**Список теоретических вопросов к экзамену**

**по дисциплине «Программирование на Си».**

[Указатель на void. Динамическое выделение памяти 8](#_Toc61120557)

[1. Для чего используется указатель на void? Приведите примеры. 8](#_Toc61120558)

[2. Каковы особенности использования указателя на void? Приведите примеры. 8](#_Toc61120559)

[3. Функции для выделения и освобождения памяти malloc, calloc, free. Порядок работы и особенности использования этих функций. 8](#_Toc61120560)

[4. Функция realloc. Особенности использования. 9](#_Toc61120561)

[5. Общие «свойства» функций malloc, calloc, realloc. 9](#_Toc61120562)

[6. Функция выделения памяти и явное приведение типа: за и против. 9](#_Toc61120563)

[7. Особенности выделения 0 байт памяти. 9](#_Toc61120564)

[8. Способы возвращения динамического массива из функции. 10](#_Toc61120565)

[9. Типичные ошибки при работе с динамической памятью. 10](#_Toc61120566)

[10. Подходы к обработке ситуации отсутствия свободной памяти при выделении. 10](#_Toc61120567)

[Указатели на функцию 11](#_Toc61120568)

[1. Для чего используется указатель на функцию? Приведите примеры. 11](#_Toc61120569)

[2. Указатель на функцию: описание, инициализация, вызов функции по указателю. 11](#_Toc61120570)

[3. Функция qsort, примеры использования. 11](#_Toc61120571)

[4. Особенности использования указателей на функцию. 11](#_Toc61120572)

[5. Указатель на функцию и адресная арифметика. 12](#_Toc61120573)

[6. Указатель на функцию и указатель на void. 12](#_Toc61120574)

[Make 13](#_Toc61120575)

[1. Утилита make: назначение, входные данные, идея алгоритма работы. 13](#_Toc61120576)

[2. Разновидности утилиты make. 13](#_Toc61120577)

[3. Сценарий сборки проекта: название файла, структура сценария сборки. 13](#_Toc61120578)

[4. Правила: составные части, особенности использования правил в зависимости от составных частей. 13](#_Toc61120579)

[5. Особенности выполнения команд. 13](#_Toc61120580)

[6. Простой сценарий сборки. 13](#_Toc61120581)

[7. Алгоритм работы утилиты make на примере простого сценария сборки. 14](#_Toc61120582)

[8. Ключи запуска утилиты make. 14](#_Toc61120583)

[9. Использование переменных. Примеры использования. 15](#_Toc61120584)

[10. Неявные правила и переменные. 15](#_Toc61120585)

[11. Автоматические переменные и их использование. 15](#_Toc61120586)

[12. Шаблонные правила. Примеры использование. 15](#_Toc61120587)

[13. Условные конструкции в сценарии сборки. Примеры использования. 15](#_Toc61120588)

[14. Переменные, зависящие от цели. Примеры использования. 16](#_Toc61120589)

[15. Автоматическая генерация зависимостей. 16](#_Toc61120590)

[Динамические матрицы 17](#_Toc61120591)

[1. Представление динамической матрицы с помощью одномерного массива. Преимущества и недостатки. 17](#_Toc61120592)

[2. Представление динамической матрицы с помощью массива указателей на строки/столбцы. Преимуществ и недостатки. 17](#_Toc61120593)

[3. Объединенный подход для представления динамической матрицы (отдельное выделение памяти под массив указателей и массив данных). Преимуществ и недостатки. 17](#_Toc61120594)

[4. Объединенный подход для представления динамической матрицы (массив указателей и массив данных располагаются в одной области). Преимуществ и недостатки. 18](#_Toc61120595)

[5. Необходимо реализовать функцию, которая может обрабатывать как статические, так и динамические матрицы. Какими способами это можно сделать? 18](#_Toc61120596)

[Чтение сложных объявлений 19](#_Toc61120597)

[1. Умение читать сложные объявления и использовать это на практике. 19](#_Toc61120598)

[Строки/структуры и динамическое выделение памяти. 20](#_Toc61120599)

[1. Функции, возвращающие динамическую строку: strdup/strndup, getline, snprintf/asprintf. 20](#_Toc61120600)

[2. Feature Test Macro. 20](#_Toc61120601)

[3. Функции memcpy, memmove, memcmp, memset. 21](#_Toc61120602)

[4. Структуры с полями указателями и особенности их использования. 21](#_Toc61120603)

[5. «Поверхностное» копирование VS «глубокое» копирование. 22](#_Toc61120604)

[6. «Рекурсивное» освобождение памяти для структур с динамическими полями. 22](#_Toc61120605)

[7. Структуры переменного размера. Приведите примеры. 22](#_Toc61120606)

[8. Что такое «flexible array member»? Какие особенности использование есть у этих полей? Для чего они нужны? Приведите примеры. 23](#_Toc61120607)

[9. Flexible array member до C99. 23](#_Toc61120608)

[10. Flexible array member VS поле-указатель. 23](#_Toc61120609)

[Динамически расширяемый массив. Cписки. Двоичные деревья поиска. 24](#_Toc61120610)

[1. Дайте определение массива. 24](#_Toc61120611)

[2. Дайте определение линейного односвязного списка. 24](#_Toc61120612)

[3. Сравните массив и линейный односвязный список. 24](#_Toc61120613)

[4. Динамически расширяемый массив. 24](#_Toc61120614)

[5. ДРМ. Описание типа. 24](#_Toc61120615)

[6. ДРМ. Добавление нового элемента. 24](#_Toc61120616)

[7. ДРМ. Удаление элемента. 24](#_Toc61120617)

[8. ДРМ. Особенности использования. 24](#_Toc61120618)

[9. ДРМ. Почему при добавлении нового элемента память необходимо выделять блоками, а не под один элемент? 25](#_Toc61120619)

[10. Линейный односвязный список. 25](#_Toc61120620)

[11. ЛОС. Описание типа. 25](#_Toc61120621)

[12. ЛОС. Добавление нового элемента в начало/конец списка. 25](#_Toc61120622)

[13. ЛОС. Вставка элемента перед/после указанного. 25](#_Toc61120623)

[14. ЛОС. Удаление элемента из списка. 25](#_Toc61120624)

[15. ЛОС. Обход списка. 25](#_Toc61120625)

[16. ЛОС. Удаление памяти из-под всего списка. 25](#_Toc61120626)

[17. ЛОС. Возможные улучшения "классической" реализации. 25](#_Toc61120627)

[18. Двоичное дерево поиска. 26](#_Toc61120628)

[19. ДДП. Описание типа. 26](#_Toc61120629)

[20. ДДП. Добавление элемента. 26](#_Toc61120630)

[21. ДДП. Поиск элемента (рекурсивный и не рекурсивный варианты). 26](#_Toc61120631)

[22. ДДП. Обход дерева. 27](#_Toc61120632)

[23. ДДП. Освобождение памяти из-под всего дерева. 27](#_Toc61120633)

[24. ДДП. Язык DOT, примеры использования. Утилита GraphViz. 27](#_Toc61120634)

[Область видимости, время жизни, связывание 29](#_Toc61120635)

[1. Что такое область видимости имени? 29](#_Toc61120636)

[2. Какие области видимости есть в языке Си? Приведите примеры. 29](#_Toc61120637)

[3. Какие правила перекрытия областей видимости есть в языке Си? Приведите примеры. 29](#_Toc61120638)

[4. Что такое блок? 30](#_Toc61120639)

[5. Какие виды блоков есть в языке Си? 30](#_Toc61120640)

[6. Что такое объявление? Приведите примеры. 30](#_Toc61120641)

[7. Что такое определение? Приведите примеры. 30](#_Toc61120642)

[8. Для чего нужны объявления? 30](#_Toc61120643)

[9. Чем отличаются определения и объявления? 30](#_Toc61120644)

[10. Что такое время жизни программного объекта? 30](#_Toc61120645)

[11. Какие виды времени жизни есть у переменных? 31](#_Toc61120646)

[12. Какие виды времени жизни есть у функций? 31](#_Toc61120647)

[13. Как время жизни влияет на область памяти, в которой располагается программный объект? 31](#_Toc61120648)

[14. Что такое связывание? 31](#_Toc61120649)

[15. Какие виды связывания есть в языке Си? 31](#_Toc61120650)

[16. Как связывание влияет на "свойства" объектного/исполняемого файла? Что это за "свойства"? 31](#_Toc61120651)

[17. Какими характеристиками (область видимости, время жизни, связывание) обладает переменная в зависимости от места своего определения? 31](#_Toc61120652)

[18. Какими характеристиками (область видимости, время жизни, связывание) обладает функция в зависимости от места своего определения? 32](#_Toc61120653)

[19. Какие классы памяти есть в языке Си? 32](#_Toc61120654)

[20. Для чего нужны классы памяти? 32](#_Toc61120655)

[21. Какие классы памяти можно использовать с переменными? С функциями? 32](#_Toc61120656)

[22. Сколько классов памяти может быть у переменной? У функции? 32](#_Toc61120657)

[23. Какие классы памяти по умолчанию есть у переменной? У функции? 32](#_Toc61120658)

[24. Расскажите о классе памяти auto. 32](#_Toc61120659)

[25. Расскажите о классе памяти static. 32](#_Toc61120660)

[26. Расскажите о классе памяти extern. 33](#_Toc61120661)

[27. Расскажите о классе памяти register. 33](#_Toc61120662)

[28. Для чего используется ключевое слово extern? 33](#_Toc61120663)

[29. Особенности совместного использования ключевых слов static и extern. 34](#_Toc61120664)

[30. Как описать автоматическую глобальную переменную? 34](#_Toc61120665)

[31. Какая переменная называется глобальной? 34](#_Toc61120666)

[32. Какая переменная называется локальной? 34](#_Toc61120667)

[33. Каким значением по умолчанию инициализируются автоматические переменные? 34](#_Toc61120668)

[34. Каким значением по умолчанию инициализируются переменные с глобальным временем жизни? 34](#_Toc61120669)

[35. Какие недостатки есть у использования глобальных переменных? 34](#_Toc61120670)

[36. Объектный файл, секции, таблица символов. 35](#_Toc61120671)

[37. Что делает компоновщик? 35](#_Toc61120672)

[38. Журналирование, подходы к реализации. 36](#_Toc61120673)

[Схема распределения памяти в программе на Си. Стек. Куча. 37](#_Toc61120674)

[1. Процесс запуска программы («превращения в процесс»). 37](#_Toc61120675)

[2. Абстрактное адресное пространство программы. 37](#_Toc61120676)

[3. Опишите достоинства и недостатки локальных переменных. 38](#_Toc61120677)

[4. Локальные переменные создаются в так называемой «автоматической памяти». Почему эта память так называется? 38](#_Toc61120678)

[5. Для чего в программе используется аппаратный стек? 38](#_Toc61120679)

[6. Что такое кадр стека? 38](#_Toc61120680)

[7. Для чего в программе используется кадр стека? Приведите примеры. 38](#_Toc61120681)

[8. Какие преимущества и недостатки есть у использования кадра стека? 39](#_Toc61120682)

[9. Что такое соглашение о вызове? 39](#_Toc61120683)

[10. Какое соглашение о вызове используется в языке Си? В чем оно заключается? 39](#_Toc61120684)

[11. Что такое переполнение буфера? Чем оно опасно? 39](#_Toc61120685)

[12. Почему нельзя из функции возвращать указатель на локальную переменную, определенную в этой функции? 39](#_Toc61120686)

[13. Для чего в программе используется куча? 40](#_Toc61120687)

[14. Происхождение термина «куча». 40](#_Toc61120688)

[15. Свойства области памяти, которая выделяется динамически. 40](#_Toc61120689)

[16. Как организована куча? 40](#_Toc61120690)

[17. Алгоритм работы функции malloc. 40](#_Toc61120691)

[18. Алгоритм работы функции free. 40](#_Toc61120692)

[19. Какие гарантии относительно выделенного блока памяти даются программисту? 40](#_Toc61120693)

[20. Что значит "освободить блок памяти" с точки зрения функции free? 41](#_Toc61120694)

[21. Преимущества и недостатки использования динамической памяти. 41](#_Toc61120695)

[22. Что такое фрагментация памяти? 41](#_Toc61120696)

[23. Выравнивание блока памяти, выделенного динамически. 41](#_Toc61120697)

[24. Что такое variable length array? 41](#_Toc61120698)

[25. Чем отличается статический массив от variable length array? 41](#_Toc61120699)

[26. Какую операцию языка Си пришлось реализовывать по-другому (не как для встроенных типов) специально для variable length array? 42](#_Toc61120700)

[27. Особенности использования variable length array. 42](#_Toc61120701)

[28. Справедлива ли для variable length array адресная арифметика? 42](#_Toc61120702)

[29. Как вы думаете почему variable length array нельзя инициализировать? 42](#_Toc61120703)

[30. Для чего используется variable length array? Приведите примеры. 42](#_Toc61120704)

[31. В какой области и «кем» выделяется память под массив переменной длины? 42](#_Toc61120705)

[32. Функция alloca. 42](#_Toc61120706)

[33. alloca VS VLA. 43](#_Toc61120707)

[Функции с переменным числом параметров 44](#_Toc61120708)

[1. Можно ли реализовать в языке Си функцию со следующим прототипом int f(...)? Почему? 44](#_Toc61120709)

[2. Покажите идею реализации функций с переменным числом параметров. 44](#_Toc61120710)

[3. Почему для реализации функций с переменным числом параметров нужно использовать возможности стандартной библиотеки? 44](#_Toc61120711)

[4. Опишите подход к реализации функций с переменным числом параметров с использованием стандартной библиотеки. Какой заголовочный файл стандартной библиотеки нужно использовать? Какие типы и макросы из этого файла вам понадобятся? Для чего? 44](#_Toc61120712)

[5. Какая особенность языка Си упрощает реализацию функций (с точки зрения компилятора) с переменным числом параметров? 45](#_Toc61120713)

[6. Почему при вызове va\_arg(argp, short int) (или va\_arg(argp, float)) выдается предупреждение? 45](#_Toc61120714)

[7. Какая "опасность" существует при использовании функций с переменным числом параметров? 45](#_Toc61120715)

[8. Как написать функцию, которая получает строку форматирования и переменное число параметров (как функция printf), и передает эти данные функции printf? (Подсказка: см. последний вариант реализации журналирования.) 45](#_Toc61120716)

[Препроцессор. inline-функции. 46](#_Toc61120717)

[1. Что делает препроцессор? В какой момент в процессе получения исполняемого файла вызывается препроцессор? 46](#_Toc61120718)

[2. На какие группы можно разделить директивы препроцессора? 46](#_Toc61120719)

[3. Какие правила справедливы для всех директив препроцессора? 46](#_Toc61120720)

[4. Что такое простой макрос? Как такой макрос обрабатывается препроцессором? Приведите примеры. 46](#_Toc61120721)

[5. Для чего используются простые макросы? 46](#_Toc61120722)

[6. Что такое макрос с параметрами? Как такой макрос обрабатывается препроцессором? Приведите примеры. 47](#_Toc61120723)

[7. Макросы с параметрами VS функции: преимущества и недостатки. 47](#_Toc61120724)

[8. Макросы с переменным числом параметров. Приведите примеры. 47](#_Toc61120725)

[9. Какими общими особенностями/свойствами обладают все макросы? 48](#_Toc61120726)

[10. Объясните правила использования скобок внутри макросов. Приведите примеры. 48](#_Toc61120727)

[11. Какие подходы к написанию "длинных" макросов вы знаете? Опишите их преимущества и недостатки. Приведите примеры. 48](#_Toc61120728)

[12. Какие предопределенные макросы вы знаете? Для чего эти макросы могут использоваться? 49](#_Toc61120729)

[13. Для чего используется условная компиляция? Приведите примеры. 49](#_Toc61120730)

[14. Директива #if VS директива #ifdef. 49](#_Toc61120731)

[15. Операция #. Примеры использования. 50](#_Toc61120732)

[16. Операция ##. Примеры использования. 50](#_Toc61120733)

[17. Особенности использования операций. 50](#_Toc61120734)

[18. Директива #error. Примеры использования. 50](#_Toc61120735)

[19. Директива #pragma (на примере once и pack). Примеры использования. 51](#_Toc61120736)

[20. Ключевое слово inline. 51](#_Toc61120737)

[21. Назовите основную причину, по которой ключевое слово inline было добавлено в язык Си. 51](#_Toc61120738)

[22. Подходы к реализации ключевого слова inline компилятором. Проанализируйте их недостатки. 51](#_Toc61120739)

[23. В чем разница между использованием <> и «» в директиве include? 51](#_Toc61120740)

[24. Можно ли операцию sizeof использовать в директивах препроцессора? Почему? 52](#_Toc61120741)

[Библиотеки 53](#_Toc61120742)

[1. Что такое библиотека? 53](#_Toc61120743)

[2. Какие функции обычно выносят в библиотеку? 53](#_Toc61120744)

[3. В каком виде распространяются библиотеки? Что обычно входит в их состав? 53](#_Toc61120745)

[4. Какие виды библиотек вы знаете? 53](#_Toc61120746)

[5. Преимущества и недостатки, которые есть у статических/динамических библиотек. 53](#_Toc61120747)

[6. Как собрать статическую библиотеку? 54](#_Toc61120748)

[7. Нужно ли "оформлять" каким-то специальным образом функции, которые входят в состав статической библиотеки? 54](#_Toc61120749)

[8. Как собрать приложение, которое использует статическую библиотеку? 54](#_Toc61120750)

[9. Нужно ли "оформлять" каким-то специальным образом исходный код приложения, которое использует статическую библиотеку? 54](#_Toc61120751)

[10. Как собрать динамическую библиотеку (Windows/Linux)? 55](#_Toc61120752)

[11. Нужно ли "оформлять" каким-то специальным образом функции, которые входят в состав динамической библиотеки (Windows/Linux)? 56](#_Toc61120753)

[12. Какие способы компоновки приложения с динамической библиотекой вы знаете? Назовите их преимущества и недостатки. 56](#_Toc61120754)

[13. Что такое динамическая компоновка? 56](#_Toc61120755)

[14. Что такое динамическая загрузка (Windows/Linux)? 56](#_Toc61120756)

[15. Нужно ли "оформлять" каким-то специальным образом исходный код приложения, которое использует динамическую библиотеку (Windows/Linux)? 56](#_Toc61120757)

[16. Особенности реализации функций, использующих динамическое выделение памяти, в динамических библиотеках. 57](#_Toc61120758)

[17. Ключи -I, -l, -L компилятора gcc. 57](#_Toc61120759)

[18. Проблемы использования динамической библиотеки, реализованной на одном языке программирования, и приложения, реализованного на другом языке программирования. 57](#_Toc61120760)

[19. Модуль ctypes. Загрузка библиотеки. Представление стандартных типов языка Си. Импорт функций из библиотеки. Проблемы, которые при этом возникают. 57](#_Toc61120761)

[20. Написание модуля расширения для Python (основные шаги). 58](#_Toc61120762)

[Бинарные операции. Битовые поля. 59](#_Toc61120763)

[1. К каким типам в языке Си применимы битовые операции? 59](#_Toc61120764)

[2. Особенности использования битовых операций со знаковыми целыми типами. 59](#_Toc61120765)

[3. Какие битовые операции есть в языке Си? Приведите примеры. 59](#_Toc61120766)

[4. Как установить указанный бит? 59](#_Toc61120767)

[5. Как сбросить указанный бит? 60](#_Toc61120768)

[6. Как проверить, что указанный бит установлен? 60](#_Toc61120769)

[7. Как изменить значение указанного бита на противоположное? 60](#_Toc61120770)

[8. Как установить сразу несколько бит? 60](#_Toc61120771)

[9. Как получить значение нескольких бит? 61](#_Toc61120772)

[10. С помощью какой битовой операции можно разделить целое число на 2^n? 61](#_Toc61120773)

[11. С помощью какой битовой операции можно умножить целое число на 2^n? 61](#_Toc61120774)

[12. Битовые операции VS логические операции. 61](#_Toc61120775)

[13. Что такое битовое поле? 62](#_Toc61120776)

[14. Преимущества и недостатки битовых полей по сравнению с битовыми операциями. 62](#_Toc61120777)

[15. Что задает значение CHAR\_BIT? В каком стандартном заголовочном файле его можно найти? 62](#_Toc61120778)

[16. Существуют ли значения SHORT\_INT\_BIT, INT\_BIT и т.д.? Почему? 62](#_Toc61120779)

[17. Напишите функцию, которая использует битовые операции, для вывода числа в двоичной системе счисления. 62](#_Toc61120780)

[18. Упакуйте четыре символа в беззнаковое целое. Длина беззнакового целого равна 4. 63](#_Toc61120781)

[19. «Распакуйте» беззнаковое целое число в четыре символа. Длина беззнакового целого равна 4. 63](#_Toc61120782)

[Неопределенное поведение 64](#_Toc61120783)

