее использовать функцию strncat, которая, как и функция strncpy, ограничивает количество обрабатываемых символов. Функция strncat (в отличие от функции strncpy) добавляет нулевой символ в строку (только для него нужно не забыть предусмотреть место!).

**char\* strncat(char \*s1, const char \*s2, size\_t n);**

int len = (sizeof(des)- 1) - strlen(des);

strncat(des, src, len);

**ВНИМАНИЕ**

Верен ли следующий фрагмент программы. Объясните почему.

char str\_1[10];

char \*str\_2 = "123";

strcat(str\_1, str\_2);

Функция strcat «складывает» строки. Переменная str\_1 – это не строка (она не содержит нулевой символ).

Функция strcmp сравнивает строки s1 и s2

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

* значение < 0, если s1 меньше s2
* 0, если s1 равна s2
* значение > 0, если s1 больше s2

Строки сравниваются в лексикографическом порядке (как в словаре).

**int strncmp(const char \*s1, const char \*s2, size\_t n);**

Иногда спрашиваю почему функция strcmp возвращает «нечеткие» значения (например, если строка s1 меньше строки s2 можно было бы вернуть значение -1). На этот вопрос можно получить ответ, рассмотрев реализацию функции strcmp (см. test\_12.c) от создателей языка Си.

**int strcmp(char \*s, char \*t)**

{

    int i;

    for (i = 0; s[i] == t[i]; i++)

        if (s[i] == '\0')

            return 0;

    return s[i] - t[i];

}



// stdio.h, c99 7.19.6.6

int sprintf(char \*s, const char \*format, ...);

Функция sprintf объявлена в заголовочном файле stdio.h, а не в string.h. Она ведет себя подобно функции printf, но при этом выполняет запись в строку, а не выводит ее на экран. В качестве первого аргумента функция sprintf принимает адрес целевой строки. Остальные аргументы аналогичны аргументам функции printf – строка форматирования и список аргументов, которые эта функция должна записать.

// stdio.h, c99 7.19.6.5

int snprintf(char \*s, size\_t n, const char \*format, ...);

Вместо функции sprintf, которая не контролирует размер результирующей строки, лучше использовать функцию snprintf, которая сохраняет в целевой строке максимум n – 1 символов, а по окончании работы помещает в нее нулевой символ.

ВНИМАНИЕ

К сожалению, реализация стандартной библиотеки компилятора gcc в MSYS использует библиотеки времени выполнения от Microsoft, которые не реализую такое поведение (не гарантируют завершение строки нулевым символом).

«Семейство atoi» На вход эти функции получают строку, а возвращают значение числа соответствующего типа. Основной недостаток функций этого семейства заключается в том, что в случае возникновения ошибки они возвращают значение 0.

// stdlib.h

// Семейство функций (atoi, atof, atoll), c99 7.20.1.2

**long int atol(const char\* str);**

«Семейство strtol» В отличие от функций «семейства atoi» эти функции не только могут учитывать разные системы счисления (для целых типов), но и возвращают указатель endptr на символ, на котором закончилось выделение числа. Этот указатель позволяет понять было ли выполнено преобразование правильно.

Обратите внимание на тип аргумента endptr. Этот аргумент имеет тип «указатель на указатель», потому что функции strtol нужно изменить значение указателя.

// stdlib.h

// Семейство функций (strtoul, strtoll, ...) ,c99 7.20.1.3

**long int strtol(const char\* string,**  **char\*\* endptr, int basis);**

ВНИМАНИЕ

Как с помощью параметра endptr можно понять, что преобразование строки в число было выполнено правильно?

Универсальный ответ дать нельзя, потому что нужно знать как организованы данные в программе.

Функция isdigit проверяет аргумент, передаваемый через параметр сharacter, является ли он десятичной цифрой. int isdigit( int character );

Значение, отличное от нуля (т.е. истинно), если аргумент функции — это десятичная цифра .

Ноль (т.е. ложь), в противном случае.

# Вопрос 28. Структура

\*что такое, зачем, поле, тег, память, инициализция структурных переменных, упаковка

Структуры, описание, для чего нужна, теги, поля, инициализация полей, операции со структурами, упаковка структур (чтобы поля структуры располагались одно за другим)

**Ответ**

***Что такое структура в языке Си?***

* Структура представляет собой одну или несколько переменных (возможно разного типа), которые объединены под одним именем.
* Структура — это объединение нескольких объектов, возможно, различного типа под одним именем, которое является типом структуры.

***Зачем?***

* Структуры помогают в организации сложных данных, потому что позволяют описывать множество логически связанных между собой отдельных элементов как единое целое.

***Что такое поле структуры?***

* Перечисленные в структуре переменные называются полями структуры.

***Что такое тег структуры?***

* Имя, которое располагается за ключевым словом struct, называется тегом структуры.

***Описание структуры:***

* Описание структуры начинается с ключевого слова struct, за которым указывается тег структуры. Тег структуры можно не указывать.
* Для описания поля структуры необходимо указать его тип и имя.

Поля структуры могут иметь любой кроме типа самой структуры

Почему? Как можно «обойти» это ограничение?

Потому что пока описание структуры не закончено компилятор еще не знает всех полей структуры и не может определить ее размер. Обойти это ограничение можно, описав внутри структуры поле типа указатель на эту структуру.

* Описание структуры обязательно заканчивается точкой с запятой.

***Пример описания структуры:***

* struct <имя>

{

<тип\_1> <имя\_1>;

<тип\_2> <имя\_2>;

...

<тип\_N> <имя\_N>;

}; // !

* Если описание структуры находится внутри функции, оно может использоваться только внутри той функции, в которой описывается.
* Если же структура должна использоваться в нескольких функциях и/или файлах, используют раздельное описание типа и переменных,
* Описание структуры, после которого нет списка переменных, не выделяет никакой памяти для объектов, а просто описывает шаблон структуры.
* Чтобы сослаться на этот шаблон при описании переменных, нужно указать ключевое слово struct и тег структуры
* Использование тега структуры без ключевого слова struct вызовет ошибку компиляции.

***Инициализация структурных переменных:***

* Структурные переменные можно иницализировать во время определения. Для этого нужно список инициализирующих значений заключить в фигурные скобки.
* Значения в списке инициализации должны идти в том же порядке, что и поля структуры.
* Список инициализации может содержать меньше значений, чем полей у структуры. Остальные поля структуры будут проинициализированы нулями.
* В C99 для структур можно использовать выделенные инициализаторы. Для идентификации конкретного поля выделенные инициализаторы используют точку и имя этого поля. Использование выделенных инициализаторов обладает несколькими преимуществами перед обычной инициализацией:
* Такую инициализацию легче читать и понимать.
* Инициализаторы могут указываться в произвольном порядке.
* Отсутствующие поля получают нулевые значения.
* Возможна комбинация со старым способом.

***Примеры инициализации структурных переменных:***

struct date

{

int day;

int month;

int year;

};

#define NAME\_LEN 15

struct person

{

char name[NAME\_LEN + 1];

struct date birth;

};

int main(void)

{

struct date today = { 21, 4, 2020 };

struct date day = { 21 };

struct date year = {, , 2020 };

// error: expected expression before ',' token

struct person rector = { "Alexandrov", { 7, 4, 1951 } };

struct date holidays[] = { { 9, 5, 2020 }, { 10, 5, 2020 }, { 11, 5, 2020 } }; // массив структур

struct date exam1 = { .day = 13, .month = 1, .year = 2019 };

struct date exam2 = { .day = 13, 1 }; // комбинация способов инициализации

struct date temp = { . year = 2020, 1 }; // ошибка компиляции из-за того, что был объявлен инициализатор, следующий за последним инициализатором year

}

***Упаковка:***

* Компилятор с целью оптимизации доступа может располагать поля в памяти не одно за другим, а по адресам кратным размеру поля.
* Для управления упаковкой отдельной структуры используется директива препроцессора pragma pack.
* Параметр “push” сообщает компилятору, что ему нужно сохранить текущее выравнивание, а параметр “1” задает нужное выравнивание. В данном случае выравнивание отсутствует, и поля структуры будут располагаться вплотную друг к другу.
* Параметр “pop” сообщает компилятору, что нужно восстановить ранее сохраненное выравнивание.
* Не следует использовать pragma pack без необходимости, потому что в результате можно получить более ресурсоемкий и медленный код.

***Пример упаковки с помощью pragma pack:***

#pragma pack (push, 1)

struct s\_2

{

char a;

int b;

} c2;

#pragma pack(pop)

sizeof(c2) == 5 // т.к. была сделана упаковка

* В общем случае, экземпляр структуры будет выровнен по самому длинному элементу. Для компилятора это самый простой способ убедиться, что все поля структуры будут также выровнены для быстрого доступа.
* Также, в C адрес структуры совпадает с адресом ее первого элемента.

***Операции над структурами:***

* Доступ к полю структуры осуществляется с помощью операции “.”, а если доступ к самой структуре осуществляется по указателю, то с помощью операции “->”.



* Если доступ к структурной переменной выполняется с помощью указателя, то в этом случае нужно сначала разыменовать этот указатель (т.е. получить саму структуру), а уже потом использовать операцию точка для доступа к полю.
* (\*some\_date).day = 1;
* (\*some\_date).month = 6; специальная операция ->
* (\*some\_date).year = 2020;
* Структурные переменные одного типа можно присваивать друг другу (замечание: у разных безымянных типов тип разный).
* Структурные переменные одного типа можно присваивать друг другу (замечание: у разных безымянных типов тип разный).

   some\_date = today;

* Когда такое присваивание плохо?

