МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное бюджетное
образовательное учреждение высшего образования
«ПЕРМСКИЙ ГОСУДАРСТВЕННЫЙ
НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ»

М. М. Бузмакова

ИНФОРМАТИКА И ОСНОВЫ ПРОГРАММИРОВАНИЯ

КУРС ЛЕКЦИЙ

*Допущено методическим советом
Пермского государственного национального
исследовательского университета в качестве
учебного пособия для студентов, обучающихся
по направлениям подготовки бакалавров «Физика»,
«Радиофизика», «Прикладная математика и физика»,
«Нанотехнологии и микросистемная техника»*

Пермь 2017

УДК 004
ББК 32.9
Б90

Бузмакова М. М.
Б90 Информатика и основы программирования: курс лекций: учеб. пособие / М. М. Бузмакова; Перм. гос. нац. исслед. ун-т. – Пермь, 2017. – 180 с.

ISBN 978-5-7944-2998-5

Курс лекций содержит материалы по дисциплине «Информатика и основы программирования», предназначенные для студентов 1 курса физического факультета, обучающихся по направлениям «Физика», «Радиофизика», «Прикладная математика и физика», «Нанотехнологии и микросистемная техника». Издание может быть использовано в качестве учебного пособия как для освоения основной дисциплины, так и для подготовки к лабораторным, самостоятельным работам и итоговым контрольным мероприятиям.

Материалы курса лекций соответствуют учебной программе и УМК. Курс включает следующие разделы: теоретические основы информатики, основы теории кодирования, основы теории алгоритмов, основы языков и технологий программирования, теоретические сведения о языке программирования С/С++. Прилагается список литературы, который студенты могут использовать для более детального изучения какого-либо раздела курса.

УДК 004
ББК 32.9

Печатается по решению редакционно-издательского совета
Пермского государственного национального исследовательского университета

Рецензенты: кафедра информатики и вычислительной техники Пермского государственного гуманитарно-педагогического университета (зав. каф. – доцент, канд. пед. наук ***А. П. Шестаков***); доцент кафедры информатики и вычислительной техники Пермского государственного гуманитарно-педагогического университета, канд. физ.-мат. наук ***Т. Н. Катанова***

ISBN 978-5-7944-2998-5

© ПГНИУ, 2017
© Бузмакова М. М., 2017

Оглавление

Лекция 1.1 ... 6

Определение информатики, ее содержательная структура ... 6

Информатика и программирование ... 8

Определение и свойства информации ... 9

Непрерывное и дискретное представление информации ... 10

Системы счисления ... 12

Лекция 1.2 ... 15

Представление информации в памяти компьютера ... 15

Представление текстовой информации ... 16

Представление графической информации ... 17

Представление звуковой информации ... 18

Представление числовой информации ... 21

Лекция 1.3 ... 25

Измерение информации ... 25

Кодирование информации ... 27

Кодирование информации для сжатия данных. Коды Шеннона-Фано, Хаффмана, Хэмминга ... 29

Лекция 1.4 ... 34

Арифметические и логические основы ЭВМ ... 34

Устройство персонального компьютера ... 39

Принципы работы ЭВМ ... 42

Архитектура фон Неймана ... 43

Структура памяти компьютера ... 45

Устройство и работа процессора ... 46

Лекция 1.5 ... 49

Этапы решения задачи на ЭВМ ... 49

Понятие, свойства, исполнители и способы описания алгоритма ... 49

Структура алгоритма ... 52

Вспомогательные алгоритмы ... 55

Парадигмы и языки программирования 55
Лекция 1.6 60
Структурное программирование 60
Структура программы на С/С++ 61
Элементы языка С/С++ 62
Данные и величины, типы данных 63
Операции с данными 67
Организация ввода/вывода данных в С/С++ 70
Программы с линейной структурой 73
Лекция 1.7 74
Программы, содержащие ветвление 74
Программы, содержащие циклы 76
Лекция 1.8 80
Программное обеспечение ЭВМ 80
Системное ПО 80
Инструментальное ПО 82
Прикладное ПО 85
Лекция 1.9 87
Функции на С/С++ 87
Рекурсия 89
Математические функции 90
Классы памяти 91
Лекция 1.10 94
Структуры данных в С/С++ 94
Указатели и ссылки в С/С++ 96
Указатели и ссылки как параметры функции 99
Массивы. Основные алгоритмы обработки массивов 101
Лекция 1.11 106
Сортировки массивов 106
Лекция 1.12 114
Строки в С/С++ 114

Функции для работы со строками 116
Основные алгоритмы обработки строк 119
Класс string 119
Лекция 1.13 124
Понятие и оценка сложности алгоритма 124
Понятие оптимизации алгоритмов 127
Понятие сложности задачи, классы сложности задач 128
Лекция 2.1 130
Структуры, объединения, перечисления в C/C++ 130
Динамические структуры данных 135
Списки 136
Алгоритмы работы со списками 140
Лекция 2.2 144
Работа с файлами в C/C++ 144
Лекция 2.3 152
Объектно-ориентированное программирование 152
Основные принципы ООП 153
Основные понятия ООП 154
Уровни доступа к членам класса 157
Лекция 2.4 157
Конструкторы и деструкторы 157
Инкапсуляция 160
Лекция 2.5 162
Наследование 162
Полиморфизм 165
Друзья класса 167
Лекция 2.6 168
Библиотека STL 168
Список литературы 179

Лекция 1.1

Определение информатики, ее содержательная структура

Информатика как наука развивается сравнительно недавно, со второй половины прошлого века. Появление информатики как науки ассоциируется с появлением первых ЭВМ. Однако нельзя полагать, что до этого момента информатики не было, ведь если бы человечество не занималось исследованиями в данной области, то и ЭВМ не была бы создана. Следует разделить развитие информатики на два периода. Первый период связан с развитием устной речи человека, появлением письменности, книгопечатания, становлением точных наук, что, несомненно, обусловило создание ЭВМ и дальнейшее бурное развитие информационных и компьютерных технологий. Это можно назвать вторым периодом развития информатики. Именно во втором периоде информатика развивается как система взаимосвязанных наук, таких как кибернетика, синергетика, криптология, программирование, моделирование и многие другие.

Информатика проникла во все сферы человеческой деятельности. Информационные и компьютерные технологии используются в науке, образовании, промышленности, медицине, экономике, политике… Этот список, я думаю, не закончится. Трудно представить в наши дни ученика или студента без образовательных интернет-ресурсов, ученого без мощной вычислительной техники, заводы без автоматизированного производства, поликлиники и больницы без диагностического оборудования… Куда не посмотри, везде задействованы информационные технологии. Информатика стала неотъемлемой частью системы наук.

Несмотря на бурное развитие информатики как науки, единого определения понятия «информатика» до сих пор нет. Впервые термин «информатика» появился в середине 60-х гг. XX в. практически одновременно во Франции и России и использовался для обозначения самой молодой науки среди других естественных и технических наук. В 1963 г. в журнале «Известия вузов» была опубликована статья Ф.Е. Темникова «Информатика». В ней была представлена наука об информации как совокупность трёх разделов: теории информационных элементов, теории информационных систем и теории информационных процессов. Эта статья осталась незамеченной, и более популярным оказалось французское толкование термина «informatique», которым обозначили науку об электронно-вычислительных машинах (ЭВМ) и их применении. В США вместо термина «информатика» используют термин «computer science».

Рассмотрим некоторые определения информатики, предложенными разными авторами.

Информатика – наука, изучающая структуру и общие свойства информации, а также вопросы, связанные с её сбором, хранением, поиском, преобразо-

ванием, распространением и использованием в различных сферах человеческой деятельности.

Определение С.В. Симоновича: «Информатика – техническая наука, систематизирующая приёмы создания, хранения, воспроизведения, обработки и передачи данных средствами вычислительной техники, а также принципы функционирования этих средств и методы управления ими» [15].

Определение А. П. Ершова: «Информатика – это фундаментальная естественная наука, изучающая процессы передачи и обработки информации» [8].

Определение Д. С. Чернавского: «Информатика – наука о процессах передачи, возникновения, рецепции, хранения и обработки информации» [19].

Определение Французской академии наук: «Информатика – это наука об осуществляемой преимущественно с помощью автоматических средств целесообразной обработке информации, рассматриваемой как представление знаний и сообщений в технических, экономических и социальных областях» [4].

Большинство авторов склоняются к следующей структуре информатики как научной дисциплины: технические средства, программные средства и алгоритмические средства. Все три компоненты развиваются в тесной взаимосвязи. Их взаимодействие привело к развитию таких направлений информатики, как теория вычислений, алгоритмы и структуры данных, методология программирования и языков, компьютерные элементы и архитектура, разработка программного обеспечения, искусственный интеллект, компьютерные сети и телекоммуникации, системы управления базами данных, параллельные вычисления, распределённые вычисления, взаимодействия между человеком и компьютером, компьютерная графика, операционные системы, числовые и символьные вычисления. Области информатики можно представить в виде следующей схемы.

Рис. 1. Области информатики

К информатике как отрасли народного хозяйства можно отнести производство компьютерной техники, производство программных продуктов и разработку современных технологий переработки информации. Роль информатики как отрасли производства состоит в том, что от нее во многом зависит рост

производительности труда в других отраслях народного хозяйства. В современном обществе информация все чаще выступает как предмет конечного потребления: людям необходима информация о событиях, происходящих в мире; о предметах и явлениях, относящихся к их профессиональной деятельности; о развитии науки и самого общества. Дальнейший рост производительности труда и уровня благосостояния возможен лишь на основе использования новых интеллектуальных средств и человекомашинных интерфейсов, ориентированных на прием и обработку больших объемов мультимедийной информации.

К информатике как науке обычно относят методологию создания информационного обеспечения и теорию информационных систем и технологий. Основные научные направления фундаментальных исследований в информатике:

– разработка сетевой структуры;
– компьютерно-интегрированные производства;
– экономическая и медицинская информатика;
– информатика социального страхования и окружающей среды;
– профессиональные информационные системы.

Информатика как прикладная наука занимается изучением закономерностей в информационных процессах, созданием информационных моделей коммуникаций и разработкой информационных систем и технологий. Главная функция информатики как прикладной дисциплины заключается в разработке методов и средств преобразования информации и их использования в организации технологического процесса переработки информации.

Информатика и программирование

Невозможно развитие информатики в целом без программирования. Программирование – это процесс создания компьютерной программы, т. е. разработка программного обеспечения, будь то операционная система для компьютера, драйвер для какого-либо оборудования или же компьютерная игра. Полной классификации программного обеспечения будет посвящена отдельная лекция. Прежде чем получить какой-либо конечный продукт, необходимо пройти несколько этапов создания программы (решения задачи) на ЭВМ: постановка задачи, разработка модели, разработка алгоритма, программирование, тестирование и отладка, анализ результатов и дальнейшее сопровождение. Процесс программирования в свою очередь состоит из следующих подпроцессов: выбор языка программирования, уточнение способов организации данных, запись алгоритма на выбранном языке. Программированием занимаются программисты, которые разделяются по следующим основным категориям. *Системный программист* (system/software programmer, toolsmith) занимается разработкой, эксплуатацией и сопровождением системного программного обеспе-

чения, поддерживающего работоспособность компьютера и создающего среду для выполнения программ. *Прикладной программист* (application programmer) ведет разработку и отладку программ для решения функциональных задач. *Программист-аналитик* (programmer-analyst) – программист, анализирующий и проектирующий комплекс взаимосвязанных программ. *Постановщик задач* – разработчик формальных постановок задач, требующих реализации на ЭВМ. *Администратор базы данных* – человек, который обеспечивает ее организационную поддержку. *Администратор сети* – человек, который обеспечивает организационную поддержку работы локальной сети.

Программированию будет уделена основная часть данного курса. Однако прежде чем приступить к освоению программирования, необходимо познать теоретические основы информатики.

Определение и свойства информации

Ключевым понятием информатики является информация. Определений «информации» множество. В рамках данного курса будем пользоваться следующим. *Информация* – это сведения о каком-либо объекте, процессе, явлении, событии и т.д. С информацией связаны три основных понятия: источник информации, приемник информации и способ передачи информации.

Информацию классифицируют по способу восприятия, по форме представления и по общественному значению. Основные виды информации по способу восприятия: визуальная, аудиальная, тактильная, обонятельная, вкусовая. Основные виды информации по форме представления: текстовая, числовая, звуковая, мультимедийная. Основные виды информации по общественному значению: личная (знания, умения, навыки, интуиция), массовая (общественная, обыденная, эстетическая), специальная (научная, производственная, техническая, управленческая). В рамках информатики интерес представляет классификация информации по форме ее представления. Представлению различных видов информации в ЭВМ будет посвящена отдельная лекция.

Информация обладает следующими основными свойствами:

Объективность. Информация объективна, если не зависит от методов ее фиксации, чьего-либо мнения, суждения. Объективную информацию можно получить с помощью исправных датчиков, измерительных приборов. Отражаясь в сознании человека, информация может искажаться (в большей или меньшей степени) в зависимости от мнения, суждения, опыта, знаний конкретного субъекта и, таким образом, перестать быть объективной.

Достоверность. Информация достоверна, если отражает истинное положение дел. Объективная информация всегда достоверна, но достоверная ин-

формация может быть как объективной, так и субъективной. Достоверная информация помогает принять нам правильное решение. Недостоверной информация может быть по следующим причинам: преднамеренное искажение (дезинформация) или непреднамеренное искажение субъективного свойства; искажение в результате воздействия помех («испорченный телефон») и недостаточно точных средств ее фиксации.

Полнота. Информацию можно назвать полной, если ее достаточно для понимания и принятия решений. Неполная информация может привести к ошибочному выводу или решению.

Точность. Определяется степенью близости информации к реальному состоянию объекта, процесса, явления и т. п.

Актуальность. Важность для настоящего времени, злободневность, насущность. Только вовремя полученная информация может быть полезна.

Полезность (*ценность*). Полезность может быть оценена применительно к нуждам конкретных потребителей информации и оценивается по тем задачам, которые можно решить с ее помощью.

С течением времени количество информации растет, информация накапливается, происходит ее систематизация, оценка и обобщение. Это свойство назвали *ростом и кумулированием информации* (кумуляция – от лат. cumulatio – увеличение, скопление).

Старение информации заключается в уменьшении ее ценности с течением времени. Старит информацию не само время, а появление новой информации, которая уточняет, дополняет или отвергает полностью или частично более раннюю.

Логичность, *компактность*, *удобная* форма представления облегчают понимание и усвоение информации.

Непрерывное и дискретное представление информации

Язык – это система обозначений и правил передачи сообщений. Различают естественные языки (взаимодействие систем «Человек» – «Человек») и искусственные или формальные языки (взаимодействие систем «Человек» – «Машина», «Машина» – «Машина»). Формальный язык обеспечивает удобное описание проблем, формулируемых человеком и решаемых с помощью ЭВМ.

