Учреждение образования
«Брестский государственный технический университет»
Факультет электронно-информационных систем
Кафедра интеллектуальные информационные технологии

СОГЛАСОВАНО	СОГЛАСОВАНО
Заведующий кафедрой	Декан факультета
В.А.Головко	А.Н.Парфиевич
«06» 12 2024г.	«06» 12 2024г.

ЭЛЕКТРОННЫЙ УЧЕБНО-МЕТОДИЧЕСКИЙ КОМПЛЕКС ПО УЧЕБНОЙ ДИСЦИПЛИНЕ «ОСНОВЫ АЛГОРИТМИЗАЦИИ И ПРОГРАММИРОВАНИЯ»

для специальностей:
6-05-0612-03 Системы управления информацией,
6-05-0611-03 Искусственный интеллект,
6-05-0612-01 Программная инженерия

Составитель: Хацкевич М.В., старший преподаватель

Расмотрено и утверждено на заседании Научно – методического Совета университета от «27» декабря 2024г., протокол №2

Рег. №УМК24/25-22

Пояснительная записка

Актуальность изучения дисциплины

Образовательный стандарт Республики Беларусь для специальностей 6-05-0612-03 Системы управления информацией,
6-05-0611-03 Искусственный интеллект, 6-05-0612-01 Программная инженерия предполагает изучение дисциплины «Основы алгоритмизации и программирования».

Дисциплина «Основы алгоритмизации и программирования» позволяет сформировать у студента способность к анализу информации и/или сущности задачи. Студенты обучаются процессу постановки формальной цели и выбору путей ее достижения. Также студенты приобретают умение самостоятельно разработать алгоритм решения задачи, что является неотьемлемой частью работы инженера.

Цель и задачи дисциплины

Цель преподавания учебной дисциплины: подготовка специалиста, уверенно владеющего возможностями, предоставляемыми современными компьютерными технологиями в среде программирования на алгоритмическом языке высокого уровня, а также программирования вычислительных алгоритмов. Освоение базовых методов структурного и функционально-ориентированного программирования.

Задачи изучения дисциплины:

Задачами изучаемой дисциплины являются:

- изучение языка программирования высокого уровня, а также приобретение практических навыков составления и отладки программ на персональных компьютерах;
- приобретение навыков алгоритмизации на примерах решения вычислительных задач и их закрепление на основе программирования алгоритмов обработки структур данных и алгоритмов вычислительной математики;
- приобретение знаний об эффективности разрабатываемых алгоритмов, оценке их временных и вычислительных ресурсов.

Электронный учебно-методический комплекс (ЭУМК) объединяет структурные элементы учебно-методического обеспечения образовательного процесса, и представляет собой сборник материалов теоретического и практического характера для организации работы студентов по специальностям 6-05-0611-03 Искусственный интеллект (дневная форма получения образования), 6-05-0612-01 Программная инженерия (дневная форма получения образования), 6-05-0612-03 Системы управления

информацией (дневной и заочной форм получения образования) по изучению дисциплины «Основы алгоритмизации и программирования».

ЭУМК разработан на основании Положения об учебно-методическом комплексе на уровне высшего образования, утвержденного Постановлением Министерства образования Республики Беларусь от 26 июля 2011 г., № 167, и предназначен для реализации требований учебной программы по учебной дисциплине «Основы алгоритмизации и программирования» для специальностей 6-05-0612-03 Системы управления информацией, 6-05-0611-03 Искусственный интеллект, 6-05-0612-01 Программная инженерия. ЭУМК разработан в полном соответствии с утвержденной учебной программой по учебной дисциплине компонента учреждения высшего образования «Основы алгоритмизации и программирования».

Цели ЭУМК:

- обеспечение качественного методического сопровождения процесса обучения;
- организация эффективной самостоятельной работы студентов.

Содержание и объем ЭУМК полностью соответствуют образовательному стандарту высшего образования специальностей 6-05-0612-03 Системы управления информацией, 6-05-0611-03 Искусственный интеллект, 6-05-0612-01 Программная инженерия, а также учебно-программной документации образовательных программ высшего образования. Материал представлен на требуемом методическом уровне и адаптирован к современным образовательным технологиям.

Структура электронного учебно-методического комплекса по дисциплине «Основы алгоритмизации и программирования»:

Теоретический раздел ЭУМК содержит материалы для теоретического изучения учебной дисциплины и представлен конспектом лекций.

Практический раздел ЭУМК содержит: материалы для проведения лабораторных работ в виде методических указаний и индивидуальных заданий; методические указания для выполнения курсовой работы.

Раздел контроля знаний ЭУМК содержит примерный перечень вопросов, выносимых на экзамен, позволяющих определить соответствие результатов учебной деятельности обучающихся требованиям образовательных стандартов высшего образования и учебно-программной документации образовательных программ высшего образования.

Вспомогательный раздел включает учебную программу по дисциплине «Основы алгоритмизации и программирования».

Рекомендации по организации работы с ЭУМК:

- лекции проводятся с использованием представленных в ЭУМК теоретических материалов, часть материала представляется с

использованием персонального компьютера и мультимедийного проектора;

- при подготовке к экзамену, выполнению и защите лабораторных работ и курсовой работы студенты могут использовать конспект лекций;
- выполнение лабораторных работ и курсовой работы проводится с использованием представленных в ЭУМК методических указаний;
- экзамен проводится в письменной форме. Вопросы к экзамену за два семестра приведены в разделе контроля знаний.

ЭУМК способствует успешному усвоению студентами учебного материала, дает возможность планировать и осуществлять самостоятельную работу студентов, обеспечивает рациональное распределение учебного времени по темам учебной дисциплины и совершенствование методики проведения занятий.

ПЕРЕЧЕНЬ МАТЕРИАЛОВ В КОМПЛЕКСЕ

1 ТЕОРЕТИЧЕСКИЙ РАЗДЕЛ 8
Раздел 1 ВВЕДЕНИЕ. ОСНОВНЫЕ ПОНЯТИЯ 8
Лекция 1 Общие сведения об алгоритмах, ЭВМ и организации вычислительного процесса 8
Лекция 2 Способы представления данных в ЭВМ 23
Лекция 3 Общие сведения о программном обеспечении и технологии производства программного обеспечения 38
Лекция 4 Интерфейс. Общие сведения 48
РАЗДЕЛ 2. ОСНОВЫ ПРОГРАММИРОВАНИЯ НА ЯЗЫКЕ С++ 51
Лекция 5 Среда программирования 51
Лекция 6 Основные понятия. Типы данных 79
Лекция 7 Операции, операторы и выражения 97
Лекция 8 Структурированные типы данных 114
Лекция 9 Процедуры и функции 132
Лекция 10 Дополнительные возможности 139
Раздел 3 СТАНДАРТНЫЕ АЛГОРИТМЫ 143
Лекция 11 Сортировка и поиск 143
Раздел 4 АДРЕСНАЯ АРИФМЕТИКА. ДИНАМИЧЕСКАЯ ПАМЯТЬ. ДИНАМИЧЕСКИЕ СТРУКТУРЫ 162
Лекция 12 Указатели и адреса 162
Лекция 13 Указатели и функции 171
Лекция 14 Структуры данных 175
Лекция 15 Массивы указателей и структуры 187
Лекция 16 Динамические линейные структуры 193
Лекция 17 Динамические нелинейные структуры 209
2 ПРАКТИЧЕСКИЙ РАЗДЕЛ 225
1.Методические указания и индивидуальные задания к выполнению лабораторных работ по дисциплине «Основы алгоритмизации и программирования» для студентов дневной и заочной форм обучения специальности 6-05-0612-03 Системы управления информацией, и для студентов дневной формы обучения по специальностям 6-05-0611-03 Искусственный интеллект, 6-05-0612-01 Программная инженерия. 225
2. ЛАБОРАТОРНЫЕ РАБОТЫ 226
Лабораторная работа №1 «Разработка схем алгоритмов для линейных и разветвляющихся процессов в соответствии с положениями действующих стандартов. Разработка схем алгоритмов для циклических процессов в соответствии с положениями действующих стандартов. Разработка структурированных схем алгоритмов» 227
Лабораторная работа №2 «Знакомство со средой программирования» . 242
Лабораторная работа №3 «Разработка, отладка и выполнение простейшей программы» 245

Лабораторная работа №4 «Разработка алгоритма, составление, отладка и выполнение программы с ветвлением (выбором вариантов)»................ 246
Лабораторная работа №5 «Разработка алгоритма, составление, отладка и выполнение циклической программы с известным числом повторений. Разработка алгоритма, составление, отладка и выполнение программы с использованием итерационных циклов. Разработка и выполнение программы с использованием разветвлений и вложенных циклов»....... 247
Лабораторная работа №6 «Разработка алгоритма, составление, отладка и выполнение программы обработки сложных типов данных (массивы)»247
Лабораторная работа №7 «Разработка алгоритма, составление, отладка и выполнение программы обработки сложных типов данных (строки)» .. 248
Лабораторная работа №8 «Разработка алгоритма, составление, отладка и выполнение программы с использованием пользовательских функций. Разработка алгоритма, составление, отладка и выполнение программы с использованием рекурсивных функций. Разработка алгоритма, составление, отладка и выполнение программы с использованием функций с произвольным числом параметров» 248
Лабораторная работа №9 «Разработка, отладка и выполнение программы с использованием модулей пользователя»... 250
Лабораторная работа №10 «Разработка, отладка и выполнение программы обработки текстовых файлов».. 250
Лабораторная работа №11 «Разработка, отладка и выполнение программы обработки файлов с типом» 251
Лабораторная работа №12 «Разработка алгоритмов, составление, отладка и выполнение программы сортировки и поиска (массивы, строки)»...... 252
Лабораторная работа №13 «Разработка, отладка и выполнение программы с использованием подпрограмм с различными типами параметров» 253
Лабораторная работа №14 «Разработка алгоритма, составление, отладка и выполнение программы с использованием структур (массивов структур)» 253
Лабораторная работа №15 «Разработка алгоритма, составление, отладка и выполнение программы обработки линейных связанных списков».... 254
Лабораторная работа №16 «Программирование с использованием древовидных структур данных».. 254
3.Методические указания к выполнению курсовой работы по дисциплине «Основы алгоритмизации и программирования» для студентов дневной и заочной форм обучения специальности 6-05-0612-03 Системы управления информацией, и для студентов дневной формы обучения по специальностям 6-05-0611-03 Искусственный интеллект, 6-05-0612-01 Программная инженерия. 255
3. РАЗДЕЛ КОНТРОЛЯ ЗНАНИЙ... 272

3.1.Перечень вопросов, выносимых на экзамен за 1 семестр по дисциплине «Основы алгоритмизации и программирования».. 272
3.2.Перечень вопросов, выносимых на экзамен за 2 семестр по дисциплине «Основы алгоритмизации и программирования».. 274
4 ВСПОМОГАТЕЛЬНЫЙ РАЗДЕЛ... 278
1.Учебная программа учреждения высшего образования по учебной дисциплине для специальностей 6-05-0612-03 «Системы управления информацией», 6-05-0611-03 «Искусственный интеллект», 6-05-0612-01«Программная инженерия».. 278

1 ТЕОРЕТИЧЕСКИЙ РАЗДЕЛ

Конспект лекций по дисциплине
«Основы алгоритмизации и программирования»

1-ый семестр

Раздел 1 ВВЕДЕНИЕ. ОСНОВНЫЕ ПОНЯТИЯ

Лекция 1 Общие сведения об алгоритмах, ЭВМ и организации вычислительного процесса

Алгоритм - это система точных и понятных предписаний о содержании и последовательности выполнения конечного числа действий, необходимых для решения любой задачи данного типа. Примеры: правила сложения, умножения, решения алгебраических уравнений и т.п.

Алгоритмизация - это процесс разработки и описания последовательности шагов, которые необходимо выполнить для решения определенной задачи или достижения конкретной цели. Алгоритмизация является ключевым этапом при программировании и разработке программного обеспечения.

При алгоритмизации задачи создаются четкие инструкции, которые компьютер может понять и выполнять. Алгоритмы могут быть записаны в виде текстового описания, блок-схемы, псевдокода или других формализованных представлений. Они служат основой для написания кода программы, который позволяет компьютеру автоматически решать задачи в соответствии с предварительно разработанными инструкциями.

Классы алгоритмов.

1.Вычислительные алгоритмы, работающие со сравнительно простыми видами данных, такими как числа и матрицы, хотя сам процесс вычисления может быть долгим и сложным.

2.Информационные алгоритмы, представляющие собой набор сравнительно простых процедур, работающих с большими объемами информации (алгоритмы баз данных).

3.Управляющие алгоритмы, генерирующие различные управляющие воздействия на основе данных, полученных от внешних процессов, которыми алгоритмы управляют.

Программное и аппаратное обеспечение.

Программное обеспечение - это программы и драйверы устройств, предназначенные для работы в конкретной системе. Аппаратное обеспечение системы включает карты адаптеров или другие устройства, обеспечивающие связь между программным обеспечением системы и физической сетью.

Аппаратное обеспечение представляет собой совокупность технических средств. Обычно это электронные и механические устройства. Такие устройства позволяют обеспечить как нормальное функционирование каких-либо систем, так и расширяющих их основные функции.

Языки программирования: уровень и тип языка программирования, характеристики.

Язык программирования — формальный язык, предназначенный для записи компьютерных программ. Язык программирования определяет набор лексических, синтаксических и семантических правил, определяющих внешний вид программы и действия, которые выполнит исполнитель (обычно - ЭВМ) под её управлением.

Язык программирования предназначен для написания компьютерных программ, которые представляют собой набор правил, позволяющих компьютеру выполнить тот или иной вычислительный процесс, организовать управление различными объектами, и т. п.

Большинство языков программирования использует специальные конструкции для определения и манипулирования структурами данных и управления процессом вычислений.

Как правило, язык программирования определяется не только через спецификации стандарта языка, формально определяющие его синтаксис и семантику, но и через реализации стандарта - программные средства, обеспечивающие трансляцию или интерпретацию программ на этом языке.

Языки низкого и высокого уровня.

Формально язык программирования не имеет такого критерия как «уровень». Условно этот термин обычно означает одно из двух:

- «приближенность» языка программирования к естественному человеческому языку и образу мысли;
- «удалённость» семантики языка программирования от машинного кода целевой архитектуры процессора - то есть наименьший масштаб преобразований, которые должен претерпеть код программы перед тем, как он сможет исполняться.

По степени «высокоуровневости» языки принято делить на пять поколений.

К первому поколению относят, в первую очередь, машинные языки (машинные коды), то есть языки, реализованные непосредственно на аппаратном уровне.

Появившиеся вскоре после них «языки ассемблера» относят ко второму поколению. Многие языки ассемблера включают макроязык. Оба первых поколения общепринято относить к языкам низкого уровня.

К 1970-м годам сложность программ выросла настолько, что превысила способность программистов управляться с ними. Ответом на эту проблему стало появление массы языков высокого уровня, предлагающих самые разные способы управления сложностью. Программы на языках «высокого уровня» гораздо легче модифицируются и совсем легко переносятся с компьютера на компьютер.

На практике, наибольшее распространение получили языки третьего поколения, которые лишь претендуют на звание «высокоуровневых», но реально предоставляют лишь те «высокоуровневые» конструкции, что находят однозначное соответствие инструкциям в машине фон Неймана.

Более «высокоуровневыми» принято считать языки четвёртого и пятого поколения.

К четвёртому поколению относят языки запросов, языки опций и параметров, генераторы приложений, комбинированные пакеты баз данных. Наиболее значимой подгруппой в четвёртом поколении принято считать функциональные языки, большая часть из которых является языками высшего порядка.

Иногда выделяется категория языков пятого поколения, но она не является общепринятой — чаще используется термин «язык сверхвысокого уровня» (англ. very high level language). Это языки, реализация которых включает существенную алгоритмическую составляющую (то есть когда интерпретация небольшого исходного кода требует весьма сложных вычислений), поэтому порой также говорят, что языки пятого поколения — это фактически языки четвёртого поколения, дополненные базой знаний. Чаще всего так называют логические языки.

Язык Си является, вероятно, самым «низкоуровневым» в третьем поколении. Он изначально позиционировался как «высокоуровневый ассемблер» или «кроссплатформенный ассемблер». Классификация потомка Си, языка C++, вызывает споры: его нередко называют «высокоуровневым», несмотря на то, что технически его семантика и система типов мало отличаются от тех, на которых основан Си.

Компилируемые, интерпретируемые и встраиваемые языки.

Можно выделить три принципиально разных способа реализации языков программирования: компиляция, интерпретация и встраивание.

Компиляция означает, что исходный код программы сначала преобразуется в целевой (машинный) код специальной программой, называемой компилятором - в результате получается исполнимый модуль, который уже может быть запущен на исполнение как отдельная программа.

Интерпретация же означает, что исходный код выполняется непосредственно, команда за командой,— так что программа просто не может быть запущена без наличия интерпретатора.

Встраивание языка можно философски рассматривать как «реализацию без трансляции» — в том смысле, что такой язык является синтаксическим и семантическим подмножеством некого другого языка, без которого он не существует.

Реализация некоторых языков, например, Java и C#, занимают промежуточную ступень между компиляцией и интерпретацией. А именно, программа компилируется не в машинный язык, а в машинно-независимый код низкого уровня, байт-код. Далее байт-код выполняется виртуальной машиной. Для выполнения байт-кода обычно используется интерпретация, хотя отдельные его части для ускорения работы программы могут быть транслированы в машинный код непосредственно во время выполнения программы по технологии компиляции «на лету» (Just-in-time compilation, JIT). Для Java байт-код исполняется виртуальной машиной Java (Java Virtual Machine, JVM), для C# - Common Language Runtime. Подобный подход в некотором смысле позволяет использовать плюсы как интерпретаторов, так и компиляторов.

Компилируемые: C, C++, Pascal.

Интерпретируемые: Visual Basic Script (VBScript), JavaScript, Python, PHP

Условно компилируемые: C# и остальные языки .Net, Java для Java-машины.

Краткий обзор парадигм программирования: процедурные языки, обьектно - ориентированные языки.

Парадигма программирования - совокупность идей и понятий, определяющих стиль написания компьютерных программ. Это способ концептуализации, определяющий организацию вычислений и структурирование работы, выполняемой компьютером.

Процедурное программирование - программирование на императивном языке, при котором последовательно выполняемые операторы можно собрать в подпрограммы, то есть более крупные целостные единицы кода, с помощью механизмов самого языка.

Выполнение программы сводится к последовательному выполнению операторов с целью преобразования исходного состояния памяти, то есть значений исходных данных, в заключительное, то есть в результаты.

Процедурный язык программирования предоставляет возможность программисту определять каждый шаг в процессе решения задачи. Особенность таких языков программирования состоит в том, что задачи разбиваются на шаги и решаются шаг за шагом. Используя процедурный

язык, программист определяет языковые конструкции для выполнения последовательности алгоритмических шагов. Важным шагом в развитии процедурного программирования стал переход к структурной парадигме, возникшей благодаря открытию возможности создавать тьюринг-полные программы без оператора безусловного перехода (теорема Бёма — Якопини).

Объектно - ориентированное программирование (ООП) - методология или стиль программирования на основе описания типов/моделей предметной области и их взаимодействия, представленных порождением из прототипов или как экземпляры классов, которые образуют иерархию наследования.

Основные принципы ООП:

1. Абстракция данных. Абстрагирование означает выделение значимой информации и исключение из рассмотрения незначимой. В ООП рассматривают лишь абстракцию данных (нередко называя её просто «абстракцией»), подразумевая набор наиболее значимых характеристик объекта, доступных остальной программе.

2. Инкапсуляция. Инкапсуляция - свойство системы, позволяющее объединить данные и методы, работающие с ними, в классе.

3. Наследование. Наследование - свойство системы, позволяющее описать новый класс на основе уже существующего с частично или полностью заимствованной функциональностью. Класс, от которого производится наследование, называется базовым, родительским или суперклассом. Новый класс — потомком, наследником, дочерним или производным классом.

4. Полиморфизм подтипов. Полиморфизм подтипов (в ООП называемый просто «полиморфизмом») — свойство системы, позволяющее использовать объекты с одинаковым интерфейсом без информации о типе и внутренней структуре объекта. Другой вид полиморфизма - параметрический - в ООП называют обобщённым программированием.

Основные понятия:

1. Класс — универсальный, комплексный тип данных, состоящий из тематически единого набора «полей» (переменных более элементарных типов) и «методов» (функций для работы с этими полями), то есть он является моделью информационной сущности с внутренним и внешним интерфейсами для оперирования своим содержимым (значениями полей).

2.Объект. Сущность в адресном пространстве вычислительной системы, появляющаяся при создании экземпляра класса (например, после запуска результатов компиляции и связывания исходного кода на выполнение).

Этапы разработки программного обеспечения.

Процесс разработки программного обеспечения (англ. software development process) - процесс, посредством которого потребности пользователей преобразуются в программный продукт. Процесс разработки

программного обеспечения является составной частью программной инженерии.

Процесс разработки состоит из множества подпроцессов, или дисциплин, некоторые из которых перечислены ниже. Процесс - совокупность взаимосвязанных или взаимодействующих видов деятельности, преобразующих входы в выходы.

Этапы разработки:

1. Анализ требований (спецификация программного обеспечения).

Анализ требований - часть процесса разработки программного обеспечения, включающая в себя сбор требований к программному обеспечению (ПО), их систематизацию, выявление взаимосвязей, а также документирование.

2. Проектирование программного обеспечения.

Проектирование программного обеспечения - процесс создания проекта программного обеспечения (ПО), а также дисциплина, изучающая методы проектирования.

3. Программирование.

Программирование - процесс создания и модификации компьютерных программ.

4. Тестирование программного обеспечения.

Тестирование программного обеспечения - процесс исследования, испытания программного продукта, имеющий своей целью проверку соответствия между реальным поведением программы и её ожидаемым поведением на конечном наборе тестов, выбранных определённым образом

5. Системная интеграция (System integration).

6. Установка программного обеспечения.

Установка программного обеспечения - процесс установки программного обеспечения на компьютер конечного пользователя.

7. Сопровождение программного обеспечения.

Сопровождение (поддержка) программного обеспечения - процесс улучшения, оптимизации и устранения дефектов программного обеспечения (ПО) после передачи в эксплуатацию. Сопровождение ПО — это одна из фаз жизненного цикла программного обеспечения, следующая за фазой передачи ПО в эксплуатацию. В ходе сопровождения в программу вносятся изменения, с тем, чтобы исправить обнаруженные в процессе использования дефекты и недоработки, а также для добавления новой функциональности, с целью повысить удобство использования и применимость ПО.

Жизненный цикл программного продукта.

Жизненный цикл программного обеспечения (ПО) - период времени, который начинается с момента принятия решения о необходимости создания программного продукта и заканчивается в момент его полного изъятия из эксплуатации.

Стандарты жизненного цикла ПО:

1. ГОСТ 34.601-90 (утратил силу в РФ с 01.12.2023).
2. ISO/IEC 15288:2015 Systems and software engineering — System life cycle processes.
3. ISO/IEC/IEEE 12207:2017 System and software engineering — Software life cycle processes.

Стандарт ГОСТ 34.601-90 предусматривал следующие стадии и этапы создания автоматизированной системы (АС):

1. Формирование требований к АС
    1. Обследование объекта и обоснование необходимости создания АС.
    2. Формирование требований пользователя к АС.
    3. Оформление отчета о выполнении работ и заявки на разработку АС.
2. Разработка концепции АС
    1. Изучение объекта.
    2. Проведение необходимых научно-исследовательских работ.
    3. Разработка вариантов концепции АС и выбор варианта концепции АС, удовлетворяющего требованиям пользователей.
    4. Оформление отчета о проделанной работе.
3. Техническое задание
    1. Разработка и утверждение технического задания на создание АС
4. Эскизный проект.
    1. Разработка предварительных проектных решений по системе и её частям.
    2. Разработка документации на АС и её части.
5. Технический проект
    1. Разработка проектных решений по системе и её частям.
    2. Разработка документации на АС и её части.
    3. Разработка и оформление документации на поставку комплектующих изделий.
    4. Разработка заданий на проектирование в смежных частях проекта.
6. Рабочая документация
    1. Разработка рабочей документации на АС и её части.
    2. Разработка и адаптация программ.
7. Ввод в действие
    1. Подготовка объекта автоматизации.
    2. Подготовка персонала.
    3. Комплектация АС поставляемыми изделиями (программными и техническими средствами, программно-техническими комплексами, информационными изделиями).

4. Строительно-монтажные работы.
5. Пусконаладочные работы.
6. Проведение предварительных испытаний.
7. Проведение опытной эксплуатации.
8. Проведение приёмочных испытаний.
8. Тестирование АС.
9. Сопровождение АС.
    1. Выполнение работ в соответствии с гарантийными обязательствами.
    2. Послегарантийное обслуживание.

Модели жизненного цикла

Модель жизненного цикла программного обеспечения характеризует подход команды к разработке программного продукта. Она отражает акценты и приоритеты во всём процессе изготовления программы, а самое главное, порядок следования этапов создания программных продуктов.

На сегодняшний день существует множество моделей жизненного цикла разработки программного продукта.

Каскадная (водопадная) модель

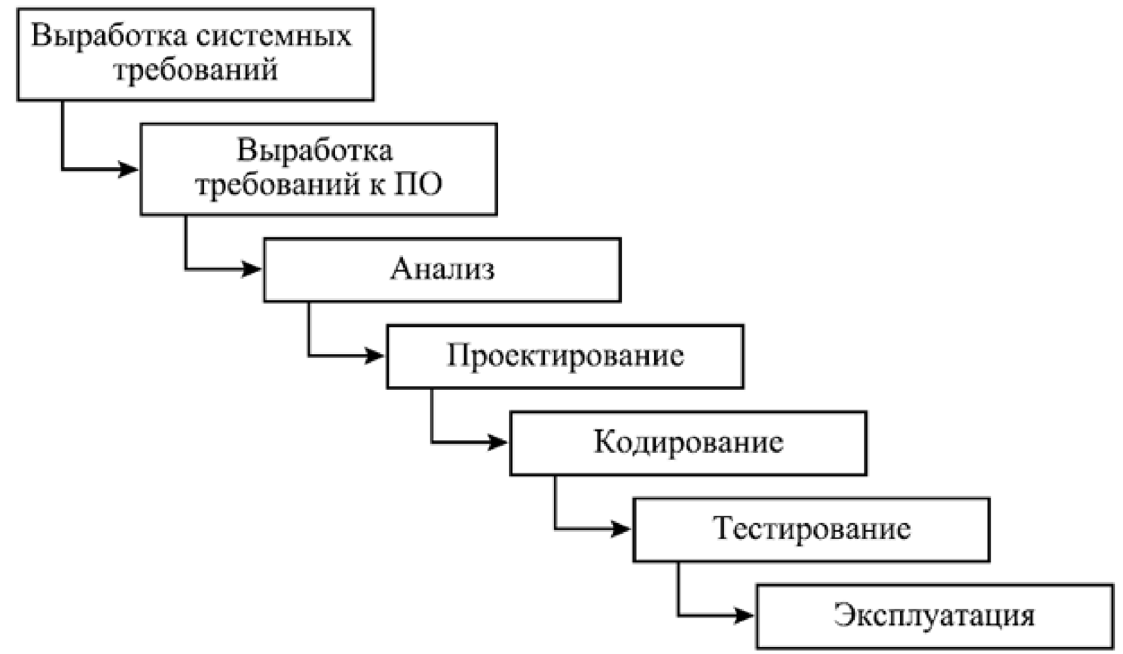


Рисунок 1 – Каскадная модель

Каскадная (водопадная) модель (рисунок 1) строго следует последовательности всех этапов разработки ПО и не предполагает возвращения с текущего этапа на предыдущий. Сейчас данная модель практически не используется, разве что в очень малых проектах.

V-образная модель разработки

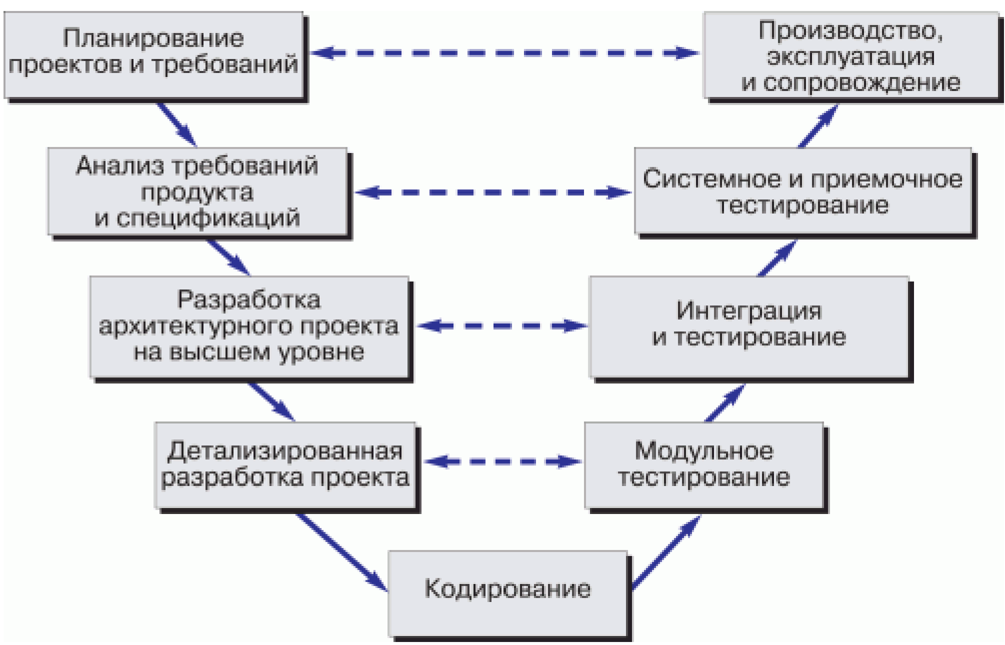


Рисунок 2 - V-образная модель

По рисунку можно проследить, что в **V-образной модели** имеется возможность вернуться на некоторые этапы разработки и уточнить нужные требования (рисунок 2).

Модель прототипирования

Рисунок 3 - Модель прототипирования

Прототипирование предполагает создание на протяжении всего процесса разработки несколько рабочих версий программы (прототипов) с неполным функционалом (рисунок 3). В первом прототипе может быть реализован исключительно один интерфейс приложения.

Модель быстрой разработки (RAD-модель)

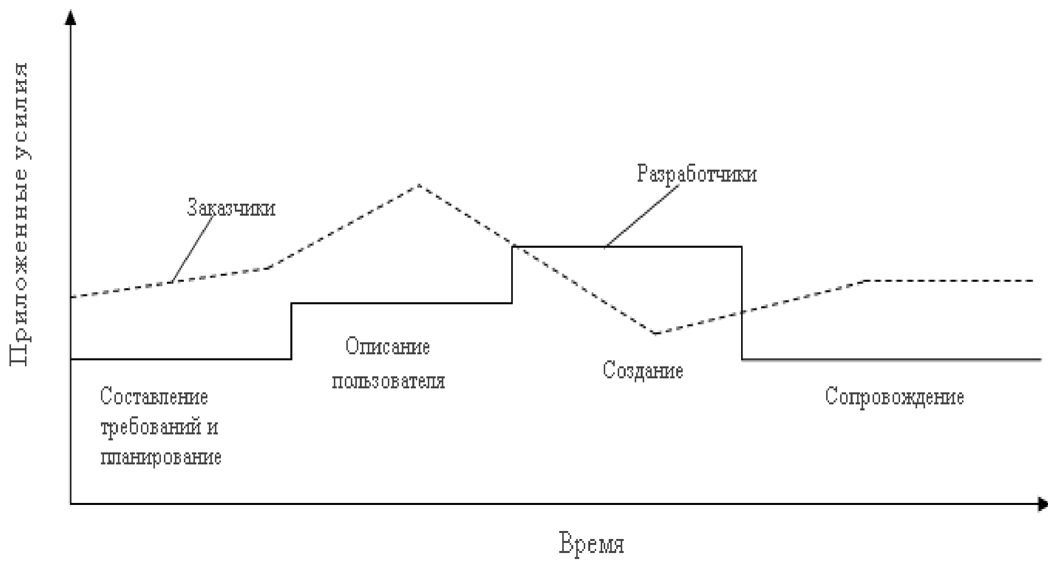


Рисунок 4 - RAD-модель

RAD-модель (rapid application development — **быстрая разработка приложений**) ориентирована в первую очередь на быстроту и удобство программирования. Команда делает акцент именно на разработке, а большая часть работы по составлению требований и описанию пользователей возлагается на заказчика(рисунок 4).

Итерационная модель

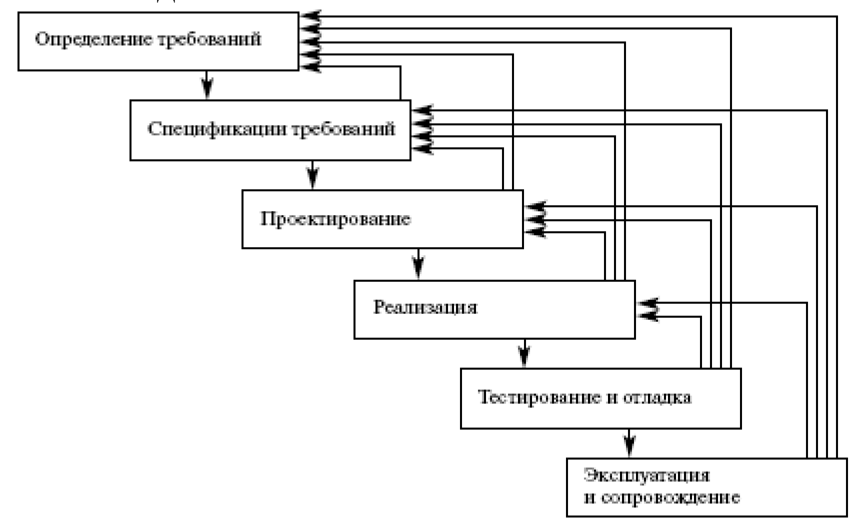


Рисунок 5 - Итерационная модель

В **итерационной модели** всегда имеется возможность вернуться на любой предыдущий этап разработки ПО для уточнений требований и исправления компонентов. Здесь главное вовремя остановиться, ведь итерации не могут продолжаться бесконечно.

Спиральная модель

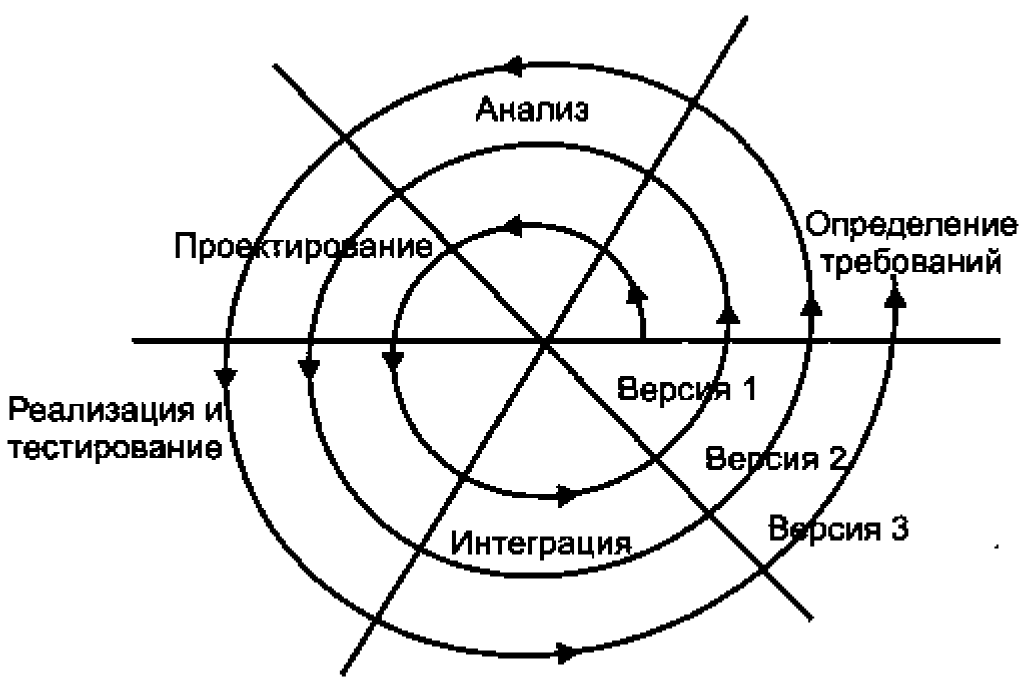


Рисунок 6 – Спиральная модель

В **спиральной модели** все этапы разработки последовательно повторяются по кругу до тех пор, пока текущая версия программы не станет полностью соответствовать требованиям. Здесь также нужно иметь предел и вовремя остановиться.

Гибкие методологии

Гибкие методологии (Agile) олицетворяют современные подходы к разработке ПО. Они используются обычно в небольших командах разработчиков. Среди них такие модели жизненного цикла программного продукта, как Scrum, DSDM, XP, FDD и другие.

Характеристики разрабатываемой программы.

Основными характеристиками разрабатываемых программ являются:

1. Алгоритмическая сложность (логика алгоритмов обработки информации).
2. Состав и глубина проработки реализованных функций обработки.
3. Полнота и системность функций обработки.
4. Объем файлов программ.
5. Требования к операционной системе и техническим средствам обработки со стороны программного средства.
6. Объем дисковой памяти.
7. Размер оперативной памяти для запуска программ.
8. Тип процессора.
9. Версия операционной системы.
10. Наличие вычислительной сети и др.

Характеристики качества программного обеспечения приведены в таблице 1

Таблица 1 – Критерии качества программного обеспечения

Фактор	**Означает**
Корректность (правильность)	Обеспечивает правильную обработку на правильных данных
Устойчивость	"Элегантно" завершает обработку ошибок
Расширяемость	Может легко адаптироваться к изменяющимся требованиям
Многократность использования	Может использоваться и в других системах, а не только в той, для которой было создано.
Совместимость	Может легко использоваться с другим программным обеспечением
Эффективность	Эффективное использование времени, компьютерной памяти, дискового пространства и т.д.
Переносимость	Можно легко перенести на другие аппаратные и программные средства
Верификация	Простота проверки, легкость разработки тестов при обнаружении ошибок, легкость обнаружения мест, где программа потерпела неудачу, и т.д.
Поддержка целостности	Защищает себя от неправильного обращения и неправильного употребления
Легкость использования	Для пользователя и для будущих программистов

Основные принципы обработки команд программы исполнителем (компьютером).

Компьютер или ЭВМ (электронно-вычислительная машина) – это универсальное техническое средство для автоматической обработки информации.

Аппаратное обеспечение компьютера – это все устройства, входящие в его состав и обеспечивающие его исправную работу.

В современном мире, компьютеры строятся по единой принципиальной схеме, основанной на фундаменте идеи программного управления Чарльза Бэббиджа(середина XIX в). Эта идея была реализована при создании первой ЭВМ ENIAC в 1946 году коллективом учёных и инженеров под руководством известного американского математика Джона фон Неймана, сформулировавшего следующие общие принципы:

1. Принцип программного управления. Из него следует, что программа состоит из набора команд, которые выполняются процессором автоматически друг за другом в определенной последовательности.

2. Принцип однородности памяти. Программы и данные хранятся в одной и той же памяти. Поэтому компьютер не различает, что хранится в данной ячейке памяти - число, текст или команда. Над командами можно выполнять такие же действия, как и над данными. Это открывает целый ряд возможностей. Например, программа в процессе своего выполнения также может подвергаться переработке, что позволяет задавать в самой программе

правила получения некоторых ее частей (так в программе организуется выполнение циклов и подпрограмм).

3. Принцип адресности. Структурно основная память состоит из пронумерованных ячеек; процессору в произвольный момент времени доступна любая ячейка. Отсюда следует возможность давать имена областям памяти, так, чтобы к значениям в них можно было впоследствии обращаться или менять их в процессе выполнения программ с использованием присвоенных имен.

Компьютер является универсальным исполнителем по обработке информации. Значит, для него, как для любого исполнителя, существует определённая система команд. Такая система команд для компьютера называется языком машинных команд.

Программа для компьютера - это алгоритм, разработанный на языке машинных команд. Или, Программа управления компьютером – это последовательность команд языка машинных команд, где каждая команда – директива для процессора на выполнение определённого действия.

Согласно принципам Джона фон Неймана, программа во время её исполнения и данные, которые она обрабатывает, находятся в оперативной памяти (принцип хранимой в памяти программы). Процессор исполняет программу начиная с первой команды и заканчивая последней.

Для компьютера вся информация должна быть представлена в двоичных кодах, т.е. необходим способ перевода. Такой способ перевода называется трансляцией, а выполняет это транслятор.

Организация ЭВМ. Принципы построения ЭВМ, машина Фон-Неймана.

Принципы организации вычислительного процесса, базируется на концепции Дж. Фон Неймана. Согласно этой концепции определена автономная работающая вычислительная машина, содержащая устройство управления, арифметико-логическое устройство (АЛУ), память и устройство ввода/вывода (рисунок 1.1). В связях, соединяющих устройства, выделены потоки данных, команд и управляющих сигналов. Преобразование данных осуществляется последовательно под централизованным управлением от программы, состоящей из команд. Набор команд составляет машинный язык низкого уровня.

Для организации вычислительной машины предложены следующие принципы:

1.Двоичное кодирование информации, разделение ее на слова фиксированной разрядности.

2. Линейно-адресная организация памяти (N ячеек по n разрядов). Аппаратные средства для записи и хранения и чтения слова из n двоичных разрядов называют ячейкой памяти. Ячейки пронумерованы по порядку (0,1, … , N-1). Номер ячейки - адрес. В командах программы адрес является

именем (идентификатором) переменной, хранящейся в соответствующей ячейке.

3. Представление алгоритма программой, состоящей из команд. Команда является предписанием, определяющим шаг процесса выполнения программы. Она содержит код операции, адреса операндов и другие служебные коды.

4. Хранение команд и данных в одной памяти. Различие их заключается только в способе использования и интерпретации считанного из памяти слова.

⇒ линии связи для данных; → линии связи для команд; — — → линии связи для управления

Рисунок 7 - Структура ЭВМ фон Неймана

5. Вычислительный процесс организуется как последовательное выполнение команд в порядке, определяемом программой.

6. Жесткость архитектуры – неизмеримость в процессе работы вычислительной машины, ее структуры, списка команд, методов кодирования данных.

При работе вычислительной машины наиболее интенсивное взаимодействие осуществляется между АЛУ и устройством управления. С развитием элементарной базы в целях повышения производительности за счет уменьшения задержек в связях эти устройства объединили в один блок, называемый процессором.

Устройство ввода преобразует входные сигналы в сигналы, принятые для представления данных на шине, соединяющей устройство ввода с АЛУ.

В памяти хранятся команды и данные, которыми оперируют процессор. В нее же записываются результаты промежуточных вычислений. Результаты выполнения программы поступают в устройство вывода.

Устройство вывода преобразует выходные сигналы в форму, удобную для восприятия человеком.

На рисунке 8 приведена функциональная организация типовой ЭВМ

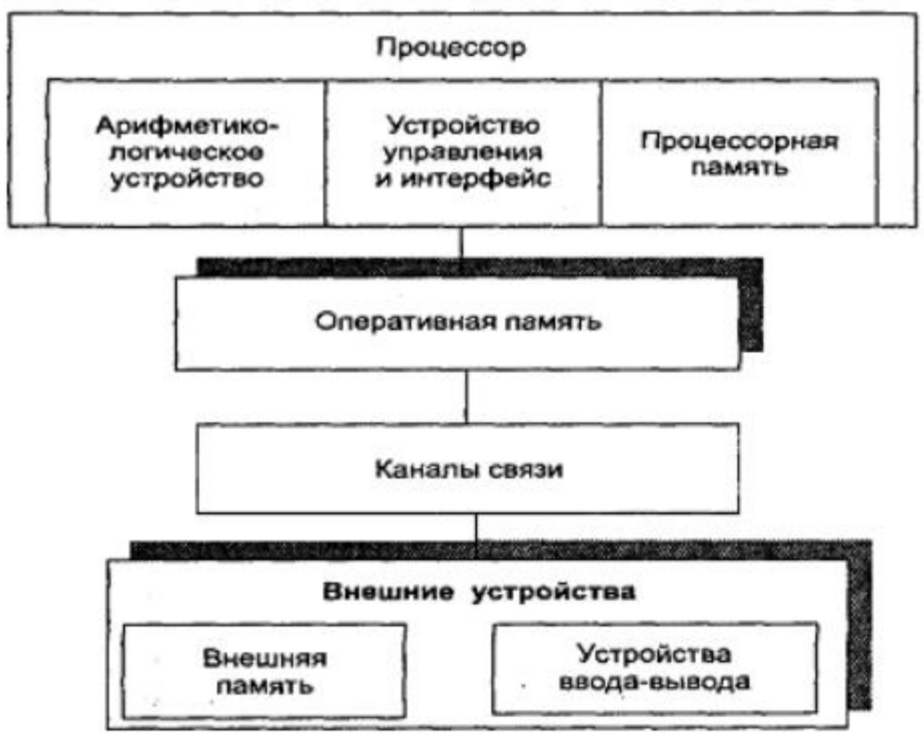


Рисунок 8 - Функциональная организация типовой ЭВМ

Процессор (центральный процессор) - основной вычислительный блок компьютера, содержит важнейшие функциональные устройства:

- устройство управления с интерфейсом процессора (системой сопряжения и связи процессора с другими узлами машины);
- арифметико-логическое устройство;
- процессорную память.

Процессор, является устройством, выполняющим все функции элементарной вычислительной машины.

Оперативная память — запоминающее устройство, используемое для оперативного хранения и обмена информацией с другими узлами машины.

Каналы связи (внутримашинный интерфейс) служат для сопряжения центральных узлов машины с ее внешними устройствами.

Внешние устройства обеспечивают эффективное взаимодействие компьютера с окружающей средой: пользователями, объектами управления, другими машинами. В состав внешних устройств обязательно входят внешняя память и устройства ввода-вывода.

На рисунке 9 приведена структурная организация типовой ЭВМ

Рисунок 9 – Структурная организация типовой ЭВМ

ЭВМ содержит следующие основные устройства: арифметико-логическое устройство, память, устройство управления, устройства ввода вывода.

АЛУ – производит арифметические и логические преобразования над поступающими в него машинными словами, т.е. кодами определённой длины, представляющими ту или иную информацию.

Память хранит информацию, передаваемую из других устройств. Память состоит из двух частей: быстродействующей основной (оперативной) памяти (ОП) и медленной, но способную хранить значительно больший объём информации, временной памяти (ВП).

Управляющее устройство (УУ) автоматически без участия человека управляет вычислительным процессом, посылая всем другим устройствам сигналы, предписывающие им те или иные действия.

Устройства ввода предназначены для преобразования информации, поступающей в ЭВМ из внешнего мира (окружающей среды). Устройства вывода предназначены, для преобразования обработанной ЭВМ информации, в удобную для человека форму.

Лекция 2 Способы представления данных в ЭВМ

Понятие данные, информация. Свойства информации.

Данные - поддающееся многократной интерпретации представление информации в формализованном виде, пригодном для передачи, связи или обработки.

Информация - осознанные сведения об окружающем мире, которые являются объектом хранения, преобразования, передачи и использования. Сведения - это знания, выраженные в сигналах, сообщениях, известиях, уведомлениях.

Основные виды информации по её форме представления, способам её кодирования и хранения, что имеет наибольшее значение для информатики, это:

– графическая;

– звуковая (акустическая);
– текстовая;
– числовая;
– видеоинформация.

Наиболее важными представляются следующие общие качественные свойства информации:

1. Объективность информации. Информация в любом своём проявлении объективна, она отображает объективную действительность.
2. Достоверность информации. Информация достоверна, если она отражает истинное положение дел. Достоверная информация помогает принять нам правильное решение.
3. Полнота информации. Информацию можно назвать полной, если ее достаточно для понимания и принятия решений. Неполная информация может привести к ошибочному выводу или решению.
4. Точность информации определяется степенью ее близости к реальному состоянию объекта, процесса, явления и т. п.
5. Актуальность информации - важность для настоящего времени, злободневность, насущность.
6. Полезность (ценность) информации. Полезность может быть оценена применительно к нуждам конкретных ее потребителей и оценивается по тем задачам, которые можно решить с ее помощью.

Представление данных разного типа в компьютере: целочисленные данные, данные и числа с плавающей точкой, строки фиксированной и переменной длины, символы, логические значения. Сравнение данных разных типов.

Любая информация представляется в компьютере в виде последовательности байтов, то есть целых положительных чисел. Базовыми типами данных, с помощью которых производится такое представление, являются целые без знака (однобайтовый, двухбайтовый и четырехбайтовый). Термин «без знака» означает, что данные могут быть только положительными.

Интерпретация содержимого байтов целиком зависит от той программы, которая работает с этими байтами. Поэтому при работе с любыми программами следует очень внимательно относиться к тому, с какими именно типами данных позволяют работать пользователю эти программы. Чаще всего используются следующие типы:

– целые (короткий, обычный и длинный),
– вещественные (с одинарной и двойной точностью),
– текстовый,
– логический,

– графические (растровый, векторный и фрактальный),

– звуковой.

Базовые типы данных.

Представление данных разных типов в компьютере производится с помощью целых положительных чисел. Основной единицей измерения объема информации является байт. Поскольку в байте 8 битов, то наибольшее число, которое можно представить одним байтом, равно 255. Следовательно, одним байтом (8 битами) можно представить 256 положительных чисел от 00000000 до 11111111 (соответствует десятичному числу 255). Такой тип данных называется *однобайтовым целым без знака*. Он позволяет работать с целыми положительными числами от 0 до 255.

Числа, превышающие 255, требуют более одного байта для своего представления. Для работы с ними используют типы двухбайтовый целый без знака, который обеспечивает представление 65536 целых положительных десятичных чисел (от 0 до 65535) и четырехбайтовый целый без знака, который позволяет представить более 4,2 млрд. целых положительных чисел (от 0 до 4 294 967 295).

Для работы с целыми числами, которые могут быть не только положительными, но и отрицательными, используются типы:

– однобайтовый целый со знаком (целый короткий);

– двухбайтовый целый со знаком (целый обычный);

– четырехбайтовый целый со знаком (целый длинный).

Они отличаются объемом памяти, которая отводится для хранения каждого числа. Для представления чисел, которые могут быть как положительными, так и отрицательными, используются разные способы. Основными из них являются:

– дополнительный код,

– смещение.

Для однобайтового представления общее количество таких кодов 256 (от 0 до 255), для двухбайтового – 65536 (от 0 до 65535), для четырехбайтового – более 4,2 млрд.

Дополнительный код. Общее количество числовых кодов, возможных для данного количества байтов, делится пополам. Первая половина используется для представления положительных чисел и нуля (прямым кодом), а другая – для представления отрицательных чисел. При этом отрицательные числа представляются как дополнение до общего количества числовых кодов (дополнительным кодом).

Например, если для представления числа используется один байт, то число –1 представляется числом 255 (256–1=255), –2 – числом 254 (256–2=254) и т.д. до 128, которое означает число –128.

Смещение. К числу перед записью его в память прибавляется положительное число, которое называется смещением. Смещение

выбирается таким образом, чтобы минимальному числу соответствовал нуль. Метод смещения упрощает вычисления и сравнение чисел.

Например, для однобайтового представления смещение равно 128. Тогда минимальное число –128 представляется в памяти компьютера нулем (–128 + 128 = 0), –127 представляется числом 1 (–127 + 128 = 1) и т.д. до 127, которое представляется числом 255 (127 + 128 = 255).

Таким образом, однобайтовое целое со знаком позволяет работать с целыми числами от –128 до +127, двухбайтовое целое со знаком – от –32768 до +32767 и четырехбайтовое целое со знаком – примерно от –2,1 млрд. до +2,1 млрд. (точнее, от –2 147 483 648 до 2 147 483 647).

Работая с вещественными числами, следует иметь в виду два аспекта:

– способы визуализации чисел (запись вещественных чисел на бумаге, их представление при вводе с клавиатуры, выводе на экран или принтер и т.п.);

– способы представления чисел в памяти ЭВМ.

Обычно привыкли записывать их в виде десятичных дробных чисел, в которых знак запятой разделяет целую и дробную часть (например, 5,8; 138,654). В информационных технологиях при записи вещественных чисел в качестве разделителя целой и дробной части принято, как правило, вместо запятой писать точку (то есть 5.8; 138.654).

При вводе или выводе вещественных чисел используют два способа (формата) записи:

– формат с фиксированной точкой (этим форматом мы пользуемся в обычной практике. Например, 138.654. Точка фиксирует позицию, после которой указана дробная часть. Изменение ее местоположения меняет смысл числа. Однако этот способ неудобен для записи чисел большой длины, то есть состоящих из большого количества цифр. В этом случае полезен формат с плавающей точкой);

– формат с плавающей точкой.

При таком виде записи точка не фиксирована, ее положение определяется величиной порядка. Точка как бы плавает в зависимости от величины порядка.

Например, число 3.186 можно записать как

31.86E-1 = 3.186E0 = 0.3186E+1,

что означает

31.86×10-1 = 3.186×100 = 0.3186×101.

Если мантисса по модулю меньше 1, причем первая цифра после точки не равна нулю, то такой вид записи вещественного числа с плавающей точкой называют нормализованным. Например,

в 10-чной системе 3.1415926 = 0.31415926E+1,

в 2-чной системе 1000.00012 = 0.100000012E+100 (здесь 100 – двоичная форма десятичного числа 4).

Если мантисса по модулю больше нуля, но меньше 10, то такой вид записи называется нормализованным экспоненциальным. У такого числа целая часть мантиссы состоит из одной цифры. Например,

в 10-чной системе 0.00234 = 2.34E–3,

в 2-чной системе 101.11 = 1.0111E+10; 0.0011 = 1.1E–11 (здесь степени 10 и 11 – это двоичная форма десятичных чисел 2 и 3).

Вещественное число с плавающей точкой состоит из двух частей – мантиссы и порядка, разделенных специальным знаком (латинская буква E для одинарной точности и буква D для двойной точности). Мантисса представляет собой вещественное число с фиксированной точкой. Порядок задается целым числом, указывающим, в какую степень надо возвести число 10, чтобы при умножении результата на мантиссу получить вещественное число в формате с фиксированной точкой. Как мантисса, так и порядок могут иметь знак (знак плюс обычно не указывается). Для представления таких чисел в памяти компьютера используются нормализованная и нормализованная экспоненциальная форма.

Программа, “зная”, что имеет дело с вещественным числом с плавающей точкой, соответственно интерпретирует эти байты.

В этом случае при записи числа в качестве разделителя мантиссы и порядка используется латинская буква E. В памяти ЭВМ такое число занимает обычно 4 байта.

В компьютере вещественное число с плавающей точкой представляется таким образом, что мантисса и порядок располагаются в соседних байтах, знак E (или D) отсутствует. Различают вещественные числа с одинарной точностью (4-байтовое) (в этом случае при записи числа в качестве разделителя мантиссы и порядка используется латинская буква E. В памяти ЭВМ такое число занимает обычно 4 байта) и вещественные числа с двойной точностью (8-байтовое) (в этом случае разделителем мантиссы и порядка является буква D (например, 3.4D–3). В памяти ЭВМ число с двойной точностью занимает обычно 8 байтов. Такое представление обеспечивает бóльшую точность в вычислениях, чем одинарная точность.)

Получить вещественное число: При таком виде записи точка не фиксирована, ее положение определяется величиной порядка. Точка как бы плавает в зависимости от величины порядка.

Например, число 3.186 можно записать как

31.86E-1 = 3.186E0 = 0.3186E+1,

что означает

31.86×10-1 = 3.186×100 = 0.3186×101.

Нормализованная: Если мантисса по модулю меньше 1, причем первая цифра после точки не равна нулю, то такой вид записи вещественного числа с плавающей точкой называют нормализованным. Например,

в 10-чной системе 3.1415926 = 0.31415926E+1,

в 2-чной системе 1000.00012 = 0.100000012E+100 (здесь 100 – двоичная форма десятичного числа 4).

Нормализованная экспоненциальная: Если мантисса по модулю больше нуля, но меньше 10, то такой вид записи называется нормализованным экспоненциальным. У такого числа целая часть мантиссы состоит из одной цифры. Например,

в 10-чной системе 0.00234 = 2.34E–3,

в 2-чной системе 101.11 = 1.0111E+10; 0.0011 = 1.1E–11 (здесь степени 10 и 11 – это двоичная форма десятичных чисел 2 и 3).

В этом случае разделителем мантиссы и порядка является буква D (например, 3.4D–3). В памяти ЭВМ число с двойной точностью занимает обычно 8 байтов. Такое представление обеспечивает большую точность в вычислениях, чем одинарная точность.

Текстовый тип данных

Текстовые данные составлены из отдельных текстовых знаков. Каждый знак представляется в виде определенной комбинации битов (то есть двоичного числа). Текст в памяти ЭВМ представляется последовательностью следующих друг за другом байтов. Для числового кодирования текстовых знаков используются специальные таблицы кодирования.

Например, ASCII (American Standard Code for Information Interchange – стандартный код информационного обмена США) введена в действие Институтом стандартизации США. В ней закреплены две части таблицы кодирования – базовая и расширенная. В базовой расположены значения кодов от 0 до 127, а в расширенной – от 128 до 255. Первые 32 кода базовой таблицы, начиная с нулевого, используются только производителями аппаратных средств. Этими номерами обозначены неотображаемые управляющие коды. С 32 по 127 коды – символы английского алфавита, знаки препинания и т.д. Коды с 128 до 255 используются различными странами для кодирования знаков алфавита своих языков. В частности, для кодирования знаков русского языка используются таблицы Windows-1251 (введена компанией Microsoft), КОИ-8 (Код обмена информацией восьмизначный – введена в действие в России и широко распространена в компьютерных сетях).

Кроме этого, в настоящее время существует таблица кодирования Unicode – 16-разрядная система кодирования знаков алфавита большинства языков планеты.

Программы могут работать на основе различных таблиц кодирования. Поэтому текстовый документ, созданный с помощью одной программы, не обязательно может быть прочитан с помощью другой.

Строковый тип данных

Строковый тип данных - тип данных, состоящий из последовательности смежных символов, которые представляют сами

символы, а не их числовые значения. А строка может включать буквы, числа, пробелы и знаки препинания. Тип данных строка может хранить строки фиксированной длины, от 0 до приблизительно 63 тысяч символов, а динамические строки могут быть длиной от 0 до приблизительно 2 миллиардов символов.

Существует два типа строк: строки переменной длины и фиксированной длины.

Строка переменной длины может содержать приблизительно до 2 миллиардов (2^31) знаков.

Строка фиксированной длины может содержать от 1 до 64 K (2^16) символов.

Коды знаков строкового типа находятся в диапазоне 0–255. Первые 128 символов (0–127) набора знаков соответствуют буквам и символам на стандартной клавиатуре США. Эти первые 128 знаков совпадают со знаками, которые определяются набором знаков ASCII. Следующие 128 знаков (128–255) представляют специальные знаки, такие как буквы международных алфавитов, диакритические знаки, символы валюты и дроби.

Логический тип данных

Логические величины принимают только два значения – TRUE (истина) и FALSE (ложь). К ним можно применять логические операции, основными из которых являются: AND (конъюнкция – логическое И), OR (дизъюнкция – логическое ИЛИ) и NOT (инверсия – логическое отрицание).

В некоторых программах конъюнкция обозначается знаками & или $\wedge$, например

A & B или A B.

В некоторых программах дизъюнкция обозначается знаками $\vee$или |, например

A B или A | B.

В некоторых программах конъюнкция обозначается знаками $\neg$(или $\overline{\ }$), например

A или $\overline{A}$.

Первые две операции применяются к двум логическим величинам (например, a AND b или c OR d), а операция NOT – к одной (например, NOT a). Результатом выражения с логическими данными (логические выражения) является логическая величина. Результат операции AND равен TRUE только в том случае, если обе величины равны TRUE, в остальных случаях результат равен FALSE. Если применяется операция OR, то результат равен FALSE только в том случае, если обе величины FALSE, в остальных случаях результат равен TRUE. Операция NOT изменяет значение логической величины: результат равен TRUE, если величина равна FALSE, и наоборот. Среди логических первой выполняется операция Not, затем And и последней

Or. Порядок выполнения операций может быть изменен использованием скобок (например, если

a = TRUE, b = FALSE и c = FALSE, то выражение a AND (b OR NOT c) равно TRUE.)

Например, если

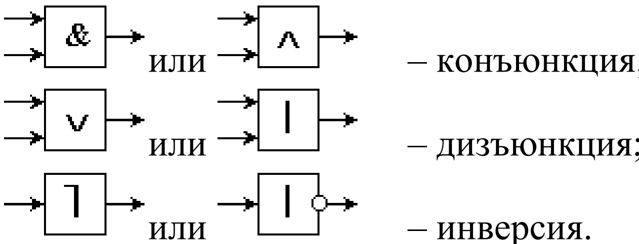
a = TRUE, b = FALSE и c = FALSE,

то выражение

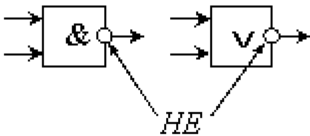
a AND (b OR NOT c)

равно TRUE

При этом используются следующие обозначения



Можно для обозначения одной вершиной сразу двух операций «И–НЕ», «ИЛИ–НЕ» использовать следующую форму:



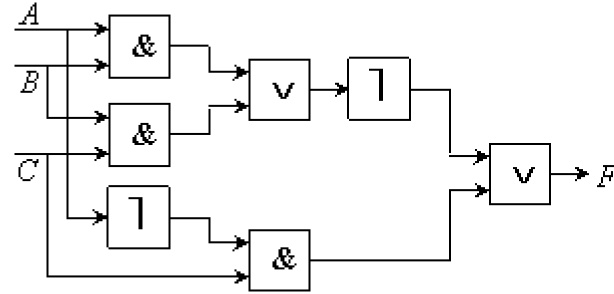
Это позволяет сократить схему. Конструкцию «ИЛИ–НЕ» еще называют логической операцией «исключающее ИЛИ» и для ее обозначения используют следующий элемент схемы:

=1

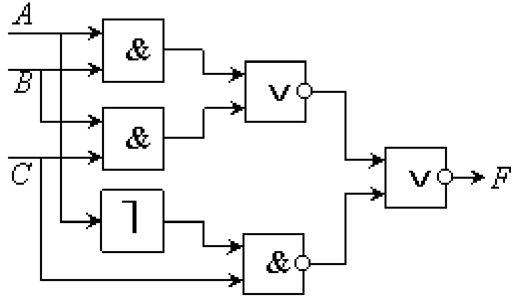
Например, логическое выражение

$$F = \overline{(A \& B) \vee (B \& C)} \vee (\overline{A} \& C)$$

может быть представлено схемой



Например, схема



представляет логическое выражение

$$F = \overline{\overline{(A \& B) \vee (B \& C)} \vee \overline{(\overline{A} \& C)}}.$$

В смешанных выражениях операции выполняются в соответствии с приоритетами. При этом наивысший приоритет у арифметических операций, затем выполняются операции сравнения и самый низкий приоритет у логических операций. Операции с одинаковым приоритетом выполняются слева направо.

Для упрощения логических выражений можно использовать следующие соотношения:

1. $\overline{A \& B}$ эквивалентно $\overline{B}$;
2. $\overline{A \vee B}$ эквивалентно & ;
3. A A & B = A & (1 B) = A.

В их справедливости можно убедиться, проверив результаты логических выражений при всех возможных значениях A и B. Первое и второе соотношения носят названия законов Моргáна.

Например, в результате упрощения логического выражения F = $\overline{B \vee \overline{C}}$ получится выражение

$$F = \overline{A \& B} \vee \overline{B \vee \overline{C}} = \overline{A} \vee \overline{B} \vee \overline{B} \& \overline{\overline{C}} = \overline{A} \vee (\overline{B} \vee \overline{B} \& C) =$$
$$\overline{A} \vee (\overline{B} \& (1 \vee C)) = \overline{A} \vee (\overline{B} \& 1) = \overline{A} \vee \overline{B}.$$

Для упрощения логических выражений можно использовать различные соотношения

Различные варианты кодирования символов.

Набор символов или кодировка (character set, charset) - это определённая таблица кодирования конечного множества символов.

Кодовая страница (code page) - это однобайтная (8-битная) кодировка.

Кодировка ASCII (American Standard Code for Information Interchange - "аски", с ударением на первом слоге) - это 7-битная (128 символов) кодировка для представления латинского алфавита, десятичных цифр, некоторых знаков препинания, арифметических операций и управляющих символов.

В 8-битных национальных кодировках нижнюю половину кодовой таблицы (0 - 127) занимают символы ASCII, а верхнюю (128 - 255) - другие нужные символы. В Юникоде первые 128 символов тоже совпадают с соответствующими символами ASCII.

Кодировка Windows-1251 (cp1251) является стандартной 8-битной кодировкой для всех русских версий Microsoft Windows. Первая часть таблицы кодировки (латиница) полностью соответствует кодировке ASCII.

Стандартом для русской кириллицы в юникс-подобных операционных системах является кодировка КОИ-8 (код обмена информацией, 8 битов), или KOI8. Существует несколько вариантов кодировки КОИ-8 для различных кириллических алфавитов. Русский алфавит описывается в кодировке KOI8-R, украинский — в KOI8-U, существуют также кодировки KOI8-RU (русско-белорусско-украинская), KOI8-T (таджикская) и т.д.

Разработчики КОИ-8 разместили символы русского алфавита таким образом, что если в тексте, написанном в КОИ-8, убирать восьмой бит каждого символа, то получается "читабельный" текст, хотя он и написан латинскими символами.

Юникод, или Уникод (Unicode) - это стандарт кодирования символов, позволяющий представить знаки практически всех письменных языков.

Чаще всего для обозначения символов Unicode используется запись вида "U+xxxx" (для кодов 0...FFFF), где xxx - шестнадцатеричные цифры. Первая версия Юникода представляла собой кодировку с фиксированным размером символа в 16 бит, то есть общее число кодов было 216 (65536). Отсюда и происходит практика обозначения символов четырьмя шестнадцатеричными цифрами (например, U+0410).

Коды в стандарте Unicode разделены на несколько областей, например:

- Область от U+0000 до U+007F содержит символы набора ASCII.
- Область от U+0400 до U+052F содержит символы кириллицы, где символы до U+045F - это собственно кириллица, а далее располагаются исторические буквы и дополнительные буквы для разных языков, использующих кириллицу.

В дальнейшем было принято решение расширить кодовую область, и коды символов стали рассматриваться не как 16-битные значения, а как абстрактные числа, которые в компьютере могут представляться множеством разных способов. Однако, поскольку в ряде компьютерных систем (например, Windows NT) до изобретения Юникода уже были реализованы 16-битные символы, было решено всё наиболее важное кодировать только в пределах первых 65536 позиций (так наз. Basic Multilingual Plane, BMP). Остальное пространство используется для "дополнительных символов" (Supplementary Characters): систем письма вымерших языков или очень редко

используемых китайских иероглифов, математических и музыкальных символов.

Стандарт Юникода содержит семейство кодировок (форм представления или UTF, Unicode Transformation Format): UTF-8, UTF-16, UTF-32 и некоторые другие, которые отличаются между собой способом хранения данных (количество байт на символ, фиксированное или нефиксированное количество байт на символ). Была разработана также форма представления UTF-7 для передачи по семибитным каналам, но из-за несовместимости с ASCII она не получила распространения и не включена в стандарт.

UTF-8 - это представление Юникода, обеспечивающее наилучшую совместимость со старыми системами, использовавшими 8-битные символы. Текст, состоящий только из символов с номером меньше 128, при записи в UTF-8 превращается в обычный текст ASCII. И наоборот, в тексте UTF-8 любой байт со значением меньше 128 изображает символ ASCII с тем же кодом. Для совместимости со старыми 16-битными системами была изобретена система UTF-16, где первые 65536 позиций отображаются непосредственно как 16-битные числа, а остальные представляются в виде "суррогатных пар".

Системы счисления. Общие понятия и конкретные системы: десятичная, двоичная, шестнадцатеричная, восьмеричная. Правила перевода чисел из одной системы в другую.

Система счисления - это способ представления числа. Одно и то же число может быть представлено в различных видах. Например, число 200 в привычной нам десятичной системе может иметь вид 11001000 в двоичной системе, 310 в восьмеричной и C8 в шестнадцатеричной.

Для указания системы счисления при записи числа используется нижний индекс, который ставится после числа:
20010 = 110010002 = 3108 = C816

Десятичная система счисления. Используется в повседневной жизни и является самой распространенной. Все числа, которые нас окружают представлены в этой системе. В каждом разряде такого числа может использоваться только одна цифра от 0 до 9.

Двоичная система счисления. Используется в вычислительной технике. Для записи числа используются цифры 0 и 1.

Восьмеричная система счисления. Также иногда применяется в цифровой технике. Для записи числа используются цифры от 0 до 7.

Шестнадцатеричная система счисления. Наиболее распространена в современных компьютерах. При помощи неё, например, указывают цвет. #FF0000 - красный цвет. Для записи числа используются цифры от 0 до 9 и буквы A,B,C,D,E,F, которые соответственно обозначают числа 10,11,12,13,14,15.

Перевод в десятичную систему счисления

Преобразовать число из любой системы счисления в десятичную можно следующим образом: каждый разряд числа необходимо умножить на X^n, где X - основание исходного числа, n - номер разряда. Затем суммировать полученные значения.

$abc_x = (a*x^2 + b*x^1 + c*x^0)_{10}$

Примеры:

$567_8 = (5*8^2 + 6*8^1 + 7*8^0)_{10} = 375_{10}$

$110_2 = (1*2^2 + 1*2^1 + 0*2^0)_{10} = 6_{10}$

$A5_{16} = (10*16^1 + 5*16^0)_{10} = 165_{10}$

Перевод из десятичной системы счисления в другие

Делим десятичное число на основание системы, в которую хотим перевести и записываем остатки от деления. Запишем полученные остатки в обратном порядке и получим искомое число.

Переведем число 375_{10} в восьмеричную систему:

375 / 8 = 46 (остаток 7)

46 / 8 = 5 (остаток 6)

5 / 8 = 0 (остаток 5)

Записываем остатки и получаем 567_8

Устройство памяти. Адресация.

Устройство памяти

Память - способность объекта обеспечивать хранение данных. Все объекты, над которыми выполняются команды, как и сами команды, хранятся в памяти компьютера.

Память состоит из ячеек, в каждой из которых содержится 1 бит информации, принимающий одно из двух значений: 0 или 1. Биты обрабатывают группами фиксированного размера. Для этого группы бит могут записываться и считываться за одну базовую операцию.

Группа из 8 бит называется байтом.

Байты последовательно располагаются в памяти компьютера.

Функцией памяти является хранение данных и кодов программ. Существенным параметром памяти является ёмкость памяти, которая оценивается в битах и байтах. Более крупные единицы измерения данных образуются добавлением префиксов кило-, мега-, гига-, тера. При этом обычно берётся первая буква): K (kilo - 1024), M (mega - 1024×1024), G (giga - 1024× 1024× 1024), T (tera - 1024× 1024× 1024× 1024).

- 1 килобайт (Кбайт) = 2^{10} = 1 024 байт
- 1 мегабайт (Мбайт) = 2^{10} Кбайт = 2^{20} байт = 1 048 576 байт
- 1 гигабайт (Гбайт) = 2^{10} Мбайт = 2^{30} байт = 1 073 741 824 байт

Другим важным параметром является скорость обмена данными (скорость записи и скорость считывания).

Для доступа к памяти с целью записи или чтения отдельных элементов информации используются идентификаторы, определяющие их расположение в памяти. Каждому идентификатору в соответствие ставится адрес. В качестве адресов используются числа из диапазона от 0 до 2^k-1 со значением k, достаточным для адресации всей памяти компьютера. Все 2^k адресов составляют адресное пространство компьютера.

Типы памяти подразделяются на оперативную (изменяемую) память (RAM - Random Access Memory) и постояную память (ROM - Read Only Memory). RAM не сохраняет записанные данные при отключении питания и её используют для сохранения работающих прогамм и данных. Оперативную память иногда называют памятью с произвольным доступом, что указывает на непосредственный, прямой доступ по всем возможным адресам.

Оперативная (изменяемая) память подразделяется на:

1. Статическая память (SRAM -Static Random Access Memory). Этот тип памяти обеспечивает наиболее высокое быстродействие, хотя технологически он сложнее (реализуется на триггерах) и, сответсвенно, дороже. Этот тип памяти используется в качестве промежуточной памяти (Cache memory) или процессорных регистров.
2. Динамическая память (DRAM - Dynamic Random Access Memory). Этот тип памяти можно представить в виде микроконденсаторов, способных накапливать заряд. Это наиболее распространённый и дешёвый тип памяти. Для запоминания одного бита информации достаточно одного транзистора.

Постоянная память используется в различных компонентах аппаратного обеспечения (firmware). Например, на материнской плате любого компьютера имеется постоянная память содержащая BIOS (Basic Input Output System).

Бывают разные типы постоянной памяти:

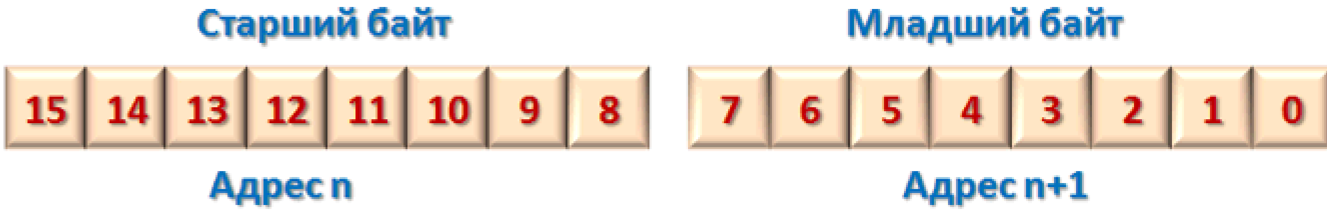
- ROM (Read-Only Memory) - постоянная память, которая программируется в ходе её изготовления и не может быть перепрограммирована в дальнейшем.
- PROM (Programmable ROM) - однократно программируемый чип постоянной памяти.
- EPROM (Erasable PROM) - перепрограммируемое ПЗУ, для перепрограммирования старая програма стирается при помощи ультрафиолетого луча.
- EEPROM (Electrically Erasable PROM) - электрически стираемое перепрограммируемое ПЗУ. Память такого типа может стираться и заполняться данными несколько десятков тысяч раз. Использование такого типа ПЗУ позволяет модифицировать используемое оборудование.

Адресация - осуществление ссылки (обращение) к устройству или элементу данных по его адресу; установление соответствия между

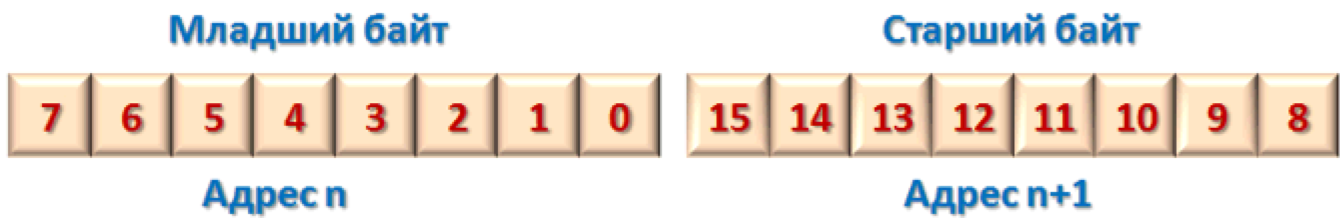
множеством однотипных объектов и множеством их адресов; метод идентификации местоположения объекта.

Способы адресации байтов

Существует прямой и обратный способы адресации байтов.
При обратном способе адресации байты адресуются слева направо, так что самый старший (левый) байт слова имеет наименьший адрес.



Прямым способом называется противоположная система адресации.
Компиляторы высокоуровневых языков поддерживают прямой способ адресации.



Объект занимает целое слово. Поэтому для того, чтобы обратиться к нему в памяти, нужно указать адрес, по которому этот объект хранится.

Организация памяти

Физическая память, к которой микропроцессор имеет доступ по шине адреса, называется оперативной памятью ОП (или оперативным запоминающим устройством - ОЗУ).

Механизм управления памятью полностью аппаратный, т.е. программа сама не может сформировать физический адрес памяти на адресной шине.

Микропроцессор аппаратно поддерживает несколько моделей использования оперативной памяти:

- сегментированную модель;
- страничную модель;
- плоскую модель.

В сегментированной модели память для программы делится на непрерывные области памяти, называемые сегментами. Программа может обращаться только к данным, которые находятся в этих сегментах.

Сегмент представляет собой независимый, поддерживаемый на аппаратном уровне блок памяти.

Физический адрес принято записывать парой этих значений, разделенных двоеточием

сегмент : смещение

Страничная модель памяти – это надстройка над сегментной моделью. ОЗУ делится на блоки фиксированного размера, кратные степени 2,

например 4 Кб. Каждый такой блок называется страницей. Основное достоинство страничного способа распределения памяти - минимально возможная фрагментация. Однако такая организация памяти не использует память достаточно эффективно за счет фиксированного размера страниц.

Плоская модель памяти предполагает, что задача состоит из одного сегмента, который, в свою очередь, разбит на страницы.

Понятие переменная. Объявление (декларация) и инициализация переменных. Правила именования.

Данные, значения которых во время выполнения программы можно изменять, называются переменными, неизменяемые данные называются константами. В программе все данные перед их использованием должны быть объявлены.

В операторах объявления данных указывается тип данных и перечисляется через запятую имена переменных, имеющих данный тип. Поскольку все данные компьютера - это лишь последовательность битов, то мы используем тип данных (или просто **«тип»**), чтобы сообщить компилятору, как интерпретировать содержимое памяти. Когда мы объявляем целочисленную переменную, то мы сообщаем компилятору, что «кусочек памяти, который находится по такому-то адресу, следует интерпретировать как целое число».

При объявлении переменной можем присвоить ей значение в этот же момент. Это называется инициализацией переменной.

Именование

Основные правила стиля кодирования приходятся на именование. Вид имени сразу же (без поиска объявления) говорит нам что это: тип, переменная, функция, константа, макрос и т.д. Правила именования могут быть произвольными, однако важна их согласованность, и правилам нужно следовать.

Общие принципы именования

1. Используйте имена, который будут понятны даже людям из другой команды.
2. Имя должно говорить о цели или применимости объекта.
3. Не экономьте на длине имени, лучше более длинное и более понятное (даже новичкам) имя.
4. Поменьше аббревиатур, особенно если они незнакомы вне проекта.
5. Используйте только известные аббревиатуры.
6. Не сокращайте слова.

Основные правила именования переменных:

1. Имя переменной может состоять только из латинских букв, цифр и символа подчеркивания.
2. Прописные и строчные буквы в именах различаются, то есть переменные abc и Abc - разные переменные.
3. Имя переменной не может начинаться с цифры.
4. Имя переменной не может повторяться, то есть нельзя объявить две переменные с одним именем.

5. В качестве имени переменной не могут использоваться ключевые слова языка.

Лекция 3 Общие сведения о программном обеспечении и технологии производства программного обеспечения

Термины и определения (ГОСТ 19.001-77). Виды программного обеспечения и программной документации (ГОСТ 19.701-90, ИСО 5807-85).

Программное обеспечение ПК делится на системное и прикладное и инструментальные системы. Прикладное программное обеспечение предназначено для решения задач определенного класса (конкретных задач). Сейчас для ПЭВМ предлагается множество прикладных программ(например, текстовый редактор).

Системное программное обеспечение используется для поддержки, выполнения и разработки других программ, предоставления пользователю определенных услуг и организует взаимодействие между пользователем и аппаратурой компьютера. Среди всех системных программ первостепенную значимость имеет операционная система.

Инструментальные системы - такие программные продукты, которые предназначены для разработки программного обеспечения. Примером таких систем является система программирования (C++ и др).

Технологии производства программного обеспечения

Технология разработки программного обеспечения - это комплекс мер по созданию программных продуктов. Технология разработки программного обеспечения – это система инженерных принципов для создания экономичного программного обеспечения, которое надежно и эффективно работает в реальных компьютерах.

Сейчас обобщённый термин, применимый к созданию программных средств, обозначают как«разработка» или «конструирование». Справедлива формула:

разработка = анализ + проектирование + программирование+ тестирование + отладка

Иногда сюда также включают “сопровождение”.

Термины и определения (ГОСТ 19.001-77). Виды программного обеспечения и программной документации (ГОСТ 19.701-90, ИСО 5807-85).

Единая система программной документации - комплекс государственных стандартов, устанавливающих взаимоувязанные правила разработки, оформления и обращения программ и программной документации.

В стандартах ЕСПД устанавливают требования, регламентирующие разработку, сопровождение, изготовление и эксплуатацию программ, что обеспечивает возможность:

- унификации программных изделий для взаимного обмена программами и применения ранее разработанных, программ в новых разработках;
- снижения трудоемкости и повышения эффективности разработки, сопровождения, изготовления и эксплуатации программных изделий;
- автоматизации изготовления и хранения программной документации.

Сопровождение программы включает анализ функционирования, развитие и совершенствование программы, а также внесение изменений в нее с целью устранения ошибок.