[1. Что такое побочный эффект? 64](#_Toc61120784)

[2. Какие выражения стандарт С99 относит к выражениям с побочным эффектом? 64](#_Toc61120785)

[3. Почему порядок вычисления подвыражений в языке Си неопределен? 64](#_Toc61120786)

[4. Порядок вычисления каких выражения в языке Си определен? 64](#_Toc61120787)

[5. Что такое точка следования? 64](#_Toc61120788)

[6. Какие точки следования выделяет стандарт С99? 64](#_Toc61120789)

[7. Почему необходимо избегать выражений, которые дают разный результат в зависимости от порядка их вычисления? 64](#_Toc61120790)

[8. Какие виды "неопределенного" поведения есть в языке Си? 65](#_Toc61120791)

[9. Почему "неопределенное" поведение присутствует в языке Си? 65](#_Toc61120792)

[10. Какой из видов "неопределенного" поведения является самым опасным? Чем он опасен? 65](#_Toc61120793)

[11. Как бороться с неопределенным поведением? 65](#_Toc61120794)

[12. Приведите примеры неопределенного поведения. 65](#_Toc61120795)

[13. Приведите примеры поведения, зависящего от реализации. 65](#_Toc61120796)

[14. Приведите примеры неспецифицированного поведения. 66](#_Toc61120797)

[15. АТД 66](#_Toc61120798)

[16. Что такое модуль? 66](#_Toc61120799)

[17. Из каких частей состоит модуль? Какие требования предъявляются к этим частям? 66](#_Toc61120800)

[18. Назовите преимущества модульной организации программы. Приведите примеры. 66](#_Toc61120801)

[19. Какие виды модулей вы знаете? Приведите примеры. 66](#_Toc61120802)

[20. Что такое тип данных? 67](#_Toc61120803)

[21. Что такое абстрактный тип данных? 67](#_Toc61120804)

[22. Какие требования выдвигаются к абстрактному типу данных? 67](#_Toc61120805)

[23. Абстрактный объект VS абстрактный тип данных. 67](#_Toc61120806)

[24. Средства реализации модулей в языке Си. 67](#_Toc61120807)

[25. Что такое неполный тип данных в языке Си? 67](#_Toc61120808)

[26. Приведите примеры описания неполного типа данных? (А кроме структур ;) ?) 67](#_Toc61120809)

[27. Какие действия можно выполнять с неполным типом данных? 68](#_Toc61120810)

[28. Для чего при реализации абстрактного типа данных используется неполный тип данных языка Си? 68](#_Toc61120811)

[29. Проблемы реализации АТД на языке Си. 68](#_Toc61120812)

[30. Есть ли в стандартной библиотеке языка Си примеры абстрактных типов данных? 68](#_Toc61120813)

# Указатель на void. Динамическое выделение памяти

## 1. Для чего используется указатель на void? Приведите примеры.

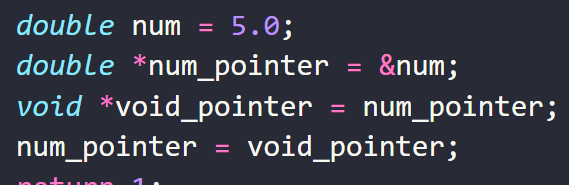
Тип указателя void используется, когда тип объекта неизвестен:

* ссылка на произвольный участок памяти, независимо от хранимых там объектов
* передача в функцию указателя на любой тип данных

Пример: структура данных список, в узле списка храниться лишь указатель на данные, сами данные хранятся в другом месте. Использование указателя на void позволяет реализовать универсальную структуру данных для любых типов обрабатываемой информации.

## 2. Каковы особенности использования указателя на void? Приведите примеры.

* В языке Си допускается присваивание указателя типа void указателю любого другого типа и наоборот без явного преобразования типа.



* Указатель типа void нельзя разыменовать.
* К указателям типа void нельзя применять адресную арифметику.

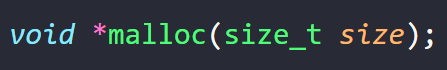
## 3. Функции для выделения и освобождения памяти malloc, calloc, free. Порядок работы и особенности использования этих функций.

Данные функции объявлены в заголовочном файле stdlib.h. Функция malloc только выделяет блок памяти (без инициализации), функция calloc выделяет блок памяти и инициализирует его нулями, функция free освобождает выделенный блок памяти.

Функции выделения не создают переменную, они лишь выделяют область памяти. В качестве результата они возвращают адрес расположения этой области памяти в памяти компьютера, т.е. указатель. Т.к. функции не знают данные какого типа будут располагаться в выделенном блоке, все они возвращают указатель на void.

В случае если запрашиваемый блок памяти не удалось выделить, функции вернут значение NULL. Выделенный блок после использования нужно обязательно освободить, сделать это можно с помощью функции free.

malloc:



calloc:



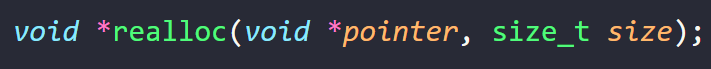
free:



Если значением data\_ptr является нулевой указатель, то ничего не происходит.

Если data\_ptr является указателем на блок памяти, который не был получен с помощью функций выделения памяти, то поведение функции free неопределенно.

## 4. Функция realloc. Особенности использования.



* Если указатель нулевой, а размер нет, то происходит выделение памяти аналогично malloc.
* Если указатель и размер нулевые, то происходит освобождение памяти аналогично free.
* Если указатель и размер ненулевые, то происходит перевыделение памяти. (в худшем случае будет выделена новая область память, в нее будут скопированы данные из старой области памяти, старая область памяти будет освобождена).

При использовании функции realloc нужно использовать временный указатель, т.к. в случае ошибки выделения памяти произойдет утечка.

## 5. Общие «свойства» функций malloc, calloc, realloc.

Функции выделения не создают переменную, они лишь выделяют область памяти. В качестве результата они возвращают адрес расположения этой области памяти в памяти компьютера, т.е. указатель. Т.к. функции не знают данные какого типа будут располагаться в выделенном блоке, все они возвращают указатель на void.

В случае если запрашиваемый блок памяти не удалось выделить, функции вернут значение NULL. Выделенный блок после использования нужно обязательно освободить, сделать это можно с помощью функции free.

## 6. Функция выделения памяти и явное приведение типа: за и против.

Преимущества:

* компиляция с помощью C++ компилятора.
* до стандарта ANSI C был другой прототип (функции возвращали char \*).
* дополнительная проверка аргументов разработчиком.

Недостатки:

* начиная с ANSI C приведение не нужно.
* может скрыть ошибку, если забыли подключить stdlib.h
* в случае изменения типа указателя придется менять и тип в приведении.

## 7. Особенности выделения 0 байт памяти.

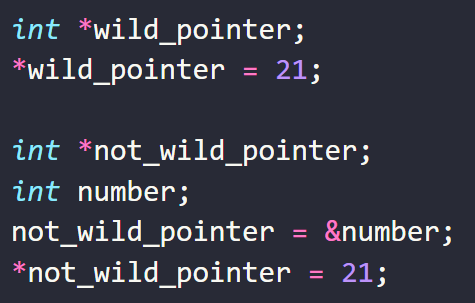
Результат вызова функций malloc, calloc, realloc, когда запрашиваемый размер блока данных равен 0, зависит от реализации (либо вернется нулевой указатель, либо вернется «нормальный указатель», но его нельзя разыименовать). Поэтому перед вызовом этих функций нужно убедиться, что запрашиваемый размер блока не равен 0.

## 8. Способы возвращения динамического массива из функции.

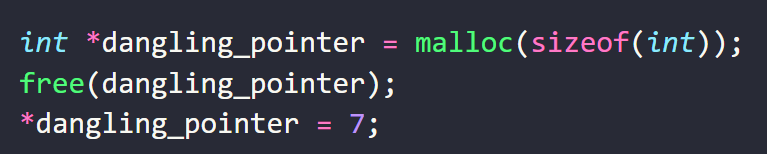
* Передача указателя на массив и его размера в качестве параметров функции и возврат кода ошибки.
* Передача только размера массива и возврат указателя на массив.

## 9. Типичные ошибки при работе с динамической памятью.

* Неверный расчет количества выделяемой памяти.
* Отсутствие проверки успешности выделения памяти.
* Утечки памяти.
* Логические ошибки:
* wild pointer – использование не проинициализированного указателя.



* dangling pointer – использование указателя после освобождения памяти.



* изменение указателя, который вернула функция выделения памяти.
* двойное освобождение памяти.
* освобождение невыделенной или нединамической памяти.
* выход на границы динамического массива.
* и другие…

## 10. Подходы к обработке ситуации отсутствия свободной памяти при выделении.

* Возвращение ошибки (return failure).
* Ошибка сегментации (segfault). Обратная сторона – проблемы с безопасностью.
* Аварийное завершение (abort). (функция xmalloc)
* Восстановление (recovery).

# Указатели на функцию

## 1. Для чего используется указатель на функцию? Приведите примеры.

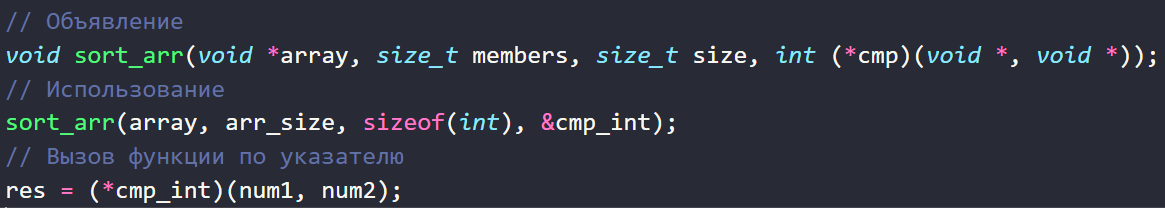
Указатель на функцию необходим для реализации универсальных функций:

* для обработки структур данных, подход к которым алгоритмически одинаков, однако их элементы имеют разный тип данных (универсальная сортировка массива)
* для реализации функций обхода структуры данных и выполнения какой-то определенной операции над ее элементами (работа с бинарными деревьями поиска).

Более сложные варианты использования указателей на функции:

* функции обратного вызова (callback).
* таблицы переходов (jump table).
* динамическое связывание (binding).

## 2. Указатель на функцию: описание, инициализация, вызов функции по указателю.

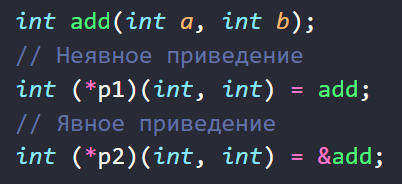


## 3. Функция qsort, примеры использования.

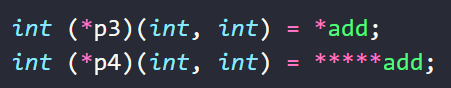
См. пример выше.

## 4. Особенности использования указателей на функцию.

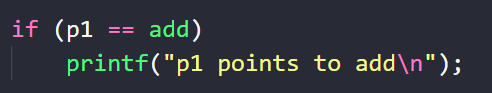
* Согласно стандарту C99 выражение из имени функции неявно преобразуется в указатель на функцию.



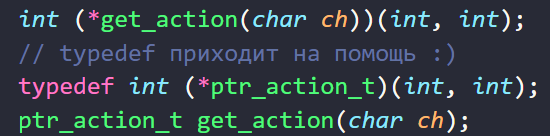
* Операция разыменования для указателя на функцию возвращает саму функцию, которая неявно преобразуется в указатель на функцию.



* Указатели на функции можно сравнивать.



* Функции могут возвращать указатель на функцию.



## 5. Указатель на функцию и адресная арифметика.

К указателям на функции адресная арифметика не применима.

## 6. Указатель на функцию и указатель на void.

Согласно стандарту C99, указатель на функцию не может быть преобразован к указателю на void и наоборот.

Но POSIX требует, чтобы такое преобразование было возможно при работе с динамическими библиотеками:

* С99 J.5.7 Function pointer casts (расширение стандарта)
* POSIX dlsym RATIONALE
* Generic Function Pointer C2X (будущее (?))

# Make

## 1. Утилита make: назначение, входные данные, идея алгоритма работы.

Утилита make предназначена для автоматизации процесса преобразования файлов из одной формы в другую. Утилита использует информацию из make-файла и время последнего изменения каждого файла для того, чтобы решить, какие файлы нужно обновить.

Входные данные – сценарий сборки, цель для сборки, дата изменения файлов.

## 2. Разновидности утилиты make.

* GNU Make (рассматриваем в курсе)
* BSD Make
* Microsoft Make (nmake)

## 3. Сценарий сборки проекта: название файла, структура сценария сборки.

Имя файла сценария по умолчанию makefile.

Структура сценария сборки:

цель: зависимость\_1 ... зависимость\_n

[tab]команда\_1

...

[tab]команда\_m

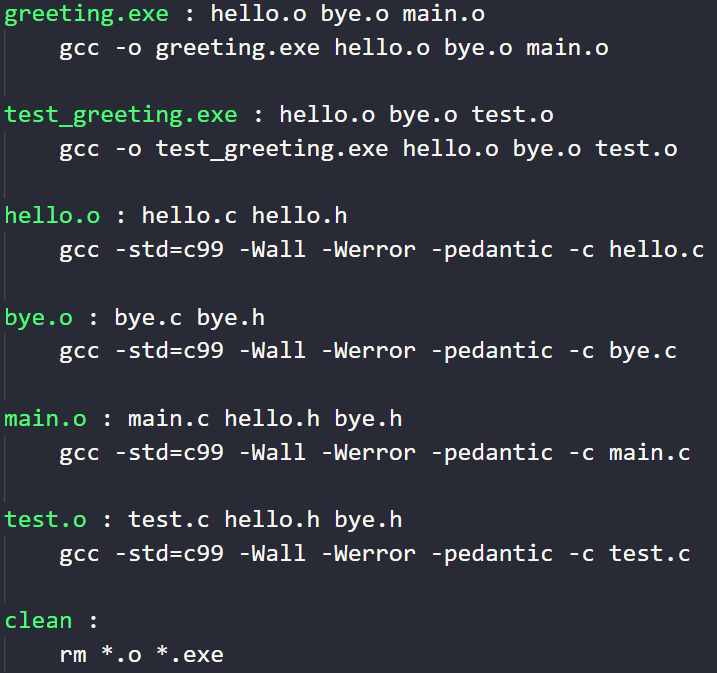
## 4. Правила: составные части, особенности использования правил в зависимости от составных частей.

См. выше.

## 5. Особенности выполнения команд.

* Нулевой код возврата может прервать выполнение сценария.
* Каждая команда выполняется в своем shell.

## 6. Простой сценарий сборки.



## 7. Алгоритм работы утилиты make на примере простого сценария сборки.

Первый запуск make:

* make читает сценарий сборки и начинает выполнять первое правило
* Для выполнения этого правила необходимо сначала обработать зависимости
* make ищет правило для создания файла hello.o
* Файл hello.o отсутствует, файлы hello.c и hello.h существуют. Следовательно, правило для создания hello.o может быть выполнено
* Аналогично обрабатываются зависимости bye.o и main.o.
* Все зависимости получены, теперь правило для построения greeting.exe может быть выполнено

Второй запуск make (hello.c был изменен):

* make читает сценарий сборки и начинает выполнять первое правило
* Для выполнения этого правила необходимо сначала обработать зависимости
* make ищет правило для создания файла hello.o
* Файлы hello.o, hello.c и hello.h существуют, но время изменения hello.o меньше времени изменения hello.c. Придется пересоздать файл hello.o
* Аналогично обрабатываются зависимости bye.o и main.o, но эти файлы были изменены позже соответствующих си-файлов, т.е. ничего делать не нужно.
* Все зависимости получены. Время изменения greeting.exe меньше времени изменения hello.o. Придется пересоздать greeting.exe

## 8. Ключи запуска утилиты make.

* Ключ «-f» (используется для указания имени файла, где расположен сценарий сборки, если имя файла отличается от стандартного)

make -f makefilename

* Ключ «-B» (используется для безусловного выполнения правил)

make -B

* Ключ «-n» (используется для вывода команд без их выполнения)

make -n

* Ключ «-i» (используется для игнорирования ошибок при выполнении команд)

make -i

* Ключ «-s» (используется для отключения печати выполняемых команд)

make -s

* Ключ «-h» (используется для печати сообщения со всеми ключами)

make -h

* Ключ «-j[N]» (используется для ограничения количества выполняемых работ одновременно)

make -j4

## 9. Использование переменных. Примеры использования.

Описание переменной:

VAR\_NAME := value

Чтобы получить значение переменной, необходимо ее имя заключить в круглые скобки и перед ними поставить символ '$'. Переменные можно использовать для описания зависимостей, директорий, флагов компиляции и др.

## 10. Неявные правила и переменные.

В утилите make описаны типичные неявные правила, чтобы пользователю не требовалось определять их детально в случае необходимости.

Ключ «-p» выводить все неявные правила и переменные на экран, ключ «-r» запрещает их использования во время работы утилиты.

## 11. Автоматические переменные и их использование.

Автоматические переменные — это переменные со специальными именами, которые «автоматически» принимают определенные значения перед выполнением описанных в правиле команд.

* Переменная "$^" означает "список зависимостей".
* Переменная "$@" означает "имя цели".
* Переменная "$<" является просто первой зависимостью.

## 12. Шаблонные правила. Примеры использование.

Шаблонное правило позволяет сделать универсальное правило сборки для файлов, имеющих одно расширение.

%.расш\_файлов\_целей : %.расш\_файлов\_зав

[tab]команда\_1

...

[tab]команда\_m

## 13. Условные конструкции в сценарии сборки. Примеры использования.

В случае необходимости можно использовать условные операторы в сценариях сборки с целью оптимизации и лучшего восприятия.

Пример:

ifeq ($(mode), debug)

  # Отладочная сборка: добавим генерацию отладочной информации

CFLAGS += -g3

endif

ifeq ($(mode), release)

# Финальная сборка: исключим отладочную информацию и утверждения

CFLAGS += -DNDEBUG -g0

endif

## 14. Переменные, зависящие от цели. Примеры использования.

Аналогично условным операторам можно использовать переменные, зависящие от цели.

debug : CFLAGS += -g3

debug : greeting.exe

release : CFLAGS += -DNDEBUG -g0

release : greeting.exe

## 15. Автоматическая генерация зависимостей.

# Все c-файлы (или так SRCS := $(wildcard \*.c))

SRCS := hello.c bye.c test.c main.c

…

%.d : %.c

$(CC) -M $< > $@

# $(SRCS:.c=.d) – заменяет в переменной SRCS имена файлов с расширением "c" на имена с расширением "d"

include $(SRCS:.c=.d)

…

# Динамические матрицы

## 1. Представление динамической матрицы с помощью одномерного массива. Преимущества и недостатки.

Память выделяется один раз под все элементы матрицы в виде одномерного массива.

Преимущества:

* Простота выделения и освобождения памяти.
* Возможность использовать как одномерный массив.

Недостатки:

* Отладчик памяти не может отследить выход за пределы строки.
* Необходимо осторожно работать с адресами элементов массива.

## 2. Представление динамической матрицы с помощью массива указателей на строки/столбцы. Преимуществ и недостатки.

Память выделяется сначала под массив указателей на строки/столбцы (каждый указатель указывает на конкретную строку/столбец, в памяти данные расположены неоднородно), затем под каждый указатель выделяется память для отдельной строки/столбца.

Преимущества:

* Возможность обмена строк/столбцов через указатели.
* Отладчик памяти может отследить выход за пределы строки.

Недостатки:

* Сложность выделения и освобождения памяти.
* Неоднородность памяти выделенной под элементы матрицы.

## 3. Объединенный подход для представления динамической матрицы (отдельное выделение памяти под массив указателей и массив данных). Преимуществ и недостатки.

Аналогичен способу выше, за исключением однородного расположения данных памяти.

Преимущество:

* Относительная простота выделения и освобождения памяти.
* Возможность использовать как одномерный массив.
* Перестановка строк через обмен указателей (операция обмена первой строки с какой-либо не тривиальна).

Недостатки.

* Относительная сложность начальной инициализации.
* Потенциальная ошибка обмена с первой строкой.
* Отладчик использования памяти не может отследить выход за пределы строки.

## 4. Объединенный подход для представления динамической матрицы (массив указателей и массив данных располагаются в одной области). Преимуществ и недостатки.

Память выделяется однородно под массив указателей и данных.

Преимущества:

* Простота выделения и освобождения памяти.
* Возможность использовать как одномерный массив.
* Перестановка строк через обмен указателей.

Недостатки:

* Сложность начальной инициализации.
* Отладчик использования памяти не может отследить выход за пределы строки.

## 5. Необходимо реализовать функцию, которая может обрабатывать как статические, так и динамические матрицы. Какими способами это можно сделать?

В случае хранения одномерным массивом, нужно передавать \*а, если нет - то в случае со статической матрицей придется делать массив указателей на строки или столбцы и передавать \*\*а.

