КОНСПЕКТЫ ОТВЕТОВ НА ТИПОВЫЕ ВОПРОСЫ К ЭКЗАМЕНУ ПО ПРОГРАММИРОВАНИЮ

|  |  |  |  |
| --- | --- | --- | --- |
| № | Вопрос | Краткое изложение | Подробное изложение |
| 1 | Этапы получения исполняемого файла из исходного кода. Опции компилятора и компоновщика. | 1. Этапы получения исполняемого файла из исходного кода:    1. препроцессирование    2. компиляция    3. ассемблирование    4. компоновка | * 1. Препроцессирование – *замена всех директив препроцессора* на команды языка Си. В частности, происходят:      + Непосредственное объявление всех функций в теле вызывающего модуля;      + Замена всех макроопределений (#define) участками кода на языке Си;      + Выполнение условной компиляции и прагматических выражений   В итоге должен был бы появиться файл \*.i, полученный из \*.с.   * 1. Компиляция – *преобразование кода на языке Си в код на языке ассемблера*. Преобразование идёт по определённым правилам. В процессе преобразования проверяется наличие синтаксических и семантических ошибок в коде. В итоге должен был бы появиться файл \*.s, полученный из \*.i или нескольких файлов под началом \*.c.   2. Ассемблирование – *преобразование ассемблерного кода в машинный код*. Поскольку ассемблерный код считается составленным правильно, ассемблирование проходит без проблем. В итоге получается объектный файл \*.o.   В действительности первые три этапа выполняются компилятором GNU (MinGW) с помощью одной команды ***gcc [compiler options] –c src\_name.c*** , и если ошибки не будут найдены, в папке, содержащей **src\_name.c**, появится объектный модуль **src\_name.o** (если не указано иное опцией компилятора “***-o obj\_name.o***”).   * 1. Компоновка – *сборка приложения* из одного или нескольких объектных модулей. В итоге получается приложение \*.exe. Компилятор GNU (MinGW) выполняет компоновку с помощью команды ***gcc [linker options] src\_obj1.c src\_obj2.c src\_objN.c***, при этом появится приложение **src\_obj1.exe** (если не указано иное опцией компилятора “***-o exe\_name.exe***”). |
| 1. Строка запуска компилятора:    1. Стандартная строка запуска    2. Опции       * Опции компилятора       * Опции компоновщика | * 1. gcc [options] [output file] file\_1 [other files]   2. Основные опции компилятора и компоновщика:      + - -std – *определяет стандарт* языка Си, согласно которому требуется компилировать код. По умолчанию стоит стандарт c89. Стандарт c99 включается опцией ***-std=c99***. В папку с компилятором можно поместить файлик, позволяющий вместо ***gcc -std=c99*** набрать ***c99***.        - -Wall – заставляет компилятор учитывать *все* ошибки и предупреждения, в том числе не влияющие на работоспособность программы, такие как:          * Неинициализированные переменные          * Неиспользуемые переменные (инициализируются, но не используются)          * Слежение за указателями (проверка ключевого слова const)          * и так далее.        - -Werror – *все предупреждения считать ошибками*, компиляцию завершить по сценарию компиляции кода с ошибками.        - -pedantic – усиленная проверка на грамотность кода, выдача замечаний о возможных предупреждениях и ошибках выполнения программы.        - -c (--compile) – только *компиляция*, без сборки приложения        - -o filename – сохранение результатов в файл с именем filename.        - -ggdb – подготовка к сканированию с помощью *Dr.Memory*. |

|  |  |  |  |
| --- | --- | --- | --- |
| 2 | Функция main. Возвращаемое значение. Аргументы функции main. Внутренняя организация исполняемого файла. Превращение исполняемого файла в процесс. | 1. Функция main:    1. Заголовки функции main согласно стандарту С99    2. Возвращаемое значение    3. Организация аргументов функции main | 1. Функция main – *точка входа* программы на Си, обязательная составляющая каждой завершённой программы.    1. Заголовки функции main согласно стандарту C99:       * ***int main(void)***. Приложение, скомпилированное с таким заголовком, не берёт параметры во время запуска. Применяется в основном в случаях, когда параметры командной строки не нужны при работе с программой.       * ***int main(int*** *argc****, char\*\**** *argv****)*** или ***int main(int*** *argc****, char\**** *argv****[])***. Этот заголовок позволяет применять параметры командной строки при работе программы. Об организации аргументов функции main смотри пункт 1.3.    2. Возвращается код ошибки выполнения программы согласно соглашениям операционной системы. Они, как правило, описаны в константах, содержащихся в заголовочном файле <errno.h>. Если возвращается значение <=0, значит, системных ошибок не было. Считается хорошим тоном несистемные ошибки выполнения программы кодировать отрицательными значениями, нормальное выполнение программы – нулём.    3. Параметр argc функции main содержит число введённых параметров командной строки, argv – это массив строк, содержащих эти значения. Строки с аргументами в командной строке разделяются пробелами и знаками табуляции. argv[0] всегда является именем приложения, поэтому всегда argc>=1. Согласно стандарту, argv[argc]==NULL. Стандарт С99 требует от компилятора возможность обработать по крайней мере 32767 аргументов командной строки, что навряд ли под силу какой-либо ОС. |
| 1. Исполняемый файл:    1. Организация исполняемого файла    2. Превращение исполняемого файла в процесс. | Исполняемый файл имеет расширение **.exe**.   * 1. Его структура состоит из следующих частей:      + Заголовочные блоки (не менее одного). Как правило, там содержится информация о том, как правильно отображать функции в память.      + Секция .text. В ней содержится исполняемый код.      + Секция .bss. В ней хранятся неинициализированные глобальные и статические переменные.      + Секция .data. В ней содержатся инициализированные глобальные и статические переменные для чтения и изменения.      + Секция .rodata. В ней содержатся данные только для чтения.      + Таблица импорта. Через неё осуществляется связь с внешними библиотеками и вызов функций или переменных по адресам.      + Таблица экспорта. Через неё указывается, какие адреса из модуля могут быть импортированы внешней программой или внешним модулем.      + Таблица перемещений. Через неё вычисляются адреса всех данных и функций, используемых в программе.      + Метаданные. Включают информацию, относящуюся к сборке продукта, включая её манифест.   2. Превращение исполняемого файла в процесс. Этапы:      + Считывание программы из диска в ОЗУ.      + Загрузка динамических библиотек.      + Настройка ссылок.      + Планирование процесса.   При запуске процессу выделяется "абстрактная память" и помещает в неё исполняемый файл. Процесс считает, что в этой памяти содержатся только его данные. |

|  |  |  |  |
| --- | --- | --- | --- |
| 3 | Многофайловая организация проекта. Компиляция и компоновка многофайлового проекта. Каноническое оформление заголовочных файлов. | 1. Сравнительная характеристика однофайловой и многофайловой организаций проекта 2. Структура многофайлового проекта | 1. Сравнение однофайловой и многофайловой организаций проектов:    1. Недостатки однофайловой:       * *Одновременная работа* над программой нескольких программистов становится неэффективной.       * *Ориентирование* в тексте программы становится сложным.       * Даже при локальном изменении кода *перекомпилируется* весь проект.    2. Достоинства многофайловой:       * Возможность *распределения* работы над проектом между несколькими программистами.       * Код программы более *удобочитаемый*.       * Сокращает время *повторной* компиляции.       * Повторное *использование* кода в других проектах. 2. Многофайловый проект состоит из:    1. Файла с *основной* программой (с точкой входа – функцией main);    2. *Заголовочных* файлов с глобальными переменными, объявлениями функций и макросами;    3. Файлов *реализаций* функций из заголовочных файлов;    4. Скомпилированных *библиотек* (стандартных, статических или динамических). |
| 1. Компиляция многофайлового проекта:    1. Объявления и определения    2. Этапы компиляции    3. Ошибки компиляции | * 1. Объекты, появляющиеся в файлах проекта, могут быть объявлены или определены.      + **Определением** называется **выделение памяти** под новый программный объект (переменную, функцию, блок и т. п.), *возможно*, с присвоением этому объекту некоторого значения.      + **Объявлением** называется **информация о существовании** программного объекта, используемого в модуле или блоке, в другом месте этого же файла или в другом файле.   2. Этапы компиляции многофайлового проекта:      + Проверка правильности и достаточности подключений заголовочных файлов;      + Сборка объектных модулей для каждого блока реализации, а также модуля(ей) с точкой(ами) входа.   3. Ошибки компиляции:      + Неявное задание функции (implicit declaration) – вызывается функция, заранее не объявленная      + Необъявленная переменная – использование переменной, не объявленной в текущем блоке или блоке более старшего уровня (в том числе глобально);      + Несоответствие типов      + Не найден файл      + Переопределение функции и т.д. (их очень много, перечисляем, что помним) |
| 1. Компоновка многофайлового проекта:    1. Этапы компоновки    2. Ошибки компоновки | 1. При компоновке:    1. Собираем все необходимые собранные объектные файлы во главе с объектным файлом, содержащим точку входа, в один исполняемый файл.    2. Возможные ошибки:       * Не найден файл       * Отсутствует точка входа       * Не определено объявленное имя;       * Переопределение функции или другого внешнего имени и т.д. |
| 1. Защита от повторных включений (include guard) – 2 способа | 1. Часто при подключении заголовочных файлов происходит ошибка переопределения, так как в соответствующем файле реализации эти функции уже объявлены и определены, а порядок подключения не определён стандартом. Чтобы избежать этого, используют 2 метода защиты от повторных включений (include guard):    1. ***#ifndef \_\_LIBNAME\_\_ #define \_\_LIBNAME\_\_*** <тело заголовочного файла> ***#endif***    2. ***#pragma once*** (только С99) |