Если одним из полей структуры является указатель, после такого присваивания оба указателя будут указывать в одно место в памяти. Плохо это или хорошо зависит от реализации программы.

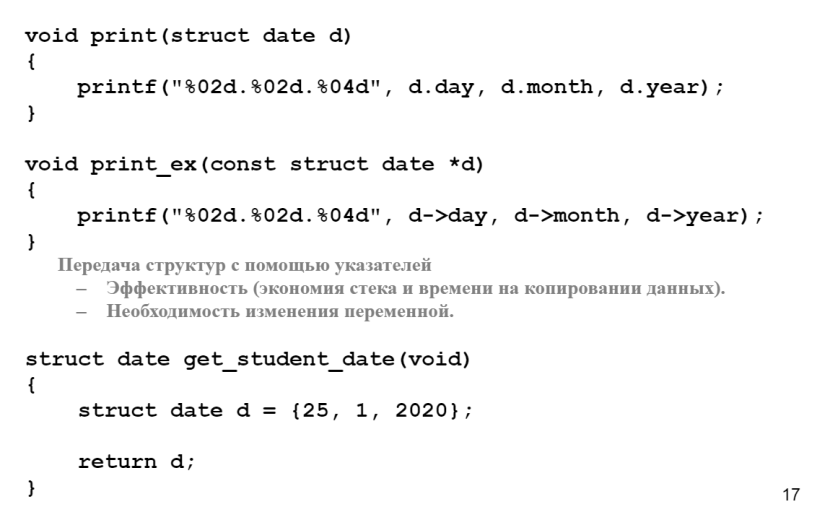
* Структуры нельзя сравнивать с помощью “==” и “!=”.. Это вызовет ошибку компиляции.

   if (today == \*tomorrow)

   // error: invalid operands to binary == (have 'struct date'

   // and 'struct date')

* Структуры могут передаваться в функцию как параметры и возвращаться из функции в качестве ее значения.



# Вопрос 29. Объединения, перечисляемый тип

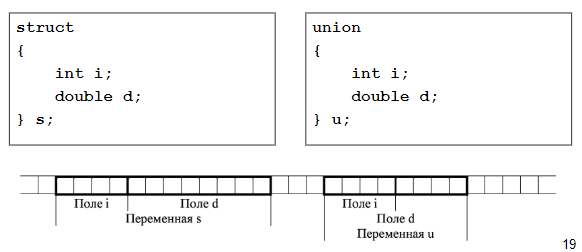
\*все про объединения и перечисляемый тип, для чего каждое используется

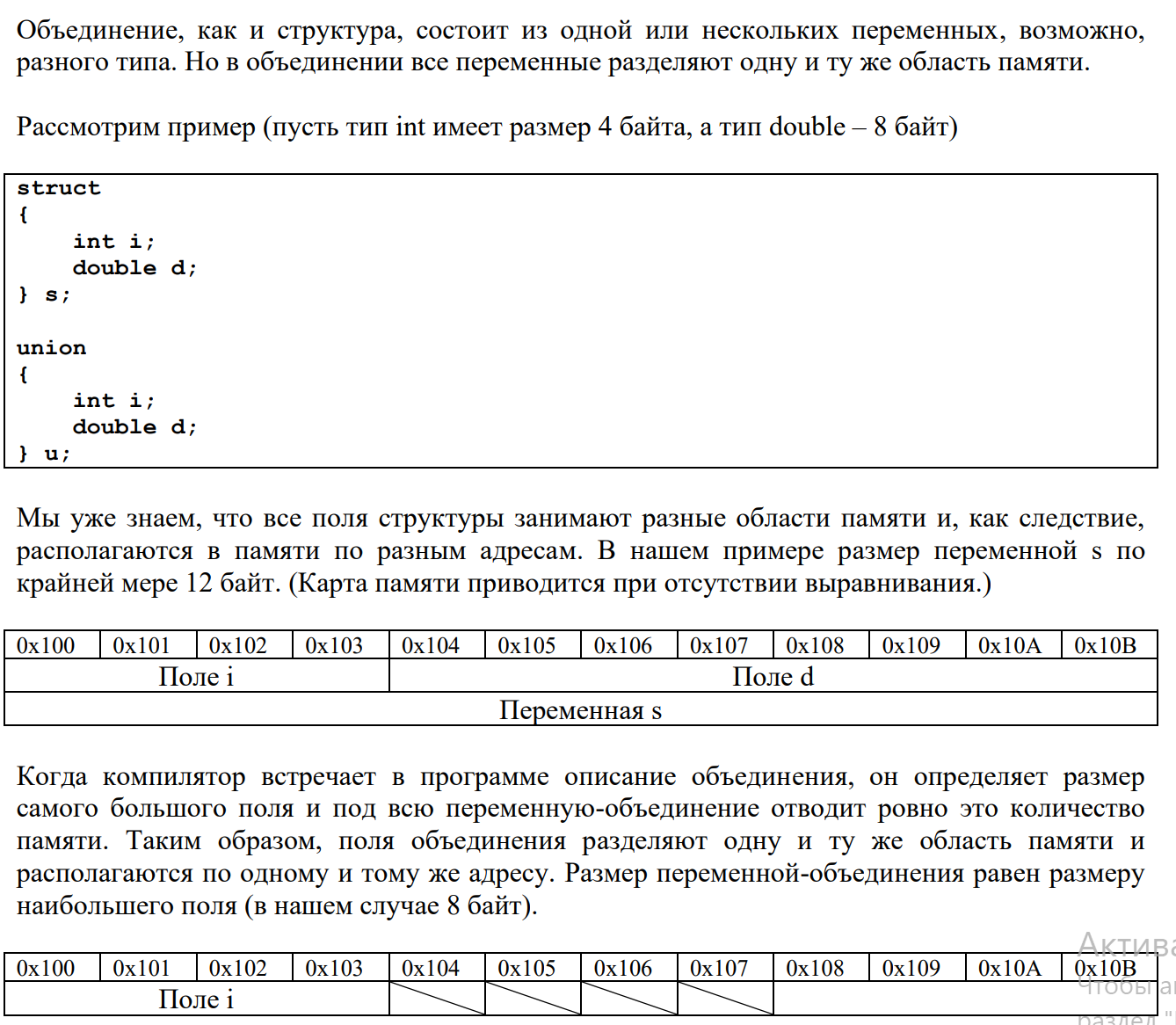
**Ответ**

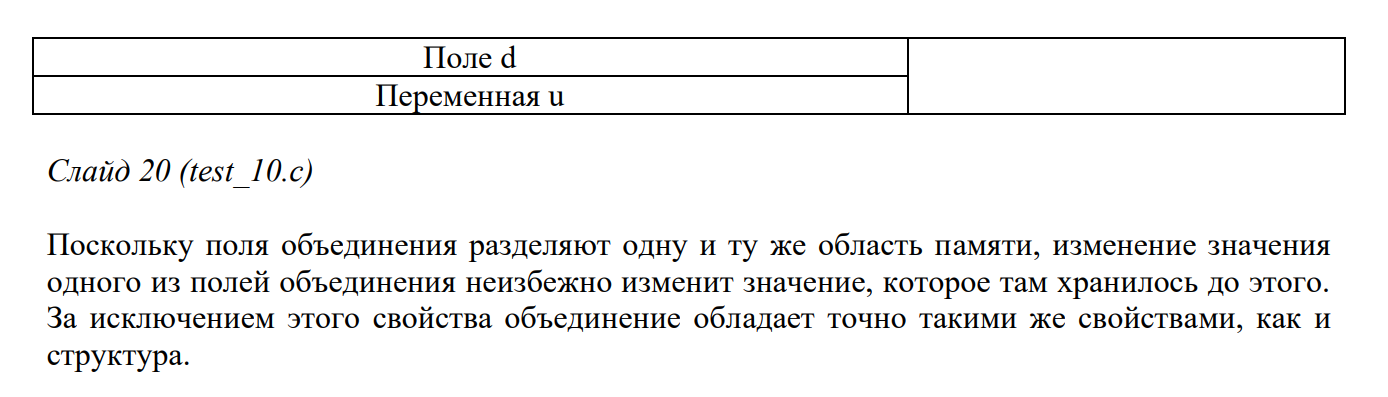
***Что такое объединение?***

Объединение, как и структура, содержит одно или несколько полей возможно разного типа. Однако все поля объединения разделяют одну и ту же область памяти.

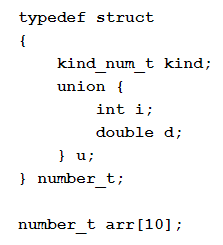
***Расположение полей объединения в памяти.***



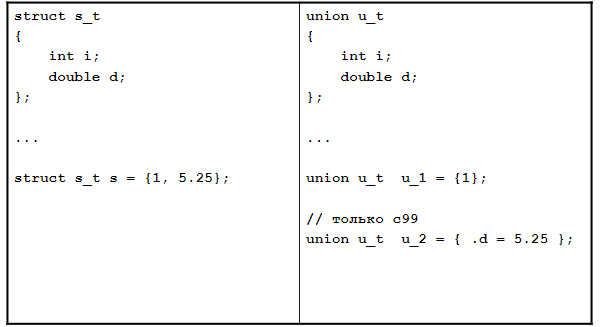




Когда компилятор встречает в программе описание объединения, он определяет размер самого большого поля и под всю переменную-объединение отводит ровно это количество памяти. Таким образом, поля объединения разделяют одну и ту же область памяти и располагаются по одному и тому же адресу. Размер переменной-объединения равен размеру наибольшего поля (в нашем случае 8 байт).