В состав ЕСПД входят:

- основополагающие и организационно-методические стандарты;
- стандарты, определяющие формы и содержание программных документов, применяемых при обработке данных;
- стандарты, обеспечивающие автоматизацию разработки программных документов.

Разработка организационно-методической документации, определяющей и регламентирующей деятельность организаций по разработке, сопровождению и эксплуатации программ, должна проводиться на основе стандартов ЕСПД.

Программы подразделяют на виды:

1. Компонент - программа, рассматриваемая как единое целое, выполняющая законченную функцию и применяемая самостоятельно или в составе комплекса.
2. Комплекс - программа, состоящая из двух или более компонентов и (или) комплексов, выполняющих взаимосвязанные функции, и применяемая самостоятельно или в составе другого комплекса

Вид программных документов и содержание программных документов:

1. Спецификация - состав программы и документации на нее.
2. Ведомость держателей подлинников - перечень предприятий, на которых хранят подлинники программных документов.
3. Текст программы - запись программы с необходимыми комментариями.
4. Описание программы - сведения о логической структуре и функционировании программы.
5. Программа и методика испытаний - требования, подлежащие проверке при испытании программы, а также порядок и методы их контроля.
6. Техническое задание - назначение и область применения программы, технические, технико-экономические и специальные требования, предъявляемые к программе, необходимые стадии и сроки разработки, виды испытаний.

7. Пояснительная записка - схема алгоритма, общее описание алгоритма и (или) функционирования программы, а также обоснование принятых технических и технико-экономических решений.
8. Эксплуатационные документы - сведения для обеспечения функционирования и эксплуатации программы.

   Виды эксплуатационных документов:

1. Ведомость эксплуатационных документов - перечень эксплуатационных документов на программу.
2. Формуляр - основные характеристики программы, комплектность и сведения об эксплуатации программы.
3. Описание применения - сведения о назначении программы, области применения, применяемых методах, классе решаемых задач, ограничениях для применения, минимальной конфигурации технических средств.
4. Руководство системного программиста - сведения для проверки, обеспечения функционирования и настройки программы на условия конкретного применения.
5. Руководство программиста - сведения для эксплуатации программы
6. Руководство оператора - сведения для обеспечения процедуры общения оператора с вычислительной системой в процессе выполнения программы.
7. Описание языка - описание синтаксиса и семантики языка.
8. Руководство по техническому обслуживанию - сведения для применения тестовых и диагностических программ при обслуживании технических средств

Типы программного обеспечения: системное, прикладное, инструментальное (средства разработчика).

Программное обеспечение ПК делится на системное и прикладное и инструментальные системы. Прикладное программное обеспечение предназначено для решения задач определенного класса (конкретных задач). Сейчас для ПЭВМ предлагается множество прикладных программ(например, текстовый редактор).

Системное программное обеспечение используется для поддержки, выполнения и разработки других программ, предоставления пользователю определенных услуг и организует взаимодействие между пользователем и аппаратурой компьютера. Среди всех системных программ первостепенную значимость имеет операционная система.

Инструментальные системы - такие программные продукты, которые предназначены для разработки программного обеспечения. Примером таких систем является система программирования (С++ и др).

Трехуровневая модель программного продукта.

Трёхуровневая архитектура (трёхзве́нная архитектура, англ. three-tier) - архитектурная модель программного комплекса, предполагающая наличие в нём трёх типов компонентов (уровней, звеньев): клиентских приложений (с которыми работают пользователи), серверов приложений (с которыми работают клиентские приложения) и серверов баз данных (с которыми работают серверы приложений).

Компоненты

Клиент (слой клиента) — это компонент комплекса(обычно графический), предоставляемый конечному пользователю. Этот уровень не должен иметь прямых связей с базой данных (по требованиям безопасности и масштабируемости), быть нагруженным основной бизнес-логикой (по требованиям масштабируемости) и хранить состояние приложения (по требованиям надёжности). На этот уровень обычно выносится только простейшая бизнес-логика: интерфейс авторизации, алгоритмы шифрования, проверка вводимых значений на допустимость и соответствие формату, несложные операции с данными (сортировка, группировка, подсчёт значений), уже загруженными на терминал.

Сервер приложений (средний слой, связующий слой) располагается на втором уровне, на нём сосредоточена бо́льшая часть бизнес-логики. Вне его остаются только фрагменты, экспортируемые клиенту (терминалу), а также элементы логики, погруженные в базу данных (хранимые процедуры и триггеры). Реализация данного компонента обеспечивается связующим программным обеспечением. Серверы приложений проектируются таким образом, чтобы добавление к ним дополнительных экземпляров обеспечивало горизонтальное масштабирование производительности программного комплекса и не требовало внесения изменений в программный код приложения.

Сервер баз данных (слой данных) обеспечивает хранение данных и выносится на отдельный уровень, реализуется, как правило, средствами систем управления базами данных, подключение к этому компоненту обеспечивается только с уровня сервера приложений.

В простейших конфигурациях все компоненты или часть из них могут быть совмещены на одном вычислительном узле. В продуктивных конфигурациях, как правило, используется выделенный вычислительный узел для сервера баз данных или кластер серверов баз данных, для серверов приложений — выделенная группа вычислительных узлов, к которым непосредственно подключаются клиенты (терминалы).

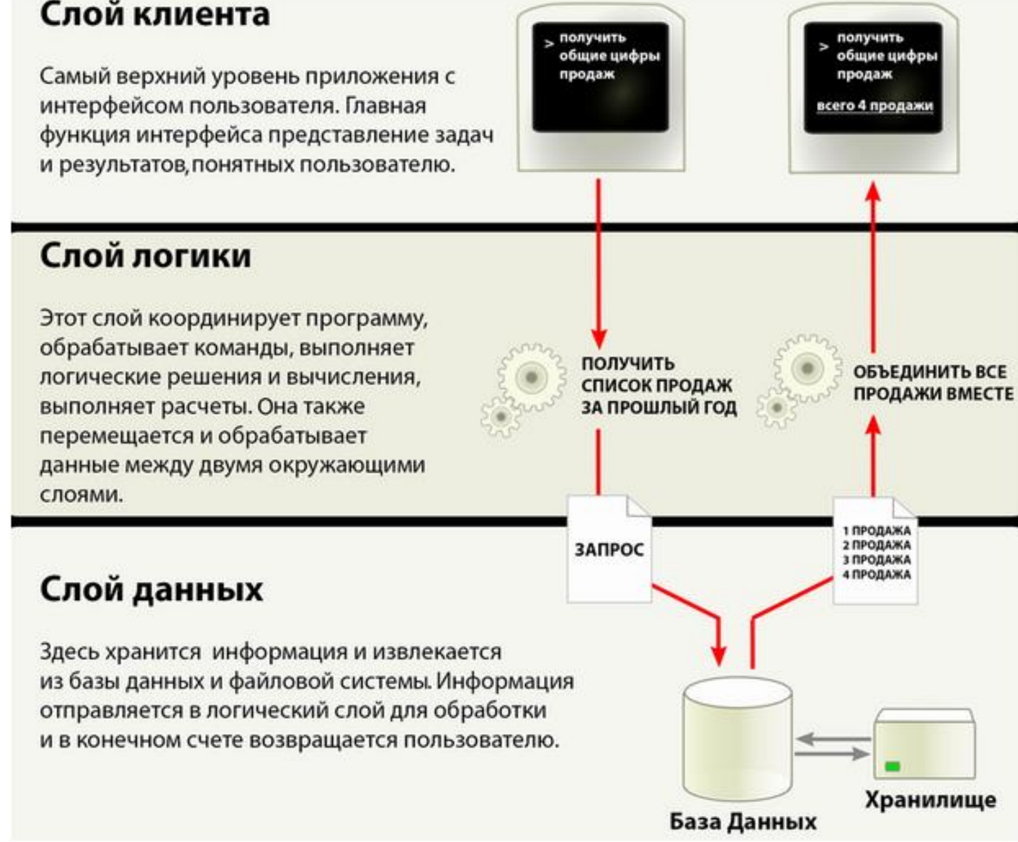


Рисунок 10 - Трехуровневая модель программного продукта

Свойства алгоритмов. Формы представления алгоритмов: естественный язык, бок-схема, формальный язык. Составление блок-схем алгоритмов.

Основные свойства алгоритмов следующие:

1. Понятность для исполнителя — исполнитель алгоритма должен понимать, как его выполнять. Иными словами, имея алгоритм и произвольный вариант исходных данных, исполнитель должен знать, как надо действовать для выполнения этого алгоритма.

2. Дискретность (прерывность, раздельность) — алгоритм должен представлять процесс решения задачи как последовательное выполнение простых (или ранее определенных) шагов (этапов).

3. Определенность — каждое правило алгоритма должно быть четким, однозначным и не оставлять места для произвола. Благодаря этому свойству выполнение алгоритма носит механический характер и не требует никаких дополнительных указаний или сведений о решаемой задаче.

4. Результативность (или конечность) состоит в том, что за конечное число шагов алгоритм либо должен приводить к решению задачи, либо после конечного числа шагов останавливаться из-за невозможности получить решение с выдачей соответствующего сообщения, либо неограниченно продолжаться в течение времени, отведенного для исполнения алгоритма, с выдачей промежуточных результатов.

5. Массовость означает, что алгоритм решения задачи разрабатывается в общем виде, т.е. он должен быть применим для некоторого класса задач, различающихся лишь исходными данными. При этом исходные данные могут выбираться из некоторой области, которая называется областью применимости алгоритма.

Формы записи алгоритмов

На практике наиболее распространены следующие формы представления алгоритмов:

1. Словесная (запись на естественном языке).
2. Графическая (изображения из графических символов).
3. Псевдокоды (полуформализованные описания алгоритмов на условном алгоритмическом языке, включающие в себя как элементы языка программирования, так и фразы естественного языка, общепринятые математические обозначения и др..
4. Программная (тексты на языках программирования).

Словесный способ записи алгоритма

Словесный способ записи алгоритмов представляет собой описание последовательных этапов обработки данных. Алгоритм задается в произвольном изложении на естественном языке.

Например. Записать алгоритм нахождения наибольшего общего делителя (НОД) двух натуральных чисел (алгоритм Эвклида).

Словесный способ не имеет широкого распространения, так как такие описания:

- строго не формализуемы;
- страдают многословностью записей;
- допускают неоднозначность толкования отдельных предписаний.

Графический способ

Наибольшее распространение благодаря своей наглядности получил графический способ записи алгоритмов. При графическом представлении алгоритм изображается в виде последовательности связанных между собой функциональных блоков, каждый из которых соответствует выполнению одного или нескольких действий.

Такое графическое представление называется схемой алгоритма или блок-схемой. В блок-схеме каждому типу действий (вводу исходных данных, вычислению значений выражений, проверке условий, управлению повторением действий, окончанию обработки и т.п.) соответствует геометрическая фигура, представленная в виде блочного символа. Блочные символы соединяются линиями переходов, определяющими очередность выполнения действий. На рисунке 11 приведены наиболее часто употребляемые символы.

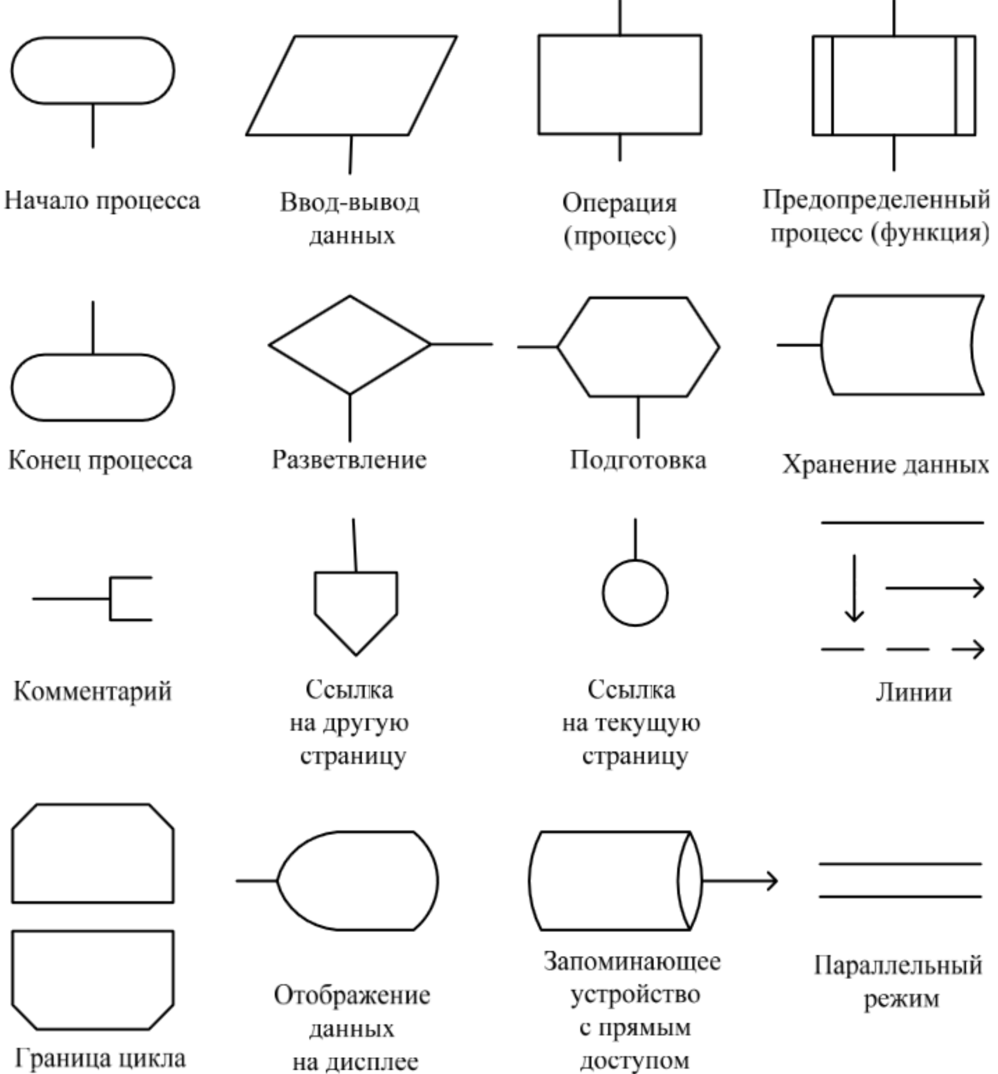


Рисунок 11 - Условные обозначения, используемые в блок-схемах

Для стандартизации и унификации языка схем алгоритмов в 1992 г. был принят ГОСТ 19.701–90 «Единая система программной документации. Схемы алгоритмов, программ, данных и систем. Условные обозначения и правила выполнения». В настоящее время данный стандарт продолжает действовать в Республике Беларусь.

Псевдокод

Псевдокод представляет собой систему обозначений и правил, предназначенную для единообразной записи алгоритмов.

Псевдокод занимает промежуточное место между естественным и формальным языками. С одной стороны, он близок к обычному естественному языку, поэтому алгоритмы могут на нем записываться и читаться как обычный текст. С другой стороны, в псевдокоде используются некоторые формальные конструкции и математическая символика, что приближает запись алгоритма к общепринятой математической записи.

В псевдокоде не приняты строгие синтаксические правила для записи команд, присущие формальным языкам, что облегчает запись алгоритма на

стадии его проектирования и дает возможность использовать более широкий набор команд, рассчитанный на абстрактного исполнителя.

Однако в псевдокоде обычно имеются некоторые конструкции, присущие формальным языкам, что облегчает переход от записи на псевдокоде к записи алгоритма на формальном языке. В частности, в псевдокоде, так же, как и в формальных языках, есть служебные слова, смысл которых определен раз и навсегда. Они выделяются в печатном тексте жирным шрифтом, а в рукописном тексте подчеркиваются.

Единого или формального определения псевдокода не существует, поэтому возможны различные псевдокоды, отличающиеся набором служебных слов и основных (базовых) конструкций.

Понятия транслятор, компилятор, интерпретатор. Синтаксическая и динамическая компиляция.

Транслятор - программа или техническое средство, выполняющее трансляцию программы.

Трансляция программы - преобразование программы, представленной на одном из языков программирования, в программу, написанную на другом языке. Транслятор обычно выполняет также диагностику ошибок, формирует словари идентификаторов, выдаёт для печати текст программы и т.д.

Цель трансляции - преобразование текста с одного языка на язык, понятный адресату. При трансляции компьютерной программы адресатом может быть:

- устройство -процессор (трансляция называется компиляцией);
- программа - интерпретатор (трансляция называется интерпретацией).

Виды трансляции:

- компиляция;
  - в исполняемый код
    - в машинный код
    - в байт-код
  - транспиляция;
- интерпретация;
- динамическая компиляция.

Компиляция

Язык процессора (устройства, машины) называется машинным языком, машинным кодом. Код на машинном языке исполняется процессором. Обычно, машинный язык - язык низкого уровня, но существуют процессоры, использующие языки высокого уровня. Однако, такие процессоры не получили распространения в силу своей сложности и дороговизны.

Компилятор - это вид транслятора, преобразующий исходный код с какого-либо языка программирования на машинный язык.

Достоинства компиляции:

- компиляция программы выполняется один раз;
- наличие компилятора на устройстве, для которого компилируется программа, не требуется.

Недостатки компиляции:

- компиляция — медленный процесс;
- при внесении изменений в исходный код, требуется повторная компиляция.

Интерпретация

Интерпретация - процесс чтения и выполнения исходного кода. Реализуется программой - интерпретатором.

Интерпретатор может работать двумя способами:

- читать код и исполнять его сразу (чистая интерпретация);
- читать код, создавать в памяти промежуточное представление кода (байт-код или p-код), выполнять промежуточное представление кода (смешанная реализация).

В первом случае трансляция не используется, а во втором - используется трансляция исходного кода в промежуточный код.

Достоинства интерпретаторов по сравнению с компиляторами:

- возможность работы в интерактивном режиме;
- отсутствие необходимости перекомпиляции исходного кода после внесения изменений и при переносе кода на другую платформу.

Недостатки интерпретаторов по сравнению с компиляторами:

- низкая производительность (машинный код исполняется процессором, а интерпретируемый код — интерпретатором; машинный код самого интерпретатора исполняется процессором);
- необходимость наличия интерпретатора на устройстве, на котором планируется интерпретация программы;
- обнаружение ошибок синтаксиса на этапе выполнения (актуально для чистых интерпретаторов).

Сравнение чистого интерпретатора и интерпретатора, создающего байт-код:

- чистый интерпретатор проще в реализации, так как для него не нужно писать код транслятора;
- интерпретатор, создающий байт-код, может выполнять его оптимизацию и добиваться большей производительности, чем чистый интерпретатор;

– интерпретатор, создающий байт-код, потребляет больше ресурсов системы (трансляция в байт-код занимает процессорное время; байт-код занимает место в памяти).

Динамическая и статическая компиляция

Компиляция – это процесс перевода исходного кода программы в машинный код, который может выполнять процессор. Существует два подхода к компиляции: статическая и динамическая.

Статическая компиляция

Статическая компиляция выполняется на этапе разработки, и скомпилированный код сохраняется в виде исполняемого файла. Этот файл не зависит от среды выполнения и может быть запущен на любом компьютере с совместимой архитектурой.

Преимущества статической компиляции:

1. Производительность: Скомпилированный код работает быстрее, так как все преобразования произошли до выполнения программы.
2. Отладка: Статически скомпилированные программы легче отлаживать, так как отсутствует связь с исходным кодом во время исполнения.
3. Безопасность: Код невозможно изменить после компиляции, что защищает его от модификаций.

Недостатки статической компиляции:

1. Гибкость: Статическая компиляция лишает программиста возможности изменять код во время выполнения.
2. Размер исполняемого файла: Скомпилированные программы могут занимать много места, особенно если они содержат дополнительные библиотеки.

Динамическая компиляция

Динамическая компиляция- трансляция, при которой исходный или промежуточный код преобразуется (компилируется) в машинный код непосредственно во время исполнения, «на лету» (англ. just in time, JIT). Компиляция каждого участка кода выполняется только один раз; скомпилированный код сохраняется в кеше и при необходимости используется повторно.

Достоинства динамической компиляции по сравнению с компиляцией:

– скорость работы динамически компилируемых программ близка к скорости работы компилируемых программ;
– отсутствие необходимости перекомпиляции программы при переносе на другую платформу.

Недостатки динамической компиляции по сравнению с компиляцией и чистой интерпретацией:

– большая сложность реализации;

– большие требования к ресурсам.

Динамическая компиляция хорошо подходит для веб-приложений.

Динамическая компиляция появилась и поддерживается в той или иной мере в реализациях Java, .NET Framework, Perl, Python.

Лекция 4 Интерфейс. Общие сведения

Варианты интерфейсов. Важность правильной разработки интерфейса. Различные методы построения диалога с пользователем.

Интерфейс пользователя, он же пользовательский интерфейс (UI — англ. user interface) - интерфейс, обеспечивающий передачу информации между пользователем-человеком и программно-аппаратными компонентами компьютерной системы (ISO/IEC/IEEE 24765-2010).

Под совокупностью средств и методов интерфейса пользователя подразумеваются:

1. Средства вывода информации из устройства к пользователю - весь доступный диапазон воздействий на организм человека (зрительных, слуховых, тактильных, обонятельных и т. д.) - экраны (дисплеи, проекторы) и лампочки, динамики, зуммеры и т.п.
2. Средства ввода информации/команд пользователем в устройство - множество всевозможных устройств для контроля состояния человека - кнопки, переключатели, потенциометры, датчики положения и движения, сервоприводы, жесты лицом и руками, даже съём мозговой активности пользователя.

По наличию тех или иных средств ввода, интерфейсы разделяются на типы - жестовый, голосовой и т. д., возможны смешанные варианты. Средства эти должны быть необходимыми и достаточными, быть удобными и практичными, расположенными и скомпонованными разумно и понятно, соответствовать физиологии человека, не должны приводить к негативным последствиям для организма пользователя .

Виды интерфейсов

1. Визуальный.
    a. текстовый (в частности, интерфейс командной строки);
    b. графический;
        i. оконный;
        ii. WIMP;
        iii. Web-ориентированный;
        iv. индуктивный;
        v. масштабируемый.
2. Тактильный.
3. Жестовый.

4. Голосовой.
5. Материальный (осязательный).
6. Нейрокомпьютерный интерфейс.

Важность правильной разработки интерфейса.

Дружественный пользователю интерфейс — это тот, который прост в использовании и навигации. Он понятен сам по себе и не требует от пользователя особых усилий, чтобы понять, как им пользоваться. Цель хорошего пользовательского интерфейса — сделать взаимодействие пользователя максимально гладким; в идеале он должен иметь возможность выполнить желаемую задачу без каких-либо затруднений и за минимально возможное время.

При проектировании интерфейса следует учитывать несколько ключевых принципов:

- Сохраняйте простоту. Чтобы интерфейс был простым для понимания и использования, он не должен содержать беспорядка и ненужных элементов.
- Будьте последовательны. Интерфейс должен использовать одну и ту же терминологию и элементы дизайна во всем приложении или на веб-сайте.
- Дайте обратную связь. Интерфейс должен предоставлять обратную связь пользователю. Например, если он нажимает кнопку, должно быть какое-то указание на то, что кнопка была нажата и в результате происходит действие.
- Используйте привычные соглашения. Несложно использовать привычные соглашения в вашем интерфейсе. Например, кнопка «Назад» должна располагаться в верхнем левом углу экрана, где она обычно находится на других сайтах и в приложениях.
- Предвосхищайте потребности пользователя. Интерфейс должен предвосхищать потребности пользователя, когда он взаимодействует с ним. Например, если он заполняет форму, следующее поле должно автоматически отображаться после заполнения текущего поля.
- Сделайте так, чтобы исправление ошибок было простым. Все совершают ошибки, и ваш интерфейс должен сделать исправление ошибок простым для пользователя . Например, если он случайно что-то удалил, должна быть кнопка «Отменить».
- Помогите пользователям распознавать, диагностировать и устранять ошибки — если кто-то совершает ошибку при взаимодействии с вашим интерфейсом, он должен иметь возможность понять это и двигаться дальше. Например, если обязательное поле не заполнено, ошибка должна отображаться красным цветом и ее должно быть легко исправить.
- Проектируйте для разных уровней знаний. Вероятно, что пользователи с разным уровнем знаний будут использовать приложение или веб-сайт, и все они должны иметь возможность ориентироваться в вашем

интерфейсе. Должен быть режим новичка и режим эксперта для определенных функций, если это возможно и применимо.

- Будьте гибкими. Интерфейс должен иметь возможность адаптироваться к разным размерам и разрешениям экрана.
- Тестирование удобства использования. Интерфейс должен быть протестирован на удобство использования, что может быть сделано путем проведения пользовательских исследований или A/B-тестирования.

Различные методы построения диалога с пользователем.

Диалоговая система – это система, созданная для общения с пользователем в естественном для него виде: в виде диалога. В первую очередь такие системы или, как их часто называют, боты, возникли для того, чтобы упростить взаимодействие людей с компаниями или сервисами. Например, спросить у умной колонки прогноз погоды быстрее, чем искать телефон, открывать браузер и печатать запрос.

По назначению можно выделить три основных типа диалоговых систем: общего назначения (general, чат-боты), задачеориентированные (task-oriented) и способные вести диалог на любую тему (open domain). Упрощенные версии диалоговых систем, такие как чат-боты имеют довольно простую архитектуру и состоят из набора правил и заранее подготовленных ответов.

Рисунок 12 – Структура целевой диалоговой системы

Современные интеллектуальные диалоговые системы имеют более сложную архитектуру, решают конкретную задачу пользователя и, как правило, взаимодействуют с внешним хранилищем данных.

Системы общего назначения не пытаются решить конкретную задачу. Их цель в том, чтобы поддержать разговор с пользователем на произвольную тему. Язык разметки базы знаний может включать в себя паттерны вопросов

и соответствующие им шаблоны ответов, также предысторию диалогов к ним и название соответствующей темы общения.

Существует два основных подхода к построению диалоговых систем генеративный и по исковый. В генеративном подходе ответ на сообщение пользователя порождается с помощью некоторой модели.

В настоящее время наиболее популярный способ построения таких систем - seq2seq-модели. Такие модели состоят из двух рекуррентных сетей: encoder-decoder. Coder строит представление входной последовательности слов. Далее полученное представление (последние выход и значение ячейки сети) копируются в decoder. По полученному представлению decoder пытается восстановить целевую последовательность слов.

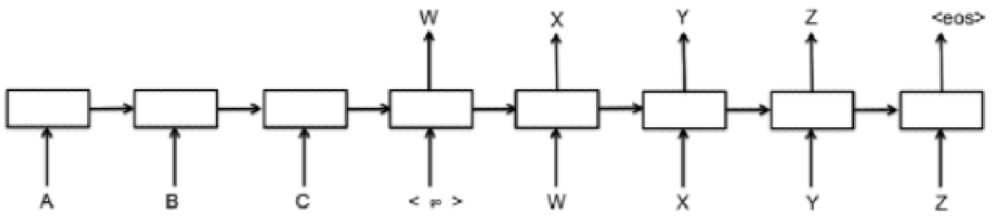


Рисунок 13 – Архитектура equence-to-sequence модели

В поисковом (селективном) подходе ответ на сообщение пользователя выбирается из большого набора готовых ответов. Как правило, для сообщения и всех ответов строятся векторные представления одной размерности, после чего ответ выбирается в соответствии с некоторой метрикой.

В селективных диалоговых моделях для каждой пары вопрос – ответ вычисляется некоторая ранжирующая функция, значение которой тем выше, чем более релевантна текущая реплика диалоговому контексту.

РАЗДЕЛ 2. ОСНОВЫ ПРОГРАММИРОВАНИЯ НА ЯЗЫКЕ С++

Лекция 5 Среда программирования

Microsoft Visual Studio 2010 доступна в следующих вариантах:

- **express** – бесплатная среда разработки, включающая только базовый набор возможностей и библиотек;
- **professional** – поставка, ориентированная на профессиональное создание программного обеспечения и командную разработку, при

которой созданием программы одновременно занимаются несколько человек;

- **premium** – издание, включающее дополнительные инструменты для работы с исходным кодом программ и создания баз данных;
- **ultimate** – наиболее полное издание Visual Studio, содержащее все доступные инструменты для написания, тестирования, отладки и анализа программ, а также дополнительные инструменты для работы с базами данных и проектирования архитектуры ПО.

Отличительной особенностью среды **Microsoft Visual Studio 2010** является то, что она поддерживает работу с несколькими языками программирования и программными платформами. Поэтому перед тем как писать программу на языке C/C++, необходимо выполнить несколько подготовительных шагов по созданию проекта и выбора и настройки компилятора языка C/C++ для трансляции исходного кода. После запуска **Microsoft Visual Studio 2010** появляется стартовая страница (рисунок 1).

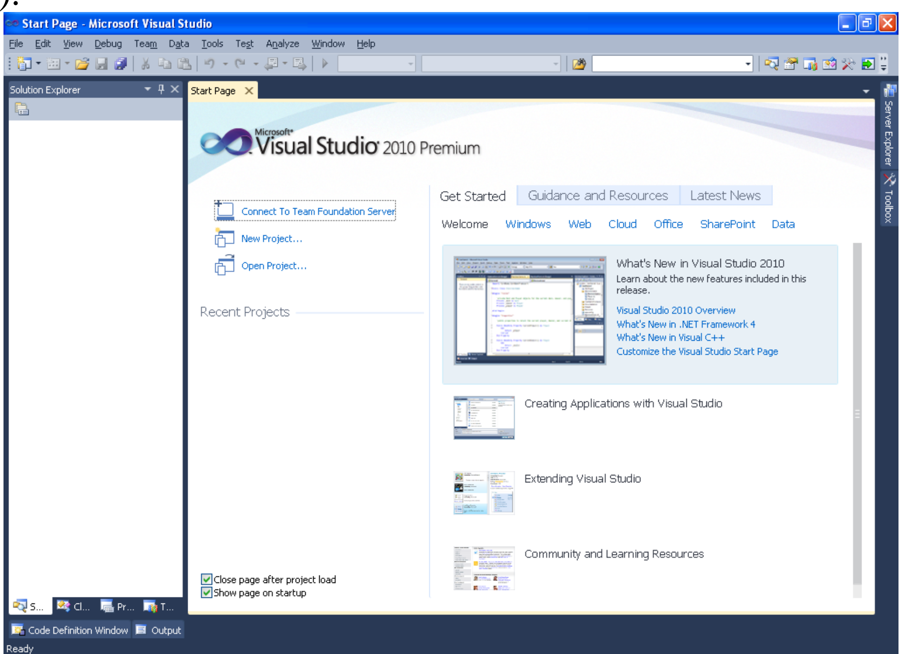


Рисунок 14 – Стартовая страница Visual Studio 2010

Следующим шагом является создание нового проекта. Для этого в меню **File** необходимо выбрать **New Project** (или нажать комбинацию клавиш **Ctrl** + **Shift** + **N**). Результат выбора пунктов меню для создания нового проекта показан на рисунке 14.

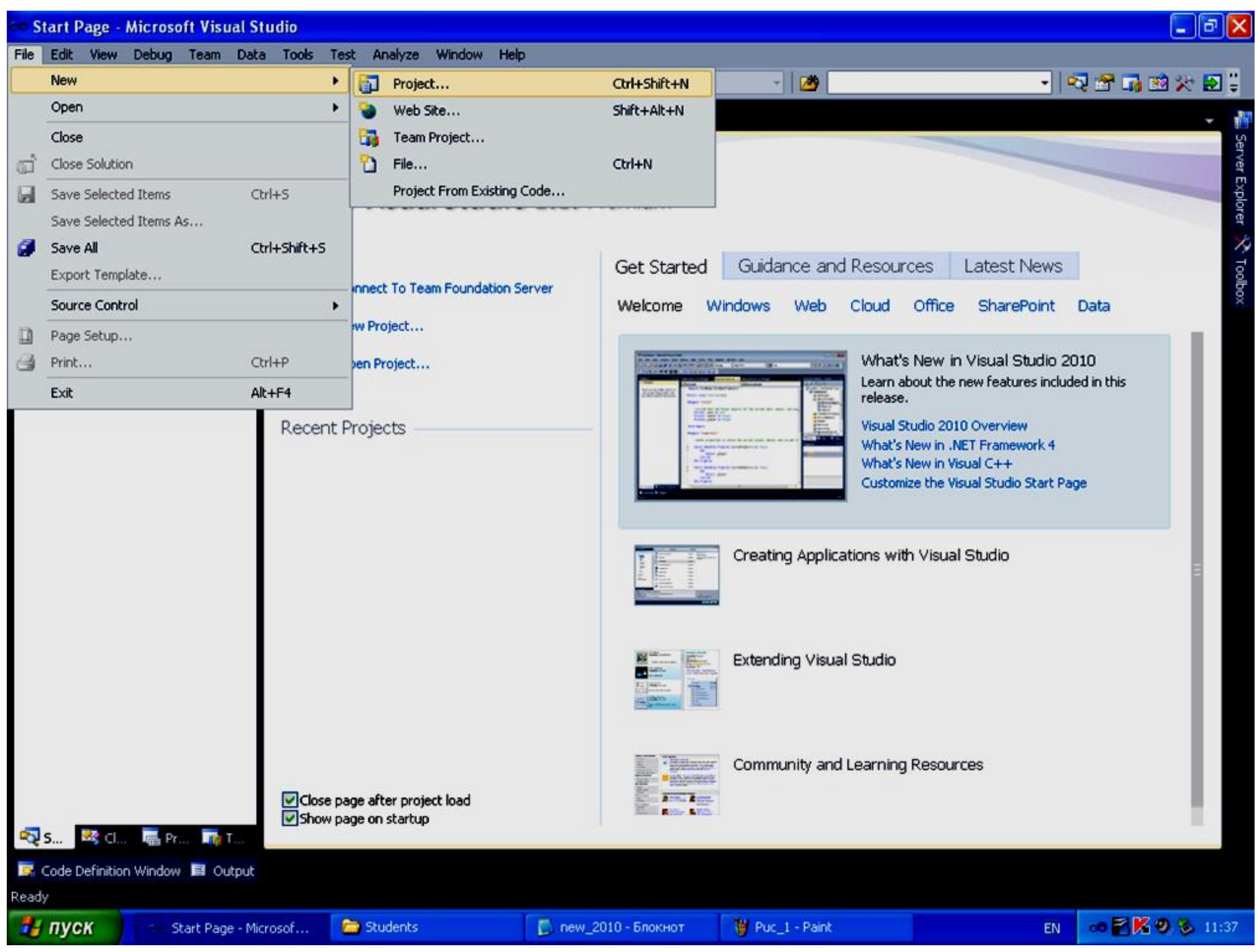


Рисунок 15 – Окно с выбором нового проекта

Среда Visual Studio отобразит окно **New Project** (Создать проект), в котором необходимо выбрать тип создаваемого проекта. *Проект* (project) используется в Visual Studio для логической группировки нескольких файлов, содержащих исходный код, на одном из поддерживаемых языков программирования, а также любых вспомогательных файлов. Обычно после сборки проекта (которая включает компиляцию всех всходящих в проект файлов исходного кода) создается один исполняемый модуль.

В окне **New Project** следует развернуть узел **Visual C++**, затем обратиться к пункту Win32 и на центральной панели выбрать Win32 Console Application (рисунок 16).

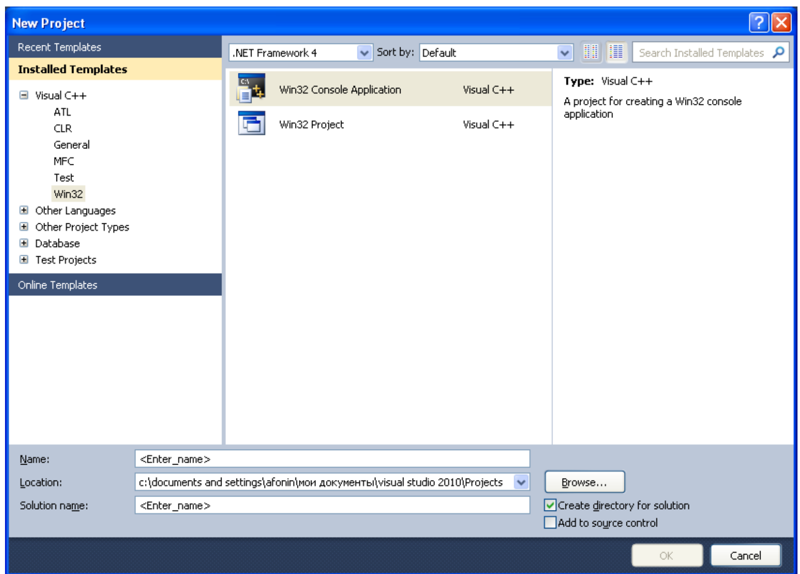


Рисунок 17 – Выбор типа проекта

После выбора типа проекта в поле редактора **Name** необходимо ввести его имя, например **hello**. В поле **Location** можно указать путь размещения проекта или выбрать его (путь) с помощью клавиши (кнопки) **Browse**. По умолчанию проект сохраняется в специальной папке **Projects**. Выбор имени проекта может быть достаточно произвольным: допустимо использовать числовое значение, допустимо имя задавать через буквы русского алфавита. В дальнейшем будем давать проекту имя, набранное с помощью букв латинского алфавита и, может быть, с добавлением цифр. Пример выбора имени проекта показан на рисунке 18.

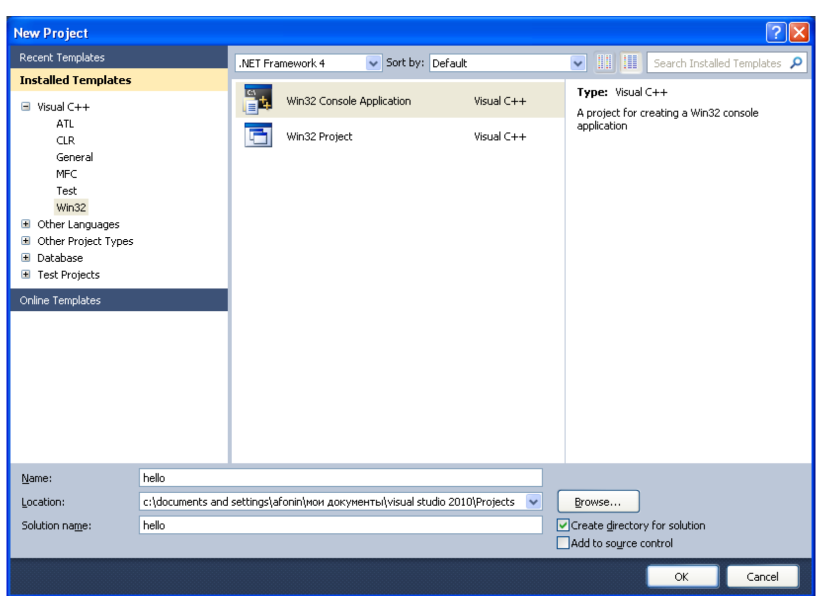


Рисунок 18 – Пример задания имени проекта

Одновременно с проектом Visual Studio создает решение. *Решение* (solution) – это способ объединения нескольких проектов для организации более удобной работы с ними.

После нажатия кнопки **OK** откроется окно **Win32 Application Wizard** (мастер создания приложений для операционных систем Windows) (рисунок 19)

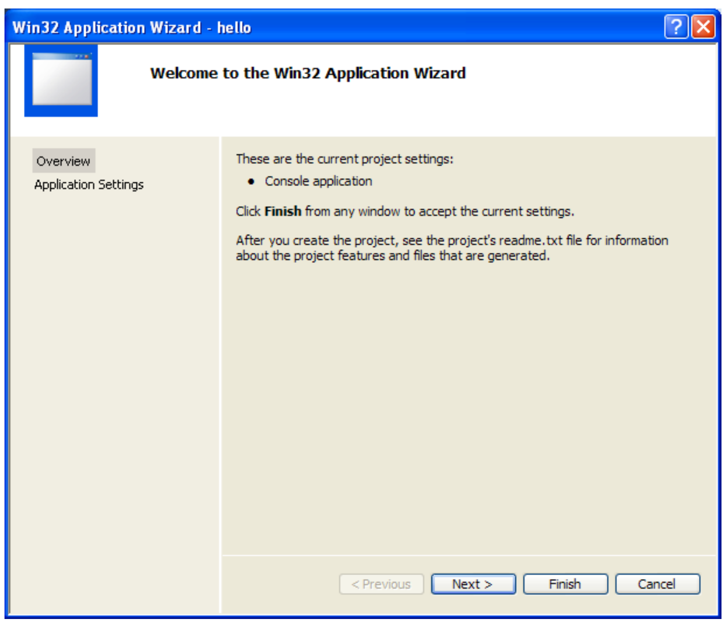


Рисунок 19 – Мастер создания приложения

На первой странице мастера представлена информация о создаваемом проекте, на второй можно сделать его первичные настройки. После обращения к странице **Application Settings** или нажатия кнопки **Next** получим окно, представленное на рисунке 20.

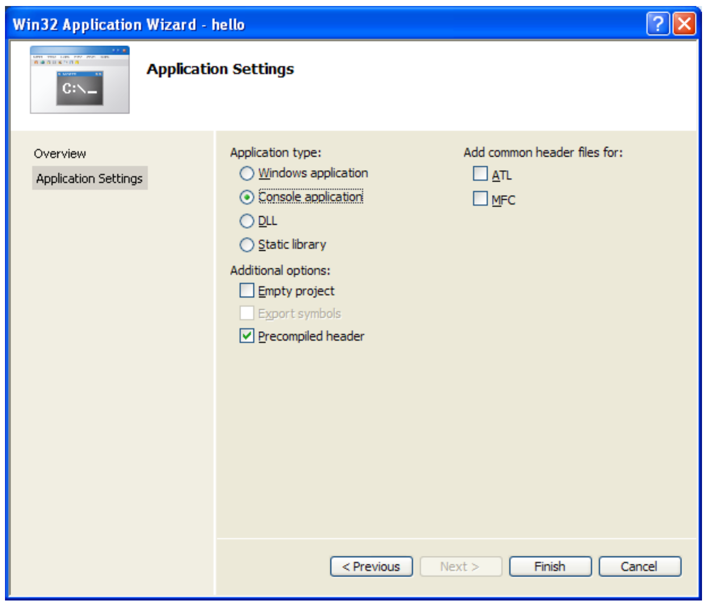


Рисунок 20– Страница мастера настройки проекта по умолчанию

В дополнительных опциях (**Additional options**) следует поставить галочку в поле **Empty project** (пустой проект) и снять (убрать) ее в поле **Precompiled header** (рисунок 21)

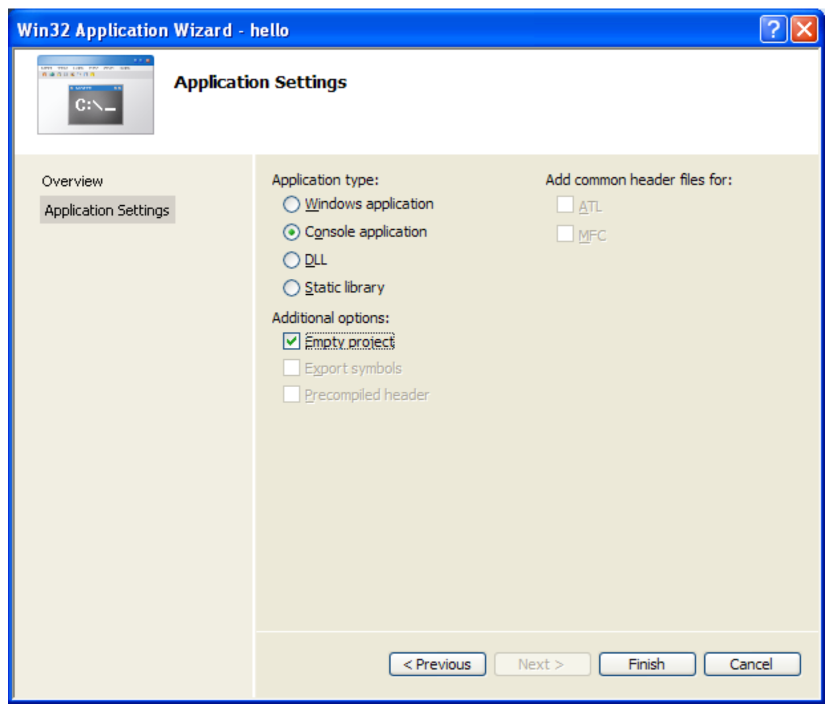


Рисунок 21 – Выполненная настройка мастера приложений

Создадим проект по приведенной схеме, т. е. проект в консольном приложении, который выполняется целиком программистом (за счет выбора **Empty project**). После нажатия кнопки **Finish** получим экранную форму (рисунок 8), где приведена последовательность действий добавления файла для создания исходного кода к проекту. Стандартный путь для этого: подвести курсор мыши к папке **Source Files** (файлы исходного кода) из узла **hello** в левой части открытого проекта приложения, щелкнуть правой кнопкой мыши и выбрать **Add**, затем **New Item** (создать новый элемент).
Экран Visual C++ разделен на четыре основные зоны. Сверху расположены **меню** и **панели инструментов**. Кроме них рабочий стол Visual C++ включает в себя три окна:

Окно **Project Workspace** (окно рабочей области) – расположено в левой части. Первоначально окно закрыто, но после создания нового проекта или загрузки существующего проекта это окно будет содержать несколько вкладок.

Окно **Editor** (окно редактирования), расположено справа. Его используют для ввода, проверки и редактирования исходного кода программы.

Окно **Output** (окно вывода) служит для вывода сообщений о ходе компиляции, сборки и выполнения программы и сообщений о возникающих ошибках.

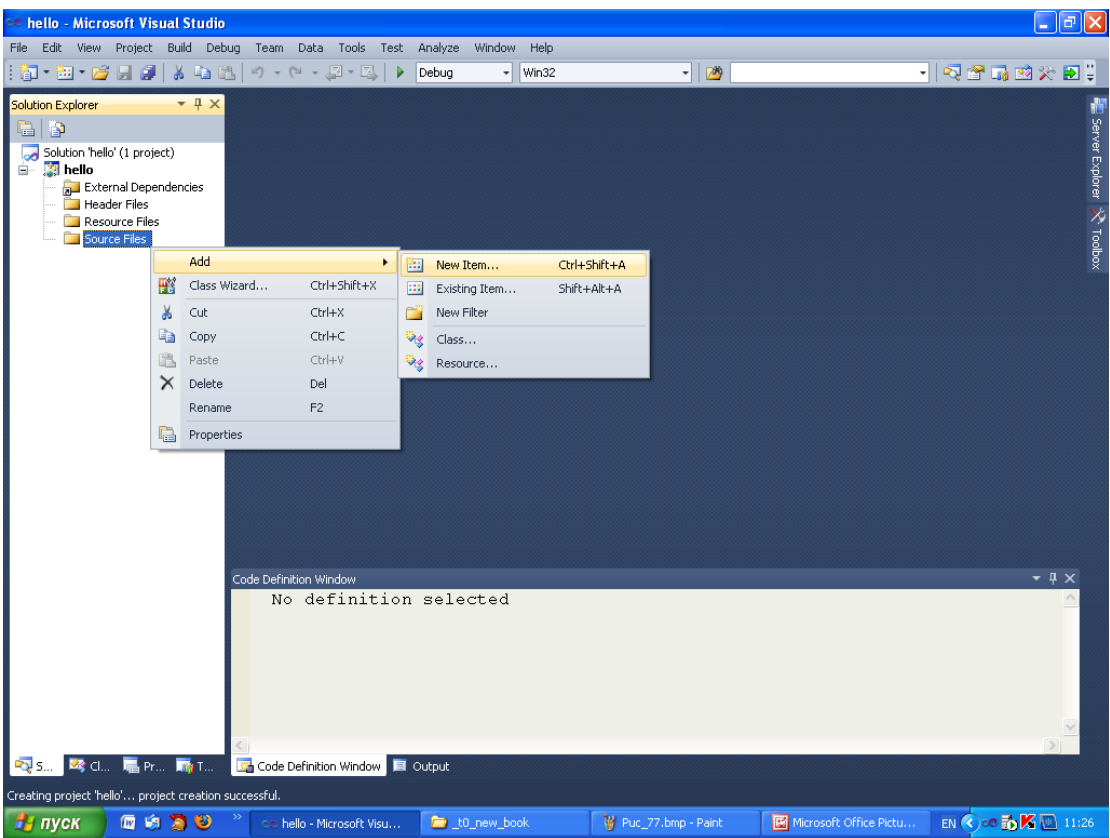
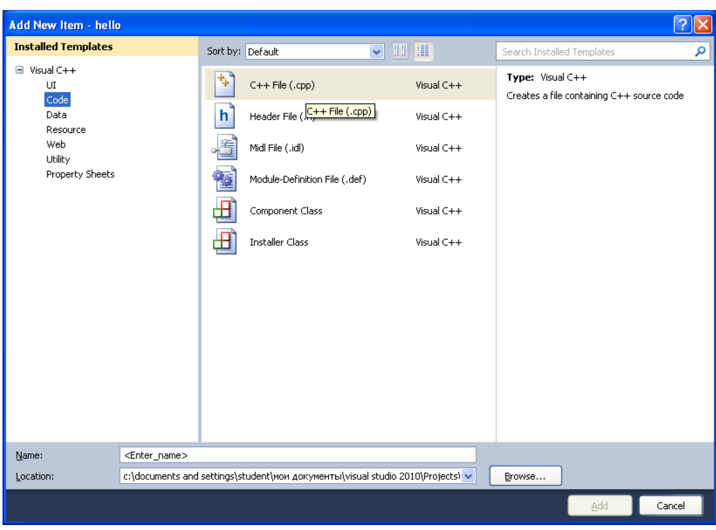


Рисунок 22 – Меню добавления нового элемента к проекту

В результате получим окно (рисунок 23),где через пункт меню **Code** узла **Visual C++** выполнено обращение к центральной части панели, в которой осуществляется выбор типа файлов. В данном случае требуется обратиться к закладке **C++ File** (.cpp).



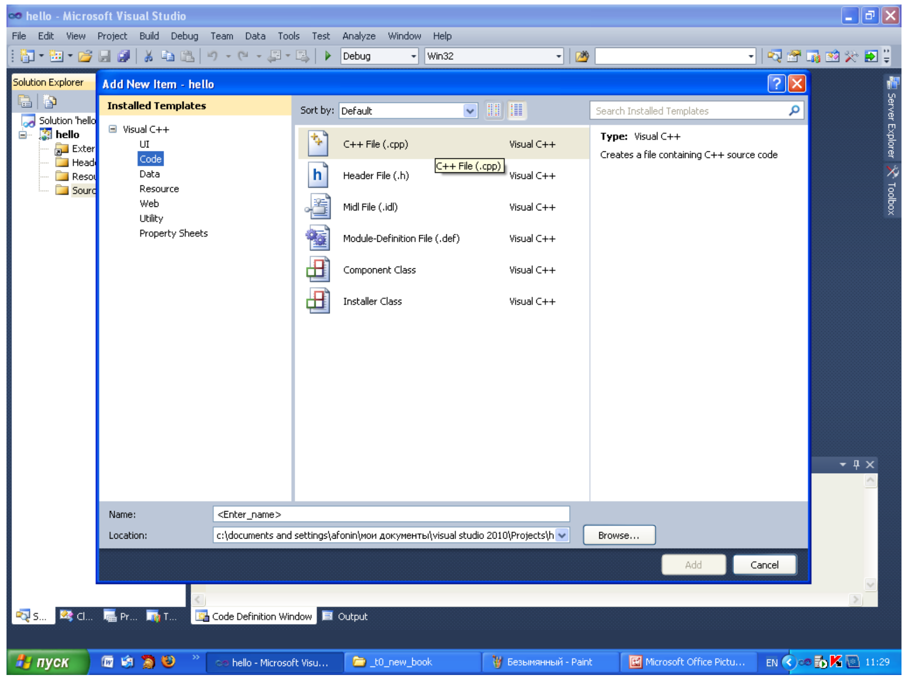


Рисунок 24 – Окно выбора типа файла для подключения к проекту

Теперь в поле редактора **Name** (в нижней части окна) следует задать имя нового файла и указать расширение **.cpp**, например, **main.cpp**. Имя может быть достаточно произвольным, но имеется негласное соглашение, что оно должно отражать назначение файла и логически описывать исходный код, который в нем содержится. В проекте, состоящем из нескольких файлов, есть смысл выделить файл, содержащий главную функцию программы, т. е. ту, с которой она начнет выполняться. Такому файлу будем задавать имя **main.cpp**, где расширение **.cpp** указывает на то, что этот файл содержит исходный код на языке **C++**, и он будет транслироваться соответствующим компилятором. Программам на языке **C++** принято давать указанное расширение. После задания имени файла в поле редактора Name получим форму, приведенную на рисунке 25.

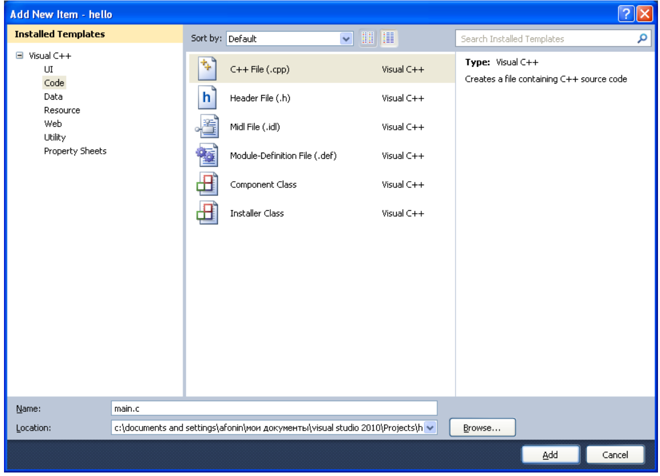


Рисунок 25 – Задание имени файла, подключаемого к проекту

Затем следует нажать кнопку **Add**. Вид среды Visual Studio после добавления первого файла к проекту показан на рисунке 26. Добавленный файл отображается в дереве **Solution Explorer** (обозреватель решений) под узлом **Source Files** (файлы с исходным кодом), и для него автоматически открывается редактор.

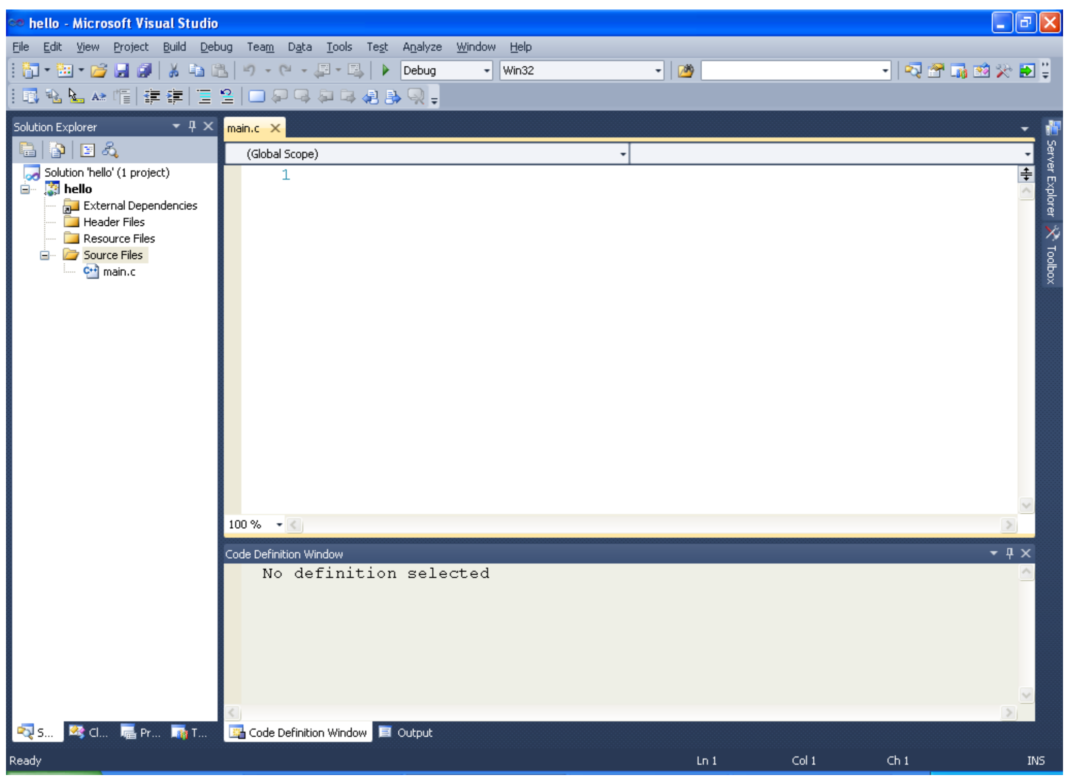


Рисунок 26 – Подключение файла проекта

На рисунке 26 в левой панели в папке **Solution Explorer** отображаются файлы, включенные в проект в папках. Приведем их описание.

Папка **Source Files** предназначена для файлов с исходным кодом. В ней отображаются файлы с расширением **.cpp**.

Папка **Header Files** (заголовичные файлы) содержит заголовочные файлы с расширением **.h**.

В папке **Resource Files** (файлы ресурсов) представлены файлы ресурсов, например изображения и т. д.

Папка **External Dependencies** (внешние зависимости) отображает файлы, не добавленные явно в проект, но использующиеся в файлах исходного кода, например включенные при помощи директивы #include. Обычно в этой папке присутствуют заголовочные файлы стандартной библиотеки, применяющиеся в проекте.

Следующий шаг состоит в настройке проекта. Для этого в меню **Project** главного меню следует выбрать **hello Properties** (или одновременно нажать клавиши Alt + F7). Пример обращения к этому пункту меню показан на рисунке 27.

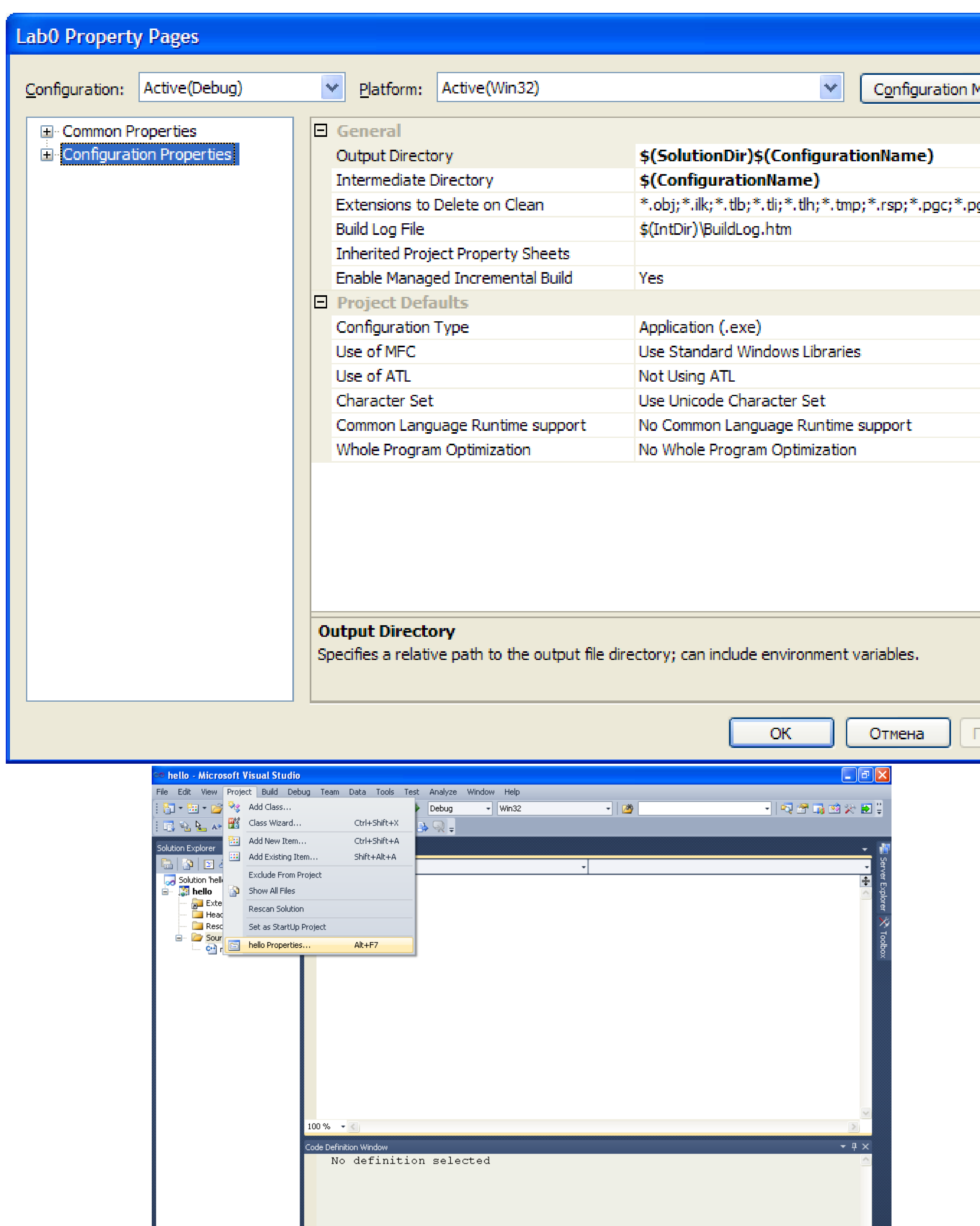


Рисунок 27 – Обращение к странице свойств проекта

После того как откроется окно свойств проекта, следует обратиться (с левой стороны) к **Configuration Properties** (Свойства конфигурации) ; появится ниспадающий список (рисунок 13) Далее нужно обратиться к узлу **Projects Defaults** (значения по умолчанию для проекта) и через него в левой панели выбрать **Character Set** (набор символов), где установить свойство **Use**

Multi-Byte Character Set. Настройка **Character Set** (набор символов) позволяет определиться, какая кодировка символов – ANSI или UNICODE – будет использована при компиляции программы. Для совместимости со стандартом C89 можно выбрать **Use Multi-Byte Character Set**. Это позволяет применять многие привычные функции, например по выводу информации на консоль.

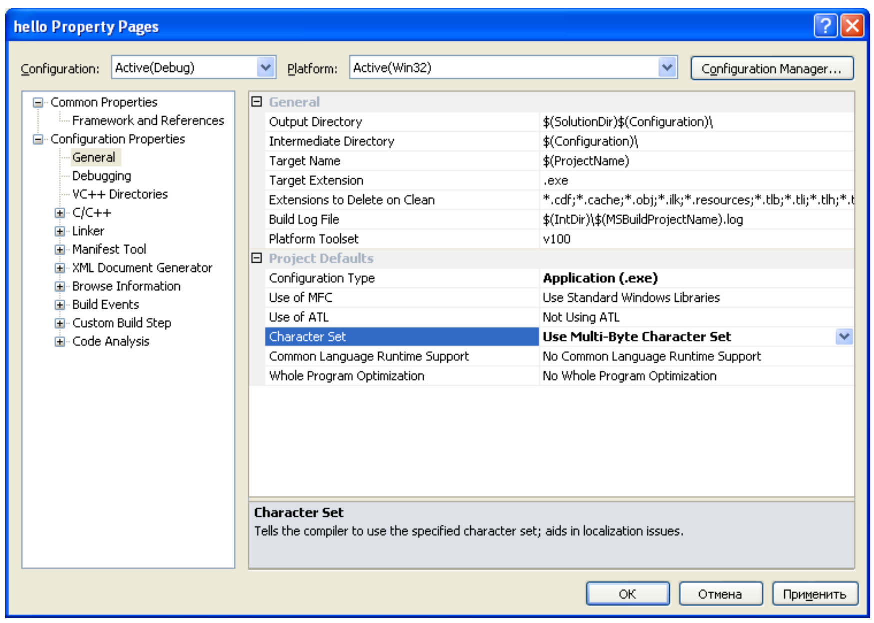


Рисунок 28 – Меню списка свойств проекта

После осуществления выбора (см. рисунок 29) следует нажать кнопку **Применить**.

Затем необходимо выбрать узел **C/C++** и в ниспадающем меню выбрать пункт **Code Generation** (создание кода), через который в правой части панели обратиться к закладке **Enable C++ Exceptions** (включить C++ исключения), установив для нее **No** (запрещение исключений **C++**). Результат установки выбранного свойства представлен на рис. 14. Затем нужно нажать кнопку **Применить**.

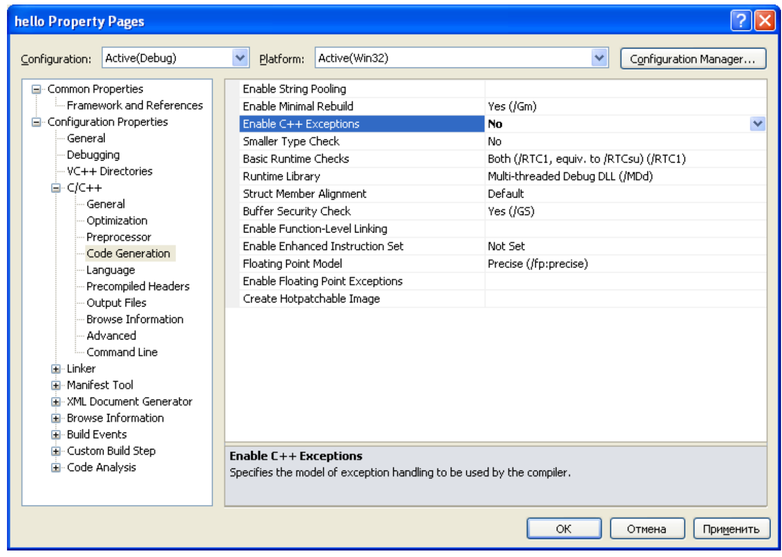


Рисунок 29 – Страница свойств для запрещения исключений С++

Далее в ниспадающем меню узла **C/C++** необходимо выбрать пункт **Language** и через него обратиться в правую часть панели, где установить следующие значения: для свойства **Disable Language Extensions** (отключить расширеня языка) – **Yes (/Za),** для **Treat Wchar_t As Built in Type** (считать тип wchar_t как встроенный тип) – **No (/Zc:wchar_t–),** для **Force Conformance in For Loop Scope** (соответствие стандарту определения локальных переменных в операторе цикла for) –**Yes(Zc:forScope)**, для **Enable Run-Time Type Information** (разрешить информацию о типах во время выполнения) – **No (/GR–)**, для свойства **Open MP Support** (разрешить расширение Open MP; используется при написании программ для многопроцессорных систем) – **No(/openmp–)**. Результат выполнения этих действий показан на рисунке 30.

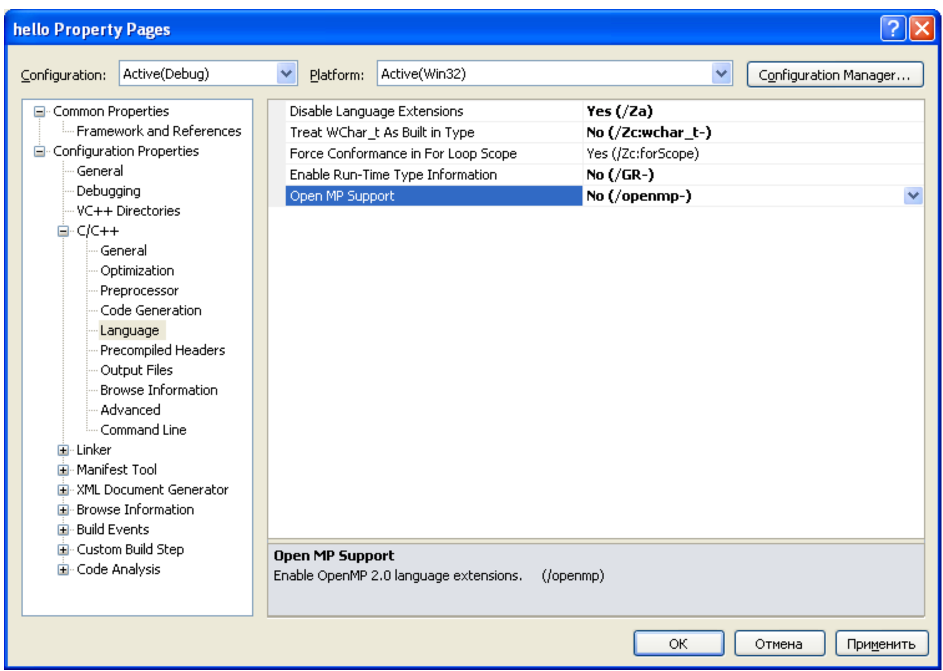


Рисунок 30 – Страница свойств закладки Language

После выполнения указанных действий следует нажать клавишу **Применить**. Далее в ниспадающем списке узла **C/C++** следует выбрать пункт **Advanced** (дополнительно) и в правой панели изменить свойство **Compile As** (компилировать как) в свойство компиляции языка **С**, т.е. **Compile as C Code (/TC)**, если вы работаете с программой на языке **С**. Или в ниспадающем списке узла **C/C++** следует выбрать пункт **Advanced** (дополнительно) и в правой панели изменить свойство **Compile As** (компилировать как) в свойство компиляции языка, т.е. **Compile as C++ Code (/TP)**, если вы работаете с программой на языке **С++**. Результат установки компилятора языка **С** представлен на рисунке 31.1. Результат установки компилятора языка **С++** представлен на рисунке 31.2

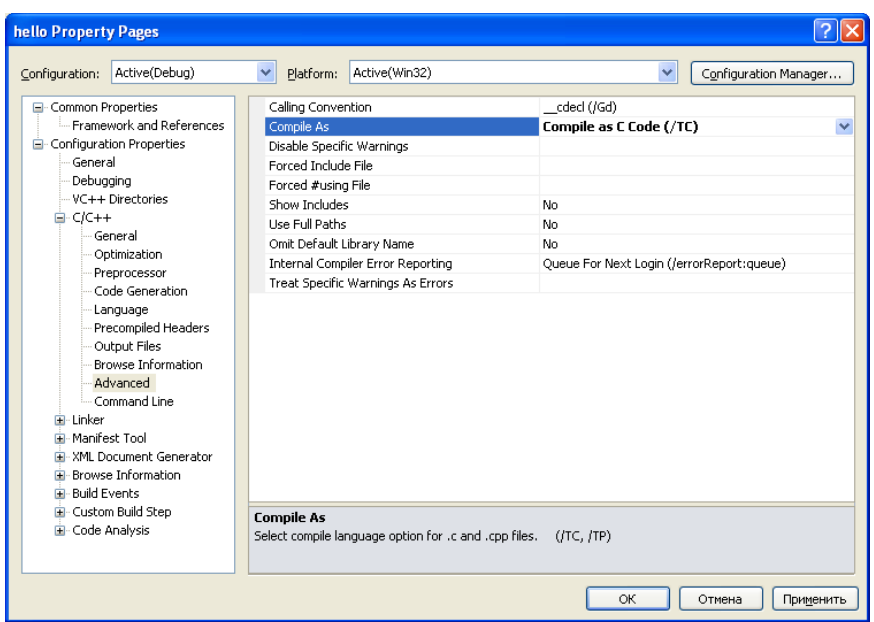


Рисунок 31.1 – Результат выбора режима компиляции языка **C**

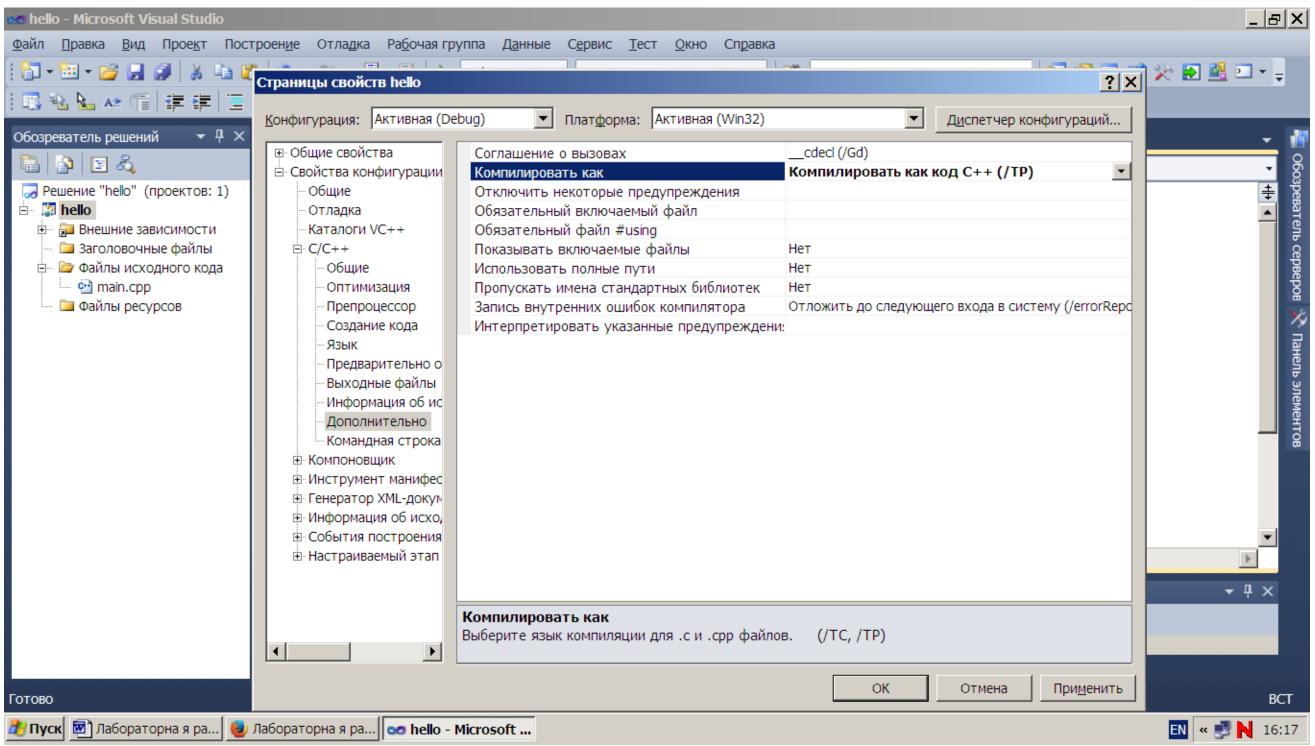


Рисунок 31.2 – Результат выбора режима компиляции языка **C++**

После нажатия клавиш **Применить** и **OK** откроется подготовленный проект с пустым полем редактора кода, в котором можно начать писать программы. В этом редакторе наберем программу, выводящую традиционное приветствие «Hello, World». Для компиляции созданной программы можно обратиться в меню **Build** или, например, нажать клавиши **Ctrl** + **F7**. В случае успешной компиляции получим экранную форму, показанную на рисунок 32.1 (пример для кода C). Рисунок 32.2 пример для кода C++.

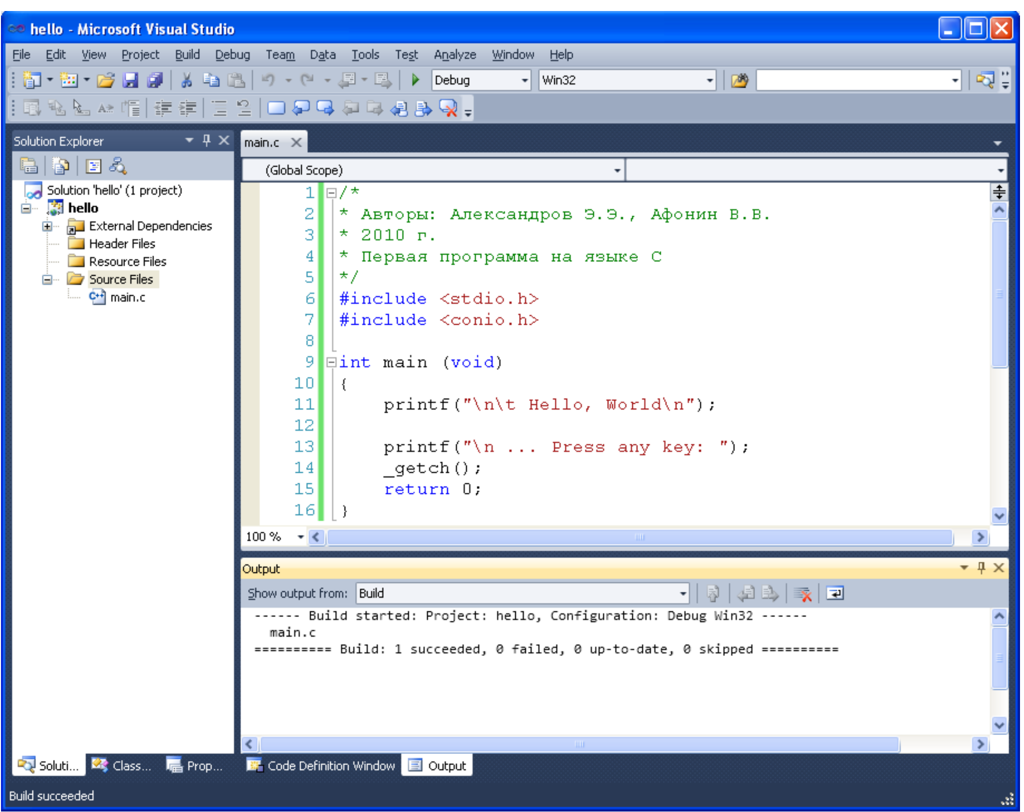


Рисунок 32.1 – Успешно скомпилированная первая программа на языке **C**

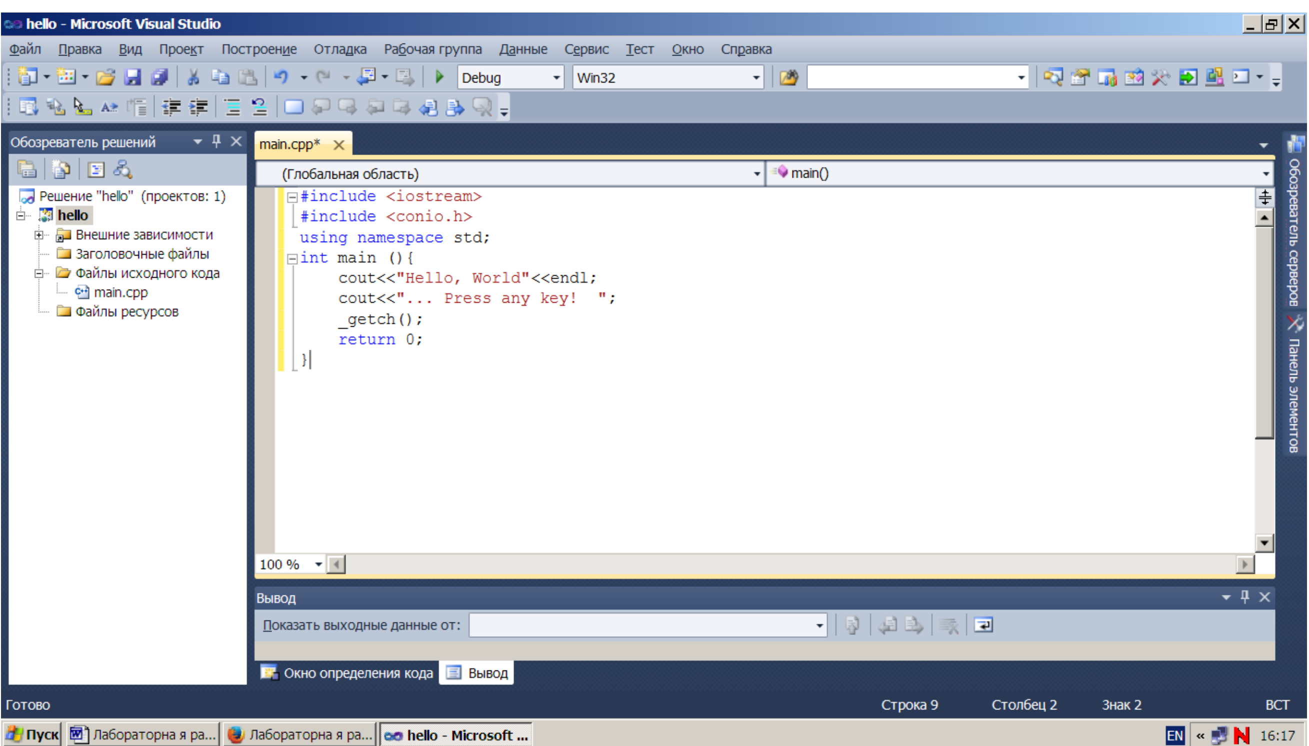


Рисунок 32.2 – Успешно скомпилированная первая программа на языке **C**

Для приведенного кода программы запуск на ее исполнение из окна редактора в Visual Studio 2010 выполняется нажатием клавиши F5 (рисунок 33).