|  |  |  |  |
| --- | --- | --- | --- |
| 4  и  5 | **Автоматизация сборки проекта, утилита make. Сценарий сборки проекта. Простой сценарий сборки.** Использование переменных и комментариев. Сборка программы с разными параметрами компиляции. Автоматические переменные. Шаблонные правила. | 1. Суть автоматизированной сборки проекта:    1. Основная задача    2. Данные для автоматической сборки    3. Средства автоматической сборки проекта 2. Утилита make:    1. Общая характеристика    2. Основные разновидности | * 1. Задача автоматизации сборки проекта – избавить программиста от необходимости каждый раз печатать объёмные вызовы компилятору и компоновщику в весьма больших проектах.   2. Данными для автоматической сборки являются файлы заголовков, реализации и библиотеки (вход), исполняемые файлы и библиотеки (выход).   3. Для автоматической сборки проекта применяют несколько способов:      + BAT-файлы      + Специализированные программные средства сборки (т.н. make)  1. Make:    1. **make** — утилита, автоматизирующая процесс преобразования файлов из одной формы в другую.    2. Известны следующие разновидности средств автоматической сборки проекта (т.н. make):       * GNU Make (мы им пользовались)       * BSD Make       * Microsoft Make (nmake) |
| 1. Сценарий сборки проекта:    1. Граф задач сборки    2. Названия файлов, их структура    3. Простой сценарий сборки    4. Условные конструкции в сценарии сборки    5. Шаблонные правила | * 1. Структура команды в мэйк-файле:      + Строка зависимостей: «список целей: список аргументов»      + [tab] Строка вызова компилятора   2. Простой сценарий сборки:      + Очищаем каталог от старых исполняемых и объектных файлов      + Собираем объектные файлы для модулей      + Собираем объектные файлы для точек входа      + Собираем исполняемые файлы, по одной точке входа на каждый, из объектных модулей.   3. Условные конструкции: **ifeq(op1, op2) oper1 [else oper2] endif**, **ifneq(op1, op2) oper1 [else oper2] endif**   4. Шаблонные правила:      + %.[расширения целей]: %.[расширения аргументов] – для всех файлов с данной маской поимённо      + \*.[расширение] – все файлы с данным расширением. |
| 1. Использование переменных в make-файле:    1. Объявление переменных    2. Использование переменных    3. Автоматические переменные | 1. Все переменные имеют строковые значения    1. Объявление и инициализация: **VAR\_NAME := value**    2. Использование переменных: **$(VAR\_NAME) VAR\_NAME += value**    3. Автоматические переменные – переменные, автоматически принимающие определённые значения перед выполнением описанных в правиле команд:       * **$^** - список зависимостей       * **$@** - имя цели       * **$<** - первая зависимость |

|  |  |  |  |
| --- | --- | --- | --- |
| 6  и  7 | **Указатели**, операции над указателями, массивы, адресная арифметика.  Указатель void\*, указатели на функции | 1. Что такое указатель 2. Классификация указателей 3. Определение переменной типа указатель 4. Операции \* (разыменование указателя) и & (возврат адреса операнда) – обе префиксные | 1. Указатель – переменная, содержащая адрес. 2. Разновидности указателей:    1. Типизированный (на данные определённого типа) – type\*    2. Бестиповой (на произвольное место в памяти) – void\*    3. Указатель на функцию 3. Шаблон определения переменной типа указатель: ***type \*ptr1\_name, \*ptr2\_name, not\_ptr\_name***; 4. Доступ к данным: ***not\_ptr\_name = \*ptr1\_name;***   Получение адреса: ***ptr2\_name = &not\_ptr\_name;*** |
| 1. Адресная арифметика:    1. Сложение указателя с числом    2. Вычитание числа из указателя    3. Разность указателей    4. Связь указателей и массивов    5. Сравнение указателей | 1. Адресная арифметика применяется к типизированным указателям, указывающим на данные одного типа данных. Предполагается, что адреса отстоят друг от друга на расстояние, кратное размеру типа, а также что размер целого числа-операнда будет сведён к 4-байтовому.    1. Указатель + число = указатель на место, адрес которого будет больше на число\*(sizeof(type)) байтов.    2. Указатель - число = указатель на место, адрес которого будет меньше на число\*(sizeof(type)) байтов.    3. Указатель – указатель = число ячеек памяти размера sizeof(type), размещенных между двумя адресами.    4. Идентификатор массива имеет тип ссылки (константного указателя) на начало области памяти, в которой расположен массив.    5. Указатели сравниваются по адресам. При этом можно сравнивать либо однотипные указатели, либо указатель и NULL. |
| 1. Указатель void\*:    1. Определение    2. Зачем он нужен 2. Указатель на функцию:    1. Что содержит    2. Как обращаться    3. void\* в качестве параметра маски функции (на примере qsort из <stdlib.h>) | 1. void\* -- бестиповой указатель, указывает на произвольное место в памяти. К нему неприменима адресная арифметика. Основная миссия – упрощение преобразований указателей, универсализация структур и формальных параметров функций. 2. Указатель на функцию: ***(\*func)(params)***; на месте params следует перечислить типы аргументов функции. Например, функция ***void qsort(void\* base, size\_t memb, size\_t size, int (\*compar)(const void\*, const void\*))*** из заголовочного файла <stdlib.h> имеет в качестве формального параметра указатель на функцию, сравнивающие некие 2 значения, задаваемые ссылками на них, без ограничения на типы параметров. То есть таким параметром может быть и функция ***int \*compar\_int(const int\*, const int\*))***, и ***long double \*compar\_longd(const long double\*, const long double\*))***. |

|  |  |  |  |
| --- | --- | --- | --- |
| 8 | Статические массивы | 1. Объявление массивов 2. Представление в памяти компьютера | 1. Объявление статических массивов: **тип имя[размер\_1][ размер\_2][ размер\_3]**. Первая размерность может быть при этом не указана либо быть переменной. 2. Представление в памяти компьютера: единый блок, от начала до конца, многомерные – по принципу «массив массивов». Пусть s1,s2 – размеры массива; i, j – индексы элемента в этом массиве. Тогда смещение элемента в массиве будет равно (i\*s2+j), то есть можно считать многомерный массив организованным аналогично одномерному. |
| 1. Построчная обработка массива с помощью указателей | 1. Указатели пробегают все смещения при переменных i и постоянных j. |
| 1. Постолбцовая обработка массива с помощью указателей | 1. Указатели пробегают все смещения при переменных j и постоянных i. |