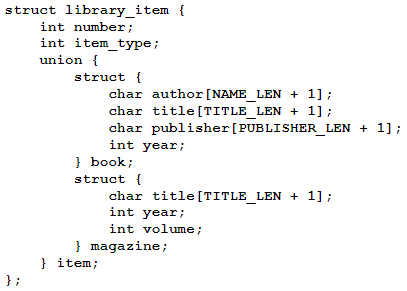
***Инициализация объединений.***



До появления выделенных инициализаторов при определении переменной-объединения можно было проинициализировать только первое поле этого объединения. Появления выделенных инициализаторов решило эту проблему.

***Использование объединений.***

Во-первых, для экономии места.

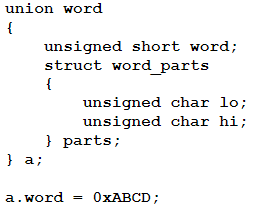
Предположим, необходимо написать программу, которая автоматизирует работу библиотеки. В библиотеке хранятся книги и журналы. Сведения о книге описываются структурой book\_t, а сведения о журнале – структурой magazine\_t. Мы могли бы в структуру library\_item\_t, которая описывает библиотечную единицу, добавить и сведения о книге, и сведения о журнале. Но библиотечная единица может быть либо книгой, либо журналом. Таки образом, при таком представлении данных часть памяти будет использоваться впустую. Чтобы минимизировать теряемую память, поместим сведения о книге и журнале сначала в объединение item\_t, а уже это объединение поместим внутрь library\_item\_t. Поле kind в структуре library\_item\_t нужно для того, чтобы понимать с чем мы имеем дело: с книгой или с журналом.

Во-вторых, для создания новых типов данных.

Мы с вами знаем, что массив состоит из элементов одного типа. Воспользовавшись объединением как показано в примере test\_12.c мы можем создать массив, который содержит одновременно как целые, так и вещественные числа.

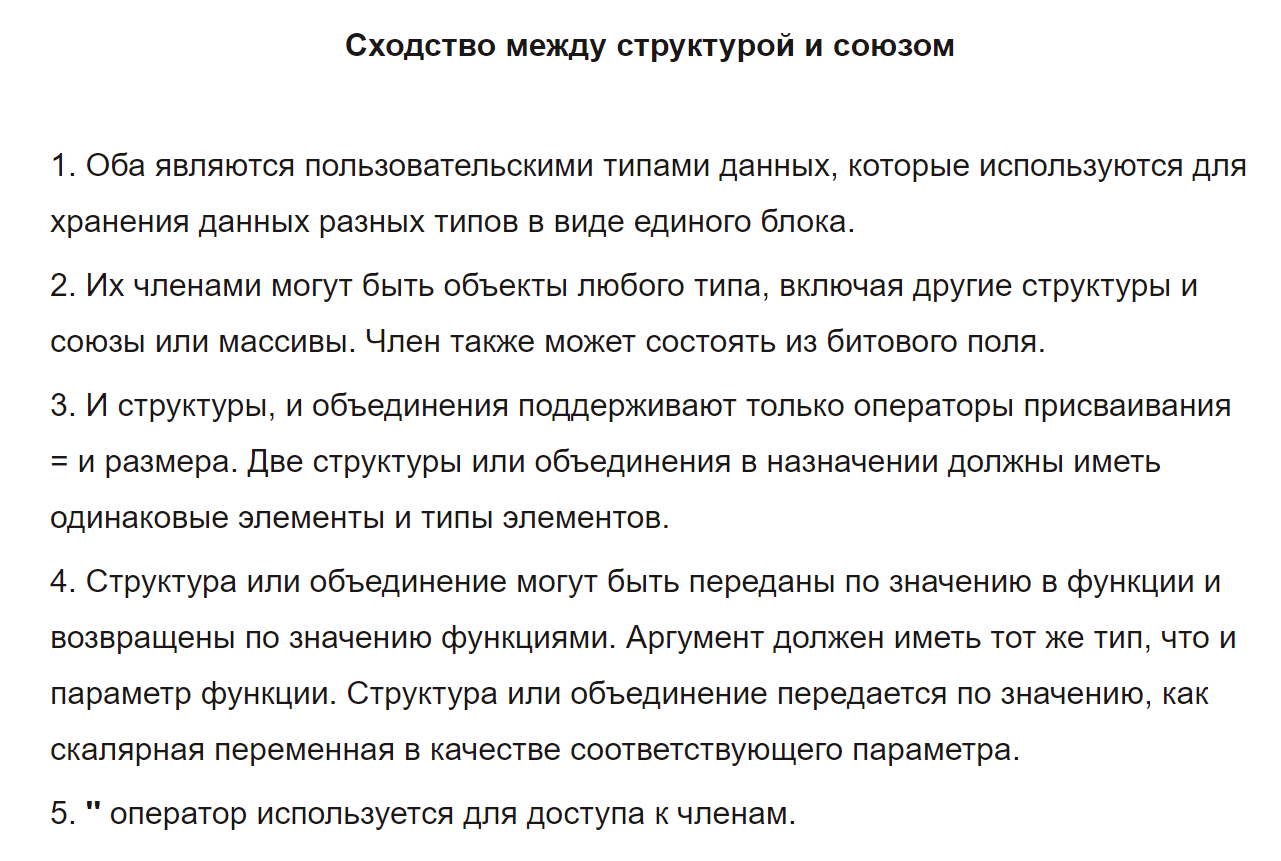
В-третьих, объединения часто используют для того, чтобы по-другому взглянуть на одни и те же данные.

Пусть размер типа unsigned short равен двум байтам (назовем такой тип «словом», англ. word). В слове можно выделить старший байт (биты с 8 по 15) и младший байт (биты с 0 по 7). С помощью объединения, приведенного в примере test\_13.c, можно работать как c целым словом, так и с его старшей и младшей половинами

. 

***Сравните структуру и объединение.***

Расположение в памяти разное



В 5 оператор точка, а не кавычки

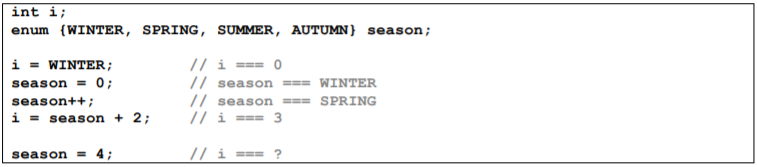
***Перечисляемый тип в языке Си.***

Некоторые переменные в программе могут принимать значения из небольшого набора. Например, переменная, которая хранит времена года, имеет всего 4 возможных значения: зима, весна, лето, осень.

В языке Си предусмотрел специальный тип, разработанный для переменных, которые принимают небольшое количество значений. Это перечисляемый тип, значения которого перечисляются программистом, который должен придумать имя для каждого значения.

Язык Си интерпретирует перечисляемый тип и его «константы» как целые. По умолчанию компилятор назначает значения 0, 1, 2, ... «константам» в каждом конкретном перечислении.

Поскольку значения перечисляемого типа — это целые числа, язык Си позволяет их перемешивать с обычными целыми.



# Вопрос 30. Текстовые файлы

Что такое файл, что такое файловая система, атрибуты файла, классификация файлов, текстовый файл, сравнение двоичных и текстовых файлов, описание и особенности работы с текстовым файлом.

Предопределенные файловые переменные, обработка ошибочных ситуаций (подходы или основные моменты). Преимущества и недостатки

**Ответ**

***Что такое файл***

***Файл*** – именованная область данных на носителе информации.

***Атрибуты файла***

Файл обладает набором различных свойств:

* имя файла (файлы идентифицируются именами);
* тип файла;
* размер файла;
* права доступа (определяют набор операций, который может быть применен к файлу);
* время (создания, последней модификации, последнего доступа и т.д.);
* и т.п.

*Атрибуты* дополнительные параметры, определяющие свойства файлов. Любой файл имеет 4 признака:

* H или Hidden, скрытый файл – файл, имя которого не выводится на экран при выдаче оглавления директории. Имеют атрибут io.sys и msdos.sys, DRVSPACE.BIN
* S или System, системный файл – файл, обладающий важными функциями в работе ОС. Его отличительная особенность в том, что его нельзя изменить средствами ОС. Имеют атрибут io.sys и msdos.sys, DRVSPACE.BIN
* A или Archive, неархивный файл – этот атрибут означает, что файл не является архивной копией (т.е. готов к архивированию)
* R или Read-only, только чтение – файлы, в которые невозможна запись. Имеют атрибут io.sys и msdos.sys, DRVSPACE.BIN

***Что такое файловая система***

С точки зрения операционной системы (ОС), жесткий диск представляет собой набор секторов. В операционной системе выделяется специальный компонент – файловая система, назначение которой состоит в том, чтобы обеспечить пользователю интерфейс для работы с данными, хранящимися на диске.