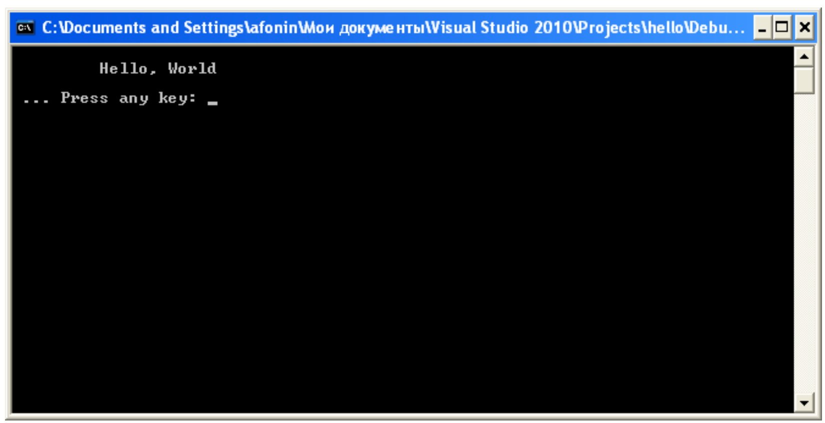


Рисунок 33 – Консольный вывод первой программы на языке **C**

Произведем разбор первой программы на языке **C.**

Во-первых, отметим, что в языке **C** нет стандартных инструкций (операторов) для вывода сообщений на консоль (окно пользователя), а предусмотрены специальные библиотечные файлы, в которых имеются функции для этих целей. В приведенной программе используется заголовочный файл с именем **stdio.h** (стандартный ввод-вывод), который должен быть включен в начало программы. Для вывода сообщения на консоль используется функция **printf**(). Для работы с консолью в программу включен также заголовочный файл **conio.h**, поддерживающий функцию **_getch().** Строка программы

int main (void)

сообщает системе, что именем программы является **main()** – главная функция и что она возвращает целое число, о чем указывает аббревиатура **int**. Имя **main()** – это специальное имя, которое указывает, где программа должна начать выполнение. Наличие круглых скобок после слова **main** свидетельствует о том, что это имя функции. Если содержимое этих скобок отсутствует или в них представлено служебное слово **void**, то это означает, что в функцию **main()** не передается никаких аргументов. Тело функции **main()** ограничено парой фигурных скобок. Все утверждения программы, заключенные в них, будут относиться к этой функции.

В теле функции **main()** имеются еще три инструкции. Вызов функции **printf(),** которая находится в библиотеке компилятора языка **C** и печатает или отображает те аргументы, которые были подставлены вместо параметров. Символ **n** означает единый символ newline (новая строка), т.е. с его помощью выполняется перевод на новую строку, символ **t** осуществляет табуляцию, т. е. начало вывода результатов программы с отступом вправо. Функция без параметров **_getch()** извлекает символ из потока ввода, т.е. предназначена для приема сообщения о нажатии какой-либо (не функциональной) клавиши на клавиатуре. С другими компиляторами, может потребоваться **getch()** без префиксного нижнего подчеркивания.

Последнее утверждение в первой программе **return 0;** указывает на то, что выполнение функции **main()** закончено и что в систему возвращается значение **0** (целое число). Нуль используется в соответствии с соглашением об индикации успешного завершения программы.

Произведем разбор первой программы на языке **С++.**

iostream — заголовочный файл с классами, функциями и переменными для организации ввода-вывода в языке программирования C++. Он включён в стандартную библиотеку C++. Название образовано от Input/Output Stream («поток ввода-вывода»). В языке C++ и его предшественнике, языке программирования Си, нет встроенной поддержки ввода-вывода, вместо этого используется библиотека функций. iostream управляет вводом-выводом, как и stdio.h в Си. iostream использует объекты **cin, cout, cerr** и **clog** для передачи информации и из стандартных потоков ввода, вывода, ошибок (без буферизации) и ошибок (с буферизацией) соответственно. Являясь частью стандартной библиотеки C++, эти объекты также являются частью стандартного пространства имён — std.

using namespace std;

Все типы и функции стандартной библиотеки C++ объявлены в пространстве имен std или в пространстве имен, вложенном в std.

Пространство имён (англ. namespace) — некоторое множество, под которым подразумевается модель, абстрактное хранилище или окружение, созданное для логической группировки уникальных идентификаторов (то есть имён).

Идентификатор, определённый в пространстве имён, ассоциируется с этим пространством. Один и тот же идентификатор может быть независимо определён в нескольких пространствах. Таким образом, значение, связанное с идентификатором, определённым в одном пространстве имён, может иметь (или не иметь) такое же значение, как и такой же идентификатор, определённый в другом пространстве. Языки с поддержкой пространств имён определяют правила, указывающие, к какому пространству имён принадлежит идентификатор (то есть его определение).

Для работы с консолью в программу включен также заголовочный файл **conio.h**, поддерживающий функцию **_getch().** Строка программы

int main ( )

сообщает системе, что именем программы является **main()** – главная функция и что она возвращает целое число, о чем указывает аббревиатура **int**. Имя **main()** – это специальное имя, которое указывает, где программа должна начать выполнение. Наличие круглых скобок после слова **main** свидетельствует о том, что это имя функции. Если содержимое этих скобок отсутствует или в них представлено служебное слово **void**, то это означает, что в функцию **main()** не передается никаких аргументов. Тело функции

main() ограничено парой фигурных скобок. Все утверждения программы, заключенные в них, будут относиться к этой функции.

В теле функции **main**() имеются еще три инструкции. Строка программы

cout<<"Hello, World"<<endl;

cout<<"... Press any key! ";

Библиотека iostream определяет три стандартных потока:

- **cin** стандартный входной поток (stdin в С)
- **cout** стандартный выходной поток (stdout в С)
- **cerr** стандартный поток вывода сообщений об ошибках (stderr в С)

Для их использования в Microsoft Visual Studio необходимо прописать строку:

using namespace std;

Для выполнения операций ввода-вывода переопределены две операции поразрядного сдвига:

- >> получить из входного потока
- << поместить в выходной поток

Функция без параметров **_getch()** извлекает символ из потока ввода, т.е. предназначена для приема сообщения о нажатии какой-либо (не функциональной) клавиши на клавиатуре. С другими компиляторами, может потребоваться **getch()** без префиксного нижнего подчеркивания.

Последнее утверждение в первой программе **return 0;** указывает на то, что выполнение функции **main**() закончено и что в систему возвращается значение **0** (целое число). Нуль используется в соответствии с соглашением об индикации успешного завершения программы.

Все файлы проекта сохраняются в той папке, которая сформировалась после указания в поле **Location** имени проекта (hello). На рисунке34.1 и 34.2 показаны папки и файлы проекта **Visual Studio 2010**.

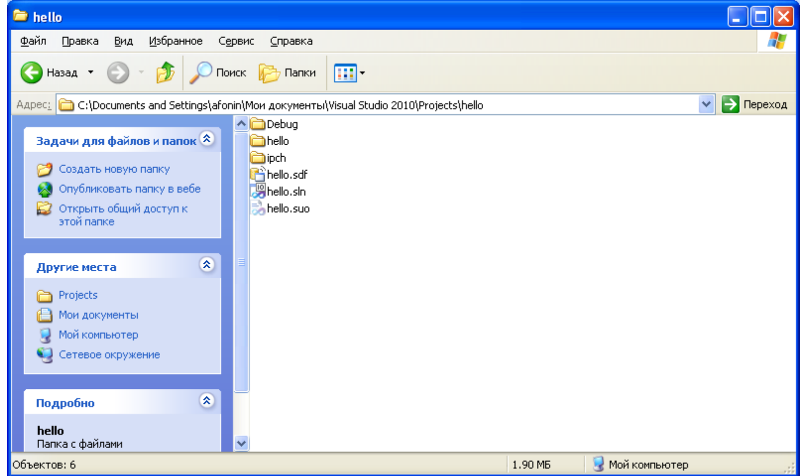


Рисунок 34.1 – Файлы и папки созданного проекта для проекта С

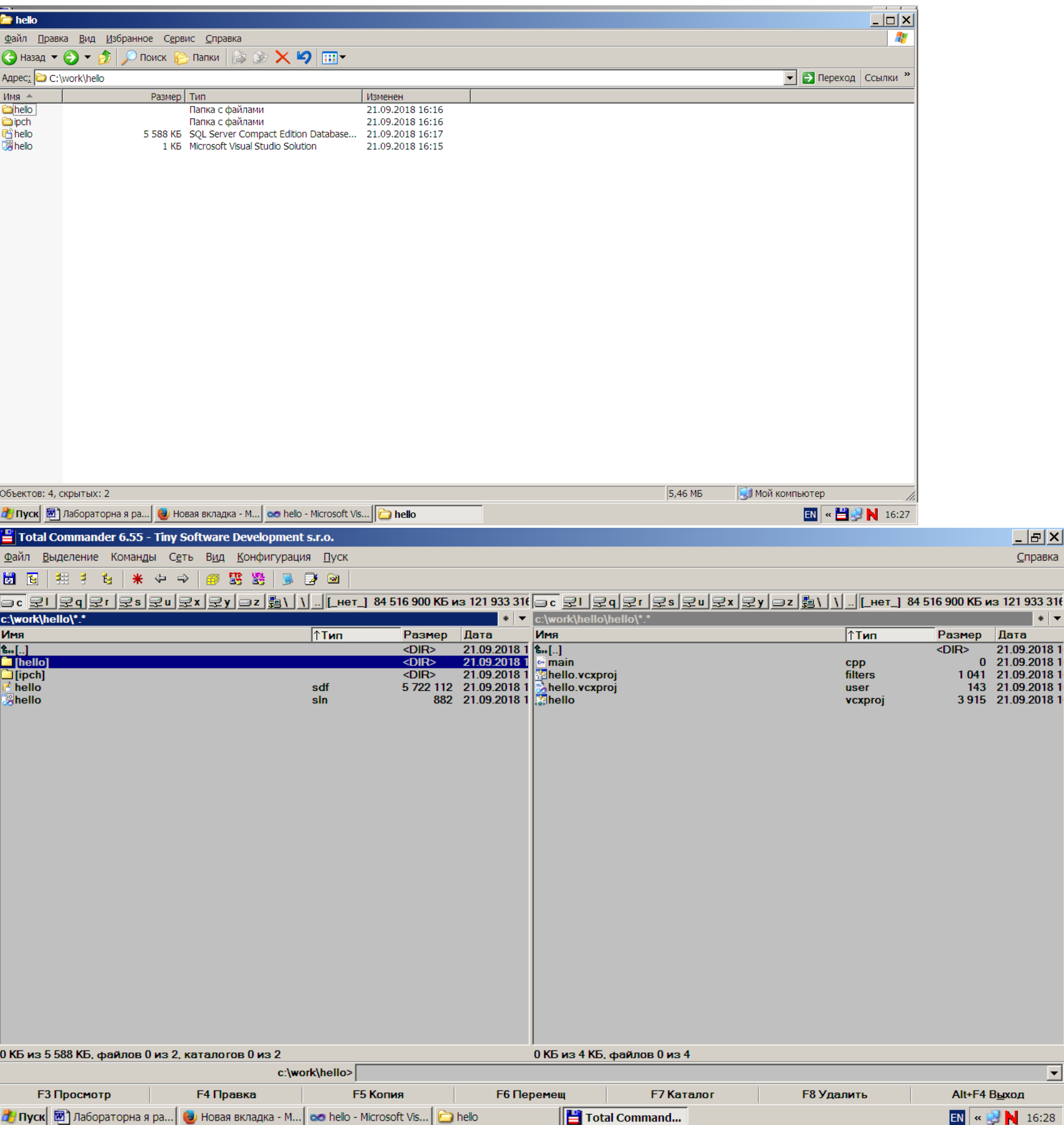


Рисунок 34.2 – Файлы и папки созданного проекта для проекта C++

Каждый файл обладает некоторым значением:

- **hello.sln** – файл решения для созданной программы. Он содержит информацию о том, какие проекты входят в данное решение. Обычно эти проекты расположены в отдельных подкаталогах. Например, наш проект находится в подкаталоге hello;
- **hello.suo** – файл настроек среды Visual Studio при работе с решением включает информацию об открытых окнах, их расположении и прочих пользовательских параметрах.
- **hello.sdf** – файл, содержащий вспомогательную информацию о проекте, который используется инструментами анализа кода Visual

Studio, такими как IntelliSense для отображения подсказок об именах и т. д.

Файлы папки **Debug** представлены на рисунке 35.1 и рисунке 35.2

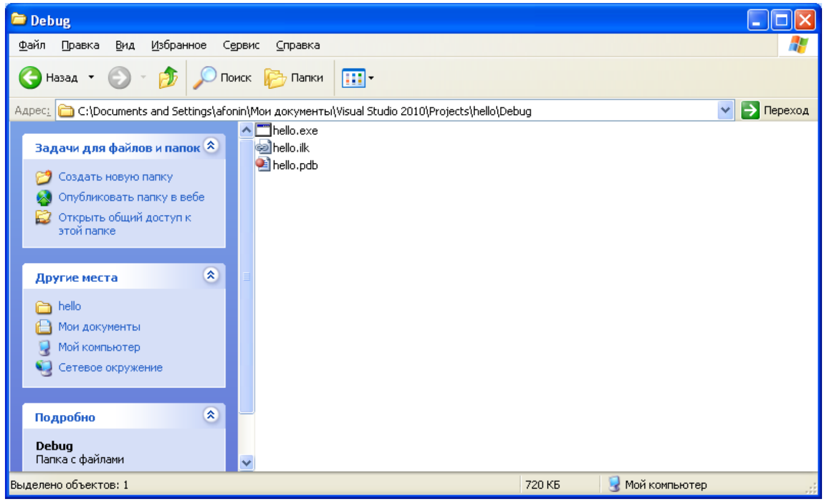


Рисунок 35.1 – Файлы папки Debug для проекта C

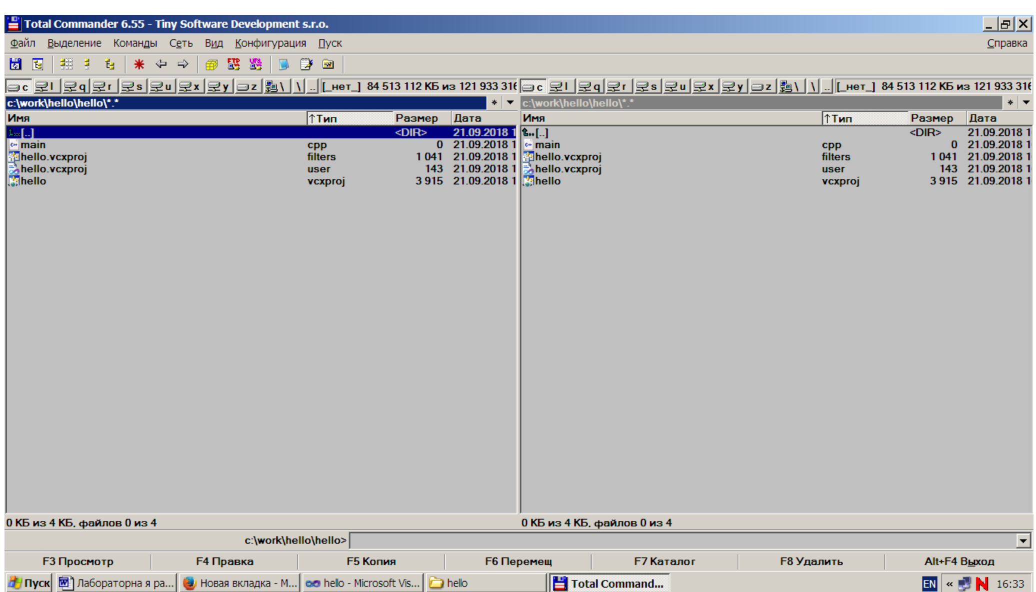


Рисунок 35.2 – Файлы папки Debug для проекта C++

Характеристика содержимого папки **hello**:

1. **main.c** / **main.cpp**– файл исходного программного кода;
2. **hello.vcxproj** – файл проекта;
3. **hello.vcxproj.user** – файл пользовательских настроек, связанных с проектом;
4. **hello.vcxproj.filters** – файл с описанием фильтров, используемых Visual Studio Solution Explorer для организации и отображения файлов с исходным кодом.

Отладка в Visual Studio

Отладка – неотъемлемый этап цикла разработки приложений, зачастую более важный, чем написание кода. Именно отладка позволяет

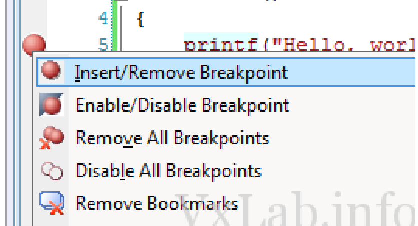
устранить проблемные места в коде, которые приводят к разного рода ошибкам.

Точки останова

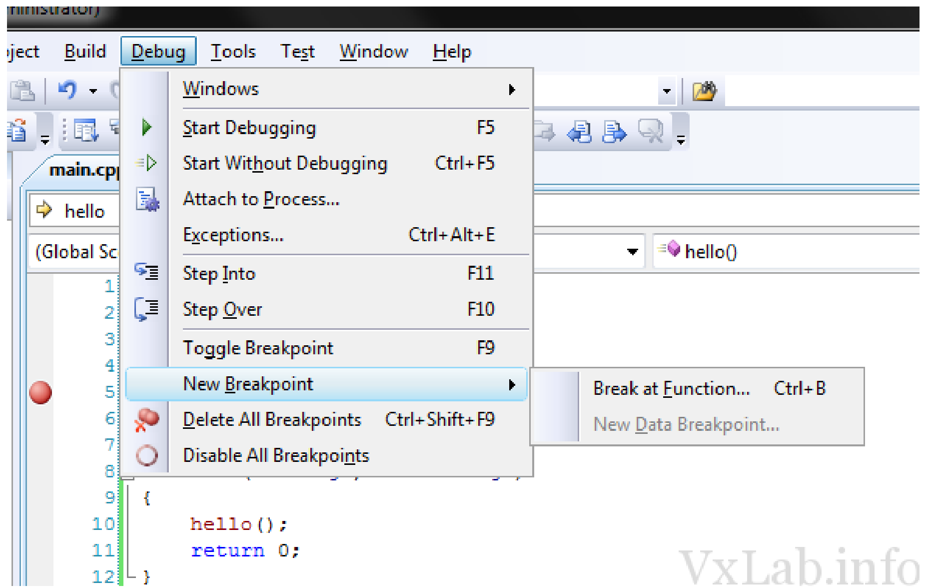
Любой алгоритм выполняется последовательно, одна инструкция за другой. Точка остановки позволяет приостановить выполнение кода на определенной инструкции. Это необходимо, чтобы начать отладку с предположительно проблемного участка. Например, при модульной структуре проекта проблемы начались при подключении нового модуля. Незачем отлаживать весь проект, когда можно отладить только модуль. Точки остановки сильно облегчают процесс в этом случае. Для того, чтобы установить точку остановки необходимо кликнуть на небольшую область, выделенную как правило особым цветом, в левой части редактора кода. Также это можно сделать, установив каретку на нужную строку и нажав F9.

При установке точки останова в окне редактора кода будет выведена иконка напротив строки кода.

лик левой кнопкой мыши по иконке приведет к удалению точки остановки, а для управления необходимо открыть контекстное меню (клик правой кнопкой).

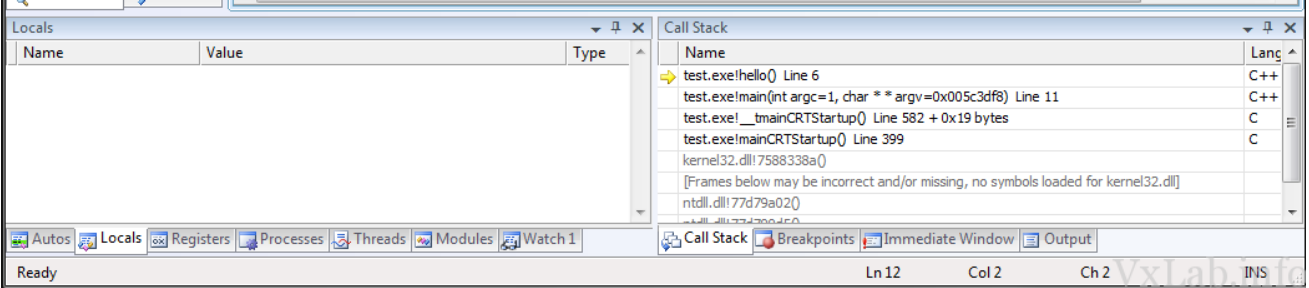


Среди доступных действий также можно отключить и удалить все точки остановки. Те же действия возможно выполнить из вкладки **Debug**, либо по нажатию соответствующих горячих клавиш

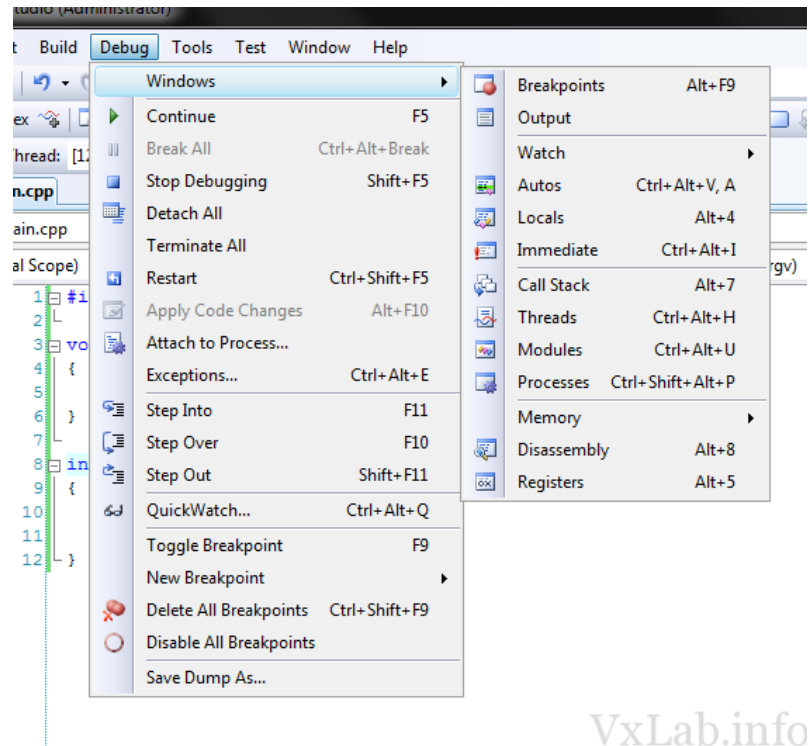


Работа с выводом отладчика

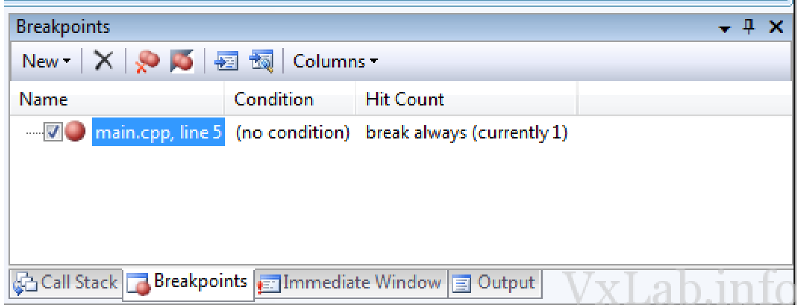
Запуск отладчика происходит при запуске проекта в среде разработки. При наличии установленных точек останова, выполнение прервется на первой из них автоматически. Важно понимать, что отладчик будет останавливать выполнение модуля напротив каждой активной точки остановки. Подробнее поговорим о том, какая информация может быть доступна во время отладки. Прежде всего, информация эта четко структурирована и выводится в отдельных окнах.



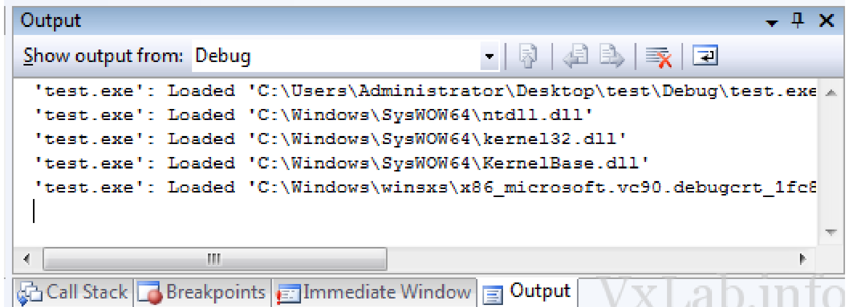
Посмотреть вывод всей возможной информации можно во вкладке Debug->Window.



Breakpoints – информация обо всех точках остановки в отлаживаемом проекте.



Output – окно вывода Visual Studio. Используется для вывода служебных сообщений при работе со средой.

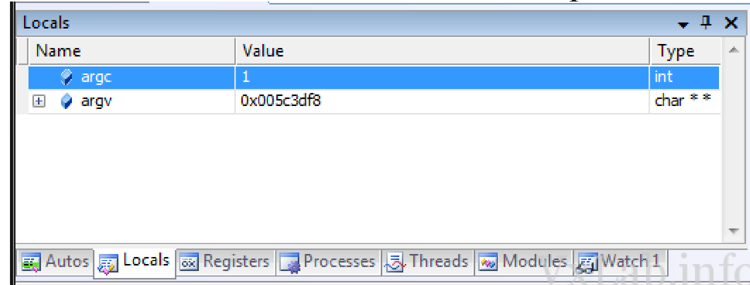


Watch – список наблюдаемых переменных. Переменные для наблюдения вносятся в список вручную и находятся там всегда, пока их не удалит разработчик.

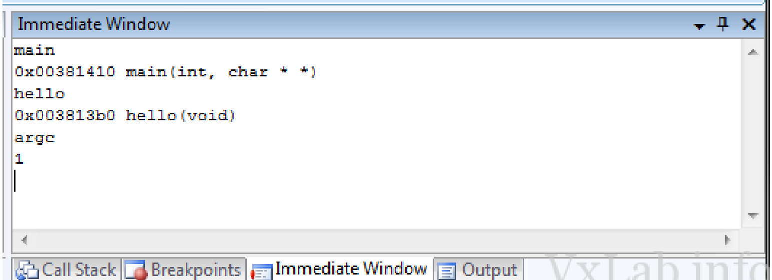
Name	Value	Type
main	0x00381410 main(int, char * *)	int (int, char * *)

Autos – переменные, с которыми идет работа в данный момент. Т.е. актуальные на момент исполнения кода переменные и их значения.

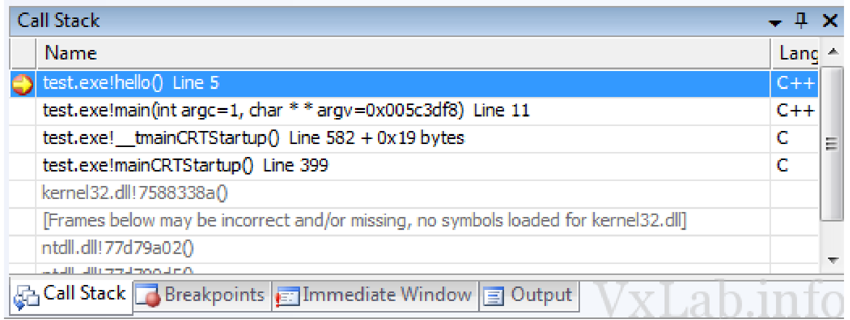
Locals – вывод локальных переменных и их значений.



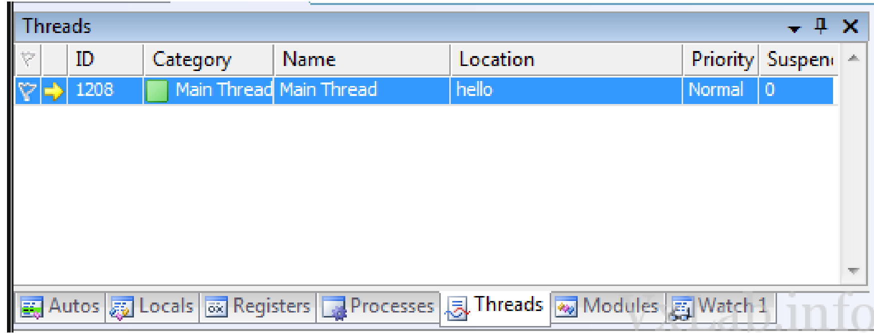
Immediate – поиск и вывод значения переменной по символьному представлению. В отличие от **Watch**, не хранит список значений, а выводит значение по требованию.



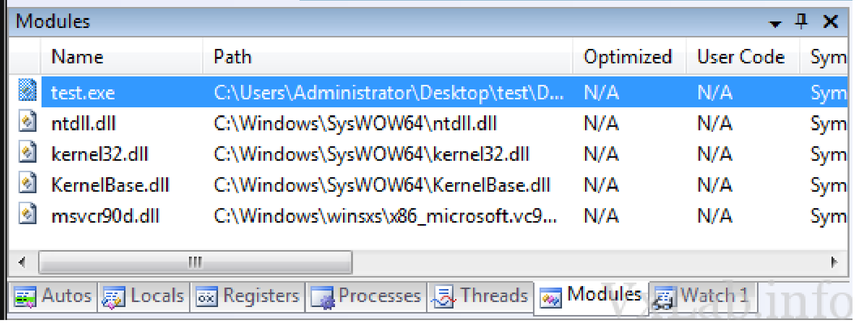
Call Stack – стек вызовов. Показывает последовательность вызовов функций для отлаживаемого модуля до точки остановки.



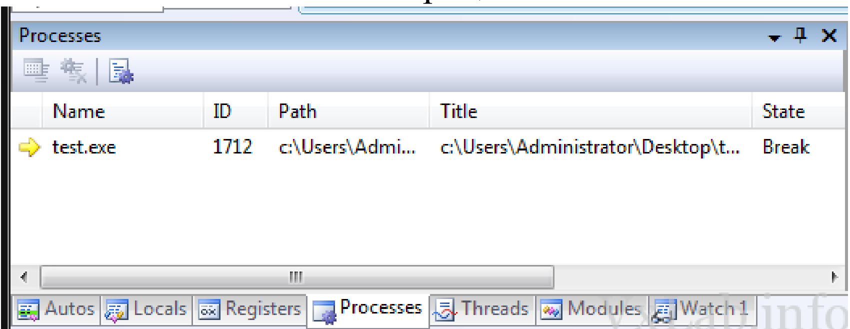
Threads – вывод информации обо всех запущенных потоках отлаживаемого модуля. Из этого окна возможна работа с потоками через контекстное меню: остановка потоков, переход к исходному коду потока и т.д.



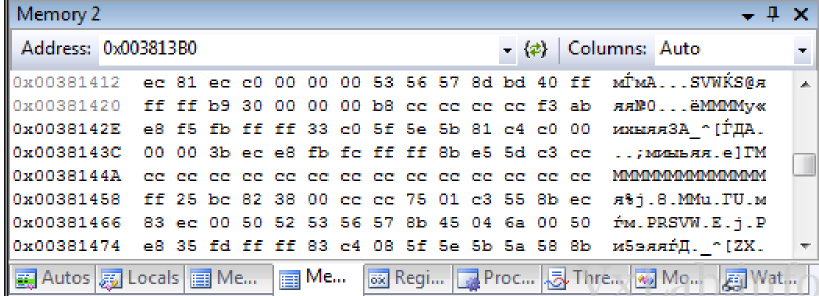
Modules – список загруженных модулей, необходимых для работы отлаживаемого (которые он подгружает для работы).



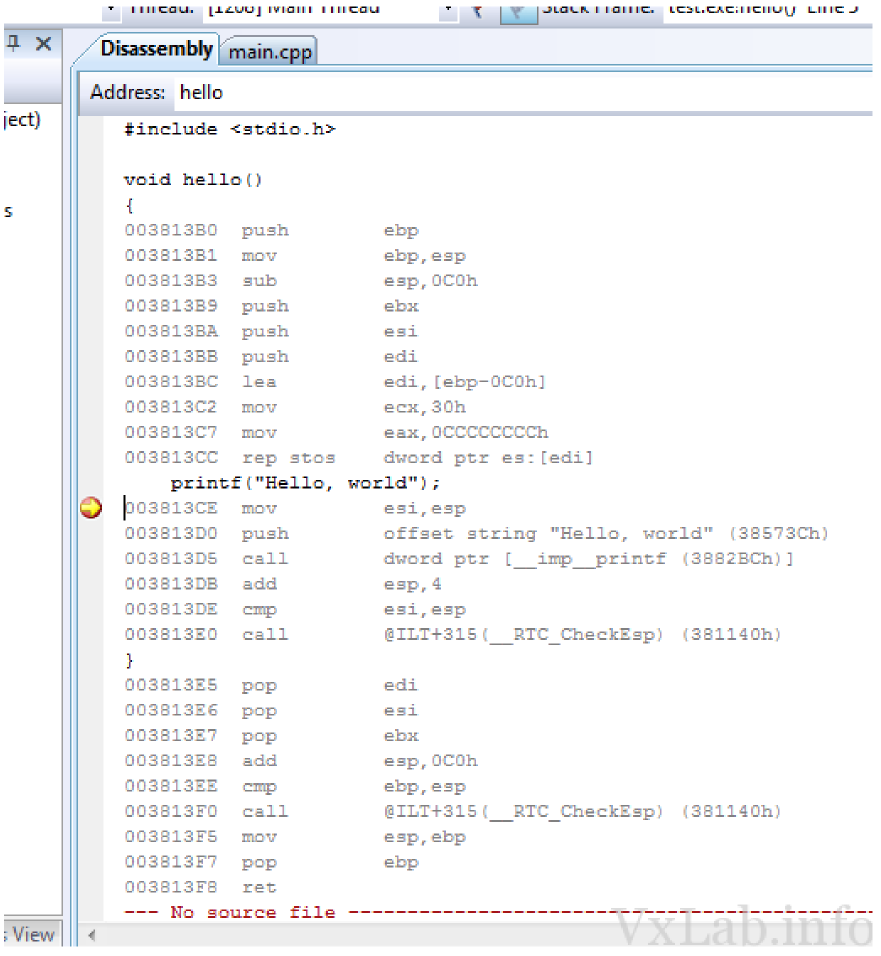
Processes – список процессов отлаживаемого модуля.



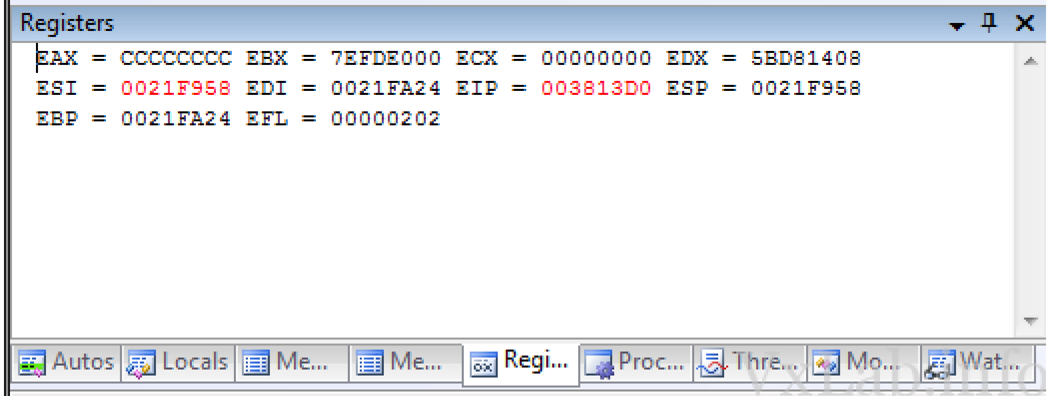
Memory – просмотр памяти отлаживаемого модуля в традиционном для любого hex-редактора виде.



Disassembly – просмотр ассемблерного листинга для отлаживаемого модуля. Очень хорошая возможность прямо из среды разработки посмотреть, как в итоге выглядит код на высокоуровневом языке.



Registers – вывод значений в регистрах



Что следует знать при работе с выводом отладчика

Информация в большинстве окон обновляется для каждой инструкции. Т.е. отладчик регистрирует и изменяет значения на новые. При этом измененные значения помечаются красным цветом.

Registers
EAX = CCCCCCCC EBX = 7EFDE000 ECX = 00000000 EDX = 5BD81408
ESI = 0021F958 EDI = 0021FA24 EIP = 003813D0 ESP = 0021F958
EBP = 0021FA24 EFL = 00000202
Autos Locals Me... Me... Regi... Proc... Thre... Mo... Wat...

Окно **Watch** поддерживает так называемые *псевдопеременные*. Своего рода макросы для определенных значений. Например, для вывода результата GetLastError после каждой строчки кода. Также Watch выполняет арифметические операции с переменными, а для *managed*-кода может выполнять код.

Watch 1

Name	Value	Type
main	0x00381410 main(int, char * *)	int (int, char * *)
$err	0	unsigned long
$err,hx	0x0000	unsigned long
argc	1	int
argc+5	6	int

Autos | Locals | Me... | Me... | Regi... | Proc... | Thre... | Mo... | Wat...

Пошаговая отладка

Для того чтобы начать процесс отладки, необходимо поставить точку остановки в нужном месте и запустить модуль. Отладка станет доступна по достижению точки остановки. Также можно принудительно остановить выполнение модуля, в таком случае курсор автоматически переместится на код, который выполнялся до остановки. Но точка остановки при этом не установится.

Во время отладки в панели инструментов **Visual Studio** доступен Debug Toolbar. Если его нет, включить его можно, отметив элемент меню Debug во вкладке View->Toolbars.

Hex

Все кнопки в панели инструментов повторяют вкладку Debug, нас в данный момент интересует навигация по отлаживаемому коду. В качестве примера рассмотрим следующую ситуацию:

(Global Scope)

```
 1 #include <stdio.h>
 2
 3 char* get_message()
 4 {
 5     return "Hello, world!";
 6 }
 7
 8 void hello(char* message)
 9 {
10     printf(message);
11 }
12
13 int main(int argc, char **argv)
14 {
15     hello(get_message());
16     return 0;
17 }
```

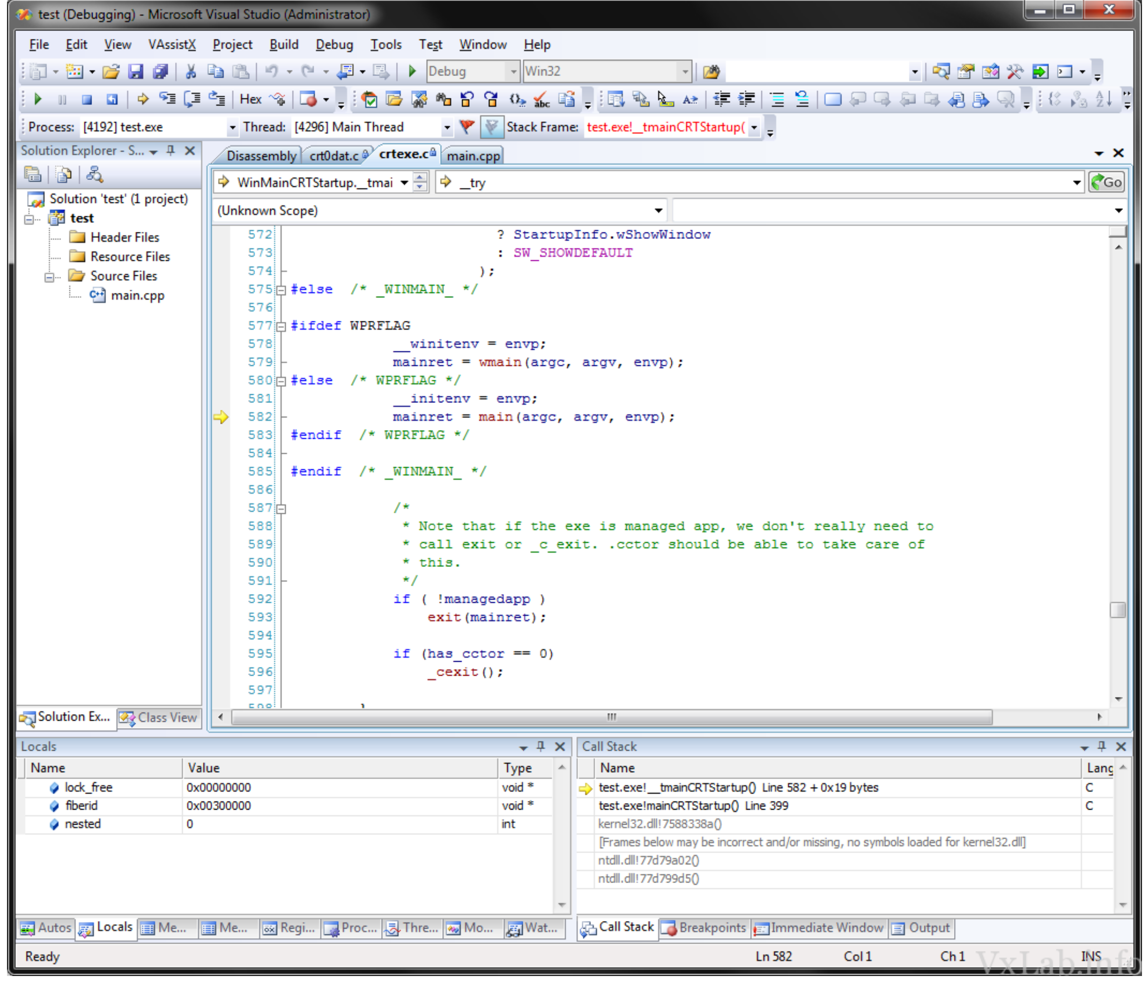
VxLab.info

шаг со входом. (f11) Если отладка остановилась на вызове функции, шаг со входом необходим для того, чтобы остановить выполнение модуля внутри тела функции. Т.е. если внутри функции нет точки остановки, можно

продолжить пошаговое выполнение, выполнив шаг со входом. Обратите внимание, в данном примере шаг со входом остановит выполнение программы внутри функции get_message, а не hello, т.к. сначала выполняется она. Т.е. шаг со входом в данном случае остановит выполнение модуля на строке 5.

шаг с обходом. (f10) Пошаговая отладка внутри функции. Не переходит внутрь функции при ее вызове, а выполняет как обычную инструкцию. При завершении функции останавливается на верхней в стеке вызовов. Вне зависимости от того, сколько функций вызывается на строчке, отладчик не прерывает внутри этих функций выполнение модуля. В примере следующий шаг будет на строке 16.

шаг с выходом. (Shift + f11) В этом случае отладчик остановит выполнение модуля после выхода из текущей функции, где происходит отладка. В данном примере будет выход из функции main и переключение на отладку кода CRT.



Во время пошаговой отладки важно также знать, что курсор можно передвинуть на необходимую строчку кода (Drag&Drop). Т.е. при желании заново отладить алгоритм не нужно перезапускать процесс отладки. Правда следует понимать, что в этом случае ожидаемое поведение программы не гарантировано. Разработчик должен отдавать отчет своим действиям.

Например, в случае ниже программист получит ошибку, если передвинет курсор на строку 9, т.к. память для с еще не была выделена.

```
1  #include <stdio.h>
2  #include <string.h>
3
4  int main(int argc, char **argv)
5  {
6      char* c = new char[10];
7      if (!c)
8          return 0;
9      strcpy_s(c, 10, "123");
10     printf(c);
11     delete c;
12     return 0;
13 }
```

VxLab.info

Основные команды меню Build

Compile — компиляция выбранного файла. Результаты компиляции выводятся в окно Output.

Build — компоновка проекта. Компилируются все файлы, в которых произошли изменения с момента последней компоновки. После компиляции происходит сборка (link) всех объектных модулей, включая библиотечные, в результирующий исполняемый файл. Сообщения об ошибках компоновки выводятся в окно Output. Если обе фазы компоновки завершились без ошибок, среда программирования создаст исполняемый файл с расширением *.exe (для данного примера: lr1.exe), который можно запустить на выполнение.

Rebuild All — то же, что и Build, но компилируются все файлы проекта независимо от того, были ли в них произведены изменения или нет.

Execute — выполнение исполняемого файла, созданного в результате компоновки проекта. Если же в исходный текст были внесены изменения – то перекомпилирование, перекомпоновку и выполнение.

Операции **Compile, Build** и **Execute** соответственно первая, вторая и четвертая кнопки панели инструментов **Build MiniBar,** которая расположена на рабочем столе (рисунок 3) справа вверху рядом с системными кнопками.

Управление конфигурациями проекта в Visual Studio 2010

Любой проект в VisualStudio2010 включает несколько самостоятельных конфигураций для компиляции разных версий программы. Конфигурацией называется набор параметров компилятора, компоновщика и библиотекаря, используемый при построении проекта. По умолчанию при создании проекта средаVisualStudio2010 автоматически включает в него две конфигурации: **Debug**(отладочную) и **Release**(финальную). Как следует из их названий, отладочная конфигурация используется в процессе написания программы, ее тестовых запусков для обнаружения и исправления ошибок, финальная – для построения конечной версии продукта, передаваемого заказчику для промышленного использования.

При создании проекта настройки отладочной **(Debug)** и финальной **(Release)** конфигураций устанавливаются в значения по умолчанию. С этими настройками выполняются следующие действия.

- **Debug** конфигурация компилируется с включением полной символьной отладочной информации и выключением оптимизации. Оптимизация кода затрудняет процесс отладки, так как усложняет или даже полностью изменяет отношение между строками исходного кода программы и сгенерированными машинными инструкциями. Такая отладочная информация используется отладчиком Visual Studio 2010 для отображения значений переменных, определения текущей выполняемой строки программы, отображения стека вызовов и так далее, т. е. для поддержки стандартных действий, выполняемых при отладке программы.
- **Release** конфигурация не содержит никакой отладочной информации и подвергается полной оптимизации. Без отладочной информации процесс отладки программы очень затруднен. Однако при необходимости такая информация может быть создана для финальной версии программы и записана в отдельный файл с расширением .pdb. Файлы отладочной информации .pdb могут оказаться очень полезными, если позднее возникнет необходимость в отладке финальной версии программы, например при обнаружении критических ошибок в процессе ее эксплуатации на компьютерах заказчика. Файлы .pdb обычно заказчику не передаются, а сохраняются у разработчиков.

Переключение между конфигурациями можно осуществлять из панели инструментов или при помощи окна ConfigurationManager(менеджер конфигураций). Для быстрого переключения конфигурации, используемой для компиляции проекта, используется стандартная панель инструментов (рисунок 36).

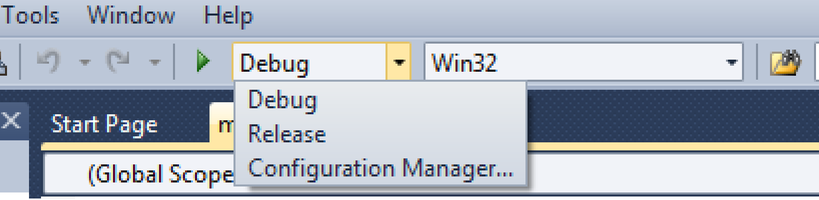


Рисунок 36 – Переключение конфигураций из панели инструментов

Для того чтобы проверить какая текущая конфигурация в проекте, нужно выбрать в подменю Project пункт Settings. Откроется диалоговое окно Project Settings. Смотрим, какое значение установлено в окне комбинированного списка Settings For:. Если это не Win32 Debug, то переключитесь на нужное значение через команду меню Build/Set Active Configuration

Лекция 6 Основные понятия. Типы данных

Состав и структура языка программирования. Понятие алфавита, синтаксиса и семантики. Основные конструкции языка: символы, зарезервированные и пользовательские имена, комментарии.

В тексте на любом естественном языке можно выделить четыре основных элемента: символы, слова, словосочетания и предложения. Подобные элементы содержит и алгоритмический язык, только слова называют лексемами (элементарными конструкциями), словосочетания — выражениями, а предложения — операторами. Лексемы образуются из символов, выражения — из лексем и символов, а операторы — из символов, выражений и лексем:

1. Алфавит языка, или его символы — это основные неделимые знаки, с помощью которых пишутся все тексты на языке.
2. Лексема, или элементарная конструкция, — минимальная единица языка, имеющая самостоятельный смысл.
3. Выражение задает правило вычисления некоторого значения.
4. Оператор задает законченное описание некоторого действия.

Для описания сложного действия требуется последовательность операторов.

Операторы могут быть объединены в составной оператор, или блок. В этом случае они рассматриваются как один оператор.

Операторы бывают исполняемые и неисполняемые. Исполняемые операторы задают действия над данными. Неисполняемые операторы служат для описания данных, поэтому их часто называют операторами описания или просто описаниями.

Каждый элемент языка определяется синтаксисом и семантикой. Синтаксические определения устанавливают правила построения элементов языка, а семантика определяет их смысл и правила использования.

Объединенная единым алгоритмом совокупность описаний и операторов образует программу на алгоритмическом языке. Для того чтобы выполнить программу, требуется перевести ее на язык, понятный процессору — в машинные коды. Этот процесс состоит из нескольких этапов. Рисунок иллюстрирует эти этапы для языка C++.

Сначала программа передается препроцессору, который выполняет директивы, содержащиеся в ее тексте (например, включение в текст так называемых заголовочных файлов — текстовых файлов, в которых содержатся описания используемых в программе элементов).

Получившийся полный текст программы поступает на вход компилятора, который выделяет лексемы, а затем на основе грамматики языка распознает выражения и операторы, построенные из этих лексем. При этом компилятор выявляет синтаксические ошибки и в случае их отсутствия строит объектный модуль.

Компоновщик, или редактор связей, формирует исполняемый модуль программы, подключая к объектному модулю другие объектные модули, в том числе содержащие функции библиотек, обращение к которым содержится в любой программе (например, для осуществления вывода на

экран). Если программа состоит из нескольких исходных файлов, они компилируются по отдельности и объединяются на этапе компоновки. Исполняемый модуль имеет расширение .exe и запускается на выполнение обычным образом.

Для описания языка в документации часто используется некоторый формальный метаязык, например, формулы Бэкуса—Наура или синтаксические диаграммы. Для наглядности и простоты изложения используется широко распространенный неформальный способ описания, при котором необязательные части синтаксических конструкций заключаются в квадратные скобки, текст, который необходимо заменить конкретным значением, пишется по-русски, а выбор одного из нескольких элементов обозначается вертикальной чертой.

Например, запись

```
[ void|int ] int имя;
```

означает, что вместо конструкции имя необходимо указать конкретное имя в соответствии с правилами языка, а перед ним может находиться либо void, либо int, либо ничего. Фигурные скобки используются для группировки элементов из которых требуется выбрать только один. В тех случаях, когда квадратные скобки являются элементом синтаксиса, это оговаривается особо.

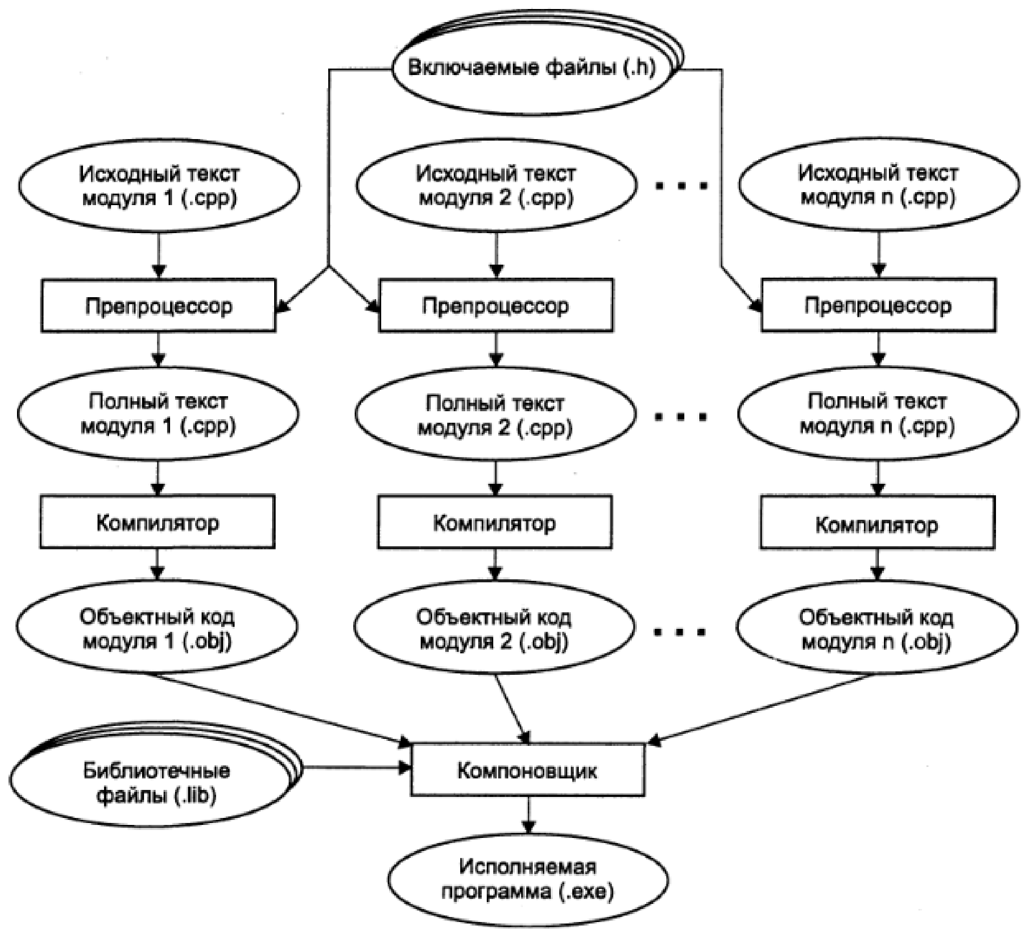


Рисунок 37 - Этапы создания исполняемой программы

Начнем изучение C++ с самого простого — с алфавита, а затем, осваивая все более сложные элементы, постепенно углубимся в дебри

объектно-ориентированного программирования и постараемся в них не заблудиться.

Алфавит языка

Алфавит С++ включает:

– прописные и строчные латинские буквы и знак подчеркивания;
– арабские цифры от 0 до 9;
– специальные знаки:

<table>
<tr><th>Символ</th><th>Наименование</th><th>Символ</th><th>Наименование</th></tr>
<tr><td>,</td><td>Запятая</td><td>)</td><td>Круглая скобка правая</td></tr>
<tr><td>.</td><td>Точка</td><td>(</td><td>Круглая скобка левая</td></tr>
<tr><td>;</td><td>Точка с запятой</td><td>}</td><td>Фигурная скобка правая</td></tr>
<tr><td>:</td><td>Двоеточие</td><td>{</td><td>Фигурная скобка левая</td></tr>
<tr><td>?</td><td>Вопросительный знак</td><td><</td><td>Меньше</td></tr>
<tr><td>'</td><td>Апостроф</td><td>></td><td>Больше</td></tr>
<tr><td>!</td><td>Восклицательный знак</td><td>[</td><td>Квадратная скобка левая</td></tr>
<tr><td>|</td><td>Вертикальная черта</td><td>]</td><td>Квадратная скобка правая</td></tr>
<tr><td>/</td><td>Дробная черта</td><td>#</td><td>Номер</td></tr>
<tr><td>\</td><td>Обратная черта (слеш)</td><td>%</td><td>Процент</td></tr>
<tr><td>~</td><td>Тильда</td><td>&</td><td>Амперсанд</td></tr>
<tr><td>*</td><td>Звездочка</td><td>^</td><td>Логическое «не»</td></tr>
<tr><td>+</td><td>Плюс</td><td>=</td><td>Равно</td></tr>
<tr><td>–</td><td>Минус</td><td>"</td><td>Кавычки</td></tr>
</table>

– пробельные символы: пробел, символы табуляции, символы перехода на новую строку.

Из символов алфавита формируются лексемы языка:

– идентификаторы;
– ключевые (зарезервированные) слова;
– знаки операций;
– константы;
– разделители (скобки, точка, запятая, пробельные символы).

Границы лексем определяются другими лексемами, такими, как разделители или знаки операций.

Идентификаторы

Идентификатор — это имя программного объекта. В идентификаторе могут использоваться латинские буквы, цифры и знак подчеркивания. Прописные и строчные буквы различаются, например, sysop, SySoP и SYSOP — три различных имени. Первым символом идентификатора может быть буква или знак подчеркивания, но не цифра. Пробелы внутри имен не допускаются.

Для улучшения читаемости программы следует давать объектам осмысленные имена. Существует соглашение о правилах создания имен, называемое венгерской нотацией (поскольку предложил ее сотрудник компании Microsoft венгр по национальности), покоторому каждое слово, составляющее идентификатор, начинается с прописной буквы, а вначале ставится префикс, соответствующий типу величины, например, iMaxLength, IpfnSetFirstDialog. Другая традиция — разделять слова, составляющие имя, знаками подчеркивания: maxjength, number_of_galosh.

Длина идентификатора по стандарту не ограничена, но некоторые компиляторы и компоновщики налагают на нее ограничения. Идентификатор создается на этапе объявления переменной, функции, типа и т. п., после этого его можно использовать в последующих операторах программы. При выборе идентификатора необходимо иметь в виду следующее:

– идентификатор не должен совпадать с ключевыми словами (см. Следующий раздел) и именами используемых стандартных объектов языка;
– не рекомендуется начинать идентификаторы с символа подчеркивания, поскольку они могут совпасть с именами системных функций или переменных, и, кроме того, это снижает мобильность программы;
– на идентификаторы, используемые для определения внешних переменных, налагаются ограничения компоновщика (использование различных компоновщ;иков или версий компоновщика накладывает разные требования на имена внешних переменных).

Ключевые слова

Ключевые слова – это зарезервированные идентификаторы, которые наделены определенным смыслом. Трансляторы языков C/C++, соответствующие требованиям стандарта ANSI, воспринимают только служебные слова, записанные строчными буквами. Не следует использовать имена объектов (идентификаторы), совпадающие со служебными словами. Далее приведены списки ключевых слов.

Ключевые слова для C/C++

auto	break	case	char
const	continue	default	do
double	else	enum	extern
float	for	goto	if
int	long	register	return
short	signed	sizeof	static
struct	switch	typedef	union
unsigned	void	volatile	while
inline (C99)		restrict (C99)	

Ключевые слова только для C++

asm	bool	catch	class
const_cast	delete	dynamic_cast	explicit
false	friend	virtual	mutable
namespace	new	operator	private
protected	public	reintepret_cast	throw
static_cast	template	this	typename
true	try	typeid	
using	wchar_t		

В дополнение к этому, идентификаторы, содержащие двойное подчеркивание (__) резервируются для реализаций C++ и стандартных библиотек. Пользователи не должны употреблять их.

В представлении программы на C++ в кодировке ASCII используются в качестве операций или разделителей следующие символы:

```
!     *     =     [ ]   :     ?
%     ( )   { }   \     "     ,
^     -     |     ;     <     .
&     +     ~     '     >     /
```

а следующие комбинации символов используются для задания операций:

```
->    ->*   >=    ||    +=    &=
++    <<    ==    *=    -=    ^=
--    >>    !=    /=    <<=   |=
.*    <=    &&    %=    >>=   ::
```

Каждая операция считается отдельной лексемой. В дополнение к этому, символы # и ## резервируются для препроцессора.

Константы

Константы, в отличие от переменных, являются фиксированными значениями, которые можно вводить и использовать на языках C/C++.

Разделяют четыре типа констант: целые константы, константы с плавающей запятой, символьные константы и строковые литералы.

Целые константы не имеют дробной части и не содержат десятичной точки. Они представляют целую величину в одной из следующих форм: десятичной, восьмеричной или шестнадцатеричной. Десятичная константа состоит из одной или нескольких десятичных цифр, причем первая цифра не должна быть нулем (в противном случае число будет воспринято как восьмеричное). Восьмеричная константа состоит из обязательного нуля и одной или нескольких восьмеричных цифр. Шестнадцатеричная константа начинается с обязательной последовательности 0x или 0X и содержит одну или несколько шестнадцатеричных цифр. Примеры целых констант:

Десятичная константа	Восьмеричная константа	Шестнадцатеричная константа
16	020	0x10
127	0117	0x2B
240	0360	0xF0

Целые константы могут быть обычной длины или длинные.

Длинные целые константы оканчиваются буквой l или L (например: 5l, 6l, 128L, 0105L, OX2A11L). Размер целых констант обычной длины зависит от реализации (для тридцатидвухразрядного процессора – 4 байта). Длинная целая константа всегда занимает 4 байта.

При использовании десятичной целой константы старший бит числа интерпретируется как знаковый (0 – положительное число, 1 – отрицательное). Для восьмеричных и шестнадцатеричных целых констант возможно представление только положительных чисел и нуля, поскольку старший разряд рассматривается как часть кода числа, а не как его знак.

Диапазон значений десятичных констант обычной длины для тридцатидвухразрядного процессора – (-2^{31} …+ (2^{31} – 1)). Диапазон значений

восьмеричных и шестнадцатеричных констант обычной длины для тридцатидвухразрядного процессора – $(0...2^{32} – 1)$.

Диапазон значений длинных десятичных констант не зависит от разрядности процессора и составляет $(–2^{31}…+ (2^{31} – 1))$. Диапазон значений длинных восьмеричных и шестнадцатеричных констант также не зависит от разрядности процессора и составляет $(0…2^{32} – 1)$.

Константа с плавающей точкой – десятичное число, представленное в виде действительной величины с десятичной точкой или экспонентой. Формат имеет вид:

[цифры].[цифры] [E|e [+|-] цифры]

Число с плавающей точкой состоит из целой и дробной части и (или) экспоненты. Константы с плавающей точкой представляют положительные величины удвоенной точности (имеют тип double).

Для определения отрицательной величины необходимо сформировать константное выражение, состоящее из знака минуса и положительной константы: 115.75, 1.5E-2, -0.025, .075, -0.85E2.

Или целая часть, или дробная часть (но не обе) могут отсутствовать. Или точка, или символ e (или E) вместе с показателем могут отсутствовать (но не оба). Если число записывается без них, то получаем целое.

В языке C++, когда в конце константы с плавающей точкой отсутствуют буквы f, F, l, L, константа имеет тип double (8 байт или 64 бита с диапазоном значений $(\pm1,7\cdot10^{-308} \ldots \pm1,7\cdot10^{+308})$. Если же константа заканчивается буквой f или F, то она имеет тип float, занимает 4 байта и диапазон значений $(\pm3,4\cdot10^{-38}\ldots\pm3,4\cdot10^{+38})$. Аналогичным образом, при завершении константы буквами l или L, константа имеет тип long double, занимает 10 байт с диапазоном значений $(\pm3,4\cdot10^{-4932} \ldots\pm3,4\cdot10^{+4932})$.

Символьные константы можно разделить на две группы: печатные и непечатные символы. Символьная константа в языках C/C++ состоит либо из одного печатного символа, заключенного в апострофы (' ', 'Q') , либо специального и управляющего кода, заключенного в апострофы ('\n', '\\'). Управляющие коды представляют непечатные символы. Символьная константа рассматривается как символьный беззнаковый тип данных с диапазоном значений от 0 до 255. Часто используется константа '\0', которая называется нулевым символом или нулевым байтом.

Символьная константа, которой непосредственно предшествует буква L, является широкой символьной константой, например, L'ab'. Такие константы имеют тип wchar_t, являющийся целочисленным типом. Широкие символы предназначены для такого набора символов, где значение символа не помещается в один байт.

Строковая константа (литерал) – последовательность символов (включая строковые и прописные буквы русского и латинского алфавитов, а также цифры), заключенная в кавычки (""): "город Тамбов", "YZPT КОД\72". Для запоминания строковых констант используется по одному байту на каждый символ строки и автоматически добавляется к ней признак конца

строки, которым служит нулевой байт '\0'. Нулевой байт является ограничителем строки.

Для составления строковых констант можно использовать любые печатные символы или управляющие коды.

Строка литералов, перед которой непосредственно идет символ L, считается широкосимвольной строкой, например, L"asdf". Такая строка имеет тип «массив элементов типа wchar_t», где wchar_t – целочисленный тип.

Ключевое слово const – модификатор типа, указывающий, что переменная является константой. Описание констант начинается с ключевого слова const, далее указывается тип и значение:

```
const int Size=2;
```

Поскольку константе ничего нельзя присвоить, она должна быть инициализирована. Описание чего-нибудь как const гарантирует, что его значение не изменится в области видимости:

```
Size=2; //ошибка
```

Раз константе нельзя присвоить значение, она должна быть инициализирована в месте своего определения. Определение константы без ее инициализации также вызывает ошибку компиляции:

```
const double pi; //ошибка: неинициализированная константа
```

Кроме констант в программе могут использоваться константные выражения.

Комментарии

Комментарий – это набор символов, которые игнорируются компилятором. Введение комментария начинается с символов /* и заканчивается символами */. Все, что помещено между ними, игнорируется:

```
const double pi; //ошибка: неинициализированная константа
const int Size=2;
Size=2; //ошибка
/*Эта программа выводит сообщение на экран*/
```

Комментарии /* */ не могут быть вложенными. В C++ используется пара символов //, указывающая начало строки комментария.

В этом случае концом комментария считается конец строки, так что нет необходимости отмечать его специальным символом:

```
//Эта программа выводит сообщение на экран
```

Этот способ наиболее полезен для коротких комментариев.

Структура программы

C/C++-программа – совокупность одного или нескольких модулей. Модулем является самостоятельно транслируемый файл, который обычно содержит одну или несколько функций. Функция состоит из операторов языка.

Термин «функция» в языках C/C++ охватывает понятия «подпрограмма», «процедура» и «функция», используемые в других языках программирования. C/C++-программа может содержать одну функцию (главная функция main) или любое количество функций. Фактически все

программы на ISO/ANSI C++ начинаются с главной функции main. Microsoft также называет эту функцию wmain, когда применяется кодировка символов Unicode и имя _tmain определено либо как main, либо как wmain. Другие функции могут быть вызваны из функции main или из какой-либо другой функции в процессе выполнения программы. Эти функции могут находиться в том же модуле (файле), что и функция main, или в других модулях.

Главная функция выглядит следующим образом:

```
void _tmain(){ /* ... */ }
```

В круглых скобках main перечисляются аргументы или параметры функции (в данном случае аргументов нет). У функции может быть результат или возвращаемое значение. Если функция не возвращает никакого значения, то это обозначается ключевым словом void.

В фигурных скобках ({...}) записывается тело функции – действия, которые она выполняет. Пустые фигурные скобки означают, что никаких действий не производится.

При определении функции с аргументами:

```
int _tmain(int argc, char* argv[]) { /* ... */ }
```

argc задает число параметров, передаваемых программе окружением. Если argc не равно нулю, параметры должны передаваться как строки, завершающиеся символом '\0', с помощью argv[0] до argv[argc-1], причем argv[0] должно быть именем, под которым программа была запущена, и должно гарантироваться, что argv[argc]==0.

В общем случае программа на С/С++ имеет следующую структуру:

```
#директивы препроцессора
. . . . . . . . .
#директивы препроцессора
описание прототипов функций
определение глобальных переменных
функция а ( )
   {операторы}
void main ( ) //функция, с которой начинается выполнение
                программы
{  описания
   присваивания
         операторы
         вызов функции а
         вызов функции b
         оператор пустой
               составной
               выбора
               циклов
               перехода
}

функция b ( )
   {операторы}
```

До компилятора исходный текст обрабатывается препроцессором – специальной программой, которая модифицирует текст программы по специальным командам – директивам. Программа начинается с директив препроцессора, с символа #.

Далее размещается блок определения данных, за которыми следуют операторы функции. Определения данных создают переменные, которые будут использованы в функции. Операторы задают действия, которые должны быть выполнены над переменными.

Все элементы данных должны быть определены перед их использованием. Определения данных и операторы всегда завершаются точкой с запятой. Этот символ помечает конец оператора, а не конец строки. Следовательно, один оператор может распространяться на несколько строк, если это помогает понять код, либо несколько операторов могут находиться в одной строке. Оператор программы – базовый элемент, определяющий то, что делает программа.

Фундаментальные типы данных и переменные

Программа оперирует информацией, представленной в виде различных объектов и величин. С точки зрения архитектуры компьютера, переменная – это символическое обозначение ячейки оперативной памяти программы, в которой хранятся данные. Доступ к значению возможен через имя переменной, а доступ к участку памяти – по его адресу. Каждая переменная перед использованием в программе должна быть объявлена, т.е. ей должна быть выделена память. Размер участка памяти, выделяемой для переменной, и интерпретация содержимого зависят от типа, указанного в определении переменной.

Тип переменной изменить нельзя. Простейшая форма объявления переменных:

```
тип список_имен_переменных;
```

Тип переменной указывает компилятору языка С++, сколько памяти надо выделить для размещения объекта. Кроме того, он указывает компилятору, каким образом надо интерпретировать значение, содержащееся в объекте.

В С++ выделяют следующие категории типов: базовые (стандартные или основные) и производные (определяемые) типы. Базовые типы имеют имена, которые являются ключевыми словами языка. Производные типы определяются на основе базовых и делятся на скалярные (перечисления, указатели и ссылки) и структурные (массивы, структуры, объединения и классы). Дерево классификации приведено на рисунке 38.

Базовые типы данных

Тип данных	Назначение	Размер, байт	Диапазон значений
`char`	Для символа	1	–128…+127
`wchar_t`	Для символа из расширенного набора	2	–32 768…+32 767
`int`	Для целого значения	4	–2 147 483 648… 2 147 483 647
`bool`	Для логического значения	1	`false, true`
`float`	Для значения с плавающей точкой	4	$\pm3{,}4 \cdot 10^{-38} \ldots \pm3{,}4 \cdot 10^{+38}$ (7 цифр)
`double`	Для значения с плавающей точкой удвоенной точности	8	$\pm1{,}7 \cdot 10^{-308} \ldots \pm1{,}7 \cdot 10^{+308}$ (15 цифр)

Рисунок 38 – Базовые типы данных

Происхождение и перевод служебных слов:

- char (CHARacter: буква, символ);
- wchar_t (Wide CHaracter Type: расширенный символьный тип);
- int (INTeger: целое число);
- float (число с плавающей точкой).

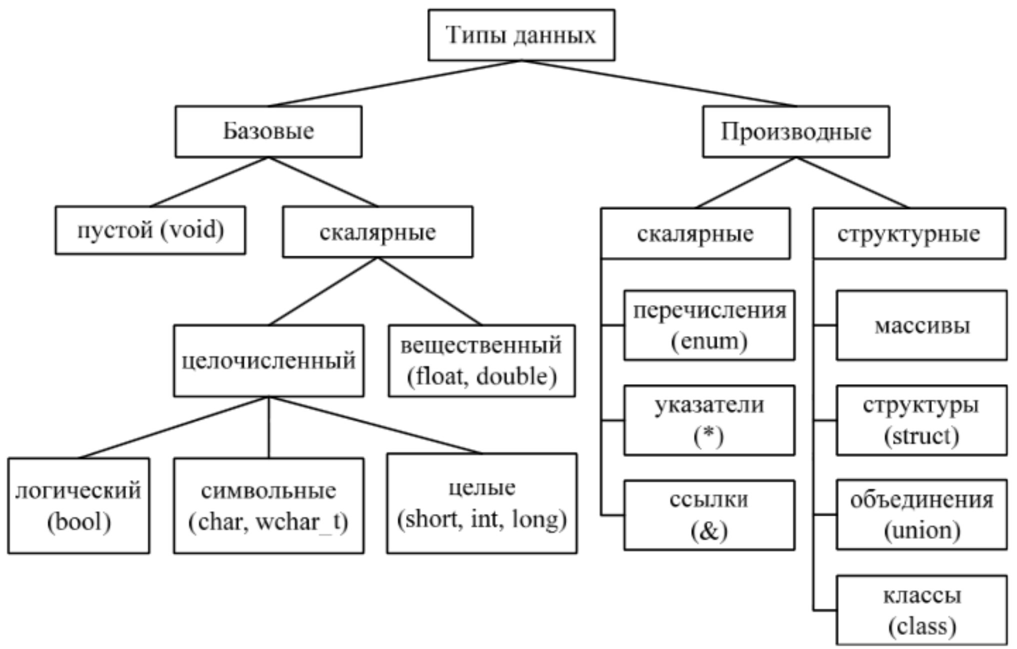


Рисунок 39 - Дерево классификации типов данных

Существуют четыре спецификатора типа (таблица 2), уточняющие внутреннее представление и диапазон значений стандартных типов: `unsigned` (без знака), `signed` (со знаком), `short` (короткий), `long` (длинный).

В стандартном заголовочном файле `<limits.h>` задаются в зависимости от реализации минимальные и максимальные значения каждого типа.

Переменным можно присваивать начальные значения, явно указывая их в определениях, этот прием называется инициализацией:

```
тип имя_переменной = начальное_значение;
```

Символьный тип (char)

Под величину символьного типа отводится один байт, что позволяет хранить в нем любой символ из 256-символьного набора ASCII (American Standart Code for Information Interchange):

```
char A;
char B;
A = 'D';
B = '!';
```

Таблица 2 - Уточняющие спецификаторы типа

Тип данных	Назначение	Размер, байт	Диапазон значений
`unsigned char`	Для байта с неотрицательным целым значением	1	0…255
`unsigned,` `unsigned int`	Для неотрицатель-ного целого значения	4	0…4 294 967 295
`short,` `short int,` `signed short int`	Для короткого це-лого значения	2	–32 768…+32 767
`unsigned short,` `unsigned short int`	Для беззнакового короткого целого	2	0…65 535
`long,` `long int,` `signed long int`	Для длинного це-лого	4	–2 147 483 648… 2 147 483 647
`unsigned long,` `unsigned long int`	Для беззнакового целого длинного	4	0…4 294 967 295
`long long,` `signed long long`	Для удвоенного длинного целого	8	–9 223 372 036 854 775 808… 9 223 372 036 854 775 807
`unsigned long long`	Для беззнакового удвоенного длин-ного целого	8	0… 18 446 744 073 709 551 615
`double`	Для вещественно-	8	$\pm 1{,}7 \cdot 10^{-308}$

long float	го с двойной точностью		…±1,7 · 10^{+308}
long double	Для длинного вещественного	10	±3,4 · 10^{-4932} …±3,4 · 10^{+4932}

Расширенный символьный тип (`wchar_t`)

Этот тип предназначен для работы с набором символов, для кодировки которого недостаточно одного байта (например, для набора Unicode). Символьные и строковые константы с типом `wchar_t` записываются с префиксом L.

Пример,

```
wchar_t letter = L'D';
```

Целые типы

Внутреннее представление величины целого типа – целое число в двоичном коде. При использовании спецификатора `signed` старший бит числа интерпретируется как знаковый (0 – положительное число, 1 – отрицательное). Спецификатор `unsigned` позволяет представлять только положительные числа, поскольку старший разряд рассматривается как часть кода числа. По умолчанию все целочисленные типы считаются знаковыми, то есть спецификатор `signed` можно опускать.

```
unsigned int Stavka;
int Symma;
```

Логический тип (bool)

Величины логического типа могут принимать только значения `false` и `true`, которые являются служебными словами. Внутренняя форма представления значения `false` – 0 (нуль). Любое другое значение интерпретируется как `true`. При преобразовании к целому типу `true` имеет значение 1.

Типы с плавающей точкой (`float`, `double`)

Внутреннее представление для типов с плавающей точкой состоит из двух частей – мантиссы и порядка. При этом величины типа `float` занимают четыре байта, из которых один двоичный разряд отводится под знак мантиссы, 8 разрядов под порядок и 23 под мантиссу. Мантисса – число большее 1.0, но меньшее 2.0. Поскольку старшая цифра мантиссы всегда равна 1, то она не хранится.

Для величин типа `double`, занимающих восемь байт, под порядок и мантиссу отводятся соответственно 11 и 52 разряда. Длина мантиссы определяет точность числа, а длина порядка – диапазон числа.

```
float pi = 3.14 , cc = 1.3456;
unsigned int year = 1999;
```

Тип (`void`)

Кроме перечисленных, к основным типам языка относится тип `void`, но множество значений этого типа пусто. Он используется для определения функций, которые не возвращают значения, для указания пустого списка

аргументов функции, как базовый тип указателей и в операции приведения типов.

Чтобы проверить размер памяти, выделяемой для объекта данного типа, можно написать программу, использующую операцию `sizeof`.

Значением этой операции является размер любого объекта или спецификации типа, выраженный в байтах

Пространство имен `std`

Пространство имен (**`namespace`**) – это механизм, предназначенный для предотвращения проблем, связанных с дублированием имен.

Определенное множество имен, вроде имен стандартной библиотеки, ассоциируется с общим именем, которое и представляет собой пространство имен. Все средства стандартной библиотеки С++ определены внутри пространства имен по имени `std`, поэтому каждый элемент стандартной библиотеки имеет свое собственное имя плюс наименование пространства имен std в качестве квалификатора. Имена `cout` и `endl` определены в стандартной библиотеке, поэтому их полные имена выглядят как `std::cout` и `std::endl`. Два двоеточия, отделяющие имя пространства имен от имени элемента, образуют операцию, называемую операцией разрешения контекста. Применение полных имен в программе делает код громоздким, поэтому можно использовать объявление `using`, сообщающее компилятору о намерении использовать имена из пространства имен `std` без указания наименования пространства имен:

```
using std::cout;using std::endl;.
```

Кроме того, программа может содержать следующую директиву `using`:

```
using namespace std;
```

Теперь все имена из пространства `std` импортируются в исходный файл, и можно ссылаться на все, что определено в этом пространстве имен, без квалификационного имени. То есть можно писать `cout` вместо `std::cout`.

Недостатком такого применения может быть возникновение непреднамеренных конфликтов.

Имя `cout` представляет стандартный выходной поток, который по умолчанию соответствует командной строке, а имя `endl` – символьной строке. Строка `cout<<"  ";` – оператор вывода, с помощью которого выводится на экран дисплея фраза, заключенная в кавычки.

Функция может возвращать значение в программу с помощью оператора возврата (`return`). Этот оператор также прекращает выполнение функции `main()` и завершает программу. Управление возвращается операционной системе.

Определение синонимов для типов данных

Ключевое слово `typedef` позволяет определить свое собственное имя для существующего типа данных. Альтернативное имя может использоваться наряду со встроенным именем типа:

```
typedef long int BigInt; //определение синонима
BigInt my_number = 0L; //определение перменной
```

Это средство играет важную роль в упрощении сложных объявлений, что делает код более читабельным.

Время существования и область видимости переменных

В дополнение к имени и типу объекта существуют еще два атрибута: область действия и время жизни объекта. Эти характеристики взаимосвязаны и существенно влияют на возможности использования переменной в программе. Взаимосвязь определяется способом выделения памяти для переменной.

Областью действия объекта (данного) называется та часть программы, в которой можно пользоваться этим объектом. В частности, областью действия может быть:

– блок операторов ({…});

– модуль (файл);

– вся программа в целом.

Временем жизни данного называется отрезок времени, в течение которого значение этого данного доступно в некоторой части программы.

Бывает локальное (при выполнении блока, в котором оно объявлена) и глобальное (при выполнении всей программы) время жизни.

В языках С/С++ область действия и время жизни объекта определяются местом объявления переменной и классом хранения (модификатором). Можно использовать следующие классы: внешний; внешний статический; внутренний статический; автоматический; регистровый.

Автоматическая (auto) переменная или константа имеет локальную область действия и известна только внутри блока, в котором она определена. Для переменной выделяется временная память при входе в блок, а при выходе уничтожается. По умолчанию переменная в блоке считается автоматической.

Регистровая (register) переменная храниться в регистре процессора, и, соответственно, доступ к ней быстрее, чем к автоматической переменной. При отсутствии свободных регистров регистровая переменная становится автоматической.

Внешняя (extern) переменная является глобальной переменной. Спецификатор extern информирует компилятор, что переменная будет объявлена (без extern) в другом файле, где ей будет выделена память.

Статической (static) переменной (константе) выделяется память после ее объявления и сохраняется до конца выполнения программы. Статические переменные при объявлении по умолчанию инициализируются нулевыми или пустыми значениями.

На рисунке 40 приведены характеристики переменных, объявленных с использованием модификаторов выделения памяти.

Характеристика модификаторов выделения памяти

Модификатор выделения памяти	Место объявления переменной	Область видимости	Время существования	Обобщенная характеристика
Не указан (по умолчанию `static`)	Вне функции	До конца файла	Глобальное	Глобальная переменная файла
Не указан (по умолчанию `auto`)	В функции	До конца блока	Локальное	Локальная переменная
`auto`	Запрещено вне функции	–	–	–
`auto`	В функции	До конца блока	Локальное	Локальная переменная
`register`	Запрещено вне функции	–	–	–
`register`	В функции	До конца блока	Локальное	Быстрая локальная переменная
`static`	Вне функции	До конца файла	Глобальное	Глобальная переменная файла
`static`	В функции	До конца блока	Глобальное	Сохраняемая локальная переменная
`extern`	Вне функции	До конца файла	Глобальное	Глобальная переменная программы

Рисунок 40 - Характеристика модификаторов выделения памяти

Базовые операции ввода-вывода

Потоковый ввод-вывод

Обмен данными между программой и внешними устройствами осуществляется с помощью операций ввода-вывода. Классы потокового ввода-вывода C++ определены в файле заголовков `#include <iostream>` Библиотека потоков ввода-вывода определяет: `cout` – стандартный поток вывода (экран дисплея), `cin` – стандартный поток ввода (связан с клавиатурой), `cerr, clog` – стандартный поток сообщений об ошибках.