При любых формах работы с информацией имеет место ее представление в виде определенных символических структур. Например, последовательность символов, схемы, графики и т.д. Представление информации в различном физическом виде называется кодированием. Для того чтобы общаться с другими

людьми, человеку приходится постоянно заниматься кодированием, перекодированием и декодированием.

Схема передачи информации (сообщения) от источника к приемнику выглядит следующим образом

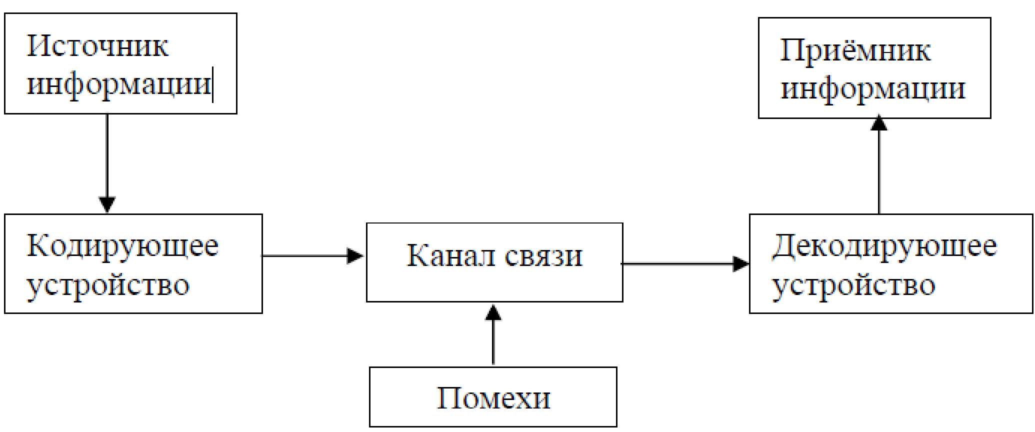

Рис. 2. Схема передачи информации

Сообщение от источника к приемнику передается в материально-энергетической форме (электрический, световой, звуковой и другие сигналы). Человек воспринимает информацию посредством органов чувств, технические приемники информации – посредством различной измерительной и регистрирующей аппаратуры. Полученное информационное сообщение можно представить в виде функции $x(t)$, t – время. Если функция $x(t)$ непрерывная, то мы имеем дело с непрерывной (аналоговой) информацией. Например, атмосферное давление, температура воздуха, влажность, давление и температура теплоносителя в ядерном реакторе. Если функция $x(t)$ дискретна, то мы имеем дело с дискретной (цифровой) информацией. Например, сигнал тревоги, звуковое сообщение и другие. В современном мире информация обрабатывается на вычислительных машинах, которые по виду обрабатываемой информации бывают аналоговыми и цифровыми. АВМ – это машина, оперирующая информацией в виде непрерывных изменений некоторых физических величин. ЦВМ – это машина, оперирующая информацией, представленной в дискретном виде. АВМ предназначена для решения определенного класса задач, ЦВМ является универсальным вычислительным средством. Электронные вычислительные машины стали наиболее популярными в последнее время. ЭВМ используют новейшие технологии электроники для обработки информации. Они универсальны и позволяют обрабатывать не только численную, но и любую другую информацию: текстовую, графическую, звуковую. ЭВМ способны принимать информацию от аналоговых источников, используя специальные устройства: аналогово-цифровые преобразователи. Информация после обработки на ЭВМ может пере-

водиться в аналоговую форму на специальных устройствах: цифроаналоговых преобразователях. Поэтому современные ЭВМ могут говорить, синтезировать музыку, рисовать, управлять машиной или станком. Не так заметно для всех, как ЭВМ, но развиваются и аналоговые системы обработки информации. А некоторые устройства аналоговой обработки информации до сих пор не нашли и, видимо, в ближайшем будущем не найдут себе достойной цифровой замены. Таким устройством, например, является объектив фотоаппарата. Вероятно, что будущее техники за так называемыми аналогово-цифровыми устройствами, использующими преимущества тех и других. По-видимому, органы чувств, нервная система и мышление построены природой как на аналоговой, так и на цифровой основе.

Системы счисления

Как уже говорилось, при работе с информацией имеет место ее представление в виде определенных символических структур. Текстовая информация представляется в виде букв (символов какого-либо алфавита). Числовая информация – в виде цифр. Рассмотрим представление числовой информации подробнее.

Система счисления (далее СС) – совокупность приемов и правил для записи чисел с помощью цифр. Любая предназначенная для практического применения СС должна обеспечивать:

- возможность представления любого числа в рассматриваемом диапазоне величин;
- единственность представления (каждой комбинации символов должна соответствовать одна и только одна величина);
- простоту оперирования числами.

В зависимости от способов изображения чисел цифрами системы счисления делятся на непозиционные и позиционные. *Непозиционной* системой называется такая, в которой количественное значение каждой цифры не зависит от занимаемой ею позиции в изображении числа. *Позиционной* системой счисления называется такая, в которой количественное значение каждой цифры зависит от её позиции в числе. Также существуют *смешанные* СС, которые объединяют в себе свойства позиционных и непозиционных. Количество знаков или символов, используемых для изображения числа, называется *основанием* системы счисления.

Позиционные системы счисления имеют ряд преимуществ перед непозиционными: удобство выполнения арифметических и логических операций, а

также представление больших чисел, поэтому в цифровой технике применяются позиционные системы счисления.

Число в позиционной СС представляется в виде

$$X = \sum_{i=0}^{n} a_i b^i ,$$

где a_i – цифра, стоящая на i-й позиции в числе, b – основание СС. Если речь идет о дробном числе, то в сумме i меняется не от 0, а от отрицательного числа, указывающего на число знаков в числе после запятой.

Пример. Число 547,35 в десятичной СС:

$$547{,}35 = 5*10^2 + 4*10^1 + 7*10^0 + 3*10^{-1} + 5*10^{-2}.$$

Среди позиционных СС чаще всего используются десятичная (0, 1, …, 9), двоичная (0, 1), восьмеричная (0, 1, …, 7), шестнадцатеричная (0, 1, …, 9, A, B, C, D, E, F). Среди непозиционных СС известны римская СС, биномиальная СС, система остаточных классов, древнегреческая и другие.

Например, число в биномиальной СС представляется по формуле

$$X = \sum_{k=1}^{n} \binom{c_k}{k},$$

где $0 \le c_1 < c_2 < ... < c_n$.

Среди смешанных СС наиболее известными являются факториальная СС, Фибоначчиева СС, формулы для которых приведены ниже:

$$X = \sum_{k=1}^{n} d_k k!,$$

где $0 \le d_k \le k$, и

$$X = \sum_{k} f_k F_k ,$$

где $f_k \in \{0.1\}$, F_k – числа Фибоначчи.

Так как человек использует десятичную СС, а вычислительная техника двоичную, восьмеричную, шестнадцатеричную (2^n), то следует рассмотреть подробнее эти СС и перевод чисел из одной системы счисления в другую. Понятно, что название СС исходит из значения основания (10, 2, 8, 16) и соответственно в каждой СС используются цифры от 0 до значения ее основания. В шестнадцатеричной после цифры «9» следуют буквы латинского алфавита, которые заменяют цифры: «A» вместо «10», «B» – «11», «C» – «12», «D» – «13», «E» – «14» и «F» – «15». Рассмотрим правила перевода из одной СС в другую.

Правила перевода целого числа из десятичной СС в СС с основанием *b*:

а) исходное целое число делится на основание системы счисления, в которую переводится (b), получается частное и остаток;

b) если полученное частное не делится на основание системы счисления так, чтобы образовалась целая часть, отличная от нуля, процесс умножения прекращается, переходят к шагу c). Иначе над частным выполняют действия, описанные в шаге a);
c) все полученные остатки и последнее частное преобразуются в соответствии с таблицей в цифры той системы счисления, в которую выполняется перевод;
d) формируется результирующее число: его старший разряд – полученное последнее частное, каждый последующий младший разряд образуется из полученных остатков от деления, начиная с последнего и кончая первым. Таким образом, младший разряд полученного числа – первый остаток от деления, а старший – последнее частное.

Правила перевода дробного числа десятичной СС в СС с основанием *b*:

a) исходная дробь умножается на основание системы счисления, получается целая и дробная части;
b) целая часть отбрасывается, оставшаяся дробная часть произведения снова умножается на основание системы счисления;
c) действие b) производится до тех пор, пока дробная часть произведения не будет равна нулю или же не будет достигнута необходимая точность вычисления;
d) формируется результирующее число: полученные целые части числа являются разрядами числа в новой системе, и их необходимо представлять цифрами этой новой системы счисления.

Правила перевода целого и дробного числа из СС с основанием *b* в десятичную СС:

Чтобы перевести число из *b*-й СС в десятичную достаточно исходное число разложить по вышеприведенной формуле $X = \sum_{i=-m}^{n} a_i b^i$.

Правила перевода из двоичной в восьмеричную (шестнадцатиричную) СС:

Исходное двоичное число делится от запятой влево и вправо на группы из трех (четырех) цифр – триада (тетрады), каждая триада (тетрада) заменяется цифрой восьмеричной (шестнадцатиричной) СС согласно таблице перевода.

Правила перевода из восьмеричной (шестнадцатиричной) в двоичную СС:

Каждая цифра исходного восьмеричного (шестнадцатиричного) числа заменяется на триаду (тетраду) двоичных чисел согласно таблице перевода (табл. 1).

Таблица 1. Соответствие цифр десятичной, двоичной, восьмеричной и шестнадцатиричной СС

10	2	2-8	8	2-16	16
0	0	000	0	0000	0
1	1	001	1	0001	1
2	10	010	2	0010	2
3	11	011	3	0011	3
4	100	100	4	0100	4
5	101	101	5	0101	5
6	110	110	6	0110	6
7	111	111	7	0111	7
8	1000	001000	10	1000	8
9	1001	001001	11	1001	9
10	1010	001010	12	1010	A
11	1011	001011	13	1011	B
12	1100	001100	14	1100	C
13	1101	001101	15	1101	D
14	1110	001110	16	1110	E
15	1111	001111	17	1111	F
16	10000	010000	20	00010000	10

Лекция 1.2
Представление информации в памяти компьютера

Компьютер может обрабатывать информацию, представленную только в числовой форме. Вся другая информация для обработки на компьютере должна быть преобразована в числовую форму. Компьютер может обрабатывать числовую, текстовую, графическую, звуковую, видеоинформацию только тогда, когда она представлена в нем в двоичном коде (двоичная форма представления информации), т. е. используется алфавит мощностью в два символа: логический 0 и логическая 1. Связано это с тем, что информацию удобно представлять в виде последовательности сигналов (электрических импульсов): сигнал отсутствует – (0), сигнал есть – (1). Такое кодирование принято называть двоичным, а сами логические последовательности нулей и единиц – машинным языком. Каждая цифра машинного двоичного кода несет количество информации, равное одному биту.

Бит – это единица информации, представляющая собой двоичный разряд, который может принимать значение 0 или 1. При записи двоичной цифры мож-

но реализовать выбор только одного из двух возможных состояний, а значит, она несет количество информации, равное одному биту. Следовательно, две цифры несут информацию 2 бита, четыре разряда – 4 бита и т. д. Чтобы определить количество информации в битах, достаточно определить количество цифр в двоичном машинном коде. Благодаря введению понятия единицы информации появилась возможность определения размера любой информации числом бит. Поэтому объем информации определяют в битах. Подробно измерение информации будет рассмотрено в следующей лекции.

Для удобства информацию, представленную в компьютере, описывают многоразрядными последовательностями двоичных чисел. Эти последовательности объединяются в группы по 8 бит. Такая группа именуется байтом. Например, число 11010011 – это информация величиной 1 байт. Байт – это восемь последовательных бит. В 1 байте можно кодировать значение одного символа из 256 (2^8) возможных комбинаций.

Более крупными единицами информации являются килобайт (Кбайт), мегабайт (Мбайт), гигабайт (Гбайт): 1 Кбайт = 1024 байт; 1 Мбайт = 1024 Кбайт; 1 Гбайт = 1024 Мбайт. В этих единицах измеряется емкость запоминающих устройств.

Представление текстовой информации

В настоящее время большинство пользователей с помощью компьютера обрабатывают текстовую информацию, которая состоит из символов: букв, цифр, знаков препинания и др. Чтобы закодировать один символ, традиционно используют количество информации, равное 1 байту, т. е. I = 1 байт = 8 бит. В 60-е гг. XX в. это было закреплено комитетом ASCII США в ASCII-стандарте.

Суть кодирования заключается в том, что каждому символу ставят в соответствие двоичный код от 00000000 до 11111111 или соответствующий ему десятичный код от 0 до 255.

В середине 90-х гг. XX в. появилась новая кодировка – Unicode, поддерживающая 65 536 различных символов. В ней на каждый символ отводится по 2 байта:

$K = 2^{16} = 65\ 536$.

Следует отметить, что кодировка Unicode используется в случаях, когда к кодированию не предъявляются дополнительные требования (например, когда необходимо указать на возникшую ошибку, исправить ошибку, обеспечить секретность информации или использовать ее в различных операционных системах).