|  |  |  |  |
| --- | --- | --- | --- |
| 9 | Динамические массивы | 1. Объявление массивов:    1. одномерный массив    2. указатели на строки    3. одним дампом 2. Представление в памяти компьютера | Динамические массивы требуют отдельного выделения динамической памяти в куче.   * 1. Как одномерный массив; представление как статического массива. Выделение и освобождение памяти тривиально.   2. Объявляется массив указателей на массивы-строки, элементы которых имеют некий тип, и т.д. Память под массив строк и массивы-строки выделяется и освобождается отдельно.   3. Выделяется единый дамп памяти, первая часть которого идёт на массивы указателей на строки, вторая содержит значения. |
| 1. Плюсы и минусы каждого вида динамического массива | |  |  |  | | --- | --- | --- | | № способа | Достоинства | Недостатки | | 1 | * Простота выделения и освобождения памяти * Минимум занимаемого места * Возможность использовать как одномерный массив | * Необходимость всё время обращаться к элементам по сложному индексу * Средства для контроля работы с памятью не могут отследить выход за пределы строки | | 2 | * + Возможность обмена строки через обмен указателей.   + СРПР может отследить выход за пределы строки. | * + Сложность выделения и освобождения памяти.   + Память под матрицу "не лежит" одним куском. | | 3 | * Простота выделения и освобождения памяти. * Возможность использовать как одномерный массив. * Перестановка строк через обмен указателей. | * Сложность начальной инициализации. * СРПР не может отследить выход за пределы строки. | |

|  |  |  |  |
| --- | --- | --- | --- |
| 10 | Строки. Инициализация строковых переменных. Массив символов и указатель на строковый литерал. Ввод/вывод строк. Особенности обработки строк. | 1. Представление строк: 2. Инициализация строковых переменных 3. Массив символов и указатель на строковый литерал | 1. Строки – char\*, специального типа нет. Представляются последовательностью байтов, заканчивающуюся нулём. 2. Инициализация:    1. Как массив, размер которого по крайней мере на 1 больше числа значащих символов.    2. Как строковый литерал (область памяти в стеке, недоступная для прямого редактирования) |
| 1. Ввод, вывод строк:    1. Ввод (стандартные функции)    2. Вывод (стандартные функции) | 1. Стандартные функции ввода-вывода строк:    1. Ввод       * scanf(“%s”,str), fscanf(f,”%s”,str); // символы-разделители строк – в том числе пробелы и табуляция. Считывание идёт в заранее выделенный буфер. Программист сам отслеживает ограничения на длину.       * gets(buf), fgets(buf, n\_max, f); // считывает строку вплоть до символа перевода строки или конца файла (записывая и его, а после него – терминирующий нуль).    2. Вывод       * printf, fprintf // вывод строки без её перевода       * puts, fputs // вывод с переводом строки |
| 1. Особенности обработки строк:    1. Возможный выход за пределы строки    2. Положение терминирующего нуля    3. Внимательная работа с указателями | Выход за пределы выделенного буфера строки – крайне нежелательный эффект, не отслеживается компилятором и ложится на плечи программиста.  Терминирующий нуль при преобразованиях строк необходимо сдвигать, а также обеспечивать его нахождение в буфере  Пробег по строке с помощью указателей также должен быть аккуратным, использование вне пределов буфера чревато последствиями. |

|  |  |  |  |
| --- | --- | --- | --- |
| 11 | Время жизни, область видимости переменных | 1. Время жизни переменных 2. Размещение объектов в памяти в зависимости от времени жизни 3. Область видимости переменных | 1. **Время жизни** – это интервал времени выполнения программы, в течение которого программный “объект” существует. Подразделяется на глобальное и локальное. 2. Глобальные и статические переменные обладают глобальным временем жизни и размещаются в секциях bss, data, rodata. Локальные переменные имеют локальное время жизни и существуют, пока работает блок, в котором они определены. Все функции имеют глобальное время жизни и видимость. 3. Область видимости переменной – та часть кода, в которой переменная может быть реализована. |
| 1. Правила, связанные с областями видимости | 1. Правила:    1. Переменные, определённые в некотором блоке, будут видны во все блоки, вложенные в данный.    2. Имя переменной во вложенном блоке, совпадающее с именем переменной, определённой в блоке-предке, легально и закрывает видимость одноимённой переменной-предка. |

|  |  |  |  |
| --- | --- | --- | --- |
| 12 | Журналирование | 1. Зачем и как применяется 2. Реализация для:    1. глобальных переменных    2. доступа через функции    3. статических переменных | Журналирование – процесс записи информации о происходящих с каким-то объектом (или в рамках какого-то процесса) событиях в журнал (например, в файл). Этот процесс часто называют также **аудитом**.  Файловую переменную для журнала определяют глобальной и объявляют во всех файлах реализации проекта. Это позволяет вызывать функции записи в файл для журналирования отовсюду.  Для записи в журнал можно написать собственные функции, где переменная-указатель на файл с журналом может быть как глобальной, так и статической переменной. |

|  |  |  |  |
| --- | --- | --- | --- |
| 13 | Классы памяти | 1. auto 2. register 3. static 4. extern 5. умолчания | 1. auto – класс автоматических переменных, создаваемых при вызове функции и удаляемых после выхода из неё. 2. register – компилятору предъявляется пожелание о том, чтобы переменная использовала какой-нибудь регистр процессора для хранения. Как правило, компилятор игнорирует это резервное слово и способен сам решать, какой переменной можно выделять регистр процессора. 3. static – класс статических переменных, создаётся при первом вызове функции, а удаляется только при завершении работы программы. Компилятор хранит значения таких переменных от одного вызова функции до другого. 4. extern – класс внешних переменных, память под них не выделяется. Это означает объявление переменной, которая может быть объявлена или в текущем, или в некотором внешнем файле. 5. По умолчанию считается:    1. если переменная объявлена в теле функции без спецификатора класса памяти, по умолчанию он равен auto;    2. если переменная объявлена вне всех функций, она считается внешней и может быть использована в любом смежном файле, в т. ч. и текущем;    3. все функции внешние. |

|  |  |  |  |
| --- | --- | --- | --- |
| 14 | Стек и куча. Последовательность действий при работе с динамической памятью. | 1. Стек. Размещение в нём локальных переменных. Плюсы и минусы 2. Динамическая память. Размещение данных в ней. Плюсы и минусы 3. Последовательность действий при работе с динамической памятью. Менеджер кучи | 1. Автоматически локальные переменные размещаются в стеке. Стек – линейная структура в памяти.    1. Плюсы размещения локальных переменных в стеке:       * Память под локальные переменные выделяет и освобождает компилятор    2. Минусы:       * Время жизни локальной переменной "ограничено" блоком, в котором она определена.       * Размер размещаемых в стеке объектов должен быть известен на этапе компиляции.       * Размер стека в большинстве случаев ограничен. 2. Динамическая память реализована кучей (двоичной). Там хранятся данные. Можно выделять в куче место под переменную    1. Плюсы кучи == минусы стека    2. Минус кучи: ручное управление временем жизни. 3. Некоторое количество памяти выделяется операционной системой под базы данных выделенной памяти. Управляет этой базой данных менеджер кучи, поведение которого зависит от компилятора. Последовательность действий при работе с динамической памятью такова:    1. Подача запроса операционной системе на выделение некоторого участка памяти. Требуемый размер прилагается.    2. Если место найдено, менеджер кучи записывает адрес и размер выделенной памяти в базу данных и возвращает адрес в программу.    3. Программа как-то использует выделенную память    4. Подача запроса на освобождение выделенной по данному адресу памяти.    5. Менеджер кучи освобождает память, занимаемую областью, в которой находится данный адрес. При этом адрес освобождённой области остаётся в программе, а не удаляется. |

|  |  |  |  |
| --- | --- | --- | --- |
| 15 | Использование аппаратного стека. Стековый кадр. Указатель на локальную переменную. Переполнение буфера. Массивы на стеке. | 1. Использование аппаратного стека 2. Кадр стека 3. Возврат указателя на локальную переменную 4. Переполнение буфера | 1. В каждой программе компилятор обязан выделить память под аппаратный стек. Последний предназначается для:    1. вызова функции (call name). В стек заносятся: адрес возврата, адрес вершины стека до вызова программы.    2. возврата из функции (ret). Из стека вынимается всё то, что было положено до вызова подпрограммы.    3. передача фактических параметров функции. Передача идёт справа налево, очистка стека ложится на вызывающую сторону, результат (если нужен) возвращается через регистр EAX (по значению/ по адресу).    4. выделения и освобождения памяти под локальные переменные. 2. Стековый кадр – механизм передачи аргументов и выделения временной памяти с использованием аппаратного стека. В стековом кадре выделяют «слои»:    1. На дне – фактические параметры, в порядке, обратном объявлению;    2. адрес возврата    3. старый EBP    4. На вершине – локальные переменные функции. 3. Распространена ошибка, когда функция возвращает адрес локальной переменной. Это неизбежно влечёт ошибку времени выполнения при обращении по этому адресу. 4. Более распространена ошибка, когда в малый буфер записывается информация, превышающая по объёму этот самый буфер. Тогда поведение программы становится неопределённым и часто приводит к сбоям. |