Вывод осуществляется с помощью операции >>, ввод – с помощью операции <<. Выражение:

```
using std::cout;
............................
cout << "Пример вывода: " << 34;
```

напечатает строку "Пример вывода: ", за которой будет выведено число 34. Используя один стандартный поток вывода `cout`, можно отобразить несколько аргументов. Между собой аргументы разделяются операторами вставки:

```
using std::cout;
..............................
int age;
age = 23;
cout << "Вам исполнилось " << age << " года.";
```

Выражение:

```
using std::cin;
..............................
int x;
cin >> x;
```

введет целое число с консоли в переменную x (для того, чтобы ввод произошел, нужно напечатать число и нажать клавишу Enter.)

Часто бывает необходимо вывести строку или число в определенном формате. Для этого используются манипуляторы – объекты особых типов, которые управляют тем, как обрабатываются следующие аргументы:

`endl` – при выводе перейти на новую строку;

`ends` – вывести нулевой байт (признак конца строки символов);

`dec` – выводить числа в десятичной системе (по умолчанию);

`oct` – выводить числа в восьмеричной системе;

`hex` – выводить числа в шестнадцатеричной системе счисления;

`setw(int n)` – установить ширину поля вывода в nсимволов (n целое);

`setfill(int n)` – установить символ-заполнитель; этим символом выводимое значение будет дополняться до необходимой ширины;

`setprecision(int n)` – установить количество цифр после запятой при выводе вещественных чисел;

`setbase(int n)` - установить систему счисления для вывода чисел; n может принимать значения 0, 2,8, 10, 16, причем 0 означает систему счисления по умолчанию, т.е. 10.

Для использования манипуляторов их надо вывести в выходной поток. Кроме того, в символьную строку, заключенную в двойные кавычки, можно включать управляющие последовательности (escape sequences):

```
using namespace std;
..............................
int x = 53;
cout << "Десятичный вид: \t " << dec << x << endl
<< "Восьмеричный вид: \t " << oct << x << endl
<< "Шестнадцатеричный вид: \t " << hex << x << endl;
```

Ниже приведен пример использования манипуляторов с параметрами для вывода:

```
using namespace std;
..............................
double x; //вывести число в поле общей шириной 6 символов
//(3 цифры до запятой, десятичная точка
//и 2 цифры после запятой)
cout << setw(6) << setprecision(2) << x << endl;
```

Те же манипуляторы (за исключением endl и ends) могут использоваться и при вводе:

```
#include <iomanip>
..............................
using namespace std;
..............................
int x;
cin >> hex >> x; //ввести шестнадцатеричное число
```

Манипуляторы определены в заголовочном файле `<iomanip>`, поэтому при их использовании надо добавлять директиву `#include<iomanip>`.

Преобразование типов

В операциях могут участвовать операнды различных типов, в этом случае они неявно преобразуются к общему типу в порядке увеличения их объема памяти, необходимого для хранения их значений. Поэтому неявные преобразования всегда идут от «меньших» объектов к «большим». Схема выполнения преобразований операндов арифметических операций:

short, char → int → unsigned → long → double
float → double

Например,

```
int ival;
float fval;
double dval;
ival + fval + dval;
//fval и ival преобразуются к double перед
сложением
```

Значения типов char и short всегда преобразуются в int; если любой из операндов имеет тип double, то второй преобразуется в double; если один из операндов long, то другой преобразуется в long.

При присваивании значение правой части преобразуется к типу левой, который и является типом результата. При некорректном использовании операций присваивания могут возникнуть ошибки.

```
float x; int i;
x = i;
i = x;
//float преобразуется в int, дробная часть
отбрасывается
```

В любом выражении преобразование типов может быть осуществлено явно, в С для этого достаточно перед выражением поставить в скобках идентификатор соответствующего типа:

```
(тип) выражение;
```

В результате значение выражения преобразуется к заданному типу:

```
float a;
int i = 6, j = 4;
a = (i + j) / 3; // a = 3
a = (float)(i + j) / 3; → // a = 3.333333
```

Эта форма является устаревшей и сохранена в стандарте С++ только для обеспечения обратной совместимости с программами, написанными для С и предыдущих версий С++.

В С++ явное преобразование типов производится при помощи следующих операторов: static_cast, dynamic_cast, const_cast и reinterpret_cast. Хотя иногда явное преобразование необходимо, оно служит потенциальным источником ошибок.

Явное преобразование типов используется для разыменования указателя void*, для того, чтобы избежать стандартного преобразования или выполнить вместо него собственное. Также может использоваться, чтобы избежать неоднозначных ситуаций, в которых возможно несколько вариантов применения правил преобразования по умолчанию.

Синтаксис операции явного преобразования:

```
cast-name< type >( expression) ;
```

Здесь `cast-name` – одно из ключевых слов `static_cast`, `const_cast`, `dynamic_cast` или `reinterpret_cast`, а `type` – тип, к которому приводится выражение `expression` . Так `const_cast` служит для трансформации константного типа в неконстантный. Любое иное использование `const_cast` вызывает ошибку компиляции, как и попытка подобного приведения с помощью любого из трех других операторов. С применением `static_cast` осуществляются те преобразования, которые могут быть сделаны неявно, на основе правил по умолчанию:

```
const double d = 97.0;
char ch = static_cast< char >( d );
```

Оператор `reinterpret_cast` работает с внутренними представлениями объектов (`reinterpret` – другая интерпретация того же внутреннего представления), причем правильность этой операции целиком зависит от программиста.

Оператор `dynamic_cast` применяется при идентификации типа во время выполнения (`runtime type identification`).

Лекция 7 Операции, операторы и выражения

Операции и выражения С/С++

Классификация операций

Операции бывают унарные (воздействуют на одно значение или выражение), бинарные (участвуют два выражения) и тернарные (три выражения). Основные унарные операции приведены на рисунке 41

Унарные операции выполняются справа налево.

Унарные операции

Операции	Назначение
&	Получение адреса операнда
*	Обращение по адресу (разыменование)
–	Унарный минус, меняет знак арифметического операнда
~	Поразрядное инвертирование внутреннего двоичного кода (побитовое отрицание)
!	Логическое отрицание (НЕ). В качестве логических значений используется 0 – ложь и не 0 – истина, отрицанием 0 будет 1, отрицанием любого ненулевого числа будет 0
++	Инкремент или увеличение на единицу: префиксная операция – увеличивает операнд до его использования, постфиксная операция увеличивает операнд после его использования.
--	Декремент или уменьшение на единицу: префиксная операция – уменьшает операнд до его использования, постфиксная операция уменьшает операнд после его использования
sizeof	Вычисление размера (в байтах) для объекта того типа, который имеет операнд

Рисунок 41 – Унарные операции

Основные бинарные операции приведены в таблице 3, они выполняются слева направо.

Таблица 3 – Бинарные операции

Операции	Назначение	
+	Бинарный плюс (сложение арифметических операндов)	Мультипликативные
-	Бинарный минус (вычитание арифметических операндов)	
*	Умножение операндов арифметического типа	
/	Деление операндов арифметического типа (если операнды целочисленные, то выполняется целочисленное деление)	
%	Получение остатка от деления целочисленных операндов (операция по модулю)	
<<	Сдвиг влево битового представления значения левого целочисленного операнда на количество разрядов, равное значению правого операнда	Операции сдвига (определены только для целочисленных операндов)
>>	Сдвиг вправо битового представления значения правого целочисленного операнда на количество разрядов, равное значению правого операнда	
&	Поразрядная конъюнкция (И) битовых представлений значений целочисленных операндов	Поразрядные операции
|	Поразрядная дизъюнкция (ИЛИ) битовых	

<table>
<tr><td></td><td>представлений значений целочисленных операндов</td><td rowspan="2"></td></tr>
<tr><td>^</td><td>Поразрядное исключающее ИЛИ битовых представлений значений целочисленных операндов</td></tr>
<tr><td><</td><td>Меньше, чем</td><td rowspan="6">Операции сравнения</td></tr>
<tr><td>></td><td>Больше, чем</td></tr>
<tr><td><=</td><td>Меньш или равно</td></tr>
<tr><td>>=</td><td>Больше или равно</td></tr>
<tr><td>==</td><td>Равно</td></tr>
<tr><td>!=</td><td>Не равно</td></tr>
<tr><td>&&</td><td>Конъюнкция (И) целочисленных операндов или отношений, целочисленный результат ложь (0) или истина (1)</td><td rowspan="2">Логические бинарные операции</td></tr>
<tr><td>||</td><td>Дизъюнкция (ИЛИ) целочисленных операндов или отношений, целочисленный результат ложь (0) или истина (1)</td></tr>
<tr><td>=</td><td>Присваивание, присвоить значение правого операнда левому</td><td rowspan="11">Операции присваивания и составного присваивания</td></tr>
<tr><td>+=</td><td rowspan="10">Выполнить соответствующую операцию с левым операндом и правым операндом и присвоить рездьтат левому операнду</td></tr>
<tr><td>-=</td></tr>
<tr><td>*=</td></tr>
<tr><td>/=</td></tr>
<tr><td>%=</td></tr>
<tr><td>|=</td></tr>
<tr><td>&=</td></tr>
<tr><td>^=</td></tr>
<tr><td><<=</td></tr>
<tr><td>>>=</td></tr>
</table>

Приоритеты операций приведены в таблице 4

Таблица 4 – Приоритеты операций

Ранг	Операции
1	( ) [ ] –> .
2	! ~ – ++ -- & * (тип) sizeof тип ( )
3	* / % (мультипликативные бинарные)
4	+ - (аддитивные бинарные)
5	<< >> (поразрядного сдвига)
6	< > <= >= (отношения)
7	== != (отношения)
8	& (поразрядная конъюнкция «И»)
9	^ (поразрядное исключающее «ИЛИ»)

<table>
<tr><td>10</td><td>| (поразрядная дизъюнкция «ИЛИ»)</td></tr>
<tr><td>11</td><td>&& (конъюнкция «И»)</td></tr>
<tr><td>12</td><td>|| (дизъюнкция «ИЛИ»)</td></tr>
<tr><td>13</td><td>?: (условная операция)</td></tr>
<tr><td>14</td><td>= *= /= %= -= &= ^= |= <<= >>= (операция присваивания)</td></tr>
<tr><td>15</td><td>, (операция запятая)</td></tr>
</table>

В отличие от унарных и бинарных операций в тернарных условных операциях используется три операнда:

```
Выражение1 ? Выражение2 : Выражение3;
```

Первым вычисляется значение выражения 1. Если оно истинно, то вычисляется значение выражения 2, которое становится результатом. Если при вычислении выражения 1 получится 0, то в качестве результата берется значение выражения 3.

```
x<0 ? -x : x; //вычисляется абсолютное значение x
```

Выражения

Комбинация знаков операций и операндов, результатом которой является определенное значение, называется выражением. Каждый операнд в выражении может быть выражением. Значение выражения зависит от расположения знаков операций и круглых скобок в выражении, а также от приоритета выполнения операций. Каждое выражение состоит из одного или нескольких операндов, символов операций и ограничителей:

```
X * 12 + Y
val < 3
-9
```

Выражение, после которого стоит точка с запятой – это оператор-выражение.

Рассмотрим подробнее некоторые операции и варианты их использования.

Операция присваивания (=) рассматривается как выражение, имеющее значение левого операнда после присваивания. Присваивание может включать несколько операций присваивания, изменяя значения нескольких операндов, например:

```
long a; char b; int c, x, y, z;
a = b = c; //эквивалентно b = c; a = b;
x = i + (y = 3) - (z = 0);
//z = 0, y = 3, x = i + y - z;
```

Недопустимыми являются: присваивание константе, присваивание функции и присваивание результату операции.

Операции инкремента и декремента (++, --) относятся к унарным арифметическим операциям, которые служат соответственно для увеличения или уменьшения значения, хранимого в переменной целого типа. Например, следующие три оператора дадут один и тот же эффект:

```
sum = sum + 1;
sum += 1;
++sum;
```

Операции инкремента и декремента не только изменяют значения переменных, но и возвращают значения. Таким образом, их можно сделать частью более сложного выражения:

```
int i, j, a;
float m, y;
........................................
m *= y;  //m = m * y;
i += 2; //i = i + 2;
m /= y + 1; //m = m / (y + 1);
--a; //a = a - 1;
j = i++; //j = i; i = i + 1;
j = ++i; //i = i + 1; j = i;
```

Имеется постфиксная и префиксная форма операторов инкремента и декремента. В постфиксной форме записи переменная, к которой применена операция, увеличивается (или уменьшается) только после того, как ее значение будет использовано в контексте.

Типичной ошибкой является попытка использовать в операции инкремента или декремента операнд, отличный от имени простой переменной:

```
++(x + 1); //ошибка
```

Общий вид *операций сравнения* (отношения):

```
<выражение 1> <знак операции> <выражение 2>
```

Выражениями могут быть любые базовые (скалярные) типы. Значения выражений перед сравнением преобразуются к одному типу.

Результат операции сравнения – значение 1, если отношение истинно, или 0 в противном случае (ложно). Операция сравнения может использоваться в любых арифметических выражениях:

```
int b = 5;
int c = 10;
a = b > c;          //Запомнить результат сравнения a=0
a = (b > c)* 2;     //a= 0
```

или

```
(a < b && b < c)
                    //если ОДНОВРЕМЕННО ОБА a < b
                    //и b < c, то истина, иначе ложь
if (a == 0 || b > 0)
                    //если ХОТЯ БЫ ОДИН a==0
                    //или b > 0, то истина, иначе ложь
!0                  //1
!10                 //0
!((x = 1) < 0)          //1
0 < x < 100             //ошибка
(0 < x) && (x < 100) //верно
```

В C/C++ предусмотрены битовые операции для работы с отдельными битами. Эти операции нельзя применять к переменным вещественного типа. Операндами операций над битами могут быть только выражения, приводимые к целому типу. Операции (~, &, |, ^) выполняются поразрядно над всеми битами операндов (знаковый разряд не выделяется).

Общий вид операции инвертирования:

```
~ <выражение>
```

Остальные операции над битами имеют вид:

```
<выражение 1> <знак операции> <выражение 2>
```

Ниже приведен рисунок 42 истинности *логических операций* &, | и ^.

Операнд 1	Операнд 2	&	|	^
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

Рисунок 42 – Логиеские операции &, | и ^.

Операция & часто используется для маскирования некоторого множества бит. Операция ! используется для включения (устанавливает в единицу те биты, которые были нулями).

Необходимо отличать *побитовые операции* & и ! от логических *бинарных операций* && и ||

```
x = 1;
y = 2;
x & y          //результат 0, т.к. 0001 & 0010=0000
x && y         //результат 1, т.к. в операц. сравнения
               //оба операнда истина
```

Арифметические операции задают обычные действия над операндами арифметического типа.

```
i % j          //i - ( i/j ) * j
12 % 6         //0
13 % 6         //1
```

Если арифметическая операция содержит операнды различных типов, то компилятор выполняет автоматическое преобразование их типов.

Часто арифметические операции используются для обработки чисел, например:

```
int n = 12345;
int low, i = 6;
low = n %10;  //младшая цифра числа n
n = n / 10;    //отбросить младшую цифру числа n
if ( n % i == 0)... //определить - n делится нацело на i ?
n = n * 10 + i;    //добавить цифру i к значению числа n
```

Операции сдвига << и >> осуществляют соответственно сдвиг вправо и влево своего левого операнда на число позиций, задаваемых правым

операндом. Операции сдвига выполняются также для всех разрядов с потерей выходящих за границы бит.

```
0x81 << 1 //10000001<<1=00000010=0x02
0x81 >> 1 //10000001>>1=01000000=0x40
```

Если левостоящее выражение имеет тип unsigned, то при сдвиге вправо освобождающиеся разряды гарантированно заполняются нулями (логический сдвиг). Выражения типа signed могут, но не обязательно, сдвигаться вправо с копированием знакового разряда (арифметический сдвиг). При сдвиге влево освобождающиеся разряды всегда заполняются нулями. Если правостоящее выражение отрицательно либо больше длины левостоящего выражения в битах, то результат операции сдвига не определен.

Операции сдвига вправо на k разрядов можно использовать для деления, а сдвиг влево – для умножения целых чисел на 2 в степени k:

```
x << 1; //x * 2
x >> 1; //x / 2
x << 3; //x * 8
```

Операция sizeof выполняется на этапе компиляции программы и дает константу, которая равна числу байтов, требуемых для хранения в памяти данного объекта. Объектом может быть имя переменной, массива, структуры или просто спецификация типа.

```
int i;
cout << sizeof(int) << sizeof(i);
```

Операторы

Основной источник операторов в программе – выражения. Любое из них, ограниченное символом ;, превращается в оператор.

Запись действий, которые должен выполнить компьютер, состоит из операторов. При выполнении программы операторы выполняются один за другим, за исключением операторов управления, которые могут изменить последовательное выполнение программы. Различают операторы объявления переменных, операторы управления и операторы-выражения. Простейшей формой оператора является пустой оператор: ;

Он ничего не делает. Однако он может быть полезен в тех случаях, когда синтаксис требует наличие оператора, а он не нужен.

Любая последовательность операторов, заключенная в фигурные скобки {}, может выступать в любой синтаксической конструкции как один составной оператор (блок):

```
{
    a = b + 2;
    b++;
}
```

Он позволяет рассматривать несколько операторов как один. Область видимости имени, описанного в блоке, простирается до конца блока.

Операция запятая (,) используется при организации строго гарантированной последовательности вычисления выражений (используется там, где по синтаксису допустима только одна операция, и для организации

множественных выражений, расположенных внутри круглых скобок). Форма записи:

```
выражение 1, …, выражение N;
```

Выражения вычисляются слева направо. Например:

```
char X, Y;
(X = Y, Y = getch());
```

//присваивает переменной X значение Y, считывает символ,
//вводимый с клавиатуры, и запоминает его в Y

```
int i, j, k, n;
m = (i = 1, j = i++, k = 6, n = i + j + k);
```

//i = 1, j = i = 1, i = 2, k = 6, n = 2 + 1 + 6, m = n = 9

Выражения, разделенные запятой, не должны быть присваиваниями. Ниже приведенный пример не является примером хорошего кода:

```
int one = 1, two = 2, three = 100, six = 0;
six = (++one, ++two, ++three);
//one = 2, two = 3, three = 101, six = 101
```

Оператор условия и оператор выбора альтернатив

Все операторы управления могут быть условно разделены на следующие категории:

– условные операторы, к которым относятся оператор условия `if` и оператор выбора `switch`;

– операторы цикла (`for, while, do while`);

– операторы перехода (`break, continue, return, goto`).

Условный оператор if

Формат оператора:

```
if <выражение-условие> оператор-1; [else оператор-2;]
```

//полная форма

```
if <выражение-условие> оператор-1;
```

//сокращенная форма

Выполнение оператора `if` начинается с вычисления `<выражения-условия>`. В качестве <выражения-условия> может использоваться арифметическое выражение, отношение и логическое выражение. Далее выполнение осуществляется по следующей схеме: если выражение истинно (т.е. отлично от 0), то выполняется оператор 1, если выражение ложно (т.е. равно 0), то выполняется оператор 2, если выражение ложно и отсутствует оператор 2 (в квадратные скобки заключена необязательная конструкция), то выполняется следующий за if оператор. Блок-схемы операторов if и if-else приведены на рисунке 43

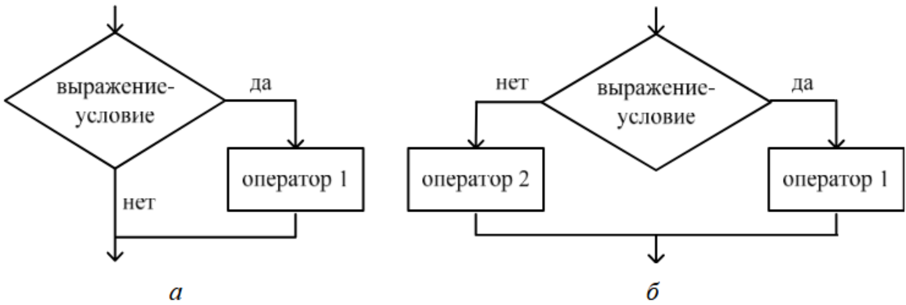


Рисунок 43 – Блок-схемы условных операторов

Пример,

```
if (i < j) i++;
     else
          {
               j = i- 3;
               i++;
          }
```

После выполнения оператора `if` значение передается на следующий оператор программы, если последовательность выполнения операторов не будет принудительно нарушена.

Допускается использование вложенных операторов `if`. Оператор `if` может быть включен в конструкцию `if` или в конструкцию `else` другого оператора `if`. Рекомендуется группировать операторы и конструкции, используя фигурные скобки. Если же фигурные скобки опущены, то компилятор связывает каждое ключевое слово `else` с наиболее близким `if`, для которого нет `else`:

```
int t = 2, b = 7, r = 3;
if (t > b)
{
     if (b < r) r = b;
}
 else r = t; //r станет равным 2
```

Если в программе опустить фигурные скобки, то получится:

```
int t = 2, b = 7, r = 3;
if (t > b)

     if (b < r) r = b;

 else r = t; //r станет равным 2
```

Конструкции, использующие вложенные операторы `if`, являются громоздкими и не всегда достаточно надежными.

Можно комбинировать условные выражения и логические операции. Ниже приведен пример, в котором определяется относится ли введенный символ к буквам и используется единственный оператор `if`:

```
char letter = 0;
cout << endl
<< "Введите символ: ";
```

```
cin >> letter;
if(((letter>='A')&&(letter<='Z'))||((letter>='a')&&(letter<='z')))
     cout << endl<< "Вы ввели букву." << endl;
else
     cout << endl<< "Вы ввели не букву." << endl;
```

Считается хорошим стилем программирования соблюдение одинакового числа пробелов в отступах структуры `if-else`.

Оператор выбора `switch`

Другим способом организации выбора из множества различных вариантов является использование специального оператора выбора `switch`. Формат оператора следующий:

```
switch (выражение)
     {[объявление]
               :
     [case константное-выражение1]: [список-операторов1]
     [case константное-выражение2]: [список-операторов2]
               :
               :
     [default: [список операторов]]
     }
```

Выражение, следующее за ключевым словом `switch` в круглых скобках, может быть любым выражением, допустимым в языке C/C++, значение которого должно быть целым. Значение этого выражения является ключевым для выбора из нескольких вариантов. Тело оператора `switch` состоит из нескольких операторов, помеченных ключевым словом `case` с последующим константным выражением.

Обычно в качестве константного выражения используются целые или символьные константы (не может содержать переменные или вызовы функций). Все константные выражения в операторе `switch` должны быть уникальны. Кроме операторов, помеченных ключевым словом `case`, может быть, но обязательно один, фрагмент, помеченный ключевым словом `default`.

Список операторов может быть пустым либо содержать один оператор или более. В операторе `switch` не требуется заключать в фигурные скобки последовательность операторов.

Можно использовать свои локальные переменные, объявления которых находятся перед первым ключевым словом `case`, однако в объявлениях не должна использоваться инициализация.

Схема выполнения оператора `switch` следующая: вычисляется выражение в круглых скобках; вычисленные значения последовательно сравниваются с константными выражениями, следующими за ключевыми словами `case`; если одно из константных выражений совпадает со значением выражения, то управление передается на оператор, помеченный соответствующим ключевым словом `case`, если ни одно из константных выражений не равно выражению, то управление передается на оператор, помеченный ключевым словом `default`, а в случае его отсутствия управление передается на следующий после `switch` оператор.

Пример,

```
int i=2;
switch (i)
     {
          case 1: i += 2;
          case 2: i *= 3;
          case 0: i /= 2;
          case 4: i -= 5;
          default: ;
```

Рассмотрим ранее приведенный пример, в котором иллюстрировалось использование вложенных операторов `if`, переписанных теперь с использованием оператора `switch`:

```
char ZNAC;
int x, y, z;
switch (ZNAC)
{
     case '+': x = y + z; break;
     case '-': x = y - z; break;
     case '*': x = y * z; break;
     case '/': x = u / z; break;
     default : ;
}
```

Использование оператора `break` позволяет в необходимый момент прервать последовательность выполняемых операторов в теле оператора `switch` путем передачи управления оператору, следующему за `switch`.

В теле оператора `switch` можно использовать вложенные операторы `switch`, при этом в ключевых словах `case` можно использовать одинаковые константные выражения:

```
switch (a)
{
     case 1: b=c; break;
     case 2:
     switch (d)
          { case 0: f=s; break;
            case 1: f=9; break;
            case 2: f-=9; break;
          }
     case 3: b-=c; break;
}
```

Циклические конструкции и операторы переходов

При выполнении программы нередко возникает необходимость неоднократного повторения однотипных вычислений над различными данными. Для этих целей используются циклы.

Цикл – участок программы, в котором одни и те же вычисления реализуются неоднократно над различными значениями одних и тех же переменных (объектов).

Для организации циклов в С++ используются следующие операторы: `for, while, do while.`

Оператор цикла for

Цикл `for` является циклом с параметрами и обычно используется в случае, когда известно точное количество повторов вычислений.

Оператор `for` – это наиболее общий способ организации цикла.

Он имеет следующий формат:

```
for (выражение 1;выражение 2;выражение 3) {тело цикла}
```

`Выражение 1` обычно используется для установления начального значения переменных, управляющих циклом. `Выражение 2` – это выражение, определяющее условие, при котором тело цикла будет выполняться. `Выражение 3` определяет изменение переменных, управляющих циклом после каждого выполнения тела цикла.

Схема выполнения оператора `for`: вычисляется выражение 1; вычисляется выражение 2; если значение выражения 2 отлично от нуля (истина), выполняется тело цикла, вычисляется выражение 3 и осуществляется переход к пункту 2; если выражение 2 равно нулю (ложь), то управление передается на оператор, следующий за оператором `for`.

Проверка условия всегда выполняется в начале цикла. Это значит, что тело цикла может ни разу не выполниться, если условие выполнения сразу будет ложным. Например:

```
int i, b;
for (i = 1; i < 10; i++) b = i * i;
//вычисление квадратов чисел от 1 до 9
for (i = 1; i > 10; i++) b = i * i;
//тело цикла не выполнится
```

Желательно в разделе задания начальных значений и изменения переменных структуры for задавать только выражения, относящиеся к управляющей переменной. Манипуляции с другими переменными должны размещаться или до цикла (если они выполняются только один раз подобно операторам задания начальных значений), или внутри тела цикла (если они должны выполняться в каждом цикле, как, например, операторы инкремента или декремента).

Некоторые варианты использования оператора for повышают его гибкость за счет возможности использования нескольких переменных, управляющих циклом:

```
int top, bot;
char strin[100], temp;
//для управления циклом используются
//две переменные top и bot
for (top = 0, bot = 100; top < bot; top++, bot--)
{
     temp = strin[top];
     strin[bot] = temp;
}
```

Важно всегда задавать начальные значения всем счетчикам и переменным сумм цикла. Управлять количеством повторений цикла нужно с помощью целой переменной. Хорошим стилем программирования является

размещение пустой строки до и после каждой управляющей структуры, чтобы она выделялась в программе.

Другим вариантом использования оператора `for` является бесконечный цикл. Для организации такого цикла можно использовать пустое условное выражение, а для выхода из цикла обычно используют дополнительное условие и оператор `break`:

```
for (;;)
{ ...
... break;
...
}
```

Так как согласно синтаксису языка оператор может быть пустым, тело оператора `for` также может быть пустым. Такая форма оператора может быть использована для организации поиска:

```
for (int i = 0; t[i] < 10; i++) ;
//i – номер первого элемента массива t,
//значение которого больше 10
```

В данном примере в теле цикла происходит объявление переменной `i`. Цикл имеет область видимости, распространяющуюся от управляющих выражений `for` до конца его тела. Так как счетчик i объявлен внутри области видимости цикла, к нему нельзя обращаться за пределами этой области (он исчезает).

Можно вообще исключить установку начальных значений из цикла. Если инициализировать `i` в отдельном операторе объявления, то цикл можно записать так:

```
int i=0;
for (; t[i] < 10 ; i++) ;
```

Фактически обе точки с запятой всегда должны присутствовать независимо от того, пропущено ли какое-то одно из управляющих выражений или даже все сразу.

Оператор цикла `while`

Оператор цикла `while` называется циклом с предусловием и имеет следующий формат:

```
while (выражение) тело ;
```

В качестве выражения допускается использовать любое выражение языка C/C++, в качестве тела – любой оператор, в том числе пустой или составной.

Схема выполнения оператора `while` следующая: вычисляется выражение; если выражение ложно, то выполнение оператора `while` заканчивается и выполняется следующий по порядку оператор; если выражение истинно, то выполняется тело оператора `while`; процесс повторяется сначала.

Рассмотрим пример:

```
double value = 0.0;      //для ввода значений
double sum = 0.0;        //сумма значений
int i = 0;               //счетчик значений
char indicator = 'y';    //индикатор продолжения
```

```
while(indicator == 'y') //повторять цикл пока 'y'
	{
		cout << endl<< "Введите значение: ";
		cin >> value;                  //ввести значение
		++i;                      //увеличить счетчик
		sum += value;       //добавить значение к сумме
		cout << endl
		<< " Хотите ввести еще значение ? ";
		cin >> indicator;            //ввести индикатор
	}
```

Внутри операторов for и while можно использовать локальные переменные, которые должны быть объявлены с определением соответствующих типов.

Типичной ошибкой программирования является отсутствие в теле структуры `while` действия, которое приведет со временем к ложному условию `while` . Оно называется зацикливание.

Оператор цикла `do while`

Оператор цикла `do while` называется оператором цикла с постусловием и используется в тех случаях, когда необходимо выполнить тело цикла хотя бы один раз. Формат оператора имеет следующий вид:

```
do тело while (выражение);
```

Схема выполнения оператора `do while`: выполняется тело цикла (которое может быть составным оператором); вычисляется выражение; если выражение ложно, то выполнение оператора `do while` заканчивается и выполняется следующий по порядку оператор; если выражение истинно, то выполнение оператора продолжается сначала.

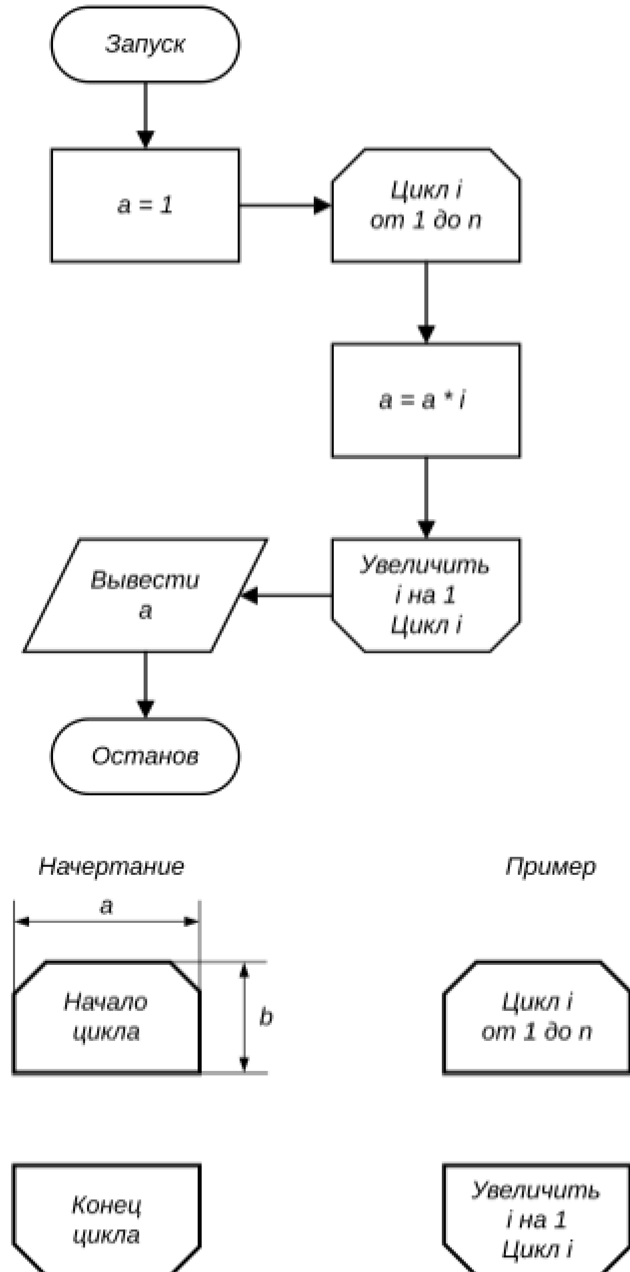
Чтобы прервать выполнение цикла до того, как условие станет ложным, можно использовать оператор `break`.

Операторы `while`, `for` и `do while` могут быть вложенными:

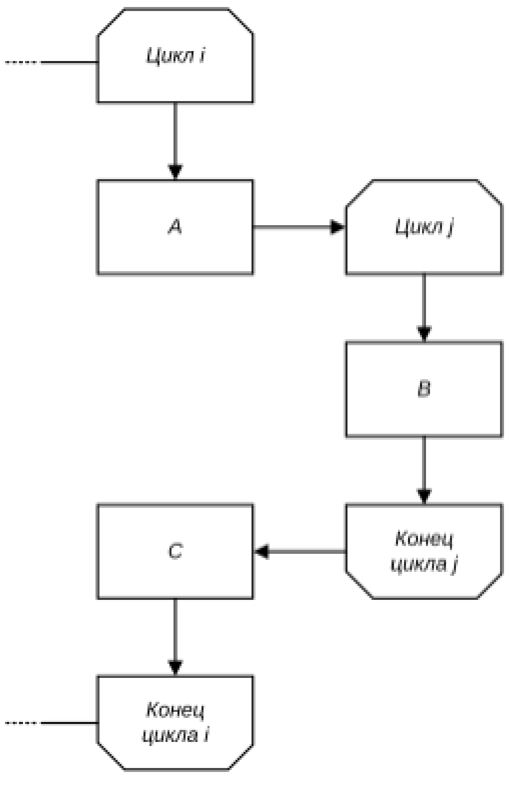
```
do
{
	cout << endl<< "Введите целое число: ";
	cin >> value;
	factorial = 1;
	for(int i = 2; i <= value; i++)
	factorial *= i;
	cout<<"Факториал"<<value<<"равен"<< factorial;
	cout << endl<<"Хотите ввести еще значение (y или n)?";
	cin >> indicator;
} while((indicator == 'y') || (indicator == 'Y'));
```

При выборе варианта циклической конструкции лучше использовать цикл `for`. Это всегда можно сделать, если заранее известно число повторений цикла. В остальных случаях используются циклы `while` и `do while`, причем цикл `do while` следует применять, если требуется тело цикла выполнить не менее одного раза.

Пример блок-схемы с оператором цикла:



Пример вложенных циклов:

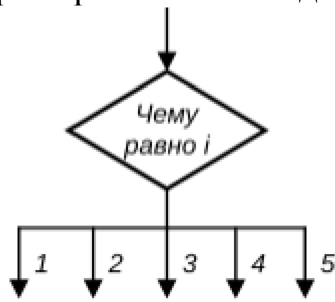


Оператор перехода `break`

Оператор `break` изменяет поток управления, он прерывает цикл. Его целесообразно использовать, когда условие продолжения итераций надо проверять в середине цикла:

```
//ищет сумму чисел до тех пор,
//пока не будет введено 100 чисел или 0
for(s = 0, i = 1; i < 100; i++)
{
     cin >> x;
     if( x == 0) break;
     //если ввели 0, то суммирование заканчивается
     s += x;
}
```

Пример блок-схемы для оператора выбора switch:



Оператор перехода `continue`

Оператор continue, как и оператор `break`, используется внутр операторов цикла, но в отличие от него выполнение программы продолжается со следующей итерации, оставшаяся часть тела структуры пропускается.

Формат оператора следующий:

```
continue;
```

Пример,

```
int a,b;
for (a = 1,b = 0; a < 100; b += a,a++)
{
     if (b % 2) continue;
     //передача управления на очередную итерацию цикла for
     //без выполнения операторов обработки четных сумм
     ... //обработка четных сумм
}
```

Оператор безусловного перехода `goto`

Использование оператора безусловного перехода `goto` в практике программирования C/C++ настоятельно не рекомендуется, так как он затрудняет понимание программы и возможность ее модификации.

Формат оператора следующий:

```
goto имя-метки;
...
имя-метки: оператор;
```

Оператор `goto` передает управление на оператор, помеченный меткой `имя-метки`. Помеченный оператор должен находиться в той же функции, что и оператор `goto`, а используемая метка должна быть уникальной, т.е. одно `имя-метки` не может быть использовано для разных операторов программы. Приведем пример организации цикла, используя только `goto` и `if`:

```
int i = 0, sum = 0;
const int max = 10;
i=1;
loop:
     sum += i; //накопление суммы целых чисел от 1 до 10
if (++i <= max) goto loop;
```

Любой оператор в составном операторе может иметь свою метку. Используя оператор `goto`, можно передавать управление внутрь составного оператора.

Любая программа может быть написана без `goto`. Однако существуют ситуации (их немного), когда `goto` рекомендуется использовать, например, выход из многократно вложенных циклов (поскольку оператор `break` осуществляет выход только из того цикла, где он использован):

```
for (...)
     for (...)
     {...
          if ( ошибка ) goto Error;
     }
     ...
Error :
```

В остальных случаях использовать `goto` не следует.

Оператор `return`

Оператор `return` завершает выполнение функции, в которой он задан, и возвращает управление в вызывающую функцию, в точку, непосредственно

следующую за вызовом. Функция `main` передает управление операционной системе. Формат оператора:

```
return [выражение];
```

Значение выражения, если оно задано, возвращается в вызывающую функцию в качестве значения вызываемой функции. Если выражение опущено, то возвращаемое значение не определено. Если функция не должна иметь возвращаемого значения, то ее нужно объявлять с типом `void`.

Пример,

```
int sum (int a, int b)
{
    return (a+b);
}
```

Лекция 8 Структурированные типы данных

Одномерные массивы и указатели

Массив (или вектор) – это группа связанных друг с другом элементов одного типа (`double`, `float`, `int` и т.п.), последовательно расположенных в памяти. Из объявления массива компилятор должен получить информацию о типе элементов массива и их количестве.

Объявление массива имеет два формата:

```
спецификатор-типа описатель [константное - выражение];
спецификатор-типа описатель [ ];
```

Описатель – это идентификатор массива. Спецификатор-типа задает тип элементов объявляемого массива. Элементами массива не могут быть функции и элементы типа `void`.

Константное выражение в квадратных скобках задает количество элементов массива. Константное выражение при объявлении массива может быть опущено в следующих случаях:

- при объявлении массив инициализируется;
- массив объявлен как формальный параметр функции;
- массив объявлен как ссылка на массив, явно определенный в другом файле.

Пример определения массива:

```
int arr[5];
```

Элементы массива занимают один непрерывный участок памяти компьютера и располагаются последовательно друг за другом (рисунок 44).

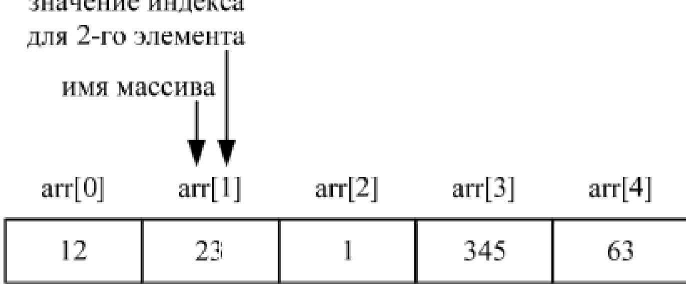


Рисунок 44 - Расположение элементов массива

Доступ к отдельным элементам массива организуется использованием номера этого элемента или индекса ([ ]). Нумерация элементов массива начинается с нуля и заканчивается n – 1, где n – число элементов массива.

Важно различать, например, седьмой элемент массива и элемент массива семь. Поскольку индексы начинаются с 0, седьмой элемент массива имеет индекс шесть, тогда как элемент массива семь имеет индекс 7 и на самом деле является восьмым элементом массива.

Это источник ошибок типа завышения (или занижения) на единицу.

Дело в том, что C++ не проверяет границ массивов и не предупреждает о ссылках на несуществующие элементы.

При объявлении автоматических и статических массивов для задания их размеров лучше использовать именованные константы:

```
const int array_size = 10;
int arr[array_size];
```

Это делает программу более масштабируемой, при изменении размера массива надо будет поменять число только в одном месте.

Инициализация и работа с массивами

Инициализация массива означает присвоение начальных значений его элементам при объявлении. Массивы можно инициализировать списком значений или выражений, отделенных запятой, заключенных в фигурные скобки:

```
double d[] = {1, 2, 3, 4, 5};
int n[5] = {2};
```

Длина массива вычисляется компилятором по количеству значений, перечисленных в фигурных скобках. Если размер массива задан, нельзя специфицировать значений больше, чем объявлено в массиве (синтаксическая ошибка), но меньше указать можно (минимально должен быть определен один элемент). Если массив явно не проинициализирован, то внешние и статические массивы инициализируются нулями. Автоматические массивы после объявления ничем не инициализируются и содержат неизвестную информацию.

Например, необходимо сформировать одномерный стаический массив целых чисел, используя датчик случайных чисел (диапазон значений от 0 до 99); подсчитать среднее арифметическое элементов в массиве; удалить элемент с заданным номером. Ниже приведен пример фрагмента листинга программы:

```
const int N = 1000;
//N - максимальный размер статического массива
.............................................
int i; //индекс массива
int array_size; //переменная для хранения размера массива
int avg; //переменная для хранения
//среднего арифметического
int arr[N]; //целочисленный статический массив длины N
int k; //индекс удаляемого элемента
int rmin = 0, rmax = 99;
```

```
//диапазон значений элементов массива
cout<<"Введите размер массива"<<endl;
cin>> array_size<<endl;
//Сгенерировать массив
srand((unsigned)time(NULL));
for(i = 0 ; i < array_size ; i++)
{
     arr[i] = (int)(((double)rand()/(double)RAND_MAX)*
     (rmax-rmin)+rmin);
     //или arr[i] = rand()%99;
     cout<<arr[i]<<endl;
}
//Подсчитать среднее арифметическое суммы элементов:
avg = 0;
for(i = 0; i < array_size; i++)
{
     //т.к. в теле цикла один оператор,
     //фигурная скобка не обязательна
     avg += arr[i];
}
avg /= array_size;
cout<<"Средний элемент массива"<<avg<<endl;
cout<<"Введите номер удаляемого элемента массива"<<endl;
cin >>k;
//удалить элемент с номером k, сдвиг элементов массива
for(i = k; i < array_size - 1; i++)
{
     arr[i] = arr[i+1];
}
array_size--; //уменьшить размер массива
for(i=0;i< array_size; i++) //вывод массива
     cout<<arr[i]<<endl;
```

Ссылочный тип

Ссылочный тип, иногда называемый псевдонимом, служит для задания объекту дополнительного имени. Ссылка позволяет косвенно манипулировать объектом, точно так же, как это делается с помощью указателя. Однако эта косвенная манипуляция не требует специального синтаксиса, необходимого для указателей. Обычно ссылки употребляются как формальные параметры функций. Ссылочный тип обозначается указанием оператора взятия адреса (`&`) перед именем переменной.

```
int ival = 1024;
int &refVal = ival; //refVal - ссылка на ival
int &refVal2; //ошибка: ссылка должна быть
//инициализирована
```

Ссылка должна быть инициализирована, она должна быть инициализирована не адресом объекта, а его значением.

```
int ival = 1024;
int &refVal = &ival; //ошибка: refVal имеет тип int,
//а не int*
int *pi = &ival; //ptrVal - ссылка на указатель
```

Определив ссылку, нельзя изменять ее так, чтобы работать с другим объектом. Все операции со ссылками воздействуют на адресуемые ими объекты, в том числе и операция взятия адреса. Если ссылки определяются через запятую, перед каждым объектом типа ссылки должен стоять амперсанд (&) (точно так же, как и для указателей). Например:

```
int ival = 1024, ival2 = 2048;
//определены переменные типа int
int &rval = ival, rval2 = ival2;
//определена одна ссылка и одна переменная
int ival3 = 1024, *pi = ival3, &ri = ival3;
//определена переменная, указатель и ссылка
int &rval3 = ival3, &rval4 = ival2;
//определены две ссылки
```

Между ссылкой и указателем существуют два основных отличия.

Во-первых, ссылка обязательно должна быть инициализирована в месте своего определения. Во-вторых, всякое изменение ссылки преобразует не ее, а тот объект, на который она ссылается. Например:

```
int *pi = 0;
const int &ri = 0;
int ival = 1024, ival2 = 2048;
int *pi = &ival, *pi2 = &ival2;
pi = pi2; //pi и pi2 указывают на переменную ival2
int &ri = ival, &ri2 = ival2;
ri = ri2;
//ival меняется,ссылка ri по-прежнему адресует ival
```

В реальных C++ программах ссылки редко используются как самостоятельные объекты, обычно они употребляются в качестве формальных параметров функций.

Генерация случайных чисел

В приведенном примере массив заполняется случайными числами. Элемент случайности может быть введен в приложения с помощью функции `rand` из стандартной библиотеки:

```
i = rand () ;
```

Функция rand генерирует целое число в диапазоне от 0 до `RAND_MAX` (символическая константа, определенная в заголовочном файле `<stdlib>`). Значение `RAND_MAX` должно быть, по меньшей мере, равно 32767 – максимальное положительное значение двухбайтового целого числа. Для того, чтобы выработать целые числа в заданном диапазоне, используется операция вычисления остатка (`%`) в сочетании с rand:

```
i = rand () % 6 ; //генерирует случайное число от 0 до 5
```

Это называется масштабированием, а число 6 называется масштабирующим коэффициентом.

Функция `rand`, на самом деле, генерирует псевдослучайные числа. Повторный вызов rand производит число, которое кажется случайным. Но то же самое число повторяется при каждом повторении программы. Для рандомизации необходимо использование стандартной библиотечной

функции srand. Функция srand получает целый аргумент unsigned и при каждом выполнении программы задает начальное число, которое функция rand использует для генерации последовательности квазислучайных чисел.

Для того, чтобы не вводить каждый раз начальное число, можно использовать оператор:

```
srand((unsigned)time(NULL));
```

Для автоматического получения начального числа считываются показания часов. Функция time (с аргументом NULL, как записано в указанном выше операторе) возвращает текущее «календарное время» в секундах. Это значение преобразуется в беззнаковое целое число и используется как начальное значение в генераторе случайных чисел.

Прототип функции для time находится в <ctime>.

Функция srand должна вызываться только один раз в программе для достижения желаемого результата рандомизации. Вызов ее более одного раза является избыточным и снижает эффективность программы.

Строки

Для представления символьной информации можно использовать символы, символьные переменные и символьные константы.

В C++ поддерживаются два типа строк – встроенный тип, доставшийся от C, и класс string из стандартной библиотеки C++. Класс string предоставляет гораздо больше возможностей и поэтому удобнее в применении.

Встроенный строковый тип

Встроенный строковый тип перешел в C++ от C. Строка символов хранится в памяти как массив, и доступ к ней осуществляется при помощи указателя типа char* . Количество элементов в таком массиве на один элемент больше, чем изображение строки, т.к. в конец строки добавлен '\0' (нулевой байт или нуль-терминатор)(рисунок 45):

A		A	\0

'A' символ (1 байт)   "A" строка (2 байта)

Рисунок 45- Пример строки

Объявление и инициализация строк

Поместить строку в массив можно либо при вводе, либо с помощью инициализаций:

```
char str1[] = "STRING";
char str2[20] = { 'S','t','r','i','n','g','\0' };
const char *str3 = "STRING\n";
char str4[20];
cin >> str4;
//строка не должна превышать 19 симв. + 0 символ
```

Символьные строки хранятся в виде массивов, поэтому их нельзя приравнивать и сравнивать с помощью операций = и ==.

Типичной ошибкой является невыделение достаточного места в массиве символов для хранения нулевого символа, завершающего строку.

Строку можно присвоить символьному массиву, используя операцию `cin`. В некоторых случая желательно вводить в массив полную строку текста. С этой целью можно использовать функцию C++ `cin.getline`. Она требует три аргумента – массив символов, в котором должна храниться строка текста, длина и символ-ограничитель:

```
char str1[80];
cin.getline(str1, 80, '\n')
```

Функция прекращает считывать символы, если встречается символ-ограничитель '\n' и если количество считанных символов оказывается на один меньше, чем указано во втором аргументе. Третий аргумент имеет '\n' в качестве значения по умолчанию, так что в вызове функции его можно опустить.

Согласно общепринятому стандарту ASCII установлено соответствие между символами и кодами. Клавиатура (совместно с драйвером) кодирует нажатие любой клавиши с учетом регистровых и управляющих клавиш в соответствующий ей код. Например:

```
' ' - 0x20, 'B' - 0x42,
'*' - 0x2A, 'Y' - 0x59,
```

Манипуляции со строками и символами, на самом деле, подразумевают манипуляции с соответствующими численными кодами, а не с самими символами. Это объясняет взаимозаменяемость символов и малых целых в C++. Так как имеет смысл говорить, что один численный код больше, меньше или равен другому численному коду, можно сопоставлять различные строки и символы друг с другом.

Некоторые программы и стандартные функции обработки символов и строк используют тот факт, что цифры, прописные и строчные (маленькие и большие) латинские буквы имеют упорядоченные по возрастанию значения кодов:

```
'0' - '9' 0x30 - 0x39
'A' - 'Z' 0x41 - 0x5A
'a' - 'z' 0x61 - 0x7A
```

Тогда, для получения символа десятичной цифры из значения целой переменной, лежащей в диапазоне 0…9, а также значения целой переменной из символа десятичной цифры можно:

```
int n = 5;
char c;
c = n + '0';
if (c >= '0' && c <= '9')
n = c - '0';
```

Для преобразования строчной латинской буквы в прописную необходимо:

```
char c;
if (c >='a' && c <='z')
c = c - 'a' + 'A';
```

Типичной ошибкой является обработка одного символа как строки.

Работа со строкой

Обычно для перебора символов строки применяется адресная арифметика. Поскольку строка всегда заканчивается нулевым символом, можно увеличивать указатель на 1, пока очередным символом не станет нуль. Например:

```
while (*str1++ ) { ... }
```

`str1` разыменовывается, и получившееся значение проверяется на истинность. Любое отличное от нуля значение считается истинным, и, следовательно, цикл заканчивается, когда будет достигнут символ с кодом 0. Операция инкремента ++ прибавляет 1 к указателю `str1` и таким образом сдвигает его к следующему символу.

Подсчет длины строки может выглядеть следующим образом:

```
char st[] = "STRING";
int cnt = 0;
if ( st )
     while ( *st++ )
          ++cnt;
```

Поскольку указатель может содержать нулевое значение (ни на что не указывать), перед операцией разыменования его следует проверять.

Строка встроенного типа может считаться пустой в двух случаях: если указатель на строку имеет нулевое значение (строки нет) или указывает на массив, состоящий из одного нулевого символа (строка не содержит ни одного значимого символа).

```
//pstr1 не адресует массив символов
char *pstr1 = 0;
//pstr2 адресует нулевой символ
const char *pstr2 = "";
```

При работе со строкой можно, также как и в массивах, использовать нотацию индексов:

```
for (int i = 0; (s1[i] = s2[i])!= '\0'; i++);
//копирование строки
```

а также нотацию указателей:

```
for (; (*s1 = *s2) != '\0'; s1++, s2++);
//копирование строки
```

Использование строк встроенного типа чревато ошибками из-за слишком низкого уровня реализации и невозможности обойтись без адресной арифметики. Рассмотрим типичные ошибки. Например:

```
const char *str = "STRING\n";
int len = 0;
while ( str++ ) ++len;
//ошибка, str не разыменовывается и не изменяется
```

Указатель str не разыменовывается, следовательно, на равенство нулю проверяется не символ, а сам указатель. Поскольку изначальноэтот указатель имел ненулевое значение (адрес строки), то он никогда не станет равным нулю, и цикл будет выполняться бесконечно.

Поскольку строка представляет собой последовательность символов, большинство программ, обрабатывающих строки, используют последовательный или посимвольный просмотр строки. Если же в процессе обработки строки предполагается изменение ее содержимого, то проще всего (но не всегда эффективно) организовать его в виде посимвольного переписывания входной строки в выходную. При этом нужно помнить, что каждой строке требуется отдельный индекс, если для входной строки он может изменяться в заголовке цикла посимвольного просмотра, то для выходной строки он меняется только в моменты добавления очередного символа. Кроме того, не нужно забывать «закрывать» выходную строку символом конца строки. В качестве примера рассмотрим пример удаления лишних пробелов из строки:

```
char str1[] = "STR ING";
char str2[10];
int i,j;
for (j = 0, i = 0; str1[i] != 0; i++)
//Посимвольный просмотр строки
{
     if (str1[i] != ' ') //Текущий символ – не пробел
          {
          if (i!=0 && str1[i-1] == ' ') //Первый в слове -
          str2[j++] = ' '; //добавить пробел
          str2[j++] = str1[i]; //Перенести символ слова
          } //в выходную строку
     }
str2[j] = 0;
```

Функции работы со строками

Стандартная библиотека С предоставляет набор функций для манипулирования строками. Стандартная библиотека С является частью библиотеки С++. Для ее использования мы должны включить заголовочный файл `#include <cstring>` . Функции приведены в таблице 4.

Таблица 4 – Функции для работы со строками

Функция	Прототип и краткое описание функции
strcmp	`int strcmp(const char *str1, const char *str2);` Сравнивает строки `str1` и `str2`. Если `str1 < str2`, то результат отрицательный, если `str1 = str2`, то результат равен 0, если `str1 > str2`, то результат положительный
strcpy	`char* strcpy(char*s1, const char *s2);` Копирует байты из строки `s1` в строку `s2`
strdup	`char *strdup (const char *str);` Выделяет память и переносит в нее копию строки `str`.
strlen	`int strlen (const char *str);` Вычисляет длину строки `str`
strncat	`char *strncat(char *s1, const char *s2, int kol);`

	Приписывает kol символов строки s2 к строке s1
strncpy	char *strncpy(char *s1, const char *s2, int kol); Копирует kol символов строки s2 в строку s1
strnset	char *strnset(char *str, int c, int kol); Заменяет первые kol символов строки s1 символом c
atoi	int atoi(char *str); Преобразует строку в целое
atof	float atof(char *str); Преобразует строку в число с плавающей точкой

Строки при передаче в функцию в качестве фактических параметров могут быть определены либо как одномерные массивы типа char[], либо как указатели типа char*. В отличие от массивов, в этом случае нет необходимости явно указывать длину строки.

Для указателя с типом указуемой переменной char допускаются различные интерпретации: указатель на отдельный байт; указатель на область памяти – массив байтов; указатель на отдельный символ; указатель на массив символов.

Многомерные массивы

Объявление и инициализация многомерных массивов

Многомерными массивами в C++ называют массивы, которые имеют 2 и более индексов. Они формализуются списком константных выражений, следующих за идентификатором массива. Причем каждое константное выражение заключается в свои квадратные скобки. Константное выражение в квадратных скобках определяет число элементов по данному измерению массива, так что объявление двухмерного массива содержит два константных выражения, трехмерного – три и т.д.

Обычным представлением многомерных массивов являются таблицы значений, содержащие информацию в строках и столбцах. Чтобы определить отдельный табличный элемент, нужно указать два индекса: первый (по соглашению) указывает номер строки, а второй (по соглашению) указывает номер столбца. Таблицы или массивы, которые требуют двух индексов для указания отдельного элемента, называются двумерными массивами. Компиляторы C++ поддерживают, по меньшей мере, 12-мерные массивы.

Инициализировать многомерные массивы можно также, как одномерные. Например:

```
int a[2][3]; /* представлено в виде матрицы
a[0][0] a[0][1] a[0][2]
a[1][0] a[1][1] a[1][2] */
double b[10];
//вектор из 10 элементов имеющих тип double
int w[3][3] = { { 2, 3, 4 },
              { 3, 4, 8 },
              { 1, 0, 9 }};
int ia[4][3] = { {0}, {3}, {6}, {9} };
//инициализация первых элементов строк
```

Списки, выделенные в фигурные скобки, соответствуют строкам массива. В случае отсутствия скобок инициализация будет выполнена подряд идущими элементами, недостающие элементы будут неявно инициализированы. Для массива ia инициализируются только первые элементы каждой строки. Оставшиеся элементы будут равны нулю.

Если необходимо инициализировать нулями все значения массива, то можно выполнить:

```
long arr[2][3] = { 0 };
```

В языке C/C++ можно использовать сечения массива. Сечения формируются вследствие опускания одной или нескольких пар квадратных скобок. Пары квадратных скобок можно отбрасывать только справа налево и строго последовательно. Для `int s[2][3]` при обращении к `s[0]` будет передаваться нулевая строка массива s.

Конструкция `arr[1,2]` является допустимой с точки зрения синтаксиса C++, однако это не объявление двумерного массива размера 1×2. Значение в квадратных скобках – это список выражений через запятую, результатом которого будет последнее значение 2. Поэтому объявление `arr[1,2]` эквивалентно `arr[2]`.

Указатели на многомерные массивы – это массивы массивов, т.е. такие массивы, элементами которых являются массивы. Например, при выполнении объявления двумерного массива `arr[4][3]` в памяти выделяется участок для хранения значения переменной arr, которая является указателем на массив из четырех указателей. Для этого массива из четырех указателей тоже выделяется память. Каждый из этих четырех указателей содержит адрес массива из трех элементов типа int, и, следовательно, в памяти компьютера выделяется четыре участка для хранения четырех массивов чисел типа int, каждый из которых состоит из трех элементов. Такое выделение памяти показано на рисунке 46.

arr				
↓				
arr[0]	→	arr[0][0]	arr[0][1]	arr[0][2]
arr[1]	→	arr[1][0]	arr[1][1]	arr[1][2]
arr[2]	→	arr[2][0]	arr[2][1]	arr[2][2]
arr[3]	→	arr[3][0]	arr[3][1]	arr[3][2]

Рисунок 46– Многомерный массив

Для доступа к элементам двумерного массива должны быть использованы два индексных выражения в форме `arr[m][n]` или эквивалентных ей `*(*(arr+m)+n)`, `*(arr[m]+n)` и `(*(arr+m))[n]`. Действительно `arr+m` ссылается на строку номер m. Выражение `*(arr+m)` – это адрес нулевого элемента строки m , поэтому `*(arr+m)+n` – адрес элемента в строке m со смещением `n`. Таким образом, полное выражение ссылается на конкретный элемент массива.

С точки зрения синтаксиса языка указатель arr и указатели arr[0], arr[1], arr[2], arr[3] являются константами, и их значения нельзя изменять во время выполнения программы.

Инициализация двумерного массива с помощью вложенных циклов будет выглядеть следующим образом:

```
const int rowSize = 3;
const int colSize = 3;
int arr[ rowSize ][ colSize ];
     for ( int = 0; i < rowSize; ++i )
          for ( int j = 0; j < colSize; ++j )
               arr[ i ][ j ] = i + j ;
```

При размещении элементов многомерных массивов они располагаются в памяти подряд по строкам. Такой порядок дает возможность обращаться к любому элементу многомерного массива, используя адрес его начального элемента и только одно индексное выражение:

```
float *ptr, *ptr3, arr[4][4][4];
ptr = arr;
ptr3 = arr;
//тогда следующие обращения к элементам будут эквивалентными
//arr[2][3][4] == ptr[3*2+4*3+4] == ptr3[22];
```

Элементы массивов могут иметь любой тип, и, в частности, могут быть указателями на любой тип. Следующие объявления переменных:

```
int a[]={10,11,12,13,14,};
int *p[]={a, a+1, a+2, a+2, a+3, a+4};
int **pp=p;
```

порождают программные объекты, представленные на рисунке 47.

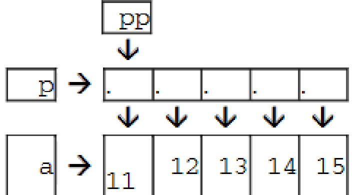


Рисунок 47 – Результат работы фрагмента программы

Если считать, что pp = p, то обращение *++pp – значение первого элемента массива a (т.е. значение 11), операция ++*pp изменит содержимое указателя p[0] таким образом, что он станет равным значению адреса элемента a[1].

Независимо от размерности массива объем занимаемой им памяти определяется достаточно просто, с помощью операции sizeof:

```
Размер массива = sizeof имя_массива/sizeof(тип_данных);
```

В заключение, рассмотрим пример работы с многомерными массивами. Необходимо найти минимальный элемент и запомнить его расположение (номер столбца и строки):

```
int Matr[N][M];
int iMin = Matr[0][0]; //минимальный элемент
int iCol = 0; //столбец
int iRow = 0; //строка
for(int i = 0; i < N; i++)
```

```
for(int j = 0; j < M; j++)
{
     if(iMin > Matr[i][j])
          {    iMin = Matr[i][j];
               //найти мин элемент и запомнить его место
               iCol = i;
               iRow = j;
          }
}
```

Структуры, объединения, битовые поля

Структуры

Структура – это составной объект (пользовательский тип данных), в который входят элементы любых типов, логически связанных между собой. Структуры впервые появились в С, и С++ включает и расширяет понятие структуры в С. В С++ структуры функционально заменяемы классами.

В отличие от массива, который является однородным объектом, структура может быть неоднородной. Элементы структуры не обязательно сохраняются в последовательных байтах памяти. Тип структуры определяется записью вида:

```
struct { список определений } идентификатор ;
```

В структуре обязательно должен быть указан хотя бы один компонент. Определение структур имеет следующий вид:

```
тип-данных описатель;
```

где `тип-данных` указывает тип структуры для объектов, определяемых в описателях. В простейшей форме описатели представляют собой идентификаторы или массивы.

Пример

```
struct
{
     double x,y;
} s1, s2, sm[9];
struct
{
     int year;
     char moth, day;
}
date1, date2;
```

Переменные `s1, s2` определяются как структуры, каждая из которых состоит из двух компонент x и y. Переменная sm определяется как массив из девяти структур. Каждая из двух переменных `date1, date2` состоит из трех компонентов `year, moth, day`. Если объявление структуры не содержит ее имени, то переменные структурного типа могут быть объявлены только в описании структуры, а не с помощью их отдельного объявления.

Существует и другой способ ассоциирования имени с типом структуры, он основан на использовании имени структуры:

```
struct имя_структуры { список описаний; };
```

где `имя_структуры` является идентификатором.

В приведенном ниже примере идентификатор student описывается как имя структуры:

```
struct student
{
     char *name;
     int id, age;
     char prp;
};
```

Имя используется для последующего объявления структур данного вида в форме:

```
struct имя_структуры список-идентификаторов;
```

Пример

```
struct student st1,st2; //структурированные переменные
student arr_stud [10]; //массив структур
student *p_stud; //указатель на структуру
```

В C++ во всех случаях использования структурированной переменной, кроме самого определения структуры, служебное слово `struct` можно опускать, то есть использовать только имя структурированной переменной.

Использование имен структуры необходимо для описания рекурсивных структур. Ниже рассматривается использование рекурсивных структур:

```
struct node
{
     int data;
     node * next;
} st1;
```

Структура `node` действительно является рекурсивной, так как она используется в своем собственном описании, т.е. в формализации указателя `next`. Структуры не могут быть прямо рекурсивными, т.е. структура `node` не может содержать компоненту, являющуюся структурой `node`, но любая структура может иметь компоненту, являющуюся указателем на свой тип, как и сделано в приведенном примере.

Инициализация структур

Структуры могут быть инициализированы списками значений элементов, заключенных в фигурные скобки и перечисленных через запятую, как и массивы:

```
struct student st1 = {
               "Петров", //начальное значение name
               1, //начальное значение id
               20, //начальное значение age
               'M' //начальное значение prp
};
```

Доступ к компонентам структуры

Доступ к компонентам структуры осуществляется при помощи составных имен двумя способами. Во-первых, с использованием операции принадлежности (.) в виде:

```
идентификатор_структуры.идентификатор_поля
(*указатель_структуры).идентификатор_поля
```

Пример

```
st1.name = "Иванов";
st2.id = st1.id;
st1_node.data = st1.age;
```

Второй способ – при помощи операции косвенной адресации – > (стрелка, минус-больше), которая понимается как выделение элемента в структурированной переменной, адресуемой указателем. Она также называется операцией непрямого доступа к члену:

```
указатель_структуры->идентификатор_поля
(&идентификатор_структуры)->идентификатор_поля
```

Операндами здесь являются указатель на структуру и элемент структуры. Операция имеет полный аналог в виде сочетания операций `*` и .

```
struct student *pA, A;
pA = &A;
pA->age //эквивалентно (*pA).age
```

Работа со структурами поддерживается средствами Intellisense (технология автодополнения Microsoft, наиболее известная в Microsoft Visual Studio. Дописывает название функции при вводе начальных букв. Кроме прямого назначения, IntelliSense используется для доступа к документации и для устранения неоднозначности в именах переменных, функций и методов, используя рефлексию).

По мере ввода операции выбора члена структуры вслед за именем структурированной переменной редактор показывает окно со списком всех членов данной структуры.

Объекты типа структур можно передавать как параметры функции и возвращать из функции в качестве результата. Передача структур (особенно больших структур) вызовом по ссылке является более эффективной, чем передача структур вызовом по значению, при которой необходимо копировать всю структуру.

Допустимыми встроенными операциями, которые могут быть выполнены со структурами, являются только следующие: операция присваивания одной структуры другой структуре того же типа; операция адреса (`&`) структуры; операция доступа к элементам структуры и операция sizeof для определения размера структуры. Остальные операции, такие как сравнение (`==` и `!=`), не определены. Однако пользователь может определить большинство операции для работы со структурами.

Два структурных типа являются различными, даже когда они имеют одни и те же члены и отличны от основных типов.

Пример

```
struct s1 { int a; };
struct s2 { int a; };
s1 x;
s2 y = x; //ошибка: несоответствие типов
int i = x; //ошибка: несоответствие типов
```

Вложенные структуры

Структуры могут быть вложенными, т.е. поле структуры может быть связующим полем с внутренней структурой, описание которой должно предшествовать по отношению к основной структуре:

```
struct date
{
     int day, month, year;
};
struct student
{
     char *name;
     date birth_day;
} ;
student A, *pA;
........................
A.birth_day.day = 12; //обращение к полям структуры
pA-> birth_day.day = 20;
struct puple
{
     char *name;
     date *birth_day;
} ;
puple B, *pB;
........................
B.birth_day->day = 24; //обращение к полям структуры
pB-> birth_day->day = 10;
```

В заключение необходимо сказать, что понятие структуры в C++ выходит далеко за пределы оригинальной концепции языка С – оно включено в объектно-ориентированное понятие класса. Ключевые слова `struct` и `class` почти идентичны в C++, за исключением управления доступом к членам.

Объединения (смеси)

Объединение – это поименованная совокупность данных разных типов, размещаемых с учетом выравнивания в одной и той же области памяти, размер которой достаточен для хранения наибольшего элемента. Объединение подобно структуре, однако в каждый момент времени может использоваться только один из элементов объединения. Тип объединения может задаваться в следующем виде:

```
union идентификатор_объединения {
     описание элемента 1;
     ...
     описание элемента n; };
```

Главной особенностью объединения является то, что для каждого из объявленных элементов выделяется одна и та же область памяти, т.е. они перекрываются. Хотя доступ к этой области памяти возможен с использованием любого из элементов, элемент для этой цели должен выбираться так, чтобы полученный результат не был бессмысленным.

В разные отрезки времени выполнения программы некоторые объекты могут быть не нужны, т.е. программе требуется только часть ее объектов. Вместо того, чтобы впустую растрачивать память на объекты, которые используются не постоянно, можно поместить их в объединение, где они будут делить между собой одну и ту же область памяти.

Память, которая соответствует переменной типа объединения, определяется величиной, необходимой для размещения наиболее длинного элемента объединения. Когда используется элемент меньшей длины, то переменная типа объединения может содержать неиспользуемую память.

Пример

```
union {
     char name[30];
     char adress[80];
     int age;
     int telephon;
} information;
union {
     int ax;
     char al[2];
} A;
```

При использовании объекта `information` типа `union` можно обрабатывать только тот элемент, который получил значение, т.е. после присвоения значения элементу `information.name` не имеет смысла обращаться к другим элементам. Объединение `A` позволяет получить отдельный доступ к младшему `A.al[0]` и к старшему `A.al[1]` байтамдвухбайтного числа `A.ax`.

Типичной ошибкой является инициализация объединения при его объявлении значением или выражением, тип которого отличается от типа первого элемента объединения.

Объектам с типом объединения можно присваивать любой класс памяти, кроме `register`. Ни один из членов объединения не может быть объявлен со спецификатором класса памяти `static`.

Единственными допустимыми встроенными операциями, которые могут выполняться над объединениями, являются: операция присваивания значения одного объединения другому объединению того же типа; операция вычисления адреса объединения (`&`) и доступ к элементу объединения при помощи операций доступа к элементу структуры (. и `- >`).

Над объединениями не могут выполняться операции сравнения по темже самым причинам, по каким они не выполняются над структурами.

Битовые поля

Элементом структуры может быть битовое поле, обеспечивающее доступ к отдельным битам памяти. Битовые поля позволяют рационально использовать память с помощью хранения данных в минимально требуемом количестве битов. Элементы битового поля должны быть объявлены как тип `int` или `unsigned`.

Вне структур битовые поля объявлять нельзя. Нельзя также организовывать массивы битовых полей и нельзя применять к полям операцию определения адреса. В общем случае тип структуры с битовым полем задается в следующем виде:

```
struct {
     unsigned идентификатор 1 : длина-поля 1;
```

```
unsigned идентификатор 2 : длина-поля 2;
}
```

Длина поля задается целым выражением или константой. Эта константа определяет число битов, отведенное соответствующему полю. Битовое поле рассматривается как целое число, максимальное значение которого определяется длиной поля. Например:

```
struct number {
     unsigned group: 4; //4 бита от 0 до 15
     unsigned department: 3; //3 бита от 0 до 8
     unsigned course: 3; //3 бита от 0 до 8
}
```

Это описание включает три битовых поля типа `unsigned`: `group`, `department` и `course`, используемых для представления номера зачетки.

При объявлении битового поля вслед за указанием типа элемента `unsigned` или `int` ставится двоеточие (:) и пишется целочисленная константа, задающая ширину поля (т.е. число битов, в которых хранится этот член структуры). Приведенное выше описание структуры показывает, что для хранения члена `group` выделено 4 бита, для `department` – 3 бита и для `course` – 3 бита. Количество битов определяется ожидаемым диапазоном значений для каждого члена структуры. Член структуры `group` хранит значения от 0 до 12 в области памяти размером 4 бита (4 бита, выделенные для элемента `group`, могут хранить значения от 0 до 15). Член структуры `department` может хранить значения от 0 до 8 (факультеты). Область памяти размером 3 бита, выделенная для члена `course`, будет хранить значения от 0 до 4 (диапазон от 0 до 8).

Сгенерируем студентам номера зачеток:

```
number Idip [90];
........................
void FillNumber( number * const doc )
{
     for ( int i = 0; i <= 99; i++)
     {
          doc[i].group = i % 3 + 9;
          doc[i].department = 5;
          doc[i].course = 1;
     }
}
```

Допустимы неименованные поля; они не влияют на смысл именованных полей, но могут улучшить размещение (рисунок 48):

```
struct {
     unsigned a1 : 4;
     unsigned : 2; //неиспользуемое
     unsigned a3 : 5;
     unsigned a4 : 2;
} prim1;
```

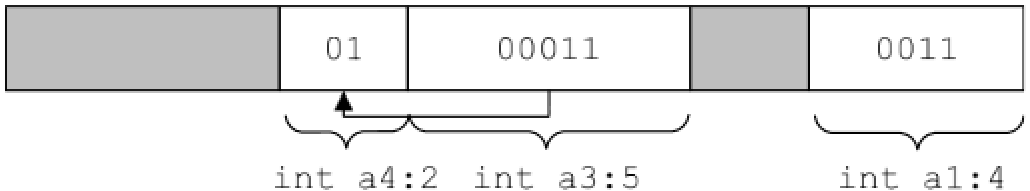


Рисунок 48 – Битовые поля

Поле нулевой длины обозначает выравнивание на границу следующего слова:

```
struct {
     unsigned b1 : 1;
     unsigned : 0;
     unsigned b3 : 5;
     unsigned b4 : 2;
} prim2;
```

Хотя битовые поля сокращают требования к памяти, их использование может привести к тому, что компилятор будет генерировать машинный код, который выполняется с низкой скоростью. Это происходит вследствие того, что приходится использовать дополнительные операции машинного языка для получения доступа к отдельным частям адресуемых элементов памяти, и является одним из множества примеров необходимости компромисса между требованиями эффективности по памяти и по времени выполнения программы.

Перечисления

Объект перечислимого типа представляет собой объект, значения которого выбираются из фиксированного множества идентификаторов, называемых константами перечислимого типа.

Пример

```
enum languages{ C, Java, Ada, PHP, Basic } ;
languages master;
```

или

```
enum languages{ C, Java, Ada, PHP, Basic } master;
```

определяет новый перечислимый тип languages с пятью целыми константами, называемыми перечислителями, и присваивает им значения.

Значения перечислителей по умолчанию присваиваются начиная с 0 в порядке возрастания, т.е. это эквивалентно записи:

```
const C = 0;
const Java = 1;
const Ada = 2;
const PHP = 3;
```

Перечисление может быть не именованным:

```
enum key { A, B, C };
```

Имя перечисления становится синонимом int, а не новым типом.

Пример

```
enum key { A, B, C };
........................
key new;
switch (new) {
```

```
        case A:
        ... break;
        case B:
        ... break;
}
```

Задавать значения перечислителей можно явно:

```
enum languages{
        C = 1,
        Java,
        Ada = 9,
        PHP,
        Basic = 9;
} ;
```

Тем константам, значения которых явно не заданы, присваивается значение предшествующей константы, увеличенное на единицу. Таким образом, константе Java соответствует значение 2, а константе PHP – значение 10. Разным константам перечислимого типа может соответствовать одно и то же значение (Basic , Ada).

Объекты перечислимого типа, которым присвоены значения констант перечислимого типа, можно использовать в любом выражении вместо целых констант. Вместо них подставляются соответствующие им целые значения. Им можно присваивать любой класс хранения, кроме `register`.

Практическое назначение перечисления – определение множества различающихся символических констант целого типа.

Лекция 9 Процедуры и функции

Функции

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

Одно из главных преимуществ, предоставляемых функцией, состоит в том, что она может быть выполнена столько раз, сколько необходимо, в различных точках программы. Без такой возможности программы были бы намного больше, поскольку один и тот же код повторялся бы много раз. Необходимость в функциях вызвана также тем, что для независимой разработки и тестирования программу можно разбивать на фрагменты.

Каждая функция должна иметь имя, которое используется для ее объявления, определения и вызова. В любой программе C/C++ должна быть функция с именем main (главная функция), именно с этой функции, в каком бы месте программы она не находилась, начинается выполнение программы.

С использованием функций связаны три понятия – определение функции (описание действий, выполняемых функцией), объявление функции (задание формы обращения к функции) и вызов функции.

Определение функции имеет следующую форму:

```
[спецификатор-класса-памяти] [спецификатор-типа]
имя-функции ([список-формальных-параметров])
```

```
{
     тело-функции
}
```

Необязательный спецификатор-класса-памяти задает класс памяти функции, который может быть `static` или `extern`.

Спецификатор - типа функции задает тип возвращаемого значения. Если он не задан, то предполагается, что функция возвращает значение типа `int`.

Имя-функции – либо `main` для основной функции, либо произвольный идентификатор, не совпадающий со служебными словами и именами других объектов программы.

Список-формальных-параметров – это последовательность объявлений формальных параметров вида `<обозначение_типа> <имя_параметра>`, разделенная запятыми. Формальные параметры – это переменные, используемые внутри тела функции и получающие значение при вызове функции путем копирования в них значений.

Если функция не использует параметров, то наличие круглых скобок обязательно, а вместо списка параметров надо или указать слово `void`, или ничего не указывать.

Тело-функции – это часть определения функции, заключенная в фигурные скобки { }, содержащая операторы, определяющие действие функции.

Для передачи результата из функции в вызывающую функцию используется оператор `return`. Он может использоваться в двух формах: `return;` – завершает функцию, не возвращающую никакого значения (т.е. перед именем функции указан тип `void`) и `return <выражение>;` – возвращает значение выражения, выражение должно иметь тип, указанный перед именем функции. Если программист не пишет оператор `return` явно, то компилятор автоматически дописывает `return` в конец тела функции перед закрывающей фигурной скобкой.

Функция не может возвращать массив или функцию, но может возвращать указатель на любой тип, в том числе и на массив, и на функцию. Тип возвращаемого значения, задаваемый в определении функции, должен соответствовать типу в объявлении этой функции. Пример:

```
int rus (unsigned char r)
{
     if (r >= 'A' && r <= ' ')
               return 1;
          else
               return 0;
}
```

Все переменные, объявленные в теле функции без указания класса памяти, имеют класс памяти `auto`, т.е. они являются локальными. При вызове функции локальным переменным отводится память в стеке и производится их инициализация. Управление передается первому оператору тела функции, и начинается выполнение функции, которое продолжается до тех пор, пока

не встретится оператор `return` или последний оператор тела функции. Управление при этом возвращается в точку, следующую за точкой вызова, а локальные переменные становятся недоступными. При новом вызове функции для локальных переменных память распределяется вновь, и поэтому старые значения локальных переменных теряются.

Параметры функции передаются по значению и могут рассматриваться как локальные переменные, для которых выделяется память при вызове функции и производится инициализация значениями фактических параметров. При выходе из функции значения этих переменных теряются.

Определения используемых функций могут следовать за определением функции `main`, перед ним, или находиться в другом файле. Для того, чтобы компилятор мог осуществить проверку соответствия типов передаваемых фактических параметров типам формальных параметров до вызова функции, нужно поместить объявление (прототип) функции.

Объявление функций

Если требуется вызвать функцию до ее определения в рассматриваемом файле или определение функции находится в другом исходном файле, то вызов функции следует предварять объявлением этой функции. Объявление (прототип) функции имеет следующий формат:

```
[спецификатор-класса-памяти] [спецификатор-типа]
имя-функции ([список-формальных-параметров])
[список-имен-функций];
```

В отличие от определения функции, в прототипе за заголовком сразу же следует точка с запятой, а тело функции отсутствует. Если несколько разных функций возвращают значения одинакового типа и имеют одинаковые списки формальных параметров, то эти функции можно объявить в одном прототипе, указав имя одной из функций в качестве имени-функции. Правила использования остальных элементов формата такие же, как при определении функции. Имена формальных параметров при объявлении функции можно не указывать, а если они указаны, то их область действия распространяется только до конца объявления.

Тип возвращаемого значения при объявлении должен соответствовать типу возвращаемого значения в определении функции. Если прототип функции не задан, а встретился вызов функции, то строится неявный прототип из анализа формы вызова функции.

Пример прототипов:

```
int rus (unsigned char r);
int rus (unsigned char);
```

Синтаксической ошибкой является случай, если прототип и определение функции не согласуются.

Вызов функций

Вызов функции имеет следующий формат:

```
адресное-выражение ([список-выражений]);
```

Поскольку синтаксически имя функции является адресом начала тела функции, в качестве обращения к функции может быть использовано

адресное выражение (в том числе и имя функции или разадресация указателя на функцию), имеющее значение адреса функции.

Список-выражений представляет собой список фактических параметров, передаваемых в функцию. Этот список может быть и пустым, но наличие круглых скобок обязательно.

Адресное выражение, стоящее перед скобками, определяет адрес вызываемой функции. Это значит, что функция может быть вызвана через указатель на функцию. Пример:

```
int (*fun)(int x, int *y);
```

Здесь объявлена переменная fun как указатель на функцию с двумя параметрами: типа int и указателем на int. Сама функция должна возвращать значение типа int. Круглые скобки, содержащие имя указателя `fun` и признак указателя *, обязательны, иначе запись

```
int *fun (intx,int *y);
```

будет интерпретироваться как объявление функции `fun`, возвращающей указатель на `int`.

Вызов функции возможен только после инициализации значения указателя `fun` и имеет вид:

```
(*fun)(i,&j);
```

Любая функция в программе может быть вызвана рекурсивно, т.е. она может вызывать саму себя. Компилятор допускает любое число рекурсивных вызовов. Пример рекурсии – это математическое определение факториала n!:

```
long fakt(int n)
{
     return ( (n == 1) ? 1 : n * fakt(n-1) );
}
```

Параметры функции по умолчанию

Часто в самом общем случае функции требуется больше параметров, чем в самом простом и более распространенном случае. Если функция объявляется:

```
extern char* func(long, int = 0);
```

то инициализатор второго параметра является параметром по умолчанию. То есть, если в вызове дан только один параметр, в качестве второго используется параметр по умолчанию.

Пример

```
cout << func(31) << func(32,3);
//интерпретируется как cout << func(31,0) << func(32,3);
```

Параметр по умолчанию проходит проверку типа во время описания функции и вычисляется во время ее вызова. Задавать параметр по умолчанию возможно только для последних параметров.

Аргументы по умолчанию должны быть указаны при первом упоминании имени функции – обычно в прототипе. Значения по умолчанию могут быть константами, глобальными переменными или вызовами функций.

Перегрузка имен функций

В C++ возможно определение нескольких функций с одинаковым именем, но с разными типами формальных параметров и результата.