Таблица 2. Стандартная часть кода ASCII

<table>
<tr><td>32</td><td>20</td><td></td><td>56</td><td>38</td><td>8</td><td>80</td><td>50</td><td>P</td><td>104</td><td>68</td><td>h</td></tr>
<tr><td>33</td><td>21</td><td>!</td><td>57</td><td>39</td><td>9</td><td>81</td><td>51</td><td>Q</td><td>105</td><td>69</td><td>i</td></tr>
<tr><td>34</td><td>22</td><td>“</td><td>58</td><td>3A</td><td>:</td><td>82</td><td>52</td><td>R</td><td>106</td><td>6A</td><td>j</td></tr>
<tr><td>35</td><td>23</td><td>#</td><td>59</td><td>3B</td><td>;</td><td>83</td><td>53</td><td>S</td><td>107</td><td>6B</td><td>k</td></tr>
<tr><td>36</td><td>24</td><td>$</td><td>60</td><td>3C</td><td><</td><td>84</td><td>54</td><td>T</td><td>108</td><td>6C</td><td>l</td></tr>
<tr><td>37</td><td>25</td><td>%</td><td>61</td><td>3D</td><td>=</td><td>85</td><td>55</td><td>U</td><td>109</td><td>6D</td><td>m</td></tr>
<tr><td>38</td><td>26</td><td>&</td><td>62</td><td>3E</td><td>></td><td>86</td><td>56</td><td>V</td><td>110</td><td>6E</td><td>n</td></tr>
<tr><td>39</td><td>27</td><td>‘</td><td>63</td><td>3F</td><td>?</td><td>87</td><td>57</td><td>W</td><td>111</td><td>6F</td><td>o</td></tr>
<tr><td>40</td><td>28</td><td>(</td><td>64</td><td>40</td><td>@</td><td>88</td><td>58</td><td>X</td><td>112</td><td>70</td><td>p</td></tr>
<tr><td>41</td><td>29</td><td>)</td><td>65</td><td>41</td><td>A</td><td>89</td><td>59</td><td>Y</td><td>113</td><td>71</td><td>q</td></tr>
<tr><td>42</td><td>2A</td><td>*</td><td>66</td><td>42</td><td>B</td><td>90</td><td>5A</td><td>Z</td><td>114</td><td>72</td><td>r</td></tr>
<tr><td>43</td><td>2B</td><td>+</td><td>67</td><td>43</td><td>C</td><td>91</td><td>5B</td><td>[</td><td>115</td><td>73</td><td>s</td></tr>
<tr><td>44</td><td>2C</td><td>,</td><td>68</td><td>44</td><td>D</td><td>92</td><td>5C</td><td>\</td><td>116</td><td>74</td><td>t</td></tr>
<tr><td>45</td><td>2D</td><td>-</td><td>69</td><td>45</td><td>E</td><td>93</td><td>5D</td><td>]</td><td>117</td><td>75</td><td>u</td></tr>
<tr><td>46</td><td>2E</td><td>.</td><td>70</td><td>46</td><td>F</td><td>94</td><td>5E</td><td>^</td><td>118</td><td>76</td><td>v</td></tr>
<tr><td>47</td><td>2F</td><td>/</td><td>71</td><td>47</td><td>G</td><td>95</td><td>5F</td><td>_</td><td>119</td><td>77</td><td>w</td></tr>
<tr><td>48</td><td>30</td><td>0</td><td>72</td><td>48</td><td>H</td><td>96</td><td>60</td><td>‘</td><td>120</td><td>78</td><td>x</td></tr>
<tr><td>49</td><td>31</td><td>1</td><td>73</td><td>49</td><td>I</td><td>97</td><td>61</td><td>a</td><td>121</td><td>79</td><td>y</td></tr>
<tr><td>50</td><td>32</td><td>2</td><td>74</td><td>4A</td><td>J</td><td>98</td><td>62</td><td>b</td><td>122</td><td>7A</td><td>z</td></tr>
<tr><td>51</td><td>33</td><td>3</td><td>75</td><td>4B</td><td>K</td><td>99</td><td>63</td><td>c</td><td>123</td><td>7B</td><td>{</td></tr>
<tr><td>52</td><td>34</td><td>4</td><td>76</td><td>4C</td><td>L</td><td>100</td><td>64</td><td>d</td><td>124</td><td>7C</td><td>|</td></tr>
<tr><td>53</td><td>35</td><td>5</td><td>77</td><td>4D</td><td>M</td><td>101</td><td>65</td><td>e</td><td>125</td><td>7D</td><td>}</td></tr>
<tr><td>54</td><td>36</td><td>6</td><td>78</td><td>4E</td><td>N</td><td>102</td><td>66</td><td>f</td><td>126</td><td>7E</td><td>~</td></tr>
<tr><td>55</td><td>37</td><td>7</td><td>79</td><td>4F</td><td>O</td><td>103</td><td>67</td><td>g</td><td>127</td><td>7F</td><td></td></tr>
</table>

Представление графической информации

В середине 50-х гг. XX в. для больших ЭВМ, которые применялись в научных и военных исследованиях, впервые было реализовано представление данных в графическом виде. В настоящее время широко используются технологии обработки графической информации с помощью персонального компьютера (ПК). Особенно интенсивно технология обработки графической информации с помощью компьютера стала развиваться в 80-х гг. Графическую информацию можно представлять в двух формах: аналоговой и дискретной. Живописное полотно, цвет которого изменяется непрерывно, – это пример аналогового представления, а изображение, напечатанное с помощью струйного принтера и состоящее из отдельных точек разного цвета, – это дискретное представление.

Для представления графической информации в двоичной форме используются *растровый* и *векторный* способы.

При *растровом* способе вертикальными и горизонтальными линиями изображение разбивается на отдельные точки; каждому элементу ставятся в соответствие коды его цвета и место, которое он занимает. При этом качество кодирования будет зависеть от размера точки и количества используемых цветов. Чем меньше размер точки (т. е. изображение составляется из большего количества точек), тем выше качество кодирования. Чем большее количество цветов используется (т. е. точка изображения может принимать больше возможных состояний), тем больше информации несет каждая точка, а значит, увеличивается качество кодирования. Поэтому информация о каждой клетке будет иметь довольно сложный вид: номер клетки, яркость, тон, насыщенность, цвет и др.

В *векторном* способе информация вычисляется по специальным формулам, описывающим какой-либо объект – геометрический примитив (точка, прямая, дуга и другие).

Можно сочетать векторный и растровый способы формирования изображений (3D графика).

Представление звуковой информации

Звуковую информацию можно представить в дискретной и аналоговой формах. Их отличие в том, что при дискретном представлении информации физическая величина изменяется скачкообразно, принимая конечное множество значений. Если же информацию представить в аналоговой форме, то физическая величина может принимать бесконечное количество значений, непрерывно изменяющихся.

Семпл – это промежуток времени между двумя измерениями амплитуды аналогового сигнала. Дословно sample переводится с английского как «образец». В мультимедийной и профессиональной звуковой терминологии это слово имеет несколько значений. Семплом называют также любую последовательность цифровых данных, которые получили путем аналого-цифрового преобразования. Сам процесс преобразования называют семплированием или дискредитацией.

Важными параметрами семплирования являются *частота* и *разрядность*.

Частота – это количество измерений амплитуды аналогового сигнала в секунду. Если частота семплирования не будет более чем в два раза превышать частоту верхней границы звукового диапазона, то на высоких частотах будут происходить потери. Так как диапазон колебаний звуковых волн находится в пределах от 20 Гц до 20 кГц, то стандартной является частота 44,1 кГц, которая

выбрана с таким расчетом, чтобы количество измерений сигнала в секунду было больше, чем количество колебаний за тот же промежуток времени.

Разрядность указывает, с какой точностью происходят изменения амплитуды аналогового сигнала. Точность, с которой при оцифровке передается значение амплитуды сигнала в каждый из моментов времени, определяет качество сигнала после цифро-аналогового преобразования.

Звук – волновые колебания давления в упругой среде (воздухе, воде, металле и т.д.). Для обозначения звука часто используют термин «звуковая волна».

Основные параметры любых волн и звуковых в том числе – частота и амплитуда колебаний. Частоту звука измеряют в герцах (Гц – количество колебаний в секунду). Человеческое ухо способно воспринимать звук в широком диапазоне частот, от 16 Гц до 20 кГц. В нетехнических областях (например, в музыке) вместо термина «частота» используют термин «тон». Амплитуду звуковых колебаний называют звуковым давлением или силой звука, эта величина характеризует воспринимаемую громкость звука.

Сила звука измеряется в паскалях (Па), при этом самые слабые, едва различимые звуки имеют амплитуду 20 мкПа (2×10^{-5} Па – порог слышимости). Самые сильные звуки, ещё не выводящие слуховые органы из строя, могут иметь амплитуду до 200 Па (болевой порог). Из-за такого широкого диапазона значений (максимальное и минимальное значения отличаются на 6-7 порядков!) абсолютными величинами звукового давления пользоваться неудобно, на практике обычно используют логарифмическую шкалу децибелов.

Относительную силу звука, или уровень звука, определяют как логарифм отношения абсолютной величины звукового давления к величине порога слышимости, умноженный на некоторый постоянный коэффициент:

$$L = 20 * \lg \frac{Pзз}{Pпп},$$

где L – уровень звука (в децибелах, обозначение - дБ), $P_{пс}$ – порог слышимости (2×10^{-5} Па), $P_{зв}$ – давление измеряемого звука.

Свойства логарифмической шкалы децибелов: весь диапазон слышимых звуков составляет 0-140 дБ: 0 дБ – порог слышимости, 140 дБ – болевой порог; человеческое ухо способно уловить различие в громкости, если звуки отличаются по силе не менее чем на 10%, что соответствует разнице в уровнях примерно на 1 дБ;

Звукозапись – процесс сохранения информации о параметрах звуковых волн. Способы хранения или записи звука разделяются на аналоговые и цифровые. При аналоговой записи на носителе размещается непрерывный «слепок» звуковой волны. Так, на грампластинке пропечатывается непрерывная канавка,

изгибы которой повторяют амплитуду и частоту звука, а на магнитной ленте параметры звука сохраняются в виде намагниченности рабочей поверхности, а степень намагниченности непрерывно изменяется, повторяя параметры звука.

В компьютерах применяется исключительно цифровая форма записи звука. При цифровой записи звук необходимо подвергнуть временной дискретизации и квантованию.

Временная дискретизация звука. Для того чтобы компьютер мог обрабатывать звук, непрерывный звуковой сигнал должен быть преобразован в цифровую дискретную форму с помощью временной дискретизации. Непрерывная звуковая волна разбивается на отдельные маленькие временные участки, для каждого такого участка устанавливается определенная величина интенсивности звука. Такой процесс называется оцифровкой звука.

Качество полученного звука зависит от количества измерений уровня громкости звука в единицу времени, т.е. частоты дискретизации. Чем большее количество измерений производится за 1 сек., тем выше качество записанного звука. Частота дискретизации звука – это количество измерений громкости звука за одну секунду. Одно измерение в секунду соответствует частоте 1Гц, 1000 измерений в секунду – 1 кГц. Частота дискретизации звука может лежать в диапазоне от 8000 до 48000 измерений громкости звука за одну секунду.

Глубина кодирования звука. Каждая звуковая карта характеризуется количеством распознаваемых уровней громкости звука, которое зависит от глубины кодирования звука. Глубина кодирования звука (измеряется в битах) – это количество информации, которое необходимо для кодирования одного значения громкости цифрового звука. Если известна глубина кодирования, то количество уровней громкости цифрового звука можно рассчитать. Пусть глубина кодирования звука составляет 16 бит, тогда количество уровней громкости звука будет равно $N = 2^{16} = 65536$.

Очевидно, что 16-битные звуковые карты точнее кодируют и воспроизводят звук, чем 8-битные. Качество звука в дискретной форме может быть очень плохим (при 8 битах и частоте дискретизации 5,5 кГц) и очень высоким (при 16 битах и частоте дискретизации 48 или 96 КГц). Формула расчета размера цифрового моноаудиофайла:

$$V = M * I * t,$$

где I – глубина звука (в битах); t – время звучания (в секундах); M – частота дискретизации (в Гц). Если звук записывается для нескольких источников воспроизведения (колонки, колонки + сабвуфер и т.д.), то объем умножается на соответствующий множитель.

Примеры:

1. Оценить информационный объем цифрового стереозвукового файла (длительность звучания 1 сек. при глубине кодирования звука 16 бит и частоте дискретизации 24 кГц).

N =24000Гц *16 бит * 2(стереозвук!) * 1сек=768000 бит = 93,75 Кбайт.

2. Какой минимальный объём памяти (в мегабайтах) потребуется для хранения звукового сигнала длительностью 4 мин., оцифрованного с частотой дискретизации 44032 Гц при 16-разрядном квантовании амплитуды?

N =44032Гц *16 бит * 240сек=169082880 бит ≈ 20,16 МБайт.

Представление числовой информации

В математике широко используются две формы записи чисел: естественная и нормальная. Примеры естественной формы записи чисел: 1 256 – целое число, 0.0357 – правильная дробь, 4.89760 – неправильная дробь.

Запись одного числа в нормальной форме может быть различной в зависимости от ограничений, накладываемых на ее форму. Например, число 1256 может быть записано так: $1256 = 1{,}256*10^3 = 0{,}1256* 10^4 = 12\ 560*10^{-1}$ и т.д.

При естественном представлении чисел в ЭВМ устанавливаются длины разрядной сетки, а также целой и дробной частей. При этом распределение разрядов между целой и дробной частями не изменяется и остается постоянным независимо от величины числа. В связи с этим такое представление числа называется формой с фиксированной запятой. В современных вычислительных машинах эта форма используется преимущественно для представления целых чисел.

Так как числа бывают положительные и отрицательные, то в разрядной сетке при машинном представлении один разряд отводится под знак числа, а остальные образуют поле числа. В знаковый разряд, который обычно располагается перед старшим разрядом числа, записывается информация о знаке числа. Знак положительного числа «+» изображается символом 0, а знак отрицательного числа «–» – символом 1. Если поле числа включает в себя *n* разрядов, то диапазон представления целых чисел в этом случае ограничивается значениями $-(2^n - 1)$ и $+(2^n - 1)$.

Представление числа в ЭВМ в нормальной форме обычно называют представлением с плавающей запятой, так как положение запятой в записи числа, как показывают приведенные примеры, в этом случае может изменяться. В нормальной форме запись числа *A* имеет структуру

$$A = m \circ q^p ,$$

где *m* – мантисса числа *A*; *p* – порядок числа *A* (характеристика числа).

Чтобы избежать неоднозначности представления чисел, используют так называемую нормализованную форму: m – число с запятой, фиксированной после определённого разряда. В двоичной системе представления чисел в ПК запятая фиксируется после первой значащей цифры (единицы) и поскольку кроме единицы там ничего быть не может, то эту единицу можно не хранить в памяти ЭВМ (скрытый или неявный бит).

В других системах счисления запятую в мантиссе принято располагать перед первой значащей цифрой и для m справедливо следующее условие:

$$q^{-1} \leq |m_A| < 1,$$

где q – основание системы счисления.

Так, числа $12{,}5*10^2$ и $-\ 0{,}00543*10^{-5}$ в нормализованном виде в соответствии с указанным выше условием должны быть записаны следующим образом:

$$0{,}125*10^4; -\ 0{,}543*10^{-7}.$$

Формат машинного изображения числа с плавающей запятой содержит знаковые части и поля для мантиссы и порядка. Кодирование знаков остается таким же, как и при представлении числа в форме с фиксированной запятой.

Существует стандарт на представление чисел в ПК: IEEE-754 и более новый и общий IEEE-854.

В процессоре все числа хранятся в едином 80-битном расширенном формате. В оперативной памяти целые числа могут занимать 16, 32 или 64 бита, а числа с плавающей точкой – 32, 64 или 80 бит. 18-разрядные десятичные числа автоматически преобразуются в расширенный формат при загрузке в регистры процессора. Результаты вычислений при записи в оперативную память преобразуются в исходный формат, который был задан до загрузки в регистры процессора.

