|  |  |  |  |
| --- | --- | --- | --- |
| 1. **Современные подходы к разработке ПО: понятие конструирования ПО, основные этапы конструирования ПО.**  Конструирование ПО - дисциплина программной инженерии. Это детальное создание работающего ПО посредством комбинации кодирования, верификации, модульного тестирования, интеграционного тестирования и отладки.  Конструирование – единственный процесс, кот. выполняется всегда – это процесс создания какого-нибудь объекта, может включать в себя некоторые аспекты планирования, проектирования и тестирования.    Результат конструирования – исходный код  После конструирования должно быть выполнено исчерпывающее, статистически контролируемое тестирование системы (может отсутствовать). Это главный этап разработки ПО, без которого не обходится ни один проект.  Основные этапы конструирования: детальное проектирование, кодирование, отладка, интеграция и тестирование приложения разработчиками (блочное и интеграционное тестирование).  Конструирование часто называют «кодированием» и «программированием».  От качества конструирования во многом зависит качество ПО.  Компетентность в конструировании ПО определяет то, насколько хорошим программистом вы являетесь.  Основные решения, которые принимаются при конструировании: выбор языка программирования; конвенции программирования; выбор технологий; выбор основных методик конструирования. | 2. **Современные подходы к разработке ПО: понятие конструирования ПО; результат конструирования; решения, которые принимаются при конструировании.**  Конструирование – часть процесса разработки ПО. В зависимости от размера проекта на конструирование обычно уходит 30–80% общего времени работы.  Конструирование занимает центральное место в процессе разработки ПО. Требования к приложению и его архитектура разрабатываются до этапа конструирования, что гарантирует его эффективность. Тестирование системы выполняется после конструирования и служит для проверки его правильности.  Результат конструирования – исходный код – часто является единственным верным описанием программы.   * конструирование – главный этап разработки ПО, без которого не обходится ни один проект; * осн. этапы конструирования: детальное проектирование, кодирование, отладка, интеграция и тестирование приложения разработчиками (блочное тестирование и интеграционное тестирование); * конструирование часто называют «кодированием» или «программированием»; * от кач-ва конструирования во многом зависит кач-во ПО; * компетентность в конструировании ПО определяет то, насколько хорошим программистом вы являетесь. | 3. **Система прог-ия: определение, состав, назначение. Классический жизненный цикл разработки ПО. Текстовый редактор. Интегрированная среда разработки. Прогр-ый продукт. Общ. схема преобраз-ия исх. кода в процесс ОС.**  Определение: комплекс программных средств для реализации ПО (написания и отладки программного кода).  Состав: **трансляторы** - преобразуют исх. код на одном языке программирования в исх. код на др. языке, **компоновщики** - принимают один или несколько объектных модулей и формируют на их основе загрузочный модуль, **отладчики** - позволяют контрол. ход вып-ия проги (приостанавливать, выполнять пошагово), просматривать и изменять области памяти, **профилировщики** - позволяют оптимиз. код программы (устранять утечки памяти, оптимиз. циклы, анализ. производительность), **программные библиотеки**, **редакторы кода**, **системы поддержки версий**.  Назначение: автомат-ия составления и отладки прог юзера.  Жизненный цикл ПО — период времени, который начинается с момента принятия решения о необходимости создания прогр. продукта и заканчивается в момент его полного изъятия из эксплуатации. Этот цикл — процесс построения и развития ПО.    Текстовый редактор - программа, позволяющая подготовить исходный код программы.  Интегрированная система разработки: набор инструментов для разработки и отладки программ, имеющий общую интерактивную графическую оболочку, поддерживающую выполнение всех основных функций жизненного цикла разработки программы. Состав: редакторы кода, транслятор, компоновщик, отладчик, система поддержки версий. Примеры: Visual Studio, NetBeans, Eclipse, Embarcadero Delphi.  Программный продукт: программа, работающая без авторского присутствия; исполняется, тестируется, конфигурируется без присутствия автора; сопровождается документацией.  Общая схема преобразования исходного кода в процесс операционной системы:    Транслятор | 4. **Интегрированная среда разработки: определение, состав, назначение. Транслятор: виды трансляторов. Транслятор: определение, назначение, примеры. Интерпретатор: определение, назначение, примеры.**  Интегрированная среда разработки: – набор инструментов для разработки и отладки программ, имеющий общую интерактивную графическую оболочку, поддерживающую выполнение всех основных функций жизненного цикла разработки программы. Примеры: Eclipse, Microsoft Visual Studio, NetBeans, Qt Creator.  Интегрированная среда разработки используется программистами для разработки, создания и тестирования программ или программного обеспечения.  Под отладкой программы понимается процесс испытания работы программы и исправления обнаруженных при этом ошибок. Обнаружить ошибки, связанные с нарушением правил записи программы на языке программирования (синтаксические и семантические ошибки), помогает используемая система программирования.    **Транслятор** — программа или техническое средство, выполняющее трансляцию программы. Трансляция программы — преобразование программы, представленной на одном из языков программирования, в программу на другом языке.  **Интерпретатор** – разновидность транслятора. Переводит и выполняет программу с языка высокого уровня в машинный код строка за строкой.  Схема работы транслятора: Исходный код на одном языке программирования 🡪 Компилятор (транслятор) 🡪 Исходный код на другом языке программирования или код целевой машины 🡪 Результат выполнения программы 🡨 ИСПОЛНЯЕМАЯ ПРОГРАММА 🡨 Данные |
| 5. **Язык прогр-ия: определение, назначение, примеры. Исх. код. Язык ассемблера. Транслятор: определение, назначение, примеры. Алфавит языка. Ассемблер. Объектный код, объект. модуль. Общ. схема преобразования исх. кода в процесс ОС.**  ЯП - формальная знаковая система, предназначенная для записи компьютерных программ. Знаковая система определяет набор лексических, синтаксических и семантических правил написания программы. ЯП представляется в виде набора спецификаций, опред. его синтаксис и семантику. Примеры: процедурно-ориентированные, объектно-ориентированные, декларативные, операторные, функциональные, скриптовые, проблемно-ориентированные, машинно-зависимые (ассемблеры) и т.п.  Исх. код (исходная программа): текст программы, написанный на языке программирования.  Программа на исх. языке (исх. код) готовится с помощью текстовых редакторов (программ, позволяющих подготовить исходный код программы) и в виде текстового файла или раздела библиотеки поступает на вход транслятора.  Язык ассемблера: машинно-ориентированный язык прогр-ия (для конкретной архитектуры компьютера, команды которого соответствуют машинным командам)  Транслятор: программа, преобразующая исходный код на одном языке программирования в исх. код на другом языке.  Трансляторы (компиляторы) полностью обрабатывают весь исх. текст программы (исх. код). Они просматривают его в поисках синтаксических ошибок (иногда несколько раз), проверяют семантические правила языка.  Интерпретатор - разновидность транслятора. Переводит и выполняет программу с языка высокого уровня в машинный код строка за строкой. Примеры: диалоговый, синтаксически-ориентированный, однопроходной, многопроходной, оптимизирующий, тестовый, обратный  Алфавит языка: набор символов, разрешенных к использованию языком. Основывается на одной из кодировок.  Ассемблер: транслятор с исходного кода на языке ассемблера в программу на машинном языке (язык, который может интерпретироваться процессором).  Объектный код - результат работы транслятора. Один файл объектного кода – объектный модуль.  Общая схема преобразования исходного кода в процесс операционной системы: | 6. **Компоновщик: определение, назначение. Загрузочный код. Загрузчик: определение, назначение. Общая схема преобразования исх. кода в процесс ОС.**  Компоновщик:  Определение: программа, принимающая один или несколько объектных модулей и формирующая на их основе загрузочный модуль.  Назначение: если программа собирается из нескольких объектных файлов, компоновщик может собирать эти файлы в единый исполнимый модуль, вычисляя и подставляя адреса вместо символов, в течение времени компоновки (статическая компоновка) или во время исполнения (динамическая компоновка).  Загрузочный код: результат работы компоновщика. Один файл загрузочного кода – загрузочный модуль.  Загрузчик - программа, обычно входящая в состав операционной системы, предназначенная для запуска процесса операционной системы на основе загрузочного модуля. Назначение: отвечает за загрузку исполнимых файлов и запуск соответствующих новых процессов.  Общая схема преобразования исходного кода в процесс операционной системы: | 7. **Иерархическая структура компонентов в Visual C++: глобальный контейнер, компоненты, входящие в его состав. Структура проекта Visual C++.**  Иерархическая структура компонентов в Visual C++.  Глобальный контейнером (компонент, включающий в себя другие компоненты) является Решение. Решение может содержать один или несколько проектов.  Проекты являются независимыми компонентами. Они имеют собственную структуру, состоящую из четырех основных каталогов:  Внешние зависимости – содержит ссылки на все модули, которые использует программа.  Файлы заголовков – содержит файлы кода С++ с расширением h.  Исходные файлы – содержит файлы кода С++ с расширением срр.  Файлы ресурсов – содержит файлы, непосредственно не относящиеся к языку С++, но необходимые для работы приложения. Например, мультимедийные файлы.  