# Чтение сложных объявлений

## 1. Умение читать сложные объявления и использовать это на практике.

Объявление может содержать только **один основной тип**, и он всегда находится слева выражения. Основные типы дополняются производными типами, в Си их три:

1)**\* — указатель на ...**

Обозначается символом \*, и важно понимать, что указатель **всегда** на что-нибудь указывает.

2)**[] — массив из…**

Массив может быть безразмерный — [], а может быть и размерный [10]. Правда размерный массив или нет, это неважно при чтении объявлений (обычно все же пишется размер массива). Должно быть понятно, что массив всегда «массив из чего-нибудь».

3)**() — функция, возвращающая ...**

Обычно обозначается парой круглых скобок (), но также возможно, что внутри их будут модели параметров. Список параметров (если он есть) не играет существенной роли при чтении объявлений, и мы его обычно игнорируем. Заметим, что круглые скобки, используемые для обозначения функций, отличаются от скобок служащих для группировки: группирующие скобки окружают переменные, тогда как скобки для обозначения функция находятся справа. Функция не имеет смысла если она ничего не возвращает (когда мы объявляем функцию с возвращаемым типом значения void, то это просто выглядит как будто функция возвращает значения типа void)

Производные типы **всегда** что-то модифицируют, будь то основной тип или производный, и что бы правильно читать объявления, всегда нужно вставлять предлог («на», «из», «возвращающая»). Используя при чтении «указатель» вместо «указатель на», вы точно прочитаете объявление неправильно.

Семантические ограничения:

* Невозможно создать массив функций.
* Функция не может возвращать функцию.
* Функция не может вернуть массив.
* В массиве только левая лексема [] может быть пустой.
* Тип void ограниченный.

void x;       // ошибка

void x[5];    // ошибка

# Строки/структуры и динамическое выделение памяти.

## 1. Функции, возвращающие динамическую строку: strdup/strndup, getline, snprintf/asprintf.

a) strdup (POSIX (при компиляции -std=gnu99))



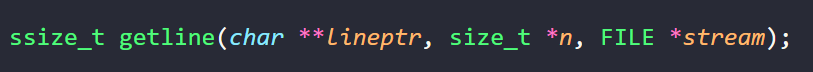
Создает копию указанной нуль-терминированной строки в куче. (копия с терминированным нулем)

б) strndup (POSIX)



Создает копию, состоящую не более чем из указанного количества байт, из указанной нуль-терминированной строки в куче (если скопирована не вся строка, то терминальный нуль добавляется в откопированную часть).

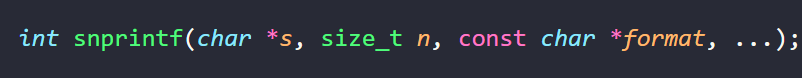
в) getline (изначально GNU, затем POSIX)



Выполняет чтение строки из потока.

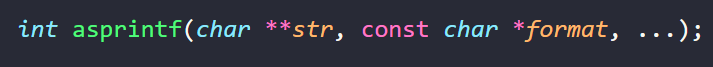
lineptr - либо NULL (и тогда в n - 0), либо указатель на буфер, выделенный с помощью malloc (и тогда в n - размер буфера). Если буфера не хватает, он будет перевыделен.

г) snprintf (стандартная функция)



Производит печать в указанную строку, как будто была вызвана функция printf. Терминальный нуль записывается автоматически. Возвращает количество символов, записанных в строку, если n достаточно большое (без учета терминального нуля). В случае ошибки возвращает отрицательное значение.

д) asnprintf (GNU)



Работает аналогично snprintf, за исключением самостоятельно выделения памяти под строку.

## 2. Feature Test Macro.

Макросы тестирования свойств позволяют программисту контролировать, какие определения будут доступны из системных заголовочных файлов при компиляции программы.

Для корректной работы программы макрос тестирования свойств должен быть определен в программе до включения всех заголовочных файлов.

Некоторые макросы тестирования свойст полезны для создания переносимых приложений: они позволяют блокировать нестандартные определения. Другие же, наоборот, можно для разблокировки этих определений. Например, \_GNU\_SOURCE позволяет использовать нестандартную функцию getline.

Влияние каждого макроса тестирования свойст можно узнать из содержимого заголовочного файла features.h.

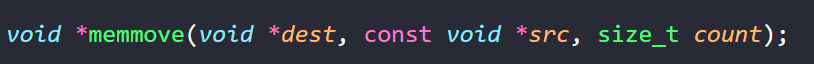
## 3. Функции memcpy, memmove, memcmp, memset.

а) memcpy



Копирует указанное количество байт из одной области памяти в другую. Если указатели нулевые, не валидные или выполняется попытка записи за пределами пункта назначения, то поведение функции неопределенно.

б) memmove



Копирует указанное количество байт из одной области памяти в другую. Объекты могут перекрываться. При копировании данные копируются сначала во временный массив, а затем из него в пункт назначения. Если указатели нулевые, не валидные или выполняется попытка записи за пределами пункта назначения, то поведение функции неопределенно.

в) memcmp



Выполняет побайтовое сравнение указанного количества байт двух областей памяти. Если указатели нулевые или выполняется попытка доступа за пределами областей, то поведение функции неопределенно.

г) memset



Выполняет копирование и инициализацию указанного количества байт памяти заданным значением указанной области памяти. Если указатель нулевой или выполняется попытка доступа за его пределами, то поведение функции неопределенно.

## 4. Структуры с полями указателями и особенности их использования.

В Си определена операция присваивания для структурных переменных одного типа. Эта операция фактически эквивалента копированию области памяти, занимаемой одной переменной, в область памяти, которую занимает другая.

При этом реализуется стратегия так называемого *«поверхностного копирования»* (англ., *shallow coping*), при котором копируется содержимое структурной переменной, но не копируется то, на что могут ссылать поля структуры.

Стратегия так называемого *«глубокого копирования»* (англ., *deep coping*) подразумевает создание копий объектов, на которые ссылаются поля структуры.

## 5. «Поверхностное» копирование VS «глубокое» копирование.

Поверхностное копирование:

* Простота реализации.
* Возможность ошибок при работе в дальнейшем.

Глубокое копирование:

* Дополнительные трудности с реализацией по сравнению с поверхностным.
* Необходимость дополнительной памяти и обращения к куче (соответственно дополнительные временные расходы).
* Необходимость контроля новой выделенной памяти.

## 6. «Рекурсивное» освобождение памяти для структур с динамическими полями.

Перед освобождением памяти из-под структуры необходимо освободить память из-под всех ее динамических полей.

Рекурсивность заключается в том, что сначала необходимо освободать память из-под вложенных полей, поднимаясь на уровни выше.

Например, если есть динамически выделенная структура, которая ссылается на другую структуру, в которой есть динамическая строка, сначала необходимо освободить память из под строки, потом из-под второй структуры, а лишь затем – из-под первой.

## 7. Структуры переменного размера. Приведите примеры.

*TLV (Type (или Tag) Length Value)* - схема кодирования произвольных данных в некоторых телекоммуникационных протоколах.

* *Type* – описание назначения данных.
* *Length* – размер данных (обычно в байтах).
* *Value* – данные.

Первые два поля имеют фиксированный размер.

TLV кодирование используется в:

* семействе протоколов TCP/IP
* спецификация PC/SC (smart cards)
* ASN.1

Преимущества TLV кодирования:

* простота разбора;
* «тройки» TLV c неизвестным типом (тегом) могут быть пропущены при разборе;
* «тройки» TLV могут размещаться в произвольном порядке;
* «тройки» TLV обычно кодируются двоично, что позволяет выполнять разбор быстрее и требует меньше объема по сравнению с кодированием, основанном на текстовом представлении.

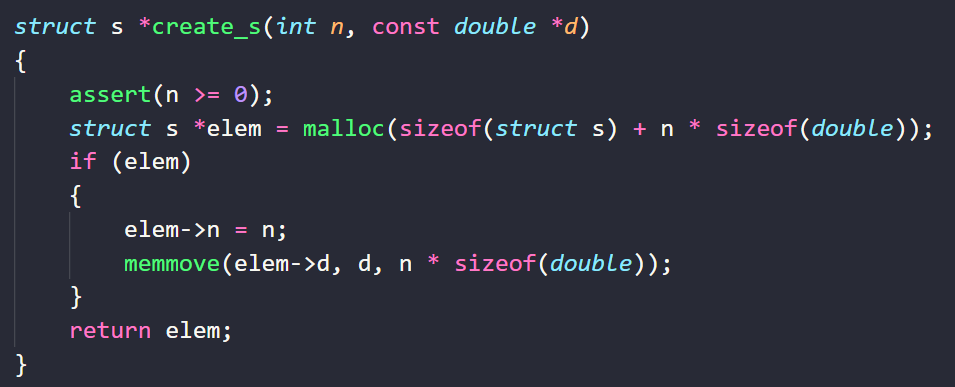
## 8. Что такое «flexible array member»? Какие особенности использование есть у этих полей? Для чего они нужны? Приведите примеры.



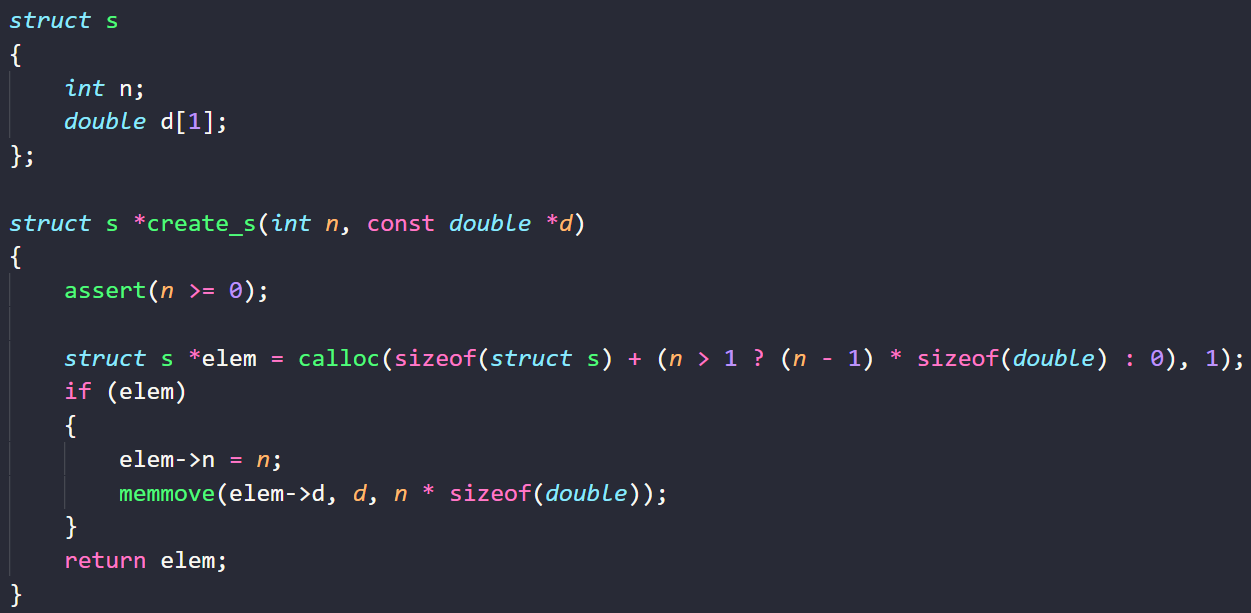
Особенности полей:

* Подобное поле должно быть последним.
* Нельзя создать массив структур с таким полем.
* Структура с таким полем не может использоваться как член в «середине» другой структуры.
* Операция sizeof не учитывает размер этого поля (возможно, за исключением выравнивания).
* Если в этом массиве нет элементов, то обращение к его элементам – неопределенное поведение.

Пример создания:



## 9. Flexible array member до C99.



## 10. Flexible array member VS поле-указатель.

* Экономия памяти.
* Локальность данных (data locality).
* Атомарность выделения памяти.
* Не требует «глубокого» копирования и освобождения (освобождаем только из-под структуры).

# Динамически расширяемый массив. Cписки. Двоичные деревья поиска.

## 1. Дайте определение массива.

*Массив* – структура данных, состоящая из объектов одного типа, объекты располагаются в памяти однородно, доступ к каждому элементу осуществляется за константное время.

## 2. Дайте определение линейного односвязного списка.

*Линейный односвязный список* – структура данных, состоящая из узлов, каждый из которых ссылается на следующий узел списка (данные в памяти располагаются неоднородно).

## 3. Сравните массив и линейный односвязный список.

Данные структуры данных используют разную стратегию выделения памяти, списки занимают больший объем памяти, чем массивы, время доступа к элементу в списке зависит от его расположения, списки более сложны для обработки, чем массивы.

## 4. Динамически расширяемый массив.

Динамически расширяемый массив аналогичен обычному за исключением того, что в случае его заполненности и попытки добавления элемента происходит перевыделение памяти крупными блоками, аналогично с удалением (правила перевыделения зависят от конкретной реализации).

## 5. ДРМ. Описание типа.

Для описания типа достаточно хранить указатель на область памяти, текущий размер, а также максимальный.

## 6. ДРМ. Добавление нового элемента.

При добавлении элемента в ДРМ в случае необходимости нужно перевыделить память, новый элемент записать в конец и сдвинуть указатель на конец вправо.

## 7. ДРМ. Удаление элемента.

При удалении элемента из ДРМ достаточно переписать последний элемент на место удаляемого если порядок не важен или сдвинуть элементы за удаляемым влево если важен, а также сдвинуть указатель на конец влево (размер области памяти уменьшается в зависимости от реализации).

## 8. ДРМ. Особенности использования.

Особенности:

* Для уменьшения потерь при распределении памяти изменение размера должно происходить относительно крупными блоками.
* Для простоты реализации указатель на выделенную память должен храниться вместе со всей информацией, необходимой для управления динамическим массивом.
* Удвоение размера массива при каждом вызове realloc сохраняет средние «ожидаемые» затраты на копирование элемента.
* Поскольку адрес массива может измениться, программа должна обращаться к элементам массива по индексам.
* Благодаря маленькому начальному размеру массива, программа сразу же «проверяет» код, реализующий выделение памяти.
* При удалении нужно обращать внимание на порядок.

## 9. ДРМ. Почему при добавлении нового элемента память необходимо выделять блоками, а не под один элемент?

Для уменьшения потерь по времени при распределении памяти изменение размера должно происходить относительно крупными блоками.

## 10. Линейный односвязный список.

*Линейный односвязный список* – структура данных, состоящая из узлов, каждый из которых ссылается на следующий узел списка (данные в памяти располагаются неоднородно).

## 11. ЛОС. Описание типа.

Для описания списка достаточно знать его «голову». Каждый узел списка хранит данные или указатель на них и указатель на следующий узел списка.

## 12. ЛОС. Добавление нового элемента в начало/конец списка.

При добавлении узла в начало необходимо, чтобы элемент указывал на голову списка, а голове присвоить указатель на элемент.

При добавлении узла в конец нужно пройти до конца списка и последний элемент должен указывать на новый элемент (у нового элемента нужно занулить указатель).

## 13. ЛОС. Вставка элемента перед/после указанного.

При вставке элемента после указанного нужно, чтобы новый узел указывал на следующий после указанного, а текущий на новый.

При вставке элемента перед указанным нужно скопировать данные и произвести операцию, указанную выше.

## 14. ЛОС. Удаление элемента из списка.

При удалении заданного элемента из списка нужно, чтобы предыдущий указывал на следующий, после чего необходимо освободить память.

## 15. ЛОС. Обход списка.

При обходе списка происходит последовательное движение по его элементам.

## 16. ЛОС. Удаление памяти из-под всего списка.

Необходимо выполнить обход списка, сохраняя текущий указатель во временный и последовательно освобождая память.

## 17. ЛОС. Возможные улучшения "классической" реализации.

Представление элемента списка

* Универсальный элемент (void\*).

Двусвязные списки

* Требует больше ресурсов.
* Поиск последнего и удаление текущего – операции порядка O (1).

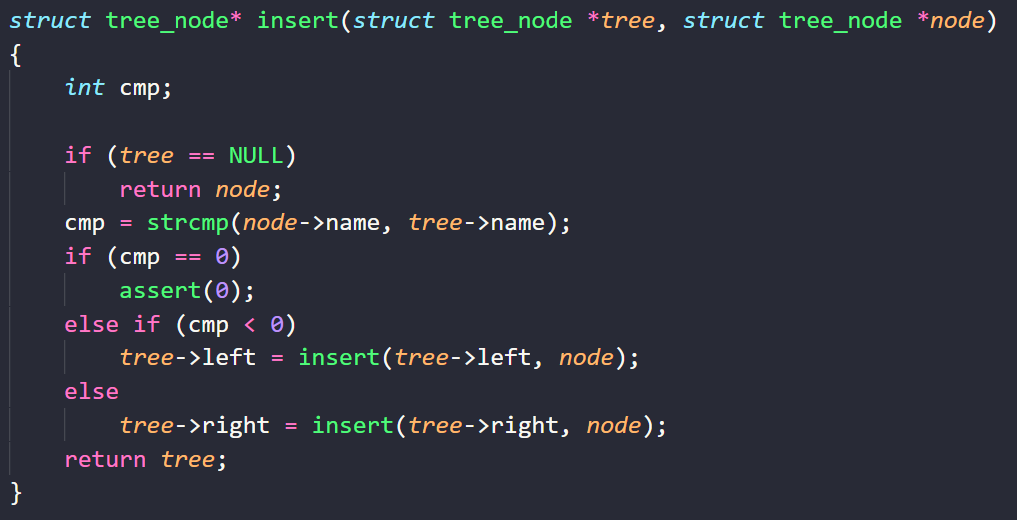
## 18. Двоичное дерево поиска.

*Двоичным деревом поиска* называют дерево, все вершины которого упорядочены (как правило левый потомок меньше родителя, а правый больше), каждая вершина имеет не более двух потомков (назовём их левым и правым), и все вершины, кроме корня, имеют родителя.

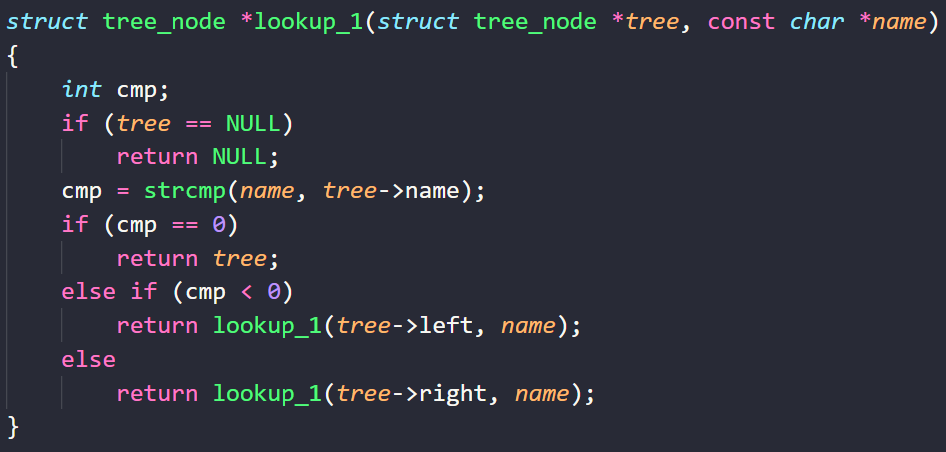
## 19. ДДП. Описание типа.

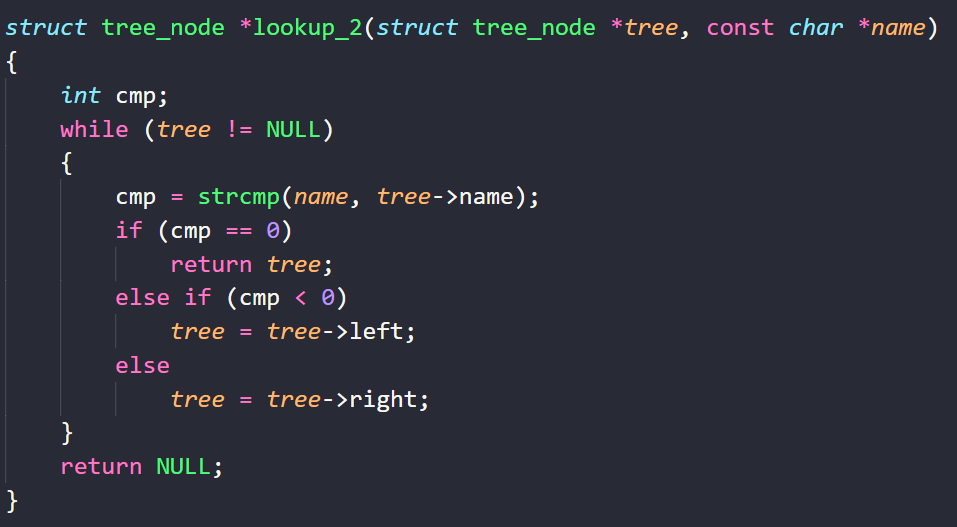
Для описания узла дерева достаточно хранить указатель на данные, а также указатель на левое и правое поддерево.