Вещественные числа в каждом из трех форматов имеют три поля: поле знака мантиссы S, поле порядка и поле мантиссы. Мантисса числа записывается в нормализованном виде:

$$1.m_1m_2 \ldots m_i.$$

Целая часть, всегда равная 1, прямо не представляется в форматах DD и DQ, а учитывается неявно. В формате DT старший бит мантиссы представляется явно. Порядок вещественных чисел E записывается в поле порядка в смещенном виде. Он равен истинному порядку, увеличенному на значение смещения:

$$E = \text{истинный порядок} + \text{смещение}.$$

Смещенный порядок называется также характеристикой; ее можно считать целым положительным или беззнаковым числом. Задание порядка в форме со смещением упрощает операцию сравнения чисел с плавающей точкой, пре-

вращая ее в операцию сравнения целых чисел. Значение смещения для соответствующих форматов равно 127,1023 и 16383:

$$(-1)^S\times(1.\ m_1m_2 \dots m_{23})\times2^{E-127} \quad \text{для DD;}$$
$$(-1)^S\times(1.\ m_1m_2 \dots m_{52})\times2^{E-1023} \quad \text{для DQ;}$$
$$(-1)^S\times(1.\ m_1m_2 \dots m_{64})\times2^{E-16383} \quad \text{для DT.}$$

Все числа хранятся в памяти так, что младшая цифра находится по меньшему (начальному) адресу. Численные команды обращаются к операндам и сохраняют результаты, используя только этот начальный адрес.

S	характеристика	мантисса	DD
31	30 23	22 0	

S	характеристика	мантисса	DQ
63	62 52	51 0	

S	характеристика	мантисса	DT
79	78 64	63 0	

S		IW
15	14 0	

S		ID
31	30 0	

S		IQ
63	62 0	

Таблица 3. Форматы чисел в памяти ПК

Формат данных	**Число байт**	**Число значащих десятичных цифр**	**Диапазон значений**
Короткое вещественное DD	4	6–7	$\pm1.2\times10^{-38}\div3.4\times10^{38}$
Длинное вещественное DQ	8	15–16	$\pm2.3\times10^{-308}\div1.8\times10^{308}$
Расширенное вещественное DT	10	19–20	$\pm3.4\times10^{-4932}\div1.2\times10^{4932}$
Целое слово IW	2	4–5	(-32768) ÷(+32767)
Короткое целое ID	4	9	$-2^{31}\div2^{31}-1$
Длинное целое IQ	8	18	$-2^{63}\div2^{63}-1$

Представление целых чисел в ПК

Чтобы получить внутреннее представление *целого положительного числа N*, нужно:

– перевести число N в двоичную систему счисления,
– полученный результат дополнить слева незначащими нулями до полного заполнения формата целого числа.

Пример. Получить внутреннее представление 97_{10}.

97|16 97_{10}=61_{16}=0110 0001_2
96 6
1

0	000 0000 0110 0001
15	14 0

Используя шестнадцатеричную систему, этот код можно написать в 4 раза короче: 0061.

Для записи внутреннего представления *целого отрицательного числа -N*, нужно:

– получить внутреннее представление целого положительного числа *N*;
– получить обратный код этого числа заменой 0 на 1 и 1 на 0 (инвертирование);
– к полученному числу прибавить 1 – получается дополнительный код.

При сложении числа в дополнительном коде с его положительным значением знаковые разряды складываются как единое целое вместе с числом. Перенос единицы из знакового разряда выходит за пределы ячейки памяти и теряется. Следовательно, $N + (-N) = 0$, как и должно быть.

Примеры:

1. Получить внутреннее представление -7_{10}.

1) Перевод: $7_{10}=111_2$;

2)

1	000 0000 0000 0111
15	14 0

3)

1	111 1111 1111 1000
15	14 0

4)

1	111 1111 1111 1001
15	14 0

В шестнадцатеричной форме: FFF9.

2. Сложим два числа $+7_{10}$ и $–7_{10}$.

	0000 0000 0000 0111	+7
	1111 1111 1111 1001	-7
1	0000 0000 0000 0000	0

- перевести $A_{10} \rightarrow A_2$;
- нормализовать число;
- выполнить смещение порядка;
- перевести смещенный порядок $E_{10} \rightarrow E_2$;
- записать число в подходящем формате.

Пример. Число 98.4 записать во внутреннем представлении.

1)

```
98 |16     98₁₀ = 62₁₆ =0110 0010₂ ;        ×|2    98.4₁₀ = 0110 0010.00110011₂ ;
96   6                                       0|4
  2                                          0|8
                                             1|6
                                             1|2
                                             0|4
```

$98_{10} = 62_{16} = 0110\ 0010_2$; $98.4_{10} = 0110\ 0010.00110011_2$;

2) $1.1000\ 1001\ 1001\ldots \times 2^6$;

3) $E_{10} = 127+6=133$;

4)

```
133  |16        133₁₀ = 85₁₆  = 1000 0101₂ ;
128    8
   5
```

$133_{10} = 85_{16} = 1000\ 0101_2$;

5) Подходящий формат DD.

0	100 0010 1	100 0100 1100 1000 0000 0000
31	30 23	22 0

=> 42C4C800

Лекция 1.3

Измерение информации

При анализе информации можно получить ее количественные характеристики. На сегодняшний день основными метолами измерения информации являются объемный, энтропийный и алгоритмический.

Объемный способ измерения информации самый распространенный. Заключается в том, что объем информации равен количеству передаваемых символов. Так как любая информация в вычислительной технике закодирована в двоичной системе счисления, то введены следующие единицы измерения – *бит* и *байт*. Восемь битов составляют один байт.

Энтропийный способ измерения информации основан на том, что получатель информации имеет представление о возможности некоторых событий, которые могут произойти с некоторой вероятностью. Общая мера неопределенности (энтропия) характеризуется некоторой математической зависимостью

от совокупности этих вероятностей. Количество информации в данном случае определяется тем, насколько уменьшится эта мера после ее получения. В 1928 г. Р. Хартли, а в 1948 г. К. Шеннон предложили формулы для вычисления количества информации, основанные на энтропии. Р. Хартли первым ввел в теорию передачи информации методологию измерения количества информации, при этом он считал, что информация – это «группа физических символов» и задача измерения кодированной информации.

Информационная энтропия – неопределённость появления какого-либо символа первичного алфавита. При отсутствии информационных потерь численно равна количеству информации на символ передаваемого сообщения.

Например, в последовательности букв, составляющих какое-либо предложение на русском языке, разные буквы появляются с разной частотой, поэтому неопределённость появления некоторых букв меньше, чем других. Если же учесть, что некоторые сочетания букв (в этом случае говорят об энтропии n-го порядка) встречаются очень редко, то неопределённость ещё более уменьшается. Информационная энтропия источника определяется соотношением

$$H(x) = -\sum_{i=1}^{n} p(i)\log_2 p(i), \qquad (3.1)$$

где $p(i)$ – вероятность появления в тексте символа i.

Формула Хартли. Пусть передается последовательность из n символов $a_1a_2...a_n$, каждый из которых принадлежит алфавиту A_m (m – число символов в алфавите). Число различных вариантов таких последовательностей $K=m^n$. Тогда количество информации, содержащейся в такой последовательности вычисляется по формуле

$$I = \log_2 K. \qquad (3.2)$$

Замечание 1. Хартли предполагал, что все символы алфавита могут с равной вероятностью встретиться в любом месте.

Замечание 2. Любое сообщение длины n в одном алфавите будет содержать одинаковое количество информации, т. е. в формуле Хартли не учитывается смысловое содержание информации.

Формула Шеннона. Пусть передается последовательность из n символов $a_1a_2...a_n$, каждый из которых принадлежит алфавиту A_m (m – число символов в алфавите). Вероятность появления символа a_i в последовательности равна P_i. Тогда количество информации, содержащейся в такой последовательности, вычисляется по формуле

$$I = -n \cdot \sum_{i=1}^{m} P_i \log_2 P_i. \qquad (3.3)$$

Количество информации, которое несет в себе один символ, равно

$$I_S = -\log_2 P_S. \quad (3.4)$$

При равновероятных событиях формула Шеннона переходит в формулу Хартли.

Алгоритмический способ измерения информации заключается в том, чтобы оценить сложность (размер) программы, созданной для воспроизведения того или иного сообщения. Понятно, что для воспроизведения случайной последовательности 0 и 1 нужна программа, по объему значительно превышающая программу, которая будет воспроизводить одни 0, например.

Кодирование информации

Теория кодирования – это раздел теории информации, изучающий способы отображения дискретных сообщений сигналами в виде определенных сочетаний символов.

Кодирование информации используется при передаче сообщений (азбука Морзе, телеграф и другие), для сжатия информации и устранения возникающих при этом ошибок (помех, шумов). Кроме того, кодирование информации используется при шифровании данных для их защиты от несанкционированного доступа. Наука, занимающаяся шифрованием и дешифрованием данных, называется *криптологией*, которая состоит из двух разделов: *криптография* занимается разработкой алгоритмов шифрования данных; а *криптоанализ* исследует способы дешифрования данных без знания ключей и оценивает надежность алгоритмов шифрования.

На сегодняшний день основными целями теории кодирования являются:

- разработка принципов наиболее экономного представления информации;
- согласование параметров передаваемой информации с особенностями канала связи;
- разработка приемов повышения надежности передачи информации.

Задача кодирования – это задача перевода дискретного сообщения из одного алфавита в другой, причем такое преобразование не должно приводить к потере информации. Алфавит, с помощью которого представляется информация до преобразования, называется *первичным*, а алфавит конечного представления – *вторичным*. При определении понятия «код» используют два подхода. С одной стороны, код – это правило, описывающее соответствие знаков или их сочетаний первичного (исходного) алфавита знакам или их сочетаниям вторичного алфавита. Кодом называют также набор знаков вторичного алфавита, используемый для представления знаков или их сочетаний первичного алфавита.

Кодирование – это перевод информации, представленной символами первичного алфавита, в последовательность кодов.

Декодирование – операция, обратная кодированию, – перевод последовательности кодов в соответствующий набор символов первичного алфавита.

Кодер – устройство, обеспечивающее выполнение операции кодирования.

Декодер – устройство, производящее декодирование.

Под *ключом* в криптографии понимают сменный элемент шифра, который применен для шифрования конкретного сообщения.

Рассмотрим несколько примеров кодирования:

- перевод письменного текста с одного естественного языка на другой (в этом случае первичный алфавит – алфавит языка, на котором написан текст, вторичный алфавит – алфавит языка перевода);
- ввод и сохранение текста на компьютере (первичный алфавит – алфавит используемого естественного языка, вторичный алфавит – набор двоичных цифр {0; 1});
- флажковый семафор (первичный алфавит – алфавит используемого естественного языка, вторичный алфавит – совокупность различных положений рук (флажков) по отношению к туловищу сигнальщика).

Операции кодирования и декодирования называются *обратимыми*, если их последовательное применение не приводит к потере информации. *Примером* обратимого кодирования является телеграф. К обратимому кодированию относят также сжатие информации без потерь, помехоустойчивое кодирование. Необратимое кодирование происходит при переводе с одного естественного языка на другой, при сжатии с потерями, при аналого-цифровом преобразовании. Различают принципиально необратимое, обратимое с помощью дополнительной информации и безусловно обратимое кодирование.

Принципиально необратимое кодирование (хэширование) используется, например, в операционных системах для хранения паролей. При первоначальном вводе пароль преобразуется с помощью так называемых односторонних функций (хэш-функций), подобранных таким образом, чтобы из полученной на их выходе строки принципиально нельзя было получить первоначальное значение пароля. При дальнейшем использовании пароль каждый раз преобразуется такой же функцией и сравнивается с первоначальным хэшем.

Чаще используется *кодирование, обратимое с помощью дополнительной информации* (ключа шифрования). Входная информация преобразуется с помощью пароля таким образом, чтобы обратное преобразование также требовало знания пароля. Именно этот способ обычно используется при шифровании архивов.

Безусловно обратимое кодирование – это кодирование, при котором обратное преобразование не требует знания какой-то дополнительной информа-

ции. В подавляющем большинстве случаев для хранения паролей к внешним ресурсам используется именно этот способ.

Условие обратимости кодирования в формализованном (математическом) виде

Пусть I_1 – это количество информации в исходном сообщении, состоящем из символов первичного алфавита A, I_2 – это количество информации в том же сообщении, записанном с помощью символов вторичного алфавита B, т.е. количество информации в сообщении после кодирования; тогда условие обратимости кодирования запишется в следующем виде:

$$I_1 \le I_2.$$

На практике при построении обратимого кода необходимо придерживаться следующих правил:

- с разными символами первичного алфавита А должны быть сопоставлены разные кодовые комбинации;
- никакая кодовая комбинация не должна составлять начальную часть какой-нибудь другой кодовой комбинации.

Кодирование информации для сжатия данных. Коды Шеннона-Фано, Хаффмана, Хэмминга

Сжатие данных (англ. data compression) – алгоритмическое преобразование данных, производимое с целью уменьшения занимаемого ими объёма. Применяется для более рационального использования устройств хранения и передачи данных. Кодирование Шеннона-Фано является одним из самых первых алгоритмов сжатия, который впервые сформулировали независимо друг от друга американские учёные Клод Шеннон и Роберт Фано. Данный метод сжатия имеет большое сходство с кодированием Хаффмана, которое появилось на несколько лет позже. Главная идея этого метода – заменить часто встречающиеся символы более короткими кодами, а редко встречающиеся последовательности более длинными кодами. Таким образом, алгоритм основывается на кодах переменной длины. Для того чтобы декомпрессор впоследствии смог раскодировать сжатую последовательность, коды Шеннона-Фано должны обладать уникальностью, т. е., несмотря на их переменную длину, каждый код уникально определяет один закодированный символ и не является префиксом любого другого кода.

При кодировании данных также существует такое понятие, как *информационная энтропия* источника, которая вычисляется по формуле

$$H = -\sum_{i=1}^{n} P_i \cdot \log(P_i) \left[\frac{бит}{символ}\right].$$

Цена кодирования (средняя длина кодового слова l) является критерием степени оптимальности кодирования. Рассчитать среднее количество элементарных символов на один символ первичного алфавита можно с помощью соотношения

$$l_{ср} = \sum_{i=1}^{n} P_i \cdot l_i \left[\frac{бит}{символ} \right].$$

Если энтропия источника сообщений не равна максимальной энтропии для алфавита с данным количеством качественных признаков (имеются в виду качественные признаки алфавита, при помощи которых составляются сообщения), то это прежде всего означает, что сообщения данного источника могли бы нести большее количество информации.

Абсолютная недогруженность на символ сообщений такого источника равна

$$\Delta D = l_{ср} - H \left[\frac{бит}{символ} \right].$$

Избыточность – термин из теории информации, означающий превышение количества информации, используемой для передачи или хранения сообщения, над его информационной энтропией. Рассчитать избыточность, которая показывает, сколько процентов букв может быть удалено из текста без утраты смысла сообщения, можно по формуле

$$D = \frac{\Delta D}{l_{ср}} \left[\frac{бит}{символ} \right].$$

Для уменьшения избыточности применяется сжатие данных без потерь, в то же время контрольная сумма используется для внесения дополнительной избыточности в поток, что позволяет исправлять ошибки при передаче информации по каналам, вносящим искажения (спутниковая трансляция, беспроводная передача и т. д.).