Код программного проекта может иметь сложную структуру и состоять из нескольких файлов исходного кода и конфигурационных файлов. | 8. **Иерархическая структура компонентов в Visual C++: многофайловый проект,** **предвар. откомпил. заголовки: назначение, создание, применения. Пример (C++).**  Страницы свойств проекта. Разделы свойств. Параметры.  Общие – ключ Уровень предупреждений позволяет отключить все предупреждения (/W0), либо ужесточить уровень проверок и считать все предупреждения ошибками (/Wall).  Предварительно откомпилированные заголовки позволяют их включить/отключить, определить имя создаваемого предварительно откомпилированного заголовочного файла и местоположение для полученного предварительно откомпилированные заголовки выходного файла (с расширением pch). Также можно настроить имена и папки, в которых будут размещаться различные Выходные файлы.  Командная строка компилятора С++ отображает, с какими параметрами (ключами) выполняется текущая компиляция.  Раздел Компоновщик отображает и позволяет изменить текущие настройки и ключи компоновки.  Ключи компилятора: /c – компиляция без компоновки; /EHsc – модель обработки исключений (перехватываются исключения С++). Ключи компоновщика: /out – указывает имя загрузочного файла.  Предкомпилированный заголовок компилируется только в том случае, если он или какие-либо содержащиеся в нем файлы были изменены.  При создании нового проекта надо указывать опцию "Использовать предкомпилированный заголовок." В настройках(свойствах) проекта необходимо указывать ключ (/Yu - Использовать) во вкладке свойства конфигурации - C/С++ - предварительно откомпилированные заголовки. Сам файл .pch будет создан автоматически.  Для того чтобы предкомпилированный заголовок правильно функционировал необходимо чтобы в "обозревателе решений" во вкладке "Заголовочные файлы". находился stdafx.h, а во вкладке "Файлы исходного кода" stdafx.cpp. Система сама автоматически их там создаёт.  В файле stdafx.cpp должно быть включение файла #include "stdafx.h"  При создании проекта в Visual Studio в pch.h проект добавляется предварительно скомпилированный файл заголовка. Цель файла — ускорить процесс сборки. Предкомпилированный заголовок компилируется только в том случае, если он или какие-либо содержащиеся в нем файлы были изменены.  Предварительная компиляция кода рекомендуется, когда известно, что исходные файлы используют общие наборы файлов заголовков, но не содержат их в том же порядке, или если требуется включить исходный код в предварительную компиляцию.  Параметры предварительно скомпилированного заголовка: /Yc (создать файл предварительно скомпилированного заголовка) и /Yu (использовать файл предварительно скомпилированного заголовка). Используется /Yc для создания предварительно скомпилированного заголовка. При использовании с необязательным hdrstop pragma /Yc позволяет предварительно скомпилировать файлы заголовков и исходный код. Выберите /Yu для использования существующего предварительно скомпилированного заголовка в существующей компиляции. С параметрами /Yc и /Yu параметрами можно также /Fp указать альтернативное имя предкомпилированного заголовка. |

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 9. **Кодировка: определение, назначение, примеры. Кодировка ASCII. Структура кодировки Windows-1251. Отличие ASCII и Windows-1251. Символы времени выполнения: установка кодовой страницы языкового стандарта.**  Кодировка - таблица, задающая кодировку конечного множества символов алфавита (обычно элементов текста: букв, цифр, знаков препинания). Такая таблица сопоставляет каждому символу последовательность длиной в один или несколько символов другого алфавита (точек и тире в коде Морзе, нулей и единиц (битов) в компьютере). Примеры: ASCII, Windows-1251, Кодировки, UTF-8, UTF-16 и UTF-32 набора символов Юникод.  ASCII — американский стандартный код для обмена информацией. ASCII представляет собой 8-битную кодировку для представления десятичных цифр, латинского и национального алфавитов, знаков препинания и управляющих символов. Нижнюю половину кодовой таблицы (0 — 127) занимают символы US-ASCII, а верхнюю (128 — 255) — разные другие нужные символы (CP866, CP1251).  Windows-1251 — набор символов и кодировка, являющаяся стандартной 8-битной кодировкой для русских версий MW.  Кодировка ASCII - кодировка для представления латинского алфавита, десятичных цифр, некоторых знаков препинания, арифметических операций и управляющих символов. Windows-1251 – для символов русского алфавита.  Набор символов времени трансляции: текст программы на языке программирования хранится в исходных файлах и основан на определенной кодировке символов  Набор символов времени выполнения: символы, отображаемыми в среде выполнения. Любые дополнительные символы зависят от локализации  Дополнительные символы времени выполнения определяются setlocale. По умолчанию, локаль SetLocale(LC\_ALL, "C") устанавливает стандартный контекст С.  Во время выполнения можно установить кодовую страницу языкового стандарта, используя вызов setlocale(LC\_CTYPE, "rus") или воспользоваться следующими функциями, необходимо:  включить заголовочный файл <windows.h>: #include <windows.h> Директива #pragma позволяет указать целевой языковой стандарт во время компиляции. Это гарантирует, что строки с расширенными символами будут сохраняться в правильном формате. | 10. **Кодировка: определение, назначение, примеры. Кодировка UNICODE: назначение, структура, UCS, UTF. Прямой (LE) и обратный (BE) порядок байт. BOM: определение, назначение, примеры.**  Кодировка - таблица, задающая кодировку конечного множества символов алфавита (обычно элементов текста: букв, цифр, знаков препинания). Такая таблица сопоставляет каждому символу последовательность длиной в один или несколько символов другого алфавита (точек и тире в коде Морзе, нулей и единиц (битов) в компьютере). Примеры: ASCII, Windows-1251, Кодировки, UTF-8, UTF-16 и UTF-32 набора символов Юникод.  UNICODE: это стандарт кодирования символов, позволяющий представить знаки почти всех письменных языков, состоит из 2х разделов:  UCS - universal character set (универсальный набор символов);  UTF - Unicode transformation format (семейство кодировок).  Принято обозначение U+xxx, где xxx- число в шестнадцатеричном формате.  UCS расположены в 17 плоскостях (0-16), 216 (65 536) символов в каждой плоскости, плоскость 0 – основная (основные символы), 1-14 – дополнительные, 15-16 – для частного использования.  UTF-8 — представление Юникода, обеспечивающее совместимость со старыми системами, использовавшими 8-битные символы.  Алгоритм кодирования в UTF-8:   * определить количество октетов (октет: 8 битов или 1 байт) – в какой диапазон значений попадает количество значащих символов (7, 11, 16, 21, 26, 31); * подготовить старшие биты первого октета: * 0xxxxxxx для одного октета; * 110xxxxx – двух; * 1110xxxx - трех и т. д. * 10xxxxxx - для остальных октетов; * заполнить оставшиеся биты (обозначены как x) в октетах кодом символа Юникода в двоичном виде. Начать с младших битов, поставив их в младшие биты последнего октета кода. И так далее, пока все биты кода символа не будут перенесены в свободные биты октетов.   В UTF-16 символы кодируются двухбайтовыми словами с использованием всех возможных диапазонов значений (от 0 до FFFF16).  LE (Little endian order, прямой порядок, от младшего к старшему), BE (Big endian order, обратный порядок, от старшего к младшему).  BOM: для определения формата представления Юникода в начало текст. файла записывается сигнатура (обозначение) — символ U+FEFF — маркер последовательности байтов.   |  |  | | --- | --- | | **Кодировка** | **Представление (**[hex](https://ru.wikipedia.org/wiki/%D0%A8%D0%B5%D1%81%D1%82%D0%BD%D0%B0%D0%B4%D1%86%D0%B0%D1%82%D0%B5%D1%80%D0%B8%D1%87%D0%BD%D0%B0%D1%8F_%D1%81%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D0%B0_%D1%81%D1%87%D0%B8%D1%81%D0%BB%D0%B5%D0%BD%D0%B8%D1%8F)**)** | | [UTF-8](https://ru.wikipedia.org/wiki/UTF-8) | EF BB BF | | [UTF-16](https://ru.wikipedia.org/wiki/UTF-16) ([BE](https://ru.wikipedia.org/w/index.php?title=Big_Endian&action=edit&redlink=1)) | FE FF | | [UTF-16](https://ru.wikipedia.org/wiki/UTF-16) ([LE](https://ru.wikipedia.org/w/index.php?title=Little_Endian&action=edit&redlink=1)) | FF FE | | [UTF-32](https://ru.wikipedia.org/wiki/UTF-32) (BE) | 00 00 FE FF | | [UTF-32](https://ru.wikipedia.org/wiki/UTF-32) (LE) | FF FE 00 00 | | 11. **Структура ЯП: алфавит языка программирования, идентификаторы, зарезервированные идентификаторы, литералы, ключевые слова, определения. Правила составления ид-ров в C++ и других языков прог-ия. Примеры.**  Алфавит языка программирования: набор символов, разрешенных к использованию языком программирования. Основывается на одной из кодировок.  Базовый набор символов исходного кода: строчные и прописные буквы латинского и национального алфавитов, цифры, знаки операций, символы подчеркивания и пробела, ограничители и разделители, специальные символы  С помощью символов алфавита записываются служебные слова, которые составляют словарь языка.  Алфавит языка программирования служит для построения слов в языке программирования, которые называют лексемами. Примеры лексем: идентификаторы; ключевые (зарезервированные) слова; знаки операций; константы; разделители (скобки, знаки операций, точка, запятая, пробельные символы и т.д.).  Границы лексем определяются с помощью других лексем, таких, как разделители или знаки операций.  Правила составления идентификаторов C++:  • идентификаторы должны начинаться с буквы или подчеркивания;  • идентификатор не может совпадать с ключевыми словами С++ или с именами библиотечных функций;  • идентификаторы могут состоять из любого кол-ва символов, но компилятор гарантирует, что будет считать значащими:  - 31 первых символов идентификаторов, не имеющих внешней связи;  - более 6 значащих символов идентификаторов с внешней связью.  • идентификаторы чувствительны к регистру.  Длина идентификатора по стандарту не ограничена. Идентификатор создается при объявлении переменной, функции, типа и т. п.  Зарезервированные идентификаторы: идентификаторы, которые предварительно определены в системе программирования. Зарезервированные идентификаторы С++: все имена с двумя подчеркиваниями считаются зарезервированным.    