|  |  |  |  |
| --- | --- | --- | --- |
| 16 | Функции с переменным числом параметров | 1. Идеи реализации:    1. Передача формальных параметров    2. Объявление функций с переменным числом параметров 2. Использование <stdarg.h> для реализаций определений функций | Фактические параметры передаются через стек в порядке, обратном их записи в вызове функции. Это позволяет сравнительно легко передавать переменное число параметров в вызываемую функцию.  Функции с переменным числом параметров объявляются так:  ***тип\_рез-та имя(<непустой список первых параметров>, …)***  Для прохождения стека с параметрами используют заголовочный файл <stdarg.h>, в котором объявлены:   * тип va\_list; * семейство функций   ***type va\_arg(va\_list ap, type)***  ***void va\_end(va\_list ap)***  ***void va\_start(va\_list ap, parmN)*** – установка указателя на стековый кадр в место нахождения фактического параметра parmN. |

|  |  |  |  |
| --- | --- | --- | --- |
| 17 | Структуры | 1. Формальное определение 2. Определение переменных структурного типа 3. Тег структуры | Структура представляет собой одну или несколько переменных (возможно, разного типа), которые объединены под одним именем, при этом каждая переменных занимает своё место в памяти. Это позволяет связать в единое целое различные по типу и значению данные, связанные между собой.  Формат определения: struct [tag] {type1 field1; type2 [field2]; …; typeN fieldN; } [variable] = {value1, value2, …, valueN}  Тег структуры – имя, позволяющее идентифицировать структуру в самых разных частях программы. Его передают в качестве спецификатора типа (вместе с ключевым словом struct) формального параметра. Тег может быть опущен. |
| 1. Расположение структуры в памяти:    1. Расположение полей    2. Размер структуры 2. Операции над структурами | Структура расположена в памяти единым дампом. Поля (перечисленные в структуре переменные) расположены чаще всего подряд, в порядке объявления в структуре. Размер структуры равен сумме размеров её полей.  Операции над структурами:   1. Обращение к полю по переменной: **структура.поле** 2. Обращение к полю по указателю на переменную: **структура->поле** 3. Присвоение одной структуре значений другой структуры того же типа. 4. Структуры могут передаваться в функцию в качестве параметра и возвращаться как значения. |
| 1. Структуры и указатели на них:    1. Как параметры функций    2. Как тип результата функции 2. Особенности использования структур:    1. char\*    2. typedef | Структуры, как правило, занимают сами по себе большой объём памяти, и при передаче в функцию в качестве параметра непосредственно структуры уходит много времени и ресурсов. Кроме того, в этом случае поля исходной структуры не могут быть изменены. Поэтому чаще всего гораздо экономичнее и проще передавать указатель на структуру (как параметр) и возвращать указатель на структуру (в качестве результата).  Поля структуры, являющиеся указателями на объекты другого типа, требуют отдельной обработки при копировании структуры (иначе, например, одноимённые поля различных структур укажут на одну и ту же строку (char\*), и редактирование этой строки вызовет фактическую потерю данных и возможные ошибки при высвобождении памяти из-под этой строки).  При этом хранить char\* гораздо эффективнее, чем char[N], так как длина буфера строки может быть изменена и приведена к наиболее выгодному значению, в отличие от постоянного N у массива.  Для удобства программиста и сокращения объёма кода можно словосочетание “**struct %tag%**” заменить на однословное имя через **“typedef struct %tag% new\_name“** |

|  |  |  |  |
| --- | --- | --- | --- |
| 18 | Объединения | 1. Формальное определение 2. Определение переменных типа объединение 3. Тег объединения | Объединение представляет собой одну или несколько переменных (возможно, разного типа), которые объединены под одним именем, при этом каждая переменных занимает одно и то же место в памяти, располагаясь по одному и тому же адресу.  Формат определения: union [tag] {type1 field1; type2 field2; …; typeN fieldN; } [var\_name]; |
| 1. Расположение объединения в памяти:    1. Расположение полей    2. Размер 2. Операции над объединениями | Поля определения располагаются по одному и тому же адресу (в начале дампа). Размер объединения равен наибольшему размеру его поля.  Операции те же, что и над структурами. |
| 1. Области применения объединений | Объединения, объявленные в программах, применяются нечасто. Как правило, в них содержат информацию, которую надо обрабатывать и выдавать как полностью, так и по частям.  НО: регистры процессора **общего назначения**, пожалуй, самый яркий пример объединений. К примеру, регистр EAX эквивалентен объединению  ***union pseudo\_eax {long eax4; short ax2; struct {char al1; char ah1;}parts; };*** |

|  |  |  |  |
| --- | --- | --- | --- |
| 19 | Битовые операции. Битовые поля | 1. Битовые операции:    1. &       * Действие       * Проверка битов       * Установка битов    2. |       * Действие       * Установка битов    3. ^       * Действие       * Смена значений битов    4. ~    5. Сдвиги       * <<       * >> | 1. Битовые операции – логические операции, проводимые над каждым битом фигурирующих операндов. Операнды при этом имеют целый тип и одинаковый размер.    1. & -- поразрядная конъюнкция. Синтаксис: a1 & a2       * На каждом бите проходит конъюнкция/логическое И.       * Проверка битов операнда на установление:         + Формируем mask, в которой на проверяемые биты установлена единица, а на остальные биты – ноль.         + В результате операции op&mask останутся единицы только в значениях проверяемых с помощью mask битов там, где они установлены в op.       * Установка битов операнда в ноль (сброс флагов)         + Формируем mask, в которой нужные биты установлены в ноль, а остальные – в один.         + В результате операции op&mask появятся нули только в устанавливаемых с помощью mask битов op, остальные биты останутся прежними.    2. | -- поразрядная дизъюнкция. Синтаксис: a1 | a2       * На каждом бите происходит дизъюнкция/логическое ИЛИ       * Установка битов операнда в единицу (установка флагов):         + Формируем mask, в которой нужные биты установлены в единицу, а остальные – в ноль.         + В результате операции op|mask появятся единицы только в устанавливаемых с помощью mask битов op, остальные биты останутся прежними.       * Проверка битов операнда на сброс проводится аналогично пункту 1.1.2    3. ^ -- поразрядная симметрическая разность. Синтаксис: a1^a2       * На каждом бите операндов происходит симметрическая разность/сумма по модулю 2/логическое ИСКЛЮЧАЮЩЕЕ ИЛИ/АНТИЭКВИВАЛЕНТНОСТЬ.       * Смена значений битов операнда         + Формируем mask, в которой нужные биты установлены в единицу, а остальные – в ноль         + В результате операции op^mask изменятся значения только в устанавливаемых с помощью mask битов op, остальные биты останутся прежними.    4. ~ -- поразрядная инверсия/логическое НЕ. Синтаксис: ~op       * Каждый бит операнда изменяет своё значение. При этом результат – обратный код числа op.    5. Логические сдвиги числа на некоторое число бит.       * Сдвиг влево. Синтаксис: op<<n\_bit. Размер n\_bit не оговорен стандартом. Действие аналогично оператору **op\*power(2, n\_bit)**, где умножение происходит в **кольце вычетов по модулю power(2, 8\*sizeof(type op))**. Иначе говоря, все биты сдвигаются на n\_bit позиций влево, освободившиеся биты заполнятся нулями, а выдвинутые биты уничтожаются.       * Сдвиг вправо. Синтаксис: op>>n\_bit. Размер n\_bit не оговорен стандартом. Действие аналогично оператору **op/power(2, n\_bit)**, где **деление целочисленное, дробная часть отбрасывается**. Иначе говоря, все биты сдвигаются на n\_bit позиций вправо, освободившиеся биты заполнятся нулями, а выдвинутые биты уничтожаются.       * Логические сдвиги работают значительно быстрее деления или умножения на соответствующую степень двойки, давая тот же результат. |
| 1. Логические операции    1. !    2. &&    3. ||    4. !=, ==, <, <= , >, >= 2. Отличие логических операций от побитовых | 1. Логические операции действуют над всем числом. Операнды имеют целый тип и одинаковый размер. Логическим нулём считается обычный ноль (все биты сброшены), остальные числа – логической единицей.    1. ! – отрицание/логическое НЕ    2. && -- конъюнкция/логическое И    3. || -- дизъюнкция/логическое ИЛИ    4. обычное арифметическое сравнение двух операндов. Типы операндов численные, при этом одинаковые или хотя бы один приводится к другому, либо указатели на одинаковый тип данных. 2. При проведении логических операций все ненулевые числа (как бы) приводятся к числу (-1), представимому всеми единицами в двоичной записи. Поэтому 00100100**&**10010010 == == 00000000 != (некая единица) == 00100100**&&**10010010. |
| 1. Битовые поля    1. Объявление    2. Представление в памяти    3. Операции над битовыми полями | 1. Битовое поле -- особый тип структуры, определяющей, какую длину имеет каждый член. (Определение структуры см. п. 17)    1. Стандартный вид объявления:   **struct имя\_структуры**  **{**  **тип имя1: длина;**  **тип имя2: длина;**  **...**  **тип имяN: длина;**  **};**  Битовые поля должны объявляться как целые, unsigned или signed.   * 1. Представление в памяти: обычным целым числом, размера не менее суммы размеров полей битового поля (контролируется программистом)   2. Необходимые действия над битовыми полями:      + Извлечение поля из структуры включает следующую последовательность действий:        - Конъюнкция с маской битового поля (на битах поля единицы, в остальных местах нули);        - Побитовый сдвиг вправо.      + Сборка одного числа из битовых полей:        - Обнуление числа        - Установка битов поля        - Сдвиг влево      + Замена битового поля        - Обнуление нужных битов с помощью маски        - Заполнение нужных битов (со сдвигом) поразрядной дизъюнкцией   3. Замечание. Битовые поля применяются тогда, когда нужно компактно записать множество разнообразной перечислимой информации, экономя при этом место в памяти, но не гонясь за сверхвысокой производительностью алгоритма вопреки стремлению написать более читабельный код. Если требуется более высокая производительность, применяются обычные числа, при этом отдельно прописываются всевозможные маски и значения перечислимых параметров. |