## 20. ДДП. Добавление элемента.

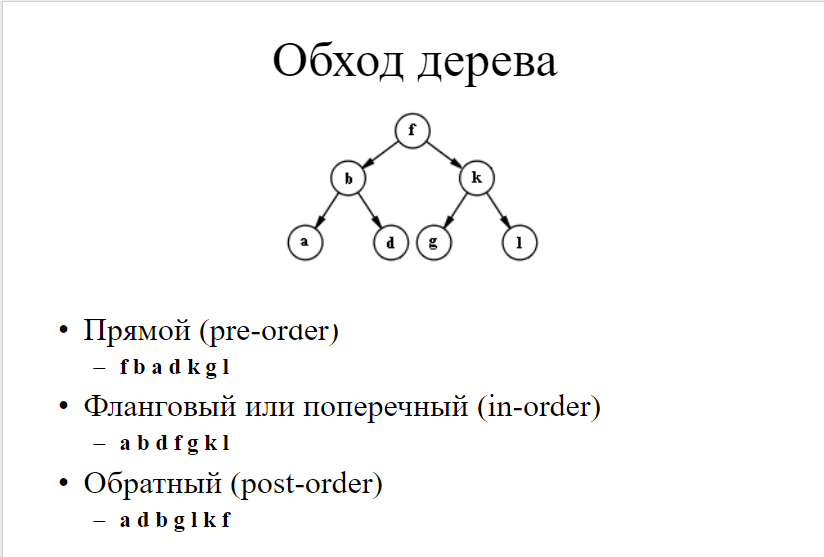


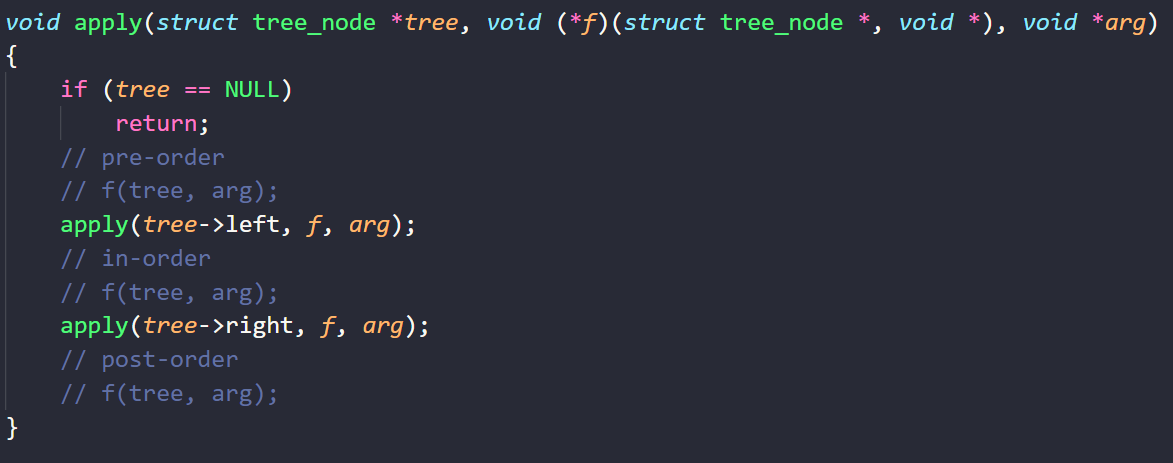
## 21. ДДП. Поиск элемента (рекурсивный и не рекурсивный варианты).





## 22. ДДП. Обход дерева.



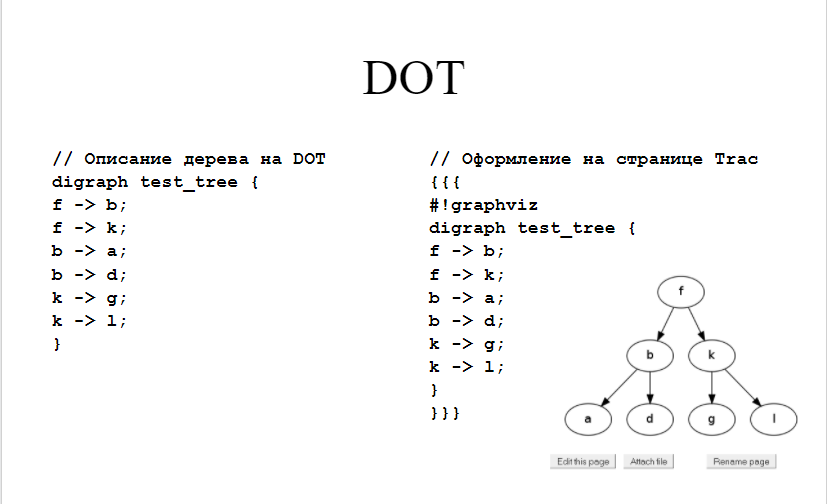


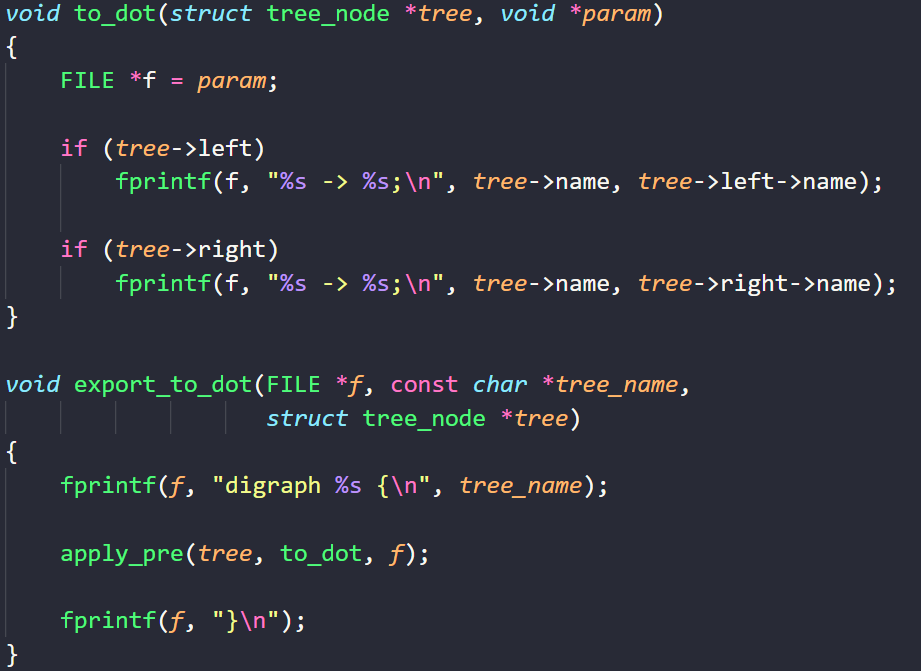
## 23. ДДП. Освобождение памяти из-под всего дерева.

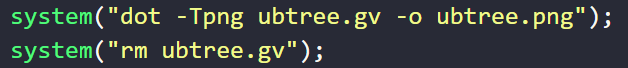
Необходимо выполнить обход дерева с пост-порядком с последовательным удалением узлов дерева.

## 24. ДДП. Язык DOT, примеры использования. Утилита GraphViz.

DOT - язык описания графов. Граф, описанный на языке DOT, обычно представляет собой текстовый файл с расширением .gv в понятном для человека и обрабатывающей программы формате. В графическом виде графы, описанные на языке DOT, представляются с помощью специальных программ, например GraphViz.







# Область видимости, время жизни, связывание

## 1. Что такое область видимости имени?

Область видимости имени – часть текста программы, в рамках которой имя может быть использовано.

## 2. Какие области видимости есть в языке Си? Приведите примеры.

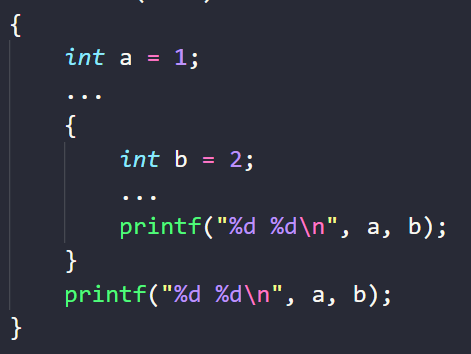
В языке Си выделяют следующие области видимости:

* блок – последовательность объявлений, определений и операторов, заключенных в фигурные скобки (два вида: составной оператор и определение функции)
* файл (область видимости в пределах файла имеют имена, описанные за пределами какой-либо функции, переменная с областью видимости файл видна от точки ее описания до конца файла, содержащего объявление, имена функций всегда имеют файловую область видимости)
* функция (данную область видимости имеют только метки (используются в операторе goto), метки видны из любого места функции, в которой они написаны, в пределах функции имена меток должны быть уникальными)
* прототип функции (область видимости в пределах прототипа функции применяется к именам переменных, которые используются в прототипах функций, от точки в которой объявлена переменная, до конца объявления прототипа).

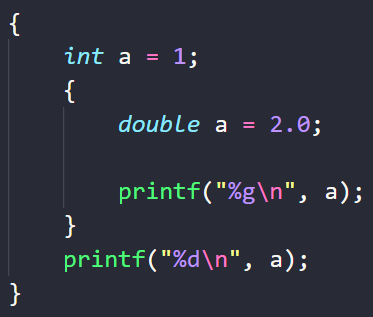
## 3. Какие правила перекрытия областей видимости есть в языке Си? Приведите примеры.

В языке Си существуют два правила перекрывания областей видимости:

* Переменные, определенные внутри некоторого блока, будут доступны из всех блоков, вложенных в данный.



* Возможно определить в одном из вложенных блоков переменную с именем, совпадающим с именем одной из "внешних" переменных.



Переменные, определенные внутри некоторого блока, будут доступны из всех блоков, вложенных в данный.

## 4. Что такое блок?

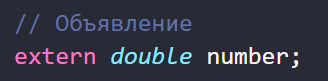
Блок – последовательность объявлений, определений и операторов, заключенных в фигурные скобки (два вида: составной оператор и определение функции). Блоки могут включать в себя составные операторы, но не определения функций.

## 5. Какие виды блоков есть в языке Си?

Два вида: составной оператор и определение функции.

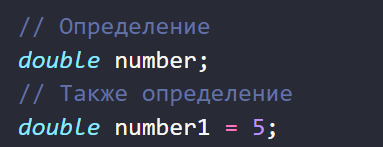
## 6. Что такое объявление? Приведите примеры.

Происходит описание типа переменной и связывание данной переменной с именем. Память не выделяется.



## 7. Что такое определение? Приведите примеры.

Происходит описание типа переменной и связывание данной переменной с именем. Память выделяется.



## 8. Для чего нужны объявления?

Объявление устанавливает свойства объекта: его тип (например, целый), размер (например, 4 байта) и т.д. Объявление позволяет компилятору корректно интерпретировать связанное с переменной имя.

## 9. Чем отличаются определения и объявления?

В объявлениях переменных происходит только связывание переменной с типом, а в определениях происходит не только связывание с именем, но и выделение памяти под конкретную переменную. Объявлений может быть сколько угодно, а определение лишь одно, объявления и определения должны быть одинакового типа.

## 10. Что такое время жизни программного объекта?

*Время жизни (storage duration)* – это интервал времени выполнения программы, в течение которого «программный объект» существует.

В языке Си время жизни «программного объекта» делится на три категории

* глобальное (по стандарту - статическое (англ. static)).
* локальное (по стандарту - автоматическое (англ. automatic)).

Локальным временем жизни обладают «программные объекты», область видимости которых ограничена блоком. Такие объекты создаются при каждом входе в блок, где они определяются. Они уничтожаются при выходе из этого «родительского» блока. Примерами таких переменных являются локальные переменные и параметры функции.

* динамическое (по стандарту - выделенное (англ. allocated)).

Время жизни «выделенных» объектов длится с момента выделения памяти и заканчивается в момент ее освобождения. В Си нет переменных, обладающих динамическим временем жизни. Динамическое выделение выполняется программистом «вручную» с помощью соответствующих функций. Единственные способ «добраться» до выделенной динамической памяти – использование указателей.

## 11. Какие виды времени жизни есть у переменных?

Время жизни переменной может быть*локальным*или*глобальным.*

## 12. Какие виды времени жизни есть у функций?

Все функции в Си имеют глобальное время жизни и существуют в течение всего времени выполнения программы.

## 13. Как время жизни влияет на область памяти, в которой располагается программный объект?

Программные объекты, обладающие глобальным временем жизни, хранятся в секции данных.  
Программные объекты, обладающие локальным временем жизни, хранятся в стеке (за исключением регистровых переменных).

Программные объекты, обладающие локальным временем жизни, хранятся в куче.

## 14. Что такое связывание?

*Связывание (linkage)* определяет область программы (функция, файл, вся программа целиком), в которой «программный объект» может быть доступен другим функциям программы.

## 15. Какие виды связывания есть в языке Си?

Стандарт языка Си определяет три формы связывания:

* внешнее (external).
* внутреннее (internal).
* никакое (none).

## 16. Как связывание влияет на "свойства" объектного/исполняемого файла? Что это за "свойства"?

Имена с внешним связыванием доступны во всей программе. Подобные имена «экспортируются» из объектного файла, создаваемого компилятором.

Имена с внутренним связыванием доступны только в пределах файла, в котором они определены, но могут «разделяться» между всеми функциями этого файла.

Имена без связывания принадлежат одной функции и не могут разделяться вообще.

Данная информация хранится в таблице символов объектного файла.

## 17. Какими характеристиками (область видимости, время жизни, связывание) обладает переменная в зависимости от места своего определения?

Если переменная определена вне какой-либо функции, то по умолчанию она обладает файловой областью видимости, глобальным временем жизни, внешнем связыванием, если же явно описать класс памяти static, то связывание изменится на внутреннее.

Если переменная определена в каком-либо блоке (класс памяти auto), то она обладает областью видимости в пределах блока, локальным временем жизни, отсутствием связывания, если же явно описать класс памяти static, то время жизни изменится на глобальное.

## 18. Какими характеристиками (область видимости, время жизни, связывание) обладает функция в зависимости от места своего определения?

Все функции в Си по умолчанию имеют файловую область видимости, глобальное время жизни и внешнее связывание (класс памяти extern), если же явно описать класс памяти static, то связывание изменится на внутреннее.

## 19. Какие классы памяти есть в языке Си?

В языке Си существует четыре класса памяти

* auto.
* static.
* extern.
* register.

## 20. Для чего нужны классы памяти?

Управлять временем жизни, областью видимости и связыванием переменной (до определенной степени) можно с помощью так называемых классов памяти.

## 21. Какие классы памяти можно использовать с переменными? С функциями?

Для переменных применимы классы памяти auto, static, extern, register.

Для функций применимы классы памяти static, extern.

Аргументы функций по умолчанию имеют класс памяти *auto*. Единственный другой класс памяти, который может использоваться с параметрами функций, - *register*.

## 22. Сколько классов памяти может быть у переменной? У функции?

Одновременно переменная и функция могут иметь 1 класс памяти.

## 23. Какие классы памяти по умолчанию есть у переменной? У функции?

По умолчанию переменные, описанные вне блоков, не имеют никакого класса памяти, если же они внутри блока, то класс памяти auto. Функции по умолчанию принадлежат классу памяти extern.

## 24. Расскажите о классе памяти auto.

Применим только к переменным, определенным в блоке. Переменная, принадлежащая к классу auto, имеет *локальное время жизни, видимость в пределах блока, не имеет связывания*. По умолчанию любая переменная, объявленная в блоке или в заголовке функции, относится к классу автоматической памяти.

## 25. Расскажите о классе памяти static.

Класс памяти static может использоваться с любыми переменными независимо от места их расположения.

* Для переменной вне какого-либо блока, static изменяет связывание этой переменной на внутреннее.
* Для переменной в блоке, static изменяет время жизни с автоматического на глобальное.

Статическая переменная, определенная вне какого-либо блока, имеет глобальное время жизни, область видимости в пределах файла и внутреннее связывание. Этот класс памяти скрывает переменную в файле, в котором она определена.

Статическая переменная, определенная в блоке, имеет глобальное время жизни, область видимости в пределах блока и отсутствие связывания.

* Такая переменная сохраняет свое значение после выхода из блока.
* Инициализируется только один раз.
* Если функция вызывается рекурсивно, это порождает новый набор локальных переменных, в то время как статическая переменная разделяется между всеми вызовами.

Использование класса памяти static для функций полезно, потому что:

* Функции, определенные со static, невидны в других файлах и могут безболезненно изменяться (инкапсуляция).
* Так как функция имеет внутренне связывание, ее имя может использоваться в других файлах.

## 26. Расскажите о классе памяти extern.

Помогает разделить переменную между несколькими файлами.

* Объявлений (extern int number;) может быть сколько угодно.
* Определение (int number;) должно быть только одно.
* Объявления и определение должны быть одинакового типа.

*Замечание.*

**extern int number = 5; // определение**

Используется для переменных, определенных как в блоке (*Глобальное время жизни, видимость в блоке, связывание непонятное)*, так и вне блока (*Глобальное время жизни, файловая область видимости, связывание непонятное)*.

Связывание определяется по определению переменной.

## 27. Расскажите о классе памяти register.

Использование класса памяти register – просьба (!) к компилятору разместить переменную не в памяти, а в регистре процессора.

* Используется только для переменных, определенных в блоке.
* Задает *локальное время жизни, видимость в блоке и отсутствие связывания*.
* Обычно не используется.

К переменным с классом памяти register нельзя применять операцию получения адреса &. Современные компиляторы являются довольно продвинутыми и используют множественные стратегии оптимизации, поэтому самостоятельное использование класса памяти register может даже замедлить работу программы.

## 28. Для чего используется ключевое слово extern?

Ключевое слово extern используется для объявления переменных.

## 29. Особенности совместного использования ключевых слов static и extern.

6.2.2 #4 For an identifier declared with the storage-class specifier extern in a scope in which a prior declaration of that identifier is visible, if the prior declaration specifies internal or external linkage, the linkage of the identifier at the later declaration is the same as the linkage specified at the prior declaration. If no prior declaration is visible, or if the prior declaration specifies no linkage, then the identifier has external linkage.

6.2.2 #7 If, within a translation unit, the same identifier appears with both internal and external linkage, the behavior is undefined.

|  |  |
| --- | --- |
| **static int i;**  **extern int i;**  // ОК | **extern int i;**  **static int i;**  // ОШИБКА |

## 30. Как описать автоматическую глобальную переменную?

???

## 31. Какая переменная называется глобальной?

Если «программный объект» имеет глобальное время жизни, он существует на протяжении выполнения всей программы.

## 32. Какая переменная называется локальной?

Локальным временем жизни обладают «программные объекты», область видимости которых ограничена блоком.

## 33. Каким значением по умолчанию инициализируются автоматические переменные?

Локальные переменные не имеют значения по умолчанию, в них хранится какой-то мусор.

## 34. Каким значением по умолчанию инициализируются переменные с глобальным временем жизни?

Переменные с глобальным временем жизни по умолчанию имеют 0 значение.

## 35. Какие недостатки есть у использования глобальных переменных?

Использование глобальных переменных снижает читабельность, делает код сложным для понимания. Очень сложно определить, где, как и с какой целью была проинициализирована глобальная переменная и как правильно ее использовать. Код с глобальными переменными сложно сопровождать. При изменении глобальной переменной вы вынуждены просматривать весь код, чтобы внести изменения везде, где используется эта переменная.

Злоупотребление глобальными переменными может привести к ошибкам, которые очень сложно отлаживать. Без какого-либо механизма контроля за использованием переменной очень просто записать в переменную невалидные данные, которые могут привести к ошибкам в других частях кода (например, если в одной части кода переменная заполняется массивом, а в другой части кода в этой переменной ожидается объект).

Можно забыть объявить переменную глобальной и работать с локальной переменной, не замечая этого, до тех пор, пока приложение не сломается. Такие ошибки сложно отлаживать.

Если вы совмещаете свой код с чужим (например, при использовании сторонних библиотек или при написании расширений для другого ПО) и обе системы используют глобальные переменные, существует вероятность того, что названия переменных могут совпасть. Это становится причиной возникновения ошибок в обеих системах, которые сложно отлавливать.

Все части кода, использующие одну глобальную переменную, сильно связаны между собой. Разделить сильно связанный код очень сложно. Это затрудняет его повторное использование.

Написание юнит-тестов становится более сложным, поскольку тесту не известно, какие глобальные переменные нужны и как проинициализировать все глобальные переменные валидными значениями.

## 36. Объектный файл, секции, таблица символов.

Объектный файл представляет собой блоки машинного кода и данных с неопределенными адресами ссылок на данные и подпрограммы в других объектных модулях, а также список своих подпрограмм и данных.