***Файловая система*** — порядок, определяющий способ организации, хранения и именования данных на [носителях информации](https://ru.wikipedia.org/wiki/%D0%9D%D0%BE%D1%81%D0%B8%D1%82%D0%B5%D0%BB%D1%8C_%D0%B8%D0%BD%D1%84%D0%BE%D1%80%D0%BC%D0%B0%D1%86%D0%B8%D0%B8) в компьютерах, а также в другом [электронном](https://ru.wikipedia.org/wiki/%D0%AD%D0%BB%D0%B5%D0%BA%D1%82%D1%80%D0%BE%D0%BD%D0%B8%D0%BA%D0%B0) оборудовании

***Классификация файлов***

Файлы бывают разных типов:

* Обычные файлы;
* *текстовые* - текстовые файлы содержат только печатные символы (т.е. символы, которые можно вводить с клавиатуры). Они организованы в виде последовательности строк, каждая из которых заканчивается символом новой строки ‘\n’.
* *двоичные*. - двоичные файлы – последовательность произвольных байтов, они часто имеют сложную внутреннюю структуру, например, объектный код программы или архивный файл
* Специальные файлы – это файлы, связанные с устройствами ввода-вывода, которые позволяют пользователю выполнять операции ввода-вывода, используя обычные команды записи в файл или чтения из файла
* Файлы-каталоги - каталог – это, с одной стороны, группа файлов, объединенных пользователем исходя из некоторых соображений (например, файлы, содержащие программы игр, или файлы, составляющие один программный пакет), а с другой стороны – это файл, содержащий системную информацию о группе файлов, его составляющих.

По ***способу организации*** файлы делятся на:

* [*файлы с произвольным доступом*](https://ru.wikipedia.org/wiki/%D0%A4%D0%B0%D0%B9%D0%BB%D1%8B_%D1%81_%D0%BF%D1%80%D0%BE%D0%B8%D0%B7%D0%B2%D0%BE%D0%BB%D1%8C%D0%BD%D1%8B%D0%BC_%D0%B4%D0%BE%D1%81%D1%82%D1%83%D0%BF%D0%BE%D0%BC) ([файлы](https://ru.wikipedia.org/wiki/%D0%A4%D0%B0%D0%B9%D0%BB), хранящие информацию в структурированном (для [поиска](https://ru.wikipedia.org/wiki/%D0%9F%D0%BE%D0%B8%D1%81%D0%BA) и обращения) виде. Поиск в таких файлах осуществляется в области адресов (ключей) и завершается обращением непосредственно к искомому участку. Дисковое пространство, занимаемое таким файлом, поделено на одинаковые участки (записи), имеющие одинаковую структуру полей. Файлы с произвольным доступом выигрывают у [последовательных файлов](https://ru.wikipedia.org/wiki/%D0%9F%D0%BE%D1%81%D0%BB%D0%B5%D0%B4%D0%BE%D0%B2%D0%B0%D1%82%D0%B5%D0%BB%D1%8C%D0%BD%D1%8B%D0%B5_%D1%84%D0%B0%D0%B9%D0%BB%D1%8B) по скорости доступа, но проигрывают по компактности.)
* [*файлы с последовательным доступом*](https://ru.wikipedia.org/wiki/%D0%9F%D0%BE%D1%81%D0%BB%D0%B5%D0%B4%D0%BE%D0%B2%D0%B0%D1%82%D0%B5%D0%BB%D1%8C%D0%BD%D1%8B%D0%B5_%D1%84%D0%B0%D0%B9%D0%BB%D1%8B) ([файлы](https://ru.wikipedia.org/wiki/%D0%A4%D0%B0%D0%B9%D0%BB), хранящие информацию в неструктурированном (для поиска и обращения) виде. Поиск в таких файлах осуществляется последовательным считыванием файла с начала и сравнением «всего» с искомым. Так же и обращение к определённому участку файла каждый раз требует «чтения с начала». Примером последовательных файлов являются [текстовые файлы](https://ru.wikipedia.org/wiki/%D0%A2%D0%B5%D0%BA%D1%81%D1%82%D0%BE%D0%B2%D1%8B%D0%B9_%D1%84%D0%B0%D0%B9%D0%BB) (\*.txt)

По ***способу хранения информации*** файлы делятся на:

* текстовые (формат ASCII)
* двоичные

По ***отношению к выполнимости*** файлы делятся на:

* выполнимые
* не выполнимые

***Работа с файлами***

FILE\* fopen(const char \*filename, const char \*mode);

Функция ***fopen*** открывает файл. Возвращает файловый указатель, с помощью которого далее можно осуществлять доступ к файлу. Если вызов функции ***fopen*** прошел неудачно, то она возвратит ***NULL***.

|  |  |
| --- | --- |
| **Режим (mode)** | **Описание** |
| "r" | Чтение (Read). Файл должен существовать. |
| "w" | Запись (Write). Если файл с таким именем не существует, он будет создан, в противном случае его содержимое будет потеряно. |
| "a" | Запись в конец файла (Append). Файл создаётся, если не существовал. |

Функция ***fopen*** может открывать файл в текстовом или бинарном режиме. По умолчанию используется текстовый режим. Если необходимо открыть файл в бинарном режиме, то в конец строки добавляется буква b, например "rb", "wb", "ab".

int fclose(FILE \*f);

Функция ***fclose*** закрывает файл: выполняет запись буферизированных, но еще незаписанных данных, уничтожает непрочитанные буферизированные входные данные, освобождает все автоматически выделенные буфера, после чего закрывает файл или устройство. Возвращает ***EOF*** в случае ошибки и ***NULL*** в противном случае.

Почему нужно использовать ***fclose***?

* У программы может быть ограничение на количество одновременно открытых файлов.
* В Windows, когда программа открывает файл, другие программы не могут его открыть или удалить.
* Вывод в файл совершается не сразу, а через вспомогательный буфер. При отсутствии вызова ***fclose*** информация из него может не попасть в файл.

***Текстовые файлы***

Текстовые файлы содержат только печатные символы (т.е. символы, которые можно вводить с клавиатуры). Они организованы в виде последовательности строк, каждая из которых заканчивается символом новой строки ‘\n’. В конце последней строки этот символ не является обязательным. Текстовые файлы можно прочитать на экране и распечатать на принтере. Примером текстовых файлов могут служить исходные тексты программ.

*Замечание*

*Требования операционной системы к символу новой строки в текстовом файле могут быть различными (например, Windows: CR+LF, Linux: LF, Mac: CR). Поэтому чтобы программа была переносимой, символ ‘\n’ преобразуется библиотекой ввода/вывода в представление, принятое в конкретной операционной системе. По этой причине в текстовом файле может не быть однозначного соответствия между символами, которые пишутся (читаются), и теми, которые хранятся в файле.*

***Сравнение двоичных и текстовых файлов***

*Текстовые файлы* содержат только печатные символы. Они организованы в виде последовательности строк, каждая из которых заканчивается символом новой строки ‘\n’. В конце последней строки этот символ не является обязательным.

*Двоичные файлы* – последовательность произвольных байтов, они часто имеют сложную внутреннюю структуру.

Текстовый файл в отличие от двоичного файла является *переносимым*. Формат текстового файла крайне прост и его можно изменять текстовым редактором, который есть практически любой ОС.

Текстовый файл является структурой данных с *последовательным доступом*. Файлы с последовательным доступом читаются от начала к концу, поэтому невозможно одновременно и считывать из них данные, и записывать. Чтобы изменить одну запись файла последовательного доступа, его нужно весь записать заново.

В двоичных файлах информация хранится в представлении, в котором она хранятся в оперативной памяти (т.е. двоичные файлы *непереносимы*).

В отличие от текстового файла двоичный файл структура данных с *произвольным доступом*: можно установить внутренний указатель файла на интересующую позицию и прочитать (записать) информацию именно туда.

***Описание и особенности работы с текстовым файлом***

int fputs(const char \*string, FILE \*filestream);

Функция ***fputs*** записывает строку, указанную в параметре ***string*** в поток ***filestream***.

Функция начинает копирование с адреса, указанного в ***string***, пока не будет достигнут нулевой символ. ***Этот нулевой символ не копируется в поток***.

В случае успеха, возвращается неотрицательное значение. В случае ошибки, функция возвращает значение ***EOF***.

int fprintf(FILE \*f, const char \*format, ...);

Функция ***fprintf*** преобразует и выводит данные в файл (или устройство), связанный с файловой переменной ***f***, под управлением строки форматирования ***format***. Возвращает количество записанных символов или, в случае ошибки – ***EOF***

int fscanf(FILE \*f, const char \*format, ...);

Функция ***fscanf*** считывает данные из файла (или устройства), связанного с файловой переменной ***f***, под управлением строки форматирования **format** и присваивает преобразованные значения последующим аргументам.

Функция завершает работу, когда «исчерпывается» строка форматирования. При этом она возвращает ***EOF***, если до преобразования очередного значения ей встретился конец файла или появилась ошибка. В противном случае функция возвращает количество введенных значений.

char\* fgets(char \*s, int n, FILE \*f);

Прекращает ввод когда (любое из):

* прочитан символ ‘\n’;
* достигнут конец файла ‘\0’;
* прочитано size – 1 символов.

fgets(str, sizeof(str), stdin);

Функция ***fgets*** считывает символы из потока и сохраняет их в виде строки в параметр ***str*** до тех пор пока не наступит конец строки или пока не будет достигнут конец файла.

Символ новой строки ‘\n’ прекращает работу функции ***fgets***, но он считается допустимым символом, и поэтому он копируется в строку ***str***.

Нулевой символ автоматически добавляется в строку ***str*** после прочитанных символов, чтобы сигнализировать о конце строки.