|  |  |  |  |
| --- | --- | --- | --- |
| 20 и 21 | Препроцессор. Макросы. Особенности макросов с параметрами. | 1. Директивы препроцессора:    1. #include    2. #define (в общих чертах), #undef    3. условные директивы    4. #error, #line    5. #pragma 2. Правила, справедливые для всех директив | #include – включение заголовочного файла  #define – макроопределение #undef – прекращение действия макроопределения  условные директивы: #if, #ifndef, #ifdef и т. п.; #elif; #endif. Обозначают условную компиляцию  #pragma – задаёт модель поведения, зависящую от конкретной реализации компилятора.  #line – изменяет номер текущей строки и имя компилируемого файла.  #error – выдача диагностического сообщения  Правила, справедливые для всех директив:   * Директивы всегда начинаются с символа "#". * Любое количество пробельных символов может разделять лексемы в директиве. * Директива заканчивается на символе '\n'. Перенос строки осуществляется символом ‘\\’. * Директивы могут появляться в любом месте программы. |
| 1. Простые макросы:    1. Синтаксис    2. Применение 2. Макросы с параметрами:    1. Синтаксис    2. Применение 3. Макросы с переменным числом параметров:    1. Синтаксис    2. Применение | 1. Простые макросы    1. #define ИМЯ\_МАКРОСА список замены    2. Применение простых макросов:       * Имена для числовых, символьных или строковых констант.       * Незначительного изменения синтаксиса языка.       * Переименования типов       * Управления условной компиляцией 2. Макросы с параметрами    1. #define ИМЯ\_МАКРОСА(пар1, пар2, …, пар№) список замены.    2. Применяют, чтобы сэкономить время на вызовах коротких функций. Правда, к параметрам в этом случае предъявляются повышенные требования, не отслеживаемые компилятором, за которыми обязан следить программист. 3. Макросы с переменным числом параметров    1. #define ИМЯ\_МАКРОСА(пар1, пар2, …) список замены    2. Аналогично предыдущему пункту 4. |
| 1. Общие свойства макросов:    1. Видимость    2. Переносы строки    3. Встраивание в код программы    4. Скобки в макросах | 1. Все макросы обладают следующими свойствами:    1. Макросы видны до конца файла, в котором объявлены, или до директивы #undef.    2. Переносы строки обозначаются символом ‘\\’, конец макроса – символом переноса строки    3. Макросы встраиваются в код программы простой заменой строк. Контроля за передаваемыми данными не происходит (это ложится на плечи программиста).    4. Заголовки макросов с параметрами следует писать так, чтобы скобки шли вплотную к имени. Кроме того, данные, передаваемые через параметры макроса, следует обрамлять в скобки при обращении к ним, чтобы не было казусов (со знаками, приоритетом операций, указателями и массивами).    5. Макрос может содержать другие макросы.    6. Препроцессор заменяет лишь целые лексемы (но не их части)    7. Макрос не может быть объявлен дважды, если эти объявления не тождественны. |
| 1. Сравнение макросов и функций | Преимущества макросов перед функциями:   * макросы могут работать несколько быстрее (не надо связывать подпрограммы) * макросы более универсальны (например, такой: **#define MAX(a,b) (a)>(b) ? (a) : (b)** )   Недостатки макросов:   * скомпилированный код становится больше, а функция содержится в коде только 1 раз * типы аргументов макроса не проверяются * нельзя объявить указатель на макрос * макрос может вычислять аргументы несколько раз (**v = MAX(f(), g());** // одна из функций будет вычислена дважды) |
| 1. Предопределённые макросы 2. Преобразователи в макросах    1. “#”    2. “##” | 1. В языке Си предопределены макросы, которые не нужно объявлять и нельзя переопределять и отменить (основные):    1. \_\_LINE\_\_ - номер строки, переопределяется только с помощью директивы #line, используется для формирования отладочной информации    2. \_\_FILE\_\_ -- заменяется на имя файла, также переопределяется лишь с помощью #line    3. \_\_DATE\_\_, \_\_TIME\_\_ -- дата и время обработки препроцессором    4. \_\_STDC\_\_ -- если равен 1, то компиляция проводится в соответствии со стандартом Си    5. \_\_STDC\_HOSTED\_\_ -- если равен 1, то программа выполняется под действием операционной системы    6. \_\_VA\_ARGS\_\_ -- макрос, заменяющий собой переменное число параметров функции или макроса    7. \_\_func\_\_ -- имя функции как строки (только в GCC, С99) 2. В списке замены макроса могут находиться следующие операции преобразования:    1. “#” – конвертирует аргумент макроса в строковый литерал (префиксный, 1 операнд)    2. “##” – объединяет две лексемы в одну (инфиксный, 2 операнда) |

|  |  |  |  |
| --- | --- | --- | --- |
| 22 | Динамически расширяемые массивы. | 1. Объявление 2. Перевыделение памяти с помощью realloc из <stdlib.h> 3. Изменение размера динамического массива    1. Проблема эффективности перевыделения памяти    2. Особенности добавления и удаления элементов | 1. Динамические массивы с регулируемой длины принято объявлять структурой вида **struct dyn\_array {int len; int allocated; int step; type\* data;}**. 2. Функция realloc(void\* ptr, size\_t bytes) перевыделяет память под объект ptr. Выделяют следующие случаи:    1. ptr==NULL && bytes>0. Происходит обычное выделение памяти, как при malloc(bytes).    2. ptr!=NULL && bytes==0. Происходит освобождение памяти, как при free(ptr).    3. ptr!=NULL && bytes!=0. Основной случай при вызове realloc. В худшем случае выделяется bytes байтов, копируются имеющиеся значения байтов из старой области памяти в новую, освобождается старая память. В лучшем случае, когда соседние справа байты свободны в достаточном количестве или bytes не больше текущего размера выделенной области, перемещений не происходит. 3. Поскольку для больших областей памяти сложность realloc порядка bytes, то изменение размера динамического массива при каждом добавлении нового элемента имеет порядок size^2, в то время как заполнение статического массива имеет порядок size. Для уменьшения сложности заполнения большого динамического массива используют **экспоненциальное** (геометрическое) увеличение длины области памяти под массив (например: 2,4,8,16,…) или **арифметическое** (10, 20, 30, 40, …). Сложность арифметического увеличения длины ~size^2 (правда, нормирующий коэффициент ~1/step), сложность экспоненциального увеличения длины оценивается ~size.   После удаления элемента обычно не сокращают длину массива, понимая, что возможно новое пополнение.  Также для добавлений и удалений элементов характерны следующие особенности:   * поскольку адрес массива может измениться, обращаться к элементам массива следует только по индексам. * Благодаря маленькому начальному размеру массива программа сразу же «проверяет» код, реализующий выделение памяти. |
| 1. Коды операций над массивами 2. Плюсы и минусы динамических массивов | Добавление элемента:  int append(struct dyn\_array \*d, int item)  {  if (!d->data)  {  d->data = malloc(INIT\_SIZE \* sizeof(int));  if (!d->data)  return -1;  d->allocated = INIT\_SIZE;  }  else  if (d->len >= d->allocated)  {  int \*tmp = realloc(d->data,  d->allocated \* d->step \* sizeof(int));  if (!tmp)  return -1;  d->data = tmp;  d->allocated \*= d->step;  }  d->data[d->len] = item;  d->len++;  return 0;  }  Удаление элемента:  int delete(struct dyn\_array \*d, int index)  {  if (index < 0 || index >= d->len)  return -1;  memmove(d->data + index, d->data + index + 1,  (d->len - index - 1) \* sizeof(int));  d->len--;  return 0;  }  Плюсы и минусы динамических массивов:   * Простота использования. * Константное время доступа к любому элементу. * Не тратят лишние ресурсы. * Хорошо сочетаются с двоичным поиском. * Хранение **меняющегося набора значений** |