Объектный файл состоит из секций, которые содержат данные в широком смысле этого слова

* заголовки (метаинформация, необходимая для организации самого файла);
* код (.text);
* данные (.data, .rodata, .bss);
* таблицу символов (.symtab);
* …

|  |  |
| --- | --- |
| **Буква** | **Расположение** |
| B, b | Секция неинициализированных данных (.bss). |
| D, d | Секция инициализированных данных (.data). |
| R, r | Секция данных только для чтения (.rodata) |
| T, t | Секция кода (.text) |
| U | Символ не определен, но ожидается, что он появится. |

## 37. Что делает компоновщик?

Компоновщик на вход получает один или несколько объектных файлов и собирает из них исполняемый файл или библиотечный файл-модуль. При компоновке выполняется перемещение (объединение) всех секций в одну (для каждого типа секций), затем происходит настройка ссылок для неизвестных символов (секция U). Также компоновщик добавляет помимо пользовательских объектных файлов некоторые системные файлы и переменные, необходимые для корректной работы программы.

Два шага:

Перемещение (relocation)

* «Склеивание» нескольких секций в одну.

Разрешение ссылок (reference resolving)

Ссылки внутри каждого файла уже разрешены.

* Поиск кода, который вызывает что-то за пределами своей исходной секции.
* Поиск места, где теперь располагается вызываемые код.
* Замена «поддельного» адреса на настоящий.

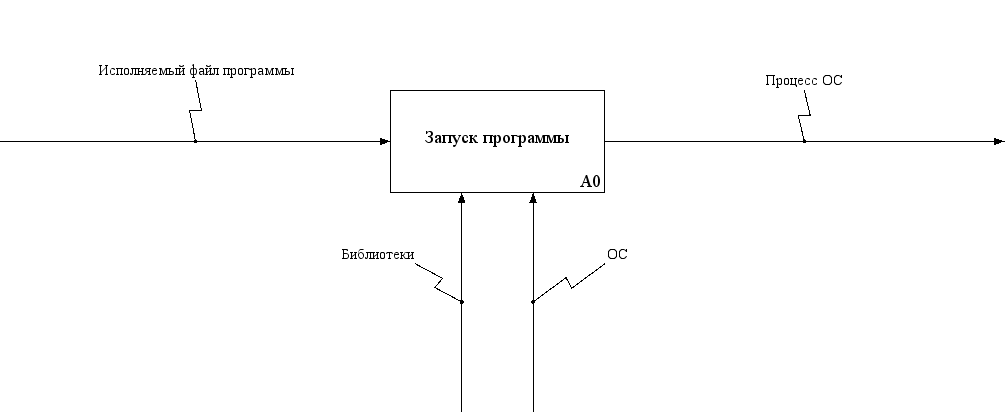
## 38. Журналирование, подходы к реализации.

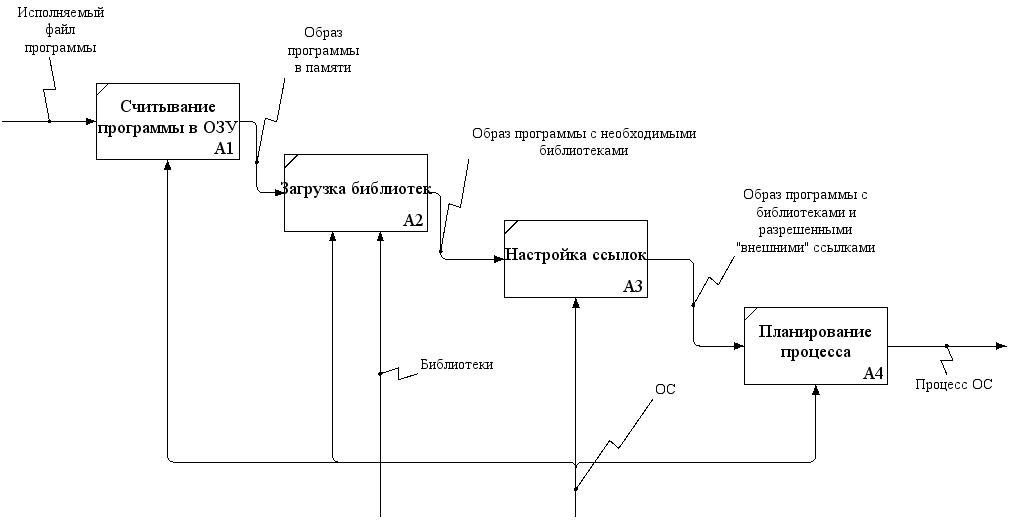
1. Можно завести глобальную переменную, в которой будет храниться имя файла, в который будет производиться печать информации. В модули будет необходимо подключить соответствующий заголовочный файл и по мере необходимости печатать информацию в журнал.

2. Первый подход является не очень безопасным, поэтому лучше реализовать функцию печать в журнал без доступа к глобальной переменной (сделать связывание внутренним), функция в свою очередь должна управляться строкой форматирования.

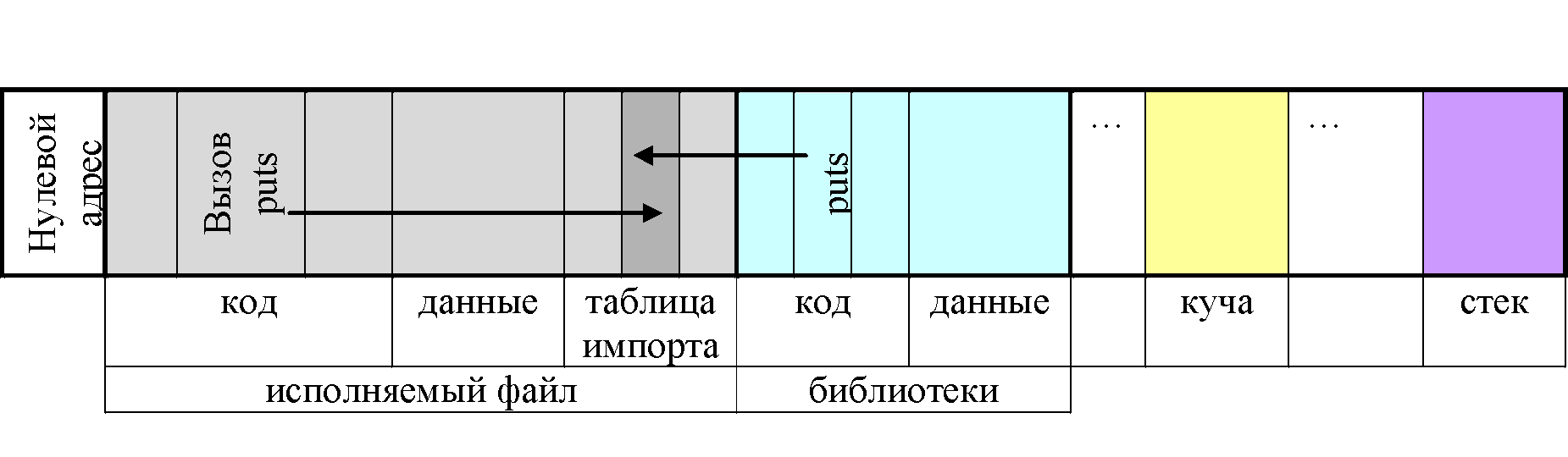
# Схема распределения памяти в программе на Си. Стек. Куча.

## 1. Процесс запуска программы («превращения в процесс»).





## 2. Абстрактное адресное пространство программы.



## 3. Опишите достоинства и недостатки локальных переменных.

Для хранения локальных переменных используется так называемая автоматическая память.

“+”

* Память под локальные переменные выделяет и освобождает компилятор.

“-”

* Время жизни локальной переменной "ограничено" блоком, в котором она определена.
* Размер размещаемых в автоматической памяти объектов должен быть известен на этапе компиляции.
* Размер автоматической памяти в большинстве случаев ограничен.

## 4. Локальные переменные создаются в так называемой «автоматической памяти». Почему эта память так называется?

Данная память имеет такое название, так как память под локальные переменные выделяет и освобождает компилятор автоматически.

## 5. Для чего в программе используется аппаратный стек?

* вызова функции (call name)
* поместить в стек адрес команды, следующей за командой call
* передать управление по адресу метки name
* возврата из функции (ret)
* извлечь из стека адрес возврата address
* передать управление на адрес address
* передачи параметров в функцию
* *соглашение о вызове*:
  + расположение входных данных;
  + порядок передачи параметров;
  + какая из сторон очищает стек;
  + etc
* *cdecl*
  + аргументы передаются через стек, справа налево;
  + очистку стека производит вызывающая сторона;
  + результат функции возвращается через регистр EAX, но …
* выделения и освобождения памяти под локальные переменные

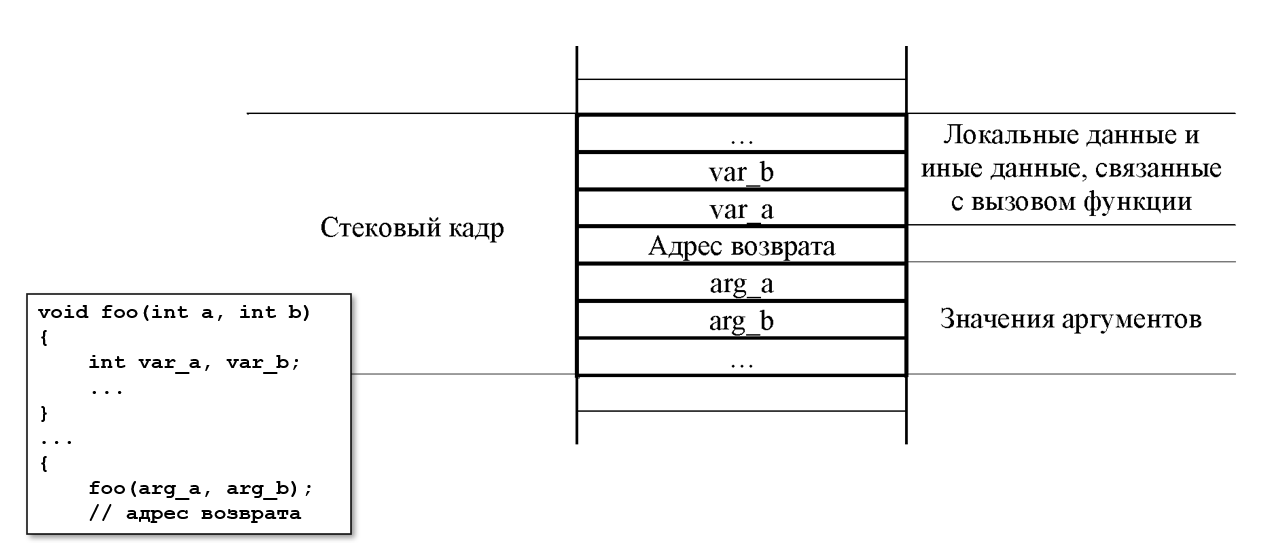
## 6. Что такое кадр стека?

*Стековый кадр (фрейм)* - механизм передачи аргументов и выделения временной памяти с использованием аппаратного стека.

## 7. Для чего в программе используется кадр стека? Приведите примеры.

В стековом кадре размещаются:

* значения фактических аргументов функции;
* адрес возврата;
* локальные переменные;
* иные данные, связанные с вызовом функции.



## 8. Какие преимущества и недостатки есть у использования кадра стека?

“+”

* Удобство и простота использования.

“-”

* Производительность (передача данных через память без необходимости замедляет выполнение программы)
* Безопасность (стековый кадр перемежает данные приложения с критическими данными - указателями, значениями регистров и адресами возврата)

## 9. Что такое соглашение о вызове?

Некий договор, заключенный между вызывающей и вызываемой стороной о том, каким образом передаются данные, в каком порядке, кто ответственен за удаление данных их стека, а также каким образом возвращается результат и др.

## 10. Какое соглашение о вызове используется в языке Си? В чем оно заключается?

* *cdecl*
  + аргументы передаются через стек, справа налево;
  + очистку стека производит вызывающая сторона;
  + результат функции возвращается через регистр EAX, но …

## 11. Что такое переполнение буфера? Чем оно опасно?

Переполнение буфера происходит, например, при попытке записи большего, чем длина строки, количества символов в строку. В таком случае поведение программы не определено, может получиться корректный ответ, может произойти зацикливание, может ошибка и т.д. Компилятор подобного рода ошибки обнаружить не может.

## 12. Почему нельзя из функции возвращать указатель на локальную переменную, определенную в этой функции?

Локальные переменные разрушаются после выхода из функции, соответственно, тогда получаем висящий указатель.

## 13. Для чего в программе используется куча?

Куча используется в программе для работы с динамическими ресурсами.

## 14. Происхождение термина «куча».

Согласно Дональду Кнуту несколько программистов в 1975 году начали использовать термин куча для обозначения пула свободной памяти.

## 15. Свойства области памяти, которая выделяется динамически.

* Выделяется по крайней мере указанной количество байт (меньше нельзя, больше можно).
* Указатель, возвращенный функцией выделения, указывает на выделенную область (т.е. область, в которую программа может писать и из которой может читать данные).
* Ни один другой вызов функции выделения памяти не может выделить эту область или ее часть, если только она не была освобождена с помощью free.

## 16. Как организована куча?

Куча представляет собой непрерывную область памяти. Память кучи можно разделить на занятую и свободную. Для хранения данных о том, какая область памяти является занятой, а какая свободной, обычно используется дополнительная область памяти.

## 17. Алгоритм работы функции malloc.

Функция, подобная malloc (), выполняет примерно следующие действия:

* просматривает список занятых/свободных областей памяти, размещённых в куче, в поисках свободной области подходящего размера;
* в случае нехватки свободной памяти может запросить дополнительную память у ОС;
* добавляет найденную область в список занятых областей (или помечает область как занятую);
* возвращает указатель на начало области памяти;
* если по тем или иным причинам выделить память не удалось, сообщает об ошибке (например, malloc () возвращает NULL).

## 18. Алгоритм работы функции free.

Функция, подобная free (), выполняет примерно следующие действия:

* просматривает список занятых/свободных областей памяти, размещённых в куче, в поисках указанной области;
* удаляет из списка занятых областей памяти найденную область;
* добавляет найденную область в список свободных областей (или помечает область как свободную).

## 19. Какие гарантии относительно выделенного блока памяти даются программисту?

* Выделяется по крайней мере указанной количество байт (меньше нельзя, больше можно).
* Указатель, возвращенный функцией выделения, указывает на выделенную область (т.е. область, в которую программа может писать и из которой может читать данные).
* Ни один другой вызов функции выделения памяти не может выделить эту область или ее часть, если только она не была освобождена с помощью free.

## 20. Что значит "освободить блок памяти" с точки зрения функции free?

С точки зрения функции free необходимо добавить данный блок памяти в список свободный областей памяти.

## 21. Преимущества и недостатки использования динамической памяти.

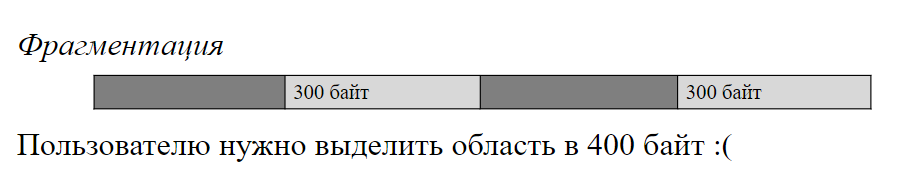
“+”

* Все «минусы» локальных переменных.
* Можно выделить гораздо больший объем памяти, нежели чем на стеке.

“-”

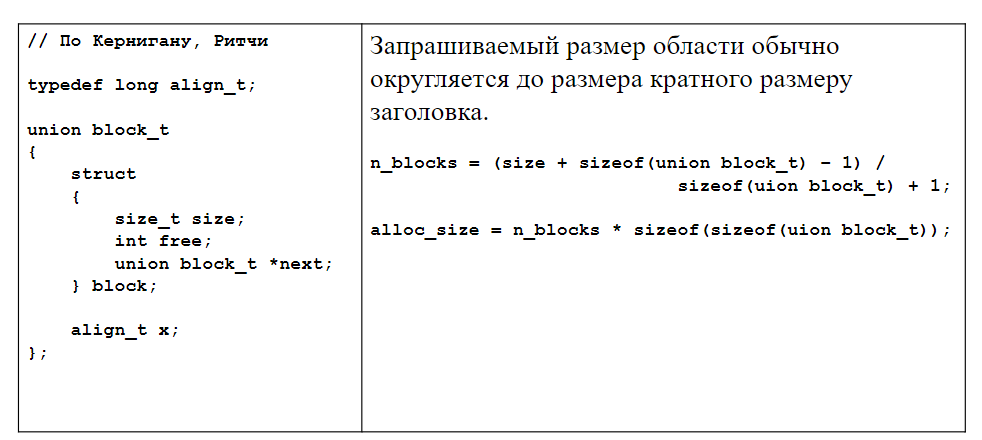
* Ручное управление временем жизни.
* Работа с динамической памятью гораздо медленнее, чем на стеке.

## 22. Что такое фрагментация памяти?



## 23. Выравнивание блока памяти, выделенного динамически.

Для хранения произвольных объектов блок должен быть правильно выровнен. В каждой системе есть самый «требовательный» тип данных – если элемент этого типа можно поместить по некоторому адресу, то любые другие элементы тоже можно поместить туда.



## 24. Что такое variable length array?

VLA – массив переменной длины.

## 25. Чем отличается статический массив от variable length array?

Размер статического массива известен на этапе компиляции, в то время как данные о массиве переменной длины программа получает только на этапе выполнения.

## 26. Какую операцию языка Си пришлось реализовывать по-другому (не как для встроенных типов) специально для variable length array?

До C99 операция sizeof выполнялась на этапе компиляции, однако затем ее пришлось модифицировать, т.к. для VLA его размер на этапе компиляции неизвестен, соответственно sizeof работает с C99 для VLA непосредственно во время выполнения программы.

## 27. Особенности использования variable length array.

* Длина такого массива вычисляется во время выполнения программы, а не во время компиляции.
* Память под элементы массива выделяется на стеке.
* Массивы переменного размера нельзя инициализировать при определении.
* Массивы переменной длины могут быть многомерными.
* Адресная арифметика справедлива для массивов переменной длины.
* Массивы переменной длины облегчают описание заголовков функций, которые обрабатывают массивы.

## 28. Справедлива ли для variable length array адресная арифметика?

Да, справедлива.

## 29. Как вы думаете почему variable length array нельзя инициализировать?

Т.к. память выделяется на стеке непосредственно во время выполнения, заранее нельзя проинициализировать область памяти, которая не выделена.

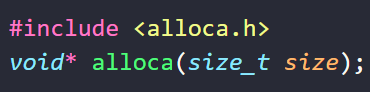
## 30. Для чего используется variable length array? Приведите примеры.

Массивы переменной длины можно использовать там же где и статические массивы.

## 31. В какой области и «кем» выделяется память под массив переменной длины?

Память выделяется на стеке операционной системой.

## 32. Функция alloca.



Функция *alloca* выделяет область памяти, размером size байт, на стеке. Функция возвращает указатель на начало выделенной области. Эта область автоматически освобождается, когда функция, которая вызвала *alloca*, возвращает управления вызывающей стороне.

Если выделение вызывает переполнение стека, поведение программы не определено.

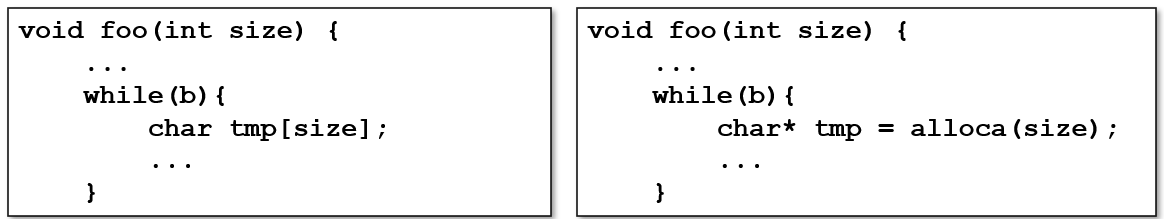
“+”

* Выделение происходит быстро.
* Выделенная область освобождается автоматически.

“–”

* Функция *нестандартная*.
* Серьезные ограничения по размеру области.