В случае успеха, функция возвращает указатель на ***str***. В противном случае возвращает ***NULL***.

int fseek(FILE \*f, long int offset, int orign);

Функция ***fseek*** устанавливает внутренний указатель положения в файле, связанном с файловой переменной **f**, в новую позицию ***offset*** (количество байт для смещения) относительно ***origin*** (позиция указателя, относительно которого будет выполняться смещение).

***origin*** может принимать три значения:

* SEEK\_SET начало файла;
* SEEK\_CUR текущее положение файла;
* SEEK\_END конец файла.

В случае успеха, функция возвращает нулевое значение. В противном ненулевое значение.

При использовании функции ***fseek*** в текстовых файлах со смещением на величину значения, отличного от нуля или значения, полученного функцией ***ftell***, имейте в виду, что на некоторых платформах, в связи с нестандартным форматом преобразования текстовых файлов, может возникнуть ситуация с некорректного позиционирования указателя.

void rewind(FILE \*f);

Функция ***rewind*** позволяет начать обработку файла сначала.

В отличие от функции ***fseek***, ***rewind*** обнуляет индикатор ошибок. Для потоков, открытых в режиме обновления (чтения + записи), вызов функции ***rewind*** позволяет переключаться между режимами чтения и записи.

***Предопределенные файловые переменные***

При инициализации программы библиотека ввода/вывода заводит три файловые переменные ***stdin***, ***stdout*** и ***stderr*** для:

* Стандартного потока ввода (сокращение от *standard input*). Обычно этот поток связан с клавиатурой. Программа может читать из него данные.
* Стандартного потока вывода (сокращение от *standard output*). Обычно этот поток связан с дисплеем. Программа может выводить в него данные.
* Стандартного потока ошибок (сокращение от *standard error*). Обычно этот поток так же связан с дисплеем. Программа может выводить в него сообщения об ошибках.

Эти файловые переменные могут использоваться в любой функции, где используется переменная типа ***FILE\****.

В следующем примере показано, как можно перенаправить ***stdin*** и ***stdout*** в файл

# весь вывод программы a.exe на экран будет перенаправлен в файл out.txt

a.exe > out.txt

# весь ввод программы b.exe будет выполняться из файла in.txt

b.exe < in.txt

***Обработка ошибочных ситуаций (подходы или основные моменты)***

Большинство функций стандартной библиотеки при возникновении ошибки возвращают отрицательное число или ***NULL*** и записывают в глобальную переменную ***errno*** код ошибки.

Эта переменная определена в заголовочном файле ***errno.h***. Заметим, что в случае успешного выполнения функции эта переменная просто не изменяется и может содержать любой мусор, поэтому проверять ее имеет смысл лишь в случае, когда ошибка действительно произошла.

void perror(const char \*message);

Функция ***perror*** преобразует значение глобальной переменной ***errno*** в строку и записывает эту строку в ***stderr***.

char\* strerror(int errnum);

Функция ***strerror*** возвращает указатель на строку, содержащую системное сообщение об ошибке, связанной со значением ***errnum***. Эту строку не следует менять ни при каких обстоятельствах.

***Преимущества и недостатки***

См. ***Сравнение двоичных и текстовых файлов***

# Вопрос 31. Двоичные файлы

Что такое файл, что такое файловая система, атрибуты файла, классификация файлов, двоичный файл, сравнение двоичных и текстовых файлов, описание и особенности работы с двоичным файлом.

Обработка ошибочных ситуаций (подходы или основные моменты). Преимущества и недостатки

**Ответ**

***Что такое файл***

***Файл*** – именованная область данных на носителе информации.

***Атрибуты файла***

Файл обладает набором различных свойств:

* имя файла (файлы идентифицируются именами);
* тип файла;
* размер файла;
* права доступа (определяют набор операций, который может быть применен к файлу);
* время (создания, последней модификации, последнего доступа и т.д.);
* и т.п.

*Атрибуты* дополнительные параметры, определяющие свойства файлов. Любой файл имеет 4 признака:

* H или Hidden, скрытый файл – файл, имя которого не выводится на экран при выдаче оглавления директории. Имеют атрибут io.sys и msdos.sys, DRVSPACE.BIN
* S или System, системный файл – файл, обладающий важными функциями в работе ОС. Его отличительная особенность в том, что его нельзя изменить средствами ОС. Имеют атрибут io.sys и msdos.sys, DRVSPACE.BIN
* A или Archive, неархивный файл – этот атрибут означает, что файл не является архивной копией (т.е. готов к архивированию)
* R или Read-only, только чтение – файлы, в которые невозможна запись. Имеют атрибут io.sys и msdos.sys, DRVSPACE.BIN

***Что такое файловая система***

С точки зрения операционной системы (ОС), жесткий диск представляет собой набор секторов. В операционной системе выделяется специальный компонент – файловая система, назначение которой состоит в том, чтобы обеспечить пользователю интерфейс для работы с данными, хранящимися на диске.

***Файловая система*** — порядок, определяющий способ организации, хранения и именования данных на [носителях информации](https://ru.wikipedia.org/wiki/%D0%9D%D0%BE%D1%81%D0%B8%D1%82%D0%B5%D0%BB%D1%8C_%D0%B8%D0%BD%D1%84%D0%BE%D1%80%D0%BC%D0%B0%D1%86%D0%B8%D0%B8) в компьютерах, а также в другом [электронном](https://ru.wikipedia.org/wiki/%D0%AD%D0%BB%D0%B5%D0%BA%D1%82%D1%80%D0%BE%D0%BD%D0%B8%D0%BA%D0%B0) оборудовании

***Классификация файлов***

Файлы бывают разных типов:

* Обычные файлы;
* *текстовые* - текстовые файлы содержат только печатные символы (т.е. символы, которые можно вводить с клавиатуры). Они организованы в виде последовательности строк, каждая из которых заканчивается символом новой строки ‘\n’.
* *двоичные*. - двоичные файлы – последовательность произвольных байтов, они часто имеют сложную внутреннюю структуру, например, объектный код программы или архивный файл
* Специальные файлы – это файлы, связанные с устройствами ввода-вывода, которые позволяют пользователю выполнять операции ввода-вывода, используя обычные команды записи в файл или чтения из файла
* Файлы-каталоги - каталог – это, с одной стороны, группа файлов, объединенных пользователем исходя из некоторых соображений (например, файлы, содержащие программы игр, или файлы, составляющие один программный пакет), а с другой стороны – это файл, содержащий системную информацию о группе файлов, его составляющих.

По ***способу организации*** файлы делятся на:

* [*файлы с произвольным доступом*](https://ru.wikipedia.org/wiki/%D0%A4%D0%B0%D0%B9%D0%BB%D1%8B_%D1%81_%D0%BF%D1%80%D0%BE%D0%B8%D0%B7%D0%B2%D0%BE%D0%BB%D1%8C%D0%BD%D1%8B%D0%BC_%D0%B4%D0%BE%D1%81%D1%82%D1%83%D0%BF%D0%BE%D0%BC) ([файлы](https://ru.wikipedia.org/wiki/%D0%A4%D0%B0%D0%B9%D0%BB), хранящие информацию в структурированном (для [поиска](https://ru.wikipedia.org/wiki/%D0%9F%D0%BE%D0%B8%D1%81%D0%BA) и обращения) виде. Поиск в таких файлах осуществляется в области адресов (ключей) и завершается обращением непосредственно к искомому участку. Дисковое пространство, занимаемое таким файлом, поделено на одинаковые участки (записи), имеющие одинаковую структуру полей. Файлы с произвольным доступом выигрывают у [последовательных файлов](https://ru.wikipedia.org/wiki/%D0%9F%D0%BE%D1%81%D0%BB%D0%B5%D0%B4%D0%BE%D0%B2%D0%B0%D1%82%D0%B5%D0%BB%D1%8C%D0%BD%D1%8B%D0%B5_%D1%84%D0%B0%D0%B9%D0%BB%D1%8B) по скорости доступа, но проигрывают по компактности.)
* [*файлы с последовательным доступом*](https://ru.wikipedia.org/wiki/%D0%9F%D0%BE%D1%81%D0%BB%D0%B5%D0%B4%D0%BE%D0%B2%D0%B0%D1%82%D0%B5%D0%BB%D1%8C%D0%BD%D1%8B%D0%B5_%D1%84%D0%B0%D0%B9%D0%BB%D1%8B) ([файлы](https://ru.wikipedia.org/wiki/%D0%A4%D0%B0%D0%B9%D0%BB), хранящие информацию в неструктурированном (для поиска и обращения) виде. Поиск в таких файлах осуществляется последовательным считыванием файла с начала и сравнением «всего» с искомым. Так же и обращение к определённому участку файла каждый раз требует «чтения с начала». Примером последовательных файлов являются [текстовые файлы](https://ru.wikipedia.org/wiki/%D0%A2%D0%B5%D0%BA%D1%81%D1%82%D0%BE%D0%B2%D1%8B%D0%B9_%D1%84%D0%B0%D0%B9%D0%BB) (\*.txt)

По ***способу хранения информации*** файлы делятся на:

* текстовые (формат ASCII)
* двоичные

По ***отношению к выполнимости*** файлы делятся на:

* выполнимые
* не выполнимые

***Работа с файлами***

FILE\* fopen(const char \*filename, const char \*mode);

Функция ***fopen*** открывает файл. Возвращает файловый указатель, с помощью которого далее можно осуществлять доступ к файлу. Если вызов функции ***fopen*** прошел неудачно, то она возвратит ***NULL***.

|  |  |
| --- | --- |
| **Режим (mode)** | **Описание** |
| "r" | Чтение (Read). Файл должен существовать. |
| "w" | Запись (Write). Если файл с таким именем не существует, он будет создан, в противном случае его содержимое будет потеряно. |
| "a" | Запись в конец файла (Append). Файл создаётся, если не существовал. |

Функция ***fopen*** может открывать файл в текстовом или бинарном режиме. По умолчанию используется текстовый режим. Если необходимо открыть файл в бинарном режиме, то в конец строки добавляется буква b, например "rb", "wb", "ab".

int fclose(FILE \*f);

Функция ***fclose*** закрывает файл: выполняет запись буферизированных, но еще незаписанных данных, уничтожает непрочитанные буферизированные входные данные, освобождает все автоматически выделенные буфера, после чего закрывает файл или устройство. Возвращает ***EOF*** в случае ошибки и ***NULL*** в противном случае.