|  |  |  |  |
| --- | --- | --- | --- |
| 23 | Линейный односвязный список. | 1. Области применения 2. Сравнение с массивами | По сравнению с массивами, списки:   * не фиксированы в размере в каждый момент времени; * легко переформировать, изменяя всего несколько указателей; * позволяют при вставке нового элемента и удалении некоторого элемента не изменять адреса остальных элементов; * занимают больше места, так как каждый элемент списка должен содержать указатель на следующий элемент).   Односвязные списки имеют широкое применение в вычислительных алгоритмах и архитектурах баз данных. |
| 1. Примеры реализации    1. Стекоподобный    2. Очередеподобный 2. Основные операции    1. Создание списка    2. Добавление элемента       * В голову       * В хвост       * В любое место    3. Удаление элемента       * Из головы       * Из хвоста       * Из любого места    4. Очистка списка | Элемент списка представляет собой структуру вида  **struct list\_el { поле1; [остальные поля;] list\_el\* next; }**  Для последнего элемента списка next должен равняться NULL (иначе список будет кольцевым). В программе обязательно хранить адрес головы. Рассмотрим сложности стандартных алгоритмов работы со списком (напиши их самостоятельно):   |  |  |  | | --- | --- | --- | | Операция | Стекоподобный список (без хранения указателя на последний элемент списка) -- сложность | Очередеподобный список (с хранением указателя на последний элемент списка) -- сложность | | Создание | O(1) | O(1) | | Добавление в начало | O(1) | O(1) | | Добавление в конец | O(N) | O(1) | | Добавление в середину | O(N) | O(N) | | Удаление из начала | O(1) | O(1) | | Удаление из любого места | O(N) | O(N) | | Очистка списка | O(N) | O(N) | |

|  |  |  |  |
| --- | --- | --- | --- |
| 24 | Списки в стиле Беркли | 1. Кольцевые двусвязные списки    1. Организация    2. Операции над списками       * Создание       * Добавление элемента       * Удаление элемента       * Очистка    3. Сравнение с односвязными списками 2. Особенности списков в стиле Беркли 3. container\_of 4. offset\_of | 1. Элемент списка представляет собой структуру вида   **struct list\_el { поле1; [остальные поля;] list\_el\* next; list\_el\* prev; }**  Для оптимизации работы над списками их делают кольцевыми, то есть **head->prev == last && last->next == head**.  Поскольку двусвязный кольцевой список не имеет настоящих начала и конца, операции создания списка, добавления и удаления элементов имеют сложность О(1), а очистка списка, как обычно, O(N).  Правда, для этого удобства требуется гораздо больше места в памяти, чем для линейного односвязного списка, но зато обработка двусвязного списка намного проще и просмотр можно делать в любую сторону и по любой траектории.  Кроме того, для универсализации функций работы со спсиками данные кладут в отдельную структуру, а в сегменте данных элемента списка кладут указатель на структуру с данными. В прототипах фукнций фигурирует тип void\* и размер структуры данных.   1. Стиль Беркли для двусвязных списков таков:   **struct berkley\_el { поле1; [остальные поля;] struct links\_t{berkley\_el\* next; berkley\_el\* next;} links; }**   1. container\_of, offset\_of – некие кривые GNUтые макросы (КТО ЗНАЕТ, КАКИЕ?) |

|  |  |  |  |
| --- | --- | --- | --- |
| 25 | Сложные объявления | 1. Правила чтений сложных объявлений 2. Недопустимые конструкции | 1. Правила чтения сложных объявлений:    1. Основные конструкции объявлений:       * type [] – массив типа type       * type [N] – массив N элементов типа type       * type (type1, type2, …) – функция, принимающая параметры типа type1, type2, …, и возвращающая результат типа type       * type\* -- указатель на type    2. Приоритеты операций:       * **Скобки всегда предпочтительнее звёздочек**, т.е. выражение ***char\*\* []*** следует читать как «адрес массива указателей на указатель на char», а не «указатель на указатель на адрес массива типа char».       * Круглые скобки всегда предпочтительнее квадратных       * Круглые скобки часто ставят для изменения приоритета операций (в частности, когда среди параметров функции фигурирует указатель на функцию ***int(\*func)(void\*),*** так как ***int \*func(void\*)*** означает «функция, …, возвращающая указатель на int» -- то есть совсем не «указатель на функцию, …, возвращающую int»!    3. Чтение идёт «от самого глубокого идентификатора» «изнутри наружу», то есть из центра скобочной последовательности к краям, стирая по пути дешифрования объявления пары скобок. 2. Недопустимые конструкции в объявлениях:    1. Массив функций ***~~int a[10](int)~~*** (именно функций, массив *указателей* на функции допустим)    2. Функция не может вернуть функцию: ***~~int g(int)(int)~~***    3. Функция не может вернуть массив: ***~~int f(int)[]~~***    4. В массиве только левая лексема может быть пустой (или переменной)    5. Тип void может применяться только вкупе с указателем (звёздочкой) или как результат работы функции. Не допускаются объявления типа ***~~void x;~~ ~~void x[5];~~***   int \*(\*x[10])(void);  х - массив из десяти элементов типа указатель на функцию, которая не принимает параметров, возвращающую указатель на int  char \*(\*(\*\*f[][8])())[];  f - массив типа массив из 8 элементов типа указатель на указатель на функцию без параметров, возвращающую указатель на массив типа указатель на char  void (\*signal(int, void (\*fp)(int)))(int);  fp - указатель на функцию, принимающую параметр типа int и ничего не возвращающую  signal - функция, принимающая параметры типа int и fp, и возвращающая указатель на функцию, принимающую параметр типа int и ничего не возвращающую. |