При этом транслятор выбирает соответствующую функцию по типу аргументов. Например:

```
int max (int arr[], int size); //прототипы перегруженных
//функций
long max (long arr[], int size);
double max (double arr[], int size);
double max (long arr[], int size);
//ошибка, недопустимая перегрузка
```

Приведенные функции разделяют одно общее имя, но разные списки параметров. Перегруженные функции можно различать по наличию параметров разного типа либо по различному количеству параметров. Функции не могут быть перегружены, если описание их параметров отличается только модификатором `const` или наличием ссылки. Разные типы возвращаемых значений не могут адекватно различать функции. Поэтому функция `double max(long arr[], int size);` неотличима от `long max (long arr[],int size);` функции.

Фактически все функции (не только перегруженные) должны иметь уникальные сигнатуры, определяемые именем и списком параметров, иначе программа не компилируется.

Назначение перегрузки – разрешить выполнять одно и то же действие с разными операндами. Если имеется серия функций, по сути выполняющих одно и то же, то их необходимо перегружать.

Передача параметров функции `main`

Функция `main` может быть определена с параметрами, которые передаются из внешнего окружения, например, из командной строки.

Во внешнем окружении все данные представляются в виде строк символов. Для передачи этих строк в функцию `main` используются два параметра, первый параметр служит для передачи числа передаваемых строк, второй – для передачи самих строк. Общепринятые (но не обязательные) имена этих параметров `argc` и `argv`. Параметр `argc` имеет тип `int`, его значение формируется из анализа командной строки и равно количеству слов в командной строке, включая и имя вызываемой программы (под словом понимается любой текст, не содержащий символа пробел). Параметр `argv` – это массив указателей на строки, каждая из которых содержит одно слово из командной строки. Если слово должно содержать символ пробел, то при записи его в командную строку оно должно быть заключено в кавычки.

Функция `main` может иметь и третий параметр, который принято называть `argp`, и который служит для передачи в функцию `main` параметров операционной системы (среды), в которой выполняется С-программа. Заголовок функции `main` имеет вид:

```
int main (int argc, char *argv[], char *argp[])
```

Если командная строка имеет вид:

```
A:\>cprog working 'C program' 1
```

то аргументы `argc, argv, argp` представляются в памяти следующим образом:

```
argc  [ 4  ]
argv  [    ]--> [    ]-->   [A:\cprog.exe\0]
                [    ]-->   [working\0]
                [    ]-->   [C program\0]
                [    ]-->   [1\0]
                [NULL]
argp  [    ]--> [    ]-->   [path=A:\;C:\\0]
                [    ]-->   [lib=D:\LIB\0]
                [    ]-->   [include=D:\INCLUDE\0]
                [    ]-->   [conspec=C:\COMMAND.COM\]
                [NULL]
```

Возврат результата

Функция в качестве результата может возвращать указатель.

В этом случае она обычно «выбирает» некоторую переменную из имеющихся или же динамически «создает» ее:

```
int *min(int A[], int n)
//функция возвращает указатель
//на минимальный элемент массива
{
     int *pmin, i;
     //рабочий указатель, содержащий результат
     for (i = 1, pmin = A; i < n; i++)
          if (A[i] < *pmin)
               pmin = &A[i];
     return pmin;
}
```

Указатель – результат функции, может ссылаться не только на отдельную переменную, но и на массив.

Существует правило относительно возвращаемых адресов: никогда не возвращать из функции адрес локальной автоматической переменной.

Ссылки как формальные параметры и результат функции

Передача параметров функции в виде ссылки изменяет метод передачи. Передаваемый параметр выступает псевдонимом передаваемого фактического параметра (аргумента). Это исключает всякое копирование и обеспечивает функции прямой доступ к аргументу. Тогда, нет необходимости в разыменовании указателей на значения, как в предыдущем примере. Например:

```
int incr (int& n)
{
     n++;
     return n;
}
............................
int s = 5;
cout << incr (s); //вызов функции incr
```

Здесь ссылка как параметр функции будет создаваться и инициализироваться при каждом вызове функции и разрушатся по ее завершении, т.е. каждый раз получается новая ссылка. Функция `incr` непосредственно модифицирует переменную, переданную ей в аргументе. Если в качестве аргумента передать числовое значение (например 10), то

компилятор выдаст сообщение об ошибке, так как изменять значения констант нельзя.

Использование модификатора `const`

Чтобы сообщить компилятору, что модификации параметров функции не будет, можно применить модификатор const. Тогда компилятор будет проверять код функции на предмет изменения значения аргумента. Изменим предыдущий пример:

```
int incr (const int& n) //функция с константным
//аргументом-ссылкой
{
     // n++; //ошибка, изменять нельзя
     return n + 1; //возвратить увеличенное значение
}
.............................
int s = 5;
const int num = 10;
cout << incr (s) << incr (num);
//вызовы функции sum
```

В данной функции имеется прямой доступ к исходному аргументу вызывающего кода (позволяет избежать копирования), и получается полная защита от непреднамеренной модификации (модификатор `const`)

Возврат ссылки

Функция может возвращать ссылку. Так как ссылка не является отдельной сущностью (это псевдоним чего-то другого), важно, что объект, на который она ссылается, все еще существует после завершения выполнения функции. Ссылки как возвращаемые типы особенно важны в контексте объектно-ориентированного программирования.

Рассмотрим пример. В массиве, содержащем смешанный набор значений, при вставке нового значения заменяется минимальный элемент.

Несмотря на то, что возврат значения выглядит как значение, тип объявлен как ссылка, и поэтому возвращается не значение `a[j]`, а ссылка на него. Будет ошибкой, если указать в качестве возвращаемого значения `&a[j]`. Это будет означать адрес `a[j]` элемента, то есть указатель.

Никогда не возвращайте из функции ссылку на локальную переменную

```
double& low_val (double val[], int size);
//прототип функции, возвращающей ссылку
...........................
double arr[] = { 3.1, 10.0, 15.9, 23.0, 17.2};
int arr_size = sizeof arr/ sizeof arr[0];
//количество элементов массива
...........................
low_val(arr, arr_size) = 1.0;
//изменить минимальное значение на 1.0
low_val(arr, arr_size) = 3.9;
//изменить минимальное значение на 3.9
.............................
double& low_val (double val[], int size)
{
     int j = 0; //индекс наименьшего
```

```
    for (int i = 1; i < size; i++)
        if (a[j] > a[i]) //поиск минимального элемента
        j = i; //запомнить его индекс
    return a[j];
    //вернуть ссылку на минимальный элемент
}
```

Статические переменные в функциях

Чтобы создать переменную, чье значение сохраняется от одного вызова функции до другого, можно объявить ее внутри функции с ключевым словом `static`.

```
static int count = 0;
```

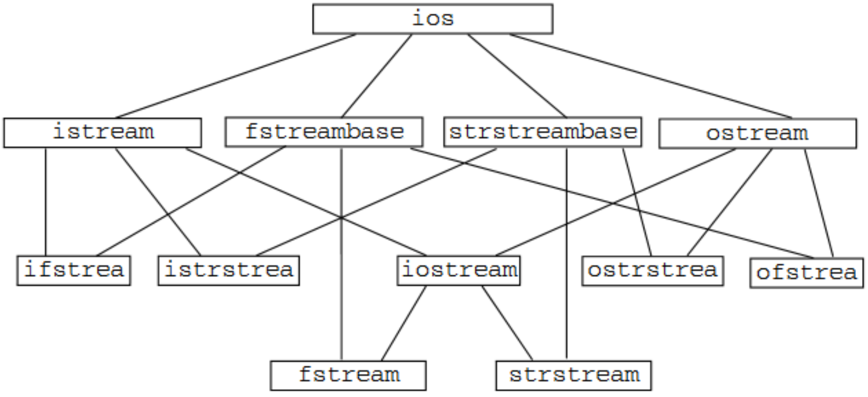
Инициализация статической переменной в функции происходит только при первом вызове функции. Затем она продолжает существовать на протяжении всего времени выполнения программы, и любое значение, которое она имеет при завершении функции, остается доступным при следующем вызове.

Лекция 10 Дополнительные возможности

Файловый ввод-вывод в С++

Для обработки файлов в С++ должны быть включены заголовочные файлы `<iostream>` и `<fstream>`. Файл `<fstream>` включает определения классов потоков `ifstream` (для ввода в файла), ofstream (для вывода из файл) и fstream (для ввода-вывода файлов). Файлы открываются путем создания объектов этих классов потоков. Эти классы потоков являются производными (т.е. наследуют функциональные возможности) от классов istream, `ostream` и `iostream` соответственно.

Таким образом, функции-члены, операции и манипуляторы для работы с потоками, описанные в главе 4, могут быть также применены и к потокам файлов. Отношения наследования классов ввода-вывода представлены на рисунке 49.



Отношения наследования классов ввода-вывода

Рисунок 49 - Отношения наследования классов ввода-вывода

Здесь `istream` – класс входных потоков; ostream – класс выходных потоков; `iostream` – класс ввода-вывода; `istrstream` – класс входных строковых потоков; ifstream – класс входных файловых потоков и т.д.

При использовании поточных классов языка C++ в программе требуется использовать стандартное пространство имен (`using namespace::std`).

Файловый ввод-вывод организован с помощью переопределенных в поточных классах операций включения (<<) и извлечения (>>). Операции << и >> имеют два операнда. Левым операндом является объект класса `istream` (`ostream`), а правым – данное.

Каждый поток имеет связанное с ним состояние. Состояния потока описываются в классе `ios` в виде перечисления enum:

```
enum io_state {
     goodbit, //нет ошибки 0X00
     eofbit, //конец файла 0X01
     failbit, //последняя операция не выполнилась 0X02
     badbit, //попытка использования недопустимой
     //операции 0X04
     hardfail //фатальная ошибка 0X08
};
```

Флаги, определяющие результат последней операции с объектом `ios`, содержатся в переменной state. Получить значение этой переменной можно с помощью функции `int rdstate()`.

Кроме того, проверить состояние потока можно следующими функциями:

```
int bad(); //1, если badbit или hardfail
int eof(); //1, если eofbit
int fail(); //1, если failbit, badbit или hardfail
int good(); //1, если goodbit
```

Работа с классом `fstream`

Члены этого класса позволяют открывать файл, записывать в него данные, перемещать указатель позиционирования, читать данные.

Объекты классов файлового ввода-вывода разрешено определять и без указания имени файла. Позже к этому объекту можно присоединить файл с помощью функции открытия файла со следующей сигнатурой:

```
void open(const char* name, int mode, int p = filebuf::openprot);
```

Пример

```
fstream inoutFile;
inoutFile.open("a.txt");
if ( ! inoutFile ) //открытие успешно
```

При открытии файла задается режим открытия файла (mode) (таблице 5), он определен в базовом классе ios, поэтому обращение к значениям в классе `fstream` должно идти с указанием класса родителя.

Таблица 5 – Режимы открытия файла

Режим	Описание
`ios_base::app`	Открыть файл для дозаписи в его конец
`ios_base::binary`	Открыть файл в бинарном режиме

`ios_base::in`	Открыть файл для чтения
`ios_base::out`	Открыть файл для записи, если файл не существует, он будет создан
`ios_base::trunc`	Уничтожить содержимое файла, если файл существует (очистить файл)
`ios_base::ate`	Установить указатель позиционирования файла на его конец

Для задания нескольких режимов используется оператор побитового ИЛИ.

Для сброса буфера потока, отсоединения потока от файла и закрытия файла используется функция:

```
void fstreambase::close();
```

Эту функцию необходимо явно вызвать при изменении режима работы с потоком. Автоматически она вызывается только при завершении программы.

Объект класса `fstream` можно позиционировать с помощью функций-членов `seekg()` или `seekp()`. Здесь буква g обозначает позиционирование для чтения (`getting`) символов (используется с объектом класса `ofstream`), а `p` – для записи (putting) символов (используется с объектом класса `ifstream`). Эти функции делают текущим тот байт в файле, который имеет указанное абсолютное или относительное смещение.

У функций есть два варианта. В первом варианте текущая позиция устанавливается в некоторое абсолютное значение, заданное аргументом `current_position`, причем значение 0 соответствует началу файла:

```
seekg( pos_type current_position);
//смещение от текущей позиции в том или ином направлении
```

Второй вариант устанавливает указатель рабочей позиции файла на заданное расстояние от текущей, от начала файла или от его конца в зависимости от аргумента `dir`, который может принимать следующие значения: `ios_base::beg` – от начала файла; `ios_base::cur` – от текущей позиции; `ios_base::end` – от конца файла:

```
seekg(off_type offset_position, ios_base::seekdir dir);
```

Текущая позиция чтения в файле типа `fstream` возвращается любой из двух функций-членов `tellg()` или `tellp()`. Рассмотрим пример, в котором записывается строка текста в файл, затем она читается из файла в буфер:

```
#include <fstream>
#include <iostream>
........................
int _tmain(int argc, _TCHAR* argv[])
{
     using namespace std;
     char p[100];
     fstream inout;
     inout.open("a.txt",ios:: base::in|ios:: base::out
     |ios:: base::trunc);
     //создание двунаправленного потока
     inout << "This is string" << endl; //вывод в файл
```

```
        inout.seekg (0); //установка позиционера на начало
        inout.getline (p,50); //чтение строки из файла
        inout.seekg (0); //установка позиционера на начало файла
        cout << endl << inout.rdbuf();
        //вывод содержимого потока на экран
        inout.close(); //закрыть поток
    }
```

Работа с классом `ofstream`

Если файл будет использоваться только для вывода, определяется объект класса `ofstream`. Например:

```
ofstream outfile("copy.out", ios::base::out );
ofstream outfile2("copy.out"); //эти объекты эквивалентны
```

Класс `ofstream` является производным от `ostream`.

Поэтому все определенные в ostream операции применимы и к `ofstream`.

В классе `ostream` определены следующие функции:

```
ostream& put(char C);
```

альтернативный метод вывода символа C в выходной поток `ostream`. Функция-член `write()` класса `ostream` дает альтернативный метод вывода массива символов. Вместо того, чтобы выводить символы до завершающего нуля, она выводит указанное число символов, включая и внутренние нули, если таковые имеются. Сигнатура функции:

```
ostream& write(const char* buffer,int size);
```

записывает в `ostream` содержимое буфера `buffer` количеством `size` символов. Буфер записывается без форматирования.

Пример использования класса `ofstream` :

```
#include <fstream>
........................
int _tmain(int argc, _TCHAR* argv[])
{
     ofstream out; //объявляем переменную типа ofstream
     ........................
     out.open ("a.txt"); //вызываем метод открытия файла
     if ( out == NULL) return 0; //выход, если не открыли файл
     for (int i = 0; i < 2; i++)
          out << "string" << i << endl; //вывод в файл
     out.close(); //закрытие файла
........................
}
```

Работа с классом `ifstream`

Чтобы открыть файл только для чтения, применяется объект класса `ifstream`, производного от `istream`.

Иногда необходимо прочитать из входного потока последовательность не интерпретируемых байтов, а типов данных, таких как `char, int, string` и т.д. Функция-член `get()` класса `istream` читает по одному байту, а функция `getline()` читает строку, завершающуюся либо символом перехода на новую строку, либо каким-то иным символом, определяемым пользователем. Сигнатура `getline()`:

```
getline (char * buffer, streamsize size, char delimiter='\n');
```

Парной для функции `write()` из класса `ostream` является функция `read()` из класса `istream` с сигнатурой:

```
read (char* buffer, streamsize size);
```

читает `size` соседних байт из входного потока и помещает их, начиная с адреса `buffer`. Функция `gcount()` возвращает число байт, прочитанных при последнем обращении к `read()`. В свою очередь `read( )` возвращает объект класса `istream`, для которого она вызвана

```
#include <fstream>
........................
int _tmain(int argc, _TCHAR* argv[])
{
     ifstream in; //объявляем переменную типа ifstream
     char c;
     ........................
     in.open ("c:\\a.txt"); //вызываем метод открытия файла
     while (!in.eof()); //пока не конец файла
     {
          c = in.peek();
          //считываем символ из файла без извлечения
          if (c == 'A') //если символ A
          {
               in.seekg(in.tellg()+1);
               //передвинуть позиционер на один
               //пропустить символ
               continue; //на продолжение цикла
          }
          in.get(c); //чтение символа
          cout << c;
     }
     in.close();
          }
```

2-ый семестр

Раздел 3 СТАНДАРТНЫЕ АЛГОРИТМЫ

Лекция 11 Сортировка и поиск

Сложность алгоритмов

Принципы анализа алгоритмов

Процесс создания компьютерной программы для решения практической задачи состоит из нескольких этапов: формализация и создание технического задания; разработка алгоритма решения задачи; написание, тестирование, отладка и документирование программы; получение решения

исходной задачи путем выполнения законченной программы. В процессе решения прикладных задач выбор подходящего алгоритма вызывает определенные трудности. Иногда необходимо, чтобы алгоритм отвечал противоречащим друг другу требованиям (минимальное время выполнения, минимальная память).

На время выполнения программы влияют следующие факторы: ввод (порядок ввода) исходной информации в программу; качество скомпилированного кода исполняемой программы; машинные инструкции (естественные и ускоряющие), используемые для выполненияипрограммы; временная сложность алгоритма.

Поскольку время выполнения программы зависит от ввода исходных данных, его можно определить как функцию от исходных данных.

Обычно говорят, что время выполнения программы имеет порядок T(n) от входных данных размера n. Например, некая программа имеет время выполнения $T(n) = cn^2$, где c – константа. Для многих программ время выполнения действительно является функцией входных данных, а не их размера. В этой ситуации T(n) определяется как время выполнения в наихудшем случае, т.е. как максимум времени выполнения по всем входным данным размера n. Величина $T_{ср}(n)$) будет рассматриваться как среднее (в статистическом смысле) время выполнения по всем входным данным размера n. Хотя $T_{ср}(n)$ является достаточно объективной мерой времени выполнения, однако часто нельзя предполагать (или обосновать) равнозначность всех входных данных. На практике среднее время выполнения найти сложнее, чем наихудшее время выполнения, так как математически это трудноразрешимая задача и, кроме того, зачастую не имеет простого определения понятие «средних» входных данных. Поэтому в основном используется наихудшее время выполнения как мера временной сложности алгоритмов.

Второй и третий фактор (компилятор и машина, на которой выполняется программа) влияют на то, что для измерения времени выполнения T(n) невозможно применить стандартные единицы измерения, такие как секунды или миллисекунды. Поэтому можно только делать заключения вида «время выполнения такого-то алгоритма пропорционально n^2». Константы пропорциональности также нельзя точно определить, поскольку они зависят от компилятора, компьютера и других факторов.

Асимптотические соотношения

Для описания скорости роста функций используется О-нотация.

О-нотация используется по трем основным причинам: чтобы ограничить ошибку, возникающую при отбрасывании малых слагаемых в математических формулах; чтобы ограничить ошибку, возникающую тогда, когда не учитываются те части программы, которые дают малый вклад в анализируемую сумму; чтобы классифицировать алгоритмы согласно верхней границе их общего времени выполнения.

Например, когда говорят, что время выполнения T(n) некоторой программы имеет порядок $O(n^2)$ (читается «о-большое от n в квадрате» или

«О от n в квадрате»), то подразумевается, что существуют положительные константы c и n_0 такие, что для всех n, больших или равных n_0, выполняется неравенство $T(n) \le cn^2$.

О-нотация используется, прежде всего, для исследования фундаментального асимптотического поведения алгоритма и представляет собой верхнюю асимптотическую оценку трудоемкости алгоритма.

Она позволяет определить, как быстро растет трудоемкость алгоритма с увеличением объема данных.

Определение. Функция T(n) имеет порядок O(f(n)), если существуют константы c и n_0 такие, что для всех $n \ge n_0$ выполняется неравенство $T(n) \le cf(n)$. Для программ, у которых время выполнения имеет порядок O(f(n)), говорят, что они имеют порядок (или степень) роста f(n).

Например, функция $T(n) = 3n^3 + 2n^2$ имеет степень роста $O(n^3)$ или время исполнения задачи растет не быстрее, чем куб количества элементов. Чтобы это показать, надо положить $n_0 = 0$ и $c = 5$, тогда, для всех целых $n \ge 0$ выполняется неравенство $3n^3 + 2n^2 \le 5n^3$.

При оценке трудоемкости используют правило сумм: в общем случае трудоемкость выполнения конечной последовательности программных фрагментов, без учета констант, имеет порядок фрагмента с наибольшим временем выполнения.

Например, пусть есть три фрагмента с временами выполнения соответственно $O(n^2)$, $O(n^3)$ и O(nlogn). Тогда время последовательного выполнения первых двух фрагментов имеет порядок $O(max(n^2, n^3))$,т.е. $O(n^3)$. Время выполнения всех трех фрагментов имеет порядок $O(max(n^3, nlogn))$, это то же самое, что $O(n^3)$.

При оценке трудоемкости используют правило произведений: если T1(n) и T2(n) имеют степени роста O(f(n)) и O(g(n)) соответственно, то произведение T1(n)T2(n) имеет степень роста O(f(n)g(n)).

Из правила произведений следует, что O(cf(n)) эквивалентно O(f(n)), если c – положительная константа. Например, $O(n^2 / 2)$ эквивалентно $O(n^2)$.

В таблице 6 приведены числа, иллюстрирующие скорость роста для нескольких разных функций. Используемый параметр n может быть степенью полинома, размером файла при сортировке или поиске, количеством символов в строке или некоторой другой абстрактной мерой размера рассматриваемой задачи. Однако, чаще всего, он прямо пропорционален величине обрабатываемого набора данных.

Таблица 6 - Скорость роста функций

n	**lgn**	**sqrt(n)**	**nlgn**	**n(lgn)^2**	**n$^{3/2}$**	**n^2**
10	3	3	33	110	32	100
100	7	10	664	4414	1000	10 000
1000	10	32	9966	99 317	31 623	1 000 000
10 000	13	100	132 877	1 765 633	1 000 000	100 000 000
100 000	17	316	1 660 964	27 588 016	31 622 777	10 000 000 000
1 000 000	20	1000	19 931 569	397 267 426	1 000 000 000	1 000 000 000 000

Как видно из таблицы 6 для небольших задач используемый метод практически не влияет на скорость решения задачи. Но по мере роста задачи числа, с которыми имеем дело, становятся огромными. Когда количество исполняемых инструкций в медленном алгоритме становится по-настоящему большим, время, необходимое для их выполнения, становится недостижимым (например, 10^7 секунд равно 3,8 месяца, а 10^8 секунд равно 3,1 года).

Для нижней асимптотической оценки роста функции T(n) используется Ω. Она задает и определяет класс функций, которые растут не медленнее, чем f(n) с точностью до постоянного множителя.

Определение. Функция T(n) имеет порядок Ω (f(n)), если существуют константы c и n_0 такие, что для всех $n \geq n_0$ выполняется неравенство $T(n) \leq cf(n)$.

Например, запись T(n) = Ω (nlogn) обозначает класс функций, которые растут не медленнее, чем f(n) nlogn, в этот класс попадают все полиномы со степенью большей единицы, равно как и все степенные функции с основанием, большим единицы.

Для задания одновременно верхней и нижней оценки роста функции T(n) используется Θ.

Определение. Оценка сложности алгоритма T(n) = Θ(f(n)), если при
g > 0 и n > 0 существуют положительные c_1 , c_2 , n_0 , такие, что
$c_1 f(n) \leq T(n) \leq c_2 f(n)$ при $n > n_0$.

В этом случае говорят еще, что функция f(n) является асимптотически точной оценкой функции T(n), так как по определению функция T(n) не отличается от функции f(n) с точностью до постоянного множителя. Важно понимать, что Θ(f(n)) представляет собой не функцию, а множество функций, описывающих рост T(n) с точностью до постоянного множителя. Равенство T(n) = Θ(f(n)) выполняется тогда и только тогда, когда T(n) = O(f(n)) и
T(n) = Ω (f(n)).

Алгоритмы, которые будут рассмтариваться далее, обычно имеют сложность, пропорциональную одной из следующих функций из таблицы 7

Таблица 7 – Функции сложности

T(n)	**Описание**
1	Большинство инструкций большинства программ запускается один или несколько раз. Тогда, говорят, что время выполнения программы постоянно
log n	Программа становится медленнее с ростом n. Такое время выполнения обычно присуще программам, которые сводят большую задачу к набору меньших задач, уменьшая на каждом шаге размер задачи на некоторый постоянный фактор. Основание логарифма не сильно изменяет константу. При удвоении logn растет на постоянную величину, а удваивается лишь тогда, когда n достигает n^2
n	Когда время выполнения программы является линейным, это значит, что каждый входной элемент подвергается небольшой обработке. Эта ситуация оптимальна для алгоритма, который должен обработать n вводов (или произвести n выводов)
nlogn	Возникает тогда, когда алгоритм решает задачу, разбивая ее на меньшие подзадачи, решая их независимо и затем объединяя решения. Когда n

	удваивается, тогда время выполнения более чем удваивается
n^2	Когда время выполнения алгоритма является квадратичным, он полезен для практического использования для относительно небольших задач. Квадратичное время выполнения обычно появляется в алгоритмах, которые обрабатывают все пары элементов данных (возможно, в цикле двойного уровня вложенности)

Таким образом, асимптотические оценки можно представить в порядке увеличения скорости их роста:

$O(1) < O(\log_2 n) < O(n) < O(n\log_2 n) < O(n^2) < O(n^3) < O(2^n)$.

Классы сложности

В рамках классической теории осуществляется классификация задач по классам сложности (P-сложные; NP-сложные, экспоненциально сложные и др.). К классу P относятся задачи, которые могут быть решены за время, полиномиально зависящее от объема исходных данных, с помощью детерминированной вычислительной машины (например, машины Тьюринга). К классу NP относятся задачи, которые могут быть решены за полиномиально выраженное время с помощью недетерминированной вычислительной машины, то есть машины, следующее состояние которой не всегда однозначно определяется предыдущими.

В частности, к классу NP относятся все задачи, решение которых можно проверить за полиномиальное время. Класс P содержится в классе NP.

Поскольку класс P содержится в классе NP, принадлежность той или иной задачи к классу NP зачастую отражает наше текущее представление о способах решения данной задачи и носит неокончательный характер. В общем случае нет оснований полагать, что для той или иной NP-задачи не может быть найдено P-решение. Вопрос о возможной эквивалентности классов P и NP (то есть о возможности нахождения P-решения для любой NP-задачи) считается многими одним из основных вопросов современной теории сложности алгоритмов.

Все классы сложности находятся в иерархическом отношении: одни включают в себя другие. Однако про большинство включений неизвестно, являются ли они строгими.

Эффективность алгоритмов поиска

В качестве еще одного примера рассмотрим два алгоритма поиска: последовательный и бинарный поиск элемента в массиве.

Последовательный поиск

При последовательном поиске элемента в массиве, имеющем длину n, элементы просматриваются по очереди, начиная с первого, пока не обнаружится искомый либо не будет достигнут конец массива. В наилучшем случае искомым элементом является первый. Для его обнаружения понадобится только одно сравнение. Следовательно, в наилучшем случае сложность алгоритма последовательного поиска равна O(1). В наихудшем случае искомый элемент является последним. Для того, чтобы его найти, понадобится n сравнений. Следовательно, в наихудшем случае сложность алгоритма последовательного поиска равна O(n). В среднем случае искомый

элемент находится в средней ячейке массива и обнаруживается после n / 2 сравнений.

Бинарный поиск

Алгоритм бинарного поиска предназначен для поиска элемента в упорядоченном массиве и основан на повторяющемся делении частей массива пополам. Алгоритм определяет, в какой из двух частей находится элемент, если он действительно хранится в массиве, а затем повторяет процедуру деления пополам. В ходе очередного разбиения массива алгоритм выполняет сравнения. Можно вычислить максимальное количество сравнений, т.е. наихудший вариант. Допустим, что $n = 2^k$, где k – некоторое натуральное число.

Чтобы проверить среднюю ячейку массива, сначала нужно поделить массив пополам. После того, как массив, состоящий из n элементов, поделен пополам, делится пополам одна из его половин. Эти деления продолжаются до тех пор, пока не останется только один элемент. Для этого потребуется выполнить k разбиений массива. Это возможно, поскольку $n/2^k = 1$. В наихудшем случае алгоритм выполнит k разбиений и, следовательно, k сравнений. Поскольку $n = 2^k$, получаем, что $k = \log_2 n$. Если число n не будет степенью двойки, то k – наименьшее число, удовлетворяющее условию
$2^{k-1} < n < 2^k$. (Например, если n равно 30, то k = 5, поскольку
$2^4 = 16 < 30 < 32 < 2^5$).

Таким образом, сложность алгоритма бинарного поиска в наихудшем случае имеет порядок $O(\log_2 n)$ для любого значения n.

Бинарный поиск намного лучше последовательного (при n = 1 000 000, $\log_2 n = 19$, т.е. при последовательном поиске в наихудшем случае будет миллион сравнений, в то время как алгоритм бинарного поиска выполнит не более 20) при условии упорядоченности массива.

Алгоритмы сортировки

Сортировка – процесс упорядочения набора элементов в возрастающем или убывающем порядке.

Алгоритмы сортировки можно классифицировать по нескольким признакам. По размещению элементов сортировки бывают внутренними – в памяти и внешними – в файле. По виду структуры данных, содержащей сортируемые элементы, можно выделить сортировки массивов (строк), массивов указателей, списков, объектов и других структур данных. Если в объектах содержатся несколько данных-членов, нужно указать, какая переменная определяет порядок следования объектов.

Эта переменная-член называется ключом сортировки (key). Например, если объекты хранят информацию о людях, их можно сортировать по имени, возрасту или почтовому индексу. По сохранению относительного порядка размещения элементов с дублированными ключами сортировки бывают устойчивые и неустойчивые. Сортировки делятся на адаптивные (выполнение различных последовательностей операций в зависимости от результата сравнения) и неадаптивные (последовательность операций не

зависит от порядка следования данных). По способу выбора элементов сортировки делятся на сортировки подсчетом, вставками, выбором, слиянием, разделением, обменные сортировки.

Для простоты будем предполагать, что сортировка применяется к числам. Все рассматриваемые алгоритмы ориентируются на возрастающий порядок.

Обменные сортировки

Метод пузырька

Алгоритм сортировки методом пузырька сравнивает между собой соседние элементы и меняет их местами, если они нарушают порядок.

Для этого приходится несколько раз просматривать одни и те же элементы. Во время первого прохода сравниваются два первых элемента массива; если они нарушают порядок, их меняют местами. Затем сравнивается другая пара, т.е. 2-й и 3-й элементы. Если они нарушают порядок, их меняют местами. Просмотр, сравнение и обмен двух элементов выполняется до тех пор, пока не будет достигнут конец массива.

На рисунке 50 представлен пример работы этого метода.

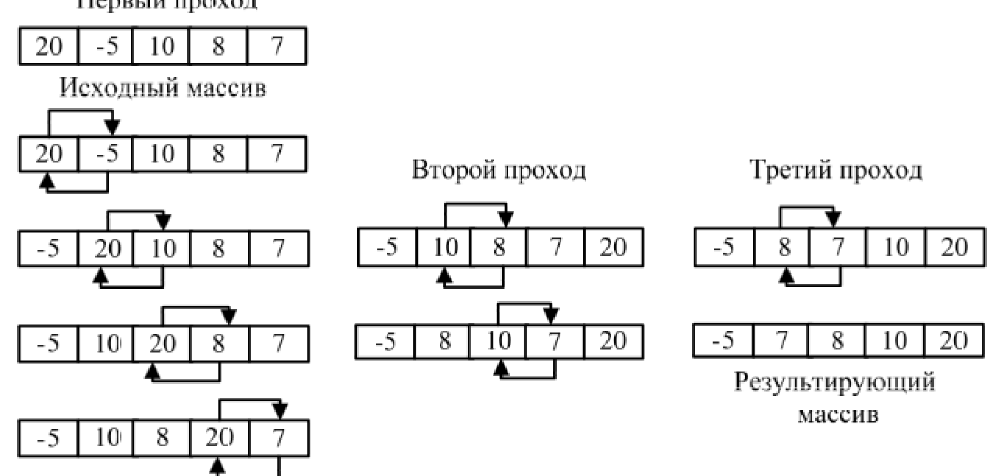


Рисунок 50 – Сортировка методом пузырька

Рассмотрим функцию, выполняющую сортировку массива методом пузырька:

```
void BubbleSort(int arr[],int n)
{    int temp;
     for(int i = n-1; i > 0; --i)
     //для каждого элемента массива
          for(int j= 0; j < i; ++j)
               if(arr[j] > arr[j+1]) //процесс всплытия
               {
                    temp = arr[j];
                    arr[j] = arr[j+1];
                    arr[j+1] = temp;
               }
}
```

Анализ

При первом проходе выполняется не больше, чем (n – l) сравнений и

(n – l) перестановок. При втором проходе выполняется (n – 2) сравнений и не больше (n – 2) перестановок. Следовательно, в худшем случае при сортировке методом пузырька будет выполнено
$(n - 1) + (n - 2) + \ldots + 1 = n (n - 1) / 2 \approx n^2 / 2$ сравнений и столько же перестановок.

При каждой перестановке выполняется три присваивания. Таким образом, общее количество основных операций в худшем случае равно
$2n(n - l) = 2n^2 - 2n$. В худшем случае сложность алгоритма сортировки методом пузырька равна $O(n^2)$. Пузырьковая сортировка не требует дополнительной памяти. Данная реализация алгоритма – устойчивая.

Шейкер-сортировка

Сортировку методом пузырька можно в некоторой степени улучшить и тем самым улучшить ее временные характеристики. Можно, например, заметить, что сортировка обладает одной особенностью: расположенный не на своем месте в конце массива элемент достигает своего места за один проход, а элемент, расположенный в начале массива, очень медленно достигает своего места. Необязательно все просмотры делать в одном направлении. Вместо этого всякий последующий просмотр можно делать в противоположном направлении и фиксировать нижнюю и верхнюю границы неупорядоченной части, т.к. просмотр имеет смысл делать не до конца массива, а до последней перестановки на предыдущем просмотре. В этом случае сильно удаленные от своего места элементы будут быстро перемещаться в соответствующее место.

Ниже показана улучшенная версия сортировки пузырьковым методом, получившая название «челночной сортировки» из-за соответствующего характера движений по массиву или шейкер-сортировки:

```
void ShakerSort(int arr[], int n)
{
      int buf, first=0, mode=1, last = n;
      for (; first<last; mode>0?++first:--last)
           { for(int i= mode>0?first:last;
           mode>0?(i<last):(i>first); mode>0?++i:--i)
                {
                     if ((arr [i]> arr [i+1])&&(mode>0))
                          {
                          buf = arr [i];
                          arr [i]= arr [i+1];
                          arr [i+1]= buf;
                          }
                     if ((m[i]<m[i-1])&&(mode<0))
                          {
                          buf = arr [i];
                          arr [i]= arr [i-1];
                          arr [i-1]= buf;
                          }
                }
           mode=-mode;
      }
```

}

Хотя эта сортировка является улучшением метода пузырька, ее нельзя рекомендовать для использования, поскольку время выполнения
по - прежнему зависит квадратично от числа элементов. Число сравнений не изменяется, а число обменов уменьшается лишь на незначительную величину.

Сортировка выбором

Прямая выборка

Работает по принципу выбора наименьшего элемента из числа неотсортированных. Отыскивается наименьший элемент массива, затем он меняется местами с элементом, стоящим первым в сортируемом массиве. Далее, находится второй наименьший элемент и меняется местами с элементом, стоящим вторым в исходном массиве. Этот процесс продолжается до тех пор, пока весь массив не будет отсортирован. На рисунке представлен пример работы этого метода.

Недостаток сортировки выбором заключается в том, что время ее выполнения лишь в малой степени зависит от того, насколько упорядочен исходный массив. Процесс нахождения минимального элемента за один проход дает очень мало сведений о том, где может находиться минимальный элемент на следующем проходе.

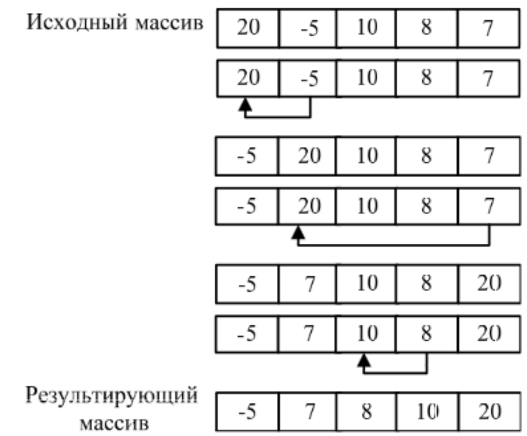


Рисунок 51 - Сортировка выбором

Рассмотрим функцию, выполняющую сортировку массива методом выбора:

```
void SelectSort(int arr[],int n){
     int k,min;
     //индекс минимального элемента, минимальный элемент
     for(int i = 0;i < n;i++)
          {
          k = i; //индекс min
          min = arr[i];
          for(int j = i+1; j < n; j++) //цикл поиска
     минимального
          if(arr[j] < min) //в неотсортированной части
               {
               k = j; //запомнить индекс
               min = arr[j]; //запомнить минимальный
               };
```

```
arr[k] = arr[i]; //поменять местами
arr[i] = min;
}}
```

Анализ

Как следует из описания алгоритма, сортировка сводится к сравнениям, присваиваниям и перестановкам элементов. Общее количество сравнений будет $(n - 1) + (n - 2) + \ldots + 1 = n(n - 1) / 2 \approx n^2 / 2$. Максимальное число перестановок $(n - 1)$ и $4n + n(n - 1)$ операций присваивания. В сумме алгоритм сортировки методом выбора выполняет
$n(n - 1) / 2 + n - 1 + 4n + n(n - 1) = n^2 + 9n / 2 - 1$ основных операций.

Применяя свойства O-нотации, можем отбросить слагаемые с младшими степенями и константные множители – получаем окончательную оценку $O(n^2)$. Хотя алгоритм выполняет $O(n^2)$ сравнений, в ходе сортировки осуществляется только $O(n)$ перестановок. Алгоритм сортировки методом выбора можно применять, когда перестановки представляют собой затратные операции, а сравнения – нет.

Квадратичная выборка

Данный метод по сравнению с прямой выборкой уменьшает число сравнений, но требует дополнительного объема памяти. Сортируемый массив, состоящий из n элементов, разделяется на n групп по n элементов в каждой. Если n не является точным квадратом, то таблица разделяется на n' групп, где n' – ближайший точный квадрат, больший n.

В каждой группе выбирается наименьший элемент, который пересылается во вспомогательный массив. Вспомогательный массив просматривается, и наименьший его элемент пересылается в зону вывода (в зоне вывода формируется отсортированный массив). Далее из группы, содержащей элемент, посылаемый в зону вывода, выбирается новый наименьший элемент, который помещается во вспомогательный массив. Затем другой просмотр вспомогательного массива выбирает новый наименьший элемент, который является вторым по величине во всем массиве. Он пересылается в зону вывода. Элементы групп, которые уже посланы во вспомогательный массив, заменяются большими фиктивными величинами.

Таким образом, при сортировке квадратичной выборкой попеременно просматриваются то вспомогательный список, то группа до тех пор, пока все группы не будут исчерпаны. Такое состояние наступает, когда все группы посылают во вспомогательный массив фиктивные величины. Модификацией данного метода является квадратичная выборка с предварительной сортировкой. В этом методе группы сначала полностью упорядочиваются, а затем уже выполняются сравнения между группами. Количество действий, требуемое для сортировки квадратичной выборкой, несколько меньше, чем в предыдущих методах и равно n^2, но требуется дополнительная память. Общее число сравнений для случая точного квадрата равно приблизительно $2n\sqrt{n}$,необходимый резерв памяти – поле длиной $n + \sqrt{n}$ элемент.

Сортировка вставками

Простая вставка

Весь массив делится на две части: упорядоченную и неупорядоченную. Вначале весь массив неупорядочен. На каждом шаге метода вставок из неупорядоченной части извлекается первый элемент, который затем вставляется в нужное место упорядоченной части.

Первый шаг: переместить нулевой элемент из неупорядоченной части в упорядоченную. Тот факт, что элементы в упорядоченной части расположены в порядке возрастания, является инвариантом алгоритма. Поскольку на каждом шаге размер упорядоченной части увеличивается на единицу, а размер неупорядоченной части, соответственно, на единицу уменьшается, в момент окончания алгоритма весь массив окажется упорядоченным. На рисунке 52 представлен пример работы этого метода.

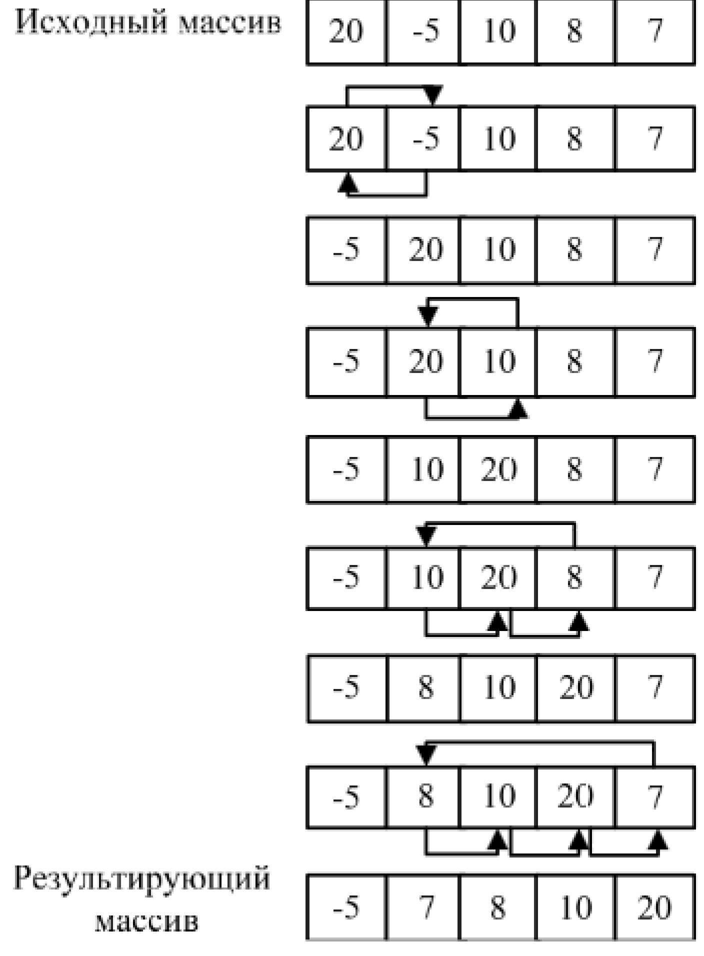


Рисунок 52 - Сортировка вставкой

Рассмотрим функцию, выполняющую сортировку массива методом вставок:

```
void InsertSort(int arr[],int n)
     {
     int temp,i,j;
     for(i = 1;i < n;i++) //для очередного элемента
          {
          temp = arr[i]; //сохранить
          for(j = i-1;j >= 0 && arr[j] > temp; j--)
               //поиск места вставки
               arr[j+1] = arr[j];
```

```
//сдвиг элементов массива
arr[j+1] = temp; //вставка очередного
//на место первого большего его}}
```

Анализ

Внешний цикл в функции выполняется n – 1 раз. Этот цикл содержит внутренний цикл, который выполняется не больше чем n – l. Таким образом, в худшем случае алгоритм выполняет 1 + 2 + ... + (n – l) = n (n – l) / 2 сравнений. Кроме того, в худшем случае столько же раз внутренний цикл сдвигает элементы. Во внешнем цикле перемещение элементов на каждой итерации выполняется дважды, т.е. в сумме 2(n – l) раз. Итак, в наихудшем варианте выполняется n(n – l) + 2(n – l) = n^2 + n – 2 основных операций. В среднем приблизительно n^2 / 4 операций сравнения и n^2 / 4 операций полуобмена элементов местами (перемещений). Следовательно, сложность алгоритма сортировки методом вставок равна $O(n^2)$.

Метод хорош устойчивостью сортировки, удобством для реализации на списках и, самое главное, естественностью поведения. То есть уже частично отсортированный массив будут досортирован им гораздо быстрее, чем многими более эффективными методами. Не требует дополнительной памяти. В отличие от сортировки выбором, время выполнения сортировки вставками зависит главным образом от исходного порядка ключей при вводе. Для больших массивов этот метод неэффективен.

Метод Шелла

Метод Шелла (сортировка с убывающим шагом) существенно превосходит метод простых вставок. Элементы перемещаются большими скачками. Упорядочиваемый массив разделяется на группы элементов, каждая из которых упорядочивается методом вставки. В процессе упорядочения размеры таких групп увеличиваются до тех пор, пока все элементы таблицы не войдут в упорядоченную группу. Группой называют последовательность элементов, номера которых образуют арифметическую прогрессию с разностью h (h называют шагом группы).

В начале процесса упорядочения выбирается первый шаг группы h_1, который зависит от размера массива. Шелл предложил брать $h_1 = [n/2]$,а
$h_i = h_{(i-1)} /2$. В более поздних работах Хиббард показал, что для ускорения процесса целесообразно определить шаг h_1 по формуле $h_1 = 2^k +1$, где $2^k < n \le 2^{(k+1)}$

.

После выбора h_1 методом вставки упорядочиваются группы, содержащие элементы с номерами позиций: i, i + h_1, i + $2h_1$, ..., i + m_i h_1 , при этом i = 1, 2, ..., h_1; m_i – наибольший целый индекс группы. Затем выбирается шаг h_2 и упорядочиваются группы, содержащие элементы с номерами позиций i, i + h_2, ..., i + m_i h_2. Эта процедура, со все уменьшающимися шагами, продолжается до тех пор, пока очередной шаг станет равным единице. Этот последний этап представляет собой упорядочение всего массива методом вставки. Но так как исходная таблица упорядочивалась отдельными группами с последовательным объединением этих групп, то

общее количество сравнений значительно меньше, чем требуется при методе вставки. Число операций сравнения пропорционально $n(\log_2 n)^2$. Ниже приведен пример функции метода Шелла:

```
void ShellSort(int arr[],int n)
{
     for(int step = n/2;step/2 != 0;step/ = 2)
     //цикл выбора шага
          {
          for(int i = 0; i < (n-step); ++i)
               {
                    int j = i ; //запомнить индекс текущего
                                   //сравнить все элементы группы
                    while((j>=0)&&(arr[j]>arr[j+step]))
                    {
                    int temp = arr[j]; //упорядочить методом вставок
                    arr[j] = arr[j+step];
                    arr[j+step] = temp;
                    --j;
                    }
               }
          }
}
```

Рекурсия

Введение

Рекурсивным называется способ построения объекта (понятия, системы), в котором определение объекта включает аналогичный объект в виде некоторой его части.

Функция называется рекурсивной, если ее значение для данного аргумента определяется через значения той же функции для предшествующих аргументов.

Рекурсивная задача в общем случае разбивается на ряд этапов.

Рекурсивная функция «знает», как решать только простейшую часть задачи – базовую (или несколько). Если функция вызывается для решения базовой задачи, она возвращает результат. Если функция вызывается для решения более сложной задачи, она делит эту задачу на две части: одну часть, которую функция умеет решать, и другую, которую функция решать не умеет. Чтобы сделать рекурсию выполнимой, последняя часть должна быть похожа на исходную задачу, но быть по сравнению с ней проще и меньше. Поскольку эта новая задача подобна исходной, функция вызывает копию самой себя, чтобы начать работать над меньшей проблемой – это называется рекурсивным вызовом или шагом рекурсии. Шаг рекурсии выполняется до тех пор, пока исходное обращение к функции не закрыто, т.е. пока не закончено выполнение функции. Чтобы завершить процесс рекурсии, каждый раз, как функция вызывает саму себя, должна формироваться последовательность все меньших и меньших задач, в конце сводящихся к базовой задаче. В этот момент функция распознает базовую задачу, возвращает результат предыдущей функции, и последовательность возвратов

повторяет весь путь назад, пока не дойдет до первоначального вызова и не возвратит конечный результат.

Виды рекурсии

Линейная рекурсия

Простейшим примером рекурсии является линейная рекурсия, при которой функция содержит единственный условный вызов самой себя. В таком случае рекурсия становится эквивалентной обычному циклу. Любой циклический алгоритм можно преобразовать в линейно-рекурсивный и наоборот. Примером линейной рекурсии является вычисление факториала: n! = 1 · 2 · 3 … (n – 1) · n = (n – 1)! · n. Математически можно записать вычисление факториала в следующем виде:

$$n! = \begin{cases} 1, \ n = 0, \\ n(n-1)!, \ n > 0. \end{cases}$$

В первой строке формулы явно указано, как вычислить факториал, если аргумент равен нулю. В любом другом случае для вычисления n! необходимо вычислить предыдущее значение (n – 1)! и умножить его на n. Уменьшающееся значение гарантирует, что в конце концов возникнет необходимость найти 0!, который вычисляется непосредственно.

```
int factorial(int n)
{
    if ( n == 0 ) return 1;
    return n * factorial(n-1);
}
```

Чтобы понять, как будет выполняться эта функция, вспомним, что на время выполнения вспомогательного алгоритма основной алгоритм приостанавливается. При вызове новой копии рекурсивного алгоритма вновь выделяется место для всех переменных, объявляемых в нем, причем переменные других копий будут недоступны. При удалении копии рекурсивного алгоритма из памяти удаляются и все его переменные. Активизируется предыдущая копия рекурсивного алгоритма, становятся доступными ее переменные.

Пусть необходимо вычислить 4!. Основной алгоритм: вводится n = 4, вызов factorial(4) . Основной алгоритм приостанавливается, вызывается и работает factorial(4): 4 ≠ 0, поэтому factorial:= factorial(3)·4. Работа функции приостанавливается, вызывается и работает factorial(3): 3 ≠ 0, поэтому factorial:= factorial(2)·3.

Заметьте, что в данный момент в памяти компьютера две копии функции factorial. Вызывается и работает factorial(2): 2 ≠ 0, поэтому factorial:= factorial(1)·2. В памяти компьютера уже три копии функции factorial и вызывается четвертая. Вызывается и работает factorial(1): 1 ≠ 0, поэтому factorial:= factorial(0)·1. Вызывается и работает factorial(0): 0 = 0 поэтому factorial(0)= 1. Работа этой функции завершена, продолжает работу factorial(1):factorial(1):= factorial(0)·1 = 1 * 1 = 1. Работа функции завершена и работает factorial(2):= factorial(1) · 2 = 1 · 2 = 2.

Работа этой функции также завершена, и продолжает работу функция factorial(3): factorial(3):= factorial(2) · 3 = 2 · 3 = 6.

Завершается работа и этой функции, и продолжает работу функция factorial(4): factorial(4):= factorial(3) · 4 = 6 · 4 = 24 .

Управление передается в основную программу.

Линейная рекурсия – наиболее простой и самый распрстраненный вид рекурсии.

Смешанная (непрямая) рекурсия

При этом виде рекурсии две или более функции вызывают друг друга циклически. Условия завершения могут содержаться во всех или в одной из функций. В качестве примера рассмотрим задачу определения четности числа. Число является четным, если предыдущее число нечетное, и наоборот – число нечетное, если предыдущее четное. Математически:

$$\mathrm{isOdd}(n) = \begin{cases} \mathrm{false}, & n = 0, \\ \mathrm{isEven}(n-1), & n > 0; \end{cases}$$

$$\mathrm{isEven}(n) = \begin{cases} \mathrm{true}, & n = 0, \\ \mathrm{isOdd}(n-1), & n > 0. \end{cases}$$

Код будет выглядеть следующим образом:

```
bool isEven(int n)
{ //условие окончания
     if (n == 0)
          return true;
     else return isOdd(n - 1);
}
bool isOdd(int n)
{
     //условие окончания
     if (n == 0)
     return false;
     else return isEven(n - 1);
}
```

Графически схема выполнения функций для n = 4 представлена на рисунке 53

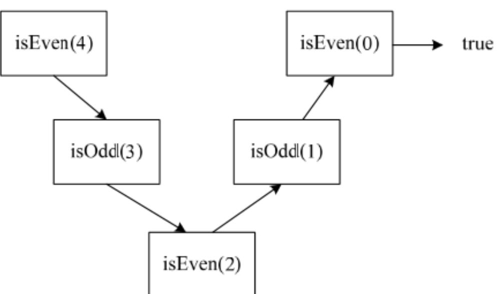


Рисунок 53 - Схема выполнения функций проверки четности

Ветвящаяся рекурсия

В случае ветвящейся рекурсии функция вызывается более одного раза. Частный случай этого вида – бинарная рекурсия (вызов других функций).

Рассмотрим пример – вычисления последовательности чисел Фибоначчи. Последовательность чисел Фибоначчи: 0, 1, 1, 2, 3, 5, 8,13, 21, ... начинается с 0 и 1 и обладает тем свойством, что каждый следующий член последовательности представляет собой сумму двух предыдущих членов. Эта последовательность часто встречается в природе, в частности она описывает форму спирали. Математически последовательность Фибоначчи можно записать следующим образом:

$$\text{Akkerman}(m,n)=\begin{cases} n+1, \ m=0, \\ \text{Akkerman}(m-1,1), \ m>0 \ \text{ и } \ n=1, \\ \text{Akkerman}((m-1), \text{Akkerman}(m,n-1)), \ m>0 \text{ и } n>0. \end{cases}$$

Код программы будет выглядеть следующим образом:

```
int Fibonacci (int n)
{
      if (n < 1) return -1; //ошибка
      if (n == 0)
            return 0;
      else
            if (n == 1)
                  return 1;
            else
                  return Fibonacci (n-1)+ Fibonacci (n-2);
}
```

На рисунке 54 приведена схема вычисления чисел Фибоначчи для n = 3.

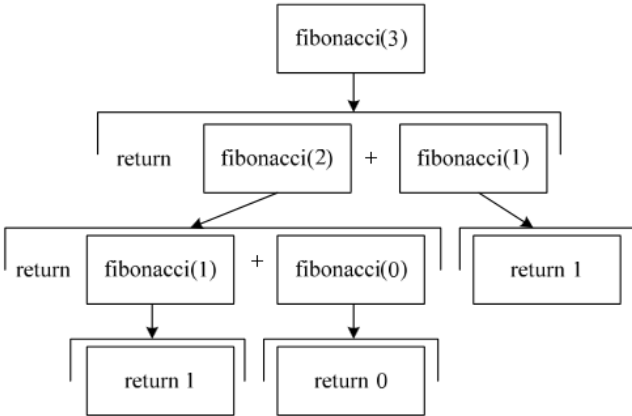


Рисунок 54 - Схема вызова функции вычисления чисел Фибоначчи

Вложенная (гнездовая) рекурсия

Выше рассмотренные рекурсии могут быть заменены итерационным циклом или итерационным циклом со стеком. Однако этот вид рекурсии сложно представить в виде циклической конструкции.

Одним из примеров вложенной рекурсии может быть функция Аккермана. Она определяется следующим образом:

$$\text{Akkerman}(m,n)=\begin{cases} n+1, \ m=0, \\ \text{Akkerman}(m-1,1), \ m>0 \ \text{ и } \ n=1, \\ \text{Akkerman}((m-1), \text{Akkerman}(m,n-1)), \ m>0 \text{ и } n>0. \end{cases}$$

Код программы будет выглядеть следующим образом:

```
int Ackkerman(int m, int n)
{
	if (m < 0 || n < 0)
		return -1;
	if (0 == m)
		return n + 1;
	//линейная рекурсия
	else
		if (m > 0 && 0 == n)
			return Ackkerman(m-1, 1);
		//вложенная
		else
			return Ackkerman(m-1, Ackkerman(m, n-1));
}
```

Особенности программирования рекурсивных функций

Техника рекурсивных вызовов функций

Очевидно, что рекурсия не может быть безусловной, в этом случае она становится бесконечной. Следовательно, рекурсия должна иметь внутри себя условие завершения, по которому очередной рекурсивный вызов уже не производится.

Рассмотрим пример вызова рекурсивной функции S. Движемся по ее тексту до тех пор, пока не встретим ее вызова, после чего опять начинает выполняться та же самая функция сначала, при этом ее первый вызов еще не закончился. Создается впечатление, что текст функции воспроизводится (копируется) всякий раз, когда функция сама себя вызывает:

```
void main()      void S()  void S()  void S()
{                {         {         {
S();             ..if()S();    ...if()S();    ...if()S();
}                }         }         }
```

Копируется при этом не весь текст функции (не вся функция), а только ее части, связанные с данными (формальные, фактические параметры, локальные переменные и точка возврата). Создается копия локальных данных по следующей схеме:

• в стеке резервируется место для формальных параметров, в которые записываются значения фактических параметров (обычно в порядке, обратном их следованию в списке);

• при вызове функции в стек записывается точка возврата – адрес той части программы, где находится вызов функции;

• в начале тела функции в стеке резервируется место для локальных (автоматических) переменных.

Алгоритм (операторы, выражения) рекурсивной функции не меняется, поэтому он присутствует в памяти компьютера в единственном экземпляре. Таким образом, формальные параметры рекурсивной функции, внешние и локальные переменные не могут быть взаимозаменяемы. Кроме того, каждый новый рекурсивный вызов порождает новый «экземпляр» формальных параметров и локальных переменных, причем старый «экземпляр» не уничтожается, а сохраняется в стеке по принципу вложенности. Здесь имеет

место единственный случай, когда в процессе работы программы одному имени переменной соответствуют несколько ее «экземпляров» (рисунок 55).

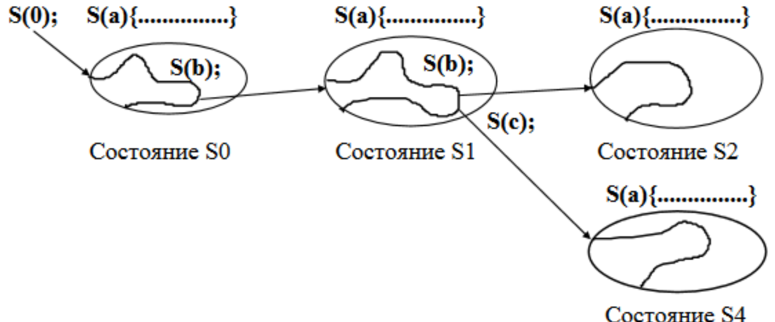


Рисунок 55- Схема вывоза рекурсивной функции

При завершении рекурсии программа возвращается к предыдущей версии рекурсивной функции и к предыдущему фрейму в стеке.

Принципы программирования рекурсивных функций

Принцип программирования рекурсивных функций имеет много общего с методом математической индукции. Этот метод используется для доказательства корректности утверждений для бесконечной последовательности состояний и неявно применяется при разработке рекурсивных функций. Действительно, сама рекурсивная функция представляет собой переход из n-го в (n + 1)-е состояние некоторого процесса.

Если этот переход корректен, то есть соблюдение некоторых условий на входе функции приводит к их соблюдению на выходе (то есть в рекурсивном вызове), то эти условия будут соблюдаться во всей цепочке состояний (при безусловной корректности начального).

Опираясь на метод математической индукции, можно сформулировать следующие правила:

• рекурсивная функция разрабатывается как обобщенный шаг процесса, который вызывается в произвольных начальных условиях и который приводит к следующему шагу в некоторых новых условиях;

• обобщенные начальные условия шага – формальные параметры функции;

• начальные условия следующего шага – фактические параметры рекурсивного вызова;

• рекурсивная функция должна проверять условия завершения рекурсии, при которых следующий шаг процесса не выполняется;

• локальными переменными функции должны быть объявлены все переменные, которые имеют отношение к протеканию текущего шага процесса.

Рекурсии и итерации

Некоторые рекурсивные функции могут быть реализованы итеративно. Сравним эти два подхода.

Как итерации, так и рекурсии включают повторение: итерации используют структуру повторения явным образом, рекурсии реализуют повторение посредством повторных вызовов функции. Как итерации, так и

рекурсии включают проверку условия окончания: итерации заканчиваются после нарушения условия продолжения цикла, рекурсии заканчиваются после распознавания базовой задачи. Как итерации, так и рекурсии могут оказаться бесконечными.

Несмотря на то, что запись рекурсивных функций бывает очень короткая, они требует больших накладных расходов. При каждом вызове в стеке должны быть размещены все параметры и локальные переменные. На эту работу (как и на последующее освобождение) расходуется и время, и пространство – в программу вставляются соответствующие команды, которые выполняются при каждом вызове, расходуется память под стек. Поскольку рекурсивный вызов выполняется до окончания выполнения функции – размер стека увеличивается при каждом вызове до тех пор, пока не будет достигнута точка, когда выполняется возврат. Размер стека пропорционален глубине рекурсии.

Глубина рекурсии – это максимальная степень вложенности рекурсивных вызовов. В общем случае глубина будет зависеть от входных данных. Поэтому при разработке рекурсивных функций необходимо минимизировать количество и размеры локальных переменных и параметров. Существует также и другой недостаток – лишние вычисления. Одним из методов борьбы с этим недостатком является сокращение количества рекурсивных обращений в тексте функции (если есть возможность вместо двух вызовов оставить один).

Вывод: рекурсивные вызовы требуют времени и дополнительных затрат. Тогда, в решениях, где требуется высокая эффективность, следует избегать рекурсии.

Тем не менее, рекурсивный подход обычно предпочитают итеративному. Он более естественно отражает задачу и ее результаты, т.е. более нагляден, легче отлаживается и считается хорошим стилем программирования. Другая причина предпочтения рекурсивного решения состоит в том, что итеративное решение может не быть очевидным.

Результат рекурсивной функции

При реализации поисковых задач следует обратить внимание на результат рекурсивной функции. Здесь возможны варианты:

- рекурсивная функция сама выводит выбранный элемент в случае успешного поиска (это приемлемо, если осуществляется поиск первого попавшегося варианта, но не очень хорошо с точки зрения технологии программирования);
- выбранный элемент записывается в область глобальных данных, в которых моделируется стек для возврата результата (само по себе использование глобальных данных не является хорошим решением);
- в качестве результата функции используются более сложные динамические структуры данных, например, список результатов.

Раздел 4 АДРЕСНАЯ АРИФМЕТИКА. ДИНАМИЧЕСКАЯ ПАМЯТЬ. ДИНАМИЧЕСКИЕ СТРУКТУРЫ

Лекция 12 Указатели и адреса

Указатели и адреса

Указатель – это символическое представление адреса. Он используется для косвенной адресации переменных и объектов. Указатели тесным образом связаны с обработкой строки и массива.

В языке С++ имеется операция определения (взятия) адреса – `&`, с помощью которой определяется адрес ячейки памяти, содержащей заданную переменную. Например, если `a` – имя переменной, то `&a` – адрес этой переменной (рисунок 56).

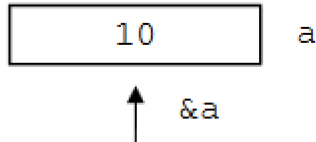


Рисунок 56 - Адрес и имя переменной а

Запись `&a` означает: «указатель на переменную `a`». Операция `&` применима только к объектам, имеющим имя и размещенным в памяти (переменным, массивам, структурам, строковым литералам и т.п.). Ее нельзя применить к переменным с классом хранения `register`, а также к полям бит.

Символическое представление адреса `&a` является константой типа указатель, ни одна операция на нее не действует:

```
int i = 0;
int& a = i;
a++;
//не увеличивает ссылку, а i увеличивается на 1
```

В С++ также существуют и переменные типа указатель. При работе с указателями, важное значение имеет операция косвенной адресации – `*`. Операция `*` позволяет обратиться к переменной не напрямую, а через указатель, содержащий адрес этой переменной. Данная операция является одноместной и имеет ассоциативность слева направо. Пусть `pa` – указатель, тогда `*pa` – это значение переменной, на которую указывает `pa`.

В С++ существует общепринятое соглашение – называть переменные-указатели именами, начинающимися с p (pointer), чтобы было ясно, что это указатель и требуется соответствующая обработка.

Формат объявления указателя:

```
спецификатор-типа *описатель;
```

Спецификатор-типа задает тип объекта и может быть любого основного типа, типа структуры, смеси (об этом будет сказано ниже) или `void`. Чтобы выполнить арифметические и логические операции над указателями или над объектами, на которые они указывают, необходимо при выполнении каждой операции явно определить тип объектов. Такие определения типов могут быть выполнены с помощью операции приведения типов. Например:

```
char *pz;
int *pk,*pi;
float *pf;
```

Здесь * – это операция разыменования, то есть ссылка на объект, на который указывает указатель. Операндом этой операции всегда является указатель. Результат операции – это тот объект, который адресует указатель-операнд (рисунок 57).

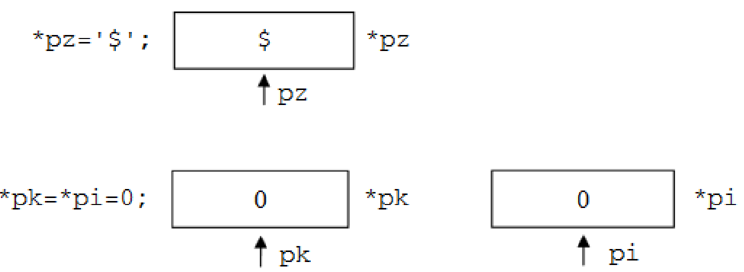


Рисунок 57 - Операции разыменования

```
int e, c, b, *pm;
....................
pm = &e ; //в pm хранится адрес e;
*pm = c + b ;
```

Графическая интерпретация фрагмента программы приведена на рисунке 58. Операция `*` в некотором смысле является обратной операции `&`.

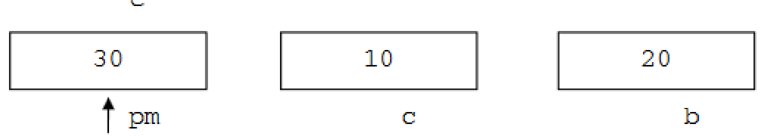


Рисунок 58 – Графическая интерпритация примера

Использование неинициализированных указателей опасно (можно перезаписать произвольную память). Объявляя указатель, необходимо инициализировать его адресом определенной переменной или нулем.

Для этого в C++ предусмотрена символическая константа `NULL`, которая уже определена (в `iostream`) как 0:

```
int *pm = NULL; //указатель, не указывающий ни на что
....................
if (pm == NULL)
cout << endl << "нулевой указатель";
```

Это гарантирует, что указатель не содержит адреса, который воспринимается как корректный.

Без указателей невозможна обработка объектов. Например, передать объект некоторой функции можно только через указатель. Указатели используются для работы с динамической памятью.

Операции над указателями

Над указателями возможны операции: присваивание (=) указателей одного типа; получение значения объекта, на который ссылается указатель (*); получение адреса самого указателя (&).

Пример

```
int date = 10;
int *pi, *pk;
pi = &date;
pk = pi;
```

Пример работы с указателями из данного фрагмента кода приведен на рисунке 59

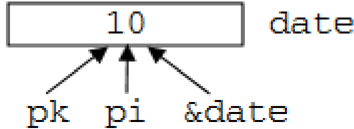


Рисунок 59 – Значение указателей

Допустимы также операции сравнения: ==, !=, <, <=, >, >=, адресной арифметики и все производные от нее: +, –, ++, — . С помощью унарных операций ++ и — числовые значения переменных типа указатель меняются по-разному, в зависимости от типа данных, с которым связаны эти переменные:

```
char *pz;
int *pk,*pi;
float *pf;
pz++; //значение указателя изменяется на 1
pi++; //значение указателя изменяется на 4
pf++; //значение указателя изменяется на 4
```

При изменении указателя на 1, указатель переходит к началу следующего (предыдущего) поля той длины, которая определяется типом объекта, адресуемого указателем.

Снабжение описания указателя префиксом `const` делает объект, но не сам указатель, константой:

```
const char* pc = "asdf"; //указатель на константу
pc[3] = 'a'; //ошибка
pc = "abc"; //ok
```

Чтобы описать сам указатель, а не указываемый объект, как константный, используется операция const*:

```
char *const cp = "asdf"; //константный указатель
cp[3] = 'a'; //ошибки нет
cp = "abc"; //ошибка
```

Чтобы сделать константами оба объекта, их оба нужно описать `const`:

```
const char *const cpc = "asdf"//const указатель на const
cpc[3] = 'a'; //ошибка
cpc = "abc"; //ошибка
```

Указателю на константу можно присваивать адрес переменной.

Однако нельзя присвоить адрес константы указателю, на который не было наложено ограничение, поскольку это позволило бы изменить значение объекта.

Указатели и массивы

Поскольку указатели представляют собой символические адреса объектов, то это дает возможность применять адреса и повышать эффективность программ.

Имя массива arr без индекса является указателем-константой (не изменяются на протяжении всей работы программы), т.е. адресом первого элемента массива (&arr[0]) (рисунок 60).

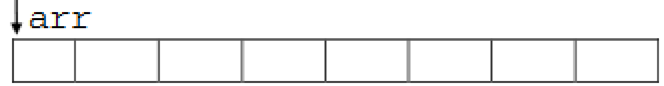


Рисунок 60 – Имя массива как указатель

```
int arr[8],*parr;
*arr = = arr[0] ; //эквивалентно
*(arr + 1) = = arr[1]; //эквивалентно
*(arr + i) = = arr[i]; //эквивалентно
parr = arr + 5;
//указатель на 5-й элемент массива
parr = &arr[2];
//указатель на 2-й элемент массива
arr[3] = *(parr+1);
//3-му элементу массива присвоить знач. 1-го
```

Необходимо иметь в виду, что указатель является переменной, так что операции parr = arr и parr++ имеют смысл, но имя массива является константой, а не переменной, поэтому следующие конструкции будут не допустимы:

```
arr = parr; //ошибка
arr++; //ошибка
parr = &arr; //ошибка
```

Листинг фрагмента программы с использованием указателей для обращения к элементам массива будет выглядеть следующим образом:

```
int arr[100];
int rmax = 10;
int rmin = 0;
.............................................
cout<<"Введите размер массива"<<endl;
cin>> array_size;
//Сгенерировать массив
srand((unsigned)time(NULL));
for(i=0;i< array_size; i++)
{
     *(arr + i) = (int)(((double)rand()/(double)RAND_MAX)*
     (rmax-rmin)+rmin);
     //или *(arr + i) = rand()%99;
     cout<<*(arr + i)<<endl;
}
```

```
//Подсчитать среднее арифметическое суммы элементов:
avg = 0;
for(i = 0; i < array_size; i++)
{
     //т.к. в теле цикла один оператор,
     //фигурная скобка не обязательна
     avg += *(arr + i);
}
avg /= array_size;
cout<<"Средний элемент массива"<<avg<<endl;
cout<<"Введите номер удаляемого элемента массива"<<endl;
cin >> k;
//удалить элемент с номером k, сдвиг элементов массива
for(i = k; i < array_size-1; i++)
{
     *(arr + i) = *(arr + i + 1);
}
array_size--; //уменьшить размер массива
for(i = 0; i < array_size; i++)
     cout<<*(arr + i)<<endl;
...........................................
```

Указатели и доступ к элементам многомерного массива

Указатели на многомерные массивы – это массивы массивов, т.е. такие массивы, элементами которых являются массивы. Например, при выполнении объявления двумерного массива `arr[4][3]` в памяти выделяется участок для хранения значения переменной arr, которая является указателем на массив из четырех указателей. Для этого массива из четырех указателей тоже выделяется память. Каждый из этих четырех указателей содержит адрес массива из трех элементов типа int, и, следовательно, в памяти компьютера выделяется четыре участка для хранения четырех массивов чисел типа int, каждый из которых состоит из трех элементов. Такое выделение памяти показано на рисунке 61.

arr

↓

arr[0]	→	arr[0][0]	arr[0][1]	arr[0][2]
arr[1]	→	arr[1][0]	arr[1][1]	arr[1][2]
arr[2]	→	arr[2][0]	arr[2][1]	arr[2][2]
arr[3]	→	arr[3][0]	arr[3][1]	arr[3][2]

Рисунок 61 – Многомерный массив

Для доступа к элементам двумерного массива должны быть использованы два индексных выражения в форме `arr[m][n]` или эквивалентных ей `*(*(arr+m)+n)`, `*(arr[m]+n)` и `(*(arr+m))[n]`. Действительно `arr+m` ссылается на строку номер m. Выражение `*(arr+m)` – это адрес нулевого элемента строки m , поэтому `*(arr+m)+n` – адрес элемента в строке m со смещением `n`. Таким образом, полное выражение ссылается на конкретный элемент массива.

С точки зрения синтаксиса языка указатель arr и указатели arr[0], arr[1], arr[2], arr[3] являются константами, и их значения нельзя изменять во время выполнения программы.

При размещении элементов многомерных массивов они располагаются в памяти подряд по строкам. Такой порядок дает возможность обращаться к любому элементу многомерного массива, используя адрес его начального элемента и только одно индексное выражение:

```
float *ptr, *ptr3, arr[4][4][4];
ptr = arr;
ptr3 = arr;
//тогда следующие обращения к элементам будут эквивалентными
//arr[2][3][4] == ptr[3*2+4*3+4] == ptr3[22];
```

Элементы массивов могут иметь любой тип, и, в частности, могут быть указателями на любой тип. Следующие объявления переменных:

```
int a[]={10,11,12,13,14,};
int *p[]={a, a+1, a+2, a+2, a+3, a+4};
int **pp=p;
```

порождают программные объекты, представленные на рисунке 62.

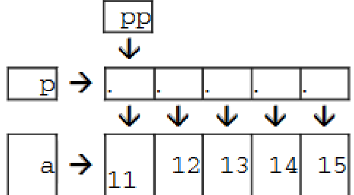


Рисунок 62 - Программные объекты

Если считать, что pp = p, то обращение *++pp – значение первого элемента массива a (т.е. значение 11), операция ++*pp изменит содержимое указателя p[0] таким образом, что он станет равным значению адреса элемента a[1].

Динамические объекты

Во многих случаях во время выполнения программы у компьютера есть неиспользуемая память. Эта память называется «кучей» или «свободным хранилищем». Внутри этого «хранилища» можно выделить память, и затем освободить выделенное пространство и вернуть его в «свободное хранилище» после того, как необходимость в переменных отпадет. Эта техника позволяет эффективно расходовать память и разрабатывать программы, способные решать масштабные проблемы, обрабатывая большие объемы данных.

Также при традиционном определении массива:

```
тип имя_массива [количество_элементов];
```

общее количество памяти, выделяемой под массив, задается определением и равно sizeof имя_массива. Но иногда бывает нужно, чтобы память под массив выделялась для решения конкретной задачи, причем ее размеры заранее не известны и не могут быть фиксированы.

Формирование объектов с переменными размерами можно организовать с помощью указателей и средств динамического распределения

памяти двумя способами: с использованием библиотечных функций (С); с использованием операций `new` и `delete` (С++).

Формирование динамических переменных с использованием операций `new` и `delete`

Для динамического распределения памяти в С++ используются операции `new` и `delete`. Формат:

```
new имя_типа;
```

или

```
new имя_типа [инициализатор];
```

позволяет выделить и сделать доступным свободный участок памяти, размеры которого соответствуют типу данных, определяемому именем типа. В выделенный участок заносится значение, определяемое инициализатором, который не является обязательным параметром. В случае успешного выделения памяти операция возвращает адрес начала выделенного участка памяти; если участок не может быть выделен, то возвращается `NULL` и программа может быть прервана по исключению. Например:

```
int *i;
i = new int (10);
//создать переменную типа int, инициализировать - 10
...
float *f;
f = new float;
//создать динамическую переменную типа float
int *mas = new[5];
//создать динамический массив из 5 целых элементов
```

Операция `delete` освобождает участок памяти, ранее выделенный операцией `new`.

```
delete i;
//освободить память по указателю i
...
delete f;
//освободить память по указателю f
delete [] mas;
//освободить память, выделенную под массив mas
```

Динамическое выделение памяти – потенциальный источник ошибок, и возможно, наиболее распространенная из них в этом контексте – утечка памяти. Это происходит, когда используется операция `new` для выделения памяти, но не используется операция `delete` для ее освобождения, когда она больше не нужна.

По отношению к массивам динамическое распределение памяти особенно полезно, поскольку иногда бывают массивы больших размеров.

При формировании динамической матрицы сначала выделяется память для массива указателей на одномерные массивы, а затем в цикле с параметром выделяется память под `n` одномерных массивов (рисунок). Например:

```
//Функция для формирования двумерного
//динамического массива
int ** make_matr(int n)
```

```
{
      int **matr; //адрес адресов - массив указателей
      int i,j;
      matr = new int*[n];
      //выделить память под массив из n элементов
      for (i = 0; i < n ; i++)
      {
            matr[i] = new int[n];
            //выделить память n раз под массив из n
            for (j = 0; j < n ;j++)
            matr[i][j] = rand()% 10; //инициализация
      }
return matr;
}
```

**matr
*matr[1] *matr[2] *matr[3] . . . . *matr[n]

Рисунок 63 - Схема выделения памяти под n одномерных массивов

Нельзя специфицировать начальные значения элементов массива, который распределен динамически. Это следует сделать явным присвоением. Чтобы освободить память, необходимо выполнить цикл для освобождения одномерных массивов:

```
for(int i = 0; i < n ;i++)
delete matr[i]; //После этого освобождаем память,
//на которую указывает указатель matr
delete [] matr;
```

Для уничтожения динамического массива применяется оператор delete [ ]. Скобки ([ ]) играют важную роль. Они сообщают оператору, что требуется уничтожить все элементы массива, а не только первый.

Константный динамический объект

Чтобы создать динамический объект и запретить изменение его значения после инициализации, необходимо объявить объект константным. Для этого применяется следующая форма оператора new:

```
const int *pci = new const int(1024);
```

Константный динамический объект имеет несколько особенностей.

Во-первых, он должен быть инициализирован, иначе компилятор сигнализирует об ошибке (кроме случая, когда объект принадлежит к типу

класса, имеющего конструктор по умолчанию; в такой ситуации инициализатор можно опустить). Во-вторых, указатель, возвращаемый выражением `new`, должен адресовать константу. В предыдущем примере `pci` служит указателем на `const int`. Константность динамически созданного объекта подразумевает, что значение, полученное при инициализации, в дальнейшем не может быть изменено. Временем его жизни управляет оператор `delete`.

Смешивание способов динамического распределения памяти в стиле `new -delete` со стилем `malloc -free` является логической ошибкой: пространство, созданное с помощью `malloc`, не может быть освобождено с помощью `delete`; объекты, созданные с помощью `new`, не могут быть уничтожены с помощью `free`.

Массивы указателей

Массив указателей является самой простой и одновременно самой распространенной динамической структурой данных. Одно из его определений имеет вид:

```
double *p[20];
```

В соответствии с принципом контекстного определения типа данных переменную `p` следует понимать как массив, каждым элементом которого является указатель на переменную типа `double`. Исходя из принятой концепции указателя, эту структуру можно рассматривать как массив указателей на отдельные переменные типа `double`, так и как указатели на массивы этих переменных:

```
char *pc[] = {"aaa", "bbb", "ccc", NULL};
```

Массивы указателей, как и все остальные структуры данных, содержащие указатели, допускают различные способы формирования, которые отличаются как способом создания самих элементов, так и способом установления связей между ними. Например:

```
double a1,a2,a3, *pd[] = { &a1, &a2, &a3, NULL};
//переменные - статические, указатели инициализируются
double d[19], *pd[20];
.......................
for (i = 0; i < 19; i++)
     pd[i] = &d[i];
pd[i] = NULL;
......................
double *p, *pd[20];
for (i = 0; i < 19; i++)
{
     p = new double; //переменные создаются динамически
     *p = i;
     pd[i] = p; //массив указателей - статически:
}
pd[i] = NULL;
......................
double **pp, *p;
pp = new double *[20];
//массив указателей создается динамически
```

```
for (i = 0; i < 19; i++)
{
     p = new double; //переменные создаются динамически
     *p = i;
     pp[i] = p;
}
pp[i] = NULL;
```

Лекция 13 Указатели и функции

Вызов функции с переменным числом параметров

При вызове функции с переменным числом параметров задается любое требуемое число аргументов. В объявлении и определении такой функции переменное число аргументов задается многоточием в конце списка формальных параметров или списка типов аргументов.

При этом должен быть указан как минимум один обычный параметр.

```
int sumVal(int first, ...)
{
     // код функции
}
```

В перемененном списке параметров нет информации о числе и типах аргументов, поэтому код должен определять, что ему передано при вызове функции. Существует два подхода: когда известно количество параметров, которое передается как обязательный параметр; когда известен признак конца списка параметров.

Рассмотрим пример функции вычисления суммы значений переменного числа параметров. Список параметров состоит из одного параметра – числа дополнительных параметров:

```
int sum (int, ...); //прототип функции
........................
cout << sum(6, 1, 1, 2, 1, 3, 5); //вызов
........................
int sum (int num_val, ...)
//num_val - число суммируемых параметров
{
     int *p = &num_val; //выход на начало списка
     int s = 0;
     for (int i = 1; i <= num_val; i++)
          {
          s += *(++p);
          }
     return s;
}
```

Указатели как формальные параметры и результат функций

В большинстве языков программирования параметры функциям передаются либо по ссылке (by reference), либо по значению (by value). В первом случае функция работает с адресом переменной, переданной ей в качестве фактического параметра. Во втором случае ей доступна не сама переменная, а только ее значение (копия числа). Различие заключается в том, что переменную, переданную по ссылке, функция может модифицировать в вызвавшей ее функции, а переданную по значению – нет. На рисунке 64 приведена диаграмма механизма передачи по значению.