## 33. alloca VS VLA.



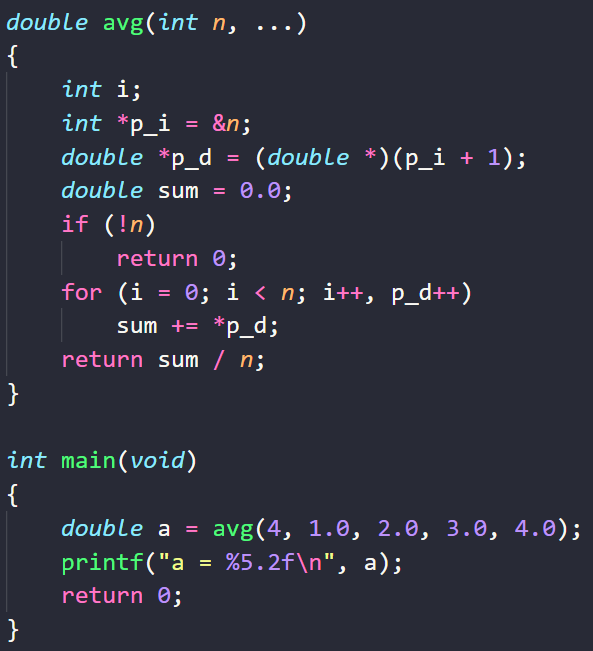
При выходе из цикла VLA будет разрушен и память очищена, в то время как переменная tmp с alloca тоже будет разрушена, однако память очищена не будет (она будет очищена только при выходе из функции), таким образом, будет происходит выделение памяти на каждой итерации и довольно легко получить переполнение стека.

# Функции с переменным числом параметров

## 1. Можно ли реализовать в языке Си функцию со следующим прототипом int f(...)? Почему?

Данную функцию реализовать можно, однако в ней нет смысла, т.к. данные передаются в функцию через стек, поэтому не получиться добраться до переменных аргументов, необходимо, чтобы хоть 1 параметр был описан явно, а переменные параметры располагались в конце.

## 2. Покажите идею реализации функций с переменным числом параметров.



Аналогично, можно явно указать первый аргумент и выполнять чтение аргументов из стека, пока не встретиться нейтральное значение.

## 3. Почему для реализации функций с переменным числом параметров нужно использовать возможности стандартной библиотеки?

Работа с кадром стека сильно зависит от реализации компилятора его разработчиками, соответственно адресная арифметика не применима, т.к. могут происходить неявные преобразования типов аргументов.

## 4. Опишите подход к реализации функций с переменным числом параметров с использованием стандартной библиотеки. Какой заголовочный файл стандартной библиотеки нужно использовать? Какие типы и макросы из этого файла вам понадобятся? Для чего?

stdarg.h

* va\_list (некий тип для описания списка аргументов)
* void va\_start(va\_list argptr, last\_param) – инициализация списка аргументов.
* type va\_arg(va\_list argptr, type) – извлечение переменной заданного типа из стека.
* void va\_end(va\_list argptr) – деинициализация списка аргументов.

## 5. Какая особенность языка Си упрощает реализацию функций (с точки зрения компилятора) с переменным числом параметров?

Т.к. переменные помещаются в стек при вызове функции, то зная указатель на начало списка параметров в стеке, манипулируя указателем, можно получить значения всех аргументов, помещенных в стек.

## 6. Почему при вызове va\_arg(argp, short int) (или va\_arg(argp, float)) выдается предупреждение?

Указанные выше типы расширяются до типа int и double соответственно, поэтому возможна некорректная работа с указателями.

## 7. Какая "опасность" существует при использовании функций с переменным числом параметров?

Компилятор не может проверить соответствие типа передаваемого аргумента с ожидаемым значением, а также при реализации функции с переменным числом параметров может быть использован нестандартный метод передачи аргументов (либо передавать количество аргументов, либо нейтральное значение), таким образом разработчик может допустить логическую ошибку.

## 8. Как написать функцию, которая получает строку форматирования и переменное число параметров (как функция printf), и передает эти данные функции printf? (Подсказка: см. последний вариант реализации журналирования.)

Для реализации такой функции существуют библиотечные функции vprintf, vfprintf, vsprintf. Данные функции работают аналогично printf, fprintf, sprintf, только разработчик должен вместо переменного количества аргументов передавать единственный указатель типа va\_list.

# Препроцессор. inline-функции.

## 1. Что делает препроцессор? В какой момент в процессе получения исполняемого файла вызывается препроцессор?

Предпроцессор выполняет 4 основные функции:

* удаление комментариев
* включение файлов (директива #include)
* текстовые замены (директива #define)
* условная компиляция (IncludeGuard)

Предпроцессор вызывается самым первым на этапе создания исполняемого файла.

## 2. На какие группы можно разделить директивы препроцессора?

Макроопределения

* #define, #undef

 Директива включения файлов

* #include

 Директивы условной компиляции

* #if, #ifdef, #endif и др.

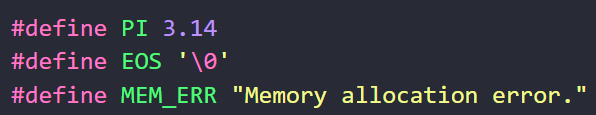
Остальные директивы (#pragma, #error, #line и др.) используются реже.

## 3. Какие правила справедливы для всех директив препроцессора?

* Директивы всегда начинаются с символа "#".
* Любое количество пробельных символов может разделять лексемы в директиве.
* Директива заканчивается на символе '\n'.
* Директивы могут появляться в любом месте программы.

## 4. Что такое простой макрос? Как такой макрос обрабатывается препроцессором? Приведите примеры.

#define идентификатор список-замены

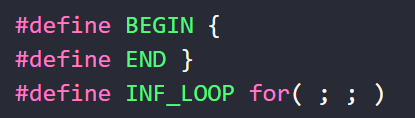


Предпроцессор при просмотре программы в случае нахождения идентификатора выполняет замену идентификатора на значение из списка замены.

## 5. Для чего используются простые макросы?

Используются для улучшения читабельности программного кода:

* в качестве имен для числовых, символьных и строковых констант.
* незначительного изменения синтаксиса языка.



* Переименования типов.

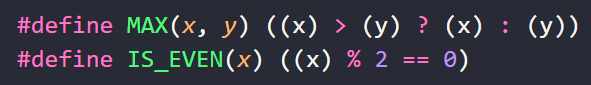


* Управления условной компиляцией.

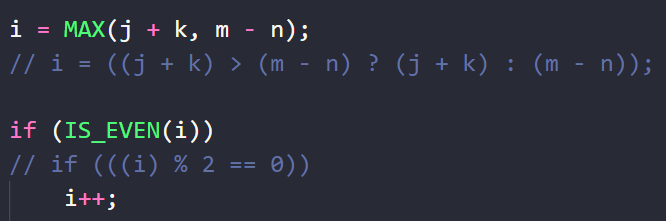
## 6. Что такое макрос с параметрами? Как такой макрос обрабатывается препроцессором? Приведите примеры.

#define идентификатор(x1, x2, ..., xn) список-замены

* Не должно быть пробела между именем макроса и (.
* Список параметров может быть пустым.



Где-то в программе



## 7. Макросы с параметрами VS функции: преимущества и недостатки.

Преимущества

* программа может работать немного быстрее;
* макросы "универсальны".

Недостатки

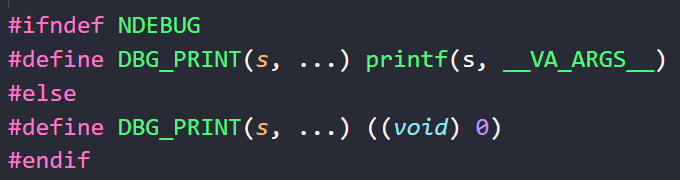
* скомпилированный код становится больше;

**n = MAX(i, MAX(j, k)));**

* типы аргументов не проверяются;
* нельзя объявить указатель на макрос;
* макрос может вычислять аргументы несколько раз.

**n = MAX(i++, j)**

## 8. Макросы с переменным числом параметров. Приведите примеры.

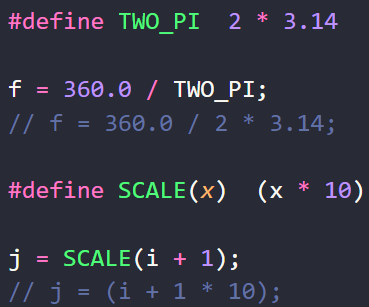


## 9. Какими общими особенностями/свойствами обладают все макросы?

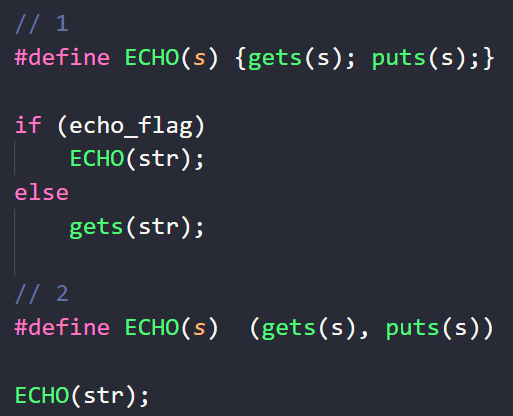
* Список-замены макроса может содержать другие макросы.
* Препроцессор заменяет только целые лексемы, не их части.
* Определение макроса остается «известным» до конца файла, в котором этот макрос объявляется.
* Макрос не может быть объявлен дважды, если эти объявление не тождественны.
* Макрос может быть «разопределен» с помощью директивы #undef.

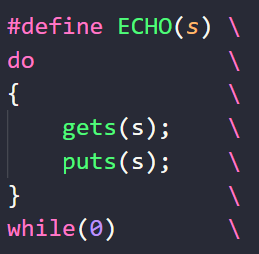
## 10. Объясните правила использования скобок внутри макросов. Приведите примеры.

* Если список-замены содержит операции, он должен быть заключен в скобки.
* Если у макроса есть параметры, они должны быть заключены в скобки в списке-замены.



## 11. Какие подходы к написанию "длинных" макросов вы знаете? Опишите их преимущества и недостатки. Приведите примеры.





## 12. Какие предопределенные макросы вы знаете? Для чего эти макросы могут использоваться?

* **\_\_LINE\_\_ -** номер текущей строки (десятичная константа)
* **\_\_FILE\_\_ -** имя компилируемого файла
* **\_\_DATE\_\_ -** дата компиляции
* **\_\_TIME\_\_ -** время компиляции

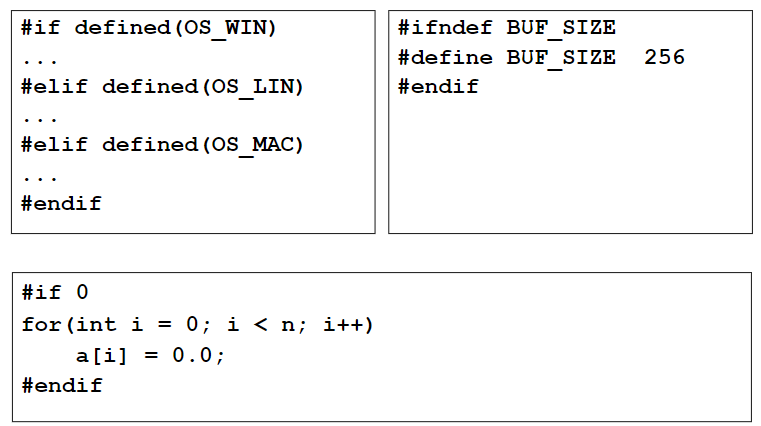
Эти идентификаторы нельзя переопределять или отменять директивой undef.

* **\_\_func\_\_ -** имя функции как строка (GCC only, С99 и не макрос)

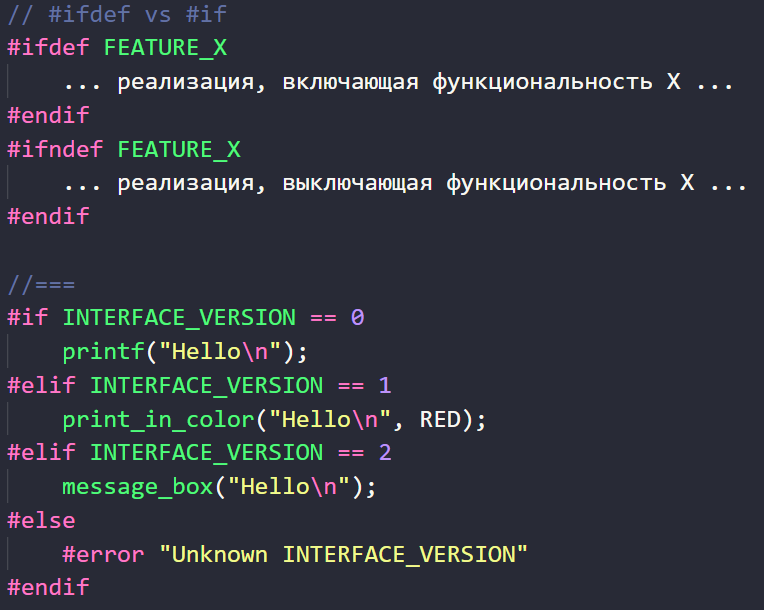
## 13. Для чего используется условная компиляция? Приведите примеры.

Использование условной компиляции:

* программа, которая должна работать под несколькими операционными системами.
* программа, которая должна собираться различными компиляторами.
* начальное значение макросов.
* временное выключение кода.

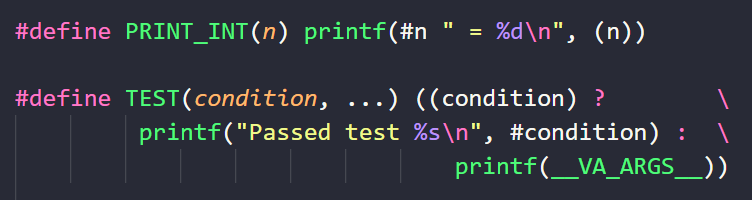


## 14. Директива #if VS директива #ifdef.

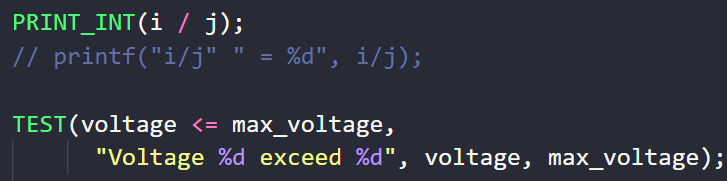


## 15. Операция #. Примеры использования.

«Операция» # конвертирует аргумент макроса в строковый литерал.

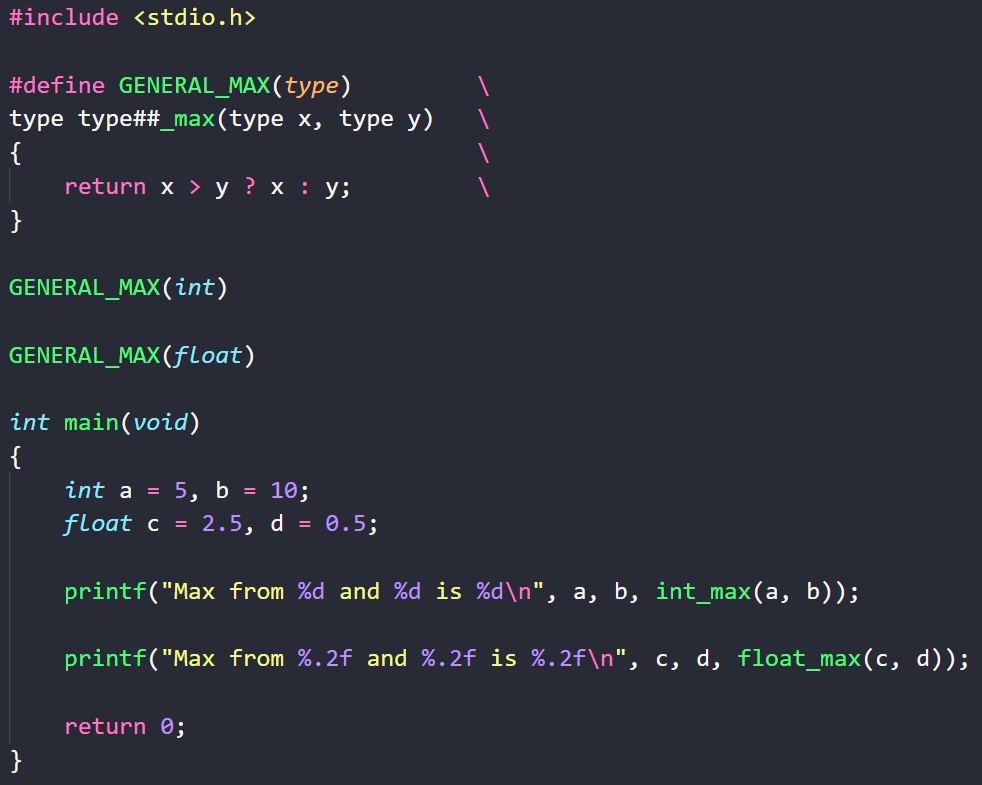


В программе



## 16. Операция ##. Примеры использования.

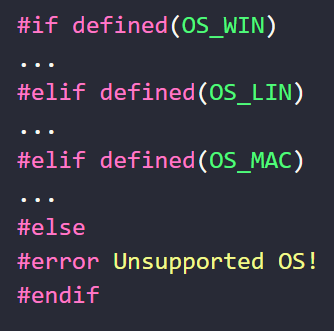
«Операция» ## объединяет две лексемы в одну.



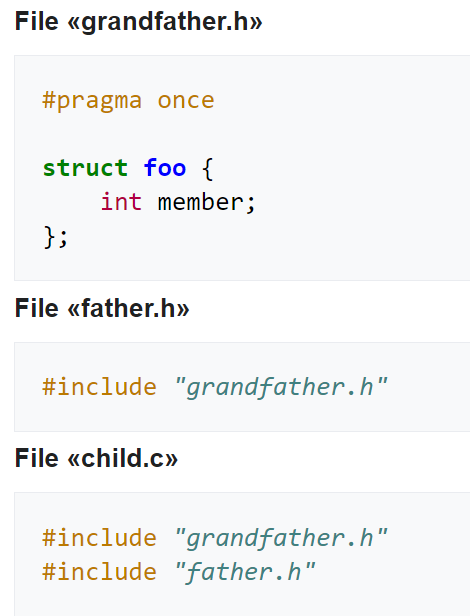
## 17. Особенности использования операций.

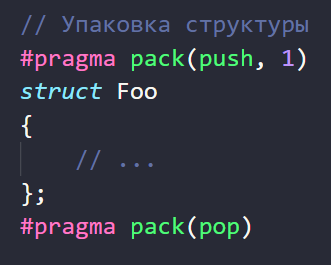
???

## 18. Директива #error. Примеры использования.



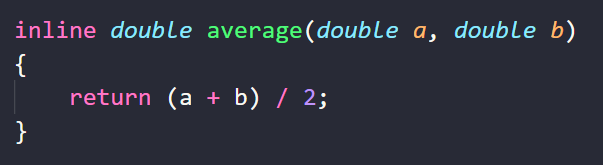
## 19. Директива #pragma (на примере once и pack). Примеры использования.





## 20. Ключевое слово inline.

inline – пожелание компилятору заменить вызовы функции последовательной вставкой кода самой функции.



inline-функции по-другому называют встраиваемыми или подставляемыми.

## 21. Назовите основную причину, по которой ключевое слово inline было добавлено в язык Си.

* Большие функции удобно порезать частями на маленькие inline функции, что позволит сделать код более читабельным и простым
* При использовании inline функций экономится время, затрачиваемое для вызова функции с соответствующими расходами для работы со стеком.

## 22. Подходы к реализации ключевого слова inline компилятором. Проанализируйте их недостатки.

* Использовать ключевое слово static со словом inline.
* Не использовать слово inline или static (ответственность на компиляторе).
* Добавить еще одно такое же не-inline определение функции где-нибудь в программе.

## 23. В чем разница между использованием <> и «» в директиве include?

Различие в том, где препроцессор будет начинать поиск файла some.h. Если использовать директиву #include "some.h", то сначала будут просмотрены локальные (по отношению к проекту) папки включения файлов. Если использовать #include <some.h>, то сначала будут просматриваться глобальные (по отношению к проекту) папки включения файлов. Глобальные папки включения — это папки, прописанные в настройке среды разработки, локальные — это те, которые прописаны в настройках проекта.

## 24. Можно ли операцию sizeof использовать в директивах препроцессора? Почему?

* Оператор sizeof () не работает в условных директивах препроцессора C, таких как #if и #elif, причина в том, что препроцессор C ничего не знает о размерах типов.
* Можно использовать sizeof () в теле макроса #define, потому что компилятор обрабатывает анализ текста замены, а препроцессор-нет.

# Библиотеки

## 1. Что такое библиотека?

Библиотека включает в себя

* + заголовочный файл;
  + откомпилированный файл самой библиотеки:
    - библиотеки меняются редко – нет причин перекомпилировать каждый раз;
    - двоичный код предотвращает доступ к исходному коду.

Библиотеки делятся на

* + статические;
  + динамические.

## 2. Какие функции обычно выносят в библиотеку?

В библиотеку наиболее выгодно выносить функции, которые используются в большом количестве проектов/модулей, это позволяет удобно подключать их в программный код.