Алгоритм кодирования Шеннона-Фано. Состоит из следующих основных этапов:

- символы первичного алфавита m_1 выписывают по убыванию вероятностей;
- символы полученного алфавита делят на две части, суммарные вероятности символов которых максимально близки друг другу;
- в префиксном коде первой части алфавита присваивается двоичная цифра «0», второй части – «1»;
- полученные части рекурсивно делятся и их частям назначаются соответствующие двоичные цифры в префиксном коде;
- если размер подалфавита становится равен нулю или единице, то дальнейшего удлинения префиксного кода для соответствующих ему симво-

лов первичного алфавита не происходит, таким образом, алгоритм присваивает различным символам префиксные коды разной длины.

На шаге деления алфавита существует неоднозначность, так как разность суммарных вероятностей $p_0 - p_1$ может быть одинакова для двух вариантов разделения. Код Шеннона-Фано можно реализовать с помощью таблицы или с помощью дерева. При построении кода Шеннона-Фано разбиение множества элементов может быть произведено несколькими способами. Выбор разбиения на уровне n может ухудшить варианты разбиения на следующем уровне $(n + 1)$ и привести к неоптимальности кода в целом. Другими словами, оптимальное поведение на каждом шаге пути ещё не гарантирует оптимальности всей совокупности действий. Поэтому код Шеннона-Фано не является оптимальным в общем смысле, хотя и дает оптимальные результаты при некоторых распределениях вероятностей.

Таблица 4. Кодирование по методу Шеннона-Фано

Символ алфавита a_i	**Вероятность** $P(a_i)$	**Шаги алгоритма**					**Количество элементарных символов** l_i	**Кодовое слово**
		1	2	3	4	5		
a_1	0,30	0 (0,55)	0 (0,3)				2	00
a_2	0,25	0	1 (0,25)				2	01
a_3	0,15	1	0 (0,25)	0 (0,15)			3	100
a_4	0,1	1	0	1 (0,1)			3	101
a_5	0,1	1 (0,45)	1 (0,2)	0 (0,1)			3	110
a_6	0,05	1	1	1 (0,1)	0 (0,05)		4	1110
a_7	0,04	1	1	1	1 (0,05)	0 (0,04)	5	11110
a_8	0,01	1	1	1	1	1 (0,01)	5	11111

Алгоритм кодирования Хаффмана. Кодирование Шеннона-Фано является достаточно старым методом сжатия и на сегодняшний день не представляет особого практического интереса. В большинстве случаев длина сжатой последовательности по данному методу равна длине сжатой последовательности с использованием кодирования Хаффмана. Но на некоторых последовательностях всё же формируются не оптимальные коды Шеннона-Фано, поэтому сжа-

тие методом Хаффмана принято считать более эффективным. Алгоритм кодирования Хаффмана состоит из следующих основных этапов:

– построение оптимального кодового дерева;
– построение отображения кода-символа на основе построенного дерева.

Кодовое дерево строится следующим образом:

– символы входного алфавита образуют список свободных узлов, каждый из которых имеет вес (вероятность или количество вхождений символа в сжимаемое сообщение);
– выбираются два свободных узла дерева с наименьшими весами, и создается их родитель с весом, равным их суммарному весу;
– родитель добавляется в список свободных узлов, а два его потомка удаляются из этого списка;
– одной дуге, выходящей из родителя, ставится в соответствие бит 1, другой − бит 0;
– шаги, начиная со второго, повторяются до тех пор, пока в списке свободных узлов не останется только один свободный узел, который будет считаться корнем дерева.

Таблица 5. Кодирование по методу Хаффмана

Символ алфавита a_i	**Вероятность** $P(a_i)$	**Кодовое дерево**	**Количество элементарных символов** l_i	**Кодовое слово**
a_1	0,30		2	00
a_2	0,25		2	01
a_3	0,15		3	100
a_4	0,1		3	101
a_5	0,1		3	110
a_6	0,05		4	1110
a_7	0,04		5	11110
a_8	0,01		5	11111

Алгоритм кодирования Хэмминга. Коды, способные корректировать ошибки (в каналах связи в цифровых вычислительных машинах и т. п.) при обработке сигналов, были предложены Хэммингом, заложившем основу теории в данной области. Они продемонстрировали инженерам практическую возможность достижения тех пределов, на которую указывали законы теории инфор-

мации. Эти коды нашли практическое применение при создании компьютерных систем. Код Хэмминга − это алгоритм самоконтролирующегося и самокорректирующегося кода, который позволяет закодировать какое-либо информационное сообщение определённым образом и после передачи (например, по сети) определить, не появилась ли в нем ошибка и, при возможности, восстановить его.

Код Хэмминга состоит из двух этапов:

– кодируется исходное сообщение, в него в определённых местах вставляются контрольные биты (вычисленные особым образом);
– в закодированном сообщении заново вычисляются контрольные биты (по тому же алгоритму); если все вновь вычисленные контрольные биты совпадают с полученными, то сообщение получено без ошибок, противном случае выводится сообщение об ошибке и при возможности ошибка исправляется.

Сообщение кодируется следующим образом:

– каждый символ текста заменяется его двоичным представлением (комбинацией 0 и 1), причем один символ сообщения соответствует восьми двоичным символам (если символов меньше, добавляются впереди нули);
– далее все действия совершаются по частям (например, по две буквы, т. е. по 16 битов), и каждая часть кодируется отдельно;
– на 1-2-4-8-16… (2^n) места в коде ставятся контрольные биты, пока нули (можно выделить их другим цветом, чтоб не спутать с битами текста);
– далее вычисляется значение контрольных битов по правилу: значение каждого контрольного бита зависит от значений информационных битов, но не от всех, а только от тех, которые этот контрольный бит контролирует (контрольный бит с номером N контролирует все последующие N бит через каждые N бит, начиная с позиции N); для каждого контрольного бита определяем количество единиц среди контролируемых им битов, получаем некоторое целое число, если оно чётное, то контрольный бит – ноль, в противном случае – единица. Можно, конечно, и наоборот: если число чётное, то ставим единицу, в противном случае – 0. Главное, чтобы в «кодирующей» и «декодирующей» частях алгоритм был одинаков (обычно применяют первый вариант).

Пример. Закодировать слово no.

1) no = 6E 6F (ACSII код) = 01101110 01101111;
2) вставляем контрольные биты (пока 0) на 1, 2, 4, 8, 16 места
3) 000011001110011001111;
4) теперь необходимо вычислить значения контрольных бит:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	
0	1	0	0	1	1	0	1	1	1	1	0	0	1	1	0	0	1	1	1	1	
X		X		X		X		X		X		X		X		X		X		X	1
	X	X			X	X			X	X			X	X			X	X			2
			X	X	X	X					X	X	X	X					X	X	4
							X	X	X	X	X	X	X	X							8
															X	X	X	X	X	X	16

5) итоговый код слова no 010011011110011001111.

Декодирование и коррекция ошибок происходят подобным образом. Алгоритм заключается в том, чтобы опять вычислить контрольные биты и сравнить их с контрольными битами, которые были переданы. Сумма неправильных контрольных бит и будет значением бита, в котором допущена ошибка. Вернемся к нашему примеру. Пусть сообщение передалось в следующем виде 010010011111010001111 с ошибкой 12 бита. Вычисляем контрольные биты для полученного сообщения:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	
0	1	0	1	1	1	0	0	1	1	1	1	0	1	1	0	0	1	1	1	1	
X		X		X		X		X		X		X		X		X		X		X	1
	X	X			X	X			X	X			X	X			X	X			2
			X	X	X	X					X	X	X	X					X	X	4
							X	X	X	X	X	X	X	X							8
															X	X	X	X	X	X	16

Четвертый и восьмой контрольный бит неправильные, значит, ошибка в 4+8=12-м бите.

Лекция 1.4

Арифметические и логические основы ЭВМ

Арифметические основы ЭВМ заключаются в правилах работы с числами в двоичной системе счисления, так как, уже было сказано, вся информация в компьютере представляется именно в этой системе счисления. Прежде чем перейти к правилам арифметических действий над двоичными числами, следует сказать о том, что двоичное число с фиксированной запятой в памяти компьютера может иметь представление в прямом, обратном и дополнительном коде.

Прямой код двоичного числа образуется из абсолютного значения этого числа и кода знака (нуль или единица) перед его старшим числовым разрядом.

Пример:

$A_{10} = +10 \Rightarrow A_2 = +1010 \Rightarrow [A_2]_П = 0000\ 1010$;

$B_{10} = -15 \Rightarrow B_2 = -1111 \Rightarrow [B_2]_П = 1000\ 1111$.

В прямом коде число 0 имеет два представления: +0 = 0000 0000 и –0 = 1000 0000. Поэтому прямой код обычно используется для представления положительных чисел. Для представления отрицательных чисел применяется обратный код числа.

Обратный код двоичного числа образуется по следующему правилу. Обратный код положительных чисел совпадает с их прямым кодом. В обратном коде отрицательного числа знаковый разряд равен единице, а разряды числа заменяются на инверсные, т.е. нули заменяются единицами, а единицы – нулями.

Пример:

$A_{10} = +5 \Rightarrow A_2 = +101 \Rightarrow [A_2]_П = [A_2]_{ОК} = 0000\ 0101$;

$B_{10} = -5 \Rightarrow B_2 = -0000\ 0101 \Rightarrow [B_2]_{ОК} = 1111\ 1010$.

В обратном коде число 0 имеет также два представления: +0 = 0000 0000 и –0 = 1111 1111.

Есть случаи, когда два разных числа в прямом и дополнительном коде имеют одинаковое представление, например, прямой код числа –127 = 1111 1111 и обратный код числа –0 = 1111 1111. Кроме этого, для чисел в прямом и обратном коде часто возникают трудности с арифметическими операциями, в частности с вычитанием, когда результат вычитания должен получиться отрицательным. Для решения этих проблем предлагается использовать дополнительный код числа.

Дополнительный код положительного числа равен прямому коду. Дополнительный код отрицательного числа можно получить двумя способами.

1. Получаем обратный код отрицательного числа и прибавляем к нему 1:

$$-10 = 1000\ 1010 \Rightarrow 1111\ 0101 \Rightarrow 1111\ 0101 + 1 = 1111\ 0110.$$

2. Вычесть число из 0:

$$-10 = 0 - 10 = 0 - 0000\ 1010 = 1111\ 0110.$$

Используя дополнительный код, операцию вычитания приводят к операции сложения двух чисел в прямом (положительного числа) и дополнительном (отрицательного числа) коде.

Правила арифметических действий в двоичной системе счисления такие же, как и в десятичной, с разницей в том, что отрицательные числа рассматриваются в обратном или дополнительном коде.

Правила выполнения арифметических действий над двоичными числами задаются таблицами сложения, вычитания и умножения.

Таблица 6. Сложение, вычитание, умножение двух чисел в двоичной СС

1 число	2 числа	Сложение	Вычитание	Умножение
0	0	0	0	0
0	1	1	10 – 1 = 1	0
1	0	1	1	0
1	1	10	0	1

Рассмотрим все арифметические операции на примере.

Пример. Произвести все арифметические операции с числами 6 и 3, рассмотреть два вычитания: 6 – 3 и 3 – 6.

1) Сложение: 6 + 3.

$6_{10} = 0000\ 0110_2$, $3_{10} = 0000\ 0011_2$;

 0000 0110
+0000 0011
 0000 1001

$0000\ 1001_2 = 9_{10}$.

2) Вычитание: 6 – 3.

Сначала произведем вычитание в прямом коде:

 0000 0110
– 0000 0011
 0000 0011

Числа довольно простые, проблем не возникло.

Теперь произведем вычитание в дополнительном коде 6 – 3 = 6 + (–3);

$-3_{10} = 1111\ 1100_2$ в обратном коде и +1 = 1111 1101 в дополнительном коде:

 0000 0110
+1111 1101
1 0000 0011

Единица, которая ушла за пределы старшего разряда, не считается. Получили ответ в виде положительного числа в прямом коде.

3) Вычитание: 3 – 6.

Сначала произведем вычитание в прямом коде:

 0000 0011
– 0000 0110
 1111 1101

В прямом коде вычесть из меньшего числа большее не получается, обращаемся к вычитанию в дополнительном коде 3 – 6 = 3 + (–6);

$-6_{10} = 1111\ 1001_2$ в обратном коде и +1 = 1111 1010 в дополнительном коде:

0000 0011
+1111 1010
1111 1101

В знаковом разряде 1, следовательно, ответ – отрицательное число, представленное в дополнительном коде. Для получения окончательного ответа необходимо перевести число в прямой код. Отнимаем 1:

1111 1101
– 0000 0001
1111 1100

Получили обратный код, инвертируя который, получаем прямой код отрицательного числа:

$1111\ 1100 = -0000\ 0011_2 = -3_{10}$.

4) Умножение 6 х 3.

0000 0110
x 0000 0011
0000 0110
00000 110_
00001 0010

$10010_2 = 2^4 + 2^1 = 18_{10}$.

5) Деление 6 / 3.

110 |11
11   10
00

$10_2 = 2_{10}$.

Логические основы ЭВМ строятся на алгебре логики (булевой алгебры). Алгебра логики оперирует с логическими переменными (высказываниями), которые могут принимать только два значения «истина» или «ложь», обозначаемые соответственно 1 и 0. Логические переменные могут быть *простыми* (одно высказывание; например, 5 – это натуральное число) и *сложными* (состоят из двух или более высказываний; например, 9 делится на три и 5 – четное число). Так как любая простая переменная может принимать значения «истина» или «ложь», то и сложные переменные также принимают эти значения, которые зависят от значения простых переменных, входящих в состав сложной переменной. Между простыми переменными могут существовать следующие основные отношения: не, и, или, тогда и только тогда и др.

Основные *логические операции*: инверсия (логическое отрицание), конъюнкция (логическое умножение), дизъюнкция (логическое сложение) и исключающее или. Рассмотрим таблицы истинности данных операций.

Таблица 7. Инверсия логической переменной

x	$\bar{x}$
0	1
1	0

Таблица 8. Таблица истинности основных логических операций

x	y	$x \cup y$	$x \cap y$	$x \oplus y$
0	0	0	0	0
0	1	1	0	1
1	0	1	0	1
1	1	1	1	0

Если логическая переменная состоит из более, чем двух простых переменных, то для установления порядка выполнения логических операций ставятся скобки. Если скобок нет, то первой выполняется операция отрицания, второй - операция конъюнкции, третьей – операция дизъюнкции.