В языке С параметры передаются только по значению. Следствие применения механизма передачи по значению состоит в том, что функция не может напрямую модифицировать переданные ей аргументы.

Общепринятый способ обеспечить функции непосредственный доступ к какой-либо переменной из вызывающей программы заключается в том, чтобы вместо самой переменной в качестве параметрапередавать ее адрес.

```
int first_val = 2;
int second_val = 5;
int result = add (first_val, second_val )
```

first_val — 2

Временные копии создаются для использования в функции

second_val — 5

5 2

```
int add (int i,  int j)
{
return i+j;
//исходные аргументы не доступны,
//здесь только копии
}
```

Рисунок 64 – Механизм передачи аргумента по значению

Рассмотрим пример, в котором происходит обмен значений переменных:

```
.............................
void swap ( int *, int *); //прототип функции
     .............................
     int a = 10;
     int b = 20;
     swap (&a, &b); //вызов функции
     .............................
     void swap ( int *x, int *y)
     {
          int c; //временная переменная
          c = *x; //запоминаем x
          *x = *y; //в x записываем y
```

```
        *y = c; //в y записываем x
    }
```

При вызове функции передаются адреса переменных `a` и `b`.

Это означает, что формальные параметры должны быть описаны как указатели на соответствующий тип данных. При таком описании формальных параметров вызываемой программе становятся доступными адреса переменных из вызывающей программы, которые являются в ней локальными. Если известны адреса, то применимы операции чтения и возврата по адресу.

Например:

```
//вычисление суммы элементов массива вариант 1
int sum (int n, int a[])
{
    int s = 0;
    for(int i = 0; i < n; i++ )
        s += a[i];
    return s;
}
.............................
int a[] = { 3, 5, 7, 9, 11, 13, 15 }; //массив
int s = sum( 7, a ); //вызов функции sum
cout << s;
```

....

//вычисление суммы элементов вариант 2

```
int sum (int n, int *a)
{
int s = 0;
for(int i = 0, s = 0; i < n; s += *a++, i++);
return s;
}
```

....

```
int a[] = { 3, 5, 7, 9, 11, 13, 15 };
int s = sum( 7, a ); //вызов функции sum
cout << s;
```

Во втором варианте функции sum везде используется нотация указателей, несмотря на то, что работа идет с массивом. В операторе цикла при суммировании элементов массива s += *a++ увеличивается адрес массива. Механизм передачи по значению создает копию исходного адреса массива, поэтому в теле функции модифицируется копия, исходный адрес массива остается неизменный.

Указатель на функцию

Указателем на функцию называется переменная, которая содержит адрес некоторой функции (адрес точки входа в функцию). Общая форма объявления указателя на функцию выглядит следующим образом:

```
тип_возврата (*имя_функции) ([список-параметров]);
```

Соответственно, косвенное обращение по этому указателю представляет собой вызов функции. Определение указателя на функцию может иметь вид:

```
int (*pf)(); //без контроля параметров вызова
```

```
int (*pf)(int, char*); //с контролем по прототипу
```

Выражение вида `&имя_функции` имеет смысл – начальный адрес функции или указатель на функцию. По аналогии с именем массива использование имени функции без скобок интерпретируется как указатель на эту функцию. Указатель может быть инициализирован и при определении. Возможны следующие способы назначения указателей:

```
int INC(int a)
{
     return a+1;
}
extern int DEC(int);
int (*pf)(int);
pf = &INC;
pf = INC; //присваивание указателя
int (*pp)(int) = &DEC; //инициализация указателя
```

Передача указателя на функцию

Указатели на функции могут входить в более сложные структуры данных. В программах указатели на функции в основном используются для сохранения адресов программ обработчиков прерываний и для передачи функций в качестве формальных параметров другим функциям. Рассмотрим пример, демонстрирующий такую возможность при вычислении интеграла функций методом площадей:

```
float integral(float(*)(float),float,float,float);
float funct(float); //прототипы функций
.............................
float intefral_value;
intefral_value =
     integral(funct(float)0.0,(float)10.0,(float)0.01)
//вызов функции
.............................
float integral(float(*f)(float),float low,float up,float delta)
{
          //float(*f)(float) -
          //указатель на интегрируемую функцию
          //float low - нижний предел интегрирования
          //float up - верхний предел интегрирования
          //float delta - шаг приращения по x
          float x, summa = 0.0;
          x = low + delta / 2.0;
          while (x <= up)
     {
     summa += delta * f(x);
     x = x + delta;
     }
     return summa;
}
float funct (float x)
{
     return (2*x+5); //интегрируемая функция y = 2x+5
}
```

Массивы указателей на функции

Разрешено объявлять массивы указателей на функции (по аналогии с обычными указателями). Их можно инициализировать в объявлении:

```
double one (double, double); //прототип функции
double two (double, double); //прототип функции
double three (double, double); //прототип функции
double (*pfun[3]) (double, double) = {one, two, three};
//массив указателей на функции
pfun[1](1.0, 2.0); //вызов функции one
```

Лекция 14 Структуры данных

Основные структуры данных, существующие в языке C++ - это переменные, массивы, структуры, объединения. Из них можно образовывать сложные структуры.

Переменные, массивы, структуры, объединения при объявлении получают имя и тип, для их хранения выделяется область оперативной памяти, в которую можно записывать некоторые значения. Таким образом, данные объекты имеют неизменяемую (статическую) структуру. Существует, однако, много задач, в которых требуются данные с более сложной (динамической) структурой. Для такой структуры характерно, что в процессе вычислений изменяются не только значения объектов, но и структура хранения информации. Поэтому такие объекты называются динамическими информационными структурами. Их компоненты, в свою очередь, на некотором уровне детализации представляют собой объекты со статической структурой, то есть они принадлежат к одному из основных типов данных.

Структуры данных — это особый способ организации данных в специализированном формате на компьютере, чтобы информация могла быть организована, обработана, сохранена и извлечена быстро и эффективно. Они являются средством обработки информации, предоставляя данные для простого использования.

Каждое приложение, часть программного обеспечения или основа программ состоит из двух компонентов: алгоритмов и данных. Данные — это информация, а алгоритмы — это правила и инструкции, которые превращают данные во что-то полезное для программирования.

Другими словами, запомните эти два простых уравнения:

Связанные данные + Допустимые операции над данными = Структуры данных

Структуры данных + Алгоритмы = Программы

Характеристики структур данных

Структура данных — это систематический способ организации данных. Характеристики структур данных:

Линейный или нелинейный

Эта характеристика упорядочивает данные в последовательном порядке, например, в виде массивов, графиков и т. д.

Статика и динамика

Статические структуры данных имеют фиксированные форматы и размеры вместе с ячейками памяти. Статическая характеристика показывает компиляцию данных.

Сложность времени

Фактор времени должен быть очень пунктуальным. Время выполнения или время исполнения программы должно быть ограничено. Время выполнения должно быть как можно меньше. Чем меньше время выполнения, тем точнее устройство.

Корректность

Каждые данные обязательно должны иметь интерфейс. Интерфейс отображает набор структур данных. Структура данных должна быть точно реализована в интерфейсе.

Сложность пространства

Пространство в устройстве должно быть тщательно использовано. Использование памяти должно быть использовано правильно. Пространство должно быть менее занятым, что указывает на правильное функционирование устройства.

Типы структур данных

До сих пор мы касались типов данных и классификаций структур данных. Наше знакомство с многочисленными элементами структур данных продолжается рассмотрением различных типов структур данных.

Множество.

Массивы — это наборы элементов данных одного типа, хранящиеся вместе в смежных ячейках памяти. Каждый элемент данных называется «элементом». Массивы — это самая базовая, фундаментальная структура данных. Начинающие специалисты по данным должны освоить построение массивов, прежде чем переходить к другим структурам, таким как очереди или стеки.

Графики.

Графы — это нелинейное графическое представление наборов элементов. Графы состоят из конечных наборов узлов, также называемых вершинами, соединенных связями, поочередно называемыми ребрами. Деревья, упомянутые ниже, являются вариацией графа, за исключением того, что последний не имеет правил, регулирующих, как соединяются узлы.

Хэш-таблицы.

Хэш-таблицы, также называемые хэш-картами, могут использоваться как линейная или нелинейная структура данных, хотя они предпочитают первую. Эта структура обычно строится с использованием массивов. Хэш-таблицы сопоставляют ключи со значениями. Например, каждой книге в библиотеке присвоен уникальный номер, который облегчает поиск информации о книге, например, кто ее брал, ее текущая доступность и т. д. Книги в библиотеке хэшируются в уникальный номер.

Связанный список.

Связанные списки хранят коллекции элементов в линейном порядке. Каждый элемент в связанном списке содержит элемент данных и ссылку или ссылку на последующий элемент в том же списке.

Куча.

Стеки хранят коллекции элементов в линейном порядке и используются при применении операций. Например, порядок может быть «первым пришел, первым ушел» (FIFO) или «последним пришел, первым ушел» (LIFO).

Очередь.

Очереди хранят коллекции элементов последовательно, как стеки, но порядок операций должен быть только «первым пришел, первым вышел». Очереди представляют собой линейные списки.

Дерево.

Деревья хранят коллекции элементов в абстрактной иерархии. Это многоуровневые структуры данных, использующие узлы. Нижние узлы называются «узлами листьев», а самый верхний узел известен как «корневой узел». Каждый узел имеет указатели, которые указывают на соседние узлы.

Попробуйте.

Не путать с деревом, Tries — это структуры данных, которые хранят строки как элементы данных и размещаются в визуальном графе. Tries также называются деревьями ключевых слов или префиксными деревьями. Всякий раз, когда вы используете поисковую систему и получаете автопредложения, вы наблюдаете структуру данных trie в действии.

Типы деревьев в структуре данных

Общее дерево

Дерево считается общим деревом, если его иерархия не ограничена. Нет ограничений на количество потомков, которые может иметь узел в общем дереве. Все остальные деревья являются подмножествами дерева.

Двоичное дерево

Двоичное дерево — это своего рода древовидная структура данных, в которой каждый родительский узел имеет не более двух дочерних узлов. Как следует из названия, двоичный означает два, поэтому каждый узел может иметь ноль, один или два узла. Популярность этого дерева выше, чем у большинства других. Двоичное дерево может быть изменено для учета определенных ограничений и функций, например, с помощью дерева AVL, дерева BST, дерева RBT и других. Мы подробно рассмотрим все эти стили по мере продвижения.

Двоичное дерево поиска

Эти структуры данных дерева нелинейны, один узел соединяется с несколькими другими. К узлу можно присоединить максимум два дочерних узла. Двоичное дерево поиска так называется, потому что:

1. Каждый узел может иметь до двух дочерних узлов.
2. Его можно использовать для поиска элемента за время 0(log(n)), поэтому его называют деревом поиска.

Дерево АВЛ

Дерево AVL — это самобалансирующееся бинарное дерево поиска. Адельсон-Велши и Ландис — изобретатели термина AVL. Здесь впервые были созданы динамически сбалансированные деревья. В зависимости от того, сбалансировано ли дерево AVL или нет, каждому узлу назначается коэффициент балансировки. Дети узлов имеют максимальную высоту в одну лозу AVL. Правильные коэффициенты балансировки в дереве AVL — 1, 0 и -1. Если к дереву добавляется новый узел, он будет повернут, чтобы обеспечить его сбалансированность. Затем он будет повернут. В дереве AVL общие операции, такие как просмотр, вставка и удаление, требуют времени O(log n). Обычно он используется при выполнении действий Lookups.

Дерево Б

AB Tree — это более общее бинарное дерево поиска. Сбалансированное по высоте m-стороннее дерево относится к этому типу деревьев, где m обозначает порядок дерева. Каждый узел дерева может иметь несколько ключей и более двух дочерних узлов. Листовые узлы бинарного дерева могут не находиться на одном уровне. Важно, чтобы все листовые узлы в B-дереве были одинаковой высоты.

Типы графиков

Нулевой график

Графы нулевого порядка — это еще одно название нулевого графа. Граф с пустым множеством ребер называется «нулевым графом». Как следует из названия, нулевой граф имеет 0 ребер и состоит только из изолированных вершин.

Тривиальный граф

Если граф содержит только одну вершину, он называется тривиальным графом. Одна вершина — это все, что нужно для построения тривиального графа, который является наименьшим возможным графом.

Конечный граф

Если количество вершин и ребер в графе ограничено, граф называется конечным графом.

Бесконечный графикк

Если число вершин и ребер в графе бесконечно, то граф называется конечным.

График с указаниями

Орграфы — это еще один термин для ориентированных графов. Граф называется ориентированным графом или орграфом, если все ребра, соединяющие любые его вершины или узлы, направлены или имеют определенное направление. Под направленными ребрами мы подразумеваем ребра графа, имеющие направление, указывающее, где они начинаются и где заканчиваются.

Простой график

Каждая пара узлов или вершин в простом графе имеет только одно ребро, соединяющее их. Как следствие, только одно ребро соединяет две

вершины, иллюстрируя взаимодействия один к одному между двумя компонентами.

Несколько графиков

Когда в графе G= (V, E) имеется много ребер, соединяющих две вершины, граф называется мультиграфом. Мультиграф не имеет петель.

Полный график

Граф является полным, если он является простым графом. Ребра, имеющие n вершин, должны быть связаны. Он также известен как полный граф, поскольку степень каждой вершины должна быть n-1.

Псевдограф

Псевдограф — это граф, который в дополнение к другим ребрам имеет петлю.

Регулярный график

Регулярный граф — это одна из таких категорий типа графа, которая является простым графом с точно таким же значением степени в каждой из вершин. В результате каждый граф в целом является регулярным графом.

Двудольный граф

Двудольные графы можно разделить на две непустые непересекающиеся части с одинаковым множеством вершин. V1(G) и V2(G) так, что каждое ребро e графа E(G) имеет один конец в V1(G), а другой конец в V2(G) (G). Двудольность графа G относится к разбиению V1 U V2 = V.

Взвешенный график

Размеченный или взвешенный граф — это граф, в котором каждое ребро имеет значение или вес, выражающий затраты на пересечение этого ребра.

Связанный граф

Граф является связанным, если существует путь, соединяющий одну вершину структуры данных графа с любой другой вершиной.

Несвязанный граф

Если вершины не соединены ребром, то нулевой граф называется несвязным графом.

Циклический граф

Граф называется циклическим, если он имеет хотя бы один цикл.

Ациклический граф

Граф называется ациклическим, если он не содержит циклов.

Ациклический направленный граф

Это тип структуры данных графа, который имеет направленные ребра, но не имеет цикла, и его также называют DAG. Полная форма DAG — это направленный ациклический граф. Поскольку он направляет вершины и поддерживает определенные данные, он изображает ребра с упорядоченной парой вершин.

Подграф

Подграф — это набор вершин и ребер одного графа, которые являются подмножествами другого.

Расширенные структуры данных

Расширенные структуры данных можно определить как специальные и уникальные методы, используемые для хранения и организации информации более компетентным образом, при котором информацию можно легко сохранять, извлекать или изменять для использования в областях компьютерной науки и программирования.

Это структуры, которые предлагают лучшие средства хранения и использования информации по сравнению с базовыми типами, такими как списки или массивы. Использование расширенной структуризации данных также позволяет программистам улучшить уровень своих алгоритмов, а также приложений, поскольку становится возможным решение сложных проблем с данными.

Линейные структуры данных

Элементы данных в линейной структуре данных связаны друг с другом в последовательном расположении, при этом каждый элемент связан с элементами перед ним и за ним. Таким образом, один проход может пройти через структуру. Линейные структуры данных состоят из четырех типов. Это:

- Куча
- Множество
- Очередь
- Связанный список

Куча

Линейная структура данных хранит элементы данных в порядке «первым пришел/последним ушел» или «последним пришел/первым ушел». Эти порядки известны как порядки FILO и LIFO соответственно. Используя Stack, элемент можно добавлять и удалять одновременно с одного и того же конца.

Множество

Это набор похожих типов данных, которые хранятся в смежных ячейках памяти. Массивы также используются в Python. Массивы работают в масштабе от 0 до (n-1), где 'n' обозначает размер массива. Массивы бывают двух типов. Это:

- Одномерный массив
- Многомерный массив

Очередь

Очередь — это линейная структура данных, которая следует порядку FIFO. FIFO означает First In and First Out (первым пришел — первым вышел). Порядок таков, что элементы, которые вставлены первыми , должны быть удалены первыми. Свойства структуры данных Queue:

- Вставка элемента
- Удаление элемента
- Время доступа.

Связанный список

Связанные списки разделяют структуры данных, которые хранятся последовательно. Последний узел структуры данных будет связан с первым узлом следующей структуры данных. Первый элемент любой структуры данных известен как Голова списка. Связанный список помогает в распределении памяти, хранит данные во внутренней структуре и т. д. Существует три типа связанных списков. Это:

- Односвязный список
- Двусвязный список
- Круговой связанный список

Нелинейные структуры данных

Структура данных, в которой элементы данных расположены случайным образом. Элементы не расположены последовательно. Элементы данных присутствуют на разных уровнях. В нелинейных структурах данных существуют разные пути для элемента, чтобы достичь другого элемента. Элементы данных в нелинейных структурах данных связаны с одним или несколькими элементами. Существует два типа нелинейных структур данных. Это:

- Древовидная структура данных
- Структура графических данных

Древовидная структура данных

Древовидные структуры данных полностью отличаются от массивов, стеков, очередей и связанных списков. Древовидные структуры данных являются иерархическими. Древовидная структура данных собирает узлы вместе, чтобы изобразить и стимулировать последовательность. Древовидная структура данных не хранит данные последовательно. Она хранит данные на нескольких уровнях. Верхний узел древовидной структуры данных известен как корневой узел. Любой тип данных может храниться в корневом узле. Каждый узел должен обязательно содержать данные. Ветви в древовидной структуре данных известны как дочерние элементы.

Различные части древовидной структуры данных:

- Корневой узел
- Дочерний узел
- Край
- Братья и сестры
- Листовой узел
- Внутренние узлы
- Высота дерева
- Степень узла

Структура графических данных

В графовой структуре данных один узел просто соединен с другим узлом через ребро графа. Графовая структура данных, очевидно, использует нелинейные структуры данных, которые не расположены последовательно.

Графовые структуры данных состоят из ребер и узлов, представленных E и V соответственно. Графовые структуры данных не имеют корневых узлов. У них нет стандартного порядка расположения данных. Каждое дерево также известно как граф с n-1 ребрами, где «n» представляет общее количество вершин в графе. Существуют различные категории в графах, такие как ненаправленные, невзвешенные, направленные и взвешенные.

Различные части графика выглядят следующим образом.

1. Вершина
2. Края
3. Направленный край
4. Ненаправленный край
5. Утяжеленное ребро
6. Степень
7. Входящая степень
8. Outdegree

Реализация физических представлений абстрактных типов данных использует структуры данных. При создании эффективного программного обеспечения структуры данных являются ключевым компонентом. Они также необходимы для проектирования алгоритмов и использования этих алгоритмов в программном обеспечении. Структуры данных используются в различных аспектах, таких как,

Хранение данных

При предоставлении набора атрибутов и соответствующих структур, которые будут использоваться для хранения записей в системе управления базами данных, структуры данных используются для эффективного сохранения данных.

Управление ресурсами и услугами

Структуры данных, включая связанные списки для распределения памяти, управления каталогами файлов и деревьями структуры файлов, а также очереди планирования процессов, используются для обеспечения основных ресурсов и функций операционной системы (ОС).

Обмен данными

Информация, которой обмениваются приложения, например пакеты TCP/IP , организуется с использованием структур данных.

Упорядочивание и сортировка

Двоичные деревья поиска, иногда называемые упорядоченными или сортированными двоичными деревьями, представляют собой структуры данных, которые предлагают практические способы сортировки вещей, например, строк символов, используемых в качестве тегов. Программисты могут управлять объектами, расположенными в заданном приоритете, используя структуры данных, такие как очереди приоритетов.

Индексация

Для индексации элементов, например тех, которые хранятся в базе данных, используются еще более сложные структуры данных, такие как B-деревья.

Идет поиск

В-деревья, хэш-таблицы и двоичные деревья поиска — это стандартные методы, используемые для создания индексов, которые ускоряют процесс поиска определенного элемента.

Масштабируемость

Структуры данных используются приложениями больших данных для распределения и управления хранилищем данных на распределенных сайтах хранения, обеспечивая производительность и масштабируемость. Чтобы упростить запросы, несколько сред программирования больших данных, таких как Apache Spark, предлагают структуры данных, которые копируют фундаментальную структуру записей базы данных.

Выбор структуры данных

Ниже приведены шаги, которые необходимо выполнить при выборе структуры данных.

1) Первым шагом в определении основных операций, которые должны поддерживаться, является анализ проблемы. Вставка элемента данных в структуру данных, удаление элемента данных из структуры данных и поиск определенного элемента данных являются примерами основных операций.

2) Определите ограничения ресурсов для каждой операции и дайте им количественную оценку.

3) Определите, какая структура данных лучше всего соответствует этим требованиям.

Поддерживаемые операции:

Если базовый тип данных атрибута может быть переведен в один из видов, для которых поддерживается операция, процессы между типами данных, не включенными в таблицу, могут быть выполнены. Данные могут иметь числа, которые добавляются или вычитаются из них. Количество дней, которые должны быть добавлены или удалены, представлено целыми числами.

Вычислительная сложность:

Вычислительная сложность — это метрика того, сколько времени и памяти (ресурсов) использует конкретный алгоритм при его выполнении. Перед разработкой кода специалисты по информатике могут спрогнозировать время выполнения алгоритма и потребности в памяти, используя математические метрики сложности. Для программистов, реализующих и выбирающих алгоритмы для практического применения, эти прогнозы являются важными ориентирами.

Элегантность программирования:

Одной из таких вещей, которую легко распознать, но трудно определить, является элегантная программа. Она эффективно использует слова, не прибегая к запутыванию. Она лаконична, не используя сложный код. Она достигает баланса между концепциями простоты и ясности, сложностью кодирования, оставаясь поверхностной для чтения и понимания.

Написание кода, эквивалентного безупречному письму, является конечной целью каждого программиста.

Нет "волшебного решения" или единственного решения этой проблемы. Стандарты кодирования могут быть использованы для помощи, но они должны быть основаны на прочной структуре, которая гарантирует, что суть вопроса будет донесена до программиста и отражена в коде.

Типы данных и их связь со структурами данных

Чтобы ответить на вопрос, что такое структура данных, необходимо понять три основных типа данных.

Абстрактный.

Абстрактные данные определяются тем, как они себя ведут. Этот тип охватывает графы, очереди, стеки и наборы.

Композитный (или составной).

Составные данные включают в себя комбинированные примитивные типы данных и включают массивы, классы, записи, строки и структуры. Они также могут состоять из других составных типов.

Примитивный.

Примитивные данные классифицируются как базовые данные и состоят из логических значений, символов, целых чисел, указателей, а также чисел с фиксированной и плавающей точкой.

Эти типы данных являются строительными блоками структур данных. Типы данных сообщают интерпретатору или компьютеру, как программист планирует использовать данные. Кроме того, аналитики данных могут выбирать из различных классификаций структур данных. Хитрость заключается в том, чтобы выбрать структуру, наиболее подходящую для ваших потребностей и ситуации.

Классификации структуры данных

Существуют различные типы и классификации структур данных и самих данных, как мы только что увидели. Этот объем информации порождает еще больше вопросов. Что такое связанный список? Что такое линейная структура данных? Что такое структура данных?

Давайте попробуем разобраться в структурах данных, взглянув на классификации. Существует три основные классификации структур данных, каждая из которых состоит из пары характеристик.

Линейная и нелинейная структура данных

Линейные структуры упорядочивают данные в линейной последовательности, например, в массиве, списке или очереди. В нелинейных структурах данные не образуют последовательность, а вместо этого соединяются с двумя или более элементами информации, например, в дереве или графике.

Статическая и динамическая структура данных

Как следует из термина, статические структуры состоят из фиксированных, постоянных структур и размеров во время компиляции.

Массив резервирует заданный объем резервной памяти, установленный программистом заранее. Динамические структуры характеризуются нефиксированными объемами памяти, сжимаясь или расширяясь в зависимости от требований программы и ее выполнения. Кроме того, расположение связанной памяти может меняться.

Однородная и неоднородная структура данных

Однородные структуры данных состоят из одного и того же типа элементов данных, например, коллекции элементов, найденные в массиве. В неоднородных структурах данные не обязательно должны быть одного и того же типа, например, структуры.

Операции со структурой данных

Ниже приведены наиболее частые операции, которые можно выполнять над структурами данных:

1. Поиск - Поиск подразумевает нахождение определенной части внутри указанной структуры данных. Когда нужный ингредиент обнаружен, это называется успехом. Поиск - это операция, которая может быть выполнена над структурами данных, такими как массивы, связанные списки, деревья, графы и т. д.
2. Сортировка. Сортировка — это процесс упорядочивания всех элементов данных в структуре данных в определенном порядке, например, по возрастанию или по убыванию.
3. Вставка подразумевает добавление новых элементов данных в структуру данных.
4. Элементы данных в структуре данных могут быть удалены.
5. Обновление — мы можем обновить или заменить существующие части структуры данных.

Сложность времени и пространства

Временная сложность — это изучение того, как потребление ресурсов алгоритмом меняется в зависимости от размера его входных данных. В широком смысле, это касается того, как количество операций алгоритма масштабируется с ростом его входных данных, что позволяет вам оценить его эффективность с точки зрения требуемой операционной работы.

Напротив, сложность пространства имеет дело с памятью, занимаемой алгоритмом во время его выполнения. Она состоит из пространства для входных значений и некоторого дополнительного умеренного пространства памяти, используемого во время операции. Понимание сложности пространства важно для разработчиков, поскольку оно дает им способ оценить объем использования памяти, который потребуется алгоритму, особенно если он включает в себя большие объемы данных.

Применение структур данных

Структуры данных имеют множество применений, таких как:

Хранение данных

Структуры данных способствуют эффективному сохранению данных, например, задавая наборы атрибутов и соответствующие структуры, используемые в системах управления базами данных для хранения записей.

Обмен данными

Организованная информация, определяемая структурами данных, может совместно использоваться приложениями, например пакетами TCP/IP.

Управление ресурсами и услугами

Структуры данных, такие как связанные списки, могут позволить основным ресурсам и службам операционной системы выполнять такие функции, как управление каталогами файлов, распределение памяти и обработка очередей планирования.

Масштабируемость

Приложения больших данных используют структуры данных для управления и распределения хранилища данных по многим распределенным местам хранения. Эта функция гарантирует масштабируемость и высокую производительность.

Преимущества структур данных

1. Структура данных способствует эффективному хранению данных на запоминающих устройствах.
2. Использование структур данных упрощает извлечение данных из устройства хранения.
3. Структура данных позволяет эффективно и результативно обрабатывать как небольшие, так и большие объемы данных.
4. Манипулирование большими объемами данных становится простым, если использовать правильную технику структурирования данных.
5. Использование хорошей структуры данных может помочь программисту сэкономить много времени или времени на обработку при выполнении таких задач, как хранение, извлечение или обработка данных.
6. Большинство хорошо организованных структур данных, включая стеки, массивы, графы, очереди, деревья и связанные списки, имеют хорошо построенные и заранее спланированные подходы для таких операций, как хранение, добавление, извлечение, изменение и удаление. Программист может полностью полагаться на эти факты при их использовании.
7. Структуры данных, такие как массивы, деревья, связанные списки, стеки, графы и т. д., тщательно проверены и доказаны, поэтому любой может использовать их напрямую без необходимости изучения и разработки. Если вы решите разработать собственную структуру данных, вам, возможно, придется провести некоторое исследование, но это почти наверняка будет связано с решением более сложной проблемы, чем та, которую они могут предоставить.
8. В долгосрочной перспективе использование структуры данных может просто способствовать повторному использованию.

Как выбрать правильную структуру данных?

Вот как выбрать правильную структуру данных для вашей программы, которая действительно подойдет вам.

Поймите потребности вашей программы

Первый шаг — понять, каковы требования программы. Спланируйте операции, которые она будет выполнять, и функции, которые она будет предоставлять. Например, если вы работаете с большим объемом информации, которую нужно извлекать практически мгновенно, вы можете использовать хэш-таблицу. Напротив, если для вас важен порядок, вы можете поискать древовидную структуру.

Оценить требования к производительности

Далее поговорим о производительности. Вам нужна структура данных, которая может быстро обрабатывать операции. В мире компьютерных наук мы используем нотацию Big O для измерения эффективности. Структура с временной сложностью $O(n)$ обычно будет работать лучше, чем структура с $O(n^2)$, особенно по мере роста объема данных.

Оцените время обработки

Также важно учитывать, сколько времени занимает обработка данных в структуре. Временная сложность имеет большое значение, особенно в таких областях, как машинное обучение , где метрики и уровень производительности могут действительно зависеть от того, насколько эффективна вся архитектура.

Учитывайте простоту использования

Еще одна область, на которую стоит обратить внимание, это то, насколько легко использовать структуру данных. Она должна быть простой и интуитивно понятной. Она должна быть понятной и простой, поскольку в ходе фактической разработки могут возникнуть временные и внешние ошибки.

Упростите удаление данных

Еще один момент, который следует рассмотреть, — насколько легко удалять данные из структуры. Некоторые параметры, такие как связанные списки, могут сделать удаление немного сложным. Убедитесь, что выбранная вами структура данных допускает прямое удаление при необходимости.

Лекция 15 Массивы указателей и структуры

Представление списков массивами

Использование указателей не единственный способ представления списков. Например, если язык программирования не позволяет использовать указатели и допускает только работу с массивами, то двусвязный список можно получить на основе, скажем, трех массивов (моделировании указателей с помощью курсоров, т.е. целых чисел, которые указывают на

позиции элементов в массивах). В первом массиве могут храниться значения элементов списка, во втором и третьем – индексы предыдущего и последующего элементов списка.

Нелинейные разветвленные списки

Нелинейным разветвленным списком является список, элементами которого могут быть тоже списками. Если один из указателей каждого элемента списка задает порядок, обратный к порядку, устанавливаемому другим указателем, то такой двусвязный список будет линейным. Если же один из указателей задает порядок произвольного вида, не являющийся обратным по отношению к порядку, устанавливаемому другим указателем, то такой список будет нелинейным. При обработке нелинейный список определяется как любая последовательность атомов и списков (подсписков), где в качестве атома берется любой объект, который при обработке отличается от списка тем, что он структурно неделим.

Разветвленные списки описываются тремя характеристиками:
порядком, глубиной и длиной.

Типичный пример применения разветвленного списка – представление алгебраического выражения в виде списка. Алгебраическое выражение можно представить в виде последовательности элементарных двухместных операций вида:

< операнд 1 > < знак операции > < операнд 2 >.

Выражение

(a + b)*(c - (d/e)) + f

будет вычисляться в следующем порядке:

a + b; d/e; c - (d/e);

(a + b)*(c – d/e); (a + b)*(c – d/e) + f.

При представлении выражения в виде разветвленного списка каждая тройка «операнд-знак-операнд» представляется в виде списка, причем, в качестве операндов могут выступать как атомы – переменные или константы, так и подсписки такого же вида. На рисунке 65 представлена схема списка алгебраического выражения.

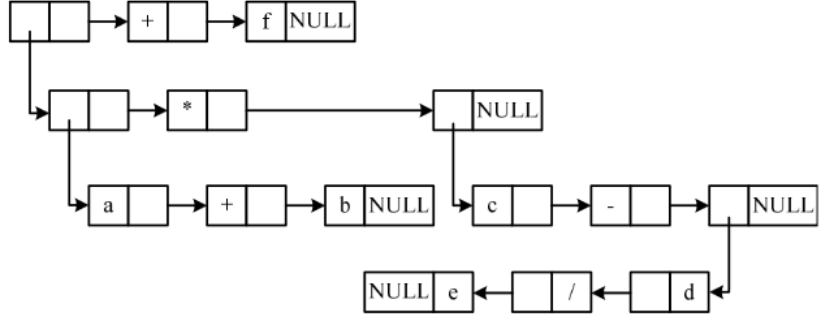


Рисунок 65 - Схема вычисления выражения на основе списка

Глубина этого списка равна 4, длина 3. Скобочное представление данного выражения будет иметь вид:

(((a,+,b),*,(c,-,(d,/,e)),+,f).

Реализация стеков на основе массивов

Стек с максимальным объемом max_size элементов можно реализовать в виде массива, например STACK[max_size], и индекса последнего заполненного элемента массива top. Таким образом, значение top = -1, соответствует пустому стеку:

```
const int max_size = 100; //размер стека
int STACK[max_size]; //стек
int top = -1; //вершина стека
```

Протестировать стек на наличие в нем элементов можно с помощью операции IsStackEmpty(), которая сводится к проверке значения индекса top:

```
//---------------------------- проверка на пустоту
bool IsStackEmpty(int top)
{ return (top < 0); }
```

Если значение top превосходит max_size, то стек переполняется:

```
//---------------------------- проверка на переполнение
bool IsStackFull(int top)
{ return (top >= max_size - 1); }
```

Операции добавления и извлечения элемента требуют предварительной проверки возможности выполнения операций. Операция извлечения элемента состоит в выборке значения, на которое указывает указатель стека, и модификации указателя стека (в направлении, обратном при включении). После выборки слот, в котором размещался выбранный элемент, считается свободным. При добавлении элемента в стек указатель модифицируется таким образом, чтобы он указывал на следующий свободный элемент. Затем элемент записывается на место, определяемое указателем стека:

```
//--------------------------- извлечь элемент из стека
int Pop (int &top, int STACK[])
{    int x = 0;
     if (!isStackEmpty(top)) x = STACK[top--];
     return x;
}
//--------------------------- добавить элемент в стек
bool Push (int STACK[], int &top, int NewItem)
{    bool x = true
     if (x = !isStackFull()) STACK [++top] = NewItem;
     return x;
}
```

Любая из рассмотренных операций имеет сложность O(1). Основной недостаток данной реализации стека – статичность: до использования массива необходимо знать его максимальный размер, чтобы распределить под него оперативную память. Данный недостаток не является недостатком стека. Это результат выбора реализации стека на базе массива.

Реализация очереди на основе массива

Очередь с максимальным объемом max_size элементов можно реализовать в виде массива – QUEUE[max_size]. Для организации очереди необходимы две индексные переменные Head, указывающие на первый

элемент очереди, и Tail, указывающая позицию, в которую будет добавляться элемент.

```
const int max_size = 100; //размер очереди
int QUEUE[max_size]; //циклическая очередь
int Head; //индекс начала очереди
int Tail; //индекс конца очереди
```

Чтобы извлечь элемент, он удаляется из начала (Head) очереди, после чего индекс Head увеличивается на единицу. Чтобы добавить элемент, он добавляется в конец (Tail) очереди, а индекс Tail увеличивается на единицу. По мере добавления элементов в очередь ее конец будет продвигаться к концу массива, то же самое будет происходить с началом при их извлечении. Выход из создавшегося положения – зациклить очередь, то есть считать, что за последним элементом массива следует опять первый.

Подобный способ организации очереди на массиве еще иногда называют циклической (или кольцевой) очередью. Элементы очереди будут расположены в последовательных слотах QUEUE[Head] , QUEUE[Head+1] … QUEUE[Tail-1], которые циклически замкнуты. Циклическая очередь устроена так, что при достижении конца массива осуществляется переход на его начало, то есть слот 1 следует сразу же после слота с номером max_size. Определение размера очереди состоит в вычислении разности индексов с учетом циклической природы очереди.

Первоначально Head = Tail = 0 . Равенство этих двух индексов (при любом их значении) Head == Tail является признаком пустой очереди. Тогда, при попытке удалить из нее элемент происходит ошибка опустошения.

Если в процессе работы с циклической очередью число операций добавления превышает число операций исключения, то может возникнуть ситуация, в которой индекс конца «догонит» индекс начала.

Это ситуация заполненной очереди, но если индексы сравняются, то ситуация будет неотличима от ситуации пустой очереди. Для различения этих двух ситуаций к циклической очереди предъявляется требование, чтобы между индексами конца и начала оставался зазор из свободных элементов. Когда этот зазор сокращается до одного элемента, очередь считается заполненной. Таким образом, если Head == (Tail + 1) % max_size), то очередь заполнена, и попытка добавить в очередь элемент приводит к переполнению. Рассмотрим функции работы с очередью: очистки Clear() и добавления Enqueue():

```
//------------------------ очистить очередь
void Clear()
     { Head = Tail = 0; } //обнулить индексы начала и конца
//------------------------ добавить в конец очереди
int Enqueue (int NewItem)
{
     int next;
     if ((next = (Tail + 1) % max_size) == Head)
          //если переполнение очереди
          return 0; //ошибка
```

```
        QUEUE[Tail] = NewItem;
        //добавить элемент в конец очереди
        Tail = next;
        //индекс конца очереди переместить на следующий
        return 1;
}
```

Функция извлечения `Dequeue()` будет выглядеть следующим образом:

```
//------- извлечь из начала очереди
int Dequeue ()
{
        int Item;
        if (Head == Tail) //если очередь пуста
                return 0; //ошибка
        else
        {       Item = QUEUE[Head++]; //извлечь элемент из начала
                Head %= max_size; //по достижении Head == max_size
                //индекс начала очереди сбросить в 0
                return Item; //вернуть извлеченный элемент
        }
}
```

Следующий вариант реализации очереди во многом похож на предыдущий, за исключением того, что очередь описывается структурой и массив для хранения элементов задается динамически в соответствии с запрашиваемым максимальным (не фактическим) размером `Size`:

```
struct QUEUE //циклическая очередь
{
        int Head; //индекс начала очереди
        int Tail; //индекс конца очереди
        int Size; //размер очереди
        int* Data; //данные очереди
        QUEUE(int size){
                Head = Tail = 0; //очередь пуста
                Data = new int[Size = size+1];
                //выделить память под данные
        };
        bool isFull(){
                return     (Head%Size     ==     (Tail+1)%Size);
        //переполнение
        };
        bool isEmpty(){
                return (Head%Size == Tail%Size);}; //очередь пуста
        };
}
```

Рассмотрим последовательность функций для манипуляции с данными очереди, лежащих в основе ее абстрактного представления.

Конструктор очереди:

```
//---------------- выделить ресурс для очереди размера n
Queue CreateQueue(int n)
{ return *(new QUEUE (n)); };
Добавление нового элемента в очередь:
//------------------ добавить NewItem в конец очереди q
bool Enqueue(QUEUE& q, int NewItem)
```

```
		{
			bool x = true;
			if  (x  =  !q.isFull())  //если  нет  переполнения
		очереди
				{q.Data[q.Tail] = NewItem;
				//добавить новый элемент в конец очереди
				q.Tail=(q.Tail+1)%q.Size;    //индекс    конца
			очереди
				//увеличить и скорректировать (циклическая)
				}
			return x;
			};
```

Функция извлечения элемента:

```
		//----------------- извлечь элемент из начала очереди
		int Dequeue(QUEUE& q)
		{
			int x = 0;
			if (!q.isEmpty()) //если очередь не пуста
		{
		x = q.Data[q.Head];
		//считать данные с начала очереди
		q.Head=(q.Head+1)%q.Size;
		//скорректировать индекс начала
		}
		return x; //вернуть элемент
	}
```

Функция чтения элемента без извлечения:

```
	//----------------------- прочитать из начала очереди
	int Peek(const QUEUE& q)
	{
		int x = 0;
		if (!q.isEmpty()) //если очередь не пуста
			x = q.Data[q.Head];
		//прочитать элемент из начала очереди
		return x; //вернуть элемент
	}
```

Функция очистки очереди сводится к записи одного и того же (не обязательно начального) значения в оба индекса:

```
	//---------------------------- очистить очередь
	void ClearQueue(QUEUE& q)
	{ q.Tail = q.Head = 0; //обнулить индексы очереди
	}
```

Функция удаления очереди сводится к освобождению памяти, занимаемой массивом, и установке индексов в ноль:

```
	//----------------------- освободить ресурсы очереди
	void ReleaseQueue(QUEUE & q)
	{
		delete[] q.Data; //вернуть память
		q.Size = 1;
		q.Head = q.Tail = 0; //сбросить индексы
	}
```

Для очереди операции извлечения и добавления имеют постоянное время выполнения O(1).

Лекция 16 Динамические линейные структуры

Список как динамическая структура данных

Динамическая структура данных представляет собой множество переменных, связанных между собой указателями. Каждый элемент такой структуры содержит один или несколько указателей на аналогичные элементы. Например:

```
struct list
{
	int value;
	list *next; //указатель
	list *next,*pred; //два указателя
	list *links[10]; //ограниченное кол-во указателей
	list **plinks; //произвольное кол-во указателей
};
```

Из данного описания нельзя определить ни количество переменных в структуре данных, ни характер связи между ними (последовательный, циклический, произвольный). Следовательно, конкретный тип динамической структуры данных (список, дерево, граф) зависит от функций, которые работают с этой структурой.

Рассматриваемые ниже структуры данных являются динамическими по двум причинам: сами переменные таких структур создаются как динамические переменные; количество связей между переменными и их характер также определяются динамически в процессе работы программы.

Последовательность обхода зависит не от физического размещения элементов в памяти, а от последовательности их связывания указателями. Нумерация элементов списка – логический номер элемента в списке, номер, получаемый им в процессе движения по списку.

Динамические структуры данных доступны, как правило, через указатель на некоторый ее элемент, который называется заголовком.

Наиболее простыми динамическими структурами данных являются списки. Список представляет собой линейную последовательность элементов, каждый из которых содержит указатели на аналогичные элементы (соседи). Элементы определяются ссылками на узлы, поэтому связные списки иногда называют самоссылочными структурами.

Основное преимущество связных списков перед массивами заключается в возможности эффективного изменения расположения элементов. За эту гибкость приходится жертвовать скоростью доступа к произвольному элементу списка, поскольку единственный способ получения элемента состоит в отслеживании связей от начала списка.

Списки подразделяют на:

• односвязные – каждый элемент списка имеет указатель на следующий (рисунок 66);

• двусвязные – каждый элемент списка имеет указатель на следующий и на предыдущий элементы;

• двусвязные циклические – первый и последний элементы списка ссылаются друг на друга (рисунок 67).

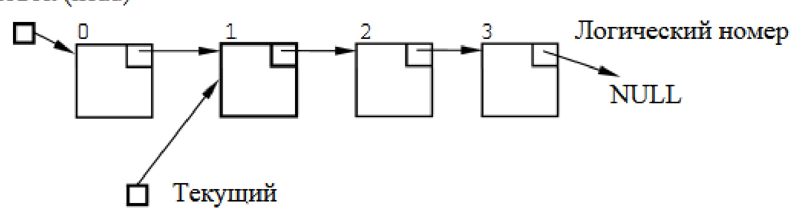


Рисунок 66 - Односвязный список

Односвязные списки являются наиболее простыми. Основным их недостатком является возможность просмотра только в одном направлении – от начала к концу. Без дополнительных «ухищрений» нельзя получить указатель на предыдущий элемент списка, который необходим при удалении текущего. Для односвязного списка наиболее простыми являются операции включения и исключения элементов в начале и конце списка, соответственно, они используются для моделирования таких структур данных, как стеки и очереди.

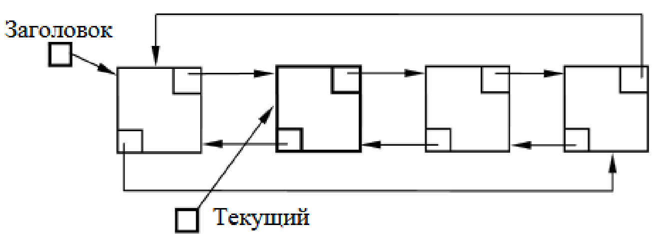


Рисунок 67 - Двусвязный список

Двусвязные списки дают возможность просмотра элементов в обоих направлениях и являются наиболее универсальными. Операции включения-исключения элементов имеют примерно одинаковый уровень сложности. Они используются для создания цепочек элементов, которые допускают частые операции включения, исключения, упорядочения, замены и прочие.

В односвязных и двусвязных списках последний элемент содержит указатель NULL для обозначения факта окончания последовательности. Аналогично первый элемент двусвязного списка содержит указатель NULL на предыдущий элемент. В этом случае работа с первым и последним элементом списка имеет свои особенности. В качестве альтернативы может быть предложен циклический список, у которого последний элемент ссылается на первый, а первый – на последний.

Даже если данная замкнутая структура используется для представления обычной линейной последовательности, работающие с ним функции являются более простыми.

Все эти структуры данных предполагают различные способы организации множества переменных. При этом обычный массив и список

имеют противоположные свойства: если массив допускает произвольный доступ к своим элементам и ускоренный (двоичный) поиск при их упорядоченности, то список допускает только последовательный просмотр. Следовательно, массив обладает преимуществами с точки зрения операций поиска. Но, с другой стороны, изменение порядка следования переменных в массиве сопровождается их физическим перемещением, а в списке приводит только к «переброске» соответствующих указателей. Таким образом, списки предпочтительнее для структур данных, в которых операции поиска встречаются значительно реже, чем операции по изменению порядка следования элементов, и наоборот. В отношении массивов указателей и списков можно заметить, что массивы указателей критичны к количеству элементов в структуре данных – при увеличении этого количества их необходимо расширять, при уменьшении – желательно сокращать. Для списков такой проблемы не существует. Таким образом, списки предпочтительнее в структурах, где количество элементов меняется в широких пределах, массивы указателей – в противном случае.

Работа со списками

Элемент списка состоит из информационной части – ключа и данных, и полей связок – указателей на аналогичные элементы.

Пример структуры двусвязного списка будет выглядеть следующим образом:

```
//-----------------list.h
struct Element //элемент списка
{
	void* Data; //данные
	int Key; //ключ
	Element* Prev; //указатель на предыдущий элемент
	Element* Next; //указатель на следующий элемент
}
```

В таком списке каждый элемент содержит два указателя, один (`prev`) указывает на предыдущий элемент, а другой (`next`) – на следующий, и два информационных поля: `Key` – ключ и `Data` – данные.

В процессе работы со списками будет получено множество экземпляров этой структуры, по одному для каждого элемента. Как только возникает необходимость использовать новый узел, для него следует зарезервировать память.

В языке С++ принято инициализировать область хранения, а не только выделять для нее память. В связи с этим обычно в каждую описываемую структуру включается конструктор (`constructor`). Конструктор представляет собой функцию, которая описывается внутри структуры и имеет такое же имя. Он предназначен для предоставления исходных значений данным структуры. Для этого конструкторы автоматически вызываются при создании экземпляра структуры. Например, если описать элемент списка при помощи следующего кода:

```
struct Element //элемент списка
{
```

```
        void* Data; //данные
        int Key; //ключ
        Element* Prev; //указатель на предыдущий элемент
        Element* Next; //указатель на следующий элемент
        //конструктор
        Element
        (Element* prev, void* data, int key, Element* next)
        {
             Prev = prev;
             Data = data;
             Key = key;
             Next = next;
        }
    }
```

Тогда, оператор

```
Element* A = new Element(NULL,data,key,Head);
```

не только резервирует достаточный для элемента объем памяти и возвращает указатель на него в переменной A, но и присваивает полю `Data` элемента значение `data`, `Key` значение `key`, а указателям поля – значение `prev` и `next`. Конструкторы помогают избегать ошибок, связанных инициализацией данных.

Особенности работы с динамическими структурами данных заключаются в частом использовании операции косвенного обращения по указателю к структуре или к ее элементу – * или ->. При этом обычно используется указатель, который ссылается на текущий элемент структуры данных. Весь просмотр структуры данных заключается в циклическом переходе от одного текущего элемента к другому.

Для успешной работы со списками необходимо прежде всего научиться интерпретировать средствами языка такие понятия, как предыдущий, текущий, последующий, первый, последний элементы, переход к предыдущему, следующему и т.д.

Интерпретация перемещений указателей

При работе со списками каждый указатель имеет определенный смысл – первый, текущий, следующий, предыдущий и т.п. элементы списка. Поля `prev`, `next` также интерпретируются как указатели на предыдущий и следующий элементы списка, доступные через указатели. Например, если `p` – указатель на новый элемент, а `q` – указатель на текущий, то выражение `q->prev->next = p` интерпретируется как «указателю на следующий элемент списка в предыдущем от текущего присвоить указатель на новый».

Операция перемещения указателя реализуется через операцию присваивания одному указателю значения другого. На рисунке 68 это соответствует перенесению (копированию) требуемого указателя из одной ячейки в другую. В левой части операции присваивания должно находиться обозначение ячейки, в которую заносится новое значение указателя. Причем ячейка может быть достижима, только через имеющиеся рабочие указатели. Этому соответствует цепочка операций `q->prev->next`. В правой части

операции присваивания должно находиться обозначение ячейки, из которой берется значение указателя – p.

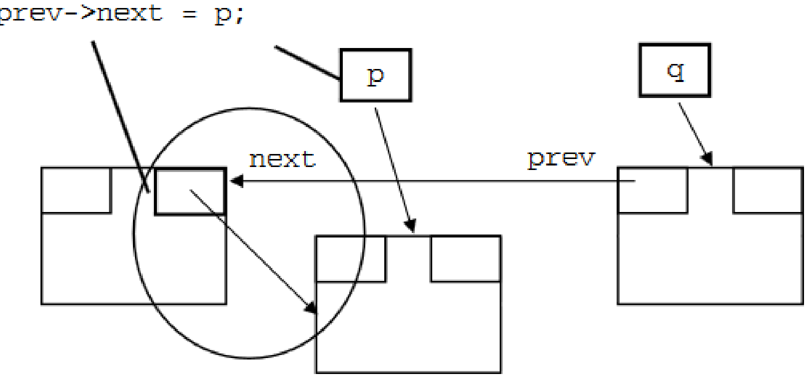


Рисунок 68 - Схема присваения укзателя

При работе со списками часто используется соглашение: в начале каждого списка содержится фиктивный узел, называемый началом или заголовком списка (head node). Поле элемента ведущего узла игнорируется, но ссылка узла сохраняется в качестве указателя узла, содержащего первый элемент списка.

```
Element* Head;
Head = NULL;
```

Ситуация, в которой удобно использовать начальный элемент, возникает, когда необходимо передать спискам указатели в качестве аргументов функций. Эти аргументы могут изменять список таким же образом, как это выполняется для массивов. Использование ведущего узла позволяет функции принимать или возвращать пустой список. При отсутствии ведущего узла функция нуждается в механизме информирования вызывающей функции в случае, когда оставляется пустой список. Одно из решений для C++ состоит в передаче указателя на список как ссылочного параметра. Второй механизм предусматривает прием функциями обработки списков указателей на списки, ввода в качестве аргументов и возврат указателей на списки вывода.

Рассмотрим интерфейсные функции работы с двусвязным списком. Структура List объявляет набор функций, которые реализуют базовый список операций, что позволяет избегать повторения кода и зависимости от деталей реализации:

```
/-----------------list.h---------------------
struct List
{
     Element* Head;
     List(){Head = NULL;};
     Element* GetFirst(); //получить первый элемент списка
     Element* GetLast(); //получить последний элемент списка
     Element* Search (void* data);
     //найти первый элемент по значению
     void PrintList(void(*f)(void*));
     //f обработка элементов списка
     int CountList(); //подсчет количества элементов списка
     bool Insert(void* data, int key);
```

```
        //добавить элемент по ключу
        bool InsertEnd(void* data,int key);
        //добавить эл. по ключу в кон.
        bool Delete(Element* e); //удалить первый по ссылке
        bool Delete(void* data); //удалить первый по значению
        bool Sort(); //сортировать
        bool isListEmpty(); //проверка на пустоту
        bool DeleteList(); //очистить список
}
.............................
int _tmain(int argc, _TCHAR* argv[])
{.............................
        //создание объекта списка в клиентской программе
        List* A = new List();
.............................
}
```

Перемещение по списку

Данные операции являются вспомогательными и используются другими функциями. Чтобы получить первый элемент списка необходимо вернуть – `Head`. Для текущего элемента следующий будет `Next`, а предыдущий `Prev`

```
//-------------------------получить первый элемент списка
Element* List::GetFirst() { return Head; };
//------------------------------получить следующий
Element* List::GetNext() {return this->Next;};
//------------------------------получить предыдущий
Element* List::GetPrev() {return this->Prev;};
//----------------------получить последний элемент списка
```

Для получения последнего элемента списка в функции `GetLast()` организуется движение по указателям, начиная с заголовка `Head`. Если текущий `temp==NULL`, это значит, что достигнут конец списка, и функция возвратит предыдущий перед пустым `NULL` . Для организации перемещения по списку используется функция получения следующего `GetNext()`:

```
Element* List::GetLast()
{
        Element* temp = Head, *x = temp;
        while (temp != NULL) //пока не конец списка
        {
                x = temp;
                temp = temp ->GetNext(); //перейти к следующему
        };
        return x;
}
```

Функция CountList() возвращает размер списка, подсчитывая его элементы:

```
//------------------------------подсчет количества
элементов списка int List::CountList()
{
        Element* temp = Head; //начиная с начала
        int count = 0; //обнулить счетчик
        while (temp != NULL) //пока не конец
```

```
	{
		count ++; //увеличить счетчик
		temp = temp ->GetNext(); //перейти к следующему
	};
	return count;
}
```

Поскольку предполагается, что такая операция не будет частой, лучше затратить здесь дополнительное время, чем терять его на модификацию переменной размера списка при каждом включении - удалении элемента.

Поиск

Функция `Search()` находит в списке с помощью простого линейного поиска первый элемент, имеющий значение data. Точнее говоря, она возвращает указатель на этот элемент или NULL, если элемент не найден:

```
//---------------------найти первый элемент по значению
Element* List::Search (void* data)
{
	Element* temp = Head; //начиная с заголовка
	//пока не конец списка
	//и не найден элемент
	while((temp != NULL) && (temp->Data != data))
		temp = temp ->Next; //перейти к следующему
	return temp; //вернуть указатель на найденный
}
```

Поиск в списке из n элементов требует в худшем случае $\Theta(n)$ операций.

Вставка

Функция вставки `Insert()` добавляет элемент с ключом `key` и данными `data` к текущему списку, помещая его в начало списка.

```
//------------добавить элемент в начало списка по ключу
bool List::Insert (void* data, int key)
{ //вставить в начало
	if (Head == NULL)
	Head = new Element(NULL,data,key,Head);
	else
	Head = (Head->Prev = new Element(NULL, data,key,Head));
	return true;
}
```

Функция выполняется за время O(1) (не зависит от длины списка).

Следующая функция `InsertEnd()` добавляет элемент с ключом `key` и данными data после последнего (в конец), если список не пуст, и в начало при пустом списке:

```
//------------- добавить элемент по ключу в конец списка
bool List::InsertEnd (void* data, int key)
	{ if (Head == NULL) Head = new
		Element(NULL,data,key,Head);
	else {
		Element * temp = GetLast(); //после последнего
	temp ->Next = new Element(temp, data,key,NULL);
	//вставить
	}
	return true;
```

```
}
```

На рисунке 69 приведена графическая интерпретация вставки элемента в конец при непустом списке.

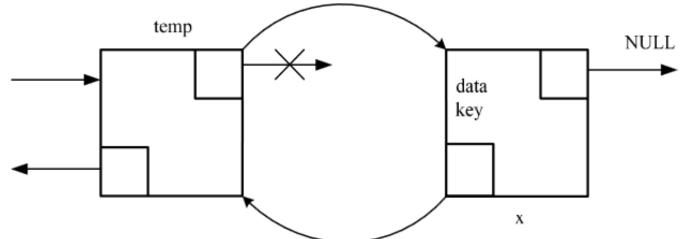


Рисунок 69 - Схема вставки элемента в конец

Удаление

Для операции удаления `Delete()` не требуется дополнительной информации об элементе, предшествующем удаляемому (либо следующим за ним) в списке (как это имеет место для односвязных списков), эта информация содержится в самом элементе. Действительно, главная особенность двусвязных списков состоит в возможности удаления элемента, когда ссылка на него является единственной информацией. В данном примере указатель на удаляемый элемент передается при вызове функции в качестве аргумента:

```
//-------------------------- удалить первый по ссылке
bool List::Delete(Element* x)
{
     if (x!= NULL)
     {
          if (x->Next != NULL) x->Next->Prev = x->Prev; //1
          if (x->Prev != NULL) x->Prev->Next = x->Next; //2
               else { Head = x->Next; Head->Prev=NULL; }
          //удаляем из начала
          delete x; //осободить память
          return true;
     }
     else
          return false;
}
```

Как было сказано выше, указатель элемента предоставляет достаточно информации для удаления узла, что видно из рисунка 70.

Для данного `x` указателю `x->Next->Prev` присваивается значение `x->Prev` (ссылка 1), а указателю `x->Prev->Next` – значение `x->Next` (ссылка 2).

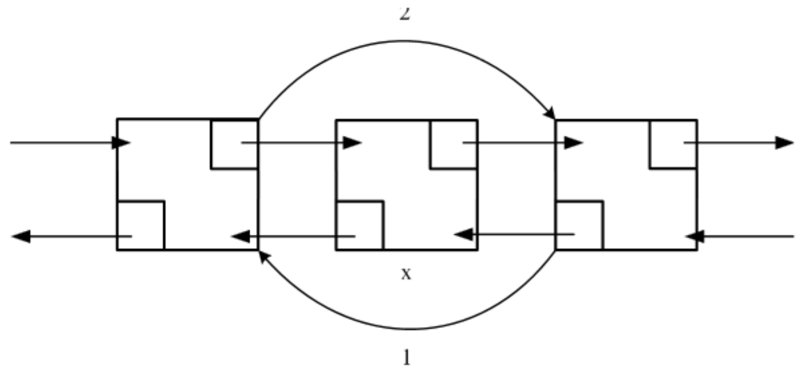


Рисунок 70 - Схема удаления элемента списка

При удалении элемента из списка задача сводится к перераспределению ссылок таким образом, чтобы элемент больше не был привязан к списку. Система утилизирует пространство, занимаемое элементом, чтобы всегда иметь возможность выделять его под новый узел в операторе `new`.

В функции `Delete()` необходимо предусмотреть обработку ошибочных ситуаций, поскольку попытка удаления элемента из пустого списка является довольно распространенной ошибкой и может нарушить структуру динамической памяти, что скажется только впоследствии.

Если задано значение `data` , по которому необходимо удалить элемент, то перед удалением надо найти его указатель с помощью функции `Search()`:

```
//------------------------- удалить первый по значению
bool List::Delete(void* data)
{ return Delete(Search(data)); };
//найти и удалить
```

Так как при добавлении элементов в список под них выделялась память, то при удалении списка память необходимо вернуть системе для повторного использования. Для этого можно поэлементно, начиная с конца списка, удалять элементы и освобождать память до тех пор, пока список не будет пустым. Функция вызывается рекурсивно:

```
//---------------------------- очистить список
bool List::DeleteList()
{
     Element *temp =GetLast(); //начиная с последнего
     if(temp) //пока есть элементы
     {
          if (temp ->Prev != NULL)
               temp ->Prev->Next = temp ->Next;
                    else Head = temp ->Next;
               delete temp; //освободить память
               DeleteList(); //рекурсивный вызов
     }
     return true;
}
```

Вывод списка

Для обработки элементов списка используется функция `Print - List()`, которая для каждого из элементов списка, начиная с заголовка `Head` вызывает функцию `f()` обработки поля `Data`:

```
/------------------------f обработка элеметов списка
void List::PrintList(void(*f)(void*))
{
     Element* temp = Head;
     while (temp != NULL)
     {
          f(temp ->Data);
          temp = temp ->GetNext();
     };
}
```

Проблема концов списка и циклические списки

Основной сложностью при работе со списками является необходимость проверки множества вариантов при выполнении операций над элементом списка: список пустой; элемент единственный; элемент в начале списка; элемент в конце списка; элемент в середине списка.

Для решения проблемы «концов списка» удобно бывает сделать его циклическим, то есть вместо указателей `NULL` записывать:

• указатель на последний элемент в качестве указателя на предыдущий в первом элементе списка;

• указатель на первый элемент в качестве указателя на последующий в последнем элементе списка.

Тогда, единственный элемент в циклическом списке будет иметь указатели на самого себя, а в процессе просмотра, конец списка будет определяться фактом возвращения на его начало.

При выполнении операций над циклическим списком необходимо следить за его заголовком. Например, операция включения в начало списка будет отличаться от включения в конец списка только перемещением заголовка на новый элемент.

Стеки и очереди

Стек

Стек – одномерная структура данных (частный случай связного списка), в которой размещение новых элементов и удаление существующих производится с одного конца, называемого вершиной (top).

В англоязычной литературе для обозначения стеков используется аббревиатура LIFO (Last In First Out: последний вошел – первый вышел).

Начало последовательности называется дном стека, конец последовательности, в который производится добавление элементов и их исключение – вершиной стека. Переменная, которая указывает на последний элемент последовательности в вершине стека – указатель стека. Таким образом, указатель стека `sp` (stack pointer) содержит в любой момент времени индекс (адрес) текущего элемента, который является единственным элементом стека, доступным в данный момент
времени для обработки (рисунок 71).

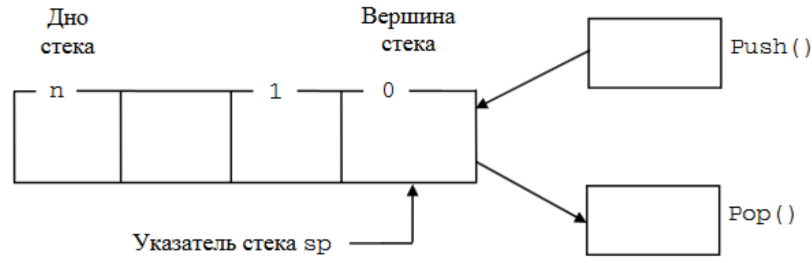


Рисунок 71 - Стек

Примеры стека: винтовочный патронный магазин, книги, сложенные в стопку, или стопка тарелок. Во всех этих моделях взять можно только верхний предмет, а добавить новый объект можно, только положив его на верхний.

При манипуляциях со стеком основными функциями являются:

- добавление в стек – `Push`;
- извлечение из стека – `Pop`;
- чтение элемента с вершины стека (без извлечения) – `Peek`;
- очистка стека – `Clear`.

Стеки могут быть реализованы статически – на основе массивов и динамически – на основе списков.

Реализация стека на основе динамического массива и списка

Можно также организовать стек на базе массива, который будет увеличиваться и уменьшаться динамически (например, при заполнении половины стека, его размер увеличивается на фиксированную величину). Например, стек можно представить следующей структурой:

```
struct Stack
{
     int Top; //вершина стека
     int Size; //размер стека
     void** Data; //данные стека
};
```

Чтобы стек мог рационально увеличиваться и уменьшаться, можно отдать предпочтение его организации на основе связного списка. Реализацию списков можно рассматривать как реализацию стеков, поскольку стеки с их операторами являются частными случаями списков с операторами, выполняемыми над списками. Для этого необходимо представить стек в виде однонаправленного списка. В этом случае операторы `push` и `pop` будут работать только с ячейкой заголовка и первой ячейкой списка.

Представление элементов односвязного списка организовано традиционно:

```
struct STACK{
     void* Data; //данные элемента стека
     STACK *next; //указатель на следующий
};
//Для добавления элемента (операция Push()) создается новый
//элемент и добавляется в начало списка:
//----------------------------- добавить в стек
```

```
void Push(STACK **ppStack, void* nItem)
{
     STACK *pNewItem;
     pNewItem = new STACK;
     //запрашиваем память под структуру для стека
     pNewItem->Data = nItem;
     //заполняем поля структуры
     pNewItem->next = *ppStack;
     //устанавливаем новый указатель на вершину стека
     *ppStack = pNewItem;
}
```

Для извлечения элемента, проверяется возможность выполнения операции (если стек не пустой), извлекается элемент из стека (удаление из начала списка):

```
//---------------------------- извлечь из стека
void* Pop(STACK **ppStack, int *nError)
{
     STACK *pOldData = *ppStack;
     //запомнить старый адрес вершины
     void* nOldData = NULL;
     if(*ppStack)
     {
          nOldData = pOldData->Data;
          //если стек не пустой, извлечь
          *ppStack = (*ppStack)->next;
          delete pOldData;
          //и освободить память
     }
     else *nError = 1;
          //в противном случае выставить ошибку
          return nOldData;
}
```

Обратите внимание, что в функциях `Push()` и `Pop()` используются двойные ссылки. Благодаря этому, каждая из функций может возвращать в качестве результата своей работы указатель на новый элемент `STACK` (используется передача параметров по адресу). Чтобы можно было вернуть значение указателя из функции, необходимо использовать именно двойные ссылки. Если в стеке есть хоть один элемент, то функции возвращают `nError = 0`, если стек пуст – `nError = 1`.

По аналогии выполняется операция чтения элемента с вершины без извлечения `Peek()`:

```
//----------прочитать не извлекая
void* Peek(STACK **ppStack, int *nError)
{
     if(*ppStack) //если стек не пустой
          {
               *nError = 0;
               return(*ppStack)->Data; //прочитать элемент
          }
     else
          {
```

```
                *nError = 1; //если стек пуст, установить 
            ошибку
                return NULL;
            }
    }
```

Очистка стека `Clear()` заключается в последовательном удалении элементов стека, начиная с заголовка:

```
    //---- очистить стек
    void Clear(STACK **ppStack)
    {
        STACK *pDelItem = *ppStack;
        while((*ppStack) != NULL //пока стек не пустой
        {
            pDelItem = *ppStack;
            *ppStack = (*ppStack)->next; //перейти к 
        следующему
            delete pDelItem; //удалить элемент
        }
    }
```

Итак, реализация стека на базе массива использует объем памяти, необходимый для размещения максимального числа элементов, которые может вместить стек в процессе вычислений; реализация на базе списка использует объем памяти пропорционально количеству элементов, но при этом всегда расходует дополнительную память для одной связи на каждый элемент, а также дополнительное время на распределение памяти при каждой операции «добавить» и освобождениепамяти при каждой операции «извлечь». Если требуется стек больших размеров, который обычно заполняется практически полностью, то предпочтение надо отдать реализации на базе массива. Если же размер стека варьируется в широких пределах и присутствуют другие структуры данных, которым требуется память, не используемая во время, когда в стеке находится несколько элементов, предпочтение следует отдать реализации на базе связного списка.

Несмотря на то, что реализация стека на базе связного списка создает впечатление, что стек может увеличиваться неограниченно в практических условиях, такой стек невозможен: рано или поздно, когда запрос на выделение еще некоторого объема памяти не сможет быть удовлетворен, оператор new сгенерирует исключение.

Очередь

Очередь – одномерная структура данных, для которой загрузка или извлечение элементов осуществляется с помощью указателей начала (`head`) и конца (`tail`) очереди в соответствии с правилом `FIFO` (`First In First Out`: первым пришел – первым ушел), другими словами, включение производится с одного, а исключение – с другого конца (рисунок 72).

При манипуляциях с очередью основными функциями являются:

- добавление элемента в конец очереди – `Enqueue`;
- извлечение элемента из начала очереди – `Dequeue`;

• чтение элемента из начала очереди (без извлечения) – `Peek`;
• очистка очереди – `Clear`.

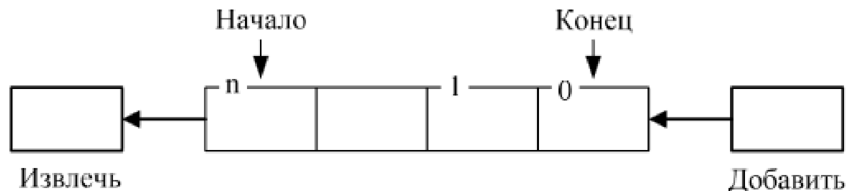


Рисунок 72 - Очередь

Также, как и стеки, очереди могут быть реализованы на основе массивов и на основе списков.

Реализация очереди на основе списка

Как и для стеков, любая реализация списков допустима для представления очередей.

```
struct QUEUE //очередь{
     void* Data; //данные элемента очереди
     QUEUE *next; //указатель на следующий
};
```

Каждая операция «извлечь» уменьшает размер очереди на 1, а каждая операция «добавить» увеличивает размер очереди на 1. Новые элементы добавляются в конец очереди (списка). Поэтому в функции добавления `Enqueue()`, если очередь содержит элементы, необходимо выделить память под новый элемент (`nItem`), получить указатель на последний элемент (`pQ`) и добавить в очередь (список) новый элемент, связывая его с элементом, на который ссылается указатель на последний (`tail`). Если очередь пуста, то необходимо добавить элемент в пустую очередь (очередь содержит единственный элемент и установление связей с другими элементами не требуется). Ниже приведена рассмотренная функция:

```
//------------------- добавить новый элемент в очередь
void Enqueue (QUEUE **ppQueue, void* nItem)
{
     if (*ppQueue) //если очередь не пустая
     {
          QUEUE *pNewItem = new QUEUE; //выделить память
          QUEUE *pQ;
          pQ = (*ppQueue);
          while(pQ->next) //перейти в конец очереди
               pQ = pQ->next;
          pNewItem->Data = nItem; //заполнить поля структуры
          pNewItem->next = NULL;
          pQ->next = pNewItem; //добавить в конец
     }
     else
     {
          (*ppQueue) = new QUEUE; //выделить память
          (*ppQueue)->Data    =    nItem;    //заполнить    поля
     структуры
          (*ppQueue)->next = NULL;
     }
```

```
}
```

Для извлечения элемента из очереди (списка) – `Dequeue()`, при условии, что очередь не пуста, извлекается и удаляется элемент из начала очереди (списка) так же, как это делалось в случае стека. Если очередь пуста, будет возвращаться указатель `NULL` и выставляться ошибка `*nError = 1`. Реализация приведена ниже:

```
//----------- извлечь элемент из очереди
void* Dequeue (QUEUE **ppQueue, int *nError)
{
     QUEUE *pOldData = *ppQueue;
     //запомнить старый адрес начала
     void* nOldData = NULL;
     if(*ppQueue)
     //если очередь не пуста
     {
          nOldData = pOldData ->Data;
          //извлечь данные
          *ppQueue = (*ppQueue)->next;
          //начало установить на следующий
          delete pOldData;
          //удалить старый начальный элемент очереди
     }
     else *nError = 1; //выставить ошибку
     return nOldData; //вернуть результат
     }
     Функция Clear() идентична аналогичной функции для стека
     с организацией на базе связного списка.
     //---------------------------- очистить очередь
     void Clear(QUEUE **ppQueue)
     {
          QUEUE *pDelItem = *ppQueue;
          while((*ppQueue) != NULL)
          //последовательно до конца очереди
          {
               pDelItem = *ppQueue;
               *ppQueue = (*ppQueue)->next;
               delete pDelItem;
               //удалить поэлементно
          }
     }
```

Очевидным преимуществом в случае представления очереди на базе связного списка является то, что оперативная память используется пропорционально числу элементов в структуре данных. Это происходит за счет дополнительного расхода памяти на связи (между элементами) и дополнительного расхода времени на распределение и освобождение памяти для каждой операции.

Очереди с приоритетами

В реальных задачах иногда возникает необходимость в формировании очередей, отличных от порядка обслуживания FIFO. Порядок выборки элементов из таких очередей определяется приоритетами элементов.

Приоритет в общем случае может быть представлен числовым значением, которое вычисляется либо на основании значений каких-либо полей элемента, либо на основании внешних факторов.

```
struct PR_QUEUE1 //приоритетная очередь
{
int Head; //начало очереди
int Tail; //конец очереди
int Size; //размер очереди (максимальное
//количество элементов + 1)
int* Data; //данные очереди
int* Prioritet; //приоритеты очереди
};
//..............................
struct PR_QUEUE2 //приоритетная очередь
{
void* Data; //данные очереди
int priority_value; //приоритеты очереди
PR_QUEUE *next;
};
```

Так, и стеки, и LIFO-очереди могут трактоваться как приоритетные очереди, в которых приоритет элемента зависит от времени его включения в очередь. Приоритетная очередь – это абстрактный тип данных, предназначенный для представления взвешенных множеств (куч). Множество называется взвешенным, если каждому его элементу однозначно соответствует число, называемое ключом или весом. При выборке элемента всякий раз выбирается элемент с наибольшим приоритетом, т.е. «первым включается – с высшим приоритетом исключается». Основными операциями над приоритетной очередью являются следующие операции:

• вставить новый элемент со своим ключом;

• найти с максимальным (минимальным) приоритетом. Если элементов несколько, то находится один из них. Найденный элемент не удаляется из очереди;

• удалить элемент с максимальным (минимальным) приоритетом.

Если элементов несколько, то удаляется один из них;

• увеличить (уменьшить) приоритет указанного элемента на заданное положительное число.

Приоритетная очередь естественным образом используется в таких задачах, как сортировка элементов массива, поиск во взвешенном неориентированном графе минимального остовного дерева, поиск кратчайших путей от заданной вершины взвешенного графа до его остальных вершин и во многих других.

Деки

Дек – особый вид очереди (от англ. deq – double ended queue, т.е очередь с двумя концами), в котором как включение, так и исключение элементов может осуществляться с любого из двух концов. Частные случаи дека – дек с ограниченным входом и дек с ограниченным выходом. Логическая и физическая структуры дека аналогичны логической и

физической структуре циклической FIFO-очереди. Однако, применительно к деку целесообразно говорить не о начале и конце, а о левом и правом конце. Над деком разрешены операции:

- включение элемента справа;
- включение элемента слева;
- исключение элемента справа;
- исключение элемента слева;
- определение размера;
- очистка.

Задачи, решение которых требует использования структуры дека, встречаются в вычислительной технике и программировании гораздо реже, чем задачи, реализуемые на структуре стека или очереди. Как правило, вся организация дека выполняется программистом без каких-либо специальных средств системной поддержки.

Лекция 17 Динамические нелинейные структуры

Деревья

В отличие от структур, очередей, стеков, деревья представляют собой иерархическую структуру некой совокупности элементов. Примерами деревьев могут служить генеалогические, организационные диаграммы, оглавление книги. Деревья используются при анализе электрических цепей, для представления структур математических формул, для организации информации в системах управления базами данных и представления синтаксических структур в компиляторах программ.

Название «дерево» проистекает из логической эквивалентности древовидной структуры абстрактному дереву в теории графов.

Дерево – это граф, который характеризуется следующими свойствами:

- существует единственный элемент (узел или вершина), на который не ссылается никакой другой элемент и который называется корнем;
- начиная с корня и следуя по определенной цепочке указателей, содержащихся в элементах, можно осуществить доступ к любому элементу структуры;
- на каждый элемент, кроме корня, имеется единственная ссылка, т.е. каждый элемент адресуется единственным указателем.

Таким образом, дерево представляет собой либо отдельную вершину, либо вершину, имеющую ограниченное число указателей – ветвей. Нижележащие деревья для текущей вершины называются поддеревьями, а их вершины – потомками (дочерними вершинами, сыновьями). По отношению к потомкам текущая вершина называется предком (рисунок 73). Те узлы, которые не ссылаются ни на какие другие узлы дерева, называются листьями (или терминальными вершинами). Узел, не являющийся листом или корнем, считается промежуточным или узлом ветвления (нетерминальной или внутренней вершиной).

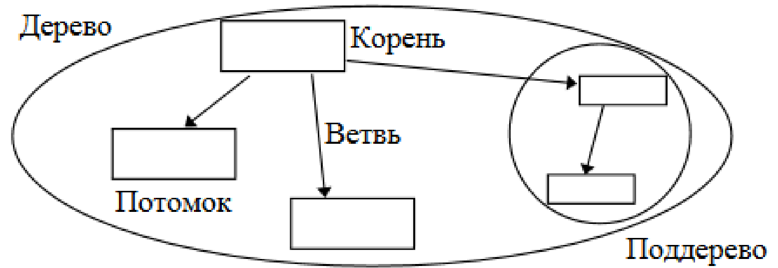


Рисунок 73 – Представление дерева

Вершину дерева можно определить таким образом:

```
struct TREE
{
int val; //значение элемента
TREE *child[4]; //указатели на потомков
};
```

Из определения элемента дерева следует, что оно имеет ограниченное число указателей на подобные элементы. Как и во всех динамических структурах данных, характер связей между элементами определяется функциями, которые их устанавливают. Рекурсивное определение дерева требует и рекурсивного характера функций, работающих со связями. Простейшей функцией является функция обхода всех вершин дерева:

```
void ScanTree (TREE *pTree) //обход дерева
{ int i;
if (pTree == NULL)
return; //следующей вершины нет
for (i=0; i<4; i++) //рекурсивный обход
ScanTree(pTree ->child[i]); //потомков с передачей
} //указателей на них
```

Само дерево обычно задается в программе указателем на его корень. Часто обход дерева используется для получения информации, которая затем возвращается через результат рекурсивной функции, например, функция определения минимальной длины ветвей дерева:

```
int MinDepth(TREE *pTree)
{ int i, min, nn;
if (pTree == NULL) return 0;
//следующей вершины нет
for (min = MinDepth(pTree->child[0], i=1; i<4; i++)
{
nn = MinDepth(pTree ->child[i]);
//обход потомков
if (nn > max) max = nn;
} //возвращается глубина
return min + 1; //с учетом текущей вершины
}
```

Другой распространенной функцией является включение нового элемента в дерево. Здесь есть проблема, общая для всех деревьев и рекурсивных алгоритмов. Войдя в поддерево, невозможно производить

какие-либо действия для вершин, расположенных на том же уровне, но в других поддеревьях. Поэтому используется функция включения с просмотром дерева на заданную глубину, а сама глубина просмотра, в свою очередь, задается равной длине минимальной ветви дерева:

```
int Insert(TREE *pTree, int newItem, int d)
//pTree - указатель на текущую вершину
//d - текущая глубина включения
{if (d == 0) return 0; //ниже не просматривать
for (int i=0; i<4; i++)
if (pTree->child[i] == NULL)
{
TREE *pn=new TREE;
pTree->child[i] = pTree;
pTree->val = newItem;
for (i=0; i<4; i++) pTree->child[i] = NULL;
return 1;
//результат логический - вершина включена
}
else
if (Insert(pTree->child[i], v , d-1)) return 1;
return 0;
}
int _tmain(int argc, _TCHAR* argv[]).
{
tree PH={1,{NULL,NULL,NULL,NULL}};
//пример вызова функции
Insert(&PH, 5, MinDepth(&PH));
}
```

Из последнего примера видно, что количество просматриваемых вершин от уровня к уровню растет в геометрической прогрессии. Таким образом, деревья можно эффективно использовать для поиска данных.

Бинарные деревья

Бинарным (или двоичным) деревом называется дерево, каждая вершина которого имеет не более двух потомков. При определении бинарного дерева поиска на данные, хранимые в вершинах дерева, вводится следующее правило упорядочения: значения вершин левого поддерева всегда меньше, а значения вершин правого поддерева – больше значения в самой вершине (рисунок 74). Это свойство позволяет применить в дереве алгоритм двоичного поиска. Действительно, каждое сравнение искомого значения и значения в вершине двоичного дерева позволяет выбрать для следующего шага правое или левое поддерево.

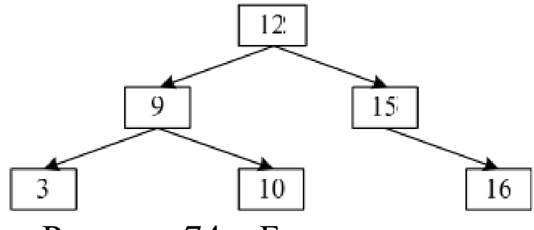


Рисунок 74 – Бинарное дерево

Вид дерева бинарного поиска (или BST – Binary Search Tree), соответствующий набору меняющихся данных, зависит от последовательности, в которой значения помещаются в дерево.

Узел бинарного дерева можно представить следующей структурой:

```
struct Node //узел бинарного дерева
{ Node* Parent; //указатель на родителя
Node* Left; //указатель на левое поддерево
Node* Right; //указатель на правое поддерево
void* Data; //данные
};
```

Нумерация вершин в деревьях. Способы обхода дерева

В любой структуре данных имеется естественная нумерация элементов по их расположению в ней. Так, каждый элемент списка или массива имеет свой логический номер в линейной последовательности, соответствующей его размещению в памяти (массив) или направлению последовательного обхода (списки). В деревьях обход вершин возможен с использованием рекурсии, поэтому и логическая нумерация вершин производится согласно последовательности их рекурсивного обхода. Рекурсивная функция в этом случае получает текущий счетчик вершин, который она увеличивает на 1 при обходе текущей вершины и который она передает и получает обратно из поддеревьев.

Бинарное дерево можно обходить тремя основными способами: нисходящим, смешанным и восходящим (возможны также обход слева-направо, справа-налево). Принятые выше названия методов обхода связаны с временем обработки корневой вершины: до того, как обработаны оба ее поддерева (Preorder); после того, как обработано левое поддерево, но до того, как обработано правое (Inorder); после того как, обработаны оба поддерева (Postorder). Используемые в переводе названия методов отражают направление обхода в дереве: от корневой вершины вниз к листьям – нисходящий обход; от листьев вверх к корню –восходящий обход и смешанный обход – от самого левого листа дерева через корень к самому правому листу.