## 3. В каком виде распространяются библиотеки? Что обычно входит в их состав?

Библиотека включает в себя

* + заголовочный файл;
  + откомпилированный файл самой библиотеки:
    - библиотеки меняются редко – нет причин перекомпилировать каждый раз;
    - двоичный код предотвращает доступ к исходному коду.

## 4. Какие виды библиотек вы знаете?

Библиотеки делятся на

* + статические;
  + динамические.

## 5. Преимущества и недостатки, которые есть у статических/динамических библиотек.

**Статические библиотеки.**

Связываются с программой в момент компоновки. Код библиотеки помещается в исполняемый файл.

«+»

* Исполняемый файл включает в себя все необходимое.
* Не возникает проблем с использованием не той версии библиотеки.

«-»

* «Размер».
* При обновлении библиотеки программу нужно пересобрать.

**Динамические библиотеки.**

Подпрограммы из библиотеки загружаются в приложение во время выполнения. Код библиотеки не помещается в исполняемый файл.

«+»

* + Несколько программ могут «разделять» одну библиотеку.
  + Меньший размер приложения (по сравнению с приложением со статической библиотекой).
  + Средство реализации плагинов.
  + Модернизация библиотеки не требует перекомпиляции программы.
  + Могут использовать программы на разных языках.

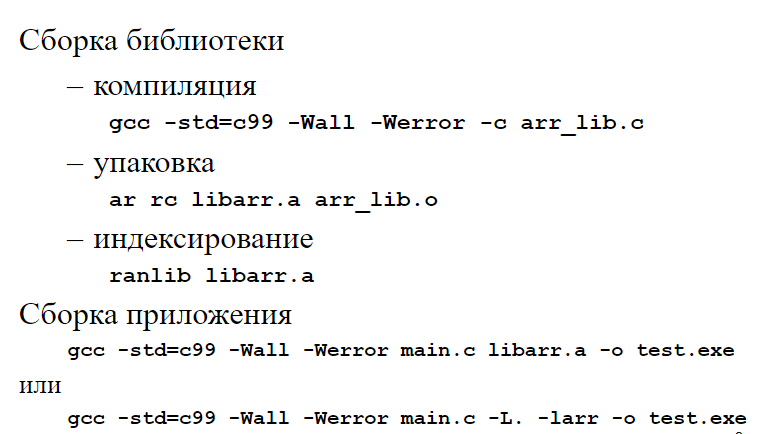
«-»

* + Требуется наличие библиотеки на компьютере.
  + Версионность библиотек.

Способы использования динамических библиотек

* + динамическая компоновка;
  + динамическая загрузка.

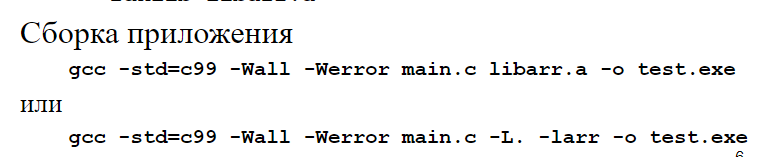
## 6. Как собрать статическую библиотеку?



## 7. Нужно ли "оформлять" каким-то специальным образом функции, которые входят в состав статической библиотеки?

Нет, не нужно.

## 8. Как собрать приложение, которое использует статическую библиотеку?

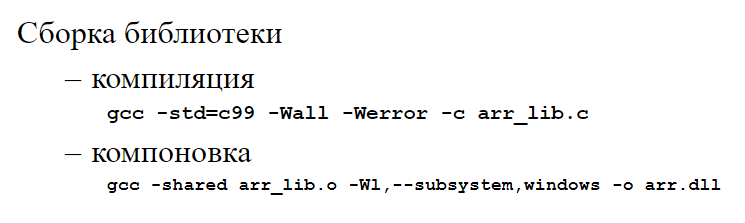


## 9. Нужно ли "оформлять" каким-то специальным образом исходный код приложения, которое использует статическую библиотеку?

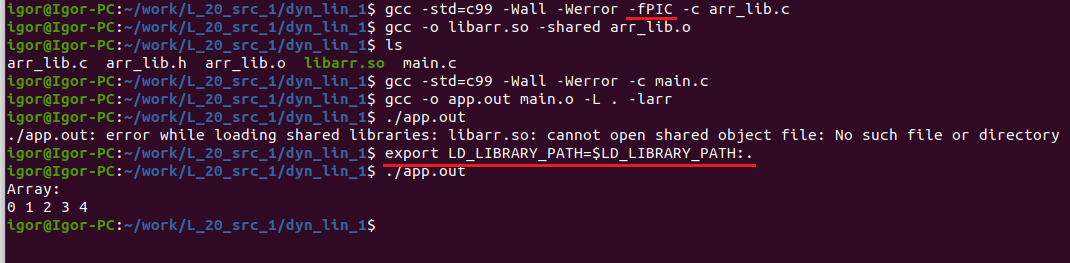
Нет, только подключить заголовочный файл библиотеки.

## 10. Как собрать динамическую библиотеку (Windows/Linux)?

**Windows**

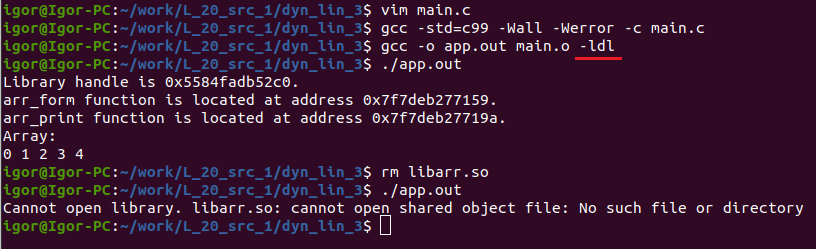


**Linux** (динамическая компоновка)



* Дополнительный флаг (-fPIC) при получении объектного файла.
* Использование переменной LD\_LIBRARY\_PATH.

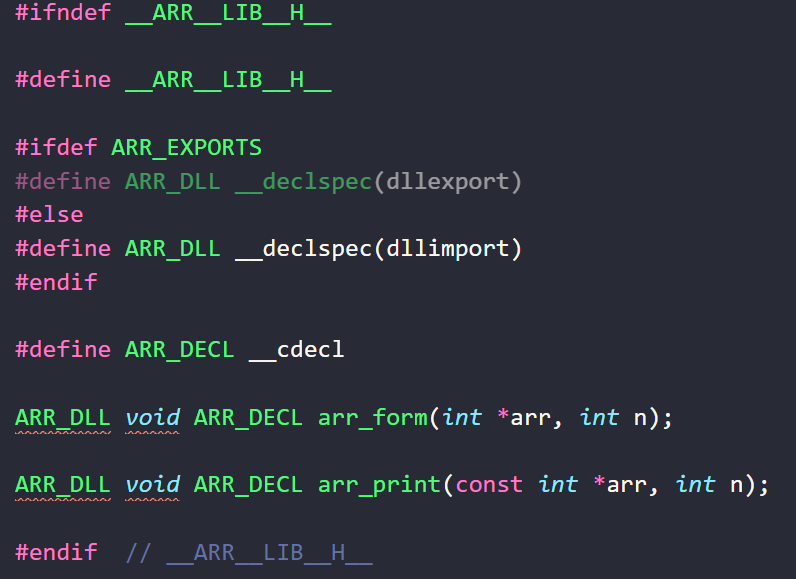
**Linux** (динамическая загрузка)



* Указание библиотеки dl при компоновке

## 11. Нужно ли "оформлять" каким-то специальным образом функции, которые входят в состав динамической библиотеки (Windows/Linux)?

При создании динамической библиотеки в Windows необходимо явно указывать какие функции будут доступны для экспорта + желательно указать используемое соглашение о вызове.



При создании динамической библиотеки в Linux ничего указывать не нужно, т.к. все функции по умолчанию экспортируются наружу.

## 12. Какие способы компоновки приложения с динамической библиотекой вы знаете? Назовите их преимущества и недостатки.

Существует два метода работы с динамическими библиотеками:

* динамическая компоновка.
* динамическая загрузка.

При использовании динамической компоновки большая часть ответственности передокладывается на компоновщика, а то каким-образом это произойдет проконтролировать нельзя, поэтому можно проделать эти операции самому вручную используя метод динамической загрузки библиотек (но это требует дополнительной модификации исходного кода программы). Также при использовании динамической загрузки программа скомпилируется и в случае отсутствия какой-то одной библиотеки, часть программы может работать, как и задумано, в то время как при динамической компоновке даже не получиться скомпилировать приложение.

## 13. Что такое динамическая компоновка?

См. выше.

## 14. Что такое динамическая загрузка (Windows/Linux)?

См. выше.

## 15. Нужно ли "оформлять" каким-то специальным образом исходный код приложения, которое использует динамическую библиотеку (Windows/Linux)?

Нет, только подключить заголовочный файл библиотеки.

## 16. Особенности реализации функций, использующих динамическое выделение памяти, в динамических библиотеках.

Память должна выделять и освобождать вызывающая сторона или необходимо вызывающей стороне предоставлять функцию-деструктор выделенных данных внутри библиотеки.

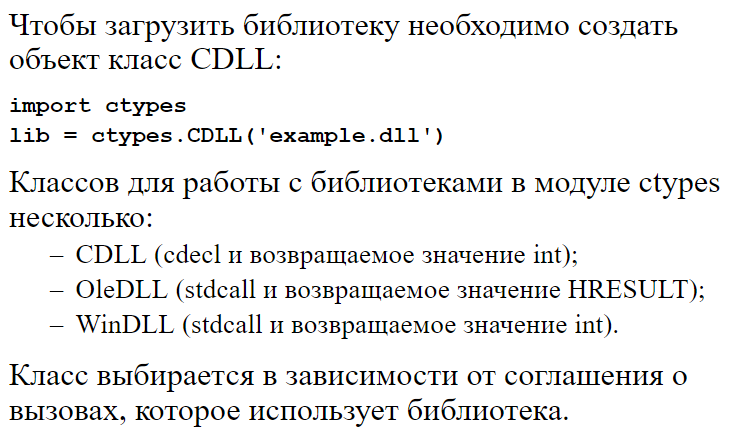
## 17. Ключи -I, -l, -L компилятора gcc.

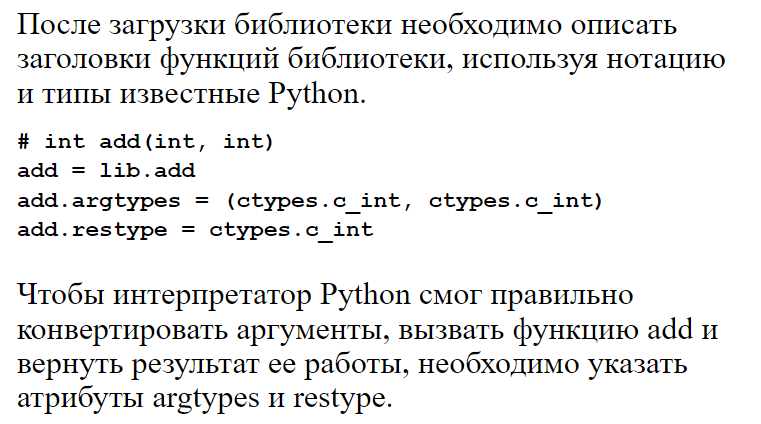
* I: добавить каталог dir в список поиска каталогов, содержащих include-файлы
* l: использовать библиотеку liblibrary.a при редактировании связей
* L: добавить директорию dir в список поиска библиотек

## 18. Проблемы использования динамической библиотеки, реализованной на одном языке программирования, и приложения, реализованного на другом языке программирования.

* Не совместимость типов данных разных ЯП
* Разные соглашения вызова функций

## 19. Модуль ctypes. Загрузка библиотеки. Представление стандартных типов языка Си. Импорт функций из библиотеки. Проблемы, которые при этом возникают.





При работе с ctypes возникают следующие проблемы:

* В языке Си используются идиомы, которых нет в языке Python. Например, функция возвращает одно из значений через свой аргумент. Поэтому решение «в лоб» обречено на неудачу.
* Целые числа в Python «неизменяемые» объекты. Попытка их изменить вызовет исключение. Поэтому для аргументов, которые «используют» указатель, необходимо с помощью описанных в модуле ctypes совместимых типов создать объект и передать именно его.

Итоги ctypes:

* Основная проблема использования этого модуля с большими библиотеками – написание большого количества сигнатур для функций и, в зависимости от сложности функций, функций-оберток.
* Необходимо детально представлять внутренне устройство типов Python и то, каким образом они могут быть преобразованы в типы Си.
* Альтернативные подходы – использование Swig или Cython.

## 20. Написание модуля расширения для Python (основные шаги).

Для интеграции Си кода в Питон может использоваться модуль расширения. Для этого в коде на Си используется библиотека “python.h”, которая предоставляет интерфейс для работы с объектами питон PyObject.

Задача программиста преобразовать все Питон объекты на входе (аргументы функции) в форматы данных Си и на выходе (результат работы функции) в объекты Питон.

# Бинарные операции. Битовые поля.

## 1. К каким типам в языке Си применимы битовые операции?

Битовые операции применимы только к целочисленным переменным.

## 2. Особенности использования битовых операций со знаковыми целыми типами.

Обычно эти операции стараются выполнять над беззнаковыми целыми, чтобы повысить переносимость программы и не было путаницы с битом, который отвечает за знак.

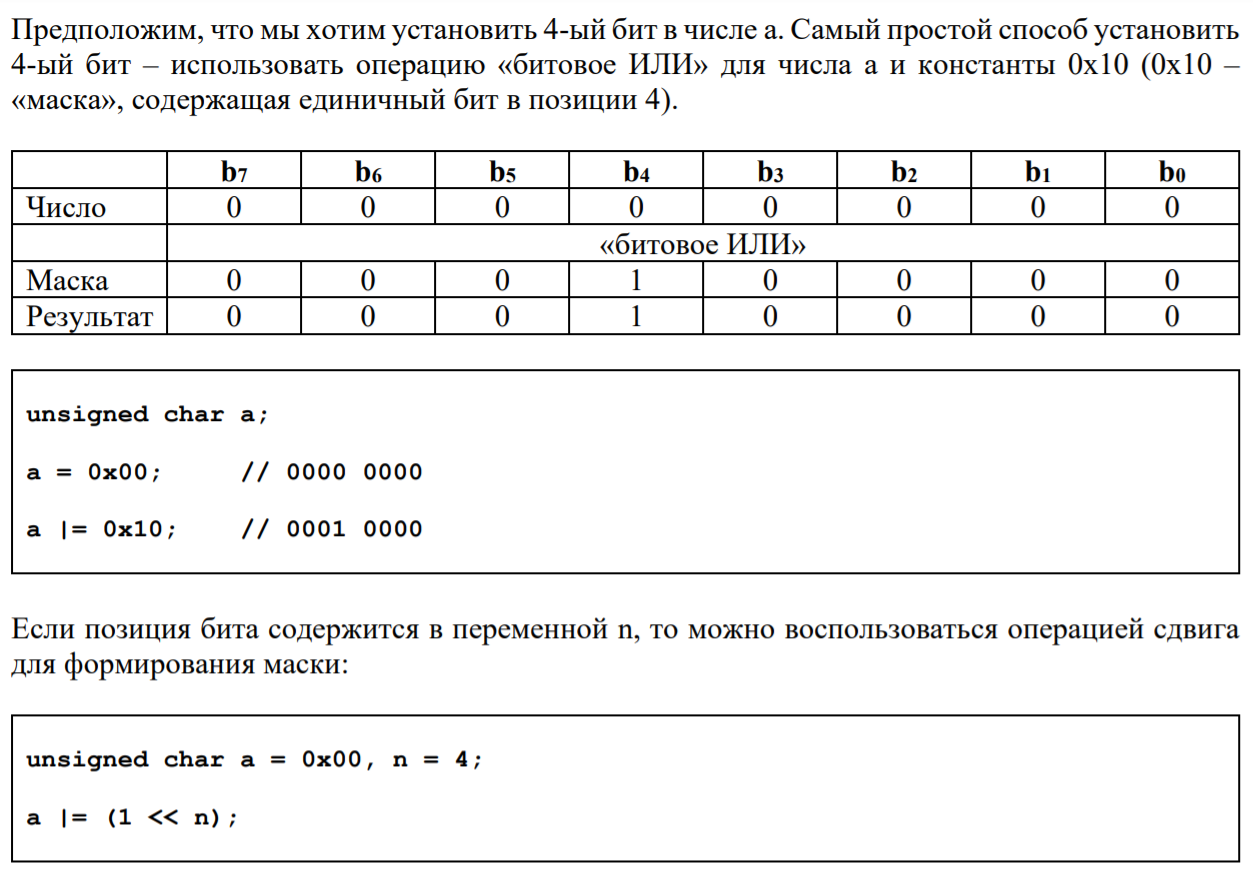
У знаковых целых поведение работы со знаковым битом зависит от реализации компилятора.

## 3. Какие битовые операции есть в языке Си? Приведите примеры.

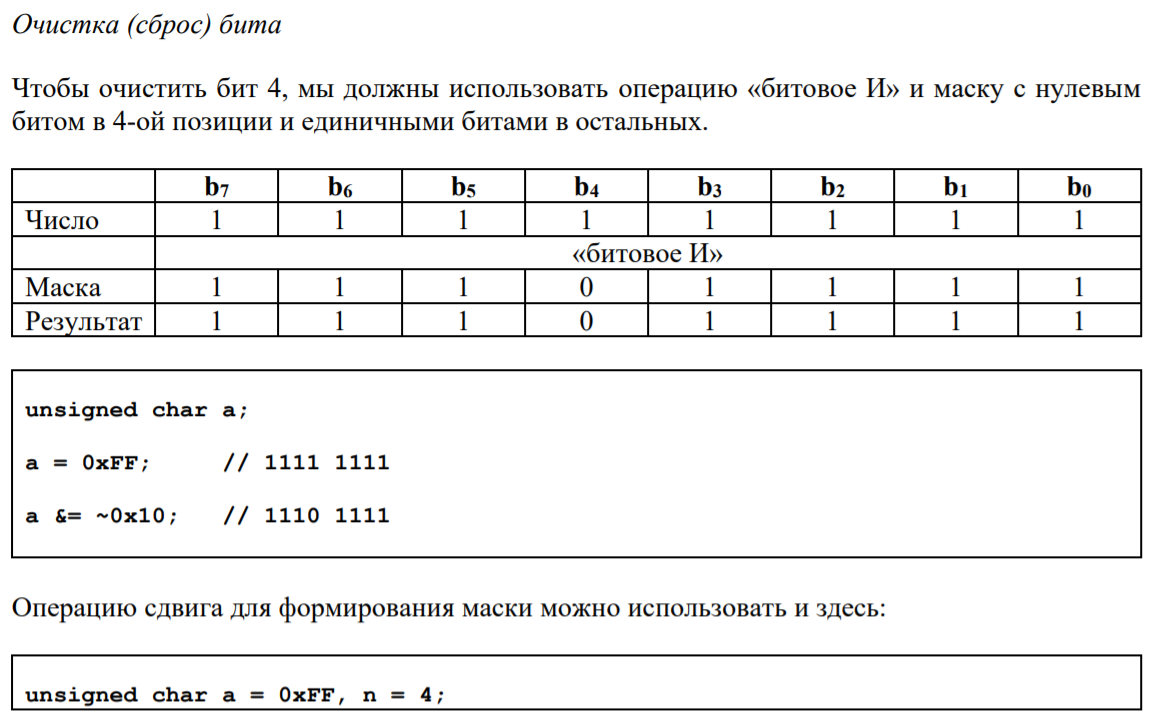
Язык Си поддерживает все битовые операции, к которым относятся

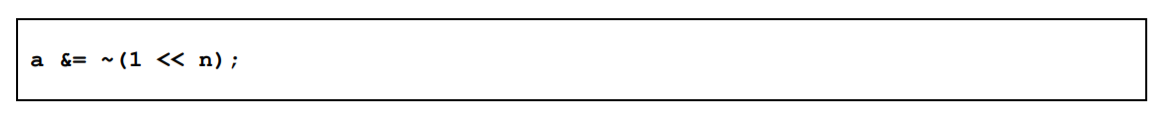
* операции сдвига
  + сдвиг влево (<<);
  + сдвиг вправо (>>);
* побитовые операции
  + Дополнение (~)
  + И (&);
  + Исключающее или (^);
  + Или (|).