В ЭВМ используются различные устройства, работу которых прекрасно описывает алгебра логики. К таким устройствам относятся переключатели, вентили, триггеры, сумматоры.

Переключательные схемы. В ЭВМ применяются электрические схемы, состоящие из множества переключателей. Переключатель может находиться только в двух состояниях: замкнутом и разомкнутом. В первом случае – ток проходит, во втором – нет. Описывать работу таких схем очень удобно с помощью алгебры логики. В зависимости от положения переключателей можно получить или не получить сигналы на выходах.

Вентиль представляет собой логический элемент, который принимает одни двоичные значения и выдает другие в зависимости от своей реализации. Так, например, есть вентили, реализующие логическое умножение (конъюнкцию), сложение (дизъюнкцию) и отрицание.

Триггеры и *сумматоры* – это относительно сложные устройства, состоящие из более простых элементов – вентилей. Триггер способен хранить один двоичный разряд за счет того, что может находиться в двух устойчивых состояниях. В основном триггеры используются в регистрах процессора. *Сумматоры* широко применяются в арифметико-логических устройствах (АЛУ) процессора и выполняют суммирование двоичных разрядов.

Устройство персонального компьютера

Распространение персональных компьютеров к концу 70-х гг. привело к некоторому снижению спроса на большие ЭВМ и мини-ЭВМ. Фирма IBM решила попробовать свои силы на рынке персональных компьютеров и сделала компьютер не единым неразъемным устройством, а обеспечила его сборку из независимо изготовленных частей по принципу открытой архитектуры.

На основной электронной плате (системной или материнской) размещены только блоки, обрабатывающие информацию (вычисления): процессор, возможно, математический сопроцессор, контроллеры, микросхемы оперативной памяти. Схемы, управляющие всеми остальными устройствами компьютера (монитором, дисками, принтером и т. д.), реализованы на отдельных платах, которые вставляются в стандартные разъемы (слоты) на системной плате. К этим электронным схемам подводится электропитание из единого блока, а для удобства и надежности все это заключается в общий металлический или пластиковый корпус – системный блок. Компьютер состоит из разрозненных частей. Для того чтобы он работал как единый механизм, необходимо осуществлять обмен данными между различными устройствами, за это отвечает системная (магистральная) шина. К ней через контроллеры подключены внешние устройства, которые обмениваются данными с оперативной памятью. Обмен данными между устройствами ЭВМ обусловлен ограничением функций, выполняемых этими устройствами, и должен быть запрограммирован. Выполняемая программа хранится в оперативной памяти компьютера и через системную шину передает в процессор команды на выполнение определенных операций. Процессор на их основе формирует свои команды управления, которые по системной шине поступают на соответствующие устройства. Для выполнения операций обработки данных процессор передает в оперативную память адреса необходимых данных и получает их. Результаты обработки направляются в оперативную память. Данные из оперативной памяти могут быть переданы на хранение во внешние запоминающие устройства для отображения на дисплее, вывода на печать, передачи в вычислительную сеть. Важными техническими характеристиками, влияющими на производительность компьютера, являются показатели частоты процессора, разрядность и машинное слово. Количество разрядов, которое может быть воспринято, передано или получено за одно обращение к процессору, называется его *разрядностью*. Количество информации, записываемое или извлекаемое из памяти за одно обращение, называется *машинным словом*.

В состав современного ПК (настольного) входят:

1. Системный блок:

Материнская плата с адаптерами HDD, FDD, CD/DVD-ROM, шины, порты, микросхема, BIOS, таймер, центральный процессор, линейки ОЗУ, видеокарта (может быть интегрирована в материнскую плату), аудиокарта (может быть интегрирована в материнскую плату), сетевая карта (может быть интегрирована в материнскую плату).

Накопители на жестких и гибких магнитных дисках.

Приводы CD- и DVD-ROM.

Блок питания.

Корпус.

2. Монитор.

3. Клавиатура.

4. Манипулятор «мышь».

5. Звуковые колонки.

6. Принтер.

7. Сканер.

8. Модем или адаптер ADSL.

Ну, и, конечно же, компьютер нельзя представить без программного обеспечения. Как архитектура IBM PC стала стандартом для аппаратной части ПК, так и продукция фирмы MicroSoft (Билл Гейтс) стала эталоном для программ. Особенно популярны ее операционные системы Windows и офисные приложения MS-Office.

Микропроцессор – небольшая электронная схема, выполняющая все вычисления и обработку информации. В компьютерах типа IBM PC используется микропроцессоры фирмы Intel. Микропроцессоры отличаются друг от друга двумя характеристиками: типом (моделью) и тактовой частотой. Наиболее распространены модели Intel-8088, 80286, 80386, 80486 и Pentium, они приведены в порядке возрастания производительности и цены. Одинаковые модели микропроцессоров могут иметь разную тактовую частоту. Тактовая частота указывает, сколько элементарных операций (тактов) микропроцессор выполняет в одну секунду. Тактовая частота измеряется в мегагерцах (Мгц).

Оперативная память. Из оперативной памяти процессор берет программы и исходные данные для обработки, в нее он записывает полученные результаты. Оперативная память работает очень быстро, содержащиеся в ней данные сохраняются, пока компьютер включен, при выключении компьютера содержимое оперативной памяти стирается.

Контроллеры и *шина*. Чтобы компьютер мог работать, в его оперативной памяти должны находиться программа (последовательность команд, записанная на языке понятном процессору) и данные. А попадают они туда из различных

устройств компьютера: клавиатуры, дисководов для магнитных дисков и т.д. Обычно эти устройства называют внешними. Таким образом, для работы компьютера необходим обмен информацией между оперативной памятью и внешними устройствами. Такой обмен называется вводом-выводом. Но этот обмен не происходит непосредственно: между любым внешним устройством и оперативной памятью в компьютере имеются два промежуточных звена:

Контроллер или *адаптер* – электронная схема, которая управляет работой какого-либо внешнего устройства.

Шина – системная магистраль передачи данных.

Дисководы – устройства для записи, считывания и длительного хранения информации на гибких магнитных дисках (дискетах). Объем информации, который может быть размещен на дискете, различен для типов дискет. Самые распространенные на сегодня дискеты – 1.44 Мбайта. Популярностью пользуются CD-диски, DVD-диски, среди которых есть такие, что могут применяться для одноразовой записи данных и последующего их многократного использования, и такие, которые можно многократно очищать и записывать новые данные.

Винчестеры – устройства для записи, считывания и длительного хранения информации на жестких магнитных дисках. Необходимый объем винчестера зависит от потребностей и материальных возможностей пользователя, на сегодняшний день – около 1 Тбайт и выше. Существуют внешние жесткие диски, карты флеш-памяти, которые используются для длительного хранения данных и подключаются к компьютеру с помощью USB.

Принтеры – печатающие устройства, предназначенные для вывода информации на бумагу. Существуют несколько тысяч моделей принтеров. Как правило, применяются принтеры следующих типов: матричные, струйные и лазерные.

Мониторы – устройства, предназначенные для вывода на экран текстовой и графической информации.

Мышь – манипулятор для ввода информации в компьютер.

Модем – устройство для обмена информацией между компьютерами через телефонные, оптоволоконные и другие сети.

Сканер – устройство для считывания графической и текстовой информации в компьютер с бумажных носителей информации.

Плоттер – устройство для вывода чертежей на бумагу.

Принципы работы ЭВМ

В основу архитектуры современных персональных компьютеров положен *магистрально-модульный* принцип. *Модульный* принцип позволяет потребителю самому комплектовать нужную ему конфигурацию компьютера и производить при необходимости ее модернизацию. Модульная организация компьютера опирается на *магистральный* (шинный) принцип обмена информацией между устройствами.

Магистраль включает в себя три многоразрядные шины: шину данных, шину адреса и шину управления. Шины представляют собой многопроводные линии.

Шина данных обеспечивает передачу данных между различными устройствами. Например, считанные из оперативной памяти данные могут быть переданы процессору для обработки, а затем полученные данные могут быть отправлены обратно в оперативную память для хранения. Таким образом, информация по шине данных может передаваться от устройства к устройству в любом направлении. Разрядность шины данных определяется разрядностью процессора, т.е. количеством двоичных разрядов, которые процессор обрабатывает за один такт. Разрядность процессоров постоянно увеличивается по мере развития компьютерной техники.

Шина адреса. Выбор устройства или ячейки памяти, куда пересылаются или откуда считываются данные по шине данных, производит процессор. Каждое устройство или ячейка оперативной памяти имеет свой адрес. Адрес передается по адресной шине, причем сигналы по ней передаются в одном направлении от процессора к оперативной памяти и устройствам (однонаправленная шина). Разрядность шины адреса определяет адресное пространство процессора, т.е. количество ячеек оперативной памяти, которые могут иметь уникальные адреса. Количество адресуемых ячеек памяти можно рассчитать по формуле

$$N = 2^I,$$

где I – разрядность шины адреса.

Разрядность шины адреса постоянно увеличивалась и в современных персональных компьютерах составляет 64 бита. Таким образом, максимально возможное количество адресуемых ячеек памяти равно $N = 2^{64}$.

Шина управления. По шине управления передаются сигналы, определяющие характер обмена информацией по магистрали. Сигналы управления определяют, какую операцию – считывание или запись информации из памяти – нужно производить, синхронизируют обмен информацией между устройствами и т.д.

Архитектура фон Неймана

В основу построения подавляющего большинства компьютеров положены следующие общие принципы, сформулированные в 1945 г. американским ученым Джоном фон Нейманом.

В 40-х гг. XX в. американец венгерского происхождения Джон (Янош) фон Нейман (1903-1957) включился в работу по созданию ЭВМ для управления береговой ПВО. Разрабатывался «ЭНИАК» – электронный численный интегратор и автоматический вычислитель. Но эта машина имела принципиальный недостаток: в ней отсутствовало устройство для запоминания и хранения команд.

В 1945 г. Джон фон Нейман выступил с докладом, в котором были сформулированы основные принципы организации нового вычислительного устройства, получившие название «*архитектура фон Неймана*». В нее входят:

АЛУ – арифметико-логическое устройство для выполнения арифметических и логических операций;

ОП – оперативная память, устройство для хранения кодов выполняющейся в данный момент программы;

ВУ – внешние устройства, или периферия. Обычно их делят на два класса: внешнюю память (накопитель на гибких магнитных дисках, накопитель на жестких магнитных дисках, CD-диски, магнитооптические диски) и устройства ввода/вывода информации (устройства ввода: клавиатура, мышь, микрофон, сканер; устройства вывода: дисплей, принтер, акустические колонки, плоттер);

УУ – управляющее устройство, которое организует работу компьютера следующим образом:

а) помещает в ОП коды программы из ВУ;

б) считывает из ячейки ОП и организует выполнение первой команды программы;

в) определяет очередную команду и организует ее выполнение;

г) постоянно синхронизирует работу устройств, имеющих различную скорость выполнения операций, путем приостановки выполнения программы.

В 1946 г. фон Нейман начинает разработку новой машины. В 1949 г. была построена электронная машина по обработке дискретных переменных «ЭДВАК», которая впоследствии была признана первым компьютером.

Норберт Винер (1894-1964), работая вместе с Джоном фон Нейманом, обратил внимание на то, что процессы, управляющие сложной электронной системой, аналогичны процессам нейрофизиологии, изучающей целенаправленную деятельность живых существ. Сохранение работоспособности таких систем достигается за счет обратной связи, она позволяет отслеживать и корректировать уже начатое, но еще не законченное до конца действие. Существова-

ние обратной связи позволяет рассматривать сложные системы различной природы – физической, социальной, биологической – с единой точки зрения. Это и есть основы кибернетики. В 1948 г. вышла в свет книга Н. Винера «Кибернетика, или Управление и связь в живом мире и машинах». Термин «кибернетика» в переводе с древнегреческого обозначает искусство управления кораблем.

Принципы, сформулированные в 1945 г. американским ученым Джоном фон Нейманом:

1. *Принцип программного управления*. Программа состоит из набора команд, выполняющихся процессором автоматически в определенной последовательности. Выборка программы из памяти осуществляется с помощью счетчика команд. Этот регистр процессора последовательно увеличивает хранимый в нем адрес очередной команды на длину команды. А так как команды программы расположены в памяти друг за другом, то тем самым организуется выборка цепочки команд из последовательно расположенных ячеек памяти. Если нужно после выполнения команды перейти не к следующей, а к какой-то другой, используются команды условного или безусловного перехода, которые заносят в счетчик команд номер ячейки памяти, содержащей следующую команду. Выборка команд из памяти прекращается после достижения и выполнения команды «стоп». Таким образом, процессор исполняет программу автоматически, без вмешательства человека.

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 организуется выполнение циклов и подпрограмм). Более того, команды одной программы могут быть получены как результаты исполнения другой программы. На этом принципе основаны методы трансляции – перевода текста программы с языка программирования высокого уровня на язык конкретной машины.

3. *Принцип адресности*. Структурно основная память состоит из перенумерованных ячеек. Процессору в произвольный момент времени доступна любая ячейка. Отсюда следует возможность давать имена областям памяти так, чтобы к запомненным в них значениям можно было впоследствии обращаться или менять их в процессе выполнения программ с использованием присвоенных имен. Компьютеры, построенные на перечисленных принципах, относятся к типу фон-неймановских. Но существуют компьютеры, принципиально отличающиеся от фон-неймановских. Они, например, могут не соблюдать принцип

программного управления, т. е. работать без счетчика команд, указывающего текущую выполняемую команду программы. Для обращения к какой-либо переменной, хранящейся в памяти, этим компьютерам необязательно давать ей имя. Такие компьютеры называются не фон-неймановскими.

Структура памяти компьютера

Схематически структура памяти компьютера изображена на рис. 3. *Оперативная* (основная) память (ОЗУ) используется для оперативного обмена информацией (данными и командами) между процессором, внешней памятью и периферийными устройствами. ОЗУ является памятью с произвольным доступом, т. е. обращаться к ячейкам памяти можно любом порядке. ОЗУ – динамическая память, т. е. данные хранятся в постоянном обновлении. ОЗУ энергозависимо.

Кэш-память – сверхоперативная память (СОЗУ), является буфером между ОЗУ и процессором и другими абонентами системной шины. Быстродействие этой памяти близко к скорости ядра процессора. Не является хранилищем данных и не адресуема клиентами подсистемы памяти, хранит копии блоков данных, областей памяти, которые чаще участвуют в обмене информацией.

Постоянная память используется для энергонезависимого хранения системной информации, таблиц знакогенераторов и т. д. Обычно эта память только считывается, но можно и изменить ее. В качестве постоянной памяти используют флэш-память. Быстродействие намного меньше оперативной.

Полупостоянная используется обычно для хранения настроек компьютера, включающих системные часы. Работа этой памяти поддерживается небольшой внутренней батарейкой.

Дополнительная память – внешняя память компьютера, энергонезависимая.