|  |  |  |  |
| --- | --- | --- | --- |
| 26 | Преобразование типов. Явное и неявное преобразование. Обычное арифметическое преобразование. | 1. Преобразование типов:    1. Определение понятия    2. Явное преобразование: допустимые переводы    3. Неявное преобразование: реализуемые переводы, недопустимые переводы 2. Обычное арифметическое преобразование: схема | * 1. Преобразование типов – приведение значения переменной от одного типа к другому.   2. Явное преобразование типов: { ***type1 i*; … *j = (type2) i*;** }. Преобразование допустимо для любых численных типов: как целых, так и вещественных – в обе стороны. Кроме того, обязательно явное преобразование типов в случае обработки битовых массивов данных, заданных указателями на разные типы данных или указателями void\*. Допускается также преобразование из целого типа разрядности, совпадающей с разрядностью процессора, в указательный, и наоборот.   3. При возникновении необходимости привести тип к требуемому, интерфейсом функции (либо при операции над различными типами численных данных) совершается неявное преобразование типов. Совершаемые при этом переходы могут быть от вещественного числа к целому, но гораздо чаще наоборот, а также от одного целого типа к другому целому, и от одного вещественного типа к другому вещественному. Переходить от целого типа к указательному и наоборот запрещено (выдастся ошибка компилятора).   Примеры неявных преобразований типов:   * + - Присвоение:   { int i = 4; double d = 3.6416; i = d; /\* i==(int)d == 3 \*/ }  { int i = 4; double d = 3.6416; d = i; /\* d==(double)i == 4.0 \*/ }   * + - Арифметические операции   {int i = 5, j = 2; double r = i/j; /\*здесь 2.0 \*/ r = (double)i/j; /\* а здесь 2.5 \*/}.   * + - Вызовы функций   {double s = sqrt(2); /\* фактически s=sqrt( (double) 2) \*/ }   1. Правила обычного арифметического преобразования типов:    1. При вычислении целочисленных выражений от одно- и двухбайтовых целых операндов последние сводятся к *int*, если последний вмещает в себя *более 2 байтов*, и unsigned int в противном случае.    2. При вычислении вещественных выражений операнды сводятся к наиболее длинному и точному типу (float 🡺 double 🡺 long lobule).    3. Если происходит операция над вещественным и целым операндом, то целый операнд приводится к double, а затем пользуемся правилом 2.2.    4. Если происходит операция над целыми операндами одинакового размера не менее 4 байтов:   (signed+unsigned)=={signed, если значение беззнакового операнда представимо в знаковом типе; unsigned иначе}   * 1. Если операнды 4- и 8-байтовые, 4-байтовый сводится к 8-байтовому.   Таким образом, целочисленные преобразования таковы:  (char, unsigned char, short, unsigned short 🡺 int ⬄ unsigned int 🡺 long long ⬄ unsigned long long) |

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 27 | Библиотеки | 1. Статические и динамические библиотеки: плюсы и минусы 2. Статические библиотеки:    1. Сборка    2. Компоновка с ними 3. Динамические библиотеки:    1. Сборка    2. Динамическая компоновка    3. Динамическая загрузка 4. Динамическая библиотека и приложение на другом языке | 1. Библиотека состоит из заголовочного файла и бинарного файла с машинным кодом. Поэтому исходный код библиотеки недоступен.   Статические библиотеки – компонуются в исполняемый файл  Динамические библиотеки – загружаются из исполняемого файла в процессе работы.  Плюсы и минусы типов библиотек:   |  |  |  | | --- | --- | --- | | ТИП | + | - | | Статические | * Исполняемый файл включает в себя все необходимое. * Не возникает проблем с использованием не той версии библиотеки. | * + «Размер».   + При обновлении библиотеки программу нужно пересобрать. | | Динамические | * 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. |  1. Сборка статической библиотеки:    * компиляция   **gcc -std=c99 -Wall -Werror -c arr\_lib.c**   * + упаковка   **ar rc libarr.a arr\_lib.o**   * + индексирование   **ranlib libarr.a**  Сборка приложения со статической библиотекой  **gcc -std=c99 -Wall -Werror main.c libarr.a -o test.exe**  или  **gcc -std=c99 -Wall -Werror main.c -L. -larr -o test.exe**   1. Сборка динамической библиотеки:    * компиляция   **gcc -std=c99 -Wall -Werror -c arr\_lib.c**   * + компоновка   **gcc -shared arr\_lib.o -Wl,--subsystem,windows -o arr.dll**  Сборка приложения с динамической библиотекой (динам. компоновка):  **gcc -std=c99 -Wall -Werror -c main.c**  **gcc main.o -L. –larr -o test.exe**  Сборка приложения с динамической библиотекой (динам. загрузка):  **gcc -std=c99 -Wall -Werror main.c –o test.exe**  Windows API для работы с динам. библиотеками содежит следующие функции модуля **windows.h:**   * **HMODULE LoadLibrary(LPCTSTR)** * **FARPROC GetProcAddress(HMODULE, LPCSTR)** * **FreeLibrary(HMODULE)**  1. Оформление заголовочного файла как модуля Дельфи происходит с аккуратным конвертированием типов, подключением с директивой {$L lib\_name}, соответствующим оформлением. При этом необходимо помнить, что менеджеры куч у Си и Дельфи разные, а стандарт выделения и освобождения памяти должен быть одинаковым. |

|  |  |  |  |
| --- | --- | --- | --- |
| 28 | Неопределённое поведение | 1. Побочные эффекты вычисления выражений 2. Точки следования 3. Undefined behavior 4. Unspecified behavior | 1. При вычислении выражений возможны следующие побочные эффекты:    1. Инкремент/декремент некоего операнда, входящего в состав выражения;    2. Изменение фактических параметров, передаваемых в функцию по указателю;    3. Вывод отладочной/листинговой информации и др. 2. Точка следования – любая точка программы, в которой **гарантируется**, что все побочные эффекты **предыдущих** выражений уже **проявились**, а побочные эффекты **последующих** ещё **отсутствуют**. Стандартом С99 определены следующие точки следования:    1. В конце всего выражения:       1. присвоение (a=b);       2. управляющие выражения в скобках оператора ветвления (**if**) или выбора (**switch**), циклов **while** и **do…while**, все 3 оператора в скобках цикла **for**:       3. инструкция **return**.    2. В объявлении с инициализацией на момент завершения вычисления инициализирующего значения.    3. Между вычислением первого и второго или третьего операндов тернарной условной операции.    4. Перед **входом** в вызываемую функцию.    5. Между вычислением левого и правого операндов логических операций && и ||, а также операции-запятой. 3. Неопределённое поведение (undefined behavior) – свойство языка Си и программных библиотек этого языка в определённых **маргинальных** ситуациях (не требующих обработки по спецификации) выдавать результат, зависящий от реализации компилятора (библиотеки, микросхемы) и случайных параметров наподобие состояния памяти и сработавшего прерывания. Это означает буквально следующее: «При условии А результат операции Б не определён».   Допускать такое поведение считается ошибочным. На некоторых платформах и после компиляции на некоторых компиляторах возможен сбой работы программы.  Примеры неопределённого поведения:   * 1. Использование переменной до инициализации;   2. Большинство функций не проверяют указатель NULL;   3. Повторный инкремент операнда в одном операторе:   {int i=5; i = ++i + ++i; } – возможны значения 13 или 14, так как побочные эффекты (инкремент) могут быть применены в любой момент между 2-мя точками следования.   1. Неуточняемое поведение (unspecified behavior) – поведение программы, зависящее от компилятора, который выбирает один из допустимых стандартом реализаций некоторой языковой конструкции. Это означает буквально следующее: «При условии А результат операции Б может принимать одно из допустимых значений Б1, Б2, …, Бн, при этом выбор результата произволен».   Программисту рекомендуется не допускать случаев неуточняемого поведения, если это важно с точки зрения алгоритма работы программы.  Примеры неуточняемого поведения:   * 1. Пусть есть вызов **t = fun(f(), g());** . Порядок вычисления значений функций f() и g() не определён стандартом, поэтому порядок вывода отладочной информации неизвестен программисту заранее и зависит от компилятора.   2. Определение спецификаций целых типов полностью зависит от компилятора и разрядности платформы. Тип **long int** может иметь размер как 4, так и 8 байтов. Таким образом, на одном компиляторе преобразование long 🡺 type\* может выполниться успешно, а на другом может завершиться с ошибкой. |

|  |  |  |  |
| --- | --- | --- | --- |
| 29 | Тернарная операция, операция запятая | 1. Тернарная операция    1. Синтаксис    2. Сравнение с условной конструкцией 2. Операция запятая    1. Синтаксис    2. Границы применения    3. Предупреждения компилятора | 1. Тернарная (условная) операция – условное присвоение в зависимости от результата первого операнда    1. Синтаксис: **r = log\_op ? val1 : val2**   Если **log\_op истинно**, то **r = val1**, иначе **r = val2**.   * 1. Достигается некоторая выгода за счёт небольшой экономии в объёме скомпилированного кода (на языке ассемблера загрузка результата произойдёт вне ветвления в случае тернарной условной операции)  1. Запятая применяется в двух принципиально разных случаях: как перечисление параметров (в объявлениях или определениях макросов или функций – не является операцией) и как операция разделения выражений (в этом случае обрабатываются все выражения из перечня, сохраняется результат только последнего).    1. Синтаксис тривиален: **op1, op2, op3, /\* и так далее \*/, opN** (N>=2). Тип результата определяется типом последнего выражения.    2. Применяется операция запятая редко, чаще всего при отладке перебором (несколько разных условий выхода из цикла, требуется выбрать наиболее удачное),    3. При отладке возможны предупреждения компилятора о том, что ожидается 1 оператор, а не несколько, разделённых запятой. Кроме того, возможно несоответствие типов при неудачном изменении выражения, содержащего операцию запятая.    4. Важно! Операция запятая имеет самый низкий приоритет среди всех операций языка Си. |