Ключевые слова: последовательности символов алфавита языка, имеющие специальное назначение. Ключевые слова зарезервированы компилятором для обозначения типов переменных, класса хранения, элементов операторов и т.д. | 12. **Структура языка программирования: фундаментальные типы данных. Что определяет тип данных? Определения, примеры (C++ и другие языки)**  Типы данных: фундаментальные (встроенные) и определенные программистом. Тип данных определяет: внутр. представление данных в памяти компа; диапазон значений, кот. могут принимать величины этого типа; операции и ф-ии, кот. можно применять к величинам этого типа.  Фундаментальные типы задаются стандартом языка и встроены в компилятор.  Фундаментальные типы C++ определены следующими ключевыми словами: int (целый); char (символьный); wchar\_t (расширенный символьный); bool (логический); float (вещественный); double (вещественный с двойной точностью); тип void.  Модификаторы основных типов, уточняющие внутреннее представление и диапазон значений стандартных типов: short (короткий); long (длинный); signed (знаковый); unsigned (беззнаковый).    Внутреннее представление величины целого типа:  - целое число в двоичном коде.  - спецификатор signed – старший разряд (бит) числа интерпретируется как знаковый (0 – положительное число, 1 – отрицательное).  - спецификатор unsigned: старший разряд (бит) рассматривается как значащий, позволяет представлять только положительные числа.  По умолчанию все целочисленные типы считаются знаковыми, то есть спецификатор signed можно опускать. Диапазон значений зависит от реализации. |
| 13. **Структура ЯП: литералы, типы литералов, способы задания. Строки. Массивы данных фундаментального типа. Примеры (С++).**    Массивы данных фундаментальных типов: коллекция однородных данных, размещенных последовательно в памяти и допускающие доступ по индексу (вычисляется: смещение = индекс\*sizeof(базовый\_тип)).  Массив — это совокупность переменных, имеющих одинаковый тип и объединенных под одним именем. Доступ к отдельному элементу массива осуществляется с помощью индекса.  Объем памяти, необходимый для хранения массива, зависит от его типа и количества элементов в нем.  Размер одномерного массива в байтах вычисляется по формуле:  Кол-во\_байтов = sizeof(базовый\_тип) \* кол-во\_элементов  Многомерные массивы С/С++: многомерность моделируется в виде «массива массивов». Доступ к элементу, стоящему на пересечении первой строки и третьего столбца, можно получить двумя способами: либо индексируя массив – mm[0][2]; либо используя указатель – \*(( \*)mm+2). Правила адресной арифметики требуют приведения типа указателя на массив к его базовому типу. Двухмерный массив можно представить с помощью указателей на одномерные массивы. | 14. **Структура ЯП: фундаментальные типы данных. Типы данных для хранения вещественных значений. Стандарт IEEE 754. Что определяет стандарт IEEE 754.**  Фундаментальные типы задаются стандартом языка и встроены в компилятор.  Фундаментальные типы C++ определены следующими ключевыми словами: int (целый); char (символьный); wchar\_t (расширенный символьный); bool (логический); float (вещественный); double (вещественный с двойной точностью); тип void.  Модификаторы основных типов, уточняющие внутреннее представление и диапазон значений стандартных типов: short (короткий); long (длинный); signed (знаковый); unsigned (беззнаковый).    Стандарт языка C++ определяет три типа данных для хранения вещественных значений: float, double и long double.  Стандарт IEEE 754 описывает формат представления чисел с плавающей точкой. Используется в программных (компиляторы разных языков программирования) и аппаратных (CPU и FPU) реализациях арифметических действий (математических операций).  Стандарт описывает:  • формат чисел с плавающей точкой: мантисса, показатель (экспонента), знак числа;  • представление положительного и отрицательного нуля, положительной и отрицательной бесконечностей, а также нечисла (англ. Not-a-Number, NaN);  • методы, исп. для преобразования числа при выполнении математических операций;  • исключительные ситуации: деление на 0, переполнение, потеря значимости, работа с денормализованными числами и другие;  • операции: арифметические и другие. | 15. **Структура ЯП: фундаментальные типы данных. Внутр. представление величины целого типа, спецификаторы, диапазоны значений целого типа.** Фундаментальные типы задаются стандартом языка и встроены в компилятор.  Фундаментальные типы C++ определены следующими ключевыми словами: int (целый); char (символьный); wchar\_t (расширенный символьный); bool (логический); float (вещественный); double (вещественный с двойной точностью); тип void.  Модификаторы основных типов, уточняющие внутреннее представление и диапазон значений стандартных типов: short (короткий); long (длинный); signed (знаковый); unsigned (беззнаковый).    Внутреннее представление величины целого типа:  • целое число в двоичном коде.  • спецификатор signed – старший разряд (бит) числа интерпретируется как знаковый (0 – положительное число, 1 – отрицательное).  • спецификатор unsigned: старший разряд (бит) рассматривается как значащий, позволяет представлять только положительные числа.  По умолчанию все целочисленные типы считаются знаковыми, то есть спецификатор signed можно опускать. Диапазон значений зависит от реализации.  Два стандартных включаемых заголовочных файла, <limits.h> и <float.h>, определяют числовые ограничения или минимальное и максимальное значения, которые может хранить переменная данного типа.  **int**: целое число. В зависимости от архитектуры процессора может занимать 2 байта или 4 байта. Диапазон предельных значений: от –32768 до 32767 (при 2 байтах) или от −2 147 483 648 до 2 147 483 647 (при 4 байтах). Но в любом случае размер должен быть больше или равен размеру типа short и меньше или равен размеру типа long. Данный тип имеет синонимы signed int и signed.  **unsigned int**: положительное целое число. Может занимать 2 байта или 4 байта, диапазон предельных значений: от 0 до 65535 (для 2 байт), либо от 0 до 4 294 967 295 (для 4 байт). В качестве синонима этого типа может использоваться unsigned  **long**: целое число в диапазоне от −2 147 483 648 до 2 147 483 647. Занимает в памяти 4 байта. У данного типа также есть синонимы long int, signed long int и signed long  **unsigned long**: целое число в диапазоне от 0 до 4 294 967 295. Занимает в памяти 4 байта. Имеет синоним unsigned long int.  **long long**: целое число в диапазоне от −9 223 372 036 854 775 808 до +9 223 372 036 854 775 807. Занимает в памяти, как правило, 8 байт. Имеет синонимы long long int, signed long long int и signed long long.  **unsigned long long**: целое число в диапазоне от 0 до 18 446 744 073 709 551 615. Занимает в памяти, как правило, 8 байт. | 16. **Структура ЯП: система типов ЯП, вывод типов. Преобразование типов: автоматическое преобразование, явное преобразование. Определения и примеры (С++)**  Тип данных – множество значений и операций над этими значениями. (IEEE Std 1320.2-1998).  Тип данных определяет:  • внутреннее представление данных в памяти компьютера;  • множество значений, которые могут принимать величины этого типа;  • операции и функции, которые можно применять к величинам этого типа.  Система типов – совокупность правил в языках программирования, назначающих свойства, называемыми типами, различным конструкциям, составляющим программу (переменным, выражениям, функциям, модулям) для возможности выполнения проверки типов во время компиляции или во время выполнения,  Основные функции системы типов данных:  - обеспечение безопасности: проверяется каждая операция на получение аргументов именно тех типов, для которых она имеет предназначена;  - оптимизация: на основе типа выбирается способ эффективного хранения значения и алгоритмов его обработки;  - документация: подчеркивается намерения программиста;  - абстракция: использование типов данных высокого уровня позволяет программисту думать о значениях как о высокоуровневых сущностях, а не как о наборе битов.  Вывод типов:  В стандарте C++11 введено новое ключевое слово auto для определения явно инициализируемой переменной.  Создается переменная, тип которой выводится из инициализирующего значения:  auto variable1 = 5; auto variable2 = 2.5;  Преобразование типов:  • автоматическое преобразование;  • явное преобразование.   |  |  | | --- | --- | | ***явное*** | задается программистом в тексте программы  с помощью:   * конструкции языка; * ф-ии, принимающей значение одного типа и возвращ. значение другого типа. | | ***неявное*** | Выполн. автоматически транслятором (компилятором или интерпретатором) по правилам, описанным в стандарте языка. | |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 17. **Структура ЯП: система типов языка программирования, Вывод типов. Преобразование типов: расширяющее преобразование, сужающее преобразование. Назначение оператора sizeof языка программирования С++. Примеры.**  Тип данных – множество значений и операций над этими значениями. (IEEE Std 1320.2-1998). Он определяет:  • внутреннее представление данных в памяти компьютера;  • множество значений, которые могут принимать величины этого типа;  • операции и функции, которые можно применять к величинам этого типа.  Система типов – совокупность правил в языках программирования, назначающих свойства, называемыми типами, различным конструкциям, составляющим программу (переменным, выражениям, функциям, модулям) для возможности выполнения проверки типов во время компиляции или во время выполнения,  Основные функции системы типов данных:  - обеспечение безопасности: проверяется каждая операция на получение аргументов именно тех типов, для которых она имеет предназначена;  - оптимизация: на основе типа выбирается способ эффективного хранения значения и алгоритмов его обработки;  - документация: подчеркивается намерения программиста;  - абстракция: использование типов данных высокого уровня позволяет программисту думать о значениях как о высокоуровневых сущностях, а не как о наборе битов.  