Рис. 3. Организация памяти в ЭВМ

Логическая организация основной памяти. Каждая ячейка имеет свой номер, называемый исполнительным адресом (АИ) или абсолютным адресом (АА). Для записи в память или считывания из памяти требуется указать исполнительный адрес ячейки памяти. Время записи и считывания для всех ячеек одинаково и не зависит от их исполнительного адреса.

Ячейка имеет фиксированный размер. Стандартным размером ячейки является байт – 8 бит. Следовательно, байт – наименьший объем информации, который может быть адресован в основной памяти. Во многих ЭВМ процессор способен одновременно получать из основной памяти, обрабатывать и записывать несколько байтов. Поле из нескольких байтов, одновременно обрабатываемых процессором, называют *машинным словом*. Адресом слова является адрес младшего байта.

Устройство и работа процессора

Процессор – один из основных компонентов компьютера. Состоит из следующих внутренних узлов:

- устройство управления, предназначенное для дешифрования и исполнения команд;
- рабочие регистры, необходимые для адресации памяти и выполнения вычислительных операций;
- арифметико-логическое устройство, выполняющее логические и арифметические операции;
- управление вводом-выводом данных в процессор или из процессора.

Процессор работает с командами. В команде предписывается, какое действие должен выполнить процессор и с какими данными. А любая программа, которая выполняется процессором, состоит из множества различных команд.

Как уже говорилось, информация в компьютере хранится в виде двоичного кода, составленного из последовательностей 0 и 1. Эти последовательности имеют разрядность, кратную 8, т. е. 8-разрядные, 16-разрядные, 32-разрядные и т. д. 0 или 1 в такой последовательности носит название бит. Соответственно:

8 бит = 1 байту;

16 бит = 1 машинному слову;

32 бита = двойному машинному слову.

Но в компьютере объём информации определяется не в машинных словах, а в следующих величинах:

1024 байта = 1 килобайту (Кб); 1024 Кб = 1 мегабайту (Мб); 1024 Мб = 1 гигабайту (ГБ); 1024 Гб = 1 терабайту (Тб).

В изучении архитектуры, работы процессора будем опираться на один из самых первых массовых процессоров фирмы Intel 8086. Все последующие поколения процессоров, включая и самые последние, используют базовые принципы архитектуры процессора 8086 и его семейства.

Команда для процессоров Intel в общем случае имеет следующий формат:

Код операции	Операнд	…	Операнд

Здесь одна группа битов образует поле код операции. Код операции показывает, что процессор должен делать. А поля, отведенные под операнды, предоставляют необходимые данные для осуществления операции процессором.

Архитектура процессора. Процессор работает с оперативной памятью, так как в ней хранятся данные, необходимые ему для работы, и в оперативную память процессор помещает результаты своих вычислений перед окончательным сохранением в долговременной памяти. Взаимодействие процессора с памятью происходит посредством шин адреса, данных и управления.

На шину адреса помещается адрес ячейки памяти с данными, необходимыми процессору.

На шину данных помещаются данные из процессора, которые необходимо записать в память или данные из памяти, которые необходимы процессору для вычислений.

На шину управления поступают сигналы из процессора, которые указывают памяти или другим устройствам, какое устройство процессор выбрал и какую операцию будет производить: чтение или запись памяти, или других устройств.

Работа процессора происходит по такому алгоритму. Программный счётчик выдаёт адрес команды на шину адреса. Память помещает команду, находящуюся по этому адресу, на шину данных. Процессор вводит команду в свой регистр команды. Там команда дешифруется, определяются код операции и длина команды в байтах. Программный счётчик адресует следующую команду, находящуюся по адресу, на длину команды от предыдущей. Сама команда после её обработки регистром команды выполняется остальными устройствами процессора, назначение которых указано выше. Когда выполнение команды завершается, содержимое программного счётчика помещается на шину адреса и цикл повторяется.

Все команды процессор выполняет последовательно, команда за командой. Однако часто бывает так, что процессор встречает команду перехода. Это значит, что по каким-то причинам процессор должен прервать выполнение последовательности команд и «перескочить» на другую последовательность. Ре-

гистры состояния процессора и указатель стека участвуют в таких операциях, указывая состояние процессора и запоминая последнее. Причинами команды перехода может оказаться логический выбор дальнейшего выполнения программы из-за каких-нибудь условий.

Процессор может иметь свою собственную память, используемую для кратковременного хранения информации. Она состоит из специальных электронных узлов – регистров. Разрядность регистров обычно совпадает с разрядностью машинного слова. Регистры доступны программе и могут использоваться ею для хранения промежуточных результатов. Такого рода регистры называются регистрами общего назначения (РОН). Каждый регистр имеет номер, который может указываться в качестве адреса в команде. Адреса регистров не совпадают с адресами основной памяти. По сравнению с основной памятью, объем которой может достигать нескольких Гб, объем регистровой памяти невелик, порядка десятков РОН. Время же считывания и записи информации для РОН на несколько порядков меньше, чем для ячейки основной памяти. Основная память может быть распределена для одновременного размещения в ней нескольких программ. Регистры же выделяются в монопольное использование выполняющейся в данный момент программой.

Команды с четырьмя адресными полями наиболее удобны с точки зрения программиста, но реализовать команды такого формата в аппаратуре процессора при ограниченной длине машинного слова практически невозможно.

Пример. Пусть ЭВМ имеет основную память объемом 1Мб, а процессор способен выполнять до двухсот различных операций, Длина каждого адресного поля составит 20 бит (220 = 1Мб), а поля КОП – 8бит. Длина команды 8+4*20 =88 бит. С другой стороны, машинное слово современных процессоров, как правило, не превышает 64 бита. Современные ЭВМ имеют значительные объемы памяти, измеряемые сотнями мегабайт и гигабайтами, что еще более обостряет проблему.

Сокращение длины команды достигается двумя способами:

- сокращение количества адресных полей.
- сокращение длины адресных полей.

При первом способе некоторые адреса определяются по умолчанию (косвенно) и в команде не указываются. Сокращение количества адресных полей оборачивается увеличением количества команд в программе.

При втором способе в команде указывается не исполнительный адрес, а так называемый логический адрес (АЛ). Логический адрес содержит информацию, на основе которой можно вычислить исполнительный адрес. Сокращение длины адресных полей в общем случае приводит к увеличению времени вы-

полнения программы, т.к. необходимы дополнительные действия по вычислению адресов.

Лекция 1.5
Этапы решения задачи на ЭВМ

Работа по решению любой задачи с использованием компьютера делится на следующие основные этапы:

1. Постановка задачи.
2. Формализация задачи.
3. Построение алгоритма.
4. Составление программы на языке программирования.
5. Отладка и тестирование программы.
6. Проведение расчетов и анализ полученных результатов.

Непосредственно к программированию в этом списке относятся построение алгоритма, составление, отладка и тестирование программы.

На этапе постановки задачи должно быть четко сформулировано, что дано и что требуется найти. Здесь очень важно определить полный набор исходных данных, необходимых для получения решения. На этапе формализации задача переводится на язык математических формул, уравнений, отношений. Далее составляется алгоритм решения задачи с помощью какого-либо специального средства (блок-схемы, псевдокоды). Алгоритмы и их составление будут изучены далее. Профессиональные программисты могут этот этап пропустить. После построения алгоритма следует собственно программирование на определенном языке. Описание языков и приемов программирования также будет представлено далее. После написания программу запускают в тестовом режиме (с набором таких начальных данных, для которых результат известен или можно легко проверить). Если программа работает без ошибок, то переходят к последнему этапу, в противном случае происходит отладка программы. Проводить отладку программы – находить и исправлять в ней ошибки, а также понимать, как происходит выполнение алгоритма на компьютере – помогает *трассировка* программы.

Трассировкой программы называется метод ее пошагового выполнения с отслеживанием значений всех переменных. Обычно трассировку программы проводят вручную (если это не очень сложный алгоритм) и все операторы, условия и изменения значений всех переменных фиксируют в таблице.

Понятие, свойства, исполнители и способы описания алгоритма

Одним из фундаментальных понятий в информатике является понятие алгоритма. Происхождение самого термина «алгоритм» связано с математикой.

Это слово происходит от Algorithmi – латинского написания имени Мухаммеда аль-Хорезми (787-850), выдающегося математика средневекового Востока. В XII в. был выполнен латинский перевод его математического трактата, из которого европейцы узнали о десятичной позиционной системе счисления и правилах арифметики многозначных чисел. Именно эти правила в то время называли алгоритмами. Сложение, вычитание, умножение столбиком, деление уголком многозначных чисел – вот первые алгоритмы в математике. Правила алгебраических преобразований, способы вычислений корней уравнений также можно отнести к математическим алгоритмам.

Алгоритм – это точное и понятное предписание (указание) исполнителю совершить определенную последовательность действий, направленных на достижение указанной цели или решение поставленной задачи, действий, приводящих к решению поставленной задачи.

Исполнителем алгоритма является человек или механическое устройство со строго определенным набором возможный действий (операций). Указание исполнителю выполнить какое-либо действие называется *командой*, совокупность всех команд, которые может выполнить исполнитель, называется *системой команд*.

ЭВМ – исполнитель алгоритмов работы с величинами. В систему команд ЭВМ входят следующие:

- присваивания,
- ввода,
- вывода,
- обращения к вспомогательному алгоритму,
- цикла,
- ветвления.

Алгоритм должен обладать следующими основными свойствами:

- дискретность (алгоритм должен быть записан в виде конечного числа шагов, выполнение каждого шага начинается после завершения выполнения предыдущего);
- понятность (исполнитель должен знать, что ему делать; алгоритм содержит только те команды, которые входят в систему команд исполнителя);
- однозначность или определенность (алгоритм не должен допускать произвольной трактовки шагов со стороны исполнителя, исполнитель должен действовать в строгом соответствии с командами, которые указаны в каждом шаге, у исполнителя не должно возникать необходимости предпринимать действия, не предусмотренные алгоритмом);

- массовость (означает, что один и тот же алгоритм можно использовать для решения многих однотипных задач, отличающихся количеством и/или значениями входных данных);
- результативность (выполнение алгоритма приводит к получению результата);
- конечность (выполнение алгоритма завершается после выполнения конечного числа шагов, при выполнении алгоритма некоторые шаги могут выполняться многократно).

Для записи алгоритмов существуют различные способы. Одни из них ориентированы на исполнителя-человека, другие – на исполнение техническими устройствами, в частности компьютерами, роботами и т.д. То есть алгоритм задается в той форме, которая наиболее понятна исполнителю. Существуют следующие способы представления алгоритмов:

- словесный (описательный);
- формульный;
- графический;
- табличный или схематичный, в виде графа;
- в виде блок-схемы;
- в виде программы.

В информатике для записи алгоритмов широко используются блок-схемы. Блок-схема представляет собой систему связанных геометрических фигур (блоков), каждая из которых обозначает один элементарный шаг алгоритма. Порядок выполнения шагов алгоритма указывается соединительными линиями или стрелками между блоками. Стандартно блоки алгоритма стараются размещать сверху вниз, в порядке их выполнения. Если весь алгоритм не помещается на одной странице, то его разделяют на части и в месте разделения на обеих страницах ставят определенную геометрическую фигуру с цифрой внутри, которая указывает на номер разделения (если их несколько). Для наглядности операции разного типа изображаются на схеме различными геометрическими фигурами, имеющими стандартный смысл. Основные геометрические фигуры и их предназанчение, которые наиболее часто используются при построении блок-схем, представлены в табл. 9. Овалом обозначается начало и конец алгоритма (в том числе вспомогательного), прямоугольником – присваивание значений переменным, ромбом – проверка условий (которые также можно испльзовать для изображения циклов с предусловием и постусловием), параллелограммом – операции ввода данных с клавиатуры и вывода данных на экран, кружочком или специальной стрелкой – операция перехода на другой лист или блок и т.д. Важной особенностью описания алгоритмов в виде блок-схем является «наполнение» каждого блока некоторыми формулами или пояснительным

текстом. Кроме того, должны соблюдаться правила по размерам соединительных линий и стрелок, ширине блоков, вложенных блоков и т. д., которые прописаны в ГОСТе 19.701-90 «СХЕМЫ АЛГОРИТМОВ, ПРОГРАММ, ДАННЫХ И СИСТЕМ». Требования стандарта являются обязательными.

Таблица 9. Основные элементы блок-схемы

Терминатор		Начало, завершение программы или подпрограммы
Процесс		Обработка данных
Данные		Ввод-вывод
Решение		Ветвление, выбор, итерационные и поисковые циклы
Подготовка		Счетные циклы
Граница цикла		Любые циклы
Предопределенный процесс		Вызов процедур

Структура алгоритма

Алгоритм любой степени сложности может быть составлен путем комбинации трех основных вариантов управления действиями исполнителя: *следование* (линейная структура), *ветвление* и *цикл*. Эти варианты управления принято называть *типовыми управляющими конструкциями*. Кроме того, используются дополнительные управляющие конструкции, представляющие собой модификацию основных конструкций. К ним относят *сокращенное ветвление* и *мно-*

жественное ветвление (переключатель). Управляющие конструкции определяют последовательность выполнения фрагментов алгоритма.

Следование предписывает последовательное выполнение блоков.

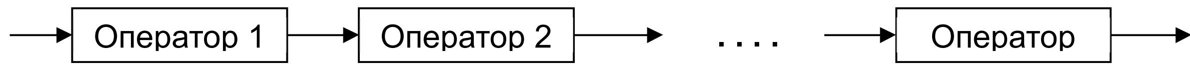

Рис. 4. Линейная структура алгоритма

Ветвление предписывает выбор одного из двух блоков в зависимости от того, выполняется условие или нет.

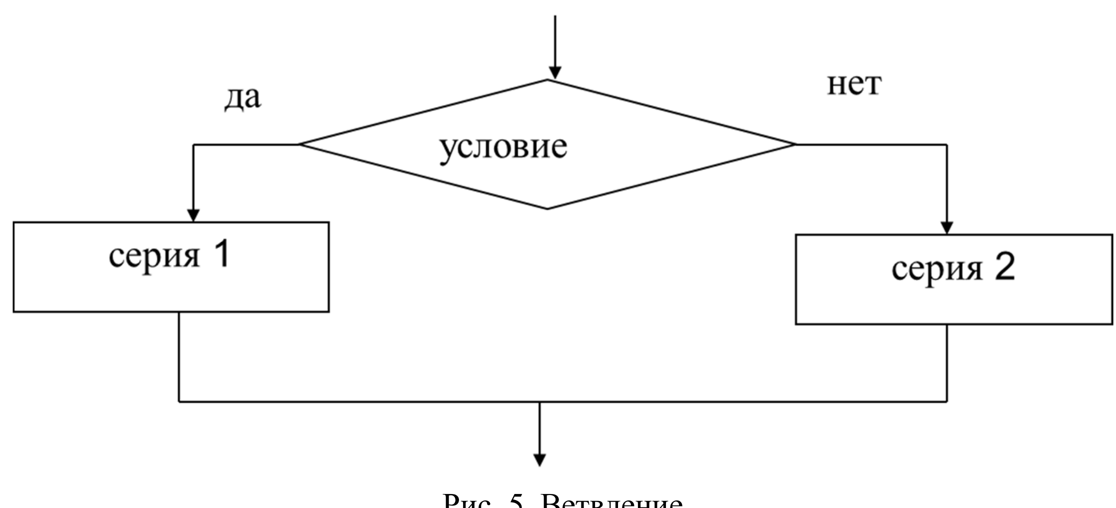

Рис. 5. Ветвление

Сокращенное ветвление подразумевает выполнение только операторов серии 1 (когда условие выполняется).