Определим следующую структуру узла с конструктором, операциями перехода к следующему, предыдущему, минимальному (в соответствии со свойством дерева – крайне левый лист), максимальному элементу (крайне правый лист) и с функциями обхода бинарного дерева:

```
struct Node //узел бинарного дерева
{
Node* Parent; //указатель на родителя
Node* Left; //указатель на левое поддерево
Node* Right; //указатель на правое поддерево
void* Data; //данные
Node(Node* p, Node* l, Node* r, void* d)
//конструктор
{ Parent = p; Left = l; Right = r; Data = d; }
Node* Next(); //следующий
Node* Prev(); //предыдущий
Node* Min(); //минимум в поддереве
```

```
Node* Max(); //максимум в поддереве
void DescendingScan(void (*f)(void* n));
//нисходящий обход
void Scan(void (*f)(void* n));
//восходящий обход
void MixedScan(void(*f)(void* n));
//смешаный обход
int TreeH(int h = 0);
};
```

Нисходящий обход

Далее рассмотрен пример обхода вершин дерева слева-направо, сверху-вниз (рисунок 75). Он заключается в посещении текущего узла, обходе левого поддерева, обходе правого поддерева:

```
//---------------------- нисходящий обход
void Node:: DescendingScan(void (*f)(void* n))
{
f(this->Data);
//обработка узла дерева
std::cout<<std::endl;
if (this->Left != NULL) this->Left->Scan(f);
//рекурсивный вызов для левого поддерева
if (this->Right != NULL) this->Right->Scan(f);
//рекурсивный вызов для правого поддерева
}
```

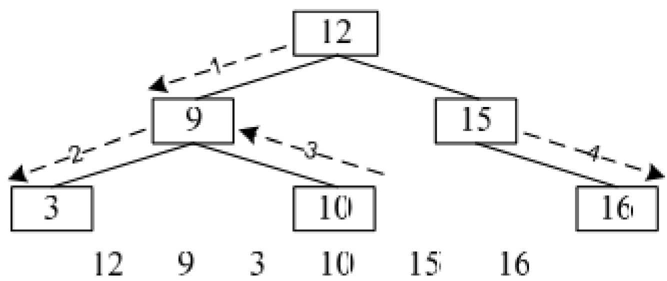


Рисунок 75 – Нисходящий обход дерева

Обход дерева можно реализовать без рекурсии с использованием стека. Алгоритм обхода бинарного дерева в соответствии с нисходящим способом может выглядеть следующим образом:

1) в качестве очередной вершины взять корень дерева, перейти к пункту 2;

2) произвести обработку очередной вершины в соответствии с требованиями задачи, перейти к пункту 3;

3а) если очередная вершина имеет обе ветви, то в качестве новой выбрать ту вершину, на которую ссылается левая ветвь, а вершину, на которую ссылается правая ветвь, занести в стек; перейти к пункту 2;

3б) если очередная вершина является конечной, то выбрать в качестве новой очередной вершины вершину из стека, если он не пуст, и перейти к пункту 2; если же стек пуст, то это означает, что обход всего дерева окончен, перейти к пункту 4;

3в) если очередная вершина имеет только одну ветвь, то в качестве очередной вершины выбрать ту вершину, на которую эта ветвь ука-

зывает, перейти к пункту 2;

4) конец алгоритма.

Восходящий обход

Он заключается в посещении при обходе левого узла поддерева, обходе правого поддерева, посещении текущего узла (рисунок 76):

```
//-------------------- восходящий обход
void Node:: Scan(void (*f)(void* n))
{
     std::cout<<std::endl;
     if (this->Left != NULL) this->Left->Scan(f);
     //рекурсивный вызов для левого поддерева
     if (this->Right != NULL) this->Right->Scan(f);
     //рекурсивный вызов для правого поддерева
     f(this->Data);
     //обработка узла дерева
}
```

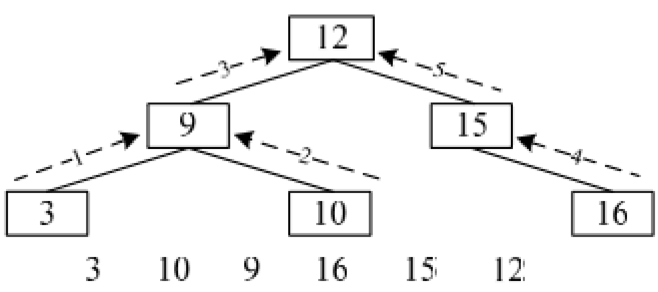


Рисунок 76 – Восходящий обход дерева

Трудность организации восходящего обхода с использованием стека заключается в том, что в отличие от нисходящего алгоритма в этом алгоритме каждая вершина запоминается в стеке дважды: первый раз – когда обходится левое поддерево, и второй раз – когда обходится правое поддерево. Таким образом, в алгоритме необходимо различать два вида стековых записей: 1-й означает, что в данный момент обходится левое поддерево; 2-й – что обходится правое, поэтому в стеке запоминается указатель на узел и признак (код-1 и код-2 соответственно). Алгоритм восходящего обхода можно представить следующим образом:

1) спуститься по левой ветви с запоминанием вершины в стеке как 1-й вид стековых записей;

2) если стек пуст, то перейти к пункту 5;

3) выбрать вершину из стека, если это первый вид стековых записей, то возвратить его в стек как 2-й вид стековых записей, перейти к правому «сыну», перейти к пункту1, иначе перейти к пункту 4;

4) обработать данные вершины и перейти к пункту 2;

5) конец алгоритма.

Смешанный обход

Рекурсивный смешанный обход описывается следующим образом: смешанный обход левого поддерева; обработка узла; смешанный обход правого поддерева (рисунок 77):

```
//-------------смешанный обход
void Node::MixedScan(void (*f)(void* n))
{
     std::cout<<std::endl;
     if (this->Left != NULL) this->Left->Scan(f);
     // рекурсивный вызов для левого поддерева
     f(this->Data);
     //обработка узла дерева
     if (this->Right != NULL) this->Right->Scan(f);
     // рекурсивный вызов для правого поддерева
}
```

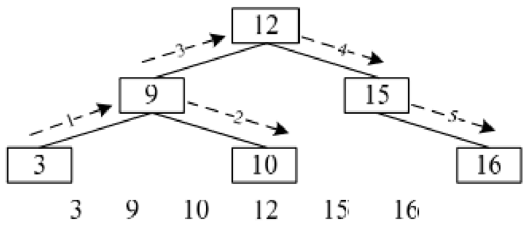


Рисунок 77 – Смешанный обход

Аналогичный обход в порядке справа-налево дает в двоичном дереве последовательность в порядке убывания.

Смешанный обход с использованием стека можно описать следующим образом:

1) спуститься по левой ветви с запоминанием вершин в стеке;

2) если стек пуст, то перейти к пункту 5;

3) выбрать вершину из стека и обработать данные вершины;

4) если вершина имеет правого «сына», то перейти к нему; перейти к пункту 1.

5) конец алгоритма.

Для обхода всего дерева требуется O(n) времени.

Операции поиска

Основные операции при работе с бинарными деревьями поиска связаны с поиском определенного значения, поиском наименьшего, наибольшего элемента дерева, предшествующего и последующего элементов для данного.

Выполнение операции поиска основано на том, что, находясь на определенной вершине, можно всегда однозначно указать, в каком из поддеревьев находится искомое значение, так как согласно свойству бинарного дерева поиска все значения узлов в левом поддереве не больше, а в правом не меньше значения в корне. Таким образом, функция поиска заданного значения имеет следующий вид:

```
//----------------------поиск заданного
Node* Search(void* Item, Node* pTree)
{ Node* x = pTree;
```

```
        if (x != NULL) //если дерево не пустое
        { if (Item < pTree->Data)
             x = Search(Item, pTree->Left);
        else if (Item > pTree->Data))
             x = Search(Item, pTree->Right);
        }
        return x;
   }
```

В случае, если искомое значение отсутствует, в дереве будет возвращен указатель на корень.

Чтобы достичь наименьшего (наибольшего) значения в дереве поиска, надо двигаться по левым (или соответсвенно правым) ветвям дерева до тех пор, пока это возможно:

```
   //------ поиск минимального
   Node* Min(Node* pTree)
   {    Node* x = pTree;
        if (x->Left != NULL) x = Min(x->Left);
        return x;
   }
   //---- поиск максимального
   Node* Max(Node* pTree)
   { Node* x = pTree;
     if (x->Right != NULL) x = Max(x->Right);
     return x;
   }
```

Поиск очередного и предшествующего узла – задача более сложная, чем предшествующие. Задача решается с использованием исключительно знаний о структуре дерева. Если правое поддерево текущего непустое, то следующий за текущим – минимальный элемент правого поддерева. Если правое поддерево текущего пустое и существует, то следующий – наименьший предок текущего, левый наследник которого также является предком текущего:

```
   //--------- поиск следующего
   Node* Next(Node* pTree)
   { Node* next = pTree, *x = pTree;
        if (next->Right != NULL) next = Min(next->Right);
        else
        {
             next = pTree->Parent;
             while (next != NULL && x == next->Right)
                  {
                  x = next;
                  next = next->Parent;
                  }
        }
        return next;
   }
```

Функция поиска узла, предшествующего данному, симметрична функции поиска последующего узла:

```
   //--------- поиск предыдущей
   Node* Prev(Node* pTree)
```

```
{
	Node* prev = pTree, *x = pTree;
	if (prev->Left != NULL) prev = Max(prev->Left);
		else
		{
			prev = pTree ->Parent;
			while (prev != NULL && x == prev->Left)
			{
				x = prev;
				prev = prev->Parent;
			}
		}
	return prev;
}
```

Включение вершины в бинарное дерево

Последние две рассматриваемые операции над бинарным деревом поиска – включение узла в дерево и исключение. Алгоритмы включения и исключения вершин дерева не должны нарушать основное свойство.

При включении узла в дерево сначала выполняется поиск места вставки, а затем узел вставляется с изменением поля у вставляемого и родительского узлов:

```
//--------------------------------------- включение узла
bool Insert(Node* pTree, void* NewItem)
{
	Node* temp = pTree, *n = NULL;
	bool x = true;
	while (x == true && temp != NULL)
	//поиск места включения
	{
	n = temp;
	if (NewItem < temp->Data) temp = temp->Left;
	else if (NewItem > temp->Data) temp = temp->Right;
	else x = false;
	}
	if (x == true && n == NULL)
		pTree = new Node(NULL, NULL, NULL, NewItem);
		//включить в корень
	else if (x == true && NewItem < n->Data)
		n->Left = new Node(n, NULL, NULL, NewItem);
		//включить слева
		else if (x == true && NewItem > n->Data))
			n->Right = new Node(n, NULL, NULL, NewItem);
			//включить справа
	return x;
};
```

Функция начинает работу с корневого узла и перемещает указатель `temp` вниз. Указатель n постоянно указывает на родительский по отношению к `temp` узел, а сам указатель перемещается в соответствии с результатами сравнения. После того, как указатель `temp` становиться равным 0, он находится в той позиции, куда следует поместить новый узел `NewItem`.

Удаление вершины из бинарного дерева

Функция удаления более сложная, поскольку необходимо рассматривать различные варианты. Если удаляемый узел – лист, то удаление сводится к обнулению в родительском узле указателя на удаляемый (рисунок 78).

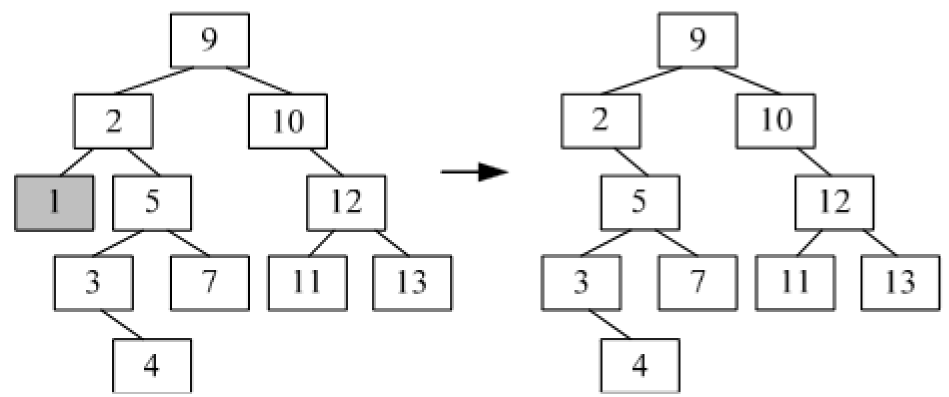


Рисунок 78 – Удаление листа дерева

Если у удаляемого только одно поддерево, то указатель в родительском должен указывать на дочерний по отношению к удаляемому и должен быть исправлен указатель на родительский узел в дочернем по отношению к удаляемому (рисунок 79).

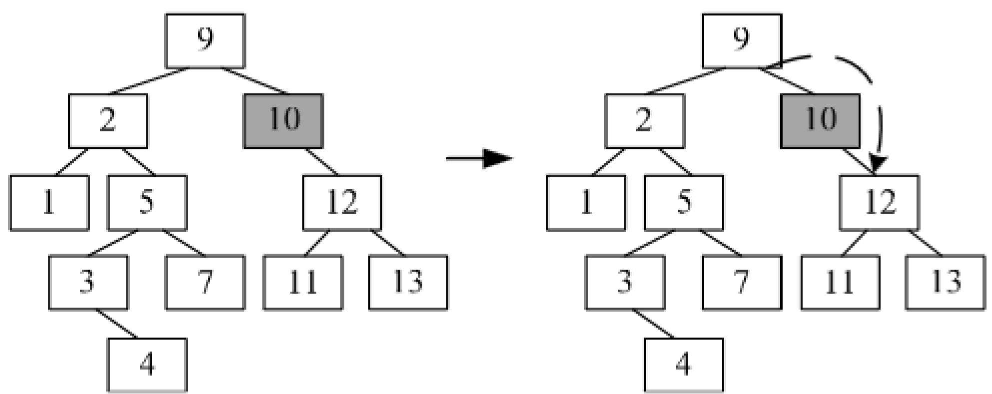


Рисунок 79 – Удаление узла дерева с одним потомком

Если у удаляемого узла два дочерних, то надо найти следующий за ним узел, извлечь его из дерева и заменить им удаляемый (рисунок 80).

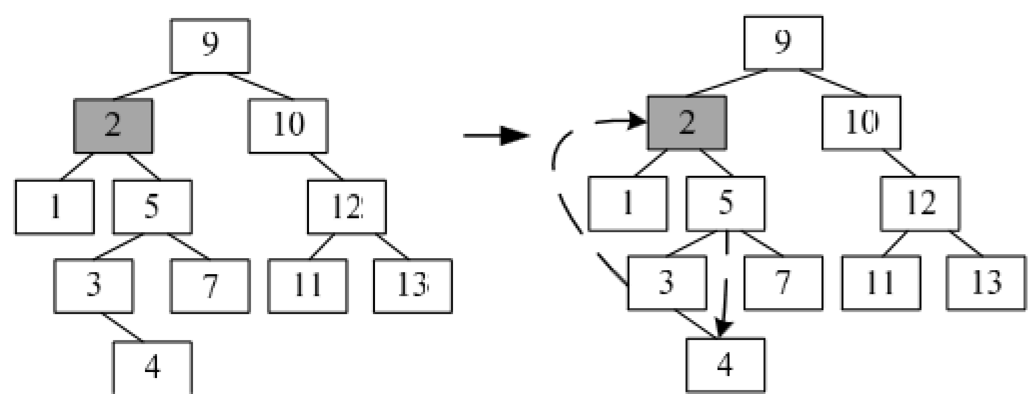


Рисунок 80 – Удаление узла дерева с двумя потомками

Реализация алгоритма представлена ниже:

```
//-------------удаление узла
bool Delete(Node* pTree, Node* n )
{
     bool x = true;
     if (x = (n != NULL))
     {
          if (n->Left == NULL && n->Right == NULL)
          //узел - лист
          {
               if (n->Parent == NULL) pTree = NULL;
               else if (n->Parent->Left == n)
               n->Parent->Left = NULL;
               else n->Parent->Right = NULL;
               delete n;
          }
     else if (n->Left==NULL && n->Right!=NULL)
     //есть правое поддерево
     {
          if (n->Parent == NULL) pTree = n->Right;
          else if (n->Parent->Left == n)
          n->Parent->Left = n->Right;
          else n->Parent->Right = n->Right;
          n->Right->Parent = n->Parent;
          delete n;
     }
          else if (n->Left!=NULL && n->Right==NULL)
          //есть левое поддерево
          {
               if (n->Parent == NULL) pTree = n->Left;
               else if (n->Parent->Left == n)
               n->Parent->Left = n->Left;
               else n->Parent->Right = n->Left;
               n->Right->Parent = n->Parent;
               delete n;
          }
               else if (n->Left != NULL && n->Right != NULL)
               //есть оба поддерева
               {
                    Node* temp = n->Next();
                    n->Data = temp->Data;
                    x = Delete(temp);
               }
     }
     return x;
}
```

Все приведенные операции имеют сложность O(h), где h – высота дерева. Когда дерево приближается к полному, т.е. когда для каждого внутреннего узла по два потомка, высота дерева составляет примерно log 2n, где n – общее количество узлов дерева. В наихудшем случае эффективность всех описанных операций над бинарным деревом поиска составляет O(logn). Однако в вырожденном дереве (вырождается в одну цепочку) эффективность падает до O(n).

Сбалансированные деревья

Одной из наиболее часто встречающихся задач является поиск необходимых данных. Одним из методов, улучшающих время поиска в бинарном дереве, является создание сбалансированных деревьев, обладающих минимальным временем поиска.

Дерево является сбалансированным тогда и только тогда, когда для каждого узла высота его двух поддеревьев различается не более чем на 1.

С тем, чтобы предупредить появление несбалансированного дерева, вводится для каждого узла (вершины) дерева показатель сбалансированности, который может принимать одно из трех значений, левое (L), правое (R), сбалансированное (B), в соответствии со следующими определениями:

– узел левоперевешивающий, если самый длинный путь по его левому поддереву на единицу больше самого длинного пути по его правому поддереву;

– узел сбалансированный, если равны наиболее длинные пути по обоим его поддеревьям;

– узел правоперевешивающий, если самый длинный путь по его правому поддереву на единицу больше самого длинного пути по его левому поддереву.

В сбалансированном дереве каждый узел должен находится в одном из этих трех состояний. Если в дереве существует узел, для которого это условие несправедливо, такое дерево называется несбалансированным.

Процесс включения узла в сбалансированное дерево состоит из последовательности трех этапов:

1) следовать по пути поиска (по ключу), пока не будет найден ключ или окажется, что ключа нет в дереве;

2) включить новый узел и определить новый показатель сбалансированности;

3) пройти обратно по пути поиска и проверить показатель сбалансированности у каждого узла.

В общем, процесс удаления элемента состоит из следующих этапов:

1) следовать по дереву, пока не будет найден удаляемый элемент;

2) удалить найденный элемент, не разрушив структуры связей между элементами;

3) произвести балансировку полученного дерева и откорректировать показатели сбалансированности.

Простыми случаями являются удаление терминальных узлов и удаление узлов с одним потомком. Если же узел, который надо удалить, имеет два поддерева, будем заменять его самым правым узлом левого поддерева.

На каждом шаге должна передаваться информация о том, увеличилась ли высота поддерева (в которое произведено включение). Поэтому, можно

ввести в список параметров переменную h, означающую: высота поддерева увеличилась. Таким образом, структура узла дерева:

```
struct AVL_Node //узел бинарного дерева
{
     AVL_Node* Parent; //указатель на родителя
     AVL_Node* Left; //указатель на левое поддерево
     AVL_Node* Right; //указатель на правое поддерево
     void* Data; //данные
     int Height; //высота узла
     int Weight; //вес узла
     char bf; //баланс-фактор
}
```

Необходимые операции балансировки полностью заключаются в обмене значениями ссылок. Фактически ссылки обмениваются значениями по кругу, что приводит:

- к однократному правому (RR);
- однократному левому (LL);
- двукратному лево-правому (LR);
- двукратному право-левому (RL) «повороту» узлов.

Рассмотрим пример удаления различных узлов из сбалансированного дерева.

Пусть в бинарном дереве на рисунке 81 надо удалить узел со значением 12. Удаление узла 12 само по себе просто, т.к. он представляет собой терминальный узел. Однако при этом появляется несбалансированность в корневом узле 9. Его балансировка требует однократного левого поворота.

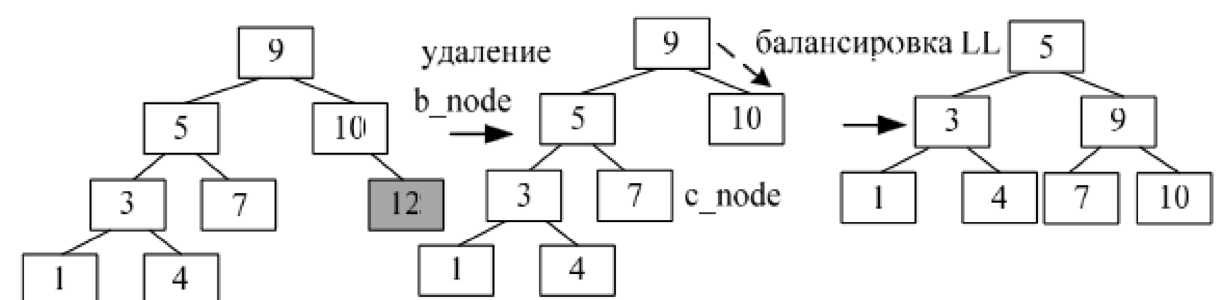


Рисунок 81 - Удаление узла и LL-балансировка дерева

При однократном левом повороте (LL) необходимо ввести следующие обозначения: левого «потомка», поворачиваемого узла обозначим переменой b_node, а правого «потомка» b_node обозначим c_node. Функции левого поворота LLRotate() передается два параметра: указатель на узел, для которого выполняется поворот pNode, и корень дерева m_pRoot. Реализация функции будет выглядеть следующим образом:

```
bool LLRotate(AVL_NODE* pNode, AVL_NODE* m_pRoot)
{
     AVL_NODE* b_node = pNode->Left;
     AVL_NODE* c_node = b_node->Right;
     //первое – меняем местами pNode и b_node
     if(pNode->Parent)
     {
          if(pNode->Parent->Left == pNode)
```

```
            pNode->Parent->Left = b_node;
            else
            pNode->Parent->Right = b_node;
        }
        else m_pRoot = b_node;
        b_node->Parent = pNode->Parent;
        //второе - меняем местами правого сына b_node и pNode
        b_node->Right = pNode;
        pNode->Parent = b_node;
        //третье - меняем местами левого сына pNode и c_node
        pNode->Left = c_node;
        if(c_node) c_node->Parent = pNode;
        //последнее, обновить высоту и баланс-фактор pNode и b_node
        if(c_node)
        {
        pNode->Height = (c_node->Height >= pNode->Right->Height)?
        (c_node->Height+1) : (pNode->Right->Height+1);
        pNode->bf = (char)(c_node->Height - pNode->Right->Height);
        }
        else
        { if(pNode->Right)
            { pNode->Height = 1;
            pNode->bf = -1;
            }
        else
            { pNode->Height = 0;
            pNode->bf = 0;
        }
        }
    b_node->Height = (b_node->Left->Height >= pNode->Height)?
    (b_node->Left->Height+1) : (pNode->Height+1);
    b_node->bf =(char)(b_node->Left->Height - pNode->Height);
    return true;
    }
```

Пусть в бинарном дереве на рисунке 81 надо удалить узел со значением 3. Для устранения разбалансировки в узле 5 требуется однократный правый поворот. Технология поворота продемонстрирована на рисунке 82. Функция правостороннего поворота `RRRotate()` будет симметрична функции `LLRotate()`.

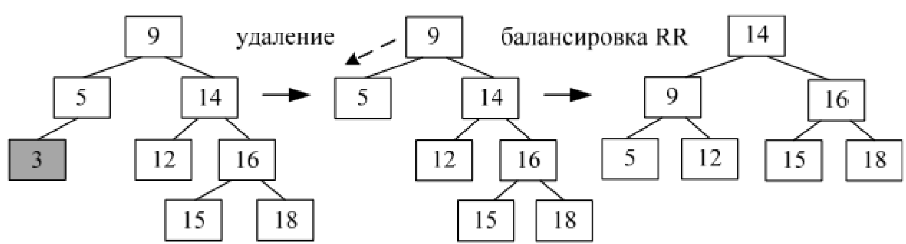


Рисунок 82- Удаление узла и RR-балансировка дерева

Технически сложнее будет выглядеть третий случай – двукратный поворот направо и налево (RL). Например, при удалении узла 3 для дерева на рисунке 83 балансировка узла 9 требует двукратного RL-поворота.

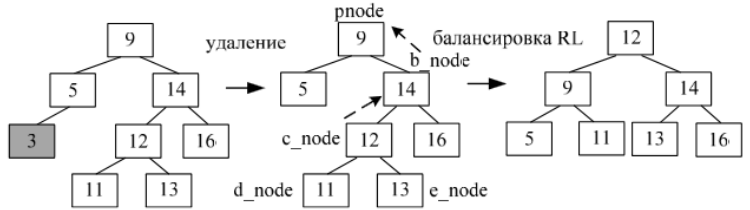


Рисунок 83 - Удаление узла и RL-балансировка дерева

Введем дополнительные обозначения: правого «потомка» поворачиваемого узла `pNode` обозначим переменной `b_node`, левого «потомка» `b_node` обозначим `c_node`, левого «потомка» `c_node` обозначим `d_node` и правого «потомка» `c_node` обозначим `e_node`.

Тогда функция `RLRotate()` будет выглядеть следующим образом:

```
bool RLRotate(AVL_NODE* pNode, AVL_NODE* m_pRoot)
{
AVL_NODE* b_node = pNode->Right;
AVL_NODE* c_node = b_node->Left;
AVL_NODE* d_node = c_node->Left;
AVL_NODE* e_node = c_node->Right;
//поменять местами pNode и c_node
if(pNode->Parent)
{ if (pNode->Parent->Left == pNode)
     pNode->Parent->Left = c_node;
     else pNode->Parent->Right = c_node;
}
else m_pRoot = c_node;
c_node->Parent = pNode->Parent;
//поменять левого потомка c_node с pNode
//и правого потомка с b_node
c_node->Left = pNode;
c_node->Right = b_node;
pNode->Parent = b_node->Parent = c_node;
//поменять правого потомка pNode с d_node
//и левого потомка b_node с e_node
pNode->Right = d_node;
if(d_node) d_node->Parent = pNode;
b_node->Left = e_node;
if(e_node) e_node->Parent = b_node;
//обновление высоты и баланс-факторов узлов
//обновить pnode
if(d_node)
{Node->Height = (pNode->Left->Height >= d_node->Height)?
      (pNode->Left->Height+1) : (d_node->Height+1);
     pNode->bf = (char)(pNode->Left->Height - d_node->Height);
}
else
{ if(pNode->Left)
     { pNode->Height = 1;
     pNode->bf = 1;
      }
else
     { Node->Height = 0;
```

```
      Node->bf = 0;
      }
    }
    //обновить b_node
    if(e_node)
    {b_node->Height =(e_node->Height >= b_node->Right->Height)?
    (e_node->Height+1) : (b_node->Right->Height+1);
    b_node->bf =(char)(e_node->Height - b_node->Right-
    >Height);}
    else
    { if(b_node->Right)
         { b_node->Height = 1; b_node->bf = -1; }
    else
         { b_node->Height = 0; b_node->bf = 0;}
    }
    //обновить c_node
    c_node->Height = (pNode->Height >= b_node->Height)?
    (pNode->Height+1) : (b_node->Height+1);
    c_node->bf =(char)(pNode->Height - b_node->Height);
    return true;
    }
```

Симметрично рассмотренному будет выглядеть двукратный поворот налево и направо (LR) (рисунок 84).

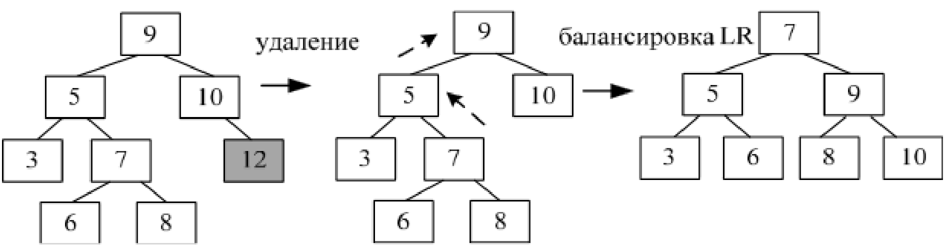


Рисунок 84- Удаление узла и LR-балансировка дерева

2 ПРАКТИЧЕСКИЙ РАЗДЕЛ

1.Методические указания и индивидуальные задания к выполнению лабораторных работ по дисциплине «Основы алгоритмизации и программирования» для студентов дневной и заочной форм обучения специальности 6-05-0612-03 Системы управления информацией, и для студентов дневной формы обучения по специальностям 6-05-0611-03 Искусственный интеллект, 6-05-0612-01 Программная инженерия.

2. ЛАБОРАТОРНЫЕ РАБОТЫ

Примерное содержание отчета

Титульный лист с указанием названия дисциплины, темы и номера лабораторной работы, группы и фамилии студента.

Обязательные разделы отчета:

1. Цель работы.
2. Описание задачи, задания, предметной области.
3. Описание контрольного (тестового примера).
4. Описание структуры программы. Описание алгоритмов. Алгоритм решения задачи (словесное описание и блок – схема алгоритма(по ГОСТ))

5.Описание реализации программы.

6.Вывод

Приложение 1. Листинги программы.

Приложение 2. Листинги и протоколы результатов выполнения программы.

Приложение 3. Результаты испытания и тестирования программы (на контрольных примерах).

Примечание: Пункты 2-5 повторяются для каждой задачи. После проверки исправления добавляются к отчету на новых листах либо (если незначительные) делаются на старом отчете.

В п.3 подбираются исходные данные и над ними выполняются вручную все действия, ведущие к решению поставленной задачи и получению результатов. Пример используется для демонстрации правильности понимания условия задачи, а для разработанной программы выполняет роль тестового примера (с известными – эталонными результатами) при ее испытании. Для тестирования программы следует подобрать пример или группу примеров, обеспечивающих проверку работы программы в различных ситуациях, включая исключительные ситуации, приводящие к ошибкам. Для этого предварительно выполняется анализ возможных ситуаций при решении задачи.

В п.4 приводится описание разработанного алгоритма программы

В п.4 приводится описание алгоритмов модулей (функций) в терминах графических схем программ ГОСТ (блок - схем)

Требования к разрабатываемым программам

1. Структурированность – текст программы должен быть составлен в соответствии с требованиями структурного программирования.
2. Документированность – до текста программы должен быть приведен текст задания и выполнена спецификация программы. Каждый модуль (функция, класс) программы специфицируется аналогично. В каждом модуле (функции, классе) должны быть специфицированы все объявляемые переменные.

3. Универсальность – все исходные данные, используемые в программе и определяемые из вне, должны вводиться с возможностью задания любых допустимых значений.
4. Сама программа готовится и предъявляется либо в двух вариантах либо в одном объединенном.
   Отладочный вариант - с автоматической инициализацией данными контрольных примеров. Рабочий вариант с возможностью задания любых допустимых исходных значений.

Лабораторная работа №1 «Разработка схем алгоритмов для линейных и разветвляющихся процессов в соответствии с положениями действующих стандартов. Разработка схем алгоритмов для циклических процессов в соответствии с положениями действующих стандартов. Разработка структурированных схем алгоритмов»

Этапы создания программного обеспечения

Программирование (programming) - теоретическая и практическая деятельность, связанная с созданием программ. Решение задач на компьютере включает в себя следующие основные этапы, часть из которых осуществляется без участия компьютера.

1. Постановка задачи:

- сбор информации о задаче;
- формулировка условия задачи;
- определение конечных целей решения задачи;
- определение формы выдачи результатов;
- описание данных (их типов, диапазонов величин, структуры и т. п.).

2. Анализ и исследование задачи, модели:

- анализ существующих аналогов;
- анализ технических и программных средств;
- разработка математической модели;
- разработка структур данных.

3. Разработка алгоритма:

- выбор метода проектирования алгоритма;
- выбор формы записи алгоритма (блок-схемы, псевдокод и др.);
- выбор тестов и метода тестирования;
- проектирование алгоритма.

4. Программирование:

- выбор языка программирования;
- уточнение способов организации данных;
- запись алгоритма на выбранном языке

программирования.

5. Тестирование и отладка:

- синтаксическая отладка;
- отладка семантики и логической структуры;
- тестовые расчеты и анализ результатов тестирования;
- совершенствование программы.

6. Анализ результатов решения задачи и уточнение в случае необходимости математической модели с повторным выполнением этапов 2-5.

7. Сопровождение программы(внедрение и поддержка):

- доработка программы для решения конкретных задач;
- составление документации к решенной задаче, к математической модели, к алгоритму, к программе, к набору тестов, к использованию.

Жизненный цикл программного продукта

Программный продукт должен быть соответствующим образом подготовлен к эксплуатации, иметь необходимую техническую документацию, предоставлять сервис и гарантию надежной работы программы, иметь товарный знак изготовителя. Только при таких условиях созданный программный комплекс может быть назван программным продуктом.

Программный продукт имеет несколько качественных характеристик:

- алгоритмическая сложность;
- полнота функций обработки;
- объём файлов программ;
- требования к операционной системе и техническим средствам обработки со стороны программного средства;
- объём дисковой памяти;
- размер оперативной памяти.

Показатели качества должны содержать следующие аспекты:

Программа является *правильной*, если она работает в соответствии с техническим заданием (ТЗ - документ, которым завершается постановка задачи).

Программа является **точной**, если выдаваемые ею числовые данные имеют допустимые отклонения от аналогичных результатов, полученных с помощью идеальных математических зависимостей.

Программа является **совместимой**, если она работает должным образом не только автономно, но и как часть программной системы.

Программа является **надежной,** если она при всех входных данных обеспечивает полную повторяемость результатов.

Программа является **универсальной**, если она правильно работает при любых допустимых вариантах исходных данных. В ходе разработки программ предусматриваются специальные средства защиты от ввода неправильных данных, обеспечивающие целостность системы.

Программа является **защищенной,** если она сохраняет работоспособность при возникновении сбоев (режим реального времени, программа большого времени выполнения).

Программа является **полезной,** если задача, которую она решает, представляет практическую ценность.

Программа является **эффективной,** если объем требуемых для ее работы ресурсов ЭВМ не превышает допустимого предела.

Программа является **проверяемой,** если ее качества могут быть продемонстрированы на практике (проверка правильности и универсальности). Существуют формальные математические методы проверки и неформальные (прогоны программы с остановками в контрольных точках, обсуждение результатов заинтересованными пользователями).

Программа является **адаптируемой,** если она допускает быструю модификацию с целью приспособления к изменяющимся условиям функционирования.

В условиях существования рынка программных продуктов важными характеристиками являются стоимость, количество продаж, время нахождения на рынке, известность фирмы-производителя и самой программы, наличие на рынке программных продуктов аналогичного назначения.

Программный продукт любого вида характеризуется жизненным циклом, состоящим из отдельных этапов (рисунок Л1).

Рисунок Л1 - Жизненный цикл программного продукта

Маркетинг предназначен для **изучения требований** к создаваемому программному продукту (технических, программных, пользовательских). Изучаются также существующие аналоги и продукты-конкуренты. Оцениваются необходимые для разработки материальные, трудовые и финансовые ресурсы, а также устанавливаются примерные сроки разработки.

Проектирование структуры - алгоритмизация процесса обработки данных, детализация функций, разработка архитектурного проекта, выбор методов и средств создания программ.

Программирование (реализация), тестирование и отладка - основной этап работы по разработке программного средства. Часто отдельные работы этого этапа ведутся параллельно, что позволяет сократить общее время разработки.

Документирование - обязательный вид работы. Документация должна содержать необходимые сведения по установке, обеспечению надёжной работы продукта, справочное пособие для пользователя, демонстрационные версии, примеры документов, создаваемых при помощи данного программного продукта, обучающие программы.

Выход программного продукта на рынок связан с организацией продаж массовому пользователю. Здесь применяются стандартные методы - реклама, увеличение числа каналов реализации, создание дилерской и дистрибьюторской сети, гибкая ценовая политика.

Эксплуатация и сопровождение идут, как правило, параллельно. В процессе эксплуатации могут выявляться ошибки, и устранение этих ошибок ведётся в режиме сопровождения, то есть оказание сервисной помощи, обеспечение новыми версиями программ, организация «горячих телефонных линий» для консультаций.

Снятие программного продукта с продажи и **отказ от его сопровождения** происходит, как правило, в случае изменения технической политики фирмы-изготовителя, неэффективности работы программного продукта, наличия в нём неустранимых ошибок, отсутствие спроса.

Длительность жизненного цикла разных программных продуктов неодинакова. Для большинства современных программ его длительность составляет 2-3 года. Хотя часто встречаются на компьютерах и давно снятые с производства программные продукты.

Понятие алгоритма

Для составления программы, предназначенной для решения на ЭВМ какой-либо задачи, требуется составление алгоритма ее решения.

Алгоритм — это точное предписание, которое определяет процесс, ведущий от исходных данных к требуемому конечному результату. Алгоритмами, например, являются правила сложения, умножения, решения алгебраических уравнений, умножения матриц и т.п. Слово алгоритм происходит от algoritmi, являющегося латинской транслитерацией арабского имени хорезмийского математика IX века аль-Хорезми. Благодаря латинскому переводу трактата аль-Хорезми европейцы в XII веке познакомились с позиционной системой счисления, и в средневековой Европе алгоритмом называлась десятичная позиционная система счисления и правила счета в ней.

Применительно к ЭВМ алгоритм определяет вычислительный процесс, начинающийся с обработки некоторой совокупности возможных исходных данных и направленный на получение определенных этими исходными

данными результатов. Термин вычислительный процесс распространяется и на **обработку** других видов информации, например, символьной, графической или звуковой.

Если вычислительный процесс заканчивается получением результатов, то говорят, что соответствующий алгоритм применим к рассматриваемой совокупности исходных данных. В противном случае говорят, что алгоритм неприменим к совокупности исходных данных. Любой применимый алгоритм обладает следующими **основными свойствами:**

- результативностью;

  Результативность означает возможность получения результата после выполнения конечного количества операций.

- определенностью;

  Определенность состоит в совпадении получаемых результатов независимо от пользователя и применяемых технических средств.

- массовостью;

  Массовость заключается в возможности применения алгоритма к целому классу однотипных задач, различающихся конкретными значениями исходных данных.

- дискретностью;

  Дискретность означает разбиение алгоритма на конечную последовательность действий или шагов при его выполнении.

- конечностью.

  Конечность означает то, что алгоритм должен выполняться за конечное время.

Для задания алгоритма необходимо описать следующие его элементы:

- набор объектов, составляющих совокупность возможных исходных данных, промежуточных и конечных результатов;
- правило начала;
- правило непосредственной переработки информации (описание последовательности действий);
- правило окончания;
- правило извлечения результатов.

Алгоритм всегда рассчитан на конкретного исполнителя. В нашем случае таким исполнителем является ЭВМ. Для обеспечения возможности реализации на ЭВМ алгоритм должен быть описан на языке, понятном компьютеру, то есть на языке программирования.

Таким образом, можно дать следующее определение программы.

Программа для ЭВМ представляет собой описание алгоритма и данных на некотором языке программирования, предназначенное для последующего автоматического выполнения.

Способы описания алгоритмов

К основным способам описания алгоритмов можно отнести следующие:

- словесно-формульный;
- структурный или блок-схемный;
- с помощью граф-схем;
- с помощью сетей Петри.

Перед составлением программ чаще всего используются **словесно-формульный** и **блок-схемный** способы. Иногда перед составлением программ на низкоуровневых языках программирования типа языка Ассемблера алгоритм программы записывают, пользуясь конструкциями некоторого высокоуровнего языка программирования. Удобно использовать программное описание алгоритмов функционирования сложных программных систем. Так, для описания принципов функционирования ОС использовался Алголоподобный высокоуровневый язык программирования.

При **словесно-формульном** способе алгоритм записывается в виде текста с формулами по пунктам, определяющим последовательность действий.

Пусть, например, необходимо найти значение следующего выражения:

$y = 2a – (x+6)$.

Словесно-формульным способом алгоритм решения этой задачи может быть записан в следующем виде:

1. Ввести значения a и x.
2. Сложить x и 6.
3. Умножить a на 2.
4. Вычесть из 2a сумму (x+6).
5. **Вывести** y как результат вычисления выражения

При блок-схемном описании алгоритм изображается геометрическими фигурами (блоками), связанными по управлению линиями (направлениями потока) со стрелками. В блоках записывается последовательность действий.

Данный способ по сравнению с другими способами записи алгоритма имеет ряд преимуществ. Он наиболее нагляден: каждая операция вычислительного процесса изображается отдельной геометрической фигурой. Кроме того, графическое изображение алгоритма наглядно показывает разветвления путей решения задачи в зависимости от различных условий, повторение отдельных этапов вычислительного процесса и другие детали.

Оформление программ должно соответствовать определенным требованиям. В настоящее время действует единая система программной документации (ЕСПД), которая устанавливает правила разработки, оформления программ и программной документации. В ЕСПД определены и правила оформления блок-схем алгоритмов (**ГОСТ 10.002-80 ЕСПД, ГОСТ 10.003-80 ЕСПД, ГОСТ 19.701-90**).

Операции обработки данных и носители информации изображаются на схеме соответствующими блоками. Большая часть блоков по построению условно вписана в прямоугольник со сторонами a и b. Минимальное значение a = 10 мм, увеличение a производится на число, кратное 5 мм. Размер b=1,5a. Для от дельных блоков допускается соотношение между a и b, равное 1:2. В пределах одной схемы рекомендуется изображать блоки одинаковых размеров. Все блоки нумеруются. Виды и назначение основных блоков приведены в таблице 1.

Линии, соединяющие блоки и указывающие последовательность связей между ними, должны проводится параллельно линиям рамки. Стрелка в конце линии может не ставиться, если линия направлена слева направо или сверху вниз. В блок может входить несколько линий, то есть блок может являться преемником любого числа блоков. Из блока (кроме логического) может выходить только одна линия. Логический блок может иметь в качестве продолжения один из двух блоков, и из него выходят две линии. Если на схеме имеет место слияние линий, то место пересечения выделяется точкой. В случае, когда одна линия подходит к другой и слияние их явно выражено, точку можно не ставить.

Схему алгоритма следует выполнять как единое целое, однако в случае необходимости допускается обрывать линии, соединяющие блоки.

Если при обрыве линии продолжение схемы находится на этом же листе, то на одном и другом конце линии изображается специальный символ соединитель — окружность диаметром 0,5 a. Внутри парных окружностей указывается один и тот же идентификатор. В качестве идентификатора, как правило, используется порядковый номер блока, к которому направлена соединительная линия.

Если схема занимает более одного листа, то в случае разрыва линии вместо окружности используется межстраничный соединитель. Внутри каждого, соединителя указывается адрес — откуда и куда направлена соединительная линия. Адрес записывается в две строки: в первой указывается номер листа, во второй — порядковый номер блока.

Блок-схема должна содержать все разветвления, циклы и обращения к подпрограммам, содержащиеся в программе.

Условные обозначения блоков схем алгоритмов приведенены в таблице Л1

Таблица Л1 - Условные обозначения блоков схем алгоритмов

1. Ввод-вывод (данные)		Преобразование данных в форму, пригодную для обработки (ввод) или отображения результатов обработки (вывод)
2. Ручной ввод		Ввод данных вручную при помощи неавтономных устройств с клавиатурой, переключателей, кнопок
3. Документ		Ввод – вывод данных, носителем которых служит бумага
4. Процесс		Выполнение операции или группы операций, в результате которых изменяется значение, форма представления или расположение данных
5. Предопределенный процесс		Использование раннее созданных и отдельно описанных алгоритмов или программ

6. Ручная операция	b a a	Автономный процесс, выполняемый вручную или при помощи неавтоматически действующих средств
7. Модификация (подготовка)	a a b	Выполнение операций, меняющих команды или группы команд, изменяющих программу
8. Решение	a b	Выбор направления выполнения алгоритма или программы в зависимости от некоторых переменных условий
9. Линия потока	————————	Поток данных или управления
10. Пунктирная линия	- - - - - - - - -	Альтернативная связь между двумя или более символами
11. Параллельные действия	0,25a	Начало или окончание двух или более одновременно выполняемых операций
12. Пропуск	—— ... —— |...|	Пропуск символов или группы символов, в которых не определены ни тип, ни число символов

13. Соединитель	Ø0,5a	Указание связи между прерванными линиями потока, связывающими символы
14. Пуск - терминатор	R=0,25a R 0,5a b	Начало, конец, прерывание процесса обработки данных или выполнения программы
15. Комментарий	a 5	Связь между элементом схемы и пояснением
16. Граница цикла	a 0,25a b	
17. Передача управления	a a	Непосредственная передача управления от одного процесса к другому, иногда с возможностью прямого возвращения к инициирующему процессу после того, как инициируемый процесс завершит свои функции
18. Канал связи	2,5 2,5 3 3 a b	Передача данных по каналу связи
19.Межстраничный соединитель	0,5a 0,6a 0,2a	Указание связи между прерванными линиями, соединяющими блоки, расположенные на разных листах.

20. Дисплей		Ввод – вывод данных, если непосредственно подключенное к процессу устройство воспроизводит данные и позволяет оператору ПК вносить изменения в процессе их обработки
21. Неавтономная память (запоминаемые данные)		Ввод – вывод данных в случае использования запоминающего устройства, управляемого непосредственно процессором
22. Оперативная память		Ввод – вывод данных, носителем которых служит магнитный сердечник
23. Запоминающее устройство с последовательным доступом		Ввод – вывод данных, носителем которых служит магнитная лента
24. Запоминающее устройство с прямым доступом		Ввод – вывод данных, носителем которых служит магнитный барабан

ВНИМАНИЕ !!! Размер a должен выбираться из ряда 10, 15, 20 мм. Допускается увеличивать размер a на число, кратное 5. Размер b=1,5a.

Структурные схемы алгоритмов

Одним из свойств алгоритма является **дискретность** — возможность расчленения процесса вычислений, предписанных алгоритмом, на отдельные

этапы, возможность выделения участков программы с определенной структурой. Можно выделить и наглядно представить графически три простейшие структуры:

- последовательность двух или более операций;
- выбор направления;
- повторение.

Любой вычислительный процесс может быть представлен как комбинация этих элементарных алгоритмических структур. Соответственно, вычислительные процессы, выполняемые на ЭВМ по заданной программе, можно разделить на три основных вида:

- линейные;
- ветвящиеся;
- циклические.

Линейным принято называть вычислительный процесс, в котором операции выполняются последовательно, в порядке их записи. Каждая операция является самостоятельной, независимой от каких-либо условий. На схеме блоки, отображающие эти операции, располагаются в линейной последовательности.

Линейные вычислительные процессы имеют место, например, при вычислении арифметических выражений, когда имеются конкретные числовые данные и над ними выполняются соответствующие условию задачи действия. На рисунке 1 показан пример линейного алгоритма, определяющего процесс вычисления арифметического выражения

$y=(b^2-ac):(a+c)$.

Вычислительный процесс называется **разветвляющимся**, если для его реализации предусмотрено несколько направлений (ветвей). Каждое отдельное направление процесса обработки данных является отдельной ветвью вычислений. Ветвление в программе — это выбор одной из нескольких последовательностей команд при выполнении программы. Выбор направления зависит от заранее определенного признака, который может относиться к исходным данным, к промежуточным или конечным результатам. Признак характеризует свойство данных и имеет два или более значений.

Разветвляющийся процесс, включающий в себя две ветви, называется простым, более двух ветвей — сложным. Сложный ветвящийся процесс можно представить с помощью простых ветвящихся процессов.

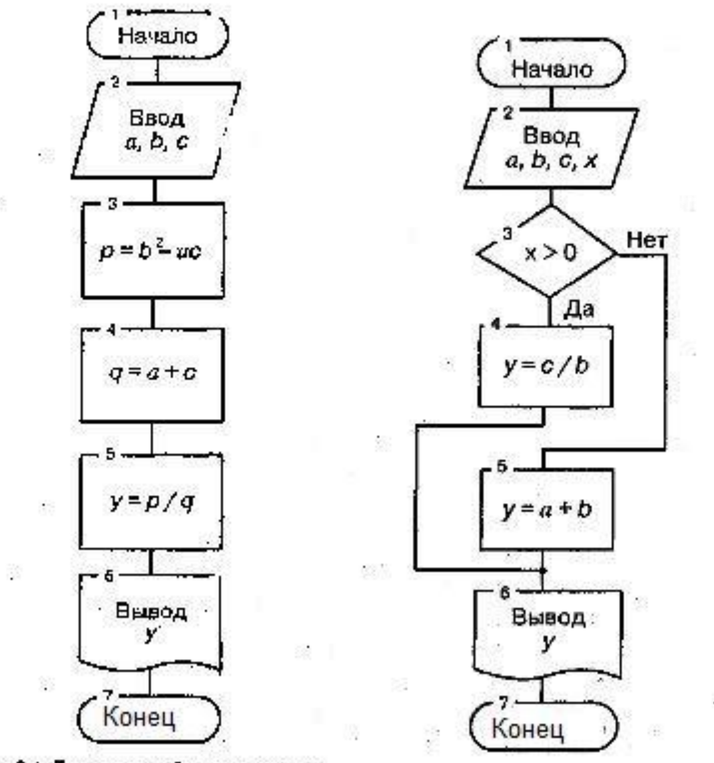


Рисунок Л2 – Пример линейного (а) и разветвяющегося алгоритма

Направление ветвления выбирается логической проверкой, в результате которой возможны два ответа: «да» — условие выполнено и «нет» — условие не выполнено.

Следует иметь в виду, что, хотя на схеме алгоритма должны быть показаны все возможные направления вычислений в зависимости от выполнения определенного условия (или условий), при однократном прохождении программы процесс реализуется только по одной ветви, а остальные исключаются. Любая ветвь, по которой осуществляются вычисления, должна приводить к завершению вычислительного процесса.

На рисунке 2. показан пример алгоритма с разветвлением для вычисления следующего выражения:

Y = (a+b), если X <0;

c/b, если X>0.

Циклическими называются программы, содержащие циклы. **Цикл** — это многократно повторяемый участок программы.

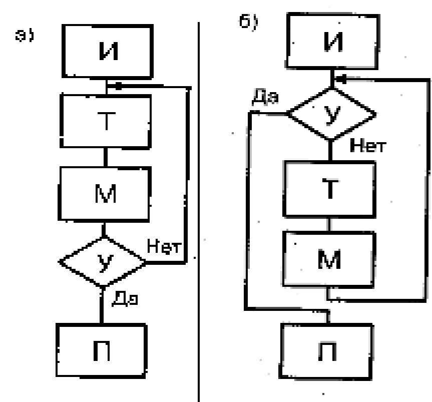


Рисунок Л3 - Примеры циклических алгоритмов

В организации цикла можно выделить следующие этапы:

- подготовка (инициализация) цикла (И);
- выполнение вычислений цикла (тело цикла) (Т);
- модификация параметров (М);
- проверка условия окончания цикла (У).

Порядок выполнения этих этапов, например, Т и М, может изменяться. В зависимости от расположения проверки условия окончания цикла различают циклы с нижним и верхним окончаниями (рисунок 3). Для цикла с нижним окончанием (**цикл с постусловием**)-(рисунок 3 а) тело цикла выполняется как минимум один раз, так как сначала производятся вычисления, а затем проверяется условие выхода из цикла. В случае цикла с верхним окончанием **(цикл с предусловием)**-(рисунок 3 б) тело цикла может не выполниться ни разу в случае, если сразу соблюдается условие выхода.

Цикл называется **детерминированным**, если число повторений тела цикла заранее известно или определено. Цикл называется **итерационным**, если число повторений тела цикла заранее неизвестно, а зависит от значений параметров (некоторых переменных), участвующих в вычислениях.

На рисунке Л4 показан пример циклического алгоритма вычисления суммы десяти чисел.

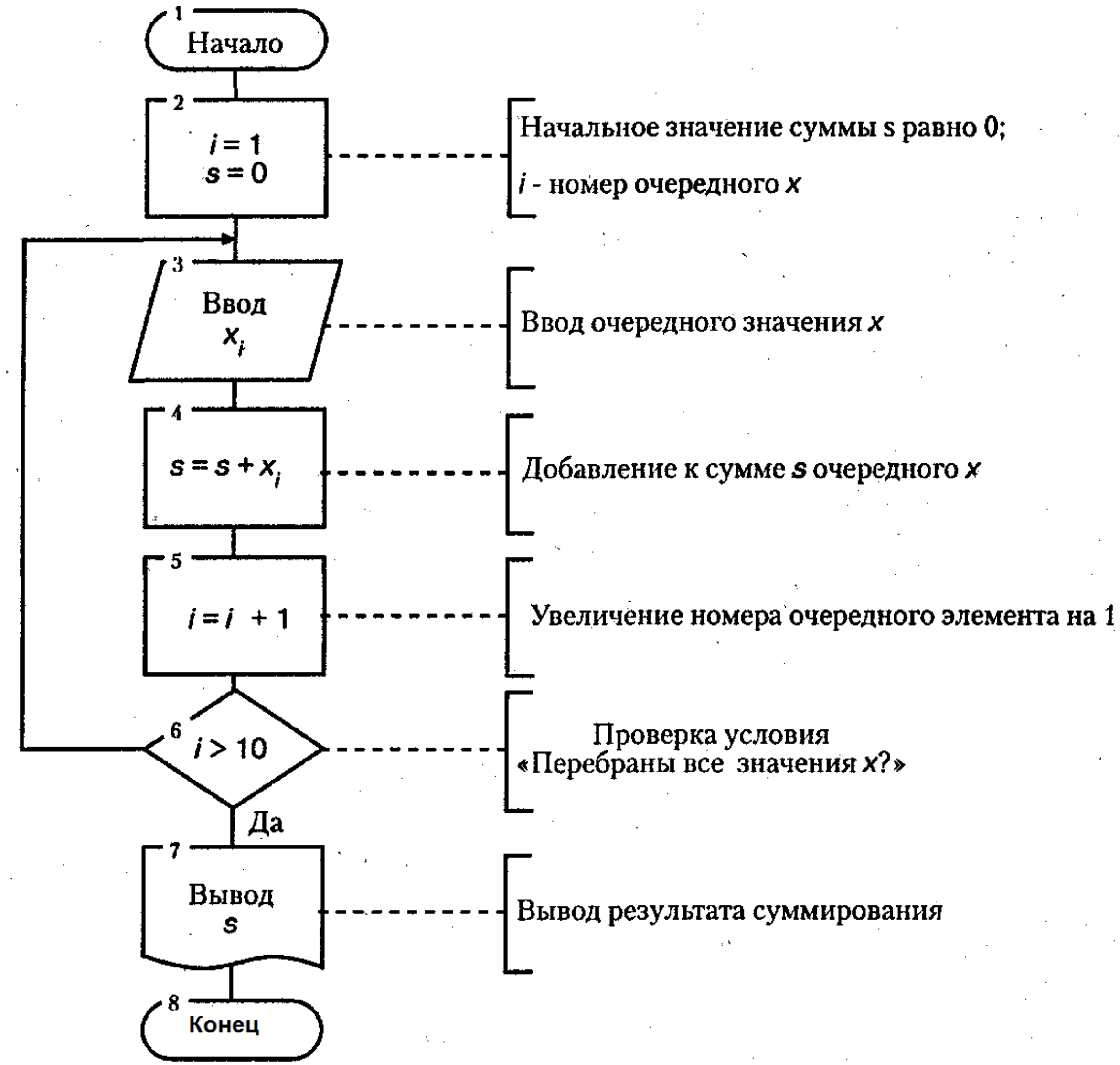


Рис .4 Алгоритм нахождения суммы 10-ти чисел

Контрольные вопросы к защите лабораторной работы

1. Что такое алгоритм?
2. Перечислите основные свойства алгоритма.
3. Что означает массовость алгоритма?
4. Что означает конечность алгоритма?
5. Какие существую способы описания алгоритмов?
6. Какие ГОСТ-ы определяют правила оформления блок-схем алгоритмов?
7. Какие условные обозначения блоков схем алгоритмов существуют?
8. Какие элементарные алгоритмические структуры существуют?
9. Приведите примеры циклических алгоритмов.
10. В чем состоит отличие детерминированного цикла от итерационного?

Задание

1. Необходимо разработать алгоритм решения задачи (в соответствии с вариантом)
2. Разработать словесное описание алгоритма решения задачи
3. Разработать блок – схему решения поставленной задачи.
4. Блок – схему реализовать в соответствии с ГОСТ
5. Блок – схему алгоритма начертить в Microsoft Visio

Варианты заданий:

1. Задана точка M с координатами (x,y). Определить местоположение этой точки в декартовой система координат(является ли эта точка началом координат, лежит ли на одной из координатных осей или расположена в одном из координатных углов).
2. Задан параллелограмм со сторонами a,b и углом α между ними. Определить тип параллелограмма (ромб, прямоугольник или квадрат), если это возмножно.
3. Известны углы α и β у основания трапеции. Выяснить, ели это возможно, тип трапеции (прямоугодьная,равнобедренная, прямоугольник).
4. Задан круг с центром в точке $O(x_0,y_0)$ и радиусом R_0 и точка $A(x_1,y_1)$. Определить местополодение точки по отношению к круга (находится внутри круга, вне его или лежит в окружности).
5. Заданы две окружности: с центром в точке $O(x_0,y_0)$ и с центром вточке $O_1(x_1,y_1)$ и радиусом R_1. Определите, во скольких точках пересекаются окружности.

Лабораторная работа №2 «Знакомство со средой программирования»

Язык C в основе своей был создан в 1972 г. как язык для операционной системы UNIX. Его автором считается Денис М. Ритчи (Dennis M. Ritchie). Некоторое время отсутствовала единая политика по стандартизации языка. В начале 1980-х гг. в Американском национальном институте стандартов (ANSI) был сформирован комитет по стандартизации языка C. В 1989 г. работа комитета по языку C была ратифицирована, и в 1990 г. вышел в свет первый официальный документ по стандарту языка C. Появился стандарт 1989, т. C89. К разработке стандарта по языку C была также привлечена Международная организация по стандартизации (ISO). Появился стандарт ISO/IEC 9899:1990.

Стандарты:
C:
K&R C
C89 | ANSI C | ANSI X3.159-1989

```
C99 | ISO/IEC 9899:1999
C11 | ISO/IEC 9899:2011
```
C++:
```
C++98 | ISO/IEC 14882:1998
http://www.cplusplus.com/doc/oldtutorial/
C++03 | ISO/IEC 14882:2003
C++07/TR1 | ISO/IEC TR 19768:2007
C++11 | ISO/IEC 14882:2011
C++14 | ISO/IEC 14882:2014
C++17
```

ПРАКТИЧЕСКАЯ ЧАСТЬ

1. Создать проект для кода С, осуществить все необходимы настройки интегрированной среды. Набрать код программы С.

```
#include <stdio.h>
#include <conio.h>

int summa(int a, int b){
        printf("Work function summa\n");
        return a+b;
        }

int mult(int a, int b){
        printf("\nWork function multiplication");
        return a*b;
        }

int fact(int a){
        int f=1;
        for(int i=1; i<a+1; i++ ) {

                f*=i;
        }
        printf("\nWork function factorial");
        return f;
        }

int main( )
{
   int a,b,s,p;
   printf("Enter the value of the variable  a=");
   scanf("%d", &a);
   printf("\nEnter the value of the variable b=");
   scanf("%d",&b);
   s=summa(a,b);
   printf("The sum of two numbers (a+b)=%d", s);
   p=mult(a,b);
   printf("\nThe multiplication of two numbers (a*b)=%d",p);
   printf("\nEnter the value of the variable f=");
   scanf("%d",&p);
   printf("\nFactorial number %d equal %d",p,fact(p));
   getchar();
   scanf ("Vvod s=",&s);
   return 0;
}
```

2. Создать проект для кода С++, осуществить все необходимы настройки интегрированной среды. Набрать код программы С++.

```
#include <iostream>
```

```
#include <conio.h>
using namespace std;
int summa(int a, int b){
	cout<<"Work function summa\n";
	return a+b;
}
int mult(int a, int b){
	cout<<"\nWork function multiplication";
	return a*b;
}
int fact(int a){
	int f=1;
	for(int i=1; i<a+1; i++ ) {
		f*=i;
	}
	cout<<"\nWork function factorial";
	return f;
}

int main( )
{
	int a,b,s,p;
	cout<<"Enter the value of the variable  a=";
	cin>>a;
	cout<<"Enter the value of the variable b=";
	cin>>b;
	s=summa(a,b);
	cout<<"The sum of two numbers (a+b)= "<<s;
	p=mult(a,b);
	cout<<"\nThe multiplication of two numbers (a*b)= "<<p;
	cout<<"\nEnter the value of the variable f=";
	cin>>p;
	cout<<"Factorial number  "<<p<<"   equal  "<<fact(p)<<endl;
	cout<<endl;
	getchar();
	system("PAUSE");
	return 0;
}
```

3. Выполнить программу C++ пошагово, использую клавишу F10. Результат занести в отчет. Пояснить в чем особенность данного режима отладки. Выводы занести в отчет.
4. Выполнить программу C++ пошагово, использую клавишу F11. Результат привести в отчете. Пояснить в чем особенность данного режима отладки. Выводы занести в отчет.
5. Создать 2 точки останова на выделенных строчках в коде программы C++.

```
cout<<"Enter the value of the variable b=";
cin>>b;
s=summa(a,b);
cout<<"The sum of two numbers (a+b)= "<<s;
p=mult(a,b);
cout<<"\nThe multiplication of two numbers (a*b)= "<<p;
```

Выполнить программу по нажатию на F5 до строчки

```
cout<<"Enter the value of the variable b=";
```

Со строчки

```
cout<<"Enter the value of the variable b=";
```

(выполнить программу пошагово по нажатию клавиши F10)

до строчки

```
cout<<"\nThe multiplication of two numbers (a*b)= "<<p;
```

После строчки

```
cout<<"\nThe multiplication of two numbers (a*b)= "<<p;
```

завершить выполнение программы по нажатию на F5.

В отчете пояснить полученный результат. Пояснить, для чего нужны точки останова.

6. Удалить точки останова в коде С++.
7. Научиться анализировать данные в окне **Output**, научиться добавлять переменные в окно отладки. Занести переменные a и b в окно отладки и проследить, как изменяются их значения, результаты занести в отчет.
8. Выполните компиляцию проекта С++ в отладочной конфигурации(Win32 Debug).Поясните назначение данной отладочной конфигурации. Посмотрите состав файлов в проекте. Поясните назначение каждого файла. Занесите результаты в отчет. Занесите размер проекта в отчет.
9. Выполните компиляцию проекта С++ в отладочной конфигурации (Win32 Release). Поясните назначение данной отладочной конфигурации. Посмотрите состав файлов в проекте. Поясните назначение каждого файла. Занесите результаты в отчет. Занесите размер проекта в отчет.
10. Проведите анализ размеров и состав файлов проекта С++ в конфигурациях (Win32 Debug) и (Win32 Release). Результаты занесите в отчет.
11. Получите исполняемый файл, созданный в результате компоновки проекта С++. Посмотрите его место размещения на диске. Запустите его и посмотрите результат. Занесите результаты в отчет.

Лабораторная работа №3 «Разработка, отладка и выполнение простейшей программы»

Задание

1. Вычислить значение выражения при различных вещественных типах данных (float и double). Вычисления следует выполнять с использованием промежуточных переменных. Сравнить и объяснить полученные результаты.

2. Вычислить значения выражений. Объяснить полученные результаты.

Варианты

№	Задание 1	Задание 2
1	$\frac{(a+b)^2-(a^2+2ab)}{b^2}$, при a=1000, b=0.0001	1) n+++m 2) m-- >n 3) n-- >m
2	$\frac{(a-b)^2-(a^2-2ab)}{b^2}$, при a=1000, b=0.0001	1) ++n*++m 2) m++<n 3) n++>m
3	$\frac{(a+b)^3-(a^3+3a^2b)}{3ab^2+b^3}$, при a=1000, b=0.0001	1) n---m 2) m--<n 3) n++>m
4	$\frac{(a+b)^3-(a^3)}{3ab^2+b^3+3a^2b}$, при a=1000, b=0.0001	1) n++*m 2) n++<m 3) m-- >m

Лабораторная работа №4 «Разработка алгоритма, составление, отладка и выполнение программы с ветвлением (выбором вариантов)»

Задание1: реализовать программу с использованием операторов ветвления:

1. Даны три целых числа. Возвести в квадрат отрицательные числа и в третью степень — положительные (число 0 не изменять).
2. Из трех данных чисел выбрать наименьшее. Begin44. Из трех данных чисел выбрать наибольшее.
3. Из трех данных чисел выбрать наименьшее и наибольшее. Begin46. Перераспределить значения переменных X и Y так, чтобы в X оказалось меньшее из этих значений, а в Y — большее.
4. Значения переменных X, Y, Z поменять местами так, чтобы они оказались упорядоченными по возрастанию.
5. Значения переменных X, Y, Z поменять местами так, чтобы они оказались упорядоченными по убыванию.

Задание2: реализовать программу с использованием оператора выбора:

1. Дан номер месяца (1 — январь, 2 — февраль, ...). Вывести название соответствующего времени года ("зима", "весна" и т.д.).
2. Дан номер месяца (1 — январь, 2 — февраль, ...). Вывести число дней в этом месяце для невисокосного года.
3. Дано целое число в диапазоне 0 – 9. Вывести строку — название соответствующей цифры на русском языке (0 — "ноль", 1 — "один", 2 — "два", ...).
4. Дано целое число в диапазоне 1 – 5. Вывести строку — словесное описание соответствующей оценки (1 — "плохо", 2 — "неудовлетворительно", 3 — "удовлетворительно", 4 — "хорошо", 5 — "отлично").

5. Арифметические действия над числами пронумерованы следующим образом: 1 — сложение, 2 — вычитание, 3 — умножение, 4 — деление. Дан номер действия и два числа A и B (B не равно нулю). Выполнить над числами указанное действие и вывести результат.

Лабораторная работа №5 «Разработка алгоритма, составление, отладка и выполнение циклической программы с известным числом повторений. Разработка алгоритма, составление, отладка и выполнение программы с использованием итерационных циклов. Разработка и выполнение программы с использованием разветвлений и вложенных циклов»

Задание

1. Найти натуральное число от 1 до N с максимальной суммой делителей
2. Дано натуральное n. Можно ли представить его в виде суммы трех квадратов натуральных чисел и, если можно, то указать все представления n в виде суммы квадратов трех натуральных чисел
3. Даны целые числа p и q. Получить все делители числа q взаимно простые с p
4. Даны натуральные m и n. Определить, сколько раз каждая цифра десятичной записи числа m встречается в десятичной записи числа n
5. Дано натуральное число n. Выбрать из записи числа n цифры 0 и 5, оставив прежним порядок остальных цифр.

Лабораторная работа №6 «Разработка алгоритма, составление, отладка и выполнение программы обработки сложных типов данных (массивы)»

Задание

Использовать индексы и указатели для доступа к элементам матриц. Дана действительная матрица порядка $n \times m$.

Варианты

1. Поменять местами строку, содержащую элемент с наибольшим значением в матрице со строкой, содержащей элемент с наименьшим значением. Вывести на экран полученную матрицу. Для каждой строки с нулевым элементом на главной диагонали вывести ее номер и значение наибольшего из элементов этой строки.
2. Среди строк заданной матрицы, содержащих только нечетные элементы, найти строку с максимальной суммой модулей элементов.
3. Найти номер строки заданной матрицы, в которой находится

самая длинная серия (последовательность одинаковых элементов).
4. Получить номера строк, элементы каждой из которых образуют монотонную последовательность (монотонно убывающую или монотонно возрастающую).
5. Подсчитать количество строк заданной матрицы, являющихся перестановкой чисел 1, 2, ..., 20.

Лабораторная работа №7 «Разработка алгоритма, составление, отладка и выполнение программы обработки сложных типов данных (строки)»

Задание

Использовать индексы и указатели для доступа к элементам строки.

Варианты

1. Написать программу, которая во введенной с клавиатуры строке меняет местами четные и нечетные слова.
2. Разработать программу, в которой с клавиатуры вводятся две строки символов. К строке с наибольшей длиной добавить текст, содержащийся в другой строке.
3. Создать программу, которая выводит на экран первую часть таблицы кодировки символов (символы с кодами от 0 до 127). Таблица должна состоять из восьми колонок и шестнадцати строк. В первой колонке должны быть символы с кодом от 0 до 15, во второй – от 16 до 31 и т.д.
4. Составить программу, которая выводит на экран сообщение в «телеграфном» стиле: буквы сообщения должны появляться по одной, с некоторой задержкой.
5. Написать программу, которая проверяет, является ли введенная с клавиатуры строка шестнадцатеричным числом.

Лабораторная работа №8 «Разработка алгоритма, составление, отладка и выполнение программы с использованием пользовательских функций. Разработка алгоритма, составление, отладка и выполнение программы с использованием рекурсивных функций. Разработка алгоритма, составление, отладка и выполнение программы с использованием функций с произвольным числом параметров»

Задание

В каждом задании данного раздела требуется описать процедуру или функцию и затем использовать ее для обработки исходных данных.

Все параметры любой функции считаются входными. Для процедур всегда указывается, какие параметры являются выходными (или одновременно входными и выходными); если о виде параметра процедуры ничего не сказано, то он считается входным.

Одномерные и двумерные массивы

При вводе исходного массива вначале следует ввести его размер (одно число для одномерных массивов, два числа — количество строк и столбцов - для двумерных массивов-матриц), а затем — все его элементы.

Если в задании явно не указывается размер одномерного массива, являющегося параметром процедуры или функции, то предполагается, что этот размер может меняться в пределах от 1 до 10. Для двумерных массивов-матриц также предполагается, что число их строк и столбцов может меняться от 1 до10. Порядковые номера начальных элементов как одномерных, так и двумерных массивов всегда считаются равными 1.

При описании процедур, выполняющих преобразование массива, не следует использовать вспомогательный массив того же размера.

Варианты

1. Описать функцию MinInt(A, N) целого типа, находящую минимальный элемент целочисленного массива A размера N. С помощью этой функции найти минимальные элементы массивов A, B, C размера NA, NB, NC соответственно.
2. Описать функцию NMax(A, N) целого типа, находящую номер максимального элемента вещественного массива A размера N. С помощью этой функции найти номера максимальных элементов массивов A, B, C размера NA, NB, NC соответственно.
3. Описать процедуру NMinmax(A, N, NMin, NMax), находящую номера минимального и максимального элемента вещественного массива A размера N. Выходные параметры целого типа: NMin (номер минимального элемента) и NMax (номер максимального элемента). С помощью этой процедуры найти номера минимальных и максимальных элементов массивов A,B, C размера NA, NB, NC соответственно.
4. Описать процедуру Invert(A, N), меняющую порядок следования элементов вещественного массива A размера N на противоположный (инвертирование массива). Массив A является входным и выходным параметром. С помощью этой процедуры инвертировать массивы A, B, C размера NA,NB, NC соответственно.
5. Описать процедуру Smooth1(A, N), выполняющую сглаживание вещественного массива A размера N следующим образом: элемент AK заменяется на среднее арифметическое первых K исходных элементов массива A. Массив A является входным и выходным параметром. С помощью этой процедуры выполнить пятикратное сглаживание данного массива A размера N, выводя результаты каждого сглаживания.

Лабораторная работа №9 «Разработка, отладка и выполнение программы с использованием модулей пользователя»

Модули C++ — это попытка уменьшить потребность в одной конкретной директиве препроцессора, #include. #include позволяет нам разделить исходный код на логические части — в частности, интерфейс (обычно расположенный в файле ".h" или "header") и реализацию (обычно расположенную в файле ".cpp" или "source"). Разделение на заголовочный и исходный файлы дает огромное количество преимуществ, включая:

- Создание многократно используемых библиотек кода, которые являются краеугольным камнем C и C++
- Разделение интерфейса и его реализации
- Модулирование кода, что потенциально ускоряет время компиляции (при правильном использовании)
- Организация кода в логические и многократно используемые части.

Задание

Доработать предыдущую лабораторную работу №8. Разбить программу на модули.

Лабораторная работа №10 «Разработка, отладка и выполнение программы обработки текстовых файлов»

Задание

Предварительно подготовить (программно или с помощью текстового редактора) текстовый файл, содержимым которого является массив (матрица) из чисел. В соответствии с индивидуальным заданием написать программу обработки содержимого файла.

Для каждого из вариантов задания необходимо выполнить следующие действия. Просмотреть содержимое исходного файла. Считать содержимое файла в одномерный (двумерный) динамический массив. Обработать динамический массив согласно варианту задания. Полученный результат записать в конец исходного файла.

Варианты

1. Дано имя файла и целые положительные числа N и K. Создать текстовый файл с указанным именем и записать в него N строк, каждая из которых состоит из K символов «*» (звездочка).

2. Дано имя файла и целое число N (0 < N < 27). Создать текстовый файл с указанным именем и записать в него N строк: первая строка должна содержать строчную (то есть маленькую) латинскую букву «a», вторая —

буквы «ab», третья — буквы «abc» и т. д.; последняя строка должна содержать N начальных строчных латинских букв в алфавитном порядке.

3. Дано имя файла и целое число N ($0 < N < 27$). Создать текстовый файл с указанным именем и записать в него N строк длины N; строка с номером K (K = 1, …, N) должна содержать K начальных прописных (то есть заглавных) латинских букв, дополненных справа символами «*» (звездочка). Например, для N = 4 файл должен содержать строки «A***», «AB**», «ABC*», «ABCD».

4. Дан текстовый файл. Вывести количество содержащихся в нем символов и строк (маркеры концов строк EOLN и конца файла EOF при подсчете количества символов не учитывать).

5. Дана строка S и текстовый файл. Добавить строку S в конец файла.

Лабораторная работа №11 «Разработка, отладка и выполнение программы обработки файлов с типом»

Задание

Сформировать бинарный файл, содержимым которого являются вещественные матрицы (структуры матриц).

При этом количество матриц (компонент структуры) и их размерность вводится в процессе выполнения программы с клавиатуры. В соответствии с индивидуальным заданием обработать содержимое полученного файла. Для каждого из вариантов задания вывести содержимое исходного файла на экран до и после преобразования. При работе с матрицами, где это возможно, использовать динамические массивы.

Варианты заданий

1. В первом файле хранится k матриц, во втором - l матриц размерности $n \times m$. Те матрицы из первого файла, у которых a00=0 , перенести в конец второго файла. Вывести на экран содержимое первого и второго файлов.

2. В первом файле хранится k матриц, во втором l матриц размерности $n \times m$. Убрать из файла, в котором больше матриц, лишние матрицы в третий файл. Вывести на экран содержимое первого файла; второго файла; третьего файла.

3. Файл состоит из k компонент структуры, где каждая компонента содержит две матрицы:первая размерности $n \times m$, вторая размерности $m \times l$.

Получить k произведений соответствующих матриц и записать их во второй файл. Вывести на экран содержимое первого и второго файлов.

4. В первом файле хранится k матриц, во втором l матриц размерности $n \times m$. Добавить во второй файл те матрицы из первого, которых нет во втором. Вывести на экран содержимое первого и второго файлов.

5. В первом файле хранится k матриц из n строк и n +1 столбцов каждая(последний столбец- столбец свободных членов). Во втором файле хранится k векторов- результатов решений соответствующих систем ЛАУ с матрицами из первого файла. Вывести на экран покомпонентно исходную

систему уравнений и результат, проверив его предварительно; добавить в файлы новые данные; удалить ненужную информацию.

Лабораторная работа №12 «Разработка алгоритмов, составление, отладка и выполнение программы сортировки и поиска (массивы, строки)»

Задание

Предварительно подготовить (программно или с помощью текстового редактора) текстовый файл, содержимым которого является массив (матрица) из чисел. В соответствии с индивидуальным заданием написать программу обработки содержимого файла.

Для каждого из вариантов задания необходимо выполнить следующие действия. Просмотреть содержимое исходного файла. Считать содержимое файла в одномерный (двумерный) динамический массив. Обработать динамический массив согласно варианту задания. Полученный результат записать в конец исходного файла.

Варианты

Вариант сортировки

1. Сортировка выбором(SelectSort).
2. Сортировка пузырьком(BubbleSort) и ее улучшения.
3. Сортировка простыми вставками(InsertSort).
4. Сортировка обменом.
5. Сортировка Шелла (ShellSort).
6. Мирамидальная сортировка (HeapSort).
7. Быстрая сортировка (QuickSort).
8. Поразрядная сортировка(RadixSort).
9. Метод Хоара.

Вариант поиска

1. Линейный поиск.
2. Поиск с барьером.
3. Двоичный (бинарный) поиск.
4. Поиск минимуму.
5. Поиск максимума.

Лабораторная работа №13 «Разработка, отладка и выполнение программы с использованием подпрограмм с различными типами параметров»

Задание

Разработать программу, которая содержит:

– подпрограмму пользователя для обработки элементов массива (имя массива и размерность параметры функции);
– подпрограмму пользователя для обработки строки, как параметра функции;
– подпрограмму для работы с файлом (имя файла как параметри функции).

Лабораторная работа №14 «Разработка алгоритма, составление, отладка и выполнение программы с использованием структур (массивов структур)»

Задание

Организовать динамический массив структур (в соответствии с вариантом). Объявить указатель на массив структур. Организовать работу с полями структур через данный указатель.

В программе реализовать:

1) Ввод массива структур.
3) Вывод массива структур.
4) Изменение заданной структуры.
5) Удаление структуры из массива.
6) Вывод на экран массива структур.
7) Выход.

Варианты:

1. Структура «Автосервис»: регистрационный номер автомобиля, марка, пробег, мастер, выполнивший ремонт, сумма ремонта.
2. Структура «Сотрудник»: фамилия, имя, отчество; должность; год рождения; заработная плата.
3. Структура «Государство»: название; столица; численность населения; занимаемая площадь.
4. Структура «Человек»: фамилия, имя, отчество; домашний адрес; номер телефона; возраст.
5. Структура «Читатель»: Фамилия И.О., номер читательского билета, название книги, срок возврата.

Лабораторная работа №15 «Разработка алгоритма, составление, отладка и выполнение программы обработки линейных связанных списков»

Задание

Написать программу, создающую произвольный список путем добавления его элементов в начало. Включите эту процедуру в программу, решающую задачу создания списка путем добавления элементов в конец списка. Добавьте меню. Протестируйте программу на наличие ошибок.

Варианты

1. Написать программу, содержащую процедуры формирования и просмотра списка и функцию вычисления среднего арифметического элементов непустого списка.
2. Написать программу, содержащую процедуры формирования и просмотра списка и подпрограмму проверки наличия в списке заданного числа.
3. Написать программу, содержащую процедуры формирования и просмотра списка и функцию, подсчитывающую количество слов списка, которые начинаются и оканчиваются одной и той же литерой.
4. Написать программу, содержащую процедуры формирования и просмотра списка и функцию, подсчитывающую количество слов списка, которые начинаются той же литерой что и следующее слово.
5. Написать программу, содержащую процедуры формирования и просмотра списка и функцию, подсчитывающую количество различных значений информационной части уже существующего списка.

Лабораторная работа №16 «Программирование с использованием древовидных структур данных»

Задание. Варианты

1. Создайте программой числовое двоичное дерево. Опишите рекурсивную логическую функцию, проверяющую наличие заданного числа в сформированном дереве. В программе используйте подпрограммы.
2. Создайте программой числовое двоичное дерево. Опишите рекурсивную числовую функцию, подсчитывающую сумму элементов дерева. В программе используйте подпрограммы.
3. Создайте программой числовое двоичное дерево. Опишите функцию, которая находит наибольший элемент непустого дерева. В программе используйте подпрограммы.
4. Напишите программу, содержащую процедуру, которая каждый отрицательный элемент дерева заменяет на положительный, а положительный превращает в ноль.

5. Напишите программу, содержащую процедуру, которая каждый элемент дерева возводит в квадрат.

3.Методические указания к выполнению курсовой работы по дисциплине «Основы алгоритмизации и программирования» для студентов дневной и заочной форм обучения специальности 6-05-0612-03 Системы управления информацией, и для студентов дневной формы обучения по специальностям 6-05-0611-03 Искусственный интеллект, 6-05-0612-01 Программная инженерия.

1 Основные сведения

Курсовая работа предусмотрена программой изучения дисциплины «Основы алгоритмизации и программирования», является самостоятельной работой студента, позволяет оценить качество знаний и отражает приобретенные студентом практические навыки.

Тема и задание курсовой работы формулируется руководителем курсовой работы и утверждается заведующим кафедрой.

Перед студентом ставится задача разработать приложение в среде программирования Visual Studio с целью решения конкретной задачи (задач).

Результатом курсовой работы являются:

1. Исполняемый файл программы и ее полный текст (листинг) на носителе.

2. Пояснительная записка.