Преобразование типов:   |  |  | | --- | --- | | ***явное*** | задается программистом в тексте программы  с помощью:   * конструкции языка; * ф-ии, принимающей значение одного типа и возвращ. значение другого типа. | | ***неявное*** | Выполн. автоматически транслятором (компилятором или интерпретатором) по правилам, описанным в стандарте языка. |   Стандарты большинства языков запрещают неявные преобразования.  Автоматическое (неявное) преобразование типов: для базовых типов преобразование типов выполняется без потери точности.      Если выбранное преобразование является расширяющим, компилятор выполняет его, не информируя о выполнении такого преобразования. Расширяющие преобразования всегда являются надежными. Если преобразование является сужающим, компилятор выдает предупреждение о возможной потере данных.  Если компилятору не удается найти допуст. преобразование, то выдается ошибка и объектный код не создается.  В С/С++ размер переменной любого типа данных зависит от компилятора и/или архитектуры компьютера. Фактический размер переменных может отличаться на разных компьютерах. Для его определения используют оператор sizeof.  Стандарт задает отношение размера между целыми типами:  1 == sizeof(char) <= sizeof(short) <= sizeof(int) <= sizeof(long) <= sizeof(long long)  и для переменных с плавающей запятой:  sizeof(float) <= slzeof(double) <= sizeof(long double)  Также поддерж. целочисленные типы с указанием их размера: \_\_int8, \_\_int16, \_\_int32, \_\_int64 и \_\_int8, \_\_int16, \_\_int32. | 18. **Структура ЯП: пользовательские типы данных (структуры, перечисления, объединения). Определение, примеры (C++). Массивы данных пользовательского типа. Перегрузка операторов для пользовательских типов**  Пользовательские типы данных: типы, создаваемые пользователем, на основе того, что доступно в языке; всегда есть объявление типа.  С++: struct address { //структура - это набор переменных, объединенных общим именем. Она обеспечивает удобный способ организации взаимосвязанных данных.  string street;  int number\_of\_house;  };  struct rgb\_color { //битовое поле – особый вид полей структуры, используемый для плотной упаковки данных.  unsigned red\_value: 3;  unsigned green\_value: 3;  unsigned blue\_value: 3;  };  union bit{ //объединение - это пользовательская переменная, которая может хранить объекты различного типа и размера, для их размещения выделяется одна общая память, размерность определяется размерностью максимального элемента объединения.  char ch;  int i;  };  enum coin{ //перечисление - набор именованных целых констант  penny, nickel, dime  };  typedef unsigned int nomer; // определяет новое имя типа данных, новый тип при этом не создается, уже существующий тип получает новое имя  Массивы данных пользовательского типа: это совокупность данных (переменных) пользовательского типа и объединенных под одним именем. Доступ к отдельному элементу массива осуществляется с помощью индекса.  • Массив struct:  Struct {  int i;  char ch;  } S[5];  Обращение: S[0].i;  • Массив union:  Union {  float f;  int i;  } U[5];  Обращение: U[1].f;  (возможность перегрузки есть только для пользовательских типов/классов). Перегрузка операторов — один из способов реализации полиморфизма, заключающийся в возможности одновременного существования в одной области видимости нескольких различных вариантов применения операторов, имеющих одно и то же имя, но различающихся типами параметров, к которым они применяются. | 19. **Структура ЯП: пользовательские типы данных. Определение структуры, примеры (C++). Перегрузка операторов для пользовательских типов.**  Пользовательские типы данных: типы, создаваемые пользователем, на основе того, что доступно в языке; всегда есть объявление типа.  С++: struct address { //структура - это набор переменных, объединенных общим именем. Она обеспечивает удобный способ организации взаимосвязанных данных.  string street;  int number\_of\_house;  };  struct rgb\_color { //битовое поле – особый вид полей структуры, используемый для плотной упаковки данных.  unsigned red\_value: 3;  unsigned green\_value: 3;  unsigned blue\_value: 3;  };  union bit{ //объединение - это пользовательская переменная, которая может хранить объекты различного типа и размера, для их размещения выделяется одна общая память, размерность определяется размерностью максимального элемента объединения.  char ch;  int i;  };  enum coin{ //перечисление - набор именованных целых констант  penny, nickel, dime  };  typedef unsigned int nomer; // определяет новое имя типа данных, новый тип при этом не создается, уже существующий тип получает новое имя  Массивы данных пользовательского типа: это совокупность данных (переменных) пользовательского типа и объединенных под одним именем. Доступ к отдельному элементу массива осуществляется с помощью индекса.  • Массив struct:  Struct {  int i;  char ch;  } S[5];  Обращение: S[0].i;  • Массив union:  Union {  float f;  int i;  } U[5];  Обращение: U[1].f;  (возможность перегрузки есть только для пользовательских типов/классов). Перегрузка операторов — один из способов реализации полиморфизма, заключающийся в возможности одновременного существования в одной области видимости нескольких различных вариантов применения операторов, имеющих одно и то же имя, но различающихся типами параметров, к которым они применяются. | 20. **Структура ЯП: понятие конфликта имен, область видимости пер-ных, пространства имен, псевдонимы пространства имен.**  Область видимости переменных в C++: доступность переменных по их идентификатору в разных частях (блоках программы).  Переменная должна быть объявлена до ее использования; переменная объявленная во внутреннем блоке (локальная переменная {…}) не доступна во внешнем; переменная объявленная во внешнем блоке доступна во внутреннем; во внутреннем блоке переменная может быть переобъявлена.  Пространство имен: именованная область видимости. Применяется для разрешения конфликтов имен. Имена, объявленные в одном пространстве имен, не будут вступать в конфликт с аналогичными именами, объявленными в другой области.  Примеры: namespace, using, псевдонимы пространства имен.  Псевдоним пространства имен: Имена пространств имен должны быть уникальными, из-за чего зачастую они получаются не слишком короткими. Если длинное имя оказывается трудночитаемым или его сложно вводить в файле заголовка, где нельзя использовать директиву using, можно создать псевдоним пространства имен, который будет служить в качестве сокращения фактического имени.  Пространства имен помогают предотвратить конфликты имен, но не устранить их полностью. Такой конфликт может произойти, когда одно и то же имя объявляется в двух разных пространствах имен и затем предпринимается попытка сделать видимым оба пространства. В таком случае для указания предполагаемого пространства имен явным образом можно воспользоваться описателем псевдонима пространства имен «::». |
| 21. **Структура ЯП: определение выражения, его состав, порядок вычисл. выражения. Символ окончания последовательности. Примеры (С++).**  Выражение – объединение литералов, имен (переменных, функций и пр.), операторов и специальных символов, служащих для вычисления выражения или достижения побочных эффектов (например: при применении в выражении функций).    Символ окончания последовательности (в языке программирования C++ – это точка с запятой) определяет точку последовательности (sequence point), в которой завершились все вычисления и побочные эффекты. Точка последовательности – момент времени, когда побочные эффекты вычисленных выражений уже случились, а побочные эффекты следующих в последовательности выражений еще не начались. | 22. **Структура ЯП: выражения (lvalue, rvalue, побочные эффекты, точка последовательности, унарные, бинарные и тернарные выражения), константные выражения, укороченное вычисление. Примеры (С++).**  lvalue (именующее выражение) – это ссылка на значение; могут использоваться в левой и правой части оператора присваивания. Имя переменной, ссылка на элемент массива по индексу, вызов функции возвращающей указатель, всегда связаны с областью памяти, адрес которой известен.    rvalue (значащее выражение) – может использоваться только в правой части оператора присваивания (не связано с адресом, связано только со значением; это могут быть литералы, вызов функции, возвращающей значение).        Унарные – с 1 операндом (\*p, &a, +, -, !, ~, ++, --. Sizeof, new, delete).  Бинарные – 2 операнда (\*, /, %, +, -, <<, >>, >, <, >=, <=, ==, !=, &&, ||).  Тернарный – 6 операнда: bool и 2 выражения.  Укороченное вычисление – вычисление, значение которого может быть вычислено по части выражения.  Таблица истинности для логических операторов:   |  |  |  |  |  | | --- | --- | --- | --- | --- | | **x** | **y** | **x && y** | **x || y** | **! x** | | 0 | 0 | 0 | 0 | 1 | | 0 | 1 | 0 | 1 | 1 | | 1 | 1 | 1 | 1 | 0 | | 1 | 0 | 0 | 1 | 0 | | 23. **Структура ЯП: инструкции языка программирования, объявление, определение, инициализация, простые и составные инструкции, инструкции выбора, инструкции циклов, инструкции переходов, примеры (С++).**  Инструкция (оператор) — наименьшая автономная часть языка программирования; команда или набор команд. Программа обычно представляет собой последовательность инструкций.  Объявление – описание некоторой сущности (сигнатура функции, определение типа, описание внешней переменной, шаблон и т.п.). Объявление уведомляет компилятор о её существовании и свойствах.  Определение – реализация некоторой сущности (переменная, функция, метод класса и т.п.). При обработке определения компилятор генерирует информацию для объектного модуля – исполняемый код, резервирование памяти под переменную и т.д.  Простые инструкции завершаются точкой с запятой. Составные инструкции, или блоки ({ }), состоят из одной или более инструкций (каждая из которых сама может быть составной), заключенных в фигурные скобки.  Инструкции выбора if и switch, позволяют выполнять те или иные участки кода в зависимости от выполнения условий.  