Почему нужно использовать ***fclose***?

* У программы может быть ограничение на количество одновременно открытых файлов.
* В Windows, когда программа открывает файл, другие программы не могут его открыть или удалить.
* Вывод в файл совершается не сразу, а через вспомогательный буфер. При отсутствии вызова ***fclose*** информация из него может не попасть в файл.

***Двоичные файлы***

*Двоичные файлы* – последовательность произвольных байтов, они часто имеют сложную внутреннюю структуру, например, объектный код программы или архивный файл.

Такая модель файла полностью совпадает с системой представлений, принятой в Си для работы с памятью на низком (физическом уровне).

* физическая память имеет байтную структуру – единицей адресации является байт;
* любая переменная занимает фиксированное количество байтов, определяемое ее типом. Операция ***sizeof*** возвращает эту размерность;
* указатель на переменную интерпретируется как ее адрес в памяти. Преобразование типа указателя к ***void\**** позволяет интерпретировать его как «чистый» адрес, а преобразование к ***char\**** - как указатель на массив байтов (физическое представление памяти).

Исходя из этих принципов, функции двоичного ввода-вывода ***fread*** и ***fwrite*** переносят содержимое памяти в двоичный файл «прозрачно», т.е. байт в байт без каких-либо преобразований. Функции используются для перенесения данных из файла в память программы (чтение) и обратно (запись).

***Сравнение двоичных и текстовых файлов***

*Текстовые файлы* содержат только печатные символы. Они организованы в виде последовательности строк, каждая из которых заканчивается символом новой строки ‘\n’. В конце последней строки этот символ не является обязательным.

*Двоичные файлы* – последовательность произвольных байтов, они часто имеют сложную внутреннюю структуру.

Текстовый файл в отличие от двоичного файла является *переносимым*. Формат текстового файла крайне прост и его можно изменять текстовым редактором, который есть практически любой ОС.

Текстовый файл является структурой данных с *последовательным доступом*. Файлы с последовательным доступом читаются от начала к концу, поэтому невозможно одновременно и считывать из них данные, и записывать. Чтобы изменить одну запись файла последовательного доступа, его нужно весь записать заново.

В двоичных файлах информация хранится в представлении, в котором она хранятся в оперативной памяти (т.е. двоичные файлы *непереносимы*).

В отличие от текстового файла двоичный файл структура данных с *произвольным доступом*: можно установить внутренний указатель файла на интересующую позицию и прочитать (записать) информацию именно туда.

***Описание и особенности работы с двоичным файлом***

size\_t fwrite(const void \*ptr, size\_t size, size\_t count, FILE \*f);

Функция ***fwrite*** записывает в файл, связанный с файловой переменной ***f***, данные из буфера ***ptr***. Количество элементов в буфере ***ptr*** указывается в переменной ***count***, а размер каждого элемента – в переменной ***size*** (размер указывается в байтах).

*Замечание*

*Вместо буфера можно передать адрес любой переменной.*

Функция ***fwrite*** возвращает количество удачно записанных элементов. Если это количество не совпадает со значением ***count***, то произошла ошибка.

size\_t fread(void \*ptr, size\_t size, size\_t count, FILE \*f);

Функция ***fread*** считывает из файла, связанного с файловой переменной ***f***, данные и помещает их в буфер ***ptr***. Количество считываемых элементов буфера указывается в переменной ***count***, а размер каждого элемента – в переменной ***size*** (размер указывается в байтах).

Функция возвращает число удачно прочитанных элементов. Если возвращаемое значение отличается от количества элементов, значит произошла ошибка или был достигнут конец файла. Вы можете использовать функции ***ferror*** или ***feof*** для определения проблемы – произошла ошибка или был достигнут конец файла.

int fseek(FILE \*f, long int offset, int orign);

Функция ***fseek*** устанавливает внутренний указатель положения в файле, связанном с файловой переменной **f**, в новую позицию ***offset*** (количество байт для смещения) относительно ***origin*** (позиция указателя, относительно которого будет выполняться смещение).

***origin*** может принимать три значения:

* SEEK\_SET начало файла;
* SEEK\_CUR текущее положение файла;
* SEEK\_END конец файла.

В случае успеха, функция возвращает нулевое значение. В противном ненулевое значение.

При использовании функции ***fseek*** в текстовых файлах со смещением на величину значения, отличного от нуля или значения, полученного функцией ***ftell***, имейте в виду, что на некоторых платформах, в связи с нестандартным форматом преобразования текстовых файлов, может возникнуть ситуация с некорректного позиционирования указателя.

long int ftell(FILE \*f);

Функция ***ftell*** возвращает текущее положение внутреннего указателя в файле, связанном с файловой переменной ***f***.

Для двоичных файлов возвращается значение, соответствующее количеству байт от начала файла. Для текстовых файлов это значение может не соответствовать точному количеству байт от начала файла. Функция ***ftell*** возвращает ***-1L*** в случае возникновения ошибки

***Предопределенные файловые переменные***

При инициализации программы библиотека ввода/вывода заводит три файловые переменные ***stdin***, ***stdout*** и ***stderr*** для:

* Стандартного потока ввода (сокращение от *standard input*). Обычно этот поток связан с клавиатурой. Программа может читать из него данные.
* Стандартного потока вывода (сокращение от *standard output*). Обычно этот поток связан с дисплеем. Программа может выводить в него данные.
* Стандартного потока ошибок (сокращение от *standard error*). Обычно этот поток так же связан с дисплеем. Программа может выводить в него сообщения об ошибках.

Эти файловые переменные могут использоваться в любой функции, где используется переменная типа ***FILE\****.

В следующем примере показано, как можно перенаправить ***stdin*** и ***stdout*** в файл

# весь вывод программы a.exe на экран будет перенаправлен в файл out.txt

a.exe > out.txt

# весь ввод программы b.exe будет выполняться из файла in.txt

b.exe < in.txt

***Обработка ошибочных ситуаций (подходы или основные моменты)***

Большинство функций стандартной библиотеки при возникновении ошибки возвращают отрицательное число или ***NULL*** и записывают в глобальную переменную ***errno*** код ошибки.

Эта переменная определена в заголовочном файле ***errno.h***. Заметим, что в случае успешного выполнения функции эта переменная просто не изменяется и может содержать любой мусор, поэтому проверять ее имеет смысл лишь в случае, когда ошибка действительно произошла.

void perror(const char \*message);

Функция ***perror*** преобразует значение глобальной переменной ***errno*** в строку и записывает эту строку в ***stderr***.

char\* strerror(int errnum);

Функция ***strerror*** возвращает указатель на строку, содержащую системное сообщение об ошибке, связанной со значением ***errnum***. Эту строку не следует менять ни при каких обстоятельствах.

***Преимущества и недостатки***

См. ***Сравнение двоичных и текстовых файлов***

# Вопрос 32. Регулярные структуры над двоичным файлом

\*Типизированный файл. Записи фиксированной длины

Структура файла, как с регулярной структурой работать, какие функции нужны для этого

Регулярные структуры над двоичным файлом. Типизированный файл. Записи фиксированной длины

Структура файла, как работать с регулярной структурой, какие функции нужны для этого

Краткий ликбез по файлам из 30 и 31

**Ответ**

***Что такое файл***

***Файл*** – именованная область данных на носителе информации.

***Атрибуты файла***

Файл обладает набором различных свойств:

* имя файла (файлы идентифицируются именами);
* тип файла;
* размер файла;
* права доступа (определяют набор операций, который может быть применен к файлу);
* время (создания, последней модификации, последнего доступа и т.д.);
* и т.п.

*Атрибуты* дополнительные параметры, определяющие свойства файлов. Любой файл имеет 4 признака:

* H или Hidden, скрытый файл – файл, имя которого не выводится на экран при выдаче оглавления директории. Имеют атрибут io.sys и msdos.sys, DRVSPACE.BIN
* S или System, системный файл – файл, обладающий важными функциями в работе ОС. Его отличительная особенность в том, что его нельзя изменить средствами ОС. Имеют атрибут io.sys и msdos.sys, DRVSPACE.BIN
* A или Archive, неархивный файл – этот атрибут означает, что файл не является архивной копией (т.е. готов к архивированию)
* R или Read-only, только чтение – файлы, в которые невозможна запись. Имеют атрибут io.sys и msdos.sys, DRVSPACE.BIN

***Что такое файловая система***

С точки зрения операционной системы (ОС), жесткий диск представляет собой набор секторов. В операционной системе выделяется специальный компонент – файловая система, назначение которой состоит в том, чтобы обеспечить пользователю интерфейс для работы с данными, хранящимися на диске.