|  |  |  |  |
| --- | --- | --- | --- |
| 30 | Типы. Концепция типа данных. Классификация типов. Спецификаторы. Логический тип. | 1. Концепция типа данных 2. Классификация типов    1. Простые (скалярные)    2. Сложные       1. Тензорные (массивы)       2. Структурные       3. Объединения       4. Множества и строки 3. Спецификаторы типов 4. Стратегия вычисления логических выражений | Тип данных определяет множество значений, набор операций, которые можно применять к таким значениям, и, возможно, способ реализации хранения значений и выполнения операций. Любые данные, которыми оперируют программы, относятся к определённым типам.   1. Типы данных:    1. Скалярные (простые)       * Перечислимый тип       * Целочисленные/символьные: {unsigned, signed}x{char(1B), short int(2B), [long] int(4B), long long int(8B)}       * Вещественные c плавающей запятой: {float (4B), double (8B), long double (12B)}       * Указатели и ссылки (т.е. неизменяемые указатели), размер равен разрядности процессора.    2. Сложные (составные)       * Массивы (статические): “type **[]**”, обязательно указание всех размерностей, кроме, возможно, самой левой. Идентификатор массива можно считать ссылкой и использовать в адресной арифметике без переопределения.       * Структуры. “**struct {fields}**”, обязательно хотя бы 1 поле. Записываются в память блоками.       * Объединения “**union {fields}**”, обязательно хотя бы 1 поле.       * Специального типа для множеств и строк в языке Си не существует. 2. Спецификаторы типов: int, char, float, double, union, void, struct. Модификаторы типов: short, long, long long (для чисел); far, near, huge (для указателей). Спецификатор доступа к данным: const 3. Логические выражения считаются следующим образом:    1. Формируется структура логического выражения согласно приоритетам операций и расставленным скобкам.    2. Если на некотором этапе вычисления становится ясно, что дальнейшие вычисления бесполезны (например, **0 && anything == 0** или **1 || anything == 1**), то вычисление выражения прерывается досрочно и выдаётся уже готовый верный результат. |

|  |  |  |  |
| --- | --- | --- | --- |
| 31 | Стандарты языка Си. Основные концепции языка Си. | 1. История языка Си    1. Появление    2. Стандарты:       * 1978 год (K&R)       * 1989 год (ANSI, C89/C90)       * 1999 год (C99)       * 2011 год (C11) 2. Некоторые различия между стандартами C89 и C99 3. Основные концепции языка Си | * 1. Язык Си появился в 1972-1973 годах ввиду потребности в создании языка, на котором можно было бы написать операционную систему UNIX весьма компактным кодом, который легко компилируется и обладает высоким быстродействием. Пока ещё распространены UNIX-подобные системы, язык Си будет жив.   2. В своей книге Керниган и Ритчи, создатели языка Си, в 1978 году опубликовали неформальные рекомендации для создателей компиляторов Си, чтобы один и тот же код смог компилироваться одинаково на различных компиляторах.   К середине 1980-х годов расхождения становятся очень серьёзными, и институт ANSI в 1989 году готовит первый формальный и документированный стандарт для создателей компиляторов и стандартных библиотек. Неформальное название стандарта – С89. Небольшие правки в этот стандарт годом позже (в 1990 году) внесла ISO (=>C90).  Программистам было крайне неудобно соответствовать существенно различавшимся стандартам С89/С90, с одной стороны, и стандартами для Си-подобных языков, с другой. Для приведения к единообразию ISO и ANSI в 1999-2000 годах разработали и внедрили стандарт С99.  Множество небезопасных функций и необходимость расширенной поддержки символов Unicode побудили разработать в 2011 году новый стандарт С11, которому, правда, не соответствует ещё б**о**льшая часть современных компиляторов.   1. По сравнению со стандартом С89 у С99:    1. появились встраиваемые (inline) функции;    2. локальные переменные можно объявлять в любом месте текста (как в C++);    3. появились массивы переменной длины;    4. появилась именованная инициализация структур;    5. закреплена поддержка однострочных комментариев;    6. несколько новых библиотечных функций и заголовочных файлов;    7. полное отсутствие типа, означавшее неявное задание типа **int**, не поддерживается. 2. Основные концепции языка Си:    1. Си - язык "низкого" уровня. Строго говоря, уровень Си высокий, но синтаксис весьма сильно приближён к ассемблеру, что позволяет сокращать время выполнения программы.    2. Си - "маленький" язык c однопроходным компилятором. Компилятор для увеличения производительности проходит по коду один раз. Поэтому все объекты, используемые в программе. должны быть предварительно объявлены.    3. Си предполагает, что программист знает, что делает. Некоторые функции, в том числе стандартные, не учитывают негативные эффекты вроде выхода за пределы выделенной памяти или границы массива, поэтому результат моет быть непредсказуемым. Поэтому программист сам должен учитывать негативные эффекты использования функций и не допускать эти эффекты. |

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | Справочные материалы | 1. Приоритеты операций языка Си 2. Стандартные библиотеки и часто употребляемые функции из них | 1. Таблица приоритетов операций языка СИ  |  |  |  |  |  | | --- | --- | --- | --- | --- | | **Лексемы** | **Операция** | **Класс** | **Приоритет** | **Ассоциативность** | | имена, литералы | простые лексемы | первичный | 16 | нет | | a[k] | индексы | постфиксный | 16 | слева направо | | f(…) | вызов функции | постфиксный | 16 | слева направо | | . | прямой выбор | постфиксный | 16 | слева направо | | -> | опосредованный выбор | постфиксный | 16 | слева направо | | (имя типа) {init} | составной литерал (C99) | постфиксный | 15 | справа налево | | ++ -- | положительное и отрицательное приращение | постфиксный | 15 | справа налево | | sizeof | размер | унарный | 15 | справа налево | | ~ | побитовое НЕ | унарный | 15 | справа налево | | ! | логическое НЕ | унарный | 15 | справа налево | | - + | изменение знака, плюс | унарный | 15 | справа налево | | & | получение адреса | унарный | 15 | справа налево | | \* | разыменование указателя | унарный | 15 | справа налево | | (имя типа) | приведение типа | унарный | 15 | справа налево | | \* / % | мультипликативные операции | бинарный | 13 | слева направо | | + - | аддитивные операции | бинарный | 12 | слева направо | | << >> | сдвиг влево и вправо | бинарный | 11 | слева направо | | < > <= >= | отношения | бинарный | 10 | слева направо | | == != | равенство/  неравенство | бинарный | 9 | слева направо | | & | побитовое И | бинарный | 8 | слева направо | | ^ | побитовое исключающее ИЛИ | бинарный | 7 | слева направо | | | | побитовое ИЛИ | бинарный | 6 | слева направо | | && | логическое И | бинарный | 5 | слева направо | | || | логическое ИЛИ | бинарный | 4 | слева направо | | ? : | условие | тернарный | 3 | справа налево | | = += -= \*= /= %= <<= >>= &= ^= |= | присваивание | бинарный | 2 | справа налево | | , | последовательное вычисление | бинарный | 1 | слева направо |  1. Список стандартных библиотек и часто употребляемых функций из них  |  |  | | --- | --- | | Заголовочный файл | Функции | | stdio.h | printf и её семейство  scanf и её семейство  puts, fputs  gets, fgets  getc  getchar  fopen  rewind  fclose  feof  fseek  EOF (макрос)  fread  fwrite | | stdlib.h | malloc  realloc  calloc  free  qsort, bsearch  strto\*, ato\*  rand  srand | | stdarg.h | va\_start  va\_end  va\_copy | | string.h | memmove, memset,  strcat, strncat  strchr, strrchr, strpbrk, strspn, strcspn  strcmp, strncmp  strstr  strlen  strdup, strndup | | assert.h | assert | | math.h | abs, fabs, fdim  acos, asin, atan, atan2  cos, sin, tan  ceil, floor  fmod, frexp, ldexp  exp, exp2, expm1  log, log10, log1p, logb  pow  random  sqrt, cbrt, hypot  acosh, asinh, atanh  fmax, fmin  lgamma, tgamma | |