Инструкции циклов используются для многократного повторения фрагментов кода. Цикл for мы можем использовать, если знаем точное количество действий (итераций). Когда мы не знаем, сколько итераций должен произвести цикл, нам понадобится цикл while или do…while. Цикл do…while почти ничем не отличается от цикла while, за исключением того, что тело цикла гарантированно выполняется хотя бы один раз.  Инструкции переходов выполняют немедленную локальную передачу контроля. Оператор break завершает выполнение ближайшего внешнего цикла или условного оператора, в котором он находится. Управление передается оператору, который расположен после оператора, при его наличии. Инструкцию continue можно располагать только внутри цикла. Она вызывает переход к следующей итерации самого внутреннего содержащего ее цикла. С помощью return функция возвращает управление в программу, откуда была вызвана. Если за return следует выражение, то его значение возвращается вызвавшей эту функцию программе. Оператор goto осуществляет безусловную передачу управления оператору, метка которого задана идентификатором. Метка оператора имеет значение только для оператора goto; в остальных случаях метки операторов игнорируются. Повторное объявление меток невозможно.  Инструкции обработки исключений используют три ключевых слова: try, catch и throw. Те инструкции программы, где ожидается возможность появления исключительных ситуаций, содержатся в бло¬ке try. Если в блоке try возникает исключение, т. е. ошибка, то генерируется исключение throw. Исклю¬чение перехватывается, используя catch, и обрабатывается. | 24. **Структура ЯП: инструкции языка программирования, инструкции обработки исключений, примеры (С++).**    **Инструкции выбора** *if*и *switch*, позволяют выполнять те или иные участки кода в зависимости от выполнения условий.  **Инструкции циклов** используются для многократного повторения фрагментов кода. Цикл *for* мы можем использовать, если знаем точное количество действий (итераций). Когда мы не знаем, сколько итераций должен произвести цикл, нам понадобится цикл *while* или *do…while*. Цикл *do…while* почти ничем не отличается от цикла *while*, за исключением того, что тело цикла гарантированно выполняется хотя бы один раз.  **Инструкции переходов** выполняют немедленную локальную передачу контроля. Оператор *break* завершает выполнение ближайшего внешнего цикла или условного оператора, в котором он находится. Управление передается оператору, который расположен после оператора, при его наличии. Инструкцию *continue* можно располагать только внутри цикла. Она вызывает переход к следующей итерации самого внутреннего содержащего ее цикла. С помощью *return* функция возвращает управление в программу, откуда была вызвана. Если за *return* следует выражение, то его значение возвращается вызвавшей эту функцию программе. Оператор *goto* осуществляет безусловную передачу управления оператору, метка которого задана идентификатором. Метка оператора имеет значение только для оператора *goto*; в остальных случаях метки операторов игнорируются. Повторное объявление меток невозможно.  **Инструкции обработки исключений** используют три ключевых слова: *try*, *catch* и *throw*. Те инструкции программы, где ожидается возможность появления исключительных ситуаций, содержатся в бло­ке *try*. Если в блоке *try* возникает исключение, т. е. ошибка, то генерируется исключение throw. Исклю­чение перехватывается, используя *catch*, и обрабатывается. |

|  |  |  |  |
| --- | --- | --- | --- |
| 25. **Структура ЯП: программные конструкции (блоки, ф-ии, процедуры и пр.). Ф-ии: передача пар-ов по значению и по ссылке, передача пер-ного числа пар-ов в ф-ии C++. Примеры.**  Блоки: логически сгруппированный набор идущих подряд инструкций в исходном коде программы, является основой парадигмы структурного программирования  Пример С++:  {  int a = 1;  int b = 2;  int c = a + b;  return c;  }  Функции: фрагмент программного кода, к которому можно обратиться из другого места программы  Пример С++:  void name(char\* text) {  std::cout<<text<< std::endl;  }  Процедуры: ф-ии не возвр-щие значения, return не обязателен, но желателен для явного указания завершения функции  Пример С++:  void ValidMessage() {  cout << "Пароль введен верно" << endl;  }  Передача параметров: передача параметров в функцию происходит через стек. Код, вызывающий ф-ию, знает, сколько параметров ей передать и каковы значения этих параметров.  Передача параметров  по значению: они при выходе из функции не изменятся  Пример С++:  int func(int k){  k\*=2;  return k;  }  void main(){  int z=1, y=3, k;  k=func(z)+func(y);  cout<<z<<" "<<y;  }  по ссылке: при выходе из функции из значения могут измениться  int func(int &k){  k\*=2;  return k;  }  void main(){  int z=1, y=3, k;  k=func(z)+func(y);  cout<<z<<" "<<y;  }  Передача переменного числа параметров в функции С++:  По умолчанию параметры передаются в ф-ию через стек. Поэтому, технически, нет ограничения на кол-во передаваемых параметров. Функции с переменным числом параметров объявляются как обычные функции, но вместо недостающих аргументов ставится многоточие. Количество параметров и их типы известны только при вызове функции.  Передать функции параметры можно двумя способами:  • явно передать обязательный аргумент, задающий число параметров;  • добавить в конец списка параметр с уникальным значением, по которому будет определяться конец списка параметров. | 26. **Структура ЯП: прогр. конструкции (блоки, функции, процедуры и пр.). Передача параметров в ф-ию. Передача параметров по значению и по ссылке, передача значений параметров по умолчанию в ф-ии C++. Примеры.**  Блоки: логически сгруппированный набор идущих подряд инструкций в исходном коде программы, является основой парадигмы структурного программирования  Пример С++:  {  int a = 1;  int b = 2;  int c = a + b;  return c;  }  Функции: фрагмент программного кода, к которому можно обратиться из другого места программы  Пример С++:  void name(char\* text) {  std::cout<<text<< std::endl;  }  Процедуры: ф-ии не возвр-щие значения, return не обязателен, но желателен для явного указания завершения функции  Пример С++:  void ValidMessage() {  cout << "Пароль введен верно" << endl;  }  Передача параметров: передача параметров в функцию происходит через стек. Код, вызывающий ф-ию, знает, сколько параметров ей передать и каковы значения этих параметров.  Передача параметров  по значению: они при выходе из функции не изменятся  Пример С++:  int func(int k){  k\*=2;  return k;  }  void main(){  int z=1, y=3, k;  k=func(z)+func(y);  cout<<z<<" "<<y;  }  по ссылке: при выходе из функции из значения могут измениться  int func(int &k){  k\*=2;  return k;  }  void main(){  int z=1, y=3, k;  k=func(z)+func(y);  cout<<z<<" "<<y;  } | 27. **Структура ЯП: программные конструкции (блоки, ф-ии, процедуры). Передача параметров в ф-ию. Реализация программных конструкций: лямбда-ф-ии в С++. Примеры.**  Блоки: логически сгруппированный набор идущих подряд инструкций в исходном коде программы, является основой парадигмы структурного программирования  {  int a = 1;  int b = 2;  int c = a + b;  return c;  }  Функции: фрагмент программного кода, к которому можно обратиться из другого места программы  void name(char\* text) {  std::cout<<text<< std::endl;  }  Процедуры: ф-ии не возвр-щие значения, return не обязателен, но желателен для явного указания завершения функции  void ValidMessage() {  cout << "Пароль введен верно" << endl;  }  Передача параметров: передача параметров в ф-ию происходит через стек. Код, выз-щий ф-ию, знает, сколько параметров ей передать и каковы значения этих параметров. | 28. **Структура ЯП: прогр. конструкции (блоки, ф-ии, процедуры). Передача параметров в ф-ию. Реализация программных конструкций в C++. Перегружаемые функции в C++. Inline-функции в C++. Шаблоны функций С++.**  Блоки: логически сгруппированный набор идущих подряд инструкций в исходном коде программы, является основой парадигмы структурного программирования  {  int a = 1;  int b = 2;  int c = a + b;  return c;  }  Функции: фрагмент программного кода, к которому можно обратиться из другого места программы  void name(char\* text) {  std::cout<<text<< std::endl;  }  Процедуры: ф-ии не возвр-щие значения, return не обязателен, но желателен для явного указания завершения функции  void ValidMessage() {  cout << "Пароль введен верно" << endl;  }  Передача параметров: передача параметров в ф-ию происходит через стек. Код, выз-щий ф-ию, знает, сколько параметров ей передать и каковы значения этих параметров.      inline int sum(int a, int b) {  return a + b;  } |
| 29. **Структура ЯП: программные конструкции (блоки, функции, процедуры и пр.). Передача параметров в ф-ию. Реализация прогр. конструкций в C++: соглашения о вызовах (\_cdecl).**  Блоки: логически сгруппированный набор идущих подряд инструкций в исходном коде программы, является основой парадигмы структурного программирования  {  int a = 1;  int b = 2;  int c = a + b;  return c;  }  Функции: фрагмент программного кода, к которому можно обратиться из другого места программы  void name(char\* text) {  std::cout<<text<< std::endl;  }  Процедуры: ф-ии не возвр-щие значения, return не обязателен, но желателен для явного указания завершения функции  void ValidMessage() {  cout << "Пароль введен верно" << endl;  }  Передача параметров: передача параметров в ф-ию происходит через стек. Код, выз-щий ф-ию, знает, сколько параметров ей передать и каковы значения этих параметров.  Соглашение о вызовах – протокол взаимодействия вызывающей и вызываемой ф-ии.  \_cdecl: Параметры функций помещаются в стек, порядок параметров "справа налево". Параметры, размер которых меньше 4-х байт, расширяются до 4-х байт. Адрес возврата кладется в стек поверх параметров. | 30. **Структура ЯП: прог. конструкции (блоки, ф-ии, процедуры и пр.). Передача параметров в ф-ию. Реализация прогр. конструкций в C++: соглашения о вызовах (\_stdcall).