***Файловая система*** — порядок, определяющий способ организации, хранения и именования данных на [носителях информации](https://ru.wikipedia.org/wiki/%D0%9D%D0%BE%D1%81%D0%B8%D1%82%D0%B5%D0%BB%D1%8C_%D0%B8%D0%BD%D1%84%D0%BE%D1%80%D0%BC%D0%B0%D1%86%D0%B8%D0%B8) в компьютерах, а также в другом [электронном](https://ru.wikipedia.org/wiki/%D0%AD%D0%BB%D0%B5%D0%BA%D1%82%D1%80%D0%BE%D0%BD%D0%B8%D0%BA%D0%B0) оборудовании

***Классификация файлов***

Файлы бывают разных типов:

* Обычные файлы;
* *текстовые* - текстовые файлы содержат только печатные символы (т.е. символы, которые можно вводить с клавиатуры). Они организованы в виде последовательности строк, каждая из которых заканчивается символом новой строки ‘\n’.
* *двоичные*. - двоичные файлы – последовательность произвольных байтов, они часто имеют сложную внутреннюю структуру, например, объектный код программы или архивный файл
* Специальные файлы – это файлы, связанные с устройствами ввода-вывода, которые позволяют пользователю выполнять операции ввода-вывода, используя обычные команды записи в файл или чтения из файла
* Файлы-каталоги - каталог – это, с одной стороны, группа файлов, объединенных пользователем исходя из некоторых соображений (например, файлы, содержащие программы игр, или файлы, составляющие один программный пакет), а с другой стороны – это файл, содержащий системную информацию о группе файлов, его составляющих.

По ***способу организации*** файлы делятся на:

* [*файлы с произвольным доступом*](https://ru.wikipedia.org/wiki/%D0%A4%D0%B0%D0%B9%D0%BB%D1%8B_%D1%81_%D0%BF%D1%80%D0%BE%D0%B8%D0%B7%D0%B2%D0%BE%D0%BB%D1%8C%D0%BD%D1%8B%D0%BC_%D0%B4%D0%BE%D1%81%D1%82%D1%83%D0%BF%D0%BE%D0%BC) ([файлы](https://ru.wikipedia.org/wiki/%D0%A4%D0%B0%D0%B9%D0%BB), хранящие информацию в структурированном (для [поиска](https://ru.wikipedia.org/wiki/%D0%9F%D0%BE%D0%B8%D1%81%D0%BA) и обращения) виде. Поиск в таких файлах осуществляется в области адресов (ключей) и завершается обращением непосредственно к искомому участку. Дисковое пространство, занимаемое таким файлом, поделено на одинаковые участки (записи), имеющие одинаковую структуру полей. Файлы с произвольным доступом выигрывают у [последовательных файлов](https://ru.wikipedia.org/wiki/%D0%9F%D0%BE%D1%81%D0%BB%D0%B5%D0%B4%D0%BE%D0%B2%D0%B0%D1%82%D0%B5%D0%BB%D1%8C%D0%BD%D1%8B%D0%B5_%D1%84%D0%B0%D0%B9%D0%BB%D1%8B) по скорости доступа, но проигрывают по компактности.)
* [*файлы с последовательным доступом*](https://ru.wikipedia.org/wiki/%D0%9F%D0%BE%D1%81%D0%BB%D0%B5%D0%B4%D0%BE%D0%B2%D0%B0%D1%82%D0%B5%D0%BB%D1%8C%D0%BD%D1%8B%D0%B5_%D1%84%D0%B0%D0%B9%D0%BB%D1%8B) ([файлы](https://ru.wikipedia.org/wiki/%D0%A4%D0%B0%D0%B9%D0%BB), хранящие информацию в неструктурированном (для поиска и обращения) виде. Поиск в таких файлах осуществляется последовательным считыванием файла с начала и сравнением «всего» с искомым. Так же и обращение к определённому участку файла каждый раз требует «чтения с начала». Примером последовательных файлов являются [текстовые файлы](https://ru.wikipedia.org/wiki/%D0%A2%D0%B5%D0%BA%D1%81%D1%82%D0%BE%D0%B2%D1%8B%D0%B9_%D1%84%D0%B0%D0%B9%D0%BB) (\*.txt)

По ***способу хранения информации*** файлы делятся на:

* текстовые (формат ASCII)
* двоичные

По ***отношению к выполнимости*** файлы делятся на:

* выполнимые
* не выполнимые

***Работа с файлами***

FILE\* fopen(const char \*filename, const char \*mode);

Функция ***fopen*** открывает файл. Возвращает файловый указатель, с помощью которого далее можно осуществлять доступ к файлу. Если вызов функции ***fopen*** прошел неудачно, то она возвратит ***NULL***.

|  |  |
| --- | --- |
| **Режим (mode)** | **Описание** |
| "r" | Чтение (Read). Файл должен существовать. |
| "w" | Запись (Write). Если файл с таким именем не существует, он будет создан, в противном случае его содержимое будет потеряно. |
| "a" | Запись в конец файла (Append). Файл создаётся, если не существовал. |

Функция ***fopen*** может открывать файл в текстовом или бинарном режиме. По умолчанию используется текстовый режим. Если необходимо открыть файл в бинарном режиме, то в конец строки добавляется буква b, например "rb", "wb", "ab".

int fclose(FILE \*f);

Функция ***fclose*** закрывает файл: выполняет запись буферизированных, но еще незаписанных данных, уничтожает непрочитанные буферизированные входные данные, освобождает все автоматически выделенные буфера, после чего закрывает файл или устройство. Возвращает ***EOF*** в случае ошибки и ***NULL*** в противном случае.

Почему нужно использовать ***fclose***?

* У программы может быть ограничение на количество одновременно открытых файлов.
* В Windows, когда программа открывает файл, другие программы не могут его открыть или удалить.
* Вывод в файл совершается не сразу, а через вспомогательный буфер. При отсутствии вызова ***fclose*** информация из него может не попасть в файл.

***Текстовые файлы***

Текстовые файлы содержат только печатные символы (т.е. символы, которые можно вводить с клавиатуры). Они организованы в виде последовательности строк, каждая из которых заканчивается символом новой строки ‘\n’. В конце последней строки этот символ не является обязательным. Текстовые файлы можно прочитать на экране и распечатать на принтере. Примером текстовых файлов могут служить исходные тексты программ.

***Сравнение двоичных и текстовых файлов***

*Текстовые файлы* содержат только печатные символы. Они организованы в виде последовательности строк, каждая из которых заканчивается символом новой строки ‘\n’. В конце последней строки этот символ не является обязательным.

*Двоичные файлы* – последовательность произвольных байтов, они часто имеют сложную внутреннюю структуру.

Текстовый файл в отличие от двоичного файла является *переносимым*. Формат текстового файла крайне прост и его можно изменять текстовым редактором, который есть практически любой ОС.

Текстовый файл является структурой данных с *последовательным доступом*. Файлы с последовательным доступом читаются от начала к концу, поэтому невозможно одновременно и считывать из них данные, и записывать. Чтобы изменить одну запись файла последовательного доступа, его нужно весь записать заново.

В двоичных файлах информация хранится в представлении, в котором она хранятся в оперативной памяти (т.е. двоичные файлы *непереносимы*).

В отличие от текстового файла двоичный файл структура данных с *произвольным доступом*: можно установить внутренний указатель файла на интересующую позицию и прочитать (записать) информацию именно туда.

***Типизированный файл***

Это такой вид файлов, в котором содержатся записи одного типа и фиксированной длины. Основным свойством этих файлов является то, что их структура данных представляет собой последовательность компонентов одного типа

Поскольку известен тип элементов файла, а следовательно, и объем памяти, отводимой под каждый из них, можно рассчитать позицию каждого из элементов внутри файла. Это позволяет организовать непосредственный доступ к любому элементу типизированного файла.

***Запись переменной длины (ЗПД)*** - единица хранения, меняющая свою размерность в различных экземплярах. В формате записей переменной длины могут быть представлены данные различного вида: динамические массивы, строки текста, последовательности данных, имеющие собственный внутренний формат.

Имеется два основных способа хранения записей переменной длины в файле:

* используется специальное значение или код - ограничитель записи. Типичным примером является строка текста в памяти, имеющая в качестве ограничителя символ '\0', который в файле превращается в последовательность символов '\r' и '\n'. В этом смысле обычный текстовый файл при работе с ним построчно есть файл записей переменной длины;
* запись предваряется переменной-счетчиком, которая содержит длину этой записи. Содержимое записи при этом может быть любым, поскольку явно выделенные коды-ограничители отсутствуют (свойство прозрачности данных). Этот формат удобен еще и тем, что при чтении из последовательного файла программа, зная размерность записи, может сразу же резервировать необходимую динамическую память для ее размещения.

По своей природе такой файл является файлом последовательного доступа, поскольку определить адрес любой записи без знания размерности всех предыдущих не представляется возможным.

Файл ***записей фиксированной длины (ЗФД)*** фактически представляет собой обычный массив, размещенный в двоичном файле. При принятом в Си низкоуровневом представлении данных в файле, адрес любого элемента массива вычисляется по простой формуле ***A=A0+i\*sizeof(type), где A0*** – начальный адрес массива, а i – индекс (номер) его элемента. Таким образом, к его элементам возможен произвольный (прямой) доступ.