## 4. Как установить указанный бит?

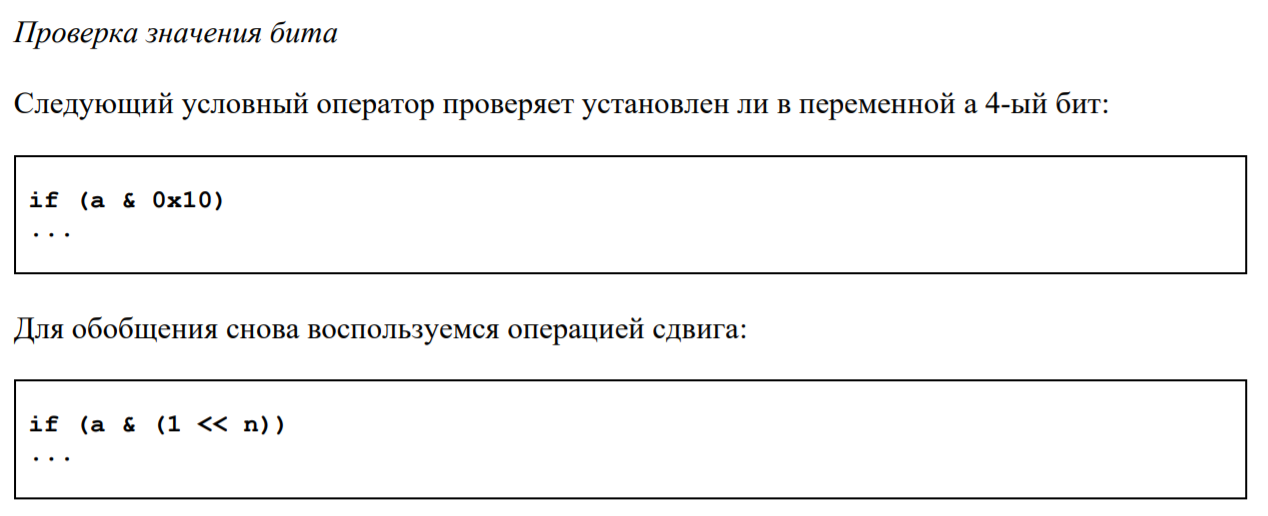


## 5. Как сбросить указанный бит?





## 6. Как проверить, что указанный бит установлен?



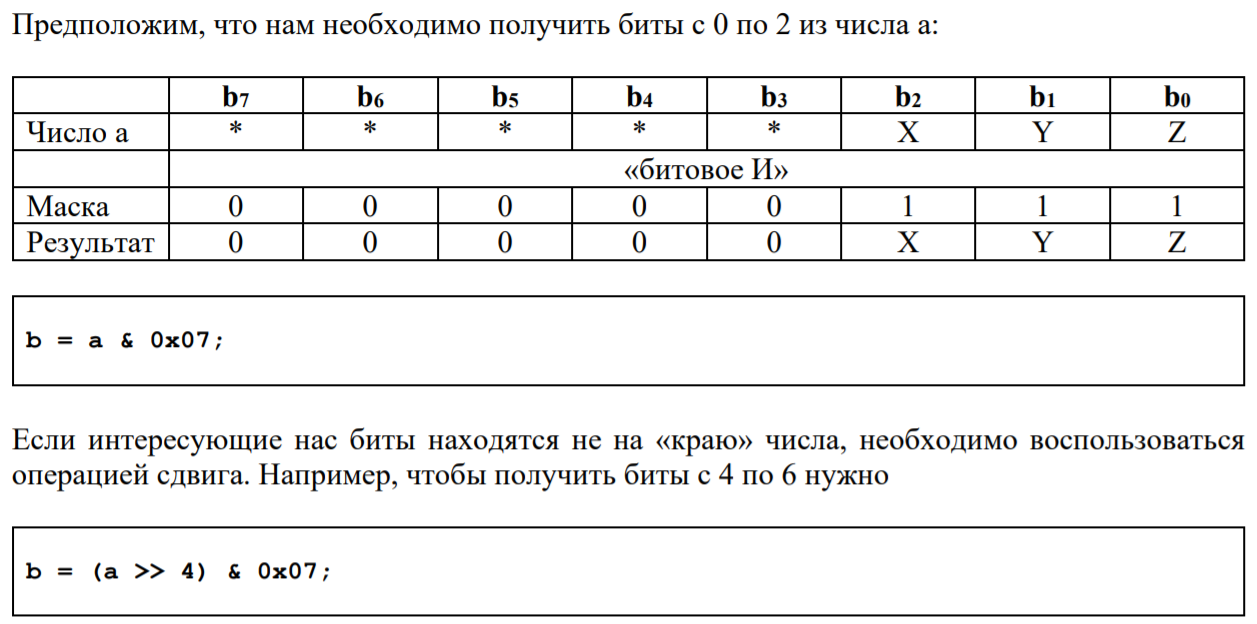
## 7. Как изменить значение указанного бита на противоположное?

Аналогично сбросу бита нужно взять исключающее или.

## 8. Как установить сразу несколько бит?

Для установки значения нескольких бит необходимо сначала занулить требуемые биты числа, а затем применить операцию битового или.

## 9. Как получить значение нескольких бит?



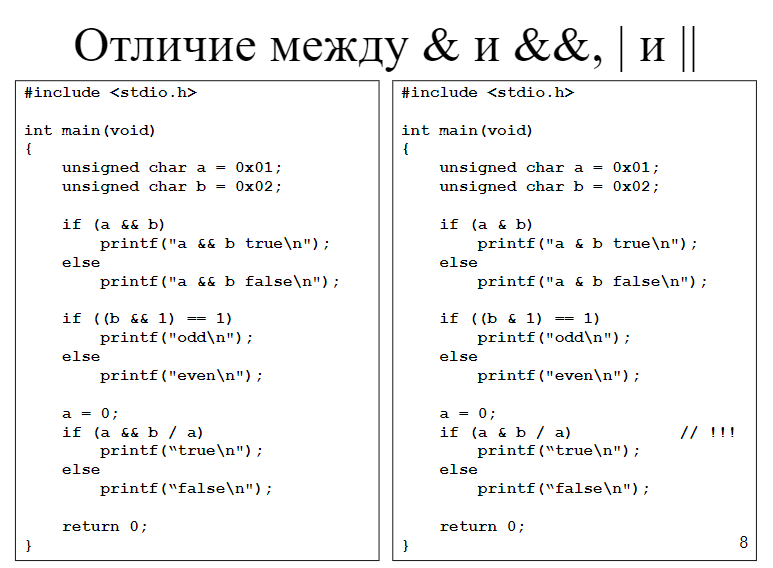
## 10. С помощью какой битовой операции можно разделить целое число на 2^n?

number >> n

## 11. С помощью какой битовой операции можно умножить целое число на 2^n?

number << n

## 12. Битовые операции VS логические операции.



## 13. Что такое битовое поле?

Битовое поле - особый тип структуры, определяющей, какую длину имеет каждый член в битах. Стандартный вид объявления битовых полей, следующий:

struct имя\_структуры

{

тип имя1: длина;

тип имя2: длина;

...

тип имяN: длина;

};

Битовые поля должны объявляться как целые, unsigned или signed.

## 14. Преимущества и недостатки битовых полей по сравнению с битовыми операциями.

Использование битовых полей позволяет более явно определить каким-образом происходит распределение битов в сущности, но при этом усложняется одновременная работа с несколькими полями.

## 15. Что задает значение CHAR\_BIT? В каком стандартном заголовочном файле его можно найти?

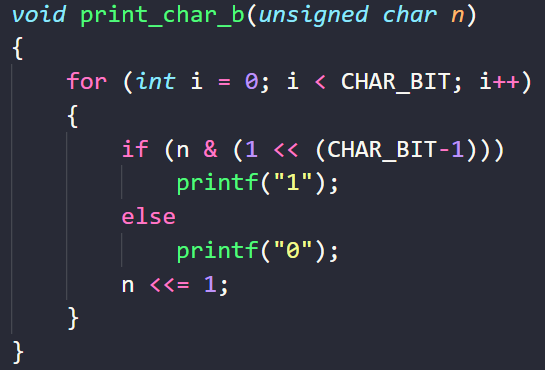
CHAR\_BIT — это число битов в char. В наши дни почти все архитектуры используют 8 бит на байт, но это не всегда так. Некоторые старые машины раньше имели 7-битный байт.

Его можно найти в <limits.h>.

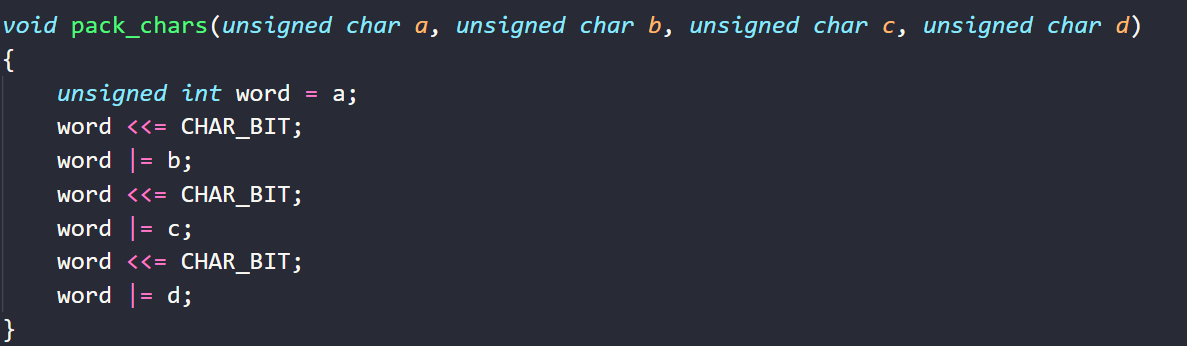
## 16. Существуют ли значения SHORT\_INT\_BIT, INT\_BIT и т.д.? Почему?

Нет, не существуют. С помощью функции sizeof всегда можно узнать сколько байт памяти выделено под конкретный тип, соответственно всегда можно узнать сколько бит памяти.

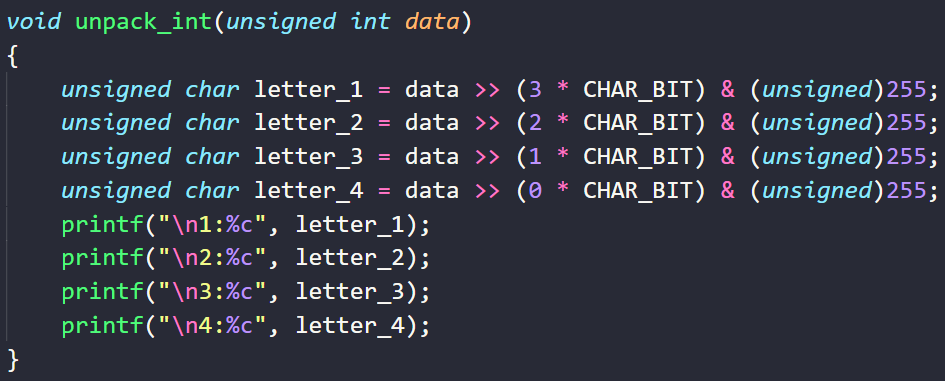
## 17. Напишите функцию, которая использует битовые операции, для вывода числа в двоичной системе счисления.



## 18. Упакуйте четыре символа в беззнаковое целое. Длина беззнакового целого равна 4.



## 19. «Распакуйте» беззнаковое целое число в четыре символа. Длина беззнакового целого равна 4.



20. Напишите функцию для циклического сдвига значения целочисленной величины на n позиций вправо/влево.

21. Запишите в одно беззнаковое целое число (длина беззнакового целого равна 4 байта) структуру, содержащую данные о файле аудиозаписи:

22. жанр ("народная "классическая "кантри "джаз "шансон "бардовская "поп "рэп "рокн-ролл "рок "электронная "экзотическая "церковная "военная "детская "аудиокнига");

23. стерео/моно;

24. длительность в секундах (от 1 до 8192);

25. размер файла в кб (от 1 до 16384).

26. Решите задачу несколькими способами.

# Неопределенное поведение

## 1. Что такое побочный эффект?

Свойство в определённых ситуациях выдавать результат, зависящий от реализации компилятора (библиотеки, микросхемы) и случайных факторов наподобие состояния памяти или сработавшего прерывания.

## 2. Какие выражения стандарт С99 относит к выражениям с побочным эффектом?

* Модификация данных.
* Обращение к переменным, объявленным как volatile.
* Вызов системной функции, которая производит побочные эффекты (например, файловый ввод или вывод).
* Вызов функций, выполняющих любое из вышеперечисленных действий.

## 3. Почему порядок вычисления подвыражений в языке Си неопределен?

Для возможности разработчикам компилятора делать оптимизации кода.

## 4. Порядок вычисления каких выражения в языке Си определен?

В Си однозначно определяется порядок вычисления логических выражений («ленивые» выражения), они вычисляются строго слева направо.

Также однозначно определен порядок вычисления операции «запятая» и тернарного оператора.

## 5. Что такое точка следования?

* Компилятор вычисляет выражения. Выражения будут вычисляться почти в том же порядке, в котором они указаны в исходном коде: сверху вниз и слева направо.
* *Точка следования* – это точка в программе, в которой программист знает какие выражения (или подвыражения) уже вычислены, а какие выражения (или подвыражения) еще нет.

## 6. Какие точки следования выделяет стандарт С99?

* Между вычислением левого и правого операндов в операциях &&, || и "," .
* Между вычислением первого и второго или третьего операндов в тернарной операции.
* В конце полного выражения.

a = b; if (); switch (); while (); do {} while (); for (x; y; z); return x;

* Перед входом в вызываемую функцию (порядок, в котором вычисляются аргументы не определен, но эта точка следования гарантирует, что все ее побочные эффекты проявятся на момент входа в функцию).
* В объявлении с инициализацией на момент завершения вычисления инициализирующего значения.

## 7. Почему необходимо избегать выражений, которые дают разный результат в зависимости от порядка их вычисления?

Т.к. программист не может гарантировать ожидаемое значение, в программе будут порождаться ошибки, которые будет очень сложно найти, ведь в один момент времени значение может совпасть с ожидаемым, а в другой момент времени нет. Также выражения с неопределенным поведением очень сложно поддерживать, т.к. никто кроме автора не может со 100% вероятностью сказать, чего хотел программист.

## 8. Какие виды "неопределенного" поведения есть в языке Си?

* Unspecified behavior.

Стандарт предлагает несколько вариантов на выбор. Компилятор может реализовать любой вариант. При этом на вход компилятора подается корректная программа.

Например: все аргументы функции должны быть вычислены до вызова функции, но они могут быть вычислены в любом порядке.

* Implementation-defined behavior.

Похоже на не специфицированное (unspescified) поведение, но в документации к компилятору должно быть указано, какое именно поведение реализовано.

Например: результат x % y, где x и y целые, а y отрицательное, может быть как положительным, так и отрицательным.

* Undefined behavior

Такое поведение возникает как следствие неправильно написанной программы или некорректных данных. Стандарт ничего не гарантирует, может случиться все что угодно.

## 9. Почему "неопределенное" поведение присутствует в языке Си?

Причина неопределенности порядка вычислений – простор для оптимизации.

## 10. Какой из видов "неопределенного" поведения является самым опасным? Чем он опасен?

Самым опасным является “undefined behavior”, так как при его возникновении может получиться абсолютно любой результат, от разрушения стека до аварийного завершения программы.

## 11. Как бороться с неопределенным поведением?

* Избегать сложных выражений
* Включайте все предупреждения компилятора, внимательно читайте их.
* Используйте возможности компилятора (-ftrapv).
* Используйте несколько компиляторов.
* Используйте статические анализаторы кода (например, clang).
* Используйте инструменты такие как valgrind, Doctor Memory и др.
* Используйте утверждения.

## 12. Приведите примеры неопределенного поведения.

* Использование неинициализированных переменных.
* Переполнение знаковых целых типов.
* Выход за границы массива.
* Использование «диких» указателей.

## 13. Приведите примеры поведения, зависящего от реализации.

Например: результат x % y, где x и y целые, а y отрицательное, может быть как положительным, так и отрицательным.

## 14. Приведите примеры неспецифицированного поведения.

Например: все аргументы функции должны быть вычислены до вызова функции, но они могут быть вычислены в любом порядке.

## 15. АТД

???

## 16. Что такое модуль?

Программу удобно рассматривать как набор независимых модулей.

## 17. Из каких частей состоит модуль? Какие требования предъявляются к этим частям?

Модуль состоит из двух частей: интерфейса и реализации.

* *Интерфейс* описывает, что модуль делает. Он определяет идентификаторы, типы и подпрограммы, которые будут доступны коду, использующему этот модуль.
* *Реализация* описывает, как модуль выполняет то, что предлагает интерфейс.

У модуля есть один интерфейс, но реализаций, удовлетворяющих этому интерфейсу, может быть несколько.

Часть кода, которая использует модуль, называют *клиентом*. Клиент должен зависеть только от интерфейса, но не от деталей его реализации.

## 18. Назовите преимущества модульной организации программы. Приведите примеры.

* Абстракция (как средство борьбы со сложностью)

Когда интерфейсы модулей согласованы, ответственность за реализацию каждого модуля делегируется определенному разработчику.

* Повторное использование

Модуль может быть использован в другой программе.

* Сопровождение

Можно заменить реализацию любого модуля, например, для улучшения производительности или переноса программы на другую платформу.

## 19. Какие виды модулей вы знаете? Приведите примеры.

* Набор данных

Набор связанных переменных и/или констант. В Си модули этого типа часто представляются только заголовочным файлом. (float.h, limits.h.)

* Библиотека

Набор связанных функций.

* Абстрактный объект

Набор функций, который обрабатывает скрытые данные.

* Абстрактный тип данных

Абстрактный тип данных – это интерфейс, который определяет тип данных и операции над этим типом. Тип данных называется абстрактным, потому что интерфейс скрывает детали его представления и реализации.

## 20. Что такое тип данных?

Тип данных (тип) — [множество](https://ru.wikipedia.org/wiki/%D0%9C%D0%BD%D0%BE%D0%B6%D0%B5%D1%81%D1%82%D0%B2%D0%BE) значений и операций над этими значениями.

## 21. Что такое абстрактный тип данных?

Абстрактный тип данных – это интерфейс, который определяет тип данных и операции над этим типом. Тип данных называется абстрактным, потому что интерфейс скрывает детали его представления и реализации.

## 22. Какие требования выдвигаются к абстрактному типу данных?

Абстрактный тип данных представляется как кластер операций. Операции получают и возвращают аргументы определенных типов, поэтому для полноценного описания АТД необходимо еще эти типы определить. Детали реализации должны быть скрыты от пользователя.

## 23. Абстрактный объект VS абстрактный тип данных.

Главный минус абстрактного объекта по сравнению с абстрактным типом данных заключается в том, что абстрактный объект существует в одном единственном экземпляре в программе, что накладывает серьезные ограничения. В свою очередь абстрактный тип данных требует большей ответственности от программиста (например, инициализация и освобождение памяти из-под переменных).

## 24. Средства реализации модулей в языке Си.

Заголовочные файлы, библиотека.

## 25. Что такое неполный тип данных в языке Си?

Стандарт Си описывает неполные типы как «типы, которые описывают объект, но не предоставляют информацию нужную для определения его размера».

Пока тип неполный его использование ограничено. Описание неполного типа должно быть закончено где-то в программе.

## 26. Приведите примеры описания неполного типа данных? (А кроме структур ;) ?)

* Структуры и объединения.
* При работе с массивами, в функциях не указывается их размер (кол-во элементов), таким образом, мы не можем вычислить размер массива.

## 27. Какие действия можно выполнять с неполным типом данных?

Допустимо определять указатель на неполный тип

**typedef struct t \*T;**

Можно

* определять переменные типа T.
* передавать эти переменные как аргументы в функцию.

Нельзя

* применять операцию обращения к полю (->).
* разыменовывать переменные типа T.

## 28. Для чего при реализации абстрактного типа данных используется неполный тип данных языка Си?

Неполный тип данных используются для скрытия деталей реализации АТД и операций над ним.

## 29. Проблемы реализации АТД на языке Си.

* Именование

В примерах использовались имена функций, которые подходят для многих АТД (create, destroy, is\_empty). Если в программе будет использоваться несколько разных АТД, это может привести к конфликту. Поэтому имеет смысл добавлять название АТД в название функций (stack\_create, stack\_destroy, stack\_is\_empty).

* Обработка ошибок
  + Интерфейс — это своего рода контракт.
  + Интерфейс обычно описывает *проверяемые* ошибки времени выполнения и *непроверяемые* ошибки времени выполнения и исключения.
  + Реализация не гарантирует обнаружение непроверяемых ошибок времени выполнения. Хороший интерфейс избегает таких ошибок, но должен описать их.
  + Реализация гарантирует обнаружение проверяемых ошибок времени выполнения и информирование клиентского кода.
* «Общий» АТД
  + Хотелось бы чтобы стек мог «принимать» данные любого типа без модификации файла stack.h.
  + Программа не может создать два стека с данными разного типа.

Решение – использовать void\* как тип элемента, НО:

* элементами могут быть динамически выделяемые объекты, но не данные базовых типов int, double.
* стек может содержать указатели на что угодно, очень сложно гарантировать правильность.

## 30. Есть ли в стандартной библиотеке языка Си примеры абстрактных типов данных?

va\_list, FILE\*.