**  Блоки: логически сгруппированный набор идущих подряд инструкций в исходном коде программы, является основой парадигмы структурного программирования  {  int a = 1;  int b = 2;  int c = a + b;  return c;  }  Функции: фрагмент программного кода, к которому можно обратиться из другого места программы  void name(char\* text) {  std::cout<<text<< std::endl;  }  Процедуры: ф-ии не возвр-щие значения, return не обязателен, но желателен для явного указания завершения функции  void ValidMessage() {  cout << "Пароль введен верно" << endl;  }  Передача параметров: передача параметров в ф-ию происходит через стек. Код, выз-щий ф-ию, знает, сколько параметров ей передать и каковы значения этих параметров.  Соглашение о вызовах – протокол взаимодействия вызывающей и вызываемой ф-ии.  \_stdcall: Параметры помещаются в стек, порядок параметров "справа налево". Адрес возврата кладется в стек поверх параметров. Стек освобождает вызываемый код. В последней инструкции вызываемого кода RET указывается суммарный размер в байтах всех параметров процедуры. Команда RET после извлечения адреса возврата прибавляет к регистру ESP указанное значение. Целостность стека восстанавливается вызываемым кодом. Возвращаемый параметр передается через регистр EAX. | 31. **Структура ЯП: программные конструкции (блоки, ф-ии, процедуры). Передача параметров в ф-ию. Реализация прогр. конструкций в C++: соглашения о вызовах (\_fastcall).**  Блоки: логически сгруппированный набор идущих подряд инструкций в исходном коде программы, является основой парадигмы структурного программирования  {  int a = 1;  int b = 2;  int c = a + b;  return c;  }  Функции: фрагмент программного кода, к которому можно обратиться из другого места программы  void name(char\* text) {  std::cout<<text<< std::endl;  }  Процедуры: ф-ии не возвр-щие значения, return не обязателен, но желателен для явного указания завершения функции  void ValidMessage() {  cout << "Пароль введен верно" << endl;  }  Передача параметров: передача параметров в ф-ию происходит через стек. Код, выз-щий ф-ию, знает, сколько параметров ей передать и каковы значения этих параметров.  Соглашение о вызовах – протокол взаимодействия вызывающей и вызываемой ф-ии.  Соглашение вызовах \_fastcall (не стандартизированный, для внутренних вызовов): | 32. **Структура ЯП: стандартная библиотека. Реализация ст. библиотеки STL в C++. Понятие контейнера, итератора и алг-ма. Стандартные ф-ии STL для работы со строками: копир-ие, сравнение, выч-е длины, поиск символа и подстроки.**  В составе ЯП, как правило, есть обязательный (стандартный) набор функций. Такие функции называют встроенными.  Встраиваться функции могут тремя способами: прямо в код транслятора; находиться в отдельной библиотеке; сочетание первого и второго случаев.  Осн. требования к набору средств стандартной библиотеки (Бьёрн Страуструп): эффективность; независимость от алгоритмов — должна предоставлять возможность задавать алгоритмы в качестве параметров; удобство и безопасность; завершённость; органично сочетаться с языком; типобезопасность; поддержка общепринятых стилей программирования; расширяемость — способность единообразно работать со встроенными типами данных и с типами, определяемыми пользователем.  Подходы к разработке стандартных библиотек ЯП:  - должна содержать в себе только те процедуры и функции, которые используются практически всеми и обладают максимальной универсальностью;  - должна содержать в себе максимально возможное количество типичных алгоритмов, обеспечивать простую работу с большинством объектов (в идеале, со всеми), с которыми может взаимодействовать программа. Пример реализации этого подхода является язык Python.  Стандартная библиотека C++ STL: библиотека стандартных шаблонов – набор согласованных обобщённых алгоритмов, контейнеров, средств доступа к их содержимому и различных вспомогательных функций в C++.  STL (Standard Template Library) – стандартная библиотека шаблонов. Библиотека содержит универсальные шаблонные классы и функции, реализующие большое количество распространенных универсальных алгоритмов и структур данных. Т.к. библиотека STL состоит из шаблонных классов, ее алгоритмы и структуры можно применять практически к любым типам данных.  Частью стандартной библиотеки С++ является библиотека STL, кот. содержит 5 основных видов компонентов:  - контейнер: управляет набором объектов в памяти.  - итератор: обеспечивает для алгоритма средство доступа к содержимому контейнера.  - алгоритм: определяет вычислительную процедуру.  - функциональный объект: инкапсулирует функцию в объекте для использования другими компонентами.  - адаптер: адаптирует компонент для обеспечения различного интерфейса.  Функции для работы со строками:  strlen() (от слова length – длина)  strcpy() – копирование  strcmp() - Функция strcmp() сравнивает две строки: s1 и s2. Она возвращает целое число, которое меньше, больше нуля или равно ему, если s1 соответственно меньше, больше или равно s2. Функция strncmp() работает аналогичным образом, но сравнивает только первые n символов строки s1.  .find() - находит первое вхождение элемента. Он использует оператор = для сравнения. |

|  |  |  |  |
| --- | --- | --- | --- |
| 33. **Классы памяти: код, стек, статические данные, динам. область памяти. Структура динам. памяти (Heap) C++. Фрагментация динамической памяти C++.**  Область кода – память, в которой размещается код программы.  Локальная статическая память.  Переменная с ключевым словом static — это статическая переменная. Время ее жизни — постоянное. Область видимости статической переменной ограничена одним файлом, внутри которого она определена, ее можно использовать только после ее объявления.  Ключевое слово static в языке С/С++ используется для двух различных целей:  - как указание типа памяти: переменная располагается в статической памяти;  - как способ ограничить область видимости переменной рамками одного файла (в случае описания переменной вне функции). | 34. **Механизм обработки исключений: опред-е, назначение, применение. Реализация обр-ки исключений в C++. Пример.**  Исключение – событие при выполнении проги, при кот. ее дальнейшее выполнение становится бессмысленным.  Исключение – это аномальное событие, произошедшее во время выполнения программы.  Примеры: логические ошибки разработчика (деление на 0, выход индекса за границы массива); ошибки времени выполнения (отсутствие свободной памяти).  Исключения нарушают нормальный ход работы программы и на подобные события нужно немедленно отреагировать.  Механизм обработки исключительных ситуаций в языке C++ основан на операторах try, catch и throw (ключевые слова).  Блок try содержит фрагмент программы, подлежащий контролю. Оператор throw генерирует исключение. Оператор catch обрабатывает исключение.  Если в ходе выполнения программы в блоке try возникает исключительная ситуация, то для ее обработки генерируется ошибка с помощью оператора. Синтаксис:  throw <тип\_генерируемого\_исключения>;  Затем исключительная ситуация перехватывается блоком catch для ее обработки. Выполнение программы переходит от оператора throw к блоку: catch (<тип\_исключения>){// тело }.  С одним блоком try может быть связано несколько сatch. Выбор нужного сatch-обработчика определяется типом исключительной ситуации.  После завершения обработки исключения выполнение возобновляется с инструкции, следующей за последним catch-обработчиком в списке. Для перехвата любых исключений используется конструкция вида: catch(...){// тело }  Управление к нему передается при исключении любого типа.  Если в программе нет подходящего обработчика catch, то вызывается функция terminate() из стандартной библиотек C++. По умолчанию terminate() активизирует функцию abort(), которая завершает программу.  В механизме исключений C++ управление передается от оператора throw в первый оператор catch, который может обработать выброшенныйтип исключения. При достижении оператора catch все автоматические переменные, находящиеся в области между операторами throw и catch, удаляются. Этот процесс называется очистка стека. Удаление происходит впорядке, обратном созданию локальных переменных. Далее выполнениепрограммы продолжается с инструкции, следующей за последним catch-обработчиком | 35. **Препроцессор: определение, назначение, применение, директивы, выражения, макросы, директивы условной компиляции. Примеры на C++.**  Препроцессор С++:  • директивы: #include, #define, #if, #else, #elif, #endif, #ifdef, #ifndef, #error, #line, #pragma, #undef;  • операторы: defined, #, ##.  Препроцессор C/С++: текстовый процессор, который обрабатывает текст исходного файла на первой фазе компиляции.  Инструкции, регламентирующие работу компилятора, называются директивами препроцессора.  Назначение: директивы препроцессора могут: заменить какие-то лексемы в исходном тексте; вставить содержимое других файлов в указанном месте; подавить компиляцию части файла.  Директивы препроцессора могут появляться в произвольном месте исходного текста, при этом они будут воздействовать только на оставшуюся часть исходного файла.  Препроцессор может быть отдельной программой, или же интегрирован в компилятор. Входные и выходные данные для препроцессора имеют текстовый формат. Препроцессор преобразует текст в соответствии с директивами препроцессора. В случае если текст не содержит директив препроцессора, то он остаётся без изменений. Препроцессор можно вызвать отдельно для обработки текста программы без ее компиляции.  Основные директивы препроцессора: **#include** – вставляет текст из указанного файла; **#define** – задаёт макроопределение (макрос) или символическую константу; **#undef** – отменяет предыдущее определение; **#if** – осуществляет условную компиляцию при истинности константного выражения; **#ifdef** – осуществляет условную компиляцию при определённости символической константы; **#ifndef** – осуществляет условную компиляцию при неопределённости символической константы; **#else** – ветка условной компиляции при ложности выражения; **#elif** – ветка условной компиляции, образ. слиянием else и if; **#endif** – конец ветки условной компиляции; **#line** – препроцессор изменяет номер текущей строки и имя компилируемого файла; **#error** – выдача диагностического сообщения; **#pragma** – действие, зависящее от конкретной реализации компилятора. **#include** (включение файла).  