Хотя функции **getc()/putc()** позволяют вносить в файл отдельные символы, но фактически мы имеем дело с бинарными файлами. Если мы записываем в файл строку, то в принципе мы даже можем открыть записанный файл любом текстовом редакторе и понять, что там было записано. Но не всегда данные могут представлять строки. И чтобы более наглядно разобраться с работой с бинарными файлами, рассмотрим еще одни пример - с записью-чтением структуры из файла:

// запись структуры в файл

int save(char \* filename, struct person \*p)

{

    FILE \* fp;

    char \*c;

    int size = sizeof(struct person); // количество записываемых байтов

    if ((fp = fopen(filename, "wb")) == NULL)

    {

        perror("Error occured while opening file");

        return 1;

    }

    // устанавливаем указатель на начало структуры

    c = (char \*)p;

    // посимвольно записываем в файл структуру

    for (int i = 0; i < size; i++)

    {

        putc(\*c++, fp);

    }

    fclose(fp);

    return 0;

}

// загрузка из файла структуры

int load(char \* filename)

{

    FILE \* fp;

    char \*c;

    int i; // для считывания одного символа

    // количество считываемых байтов

    int size = sizeof(struct person);

    // выделяем память для считываемой структуры

    struct person \* ptr = (struct person \*) malloc(size);

    if ((fp = fopen(filename, "rb")) == NULL)

    {

        perror("Error occured while opening file");

        return 1;

    }

    // устанавливаем указатель на начало блока выделенной памяти

    c = (char \*)ptr;

    // считываем посимвольно из файла

    while ((i = getc(fp))!=EOF)

    {

        \*c = i;

        c++;

    }

    fclose(fp);

    // вывод на консоль загруженной структуры

    printf("%-20s %5d \n", ptr->name, ptr->age);

    free(ptr);

При записи мы получаем указатель на структуру, который содержит начальный адрес блока памяти, по которому располагается структура. Для структуры выделяется 16 + 4=20 байт.

c = (char \*)p;

Функция putc записывает отдельный символ в файл, однако нам надо записать структуру. Для этого мы создаем указатель на символ (который по сути представляет один байт) и устанавливаем этот указатель на начало блока памяти, выделенного для структуры.

for (int i = 0; i < size; i++)

{

    putc(\*c++, fp);

}

И в данном случае нам не важно, какие поля имеет структура, какой она имеет размер. Мы работаем с ней как с набором байт и заносим эти байты в файл. После занесения каждого отдельного байта в файл указатель c в блоке памяти перемещается на один байт вперед.

При чтении файла используется похожий принцип только в обратную сторону.

# Вопрос 33. Этапы получения исполняемого файла

\*все все все)

Этапы получения исполняемого файла

Препроцессирование

ассемблирование

компоновка

**Ответ**

Процесс получения исполняемого файла из исходного кода состоит из четырех шагов, которые показаны на следующей IDEF0 диаграмме:

***Препроцессирование***

Препроцессор выполняет четыре действия:

* удаление комментариев;
* вставку файлов (директива include);
* текстовые замены (по-другому говорят - раскрытие макросов, директива define);
* условную компиляцию (директива if).

Файл, который получен в результате работы препроцессора, называется единицей трансляции.

***Трансляция на язык ассемблера***

Файл, полученный препроцессором, передается на вход транслятору с99, который переводит его с языка Си на язык ассемблера

***Язык ассемблера*** – это машинно-ориентированный язык низкого уровня.

Благодаря ключу *"-fverbose-asm"* в файл помещаются дополнительные комментарии (строки, которые начинаются с символа '#'), которые позволяют сопоставить исходную программу на языке Си с полученной программой на языке ассемблере.

***Трансляция программы с языка ассемблера в машинный код***

С языка ассемблера программа переводится в машинный код с помощью транслятора as. На выходе этого транслятора получается не текстовый (как на двух предыдущих этапах), а двоичный файл. Этот файлы называется объектным файлом.

***Объектный файл*** – это двоичный файл, который содержит блоки машинного кода и метаданные

Метаданные обычно образуют две таблицы:

* таблицу символов (symbol table), в которой перечисляются имена, определенные (main) и упомянутые (puts) в исходном тексте программы;
* таблицу релокации (relocation table), в которой перечисляются «места» в объектном файле, которые нужно будет «поправить» при получении исполняемого файла.

Ключ "-d" отвечает за дизассемблирование (т.е. обратное преобразование из машинного кода на язык ассемблера) объектного файла.

Ключ "-r" отвечает за вывод «мест», которые нужно будет поправить компоновщику.

***Компоновка***

Чтобы получить исполняемый файл необходимо вызвать компоновщик

В процессе получения исполняемого файла компоновщик решает несколько задач

* объединяет несколько объектных файлов в единый исполняемый файл;
* выполняет связывание переменных и функций, которые требуются очередному объектному файлу, но находятся где-то в другом месте (в стандартной библиотеке или другом объектном файле);
* добавляет специальный код, который подготавливает окружение для вызова функции main, а после ее завершения выполняет обратные действия.

***Исполняемый файл*** - файл, содержащий программу в виде, в котором она может быть (после загрузки в память и настройки по месту) исполнена компьютером.

Файл, который получен в результате работы препроцессора, называется единицей трансляции.

***Ключи компилятора***

Использовать четыре разные утилиты для получения объектного файла неудобно. Поэтому компилятор умеет выполнять все эти действия самостоятельно или с помощью вызова внешних утилит. Работа компилятора управляется ключом. В большинстве POSIX-систем строка вызова компилятора выглядит следующим образом

В нашем случае ***имя\_компилятора*** – это ***gcc***. Рассмотрим назначение основных ключей этого компилятора.

**-E** Компилятор выполняет только этап препроцессирования.

**-S** Компилятор выполняет только трансляцию программы на язык ассемблера.

**-с** Компилятор выполняет только получение объектного файла (т.е. будут выполнены три первых этапа получения исполняемого файла).

**-o имя\_файла** Задает имя выходного файла.

**-std=XY** Задает стандарт языка Си, которые будет использовать при трансляции программы. В нашем случае -std=c99.

**-Wall** Вынуждает компилятор выводить информацию о всех предупреждениях, с которыми он столкнулся во время компиляции.

Наличие предупреждений не мешает получению исполняемого файла. Однако с их помощью компилятор пытается обратить внимание программиста на нестандартное использование той или иной конструкции языка, которое может привести к возникновению ошибки.

**-Werror** Вынуждает компилятор интерпретировать предупреждения, которые выдаются в процессе компиляции, как ошибки.

**-pedantic** Вынуждает компилятор педантично следовать указанному стандарту.

**-g[level]** Задает уровень отладочной информации, которая добавляется к объектному файлу. level – это число, которое принимает значения от 0 (отсутствие отладочной информации) до 3 (максимальное количество отладочной информации).

**-O[level]** Задает уровень оптимизации, которую выполняет компилятор. level – это число, которое принимает значения от 0 (отсутствие оптимизации) до 3 (максимальный уровень оптимизации).

Обязательными для использования при сборке лабораторных работ являются ключи **"- std=c99 -Wall -Werror"**.

# Вопрос 34. Многофайловая организация проекта

\*особенности компиляции, ошибки компиляции/компоновки, заголовочные файлы

Многофайловая, однофайловая - отличие, преимущества

При многофайловой есть ошибки компоновки и компиляции

Особенности создания заголовочных файлов

**Ответ**

***Многофайловый проект, раздельная компиляция***

Преимущества многофайловых проектов

* Удобное ориентирование в коде программы ( в однофаловых проектах, со временем, становится неудобно ориентироваться, из-за обьема текста )
* Улучшение структурированности программы ( обычно в один файл выносят функции, которые логически связаны друг с другом, это, способствует повторному использованию функций, выделенных в отдельный файл )
* Упрощенная организационная работа в команде ( вносятся изменения в отдельные файлы )
* Отсутвие необходимости перекомпилировать файлы, в которые изменения не вносились

***Заголовочные файлы***

В случае когда мы хотим скомпилировать файл, отдельно от остальных, используются заголовочные файлы. В заголовочные файлах размещают заголовки функций и называют в соответствие с файлом, в котором эти функции размещены ( расширение “.h” ).

Стоит обратить внимание, что заголовочный файл, подключается как в файле который мы хотим скомпилировать, так и в файле в файле, к которому, этот заголовочный файл относится. Это сделано для того, чтобы не произошел конфликт, оба файла содержат дублирующуюся информацию – заголовок функции. Если по какой-то причине эти заголовки в файлах начнут расходиться (например, из-за отпечатки или потому что заголовок функции был изменен при внесении каких-то изменений в определение этой функции), это может привести к получению неработающей программы.

***Include guard***

В заголовочный файл помимо обьявлений функций помещают определения типов и макросов. При обработки файлов, включения могут быть определенны повторно, что приведетт к возникновению ошибки компиляции.

Для того чтобы защититься от повторного включения используется конструкция include guard.

1. #ifendef \_FILE\_NAME\_H\_

2. #define \_FILE\_NAME\_H\_

3. Int function();

4. #endif

1. Имя файла определенно? Если нет, продолжаем.

2. Определяем имя

4 . Директива #if закончилась

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

**ВСЕ ОТВЕТЫ ПИСАТЬ СВОИМИ СЛОВАМИ! ИНАЧЕ БАН!**