2 Структура пояснительной записки к курсовой работе

Проект должен иметь четкое и логическое построение. Он должен включать следующие структурные элементы (в порядке их представления в работе):

1 ТИТУЛЬНЫЙ ЛИСТ пояснительной записки

2 БЛАНК ЗАДАНИЯ к курсовой работе (заполненный руководителем и подписанный обеими сторонам – и преподавателем, и студентом, и утвержденный заведующим кафедрой)

3 СОДЕРЖАНИЕ (оглавление)

4 ВВЕДЕНИЕ

5 ОСНОВНАЯ ЧАСТЬ

6 ЗАКЛЮЧЕНИЕ

7 СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

8 ПРИЛОЖЕНИЯ (ПРИЛОЖЕНИЕ А – ТЕКСТ ПРОГРАММЫ, ПРИЛОЖЕНИЕ Б – ГРАФИЧЕСКИЙ МАТЕРИАЛ (ГОСТ 19.701-90))

Офоромление титульного листа

Титульный лист является первым листом работы (нумерация на нем не проставляется). Титульный лист должен содержать все установленные реквизиты: тему курсовой, фамилию и инициалы полностью, напротив фамилии необходимо поставить подпись автора, шифр (составление шифра), количество листов указывается для пояснительной записки без приложений (они нумеруются отдельно).

Титульный лист оформляется с рамкой. Отступы для рамки: левое поле – 20 мм, правое поле – 5 мм, верхнее и нижнее поле – 5 мм.

Оформление бланка задания

Бланк задания на тему курсовой работы, подписанный преподавателем и студентом, располагается за титульным листом.

Лист задания оформляется на одном листе (2 страницы на одном листе с двух сторон), листы не нумеруются.

Даты выдачи и сроки сдачи студентом законченной курсовой работы согласовать с руководителем курсовой работы, впечатать или вписать тему курсовой работы согласно теме выданной руководителем, ФИО, необходимо поставить подпись на обороте листа задания.

Оформление содержания

Содержание курсовой работы является третьей по порядку страницей, которая оформляется, используя возможности программы Microsoft Word.

Слово «сдержание» записывают в виде заголовка после отступа табуляции (15 мм).

В содержание включаются все заголовки, имеющиеся в работе, в том числе список использованных источников и приложения. Название разделов и подразделов следует приводить в полном соответствии с их названиями, указанными в тексте проекта.

СОДЕРЖАНИЕ курсовой работы должно включать ВВЕДЕНИЕ, три–четыре раздела ОСНОВНОЙ ЧАСТИ, ЗАКЛЮЧЕНИЕ, СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ и ПРИЛОЖЕНИЯ.

В содержание включаются разделы и подразделы с указанием страниц.

Содержание оформляется с большим штампом под рамку. Необходимо внимательно заполнить в рамке поля штампа: вписать свою фамилию, тему, количество страниц и шифр (составление шифра см. ниже), номер страницы указывается в штампе.

При заполнении штампа тема проекта должна в точности соответствовать заданию на курсовую работу.

Шифр документа содержит буквы КР, номер зачетной книжки, номер программы, разработанной студентом в вузе (присваивается студентом самостоятельно, обычно нумеруются все курсовые работы и проекты), номер

документа ПЗ – 81 и версия документа ПЗ – 00. Пример: КР.567.89977 – 01 81 00.

При размещении содержания на нескольких листах штамп высотой 40 мм выполняется только на первом листе содержания.

Все остальные страницы пояснительной записки оформляются с маленьким штампом под рамку.

Оформление введения

Во введении кратко характеризуется проблема, решению которой посвящена курсовая работа, определяются цель и задачи, которые необходимо решить для раскрытия темы, описываются средства, посредством которых реализуется разрабатываемая программа, например среда разработки Visual Stuio.

Общий объем введения составляет 1–2 страницы. Введение, как и заключение, рекомендуется писать после полного завершения основной части.

Рекомендации по оформлению общей части курсовой работы

Общий объем курсовой работы без учета приложений должен составлять 35–40 страниц.

В общем случае общая часть пояснительной записки курсовой работы должна содержать описание следующих этапов создания программного средства:

1. Постановка задачи.
2. Разработка алгоритмов.
3. Разработка программы.
4. Тестирование.

Постановка задачи

Общее наименование задачи, описание подзадач и запросов согласно варианту.

Разработка алгоритмов

В разделе дается обобщенное словесное описание алгоритма решения поставленной задачи, излагаются основные требования к алгоритму и пути их реализации. Приводится схема алгоритма, состоящая из укрупненных модулей. Дается пояснение назначения и состава каждого модуля. Обобщённый алгоритм обычно использует обозначения и термины исходной задачи. Далее каждый модуль детализируется. Выделяются укрупненные команды, реализуемые по вспомогательным алгоритмам. Тот же подход применяется при разработке вспомогательных алгоритмов. В разделе приводятся описания процедур. Результатом должна стать детализированная модель системы, именно данная модель должна «служить» исходной информацией для написания программного кода.

Словесное описание алгоритмов, определенных в первом разделе «Постановки задачи», и подзадач. На основе этого раздела строится блок-

схема, алгоритмическая запись решения задачи, блок-схема может быть очень упрощенной – 8–10 блоков.

Разработка программы

Приводится следующая информация:

1 Обоснование выбранных средств и инструментов разработки указать среду разработки, ОС, версию, описать все используемые стандартные функции и библиотеки (кратко – что используем и для чего, не более одной страницы).

2 Спецификация программы. Спецификация, определение требований к программе – один из важнейших этапов, на котором подробно описывается исходная информация, формулируются требования к результату, поведение программы в особых случаях (например, при вводе неверных данных).

3 Текст программы.

4 Описание программы. Необходимо описать все разработанные вами модули и функции. Описание включает название, входные параметры, возвращаемые данные и назначение (выполняемое действие, запрос).

Программа должна быть по возможности универсальной. Входные форматы должны быть разработаны с учетом максимального удобства для пользователя и минимальной возможности ошибок. Порядок переменных и форматы данных, привычные для пользователя, помогут избежать ошибок и облегчат использование программ.

При написании программы следует применять операторы, позволяющие использовать основные алгоритмические структуры.

Не следует забывать о хорошем стиле программирования. После заголовка процедуры или функции записывается комментарий, содержащий поясняющий текст, а именно: назначение подпрограммы; перечень и назначение формальных параметров, их тип. Комментариями должны быть снабжены и основные смысловые блоки программы или подпрограммы.

Результатом данного этапа является программное приложение, которое обладает требуемой функциональностью и способно решать нужные задачи в конкретной предметной области.

Тестирование

На этапе тестирования программы проводится проверка работоспособности программы на некоторой совокупности исходных данных или при некоторых специальных режимах эксплуатации. Результатом является повышение надежности программы, исключающее возникновение критических ситуаций.

Описание входных и выходных данных

Приводится структура файла, на котором проводится тестирование, можно фрагмент содержания файла – 2-3 строки данных, не более. Также описание структуры выходного файла.

Результаты тестирования

Указать среду тестирования (аппаратное и программное обеспечение), описание результатов тестирования всех задач согласно заданию на курсовой, в том числе и по варианту. Все тесты сопровождаются скриншотами результатов выполнения.

Оформление заключения

Заключение содержит перечисление основных результатов, характеризующих полноту решения поставленных задач и подводящих итог содержания курсовой работы, рекомендации по конкретному использованию результатов работы; ее значимость. Результаты следует излагать в форме констатации фактов с использованием слов «изучены», «сформулированы», «разработаны», «показаны», «предложены», «подготовлены» и т. п.

Текст должен быть кратким и ясным.

Оформление списка использованных источников

В разделе «список использованных источников» приводится список литературы, использованной ходе выполнения курсовой работы. Данный раздел должен содержать перечень источников, цитируемых и используемых при написании курсовой работы, которые следует располагать по алфавиту авторов или заглавий (в случае четырех авторов и более). Можно ссылаться на электронные источники. Сведения об источниках необходимо привести в соответствии с требованиями ГОСТ 7.1–2003.

Оформление приложений

Приложения оформляются как продолжение работы на последующих ее страницах.

В приложения следует включать вспомогательный материал, необходимый для полноты восприятия: формы; схемы алгоритмов и программ, разработанных в процессе выполнения работы; иллюстрации вспомогательного характера; текст программы.

Текст программы должен содержать достаточное количество комментариев (вводных и контекстных). Во вводном комментарии программного модуля должны содержаться назначение модуля, сведения о разработчике, сведения об обрабатываемых в данном модуле данных и т. п. Каждая процедура и функция пользователя должны также содержать сведения о назначении данной подпрограммы, ее применении, описание формальных параметров.

Приложения нумеруются с помощью букв русского алфавита, за исключением букв Ё, З, Й, О, Ч, Ь, Ы, Ъ. Допускается использовать размер шрифта меньше, чем в ПЗ

Приложения можно оформлять без рамок (случай иллюстрирующего приложения), в этом случае код документа пишется в виде верхнего колонтитула размером шрифта 11, на этой же строке колонтитула ставится номер страницы приложения с выравниванием вправо. Следует различать два вида приложений: документированные и иллюстрирующие.

Оформление документированных приложений

Документированными являются приложения, которые по ГОСТ 19.101-87 выделены как отдельный вид программного документа и ему присвоен код обозначения.

Структура приложения:

1 Титульный лист

2 Содержание

3 Основные разделы

Первым листом документированного приложения является титульный лист, образец

которого можно посмотреть в ПРИЛОЖЕНИИ Д на примере титульного листа текста про-

граммы. Названием программного продукта на титульном листе всех приложений должна быть тема курсовой работы по приказу.

На титульном листе в виде верхнего колонтитула, выровненного вправо, должен быть указан номер приложения, например ПРИЛОЖЕНИЕ Б. На титульном листе номер страницы не ставится.

В содержании указываются разделы и, по желанию, подразделы с указанием страниц. Перечень основных разделов в целом соответствует ГОСТу, но допускается объединять разделы (с сохранением всей требуемой информации) или вводить новые разделы.

Оформление иллюстрирующих приложений

К иллюстрирующим относится любое приложение, которое не предусмотрено ГОСТом. Иллюстрирующее приложение считается продолжением ПЗ и имеет шифр 81.

Нумерация страниц сквозная с ПЗ. Помещается иллюстрирующее приложение сразу после ПЗ, т. е. после списка сокращений (если он присутствует). Номер первого приложения – А, далее по порядку совместно с документированными приложениями.

Титульный лист для иллюстрирующего приложения отсутствует. На первом листе приложения сверху по центру следует написать заголовок приложения, определяемый студентом, исходя из назначения приложения.

На каждом листе иллюстрирующего приложения в виде верхнего колонтитула следует написать шифр (совпадает с шифром ПЗ) по центру, справа на этой строке – слова ПРИЛОЖЕНИЕ А, на следующей строке – номер страницы с выравниванием вправо (не забыть про сквозную нумерацию). Заголовок приложения указывается только на первом листе.

Сначала размещаются все иллюстрирующие приложение, затем –

Оформление объектов в тексте (рисунков, таблиц, формул) производится по правилам оформления ПЗ. К нумерации объектов перед номером раздела добавляется буква – номер приложения, например, формула (А2.1).

Перечень приложений курсовой работы

Данная курсовая работа предусматривает: ПРИЛОЖЕНИЕ А ТЕКСТ ПРОГРАММЫ и ПРИЛОЖЕНИЕ Б ГРАФИЧЕСКИЙ МАТЕРИАЛ. ПРИЛОЖЕНИЕ А ТЕКСТ ПРОГРАММЫ В ПРИЛОЖЕНИИ А ТЕКСТ ПРОГРАММЫ необходимо оформить титульный лист (оформляется без рамки), исправить шифр (код документа 12),указать ФИО, указать количество листов в приложении (независимая нумерация от пояснительной записки), указать тему согласно заданию на курсовую работу.

Содержание приложения оформляется без рамки и без штампа.

ПРИЛОЖЕНИЕ Б ГРАФИЧЕСКИЙ МАТЕРИАЛ

Схема алгоритма по ГОСТ 19.701-90 (ИСО 5807-85) Схемы алгоритмов, программ, данных и систем. Обозначения условные и правила выполнения (обощенная, достаточно на один лист – чертеж А4) титульный лист оформляется без рамки аналогично как для ПРИЛОЖЕНИЯ А, вместо «ТЕКСТ ПРОГРАММЫ» указываем название

ПРИЛОЖЕНИЯ Б («ГРАФИЧЕСКИЙ МАТЕРИАЛ»), в шифре указываем нужный ГОСТ (в данном случае 90). Схема оформляется под большую рамку как СОДЕРЖАНИЕ, в штампе необходимо заменить «Пояснительная записка» на «Графический материал», тему курсовой работы указать в соответствии с листом задания, в шифре указываем нужный ГОСТ (в данном случае 90).

1 ОБЩИЕ ТРЕБОВАНИЯ К ОФОРМЛЕНИЮ КУРСОВОЙ РАБОТЫ

1.1 ОФОРМЛЕНИЮ ТЕКСТА ПОЯСНИТЕЛЬНОЙ ЗАПИСКИ

Курсовая работа оформляется на стандартных листах формата А4.

Текст излагается от третьего лица или в форме безличных предложений без использования личных местоимений.

Слова «СОДЕРЖАНИЕ», «ВВЕДЕНИЕ», «ЗАКЛЮЧЕНИЕ», «СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ» записывают от отступа табуляции (15 мм) прописными буквами и включают в содержание курсовой работы. Данные заголовки не нумеруют.

К оформлению работы предъявляются следующие требования:

1. Односторонняя печать.
2. Текст размещается по ширине листа.
3. Поля: слева не менее 25 мм, справа не менее 10 мм, снизу и сверху не менее 25 мм.
4. Межстрочный интервал 1,2 (допускается использование междустрочного интервала 1,0 в таблицах и рисунках).
5. Шрифт Times New Roman, кегль – 13 пт.
6. Номер страницы проставляется сверху, справа, шрифт 10 пунктов.
7. Абзацный отступ должен быть одинаковый – 15 мм.
8. Каждая глава нумеруется и начинается с новой страницы.

9.Все заголовки глав и параграфов должны быть выделены полужирным шрифтом и без точки.

10.Расстояние между нумеруемым заголовком любого уровня и текстом – 3 межстрочных интервала (интервал после абзаца – 39 пт. для шрифта для шрифта 13). Если между двумя заголовками текст отсутствует (например, между заголовками раздела и подраздела), то расстояние между ними – 2 межстрочных интервала.

11. Сокращения слов в таблицах и рисунках не допускается (разрешается в таблицах и рисунках необходимые надписи делать более мелким шрифтом, чем в текстовой части).

12.Ссылки на использованную литературу приводятся в тексте в квадратных скобках (например [1], [4–8]), если ссылки идут по тексту пояснительной записки. В случае, если по тексту пояснительной записки ссылки на использованную литературу отсутствуют, то допускается в разделе «СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ» перечислить литературу согласно ГОСТу.

13. На всех листах, кроме титульного, бланках задания и приложений, выполняется рамка (20–5–5–5) со штампом высотой 15 мм, за исключением первой рамки, которая выполняется на первом листе содержания со штампом высотой 40 мм (ГОСТ 2.104-68).

14. Для оформления заголовков, списков, формул, подписей таблиц, рисунков и т.п. выбирается стиль на усмотрение студента, не противоречащий данному документу, который должен быть применен для всего документа, т.е. все единицы текста должны быть оформлены единообразно.

Порядок нумерации страниц курсовой работы следующий: на первой странице располагается титульный лист (номер страницы не ставится), затем постранично (последовательно) размещаются задание на разработку курсовой работы, содержание и т. д.

Нумерация страниц работы начинается с титульного листа.

Все нумерации (страниц, глав, параграфов, рисунков и т. д.) выполняют только арабскими цифрами. При необходимости в оглавление и, соответственно, в основную часть вводят рубрикации типа 1.1 или 2, 2.2, 2.2.1и т. п.

1.2 ОФОРМЛЕНИЕ ЗАГОЛОВКОВ РАЗДЕЛОВ И ПОДРАЗДЕЛОВ

Все основные разделы имеют порядковую нумерацию, кроме ВВЕДЕНИЯ, ЗАКЛЮЧЕНИЯ и СПИСКА ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ: <НОМЕР РАЗДЕЛА> НАЗВАНИЕ РАЗДЕЛА

Если название раздела не помещается в одну строку, то вторая строка начинается под первой буквой названия раздела:

<НОМЕР РАЗДЕЛА> НАЗВАНИЕ РАЗДЕЛА, КОТОРЫЙ НЕ ПОМЕЩАЕТСЯ В ОДНУ СТРОКУ ТЕКСТА ПОЯСНИТЕЛЬНОЙ ЗАПИСКИ

При необходимости основные разделы ПЗ можно разбить на подразделы. Подразделы нумеруются:

<НОМЕР РАЗДЕЛА>.<НОМЕР ПОДРАЗДЕЛА> НАЗВАНИЕ ПОДРАЗДЕЛА

Каждый раздел ПЗ должен начинаться с новой страницы. Подразделы разделяются пустой строкой.

Названия разделов и подразделов печатаются с абзаца жирным шрифтом того же кегля, что и основной текст и отделяются от текста пустой строкой; выравнивание по ширине.

Оформление рисунков

Рисунки нумеруются в пределах раздела, содержат номер раздела, точку, номер рисунка в разделе. На каждый рисунок в тексте ПЗ должна быть ссылка.

Примеры:

Разработанная структурная схема приведена на рисунке 3.1

или … (см. рисунок 3.1).

Рисунки могут располагаться в любом месте ПЗ, но обязательно после ссылки.

Под рисунком помещается подпись по центру, которая содержит слово «Рисунок», пробел, номер рисунка (номер раздела, точка, номер рисунка в разделе), тире, название рисунка (без точки).

Рисунки отделяются от текста одной пустой строкой сверху и снизу (после подписи).

Допускается выносить рисунок на отдельный лист и поворачивать его по часовой стрелке (в альбомной ориентации), чтобы верх рисунка находился около переплета.

В этом случае подпись рисунка выполняется также в альбомной ориентации, а расположение рамки и штампа не меняется.

Допускается располагать рисунок не нескольких страницах, при этом на первой странице делается обычная подпись рисунка, а на последующих подпись выглядит так:

Продолжение рисунка 3.1

и располагается по центру (без названия).

Оформление таблиц

Таблицы нумеруются в пределах раздела, содержат номер раздела, точку, номер

таблицы в разделе. Таблица должна быть отцентрирована относительно текста. Ячейки могут объединяться. Допускается уменьшать шрифт в таблице. Минимальное расстояние между строками таблицы – 8 мм.

На каждую таблицу в тексте ПЗ должна быть ссылка. Примеры:

Результаты экспериментов приведены в таблице 3.1

или … (см. таблицу 3.1).

Таблицы могут располагаться в любом месте ПЗ, но обязательно после ссылки. Над таблицей помещается заголовок по центру, который содержит слово «Таблица», пробел, номер таблицы (номер раздела, точка, номер таблицы в разделе), тире, название таблицы (без точки).

Таблицы отделяются от текста одной пустой строкой сверху (до заголовка) и снизу.

Допускается выносить таблицу на отдельный лист и поворачивать его по часовой стрелке (в альбомной ориентации), чтобы верх таблицы находится около переплета.

В этом случае заголовок таблицы выполняется также в альбомной ориентации, а расположение рамки и штампа не меняется.

Не допускаются пустые графы в таблице. Если по смыслу не требуется указывать значение, то в этой графе нужно поставить прочерк.

Допускается располагать таблицу на нескольких страницах, при этом на первой странице делается обычный заголовок, а на последующих заголовок выглядит так:

Продолжение таблицы 3.1

и располагается по центру (без названия).

При разбиении таблицы формируются номера столбцов, которые указываются во второй строке под шапкой таблицы (см. таблицу 3.1), в продолжении таблицы шапка не повторяется, повторяются только номера столбцов (номера столбцов могут быть латинскими заглавными буквами либо арабскими цифрами):

Оформление формул

Формула пишется на отдельной строке и отделяется от текста одной пустой строкой сверху и снизу, начинается в строке с отступом, равным абзацу. Формулы нумеруются в пределах раздела (номер раздела, точка, номер формулы), номер помещается в последней строке формулы в круглых скобках с правой стороны на расстоянии абзацного отступа от правого края текста.

Ссылка на формулу может выглядеть так:

… по формуле (3.1).

Если нет необходимости использовать ссылку на формулу в тексте, то ее можно не нумеровать.

Каждый символ, входящий в формулу, должен быть пояснен после формулы. Первая строка пояснения начинается со слова «где», далее после следует символ, тире, его пояснение. Каждый символ начинается с новой строки под предыдущим символом.

Пояснение может отсутствовать только в случае, если в ранее приведенных формулах этот символ уже использовался и пояснение его было приведено.

Если необходимо привести численные вычисления, то сначала приводится формула с пояснением обозначений, затем в последующих строках приводится числовое выражение и результат. При этом единицы измерений результата пишутся в круглых скобках.

Оформление списков

Допускается использовать нумерованные и маркированные списки и их комбинацию.

В качестве маркеров можно использовать тире, точку, квадрат, в качестве номера пункта списка – арабские цифры, русские и латинские символы, в качестве разделителя – точку, тире, скобку.

Оформление вложенного списка должно отличаться от оформления списка верхнего уровня. Следует придерживаться единообразного оформления списков одного уровня во всей ПЗ. Например, маркированные списки изображать с квадратом, нумерованные списки первого уровня – с арабской цифрой и скобкой, вложенный нумерованный список – латинской буквой и скобкой.

Оформление фрагментов программ

Можно включать в текст ПЗ фрагменты программ, файлов, консольные команды и т. д.

Их рекомендуется выносить на отдельные строки, отделяя от текста одной пустой строкой сверху и снизу. Начинать на строке их следует с абзацного отступа. Рекомендуется использовать шрифт Courier New. Примеры:

```
lea dx,a
mov ax,dx
C:> echo “Good Morning”
```

ЗАДАНИЕ НА КУРСОВУЮ РАБОТУ

При выполнении курсовой работы студент выполняет все задания. При отсутствии хотя бы одного задания курсовая работа не может быть зачтена.

Номер варианта индивидуального задания на курсовую работу определяется исходя из последних двух цифр номера зачетной книжки студента.

Не допускается выполнение студентом вариантов заданий, не соответствующих последним двум номерам его зачетной книжки.

Обобщенная тема курсовой работы: «Обработка массивов структурированных данных «вписать название множества данных согласно варианту».

Необходимо оформить задание на курсовую работу на бланке ЗАДАНИЯ установленного образца(печатается на одном листе с двух сторон), согласовать его с руководителем курсовой работы по дисциплине «Основы алгоритмизации и программирования».

Курсовой принимается на проверку не позднее, чем за две недели до начала летней сессии. Списанные работы не проверяются, а просто возвращаются с пометкой «Списано». Если вы приносите работу, текст пояснительной либо программа которой уже присутствовали в работе у другого студента(ки), то она считается списанной.

Защита курсовых работ проводится в течение летней сессии и заключается в демонстрации работы программы и ответы на вопросы по разработке программы.

Обобщенная формулировка задания. Выбрать предметную область (в соответствии с вариантом) для базы данных и доработать структуру для описания отдельных записей базы данных. Выбранная структура должна иметь не менее пяти полей (элементов) двух и более типов, включая пользовательский тип union и enum.

Работа содержит описание разработанного студентом программного обеспечения по обработке заданного массива структур. Для всех вариантов обеспечить реализацию следующих запросов к заданному массиву структурированной информации:

1. Ввод информации из текстового файла в массив указателей на записи.
2. Добавление новых элементов в конец массива.
3. Просмотр всех элементов массива.
4. Вывод информации из массива в текстовый файл.
5. Корректировка полей выбранного элемента.
6. Удаление выбранного элемента.

Максимальная оценка за реализацию данного задания составляет 6 (шесть) баллов.

Для получения оценки из расчета 10 баллов необходимо обеспечить в соответствии

с заданием варианта реализацию еще трех запросов:

1. Удаление элементов по условию (поле < или > заданного значения).
2. Сортировка массива по числовому полю.
3. Вставка нового элемента перед выбранным элементом.
4. Вставка нового элемента после выбранного элемента.
5. Замена выбранного элемента.
6. Удаление элементов, начиная от выбранного.
7. Просмотр элементов и вычисление среднего на множестве тех элементов, которые попадают в заданный диапазон по заданному полю (поле типа float или longint).

8. Просмотр элементов и вычисление минимума и максимума на множестве тех элементов, которые попадают в заданный диапазон по заданному полю (поле типа float или longint).

Условия и ограничения

1. Главную процедуру программы с реализацией простейшего меню следует определить в отдельном модуле.

2. Процедуры, реализующие запросы, должны быть размещены в одном или более модулях.

3. Глобальные данные использовать нельзя.

4. На экран выводить элементы в виде таблицы (один элемент – одна строка таблицы).

5. Если после выполнения запроса изменяется хотя бы один элемент, то заканчивать запрос выводом всего множества элементов.

6. Тестами к заданиям служат 2 текстовых файла с правдоподобной информацией.

Перечень тем курсового проекта:

Вариант № 1

Массив данных – «Стадионы города».

1. Название (char[]);
2. вместимость в тысячах человек (float);
3. основной вид спорта (char[]).

Выполнить из общего списка запросы с номерами 1, 4 и 8.

Вариант № 2

Массив данных – «Города страны».

1. Количество жителей в тысячах (float);
2. название (char[]);
3. главная достопримечательность (char[]).

Выполнить из общего списка запросы с номерами 2, 3 и 6.

Вариант № 3

Массив данных – «Популярные телепередачи».

1. Название (char[]);
2. длительность в минутах (int);
3. рейтинг (float).

Выполнить из общего списка запросы с номерами 2, 5 и 7.

Вариант № 4

Массив данных – «Картины на выставке».

1. Автор (char[]);
2. название (char[]);
3. стоимость (longint).

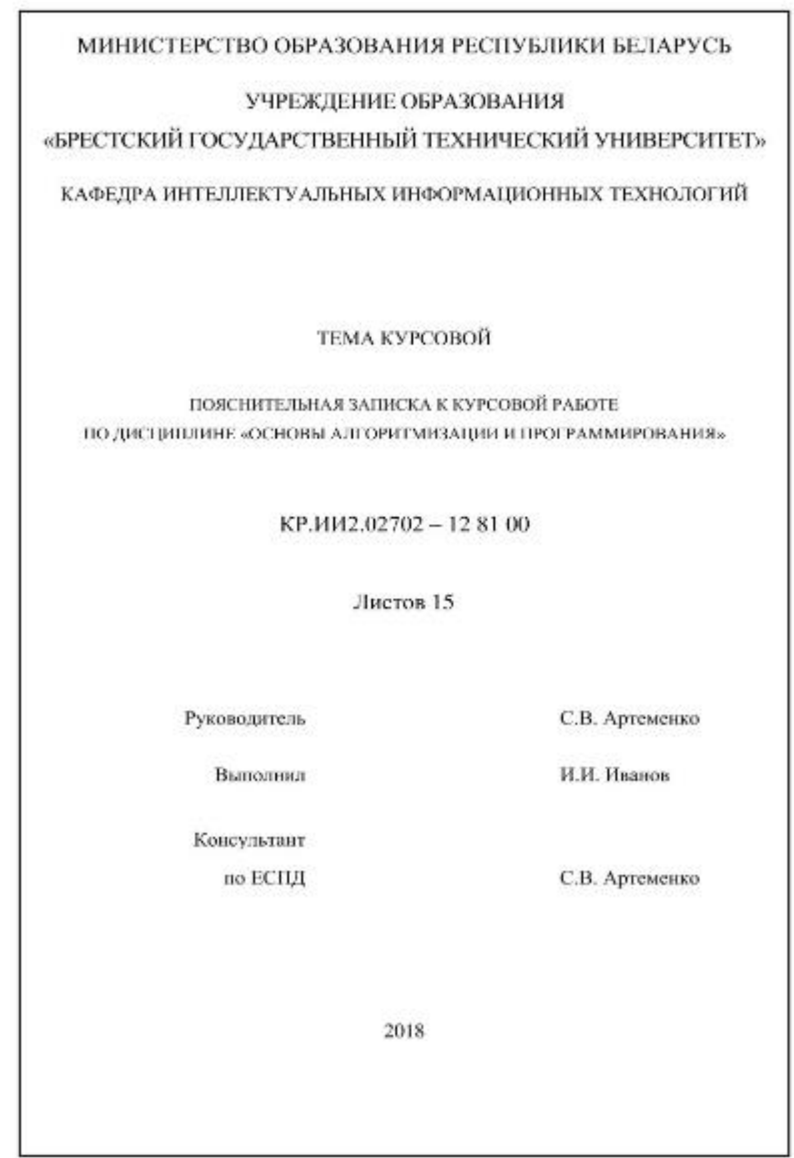
Выполнить из общего списка запросы с номерами 4, 6 и 8.

Вариант № 5

Массив данных – «Абоненты АТС».

1. Фамилия (char[]);
2. оплата в месяц (longint);
3. номер АТС (int).
Выполнить из общего списка запросы с номерами 2, 6 и 7.

Пример ТИТУЛЬНОГО ЛИСТА на рисунке К1



МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ

УЧРЕЖДЕНИЕ ОБРАЗОВАНИЯ
«БРЕСТСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»

КАФЕДРА ИНТЕЛЛЕКТУАЛЬНЫХ ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ

ТЕМА КУРСОВОЙ

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА К КУРСОВОЙ РАБОТЕ
ПО ДИСЦИПЛИНЕ «ОСНОВЫ АЛГОРИТМИЗАЦИИ И ПРОГРАММИРОВАНИЯ»

КР.ИИ2.02702 – 12 81 00

Листов 15

Руководитель С.В. Артеменко

Выполнил И.И. Иванов

Консультант
по ЕСПД С.В. Артеменко

2018

Рисунок К1 – Пример титульного листа

Пример ЛИСТА ЗАДАНИЯ на рисунке К2

Учреждение образования
«Брестский государственный технический университет»

Факультет *ЭИС*

«Утверждаю»

Зав. кафедрой ____________

(подпись)

«__» ____________ 2018 год.

З А Д А Н И Е
по курсовой работе

Студенту ______________________________

1. Тема работы ______________________________

2. Сроки сдачи студентом законченной работы *20.05.2018*
3. Исходные данные к работе ______________________________

1. Среда программирования на языке Си/Си++
2. Алгоритмы поиска и сортировки данных в массивах.

4. Содержание расчетно-пояснительной записки (перечень подлежащих разработке вопросов) ______________________________

Введение
1. Постановка задачи
2. Разработка алгоритмов
3. Разработка программы
4. Тестирование
Заключение
Список использованных источников
Приложение А Текст программы
Приложение Б Графический материал

Рисунок К2 – Пример бланка задания

5. Перечень графического материала (с точным указанием обязательных чертежей и графиков) ______________________________

6. Консультанты по работе (с указанием разделов работы) ______________

7. Дата выдачи задания _06.02.2018_
8. Календарный график работы над курсовой работой на весь период проектирования (с указанием сроков выполнения и трудоемкости отдельных этапов)

1. Раздел 1 (10%) –07.02.18 – 02.03.18
2. Раздел 2 (10%) – 03.03.18 – 15.03.18
3. Раздел 3 (30%) – 17.03.18 – 15.04.18
4. Раздел 4 (25%) – 17.04.18 – 02.05.18
4. Раздел 5 (15%) – 03.05.18 – 12.05.18
6. Оформление пояснительной записки (10%) – 13.05.18 – 19.05.18

Руководитель ______________________
(подпись)

Задание принял к исполнению _07.02.2018_ ______________________
(дата и подпись студента)

Рисунок К3 – Пример бланка задания

Пример Листа СОДЕРЖАНИЕ на рисунке К4

ВВЕДЕНИЕ 5

1 ОБЗОР ПРИМЕНЕНИЯ МОБИЛЬНЫХ РОБОТОВ 10

1.1 Сферы применения роботов 10

1.2 Основные предпосылки применения роботов 15

1.3 Социальные аспекты применения промышленных роботов 17

1.4 Поколения роботов 21

1.5 Автоматически управляемое транспортное средство 23

1.6 Постановка задачи 29

2 УСТРОЙСТВО РОБОТА "MAX" 31

2.1 Механическая и электрическая конструкции 31

2.2 Движение робота 33

2.3 Кинематическая модель 35

2.4 Контроллер мотора 41

2.5 Алгоритм контроллера мотора 46

2.6 Характеристики веб-камеры 47

2.7 Математическое описание задачи 48

3 ДЕТЕКТИРОВАНИЕ ЛИНИИ 51

3.1 Взаимодействие с веб-камерой 51

3.2 Детектирование точек 53

3.3 Аппроксимация детектированной линии прямой 58

3.4 Аппроксимация детектированной линии дугой 61

4 АЛГОРИТМЫ УПРАВЛЕНИЯ 68

5 ТЕСТИРОВАНИЕ СИСТЕМЫ 74

5.1 Разработка тестовой трассы для движения робота 74

5.2 Тестирование и оценка результатов 75

ЗАКЛЮЧЕНИЕ 87

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ 88

ПРИЛОЖЕНИЕ А ТЕКСТ ПРОГРАММЫ

					КР.АС1.02702 – 01 81 00			
Изм	Лист	Докум №	Подп.	Дата				
Разраб.		Иванов И.И.			Тема работы. Пояснительная записка	Лит	Лист	Листов
Проверил		Артемчико С.В.				К	3	15
						БрГТУ		
Н. контр.		Артемчико С.В.						
Утв.								

Рисунок К4– Пример листа СОДЕРЖАНИЕ

3. РАЗДЕЛ КОНТРОЛЯ ЗНАНИЙ

3.1.Перечень вопросов, выносимых на экзамен за 1 семестр по дисциплине «Основы алгоритмизации и программирования»

1. Алгоритм. Классы алгоритмов. Способы записи алгоритмов. Свойства алгоритма.
2. Архитектура Фон – Неймана.
3. Базовые конструкции структурированного программирования. Цель использования базовых конструкций.
4. Базовые средства языка С++. Состав языка. Основные элементы алгоритмического языка.
5. Ввод\вывод строки.
6. Двумерный динамический массив. Выделение/высвобождение памяти(С++).
7. Двумерный массив. Инициализация многомерного массива.
9. Доступ к элементам массива. Действия над элементами массива.
10. Идентификаторы. Ключевые слова. Константы.
11. Классификация языков программирования.
12. Массивы. Объявления. Инициализация.
13. Массивы. Объявления. Работа с указателями.
14. Одномерный массив. N-мерный массив.
15. Операторы цикла. Цель использования операторов цикла. Итерация. Параметры цикла.
16. Операторы. Оператор перехода.
17. Операторы. Оператор условия.
18. Операторы. Операторы безусловного перехода.
19. Операторы. Синтаксис. Семантика.
20. Операции сдвига. Операции отношения.
21. Операции увеличения и уменьшения на 1. Операция определения размера. Операции отрицания. Операция деления и остаток от деления.
22. Операции. Поразрядные операции. Логические операции.
23. Операции. Приоритеты операций. Операция определения размера.
24. Операции. Унарные. Бинарные и тернарные операции.
25. Основные типы данных. Спецификаторы типов.
26. Переменные и выражения.
27. Поразрядные операции. Логические операции. Операции присваивания.
28. Правила применения символов и выполнения схем. Описание символов

(символы данных, символы процесса, символы линий, специальные символы).
29. Правило общих арифметических преобразований. Приоритеты операций.
30. Преобразование типов. Преобразование плавающих типов к целым.
31. Приведение типов. Преобразование целых типов со знаком. Преобразование целого типа без знака.
32. Принципы работы компьютера.
33. Приоритет в выражениях. Приоритет операций.
34. Рекомендации при программировании циклических структур.
36. Средства программирования. Этапы получения *.exe модуля.
37. Ссылки. Основные свойства. Пример.
38. Строка. Объявление. Инициализация. Действия над строками и элементами строки.
41. Структура программы на языке С++.
42. Структурное программирование. Основные принципы структурного программирования. Стадии разработки проекта. Критерии качества программ.
43. Схема обработки многомерного динамического массива.
44. Типы с плавающей (фиксированной) точкой. Внутреннее представление.
45. Указатели и массивы. Одномерный динамический массив. Выделение/высвобождение памяти.
46. Указатели. Инициализация указателей.
47. Указатели. Операции над указателями.
48. Указатели. Типы указателей.
51. Цели структурного программирования. Принципы структурного программирования. Требования к структурным программам.
52. Целый тип. Внутреннее представление.
53. Цикл с параметром (For). Инициализации. Модификации. Выражения. Пример.
54. Цикл с постусловием. Пример.
55. Цикл с предусловием. Пример.
57. Этапы решения задач на ЭВМ.
58. Оператор cin и cout. Манипуляторы форматирования.
59. Оператор cin и cout. Форматирующие функции-члены.
60. Оператор cin и cout. Флаги и манипуляторы.
61. Библиотека iostream;
62. Пространство имен (using namespace std;).
63. Динамическое выделение памяти для одномерного массива С++.
64. Динамическое выделение памяти для двумерного массив С++.
65. Динамическое выделение памяти для динамической строки С++.
66. Динамическое выделение памяти для массива строк С++.

67. Динамическое выделение памяти для массива строк с переменной длиной С++.
68. Высвобождение динамической памяти, выделенной под одномерный массив.
69. Высвобождение динамической памяти, выделенной под двумерный массив.
70. Высвобождение динамической памяти, выделенной под n –мерный массив.
71. Высвобождение динамической памяти, выделенной под динамическую строку.
72. Динамическое перераспределение памяти после удаления N-элементов массива С++.
73. Динамическое перераспределение памяти после удаления n-символов строки С++.
74. Заполнение массива С++ случайными значениями.
75. Заполнение массива С++ случайными значениями с заданными диапазоном. Формула.
76. Работа со строками С++. Библиотека. Функции.
77. Строки. Ввод/вывод строк в С++.
78. Таблицы кодировок. Пример применения кодов управляющих символов при разработке программ. Таблицы кодировок. Пример преобразования символов (англ./русск.) в разные регистры без использования функций для работы со строками.
79. Таблицы кодировок. Пример применения таблиц кодировок при разработке программ с использованием алфавита (англ./русск.). Анализ диапазонов символов алфавитов (англ./ русск.). Особенность расположения русского алфавита.
81. Потоки. Высвобождение потока. Функции для С++.
82. Прямой код. Обратный код. Дополнительный код.
83. Стандарты языка С. Стандарты языка С++. Основные моменты.
84. Сортировка пузырьком.
85. Сортировка вставками.
86. Сортировка выбором.
87. Сортировка Шелла.
88.Сортировка Шейкерная.
89. Линейный поиск.
90. Бинарный поиск.

3.2.Перечень вопросов, выносимых на экзамен за 2 семестр по дисциплине «Основы алгоритмизации и программирования»

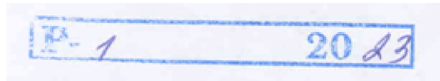
1. Указатели. Инициализация указателей. Операции с указателями.
2. Указатели в С++. Операция разыменования. Константные указатели и указатели на константы. Ссылки в С++.
3. Массивы. Динамические массивы. Динамические многомерные массивы.
4. Псевдослучайные числа. Генерация псевдослучайных чисел на С++.
5. Строки. Объявление и инициализация.
6. Функции для работы со строками и символами.
7. Перечисления. Инициализация. Доступ к полям.
8. Структуры. Инициализация. Доступ к полям.
9. Структуры. Массивы структур.
10. Битовые поля. Объединения. Инициализация. Доступ к полям
11. Модульное программирование.
12. Определение и вызов функций. Фактические и формальные параметры. Передача параметров в функции по значению, по ссылке, по указателю
13. Функции. Возвращаемые значения. Глобальные и локальные данные.
14. Подпрограммы. Передача строк и структур в качестве параметров.
15. Подпрограммы. Передача массивов в качестве параметров. Способы.
16. Подпрограммы. Передача указателя на функцию в качестве параметра функции.
17. Подпрограммы. Указатель на функцию в качестве возвращаемого значения.
18. Функции. Параметры со значением по умолчанию. Способы передачи параметров
19. .Функции. С переменным числом параметров.
20. Функции. Указатели на функции. Массив указателей на функции.
21. Параметры функции main()
22. Функции. Рекурсивные функции. Распределение памяти при рекурсивном вызове. Типы рекурсии. Рекомендации по использованию рекурсивного вызова.
23. Функции. Распределение памяти при вызове функции.
24. Функции. Перегрузка функций.
25. Потоки и файлы. Стандартные потоки. Обработки исключительных ситуаций.
26. Ввод вывод в поток. Закрытие потока.
27. Динамические структуры данных. Рекомендации для использования. Основные особенности.
28. Стек. Создание стека. Общие принципы работы
29. Стек. Удаление элемента. Добавление элемента. Общие принципы работы
30. Очередь. Создание очереди.
31. Очередь. Удаление элемента. Добавление элемента. Общие принципы работы
32. Линейные списки. Создание элемента списка.

33. Линейные списки. Добавление узла в начало списка.
34. Линейные списки. Добавление узла после заданного.
35. Линейные списки. Добавление узла перед заданным.
36. Линейные списки. Добавление узла в конец списка.
37. Линейные списки. Проход по списку.
38. Линейные списки. Удаление узла.
39. Двусвязный список. Создание. Удаление узла.
40. Двусвязный список. Создание. Добавление узла.
41. Деревья. Общие понятия.
42. Деревья. Формирование бинарного дерева
43. Бинарные деревья. Обход дерева в ширину.
44. Бинарные деревья. Обход дерева в глубину.
45. Бинарные деревья. Обход дерева в глубину – прямой (префиксный, pre-ordered)
46. Бинарные деревья. Обход дерева в глубину – обратный (инфиксный, in-ordered)
47. Бинарные деревья. Обход дерева в глубину – концевой (постфиксный, post-ordered)
48. Бинарные деревья. Поиск в дереве.
49. Файлы. Типы файлов.
50. Доступ к файлам.
51. Файлы. Текстовые файлы. Открытие \ закрытие. Диагностика ошибок
52. Текстовые файлы. Ввод \ вывод в текстовые файлы.
53. Файлы. Бинарные файлы.
54. Бинарные файлы. Функции чтения \ записи .
55. Бинарные файлы. Открытие \ закрытие файла. Диагностика ошибок.
56. Бинарные файлы. Позиционирование в файле.
57. Потоки и файлы. Стандартные потоки. Обработки исключительных ситуаций.
58. Стандартные потоки ввода/вывода. Средства работы с потоками ввода/вывода. Специальные символы (символ перевода строки, символ табуляции, символ конца строки).
59. Ввод вывод в поток. Закрытие потока.
60. Поиск. Линейный.
61. Поиск. Двоичный.
62. Сортировка обменом.
63. Сортировка вставкой.
64. Сортировка выбором.
65. Сортировка Шелла
66. Сортировка Шейкерная.
67. Виды программ и программных документов (ГОСТ 19.101-77).
68. Стадии разработки (ГОСТ 19.102-77).

69.Схемы алгоритмов, программ, данных и систем (ГОСТ 19.701-90).

4 ВСПОМОГАТЕЛЬНЫЙ РАЗДЕЛ

1.Учебная программа учреждения высшего образования по учебной дисциплине для специальностей 6-05-0612-03 «Системы управления информацией», 6-05-0611-03 «Искусственный интеллект», 6-05-0612-01«Программная инженерия»



Учреждение образования
«Брестский государственный технический университет»

УТВЕРЖДАЮ
Первый проректор
Первый проректор М.В.Нерода
23.06 2023г.

Регистрационный № УД-
23-1-209 /уч.

Основы алгоритмизации и программирования

Учебная программа учреждения высшего образования по учебной дисциплине для специальностей

6-05-0612-03 «Системы управления информацией»

6-05-0611-03 «Искусственный интеллект»

6-05-0612-01 «Программная инженерия»

2023

Учебная программа составлена на основе учебных планов специальностей 6-05-0612-03 Системы управления информацией, 6-05-0611-03 Искусственный интеллект, 6-05-0612-01 Программная инженерия

СОСТАВИТЕЛЬ:
Хацкевич М.В., старший преподаватель кафедры интеллектуальных информационных технологий Учреждения образования «Брестский государственный технический университет»

РЕЦЕНЗЕНТЫ:
Грицук Д.В., заведующий кафедрой прикладной математики и информатики БрГУ имени А.С. Пушкина, кандидат физико-математический наук, доцент
Махнист Л.П., заведующий кафедрой «Высшая математика» БрГТУ, кандидат технических наук, доцент

РЕКОМЕНДОВАНА К УТВЕРЖДЕНИЮ:
Кафедрой интеллектуальных информационных технологий
Заведующий кафедрой В.А.Головко
(протокол №___10___ от 13.06.2023);

Методической комиссией факультета
Председатель методической комиссии
Председатель методической комиссии С.С.Дереченник
(протокол № 11 от 20.06. 2023);

Научно-методическим советом БрГТУ (протокол № 6 от 23.06.2023)
Методист УМО А.И. Седун

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

Дисциплина «Основы алгоритмизации и программирования» относится к модулям «Программирование» (Системы управления информацией), «Основы алгоритмизации и программирования» (Искусственный интеллект), «Фундаментальные основы разработки программного обеспечения» (Программная инженерия) государственного компонента.

Цель преподавания учебной дисциплины: подготовка специалиста, уверенно владеющего возможностями, предоставляемыми современными компьютерными технологиями в среде программирования на алгоритмическом языке высокого уровня, а также программирования вычислительных алгоритмов. освоение базовых методов структурного и функционально-ориентированного программирования.

Задачи учебной дисциплины:

- изучение языка программирования высокого уровня, а также приобретение практических навыков составления и отладки программ на персональных компьютерах;
- приобретение навыков алгоритмизации на примерах решения вычислительных задач и их закрепление на основе программирования алгоритмов обработки структур данных и алгоритмов вычислительной математики;
- приобретение знаний об эффективности разрабатываемых алгоритмов, оценке их временных и вычислительных ресурсов.

В результате изучения дисциплины «Основы алгоритмизации и программирования» формируются следующие компетенции:

а) универсальные:

- владеть основами исследовательской деятельности, осуществлять поиск, анализ и синтез информации;
- решать стандартные задачи профессиональной деятельности на основе применения информационно-коммуникационных технологий;
- обладать навыками саморазвития и совершенствования в профессиональной деятельности;
- проявлять инициативу и адаптироваться к изменениям в профессиональной деятельности.

б) базовые профессиональные:

- применять основные методы алгоритмизации, способы и средства получения, хранения и обработки информации при решении профессиональных задач;

– применять базовые аспекты различных парадигм программирования и практические навыки их использования на всех этапах разработки в современных интегрированных инструментальных средах.

В результате изучения учебной дисциплины студент должен:
знать:
– основы и современное состояние алгоритмического языка высокого уровня (C++);

– способы построения и представления алгоритмов;

– основные динамические структуры данных и алгоритмы их обработки;

– вычислительные алгоритмы решения инженерных задач;

– теоретические основы алгоритмизации и проектирования программ;

– принципы оценки вычислительной сложности и эффективности алгоритмов.

уметь:
– выполнять алгоритмизацию инженерных задач;

– реализовывать разработанный алгоритм в виде собственной программы на алгоритмическом языке или с использованием стандартных программ;

– применять разработанные программы в профессиональной деятельности.

владеть:
– современными средствами программирования;

– навыками анализа исходных и выходных данных решаемых задач и формами их представления;

– навыками отладки программ.

Базовыми дисциплинами по курсу «Основы алгоритмизации и программирования» являются: «Математика» (в объеме уровня общего среднего образования), «Информатика» (в объеме уровня общего среднего образования).

План учебной дисциплины для дневной формы получения высшего образования

Код специальности (направления специальности)	Наименование специальности (направления специальности)	Курс	Семестр	Всего учебных часов	Количество зачетных единиц	Аудиторных часов (в соответствии с учебным планом УВО)					Академических часов на курсовой проект (работу)	Форма текущей аттестации
						Всего	Лекции	Лабораторные занятия	Практические занятия	Семинары		
6-05-0612-03	Системы управления информацией	1	1	108	3,0	60	28	32				Экзамен
			2	108	3,0	60	28	32			30	Экзамен
6-05-0611-03	Искусственный интеллект	1	1	108	3,0	60	28	32				Экзамен
			2	108	3,0	60	28	32			30	Экзамен
6-05-0612-01	Программная инженерия	1	1	108	3,0	64	32	32				Экзамен
			2	108	3,0	56	24	32			30	Экзамен

План учебной дисциплины для заочной формы получения высшего образования

Код специальности (направления специальности)	Наименование специальности (направления специальности)	Курс	Семестр	Всего учебных часов	Количество зачетных единиц	Аудиторных часов (в соответствии с учебным планом УВО)					Академических часов на курсовой проект (работу)	Форма текущей аттестации
						Всего	Лекции	Лабораторные занятия	Практические занятия	Семинары		
6-05-0612-03	Системы управления информацией	1	1	108	3,0	14	6	8				Экзамен
			2	108	3,0	14	6	8			30	Экзамен

1. СОДЕРЖАНИЕ УЧЕБНОГО МАТЕРИАЛА

1.1. ЛЕКЦИОННЫЕ ЗАНЯТИЯ, ИХ СОДЕРЖАНИЕ

Семестр 1

РАЗДЕЛ 1. ВВЕДЕНИЕ. ОСНОВНЫЕ ПОНЯТИЯ

Тема 1.1. Общие сведения об алгоритмах, ЭВМ и организации вычислительного процесса

Программное и аппаратное обеспечение. Языки программирования: уровень и тип языка программирования, характеристики. Краткий обзор парадигм программирования: процедурные языки, объектно-ориентированные языки. Этапы разработки программного обеспечения. Жизненный цикл программного продукта. Характеристики разрабатываемой программы. Основные принципы обработки команд программы исполнителем (компьютером). Организация ЭВМ. Принципы построения ЭВМ, машина Фон Неймана.

Тема 1.2. Способы представления данных в ЭВМ

Понятия данных, информации. Свойства информации. Представление данных разного типа в компьютере: целочисленные данные и числа с плавающей точкой, строки фиксированной и переменной длины, символы, логические значения, даты. Различные варианты кодировки символов. Сравнение данных разных типов. Системы счисления. Общие понятия и конкретные системы: десятичная, двоичная, шестнадцатеричная, восьмеричная. Правила перевода числа из одной системы в другую. Устройство памяти. Адресация. Понятие переменная. Объявление (декларация) и инициализация переменных. Правила именования.

Тема 1.3. Общие сведения о программном обеспечении и технологии производства программного обеспечения

Термины и определения (ГОСТ 19.001-77). Виды программного обеспечения и программной документации (ГОСТ 19.701-90, ИСО 5807-85). Понятие программа, алгоритм, исполнитель. Типы программного обеспечения: системное, прикладное, инструментальное (средства разработчика). Трехуровневая модель программного продукта. Свойства алгоритмов. Формы представления алгоритмов: естественный язык, блок-схема, формальный язык. Составление блок-схем алгоритмов. Иерархическая организация программы, модульность. Область видимости и время жизни. Секции программного модуля. Понятия транслятор, компилятор, интерпретатор. Статическая и динамическая компиляция.

Тема 1.4. Интерфейс. Общие сведения

Варианты интерфейсов. Важность правильной разработки интерфейса. Различные методы построения диалога с пользователем.

РАЗДЕЛ 2. ОСНОВЫ ПРОГРАММИРОВАНИЯ НА АЛГОРИТМИЧЕСКОМ ЯЗЫКЕ С++

Тема 2.1. Среда программирования

Изучение системы программирования Visual Studio. Состав компонентов, библиотеки, заголовочные файлы. Интегрированная среда. Использование справочной системы.

Тема 2.2. Основные понятия. Типы данных

Состав и структура языка программирования. Понятия алфавита, синтаксиса и семантики. Основные конструкции языка: символы, зарезервированные и пользовательские имена, комментарии. Типы данных. Характеристика, описание и использование скалярных типов. Переменные и константы. Определение имени переменной. Объявление переменной. Стандартные операции с переменными. Типы переменных. Преобразование типов. Автоматическое и управляемое преобразование типов. Обращение к стандартной библиотеке ввода\вывода. Функции стандартной библиотеки, заголовочный файлы. Форматный ввод\вывод, форматные коды, управляющие символы.

Тема 2.3. Операции, операторы и выражения

Операции языка C++. Унарные операции. Мультипликативные операции. Аддитивные операции. Операции сдвига. Операции сдвига. Операции отношения. Операции равенства. Побитовая операция «И». Побитовая операция исключающего «ИЛИ». Логическая операция «И». Операция логического «ИЛИ». Условная операция. Операция присваивания. Операция запятая. Операторы. Понятие оператора. Запись операторов. Классификация, приоритеты и особенности выполнения операторов. Операторное выражение. Составной оператор. Условные операторы. Составления условия: сравнение числовых значений, дат, строковых и логических значения. Составление сложных условий: использование логических операций OR, AND, XOR, NOT. Приоритет операций. Вложенные операторы. Оптимизация условий. Вложенные операторы. Оператор выбора. Оптимизация оператора выбора. Оператор цикла: циклы с предусловием, с постусловием, с параметром. Цикл для обхода элементов группы. Понятия: тело цикла, условие цикла, счетчик, итерация. Использование счетчика цикла. Операторы безусловных переходов. Пустой оператор.

Тема 2.4. Структурированные типы данных

Роль организации данных в программе. Понятие массива. Массивы одномерные и многомерные. Понятие индекса и элемента массива. Подсчет объема памяти, занимаемой массивом. Типовые задачи с массивами: доступ к элементу, обход элементов, инициализация элементов. Представление текстовой информации. Строки: особенности описания, инициализации, ввода\вывода, обработки, функции для обработки строк и символов. Строки и функции. Массивы строк. Составной тип данных (структура, объединения, поля бит). Массивы составных типов. Описание, инициализация и использование составного типа. Совместное использование составных типов данных и массивов. Указатели на составные типы.

Тема 2.5. Процедуры и функции

Декомпозиция задачи на подзадачи. Программирование сверху вниз и снизу вверх. Парадигма черного ящика. Входные и выходные данные подпрограммы. Понятия подпрограмма, процедура, функция. Процедуры стандартные, пользовательские. Описание, объявление, вызов процедуры.

Библиотеки функций. Вызов процедуры и функции. Прототипы. Аргументы формальные и фактические. Тип аргумента. Передача аргументов по значению и по ссылке. Значение, возвращаемой функцией. Передача параметров через командную строку. Массивы и функции, способы передачи в качестве аргументов и результатов. Указатели на функцию. Составные типы и функции. Способы взаимодействия различных фрагментов программы. Структурная декомпозиция.

Тема 2.6. Дополнительные возможности

Файлы, логические устройства, потоки. Общие сведения, классификация, уровни и средства доступа, режимы работы. Функции стандартной библиотеки ввода\вывода. Текстовые файлы. Последовательный доступ. Бинарные файлы. Блочный ввод\вывод. Позиционирование. Произвольный доступ.

Семестр 2

РАЗДЕЛ 3. СТАНДАРТНЫЕ АЛГОРИТМЫ

Тема 3.1. Сортировка и поиск

Поиск элемента в массиве: линейный, двоичный и интерполяционные алгоритмы. Поиск наибольшего и наименьшего элемента в массиве. Различные способы сортировки элементов массива: метод прямого выбора, метод вставки, пузырьковая сортировка, сортировка Шелла, шейкерная сортировка. Оценка сложности алгоритма и сравнение алгоритмов. Алгоритмы работы со строками. Разбиение и объединение строк, поиск и извлечение подстроки, удаление подстроки, синтаксический анализ текста. Рекурсивные и итерационные алгоритмы. Рекурсивные математические функции на примере вычисления факториала.

РАЗДЕЛ 4. АДРЕСНАЯ АРИФМЕТИКА. ДИНАМИЧЕСКАЯ ПАМЯТЬ. ДИНАМИЧЕСКИЕ СТРУКТУРЫ

Тема 4.1. Указатели и адреса

Определение и инициализация указателя. Тип указателя, значение указателя. Понятие нулевого адреса NULL. Операция разыменования указателя. Указатели и операции адресной арифметики. Понятие родового указателя *void. Многоуровневый указатель. Ассоциативность операции разыменования многоуровневого указателя. Организация памяти. Сегментация и адресация памяти. Модели памяти. Массивы и указатели. Имя массива как константный указатель. Общность операций индексирования и адресной арифметики. Многомерный массив и указатель. Доступ к элементам многомерного массива через указатель. Представление многомерного массива с помощью массива указателей. Динамические переменные и массивы. Роль указателя в обработке динамических переменных и массивов. Динамическая область программы и модели памяти. Стандартные функции управления динамической памятью и модели памяти.

Тема 4.2. Указатели и функции

Адресный способ передачи параметров и результатов функции. Указатели, как формальные параметры функции. Массив, как формальный параметр или результат функции. Способы представления многомерного

массива, как формального параметра функции. Функции с переменным количеством параметров. Доступ к списку параметров через указатель. Указатели на функции. Имя функции, как константный указатель на функцию. Определение и инициализация указателя на функцию. Вызов функции через указатель. Указатель на функцию, как формальный параметр функции.

Тема 4.3. Структуры данных

Классификация структур данных. Статические и динамические структуры. Одноранговые и иерархические структуры. Типизированные и универсальные структуры. Линейные и нелинейные структуры. Структуры с последовательным или прямым доступом: стек, очередь, дек, ассоциативное множество. Комбинированные структуры.

Тема 4.4. Массивы указателей и структуры

Одноранговые ограниченные и неограниченные структуры на базе статического и динамического массива указателей. Тип массива указателей и типизированные структуры. Массив указателей типа void* и универсальные структуры. Стек, очередь, циклическая очередь, дек на базе массива указателей. Программирование основных операций: включение, извлечение. Ассоциативное множество с доступом по ключу на базе массива указателей. Программирование основных операций: включение, извлечение, поиск, итераторы доступа. Ассоциативные рассеянные таблицы на базе массива указателей. Метод преобразования ключа (хэширование). Функция расстановки по ключу. Разрешение конфликтов ключей: линейное и квадратичное апробирование. Программирование основных операций: включение, извлечение, поиск, итераторы доступа. Ассоциативные иерархические структуры на базе массива многоуровневых указателей. Статические и динамические структуры. Типизированные и универсальные иерархические структуры. Программирование основных операций: включение, извлечение, поиск, итераторы доступа.

Тема 4.5. Динамические линейные структуры

Одноранговые динамические структуры на базе линейных списков со связями. Виды линейного списка: односвязный, двусвязный, с барьером, циклический. Типизированные и универсальные списки. Терминология: элемент списка, заголовок списка. Определение заголовка и элемента списка на С++. Стек, очередь на базе линейного списка. Программирование основных операций: включение, извлечение. Ассоциативное множество с доступом по ключу на базе линейного списка. Программирование основных операций: включение, извлечение, поиск, итераторы доступа. Иерархические динамические структуры на базе линейных списков. Разреженный многомерный массив на базе иерархической структуры списков. Программирование основных операций: индексирование, включение и извлечение по индексу, итераторы доступа. Комбинированные иерархические структуры на базе линейных списков и массивов указателей. Ассоциативные рассеянные таблицы со списками конфликтов ключей.

Программирование основных операций: включение, извлечение, поиск, итераторы доступа.

Тема 4.6. Динамические нелинейные структуры

Ассоциативные множества с доступом по ключу на базе деревьев поиска. Терминология: узел, корень, лист, поддерево, уровень, путь, степень узла, степень дерева. Рекурсивная природа дерева. Обход дерева, схемы обхода. Бинарное дерево. Внутренний поиск. Упорядоченное дерево. Сбалансированное дерево. Оптимальное дерево поиска. Представление бинарного дерева на C++ в виде массива указателей. Представление бинарного дерева на C++ в виде динамической структуры со связями, определение узла дерева. Программирование основных операций: включение, извлечение, поиск с включением, поиск с извлечением, итераторы обхода. Определение узла сбалансированного бинарного дерева на C++. Программирование операций поиска с включением и поиска с извлечением.

1.2. ПЕРЕЧЕНЬ ТЕМ ЛАБОРАТОРНЫХ ЗАНЯТИЙ, ИХ НАЗВАНИЕ

Лабораторная работа №1 «Разработка схем алгоритмов для линейных и разветвляющихся процессов в соответствии с положениями действующих стандартов. Разработка схем алгоритмов для циклических процессов в соответствии с положениями действующих стандартов. Разработка структурированных схем алгоритмов»

Лабораторная работа №2 «Знакомство со средой программирования»

Лабораторная работа №3 «Разработка, отладка и выполнение простейшей программы»

Лабораторная работа №4 «Разработка алгоритма, составление, отладка и выполнение программы с ветвлением (выбором вариантов)»

Лабораторная работа №5 «Разработка алгоритма, составление, отладка и выполнение циклической программы с известным числом повторений. Разработка алгоритма, составление, отладка и выполнение программы с использованием итерационных циклов. Разработка и выполнение программы с использованием разветвлений и вложенных циклов»

Лабораторная работа №6 «Разработка алгоритма, составление, отладка и выполнение программы обработки сложных типов данных (массивы)»

Лабораторная работа №7 «Разработка алгоритма, составление, отладка и выполнение программы обработки сложных типов данных (строки)»

Лабораторная работа №8 «Разработка алгоритма, составление, отладка и выполнение программы с использованием пользовательских функций. Разработка алгоритма, составление, отладка и выполнение программы с использованием рекурсивных функций. Разработка алгоритма, составление, отладка и выполнение программы с использованием функций с произвольным числом параметров»

Лабораторная работа №9 «Разработка, отладка и выполнение программы с использованием модулей пользователя»

Лабораторная работа №10 «Разработка, отладка и выполнение программы обработки текстовых файлов»

Лабораторная работа №11 «Разработка, отладка и выполнение программы обработки файлов с типом»

Лабораторная работа №12 «Разработка алгоритмов, составление, отладка и выполнение программы сортировки и поиска (массивы, строки)»

Лабораторная работа №13 «Разработка, отладка и выполнение программы с использованием подпрограмм с различными типами параметров»

Лабораторная работа №14 «Разработка алгоритма, составление, отладка и выполнение программы с использованием структур (массивов структур)»

Лабораторная работа №15 «Разработка алгоритма, составление, отладка и выполнение программы обработки линейных связанных списков»

Лабораторная работа №16 «Программирование с использованием древовидных структур данных»

2. ТРЕБОВАНИЯ К КУРСОВОЙ РАБОТЕ

Курсовая работа выполняется во втором семестре и на нее, в соответствии с планами, отведено 30 часов.

Целью курсовой работы является закрепление и углубление теоретических знаний, формирование практических умений и навыков по следующим направлениям:

– составление и отладка программ на персональных компьютерах;

– алгоритмизация на примерах решения вычислительных задач;

– эффективность разрабатываемых алгоритмов, оценка их временных и вычислительных ресурсов.

Примерный перечень тем курсовой работы:

1. Обработка множества структурированных данных заданной тематики.
2. Написание консольной мини-игры с применением методов структурного программирования.
3. Написание консольного приложения «многофункциональный будильник».
4. Написание консольного приложения «многофункциональный калькулятор».

Примерная структура пояснительной записки по курсовой работе включает следующие разделы:

ВВЕДЕНИЕ
1. АНАЛИЗ ПРЕДМЕТНОЙ ОБЛАСТИ
2. ПРОЕКТИРОВАНИЕ СИСТЕМЫ
3. РЕАЛИЗАЦИЯ СИСТЕМЫ
4. ТЕСТИРОВАНИЕ СИСТЕМЫ

ЗАКЛЮЧЕНИЕ

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

ПРИЛОЖЕНИЕ А. ТЕКСТ ПРОГРАММЫ

3.1. УЧЕБНО-МЕТОДИЧЕСКАЯ КАРТА УЧЕБНОЙ ДИСЦИПЛИНЫ
для дневной формы получения высшего образования специальностей 6-05-0612-03 Системы управления информацией, 6-05-0611-03 Искусственный интеллект

Номер раздела, темы	Название раздела, темы	Количество аудиторных часов				Количество часов самост. работы	Форма контроля знаний
		Лекции	Лабораторные занятия	Практические занятия	Семинарские занятия		
1-ый семестр							
1	ВВЕДЕНИЕ. ОСНОВНЫЕ ПОНЯТИЯ	8	4			14	
1.1	Общие сведения об алгоритмах, ЭВМ и организации вычислительного процесса	2				2	
1.2	Способы представления данных в ЭВМ	2				4	
1.3	Общие сведения о программном обеспечении и технологии производства программного обеспечения	2				4	
	Лабораторная работа №1 «Разработка схем алгоритмов для линейных и разветвляющихся процессов в соответствии с положениями действующих стандартов. Разработка схем алгоритмов для циклических процессов в соответствии с положениями действующих стандартов. Разработка структурированных схем алгоритмов»		2				Зашита отчёта
1.4	Интерфейс. Общие сведения	2				4	
	Лабораторная работа №2 «Знакомство со средой программирования»		2				Защита отчёта
2	ОСНОВЫ ПРОГРАММИРОВАНИЯ НА АЛГОРИТМИЧЕСКОМ ЯЗЫКЕ С++	20	28			34	
2.1	Среда программирования	2				4	
	Лабораторная работа №3 «Разработка, отладка и выполнение простейшей программы»		2				Защита отчёта
2.2	Основные понятия. Типы данных	2				6	
	Лабораторная работа №4 «Разработка алгоритма, составление, отладка и выполнение программы с ветвлением (выбором вариантов)»		2				Защита отчёта
2.3	Операции, операторы и выражения	4				6	
	Лабораторная работа №5 «Разработка алгоритма, составление, отладка и выполнение циклической программы с известным числом повторений. Разработка алгоритма, составление, отладка и выполнение программы с использованием итерационных циклов. Разработка и выполнение программы с использованием разветвлений и вложенных циклов»		4				Защита отчёта

2.4	Структурированные типы данных	4				6	
	Лабораторная работа №6 «Разработка алгоритма, составление, отладка и выполнение программы обработки сложных типов данных (массивы)»		2				Защита отчёта
	Лабораторная работа №7 «Разработка алгоритма, составление, отладка и выполнение программы обработки сложных типов данных (строки)»		2				Защита отчёта
2.5	Процедуры и функции	4				6	
	Лабораторная работа №8 «Разработка алгоритма, составление, отладка и выполнение программы с использованием пользовательских функций. Разработка алгоритма, составление, отладка и выполнение программы с использованием рекурсивных функций. Разработка алгоритма, составление, отладка и выполнение программы с использованием функций с произвольным числом параметров»		4				Защита отчёта
	Лабораторная работа №9 «Разработка, отладка и выполнение программы с использованием модулей пользователя»		4				Защита отчёта
2.6	Дополнительные возможности	4				6	
	Лабораторная работа №10 «Разработка, отладка и выполнение программы обработки текстовых файлов»		4				Защита отчёта
	Лабораторная работа №11 «Разработка, отладка и выполнение программы обработки файлов с типом»		4				Защита отчёта
Итого за 1-ый семестр:		28	32			48	Экзамен
2-ой семестр							
3	СТАНДАРТНЫЕ АЛГОРИТМЫ	12	4			24	
3.1	Сортировка и поиск	12				24	
	Лабораторная работа №12 «Разработка алгоритмов, составление, отладка и выполнение программы сортировки и поиска (массивы, строки)»		4				Защита отчёта
4	АДРЕСНАЯ АРИФМЕТИКА. ДИНАМИЧЕСКАЯ ПАМЯТЬ. ДИНАМИЧЕСКИЕ СТРУКТУРЫ	16	28			24	
4.1	Указатели и адреса	2				4	
4.2	Указатели и функции	2				4	
	Лабораторная работа №13 «Разработка, отладка и выполнение программы с использованием подпрограмм с различными типами параметров»		4				Защита отчёта
4.3	Структуры данных	2				4	
	Лабораторная работа №14 «Разработка алгоритма, составление, отладка и выполнение программы с использованием структур (массивов структур)»		8				Защита отчёта
4.4	Массивы указателей и структуры	2				4	
4.5	Динамические линейные структуры	4				4	

	Лабораторная работа №15 «Разработка алгоритма, составление, отладка и выполнение программы обработки линейных связанных списков»		8				Защита отчёта
4.6	Динамические нелинейные структуры	4				4	
	Лабораторная работа №16 «Программирование с использованием древовидных структур данных»		8				Защита отчёта
Итого за 2-ой семестр:		28	32			48	Экзамен

3.2. УЧЕБНО-МЕТОДИЧЕСКАЯ КАРТА УЧЕБНОЙ ДИСЦИПЛИНЫ
для дневной формы получения высшего образования специальности 6-05-0612-01 Программная инженерия

Номер раздела, темы	Название раздела, темы	Количество аудиторных часов				Количество часов самост. работы	Форма контроля знаний
		Лекции	Лабораторные занятия	Практические занятия	Семинарские занятия		
1-ый семестр							
1	ВВЕДЕНИЕ. ОСНОВНЫЕ ПОНЯТИЯ	8	4			14	
1.1	Общие сведения об алгоритмах, ЭВМ и организации вычислительного процесса	2				2	
1.2	Способы представления данных в ЭВМ	2				4	
1.3	Общие сведения о программном обеспечении и технологии производства программного обеспечения	2				4	
	Лабораторная работа №1 «Разработка схем алгоритмов для линейных и разветвляющихся процессов в соответствии с положениями действующих стандартов. Разработка схем алгоритмов для циклических процессов в соответствии с положениями действующих стандартов. Разработка структурированных схем алгоритмов»		2				Зашита отчёта
1.4	Интерфейс. Общие сведения	2				4	
	Лабораторная работа №2 «Знакомство со средой программирования»		2				Защита отчёта
2	ОСНОВЫ ПРОГРАММИРОВАНИЯ НА АЛГОРИТМИЧЕСКОМ ЯЗЫКЕ С++	24	28			30	
2.1	Среда программирования	4				4	
	Лабораторная работа №3 «Разработка, отладка и выполнение простейшей программы»		2				Защита отчёта
2.2	Основные понятия. Типы данных	4				4	
	Лабораторная работа №4 «Разработка алгоритма, составление, отладка и выполнение программы с ветвлением (выбором вариантов)»		2				Защита отчёта

2.3	Операции, операторы и выражения	4				4	
	Лабораторная работа №5 «Разработка алгоритма, составление, отладка и выполнение циклической программы с известным числом повторений. Разработка алгоритма, составление, отладка и выполнение программы с использованием итерационных циклов. Разработка и выполнение программы с использованием разветвлений и вложенных циклов»		4				Защита отчёта
2.4	Структурированные типы данных	4				6	
	Лабораторная работа №6 «Разработка алгоритма, составление, отладка и выполнение программы обработки сложных типов данных (массивы)»		2				Защита отчёта
	Лабораторная работа №7 «Разработка алгоритма, составление, отладка и выполнение программы обработки сложных типов данных (строки)»		2				Защита отчёта
2.5	Процедуры и функции	4				6	
	Лабораторная работа №8 «Разработка алгоритма, составление, отладка и выполнение программы с использованием пользовательских функций. Разработка алгоритма, составление, отладка и выполнение программы с использованием рекурсивных функций. Разработка алгоритма, составление, отладка и выполнение программы с использованием функций с произвольным числом параметров»		4				Защита отчёта
	Лабораторная работа №9 «Разработка, отладка и выполнение программы с использованием модулей пользователя»		4				Защита отчёта
2.6	Дополнительные возможности	4				6	
	Лабораторная работа №10 «Разработка, отладка и выполнение программы обработки текстовых файлов»		4				Защита отчёта
	Лабораторная работа №11 «Разработка, отладка и выполнение программы обработки файлов с типом»		4				Защита отчёта
Итого за 1-ый семестр:		32	32			44	Экзамен
2-ой семестр							
3	СТАНДАРТНЫЕ АЛГОРИТМЫ	8	4			24	
3.1	Сортировка и поиск	8				24	
	Лабораторная работа №12 «Разработка алгоритмов, составление, отладка и выполнение программы сортировки и поиска (массивы, строки)»		4				Защита отчёта
4	АДРЕСНАЯ АРИФМЕТИКА. ДИНАМИЧЕСКАЯ ПАМЯТЬ. ДИНАМИЧЕСКИЕ СТРУКТУРЫ	16	28			28	
4.1	Указатели и адреса	2				4	
4.2	Указатели и функции	2				4	

	Лабораторная работа №13 «Разработка, отладка и выполнение программы с использованием подпрограмм с различными типами параметров»		4				Защита отчёта
4.3	Структуры данных	2				4	
	Лабораторная работа №14 «Разработка алгоритма, составление, отладка и выполнение программы с использованием структур (массивов структур)»		8				Защита отчёта
4.4	Массивы указателей и структуры	2				4	
4.5	Динамические линейные структуры	4				6	
	Лабораторная работа №15 «Разработка алгоритма, составление, отладка и выполнение программы обработки линейных связанных списков»		8				Защита отчёта
4.6	Динамические нелинейные структуры	4				6	
	Лабораторная работа №16 «Программирование с использованием древовидных структур данных»		8				Защита отчёта
Итого за 2-ой семестр:		24	32			52	Экзамен

3.3. УЧЕБНО-МЕТОДИЧЕСКАЯ КАРТА УЧЕБНОЙ ДИСЦИПЛИНЫ для заочной формы получения высшего образования специальности 6-05-0612-03 Системы управления информацией

Номер раздела, темы	Название раздела, темы	Количество аудиторных часов				Количество часов самост. работы	Форма контроля знаний
		Лекции	Лабораторные занятия	Практические занятия	Семинарские занятия		
1-ый семестр							
1	ВВЕДЕНИЕ. ОСНОВНЫЕ ПОНЯТИЯ	2	1			30	
1.1	Общие сведения об алгоритмах, ЭВМ и организации вычислительного процесса	0,5				6	
1.2	Способы представления данных в ЭВМ	0,5				8	
1.3	Общие сведения о программном обеспечении и технологии производства программного обеспечения	0,5				8	
	Лабораторная работа №1 «Разработка схем алгоритмов для линейных и разветвляющихся процессов в соответствии с положениями действующих стандартов. Разработка схем алгоритмов для циклических процессов в соответствии с положениями действующих стандартов. Разработка структурированных схем алгоритмов»		0,5				Зашита отчёта
1.4	Интерфейс. Общие сведения	0,5				8	

	Лабораторная работа №2 «Знакомство со средой программирования»		0,5				Защита отчёта
2	ОСНОВЫ ПРОГРАММИРОВАНИЯ НА АЛГОРИТМИЧЕСКОМ ЯЗЫКЕ С++	4	7			64	
2.1	Среда программирования	0,5				8	
	Лабораторная работа №3 «Разработка, отладка и выполнение простейшей программы»		0,5				Защита отчёта
2.2	Основные понятия. Типы данных	0,5				8	
	Лабораторная работа №4 «Разработка алгоритма, составление, отладка и выполнение программы с ветвлением (выбором вариантов)»		0,5				Защита отчёта
2.3	Операции, операторы и выражения	0,5				8	
	Лабораторная работа №5 «Разработка алгоритма, составление, отладка и выполнение циклической программы с известным числом повторений. Разработка алгоритма, составление, отладка и выполнение программы с использованием итерационных циклов. Разработка и выполнение программы с использованием разветвлений и вложенных циклов»		0,5				Защита отчёта
2.4	Структурированные типы данных	0,5				8	
	Лабораторная работа №6 «Разработка алгоритма, составление, отладка и выполнение программы обработки сложных типов данных (массивы)»		0,5				Защита отчёта
	Лабораторная работа №7 «Разработка алгоритма, составление, отладка и выполнение программы обработки сложных типов данных (строки)»		1				Защита отчёта
2.5	Процедуры и функции	1				16	
	Лабораторная работа №8 «Разработка алгоритма, составление, отладка и выполнение программы с использованием пользовательских функций. Разработка алгоритма, составление, отладка и выполнение программы с использованием рекурсивных функций. Разработка алгоритма, составление, отладка и выполнение программы с использованием функций с произвольным числом параметров»		1				Защита отчёта
	Лабораторная работа №9 «Разработка, отладка и выполнение программы с использованием модулей пользователя»		1				Защита отчёта
2.6	Дополнительные возможности	1				16	
	Лабораторная работа №10 «Разработка, отладка и выполнение программы обработки текстовых файлов»		1				Защита отчёта
	Лабораторная работа №11 «Разработка, отладка и выполнение программы обработки файлов с типом»		1				Защита отчёта
Итого за 1-ый семестр:		6	8			94	Экзамен
2-ой семестр							
3	СТАНДАРТНЫЕ АЛГОРИТМЫ	2				30	

3.1	Сортировка и поиск	2				30	
	Лабораторная работа №12 «Разработка алгоритмов, составление, отладка и выполнение программы сортировки и поиска (массивы, строки)»		1				Защита отчёта
4	АДРЕСНАЯ АРИФМЕТИКА. ДИНАМИЧЕСКАЯ ПАМЯТЬ. ДИНАМИЧЕСКИЕ СТРУКТУРЫ	4				64	
4.1	Указатели и адреса	0,5				8	
4.2	Указатели и функции	0,5				8	
	Лабораторная работа №13 «Разработка, отладка и выполнение программы с использованием подпрограмм с различными типами параметров»		1				Защита отчёта
4.3	Структуры данных	0,5				8	
	Лабораторная работа №14 «Разработка алгоритма, составление, отладка и выполнение программы с использованием структур (массивов структур)»		2				Защита отчёта
4.4	Массивы указателей и структуры	0,5				8	
4.5	Динамические линейные структуры	1				16	
	Лабораторная работа №15 «Разработка алгоритма, составление, отладка и выполнение программы обработки линейных связанных списков»		2				Защита отчёта
4.6	Динамические нелинейные структуры	1				16	
	Лабораторная работа №16 «Программирование с использованием древовидных структур данных»		2				Защита отчёта
Итого за 2-ой семестр:		6	8			94	Экзамен

4. ИНФОРМАЦИОННО-МЕТОДИЧЕСКАЯ ЧАСТЬ

4.1. Перечень литературы (учебной, учебно-методической, научной, нормативной, др.)

Основная

1. Брауде, Эрик Дж.Технология разработки программного обеспечения: пер. с англ. / Эрик Дж. Брауде. – СПб. и др.: Питер, 2018. – 654 с

2. Страуструп, Б. Язык программирования С++ / Б. Страуструп. – Москва : Бином, 2023. – 1216 с.

3. Лафоре, Р. Объектно-ориентированное программирование в С++ / Р. Лафоре. – 4-е изд. – Санкт-Петербург : Питер, 2016. – 928 с.

4. Навроцкий, А. А. Основы алгоритмизации и программирования: учебное пособие / А. А. Навроцкий, А. Б. Гуринович. – Минск : БГУИР, 2022. – 152 с.

Дополнительная

5. Брауде, Эрик Дж.Технология разработки программного обеспечения: пер. с англ. / Эрик Дж. Брауде. – СПб. и др.: Питер, 2018. – 654 с

6. Вирт, Н. Алгоритмы и структуры данных / Н. Вирт. – Санкт-Петербург : Невский Диалект, 2001. – 352 с.

7. ГОСТ 19.701-90 – Единая система программной документации – Схемы алгоритмов, программ, данных и систем – Условные обозначения и правила выполнения.

8. Кнут, Д. Искусство программирования. Т. 1–3 / Д. Кнут. – Москва : Вильямс, 2004. – 486 с.

9. Константайн, Ларри Разработка программного обеспечения / Ларри Константайн, Люси Локвуд. – СПб. и др.: Питер, 2014. – 592 с.

10. Шилдт, Г. Искусство программирования на С++ / Г. Шилдт. – Санкт-Петербург : БХВ-Петербург, 2005. – 496 с.

4.2. Перечень средств диагностики результатов учебной деятельности

– Письменные отчеты по лабораторным работам
– Письменный экзамен

4.3. Методические рекомендации по организации и выполнению самостоятельной работы обучающихся по учебной дисциплине

Самостоятельная работа предполагает:

– ведение и систематическую проработку конспекта лекций и учебной литературы;

– подготовку письменных отчетов по результатам выполнения индивидуальных заданий лабораторных работ;

– проработку учебного материала для защиты отчёта по выполненной работе;

– подготовку к письменному экзамену по дисциплине.

Ссылки на рекомендуемые источники, учебную литературу (в разрезе тем изучаемой дисциплины) представлены в таблице ниже.

	Тема учебной дисциплины	Литература
1	ВВЕДЕНИЕ. ОСНОВНЫЕ ПОНЯТИЯ	
1.1	Общие сведения об алгоритмах, ЭВМ и организации вычислительного процесса	1
1.2	Способы представления данных в ЭВМ	1
1.3	Общие сведения о программном обеспечении и технологии производства программного обеспечения	1, 5, 9
1.4	Интерфейс. Общие сведения	1, 5, 9
2	ОСНОВЫ ПРОГРАММИРОВАНИЯ НА АЛГОРИТМИЧЕСКОМ ЯЗЫКЕ С++	

2.1	Среда программирования	2, 4, 9
2.2	Основные понятия. Типы данных	2, 4, 6, 10
2.3	Операции, операторы и выражения	2, 4, 10
2.4	Структурированные типы данных	2, 4, 6, 10
2.5	Процедуры и функции	2, 4, 10
2.6	Дополнительные возможности	3, 10
3	СТАНДАРТНЫЕ АЛГОРИТМЫ	
3.1	Сортировка и поиск	6, 7, 8
4	АДРЕСНАЯ АРИФМЕТИКА. ДИНАМИЧЕСКАЯ ПАМЯТЬ. ДИНАМИЧЕСКИЕ СТРУКТУРЫ	
4.1	Указатели и адреса	4, 10
4.2	Указатели и функции	4, 10
4.3	Структуры данных	4, 10
4.4	Массивы указателей и структуры	4, 10
4.5	Динамические линейные структуры	4, 6, 10
4.6	Динамические нелинейные структуры	4, 6, 10