Директива #include вставляет содержимое заданного файла в место расположения этой директивы в исходном тексте программы. Синтаксис:  #include "путь к файлу" или #include <путь к файлу>  Управление процессом компиляции: «Прагма» – это инструкция компилятору С/С++, используется для указания опций компилятору для управления его работой. Синтаксис: #pragma параметры. Так #pragma once контролирует, чтобы конкретный включаемый файл при компиляции подключался строго один раз.  Директива условной компиляции #if позволяет подавить компиляцию части исходного файла. Каждая директива #if в исходном файле должна иметь соответствующую закрывающую директиву #endif.  Стрингификация (#) – это преобразование фрагмента кода в строковую константу, т.е. преобразование аргумента в строку. Например, в результате стрингификации STR(hello) аргумент преобразуется в символьную строку "hello".  Конкатенация (##) – это конкатенация двух строковых констант. При работе с макросами, это означает объединение двух лексических единиц в одну более длинную. Например, один аргумент макроса может быть объединен с другим аргументом или с каким-либо текстом. | 36. **Статическая библиотека: определение, назначение, применение. Реализация стат.** **библиотеки в Visual C++. Утилита LIB. Создание стат. библиотеки. Применение**  Библиотека – файл, содержащий либо код на интерпретируемом языке, либо байт-код для виртуальной машины. Например, библиотеки для языка Python могут распространяться либо в виде файлов с исходным кодом (расширение «py»), либо в виде файлов с байт-кодом (расширение «pyc», (пайк) «py» + буква «c» от англ. compiled).  Статическая библиотека: файл (обычно с расширением lib), содержащий объектные модули; входной файл для компоновщика (linker).  Достоинства: просто использовать; не требуется наличие самой библиотеки; 1 исполняемый файл (расширение .exe).  Недостатки: платформенно зависима; загружается в память с каждым экземпляром запущенного приложения; при изменении кода библиотеки необходима компоновка всех приложений, которые используют библиотеку.  Статическая библиотека – файл с объектными модулями, который подключается к программе на этапе компоновки.  Компоновщик (linker, редактор связей) – программа, принимающая один или несколько объектных модулей и формирующая на их основе загрузочный модуль.  Если программа состоит из нескольких объектных файлов, компоновщик собирает эти файлы в единый исполнимый модуль, вычисляя и подставляя адреса вместо неопределенных внешних имен, в течение времени компоновки (статическая компоновка) или во время исполнения (динамическая компоновка).  Статическая библиотека Microsoft: файл с расширением lib. Для работы с библиотекой предназначена утилита LIB.  1) Создание статической библиотеки (определяем имя решения, проекта и выбираем место размещения на диске).  2) Выбираем тип приложения «Статическая библиотека», в проект добавляем один или несколько файлов, содержащих реализации функций библиотеки.  3) Запускаем проект на выполнение. В пункте раздела «Библиотекарь» -> «Команд. строка» отображается текущее значение параметра /OUT. Расширение выходных файлов определено как .lib. Директорий – папка проекта Debug.  После построения проекта в папке решения Debug размещен файл статической библиотеки (.lib). В журнале проекта зафиксировано выполнение сборки проекта.  Режимы использования утилиты LIB:  - построение или изменение библиотеки;  - извлечение элемента-объекта библиотеки в файл;  - создание файла экспорта и библиотеки импорта.  Эти режимы взаимоисключающие, LIB можно использовать только в одном режиме. |
| 37. **Теория форм. языков. Определения: алфавит языка, цепочка, пустая цепочка, длина** **цепочки, равенство цепочек, конкатенация цепочек, итерация цепочки,** **операторы + и \*.**  Формальный язык – это множество конечных слов над конечным алфавитом, например, язык программирования.  Алфавит – конечное непустое множество допустимых символов языка (букв языка). Пример: V = {a,b} – алфавит V, состоящий из двух символов a и b.      **Длиной цепочки** α (обозначается ***|***α***|***) называется число составляющих ее символов.  ***Пример*:** *|*α*|* = 3*, |*β*|* = 4*, |*γ*| =* 7*.* Длина пустой цепочки *|*ε*|*=0.  Цепочки α и β **равны** (α = β), если они имеют один и тот же состав символов, одно и тоже количество символов α = β и тот же порядок следования символов.  **Конкатенацией** (сцеплением) цепочек α и β называется цепочка γ*=*αβ, в которой символы данных цепочек записаны друг за другом. ***Пример*:** α=*abc,* β=*aaaa,* αβ*= abcaaaa.*  Итерация - повторение цепочки n раз, где n принадлежит натур. числам, – это конкатенация цепочки самой с собой n раз. | 38. **Теория форм. языков. Определение форм. языка, эквивалентность двух языков, способы задания форм. языка. Лексика, синтаксис и семантика языка. Примеры.**  Теория формальных языков (формальных грамматик) занимается описанием, распознаванием и переработкой языков. Описание любого языка должно быть конечным, хотя сам язык может содержать бесконечное множество цепочек.  Формальный язык – это множество конечных слов над конечным алфавитом, например, язык программирования.    Способ задания языка называется грамматикой этого языка. Грамматикой мы называем любой способ задания языка.  Язык L можно определить 3 способами: перечислением всех цепочек языка; указанием способа (алгоритма) порождения цепочек; определением метода (алгоритма) распознавания цепочек.  Лексика ЯП – множество цепочек языка. Синтаксис языка – набор формальных правил, определяющий конструкции (последовательности цепочек) языка. Семантика языка – набор неформальных правил, которые описываются словесно (например, в руководстве программиста). Пример: переменную надо объявить до ее применения.  Чтобы создать ЯП, следует определить: множество допустимых символов языка (алфавит); формально описать множество правильных программ; задать семантические правила языка.  Множество допустимых символов может быть проверено. Определить множество формально правильных программ можно с помощью алгоритма-распознавателя. Распознаватель строится на основе формального описания языка – его формальной грамматики.  Семантические правила могут быть реализованы в виде эвристическихалгоритмов (алгоритм, не имеющий строгого обоснования, но дающийприемлемое решение) или в виде словесного (неформального) описания правил языка. | 39. **Теория форм. языков. Форм. Грамм-ка (определение, назначение),** **выводимость цепочки символов в грамм-ке, сентенциальная форма грамматики, язык,** **порождаемый грамматикой,** **способы задания грамматик. Примеры.**  Формальный язык – матем. модель реального языка.  Сентенциальная форма — последовательность терминалов и нетерминалов, выводимых из начального символа.  Терминалы представляют собой базовые символы, из которых формируются строки. Слово "токен" является синонимом слова "терминал", когда мы говорим о грамматиках языков программирования. Нетерминалы представляют собой синтаксические переменные, которые обозначают множества строк. Нетерминалы определяют множества строк, которые помогают в определении языка, порождаемого грамматикой.  Язык, порождаемый грамматикой – это множество всех выводимых из аксиомы грамматики терминальных цепочек.  Язык L можно определить 3 способами: перечислением всех цепочек языка; указанием способа (алгоритма) порождения цепочек; определением метода (алгоритма) распознавания цепочек. | 40. **Теория форм. языков. Определение порождающей грамматики. Определение** **вывода, левосторонний и правосторонний выводы, дерево вывода. Примеры.**    Язык, порождаемый грамматикой – это множество всех выводимых из аксиомы грамматики терминальных цепочек.  терминальный символ, принадлежащий алфавиту языка;  нетерминальный символ, опред. название правила. |

|  |  |  |  |
| --- | --- | --- | --- |
| 41. **Теория форм. языков: способы задания грамматик. Общая хар-ка формы Бэкуса-Наура. Расширенная БНФ. Примеры.**  Язык L можно определить 3 способами: перечислением всех цепочек языка; указанием способа (алгоритма) порождения цепочек; определением метода (алгоритма) распознавания цепочек.  Для задания схем грамматик используются различные формы описания: символическая; форма Бэкуса-Наура; итерационная форма; синтаксические диаграммы.  Цепочки языка могут содержать метасимволы, имеющие особое назначение. Метаязык, предложенный Бэкусом и Науром (БНФ), использует следующие обозначения  - символ «::=» отделяет левую часть правила от правой (читается: «определяется как»);  - нетерминалы обозначаются произвольной символьной строкой, заключенной в угловые скобки «<» и «>»;  - терминалы – это символы, используемые в описываемом языке;  - правило может определять порождение нескольких альтернативных цепочек, которые отделяются друг от друга символом вертикальной черты «|» (читается: «или»).  Форма Бэкуса — Наура (сокр. БНФ, Бэкуса — Наура форма) – формальная система описания синтаксиса, в которой одни синтаксические категории последовательно определяются через другие категории.  Расширенная БНФ-нотация включает две конструкции, полезные при спецификации практических языков программирования.  Первая конструкция с помощью фигурных скобок позволяет описать повторение 0 или произвольное число раз некоторой цепочки. Вторая конструкция с помощью квадратных скобок определяет опцию – необязательный элемент. | 42. **Класс-ия языков и грамматик по Хомскому: иерархия Хомского (неогран., контекстно-завис., контекстно-независ., регулярные грамматики и вид правил этих грамматик), соотн-ия грамматик и порожд. ими языков.**  Хомский Ноам: 1928, США, лингвист, профессор Массачусетского технологического института, автор классификации формальных языков (иерархия Хомского), ввел понятие порождающей грамматики (1950).  T – алфавит (множество) терминальных символов — терминалов,  N – алфавит (множество) нетерминальных символов — нетерминалов,  P – конечное множество продукций (правил) грамматики  S - начальный символ (источник).  **Иерархия Хомского:**  *G*0 ⊃ *GI* ⊃ *GII* ⊃ *GIII (*множества грамматик типа 0, 1, 2 и 3).  **Грамматики типа 0:** *G*0 = *T*, *N*,*P*,*S* – **неограниченные** грамматики, у которых нет никаких ограничений для правил.  Правила имеют вид: α → βα∈*V* + β∈*V*\*. Для грамматики без ограничений на вид правил такой алгоритм распознавания в общем случае построить нельзя.  **Грамматики типа 1:** *G1 = <T N P S>* – **контекстно-зависимые** (КЗ) грамматики.  Правила контекстно-зависимых грамматик имеют вид:  α1*A*α2 →α1βα2 , где α1,α2 ∈*V* \* , *A*∈*VN* , β∈*V* + .  Правила неукорачивающих грамматик имеют вид:  α → β, где α∈*V* + , β∈*V* + и α ≤ β .  **Контекстно-зависимая грамматика**: один и тот же нетерминальный символ может быть заменен на ту или иную цепочку символов в зависимости от контекста (цепочки) в которой они встречаются. В **неукорачивающих грамматиках** любая цепочка символов может быть заменена на цепочку символов не меньшей длины.  **Грамматики типа 2:** *G2 = <T N P S>* – **контекстно-свободные** (КС) грамматики.  Правила имеют вид: *A*→α, где *A*∈*N* , α∈*V* \* , где A — нетерминал, β — цепочка нетерминалов и терминалов.  5) **Грамматики типа 3:** *G3 = <T N P S>* – **регулярные грамматики**. Регулярные грамм-ки бывают праволинейными и леволинейными. Правила праволинейной грамматики имеют вид:  *A*→αили *A*→α*B*, где *A*,*B*∈ *N* , α ∈*T*\*.  Правила леволинейной грамматики имеют вид:  *A*→αили *A*→ *B*α, где *A*,*B*∈ *N* , α ∈*T*\*.  Тип 3: в левой части только нетерминалы (большие буквы), в правой части терминалы/нетерминалы + “плавный” переход (a->Ba, C->Cb…).  Тип 2: в левой части только нетерминалы, в правой части терминалы смешиваются с нетерминалами с “резким” переходом (aF->B, Cb->aab…).  Тип 1: в левой части пересечение терминалов с нетерминалами, в правой части терминалы с нетерминалами (или пустой символ). При этом длина левой части должна быть меньше или равна правой (a->ab, c->a).  Тип 0: без каких - либо ограничений. | 43. **Лекс. анализ: определение, назначение, применение. Лекс. анализатор: место в структуре транслятора, входная и выходная информация, способы взаимодействия с синтакс. анализатором (последовательное и параллельное).**  Лексический анализ – первая (наиболее простая) фаза трансляции. Лексический анализ выполняется программой (входящей в состав транслятора), называемой лексическим анализатором (сканером).    С точки зрения лексического анализатора – язык программирования набор лексем (токенов), которые распознаются (классифицируются) лексическим анализатором.  Язык программирования на уровне лексического анализа представляет собой регулярный язык (язык типа 3 иерархии Хомского).    На вход лекс. анализатора поступает текст исх. программы, а выходная инфа передается для дальнейшей обработки компилятором на этапе синтаксического анализа и разбора. | 44. **Лекс. анализ: определение, назначение, применение. Лекс. анализатор: вид правил регулярной грамматики, нисход. и восход. разбор цепочек, дерево разбора.**  Лексический анализ – первая (наиболее простая) фаза трансляции. Лексический анализ выполняется программой (входящей в состав транслятора), называемой лексическим анализатором (сканером).    С точки зрения лексического анализатора, ЯП - это набор лексем (токенов), кот. распознаются (классифицируются) лексическим анализатором.  ЯП на уровне лексического анализа представляет собой регулярный язык (язык типа 3 иерархии Хомского).  Грамматика языка описывает множество правильных цепочек символов над заданным алфавитом.  Для описания регулярных языков используют другую форму описания – регулярные выражения.  Дерево разбора грамматики - дерево, в вершинах которого записаны терминалы или нетерминалы. Все вершины, помеченные терминалами, являются листьями.  Нисходящий:    Восходящий: |
| 45. **Регулярные выражения: рекурсивное определение, назначение, применение, метасимволы, примеры. Пример применения в функциях стандартной библиотеке STL C++.**  Регулярное выражение описывает множество цепочек – формальный язык. Для записи регулярного выражения используются метасимволы. Множество цепочек, описанных регулярным выражением, называется регулярным множеством (или регулярным языком).  Пусть – алфавит.Регулярные выражения над алфавитом и языки, представляемые ими, рекурсивно определяются следующим образом:    Символы, применяемые для описания регулярных выражений, называются **метасимволами** или **символами-джокерами**. Джокерами являются символы: \*,+ ,+,(,),∅. | 46. **Конечный автомат (КА): определение, назначение, схема работы КА, примеры.**      Конечные автоматы обычно используются для организации и представления потока выполнения чего-либо. Это особенно полезно при реализации ИИ в играх. Например, для написания «мозга» врага: каждое состояние представляет собой какое-то действие (напасть, уклониться и т. д.).  Конечный автомат (КА) в теории алгоритмов — математическая абстракция, модель дискретного устройства, имеющего один вход, один выход и в каждый момент времени находящегося в одном состоянии из множества возможных. Является частным случаем абстрактного дискретного автомата, число возможных внутренних состояний которого конечно.  При работе на вход КА поступают последовательно входные воздействия, а на выходе КА формирует выходные сигналы. Обычно под входными воздействиями принимают подачу на вход автомата символов одного алфавита, а на выход КА в процессе работы выдаёт символы в общем случае другого, возможно даже не пересекающегося со входным, алфавита.  Помимо конечных автоматов существуют и бесконечные дискретные автоматы — автоматы с бесконечным числом внутренних состояний, например, машина Тьюринга. | 47. **Конечный автомат: недетерминированные и детерминированные КА. Мгновенное описание КА. Последовательность мгновенных описаний, диаграмма мгновенных описаний. Пример.**    Различают детерминированные КА — автоматы, в которых следующее состояние однозначно определяется текущим состоянием и выход зависит только от текущего состояния и текущего входа, и недетерминированные КА, следующее состояние у которых в общем случае неопределённо и, соответственно, не определён выходной сигнал. Если переход в последующие состояния происходит с некоторыми вероятностями, то такой КА называют вероятностным КА. | 48. **Конечный автомат: определение, схема работы, примеры. Соотношение регулярного языка, регулярной грамматики, регулярного языка и конечного автомата.**    Любой регул. язык может быть задан регул. грамматикой, регулярным выражением или конечным автоматом. |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 49. **Конечный автомат: определение графа переходов конечного автомата и метод его построение по регулярному выражению. Примеры построения графов переходов**  Конечный автомат может быть однозначно задан своим графом переходов.    Метка ребра (si, sj) – все *a* , для которых *sj*(*si*, *a*). | 50. **Конечный автомат: алгоритм разбора цепочки символов, основанный на двух массивах.**    Один массив вспомогательный (почему 2). Количество ячеек == количеству состояний. Изначально инициализируем -1. В последнем состоянии должна получиться цифра == количеству символов в цепочке. Во втором массиве индексы на единицу больше, чем номер шага. | 51. **Синтакс. анализ: опред-е, назначение, применение. Синтаксический анализатор: место в структуре транслятора, входная и выходная информация, способы взаимодействия с лекс. анализатором (последовательное и параллельное)**  Синтаксический анализ: вторая фаза трансляции.    Синтакс. анализ выполняется после фазы лекс. анализа и предназначен для распознавания синтакс. конструкций и формирования промежут. кода.  Синтакс. анализ: основная фаза трансляции. Без нее процесс трансляции не имеет смысла. Все задачи лекс. анализа могут быть решены в рамках синтакс. анализа. Т.е. можно создать транслятор без лекс. анализатора. Лекс. анализ необходим для освобождения алгоритма синтакс. разбора от рутинных алгоритмов.  Программа, выполняющая синтакс. анализ называется синтакс. анализатором. Вход синтакс. анализатора: таблица лексем (ТЛ); таблица идентификаторов (ТИ). Выход синтакс. анализатора: дерево разбора. | 52. **Привед-е контекстно-свободной грамматики: удаление бесплодных символов, недостижимых символов, λ-правил, цепных правил.**      **Алгоритм удаления бесплодных символов**  ***Рекурсивно*** строим множества *N*0, *N*1, *N*2,...  1) *N*0 = ∅  2) *N*1 ={*A*| (*A* → α)∈ *P* ∧ α∈(*N*0 ∪*T*)\*}∪ *N*0  3) если *N*1 ≠ *N*0, то переход на шаг 4  иначе *G*′ = (*T*, *N*1,*P*′,*S*) ,  где *P*′ – правила из *P*, содержащие только символы *V* ′ = *N*1 ∪*T*  4) *N*2 ={*A*| (*A* → α)∈ *P* ∧ α∈(*N*1 ∪*T*)\*}∪ *N*1  5) если *N*2 ≠ *N*1, то переход на шаг 6  иначе *G*′ = (*T*, *N*2,*P*′,*S*), где *P*′ – правила из *P*, содержащие только символы *V* ′ = *N*2 ∪*T*  6) *N*3 ={*A*| (*A* → α)∈ *P* ∧ α∈(*N*2 ∪*T*)\*}∪ *N*2  7) если *N*3 ≠ *N*2 , то переход на шаг 8  иначе *G*′ = (*T*, *N*3,*P*′,*S*), где *P*′ – правила из *P*, содержащие только символы *V* ′ = *N*3 ∪*T*  Непост. Символы:  Первоначально в это множество входит только стартовый (целевой) символ *S* грамматики, затем множество пополняем на основе правил грамматики. Все символы, которые не войдут в это множество, являются недостижимыми и могут быть исключены в новой грамматике из словаря и из правил.  1) *V*0 ={*S*0} , ***i*** =1   |  |  | | --- | --- | | 2) | , |   *Vi* ={*x*|*x*∈(*N* ∪*T*) *и* (*A*→α*x*β)∈*P*, *A*∈*Vi*−1}∪*Vi*−1где  α,β∈(*N* ∪*T*)\*  3) если *Vi* ≠ *Vi*−1, то i=i+1 и переход на шаг 2,  иначе *G*′ = (*T*′, *N*′,*P*′,*S*), где *N*′ = *N* ∩*Vi* ,*T*′ = *T* ∩*Vi P*′ ⎯правила из *P*, содержащие только символы *V* |
| 53. **Автомат с магазинной памятью: определение, назначение, схема работы, конфигурация (текущее состояние автомата), алгоритм работы МП-автомата, пример.**    Конфигурация автомата (текущее состояние) описывается тройкой: (*q*,α,ω) *q* – текущее состояние автомата; α – остаток цепочки. Первый символ этой цепочки просматривается входной головкой автомата. Если α= {λ}, то входной символ прочитан; ω – цепочка-содержимое магазина (стека). Если ω= {λ}, то магазин пустой. |