Циклический алгоритм – это такой алгоритм, при котором одно и то же действие повторяется несколько раз.

Команда повторения – это составная команда, в которой тело цикло выполняется несколько раз. Количество повторений может быть фиксировано алгоритмом или же на количество повторений может быть наложено какое-либо условие. Различают три типа цикла: цикл ДЛЯ (с параметрами), цикл ПОКА (предусловие) и цикл ДО (постусловие). Каждый из циклов выполняет некоторую последовательность команд, называемую *телом цикла*.

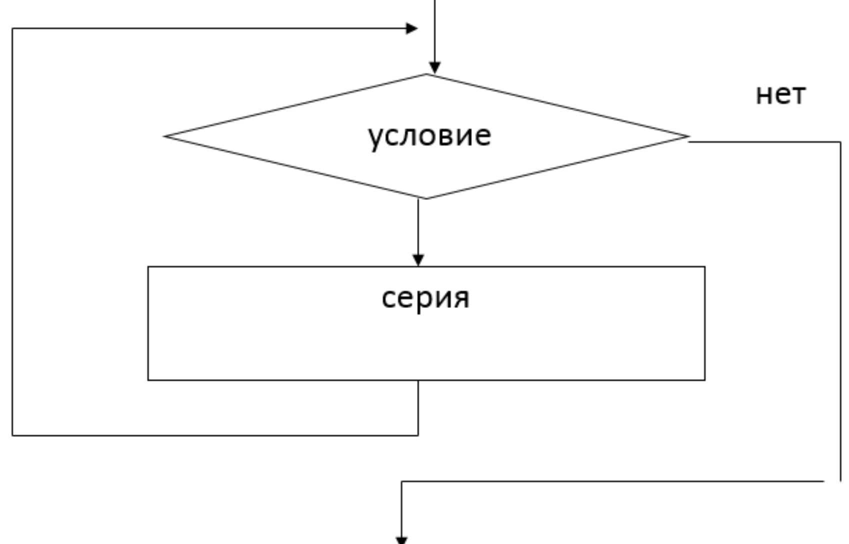

Рис. 6. Цикл ПОКА (выполняется пока условие истинно)

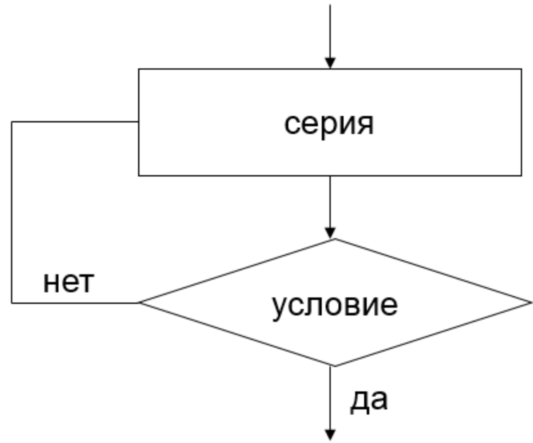

Рис. 7. Цикл ДО (выполняется до тех пор, пока условие не станет истинным)

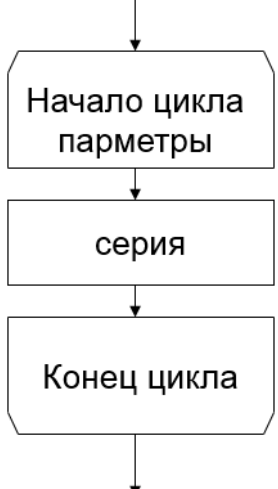

Рис. 8. Цикл ДЛЯ (все три блока собираются в один, в котором указывается начальное, конечное значение параметра и шаг изменения)

Вспомогательные алгоритмы

Если задача, для которой составляется алгоритм, очень сложная, то ее решение можно разбить на решение некоторого количества простых подзадач. Такой прием называется *декомпозицией*. Аналогично сложный алгоритм разбивается на *базовые структуры*, которые между собой могут быть соединены *последовательным* или *вложенным* способами. При *последовательном* соединении базовые структуры следуют одна за другой. При *вложенном* соединении одна базовая структура находится внутри другой базовой структуры. При таком соединении существует такое понятие, как *уровень вложенности*, который равен количеству вложенных базовых структур. Базовые структуры можно описать в одном большом алгоритме, а можно использовать *вспомогательные алгоритмы*.

Основным алгоритмом называется алгоритм решения основной задачи.

Вспомогательным алгоритмом называется алгоритм решения подзадачи.

Пути построения основного алгоритма:

- «сверху-вниз»: сначала строится основной алгоритм, затем – вспомогательные алгоритмы (метод последовательной детализации);
- «снизу-вверх»: сначала составляются вспомогательные алгоритмы, затем – основной (сборочный метод).

Вспомогательный алгоритм просто вызывается командой (командой присваивания или вызова) в основном алгоритме.

Парадигмы и языки программирования

Парадигма программирования – набор элементов понятийного аппарата, способ их представления и концепция взаимодействия, лежащие в основе базовых правил построения программ из сформированного множества базовых конструкций.

Парадигма программирования определяет стиль программирования, модели, методы и приемы.

Основные парадигмы программирования:

- императивное программирование – описывает процесс вычисления в виде инструкций, изменяющих состояние программы;
- декларативное программирование – программа генерируется по ее описанию;
- структурное программирование – представление программы в виде иерархической структуры блоков;

– функциональное программирование – заключается в выполнении ряда функций;
– объектно-ориентированное программирование – основными концепциями являются понятия объектов и классов.

Существует множество парадигм программирования. Согласно парадигмам и их взаимодействию можно рассматривать такие понятия, как *модели* и *методы* программирования. Модели программирования обычно носят те же названия, что и парадигма программирования, которая была выбрана основой для модели: императивные, декларативные, структурные, функциональные, логические, объектно-ориентированные (программирование, основанное на классах, прототипах, или субъектно-ориентированное программирование).

Среди методов программирования известны следующие:

– *структурное программирование* (программа в виде структуры, состоящей из блоков);
– *процедурное программирование* (выполнение программы сводится к последовательному выполнению операторов с целью преобразования исходного состояния памяти, т. е. значений исходных данных, в заключительное, т. е. в результаты. Таким образом, с точки зрения программиста имеются программа и память, причем первая последовательно обновляет содержимое последней);
– *аппликативное программирование* (подход к написанию программы состоит в систематическом применении одного объекта к другому. Результатом такого применения вновь является объект, который может участвовать в применениях как в роли функции, так и в роли аргумента и т. д. Это делает запись программы математически ясной. Тот факт, что функция обозначается выражением, свидетельствует о возможности использования значений-функций на равных правах с прочими объектами. Следовательно, значения-функции можно передавать как аргументы либо возвращать как результат вычисления других функций. Конструкции аппликативных, или функциональных, языков программирования в целом отличает простота исходных посылок. Их базовыми строительными блоками являются представления о выражении и функции, а все прочие понятия являются производными и вводятся шаг за шагом, что придаёт конструктивный стиль самому процессу программирования);
– *обобщённое программирование* (заключается в таком описании данных и алгоритмов, которое можно применять к различным типам данных, не меняя само это описание. В том или ином виде поддерживается разными языками программирования. Возможности обобщённого программирования впервые появились в 1970-х гг. в языках Клу и Ада, а затем во многих

объектно-ориентированных языках, таких как C++, Java, Object Pascal, D,Eiffel, языках для платформы .NET и других);

- *доказательное программирование* (использовавшаяся в 1980-х гг. в академических кругах технология разработки программ для ЭВМ с доказательствами правильности – доказательствами отсутствия ошибок в программах; доказательство может быть автоматизировано полностью лишь для очень небольшого круга простых теорий, поэтому важное значение получает его автоматическая проверка и для этого преобразование к проверяемому виду; для поддержания строгости при проверке доказательства верификатором следует проверить ещё и верификатор, для чего нужен ещё один верификатор и т. д. Получившуюся бесконечную цепь верификаторов можно было бы свернуть, построив верифицирующий себя верификатор, обладающий способностью развернуться до применимого на практике);
- *порождающее программирование* (технология разработки программного обеспечения, основанная на моделировании семейства программных систем, используя которые, можно по конкретным техническим требованиям автоматически получить специализированный и оптимизированный промежуточный или конечный программный продукт из элементарных, многократно используемых компонентов реализации с помощью базы знаний о конфигурациях);
- *аспектно-ориентированное программирование* (основные понятия АОП: *Аспект* () – модуль или класс, реализующий сквозную функциональность. Аспект изменяет поведение остального кода, применяя совет в точках соединения, определённых некоторым срезом. *Совет* () – средство оформления кода, который должен быть вызван из точки соединения. Совет может быть выполнен до, после или вместо точки соединения. *Точка соединения* () – точка в выполняемой программе, где следует применить совет. Многие реализации АОП позволяют использовать вызовы методов и обращения к полям объекта в качестве точек соединения. *Срез* () – набор точек соединения. Срез определяет, подходит ли данная точка соединения к данному совету. Самые удобные реализации АОП используют для определения срезов синтаксис основного языка (например, в AspectJ применяются Java-сигнатуры) и позволяют их повторное использование с помощью переименования и комбинирования. *Внедрение* (*введение*) – изменение структуры класса и/или изменение иерархии наследования для добавления функциональности аспекта в инородный код. Обычно реализуется с помощью некоторого *метаобъектного протокола* ());

– *агентно-ориентированное программирование* (агентом является всё, что может рассматриваться как воспринимающее свою среду с помощью датчиков и воздействующее на эту среду с помощью исполнительных механизмов);
– *рекурсия* (наличие в определении, описании, изображении какого-либо объекта или процесса самого этого объекта или процесса, т. е. ситуация, когда объект является частью самого себя);
– *автоматное программирование* (программа или её фрагмент осмысливается как модель какого-либо формального автомата);
– *событийно-ориентированное программирование* (способ построения компьютерной программы, при котором в коде (как правило, в головной функции программы) явным образом выделяется главный цикл приложения, тело которого состоит из двух частей: выборки события и обработки события);
– *компонентно-ориентированное программирование* (ключевой фигурой которого является компонент);
– *литературное программирование* (концепция, методология программирования и документирования).

Языком программирования называется система обозначений для описания алгоритмов и структур данных, определенная искусственная формальная система, средствами которой можно выражать алгоритмы. Язык программирования определяет набор лексических, синтаксических и семантических правил, задающих внешний вид программы и действия, которые выполняет исполнитель (компьютер) под ее управлением.

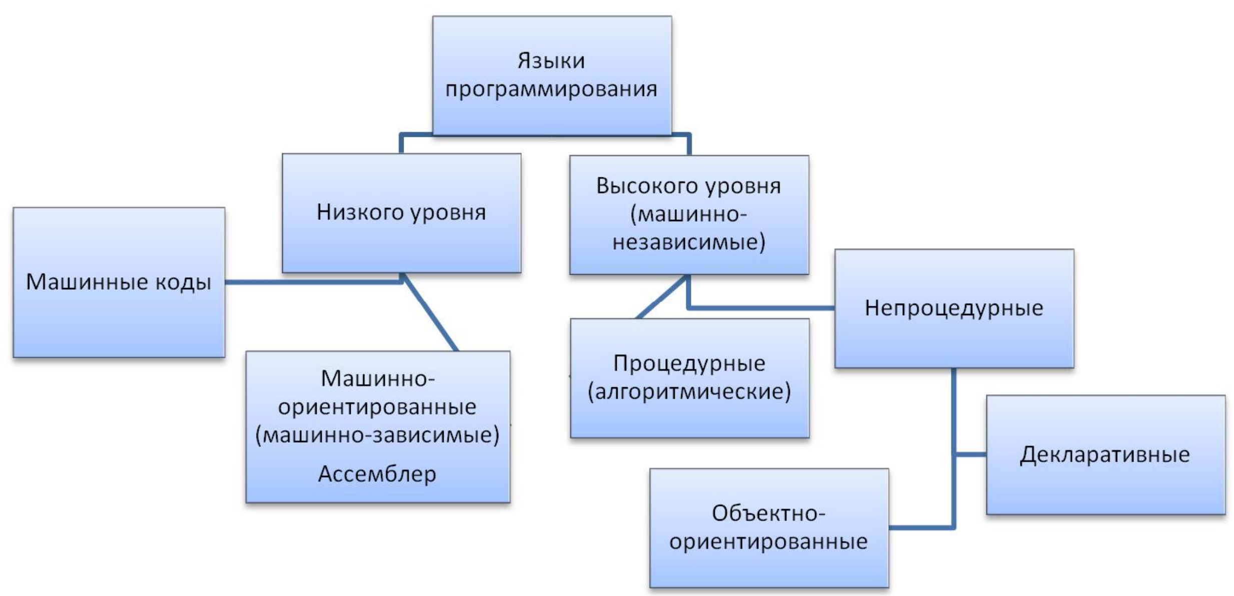

Рис. 9. Классификация языков программирования

Первым компьютерам приходилось программировать двоичными машинными кодами, что было сложно и трудоемко. Для упрощения стали появляться *языки программирования низкого уровня*, которые позволяли задавать машинные команды в более понятном для человека виде. Для преобразования их в двоичный код были созданы специальные программы – трансляторы.

Трансляторы делятся:

на *компиляторы* – превращают текст программы в машинный код, который можно сохранить и затем использовать уже без компилятора (примером являются исполняемые файлы с расширением *. exe);

интерпретаторы – превращают часть программы в машинный код, выполняют и после этого переходят к следующей части. При этом каждый раз при выполнении программы используется интерпретатор.

Примером языка низкого уровня является *ассемблер*. Языки низкого уровня ориентированы на конкретный тип процессора и учитывают его особенности, поэтому для переноса программы на ассемблере на другую аппаратную платформу ее нужно почти полностью переписать. Определенные различия имеются и в синтаксисе программ под разные компиляторы. Правда, центральные процессоры для компьютеров фирм AMD и Intel практически совместимы и отличаются лишь некоторыми специфическими командами. А вот специализированные процессоры для других устройств, например видеокарт, телефонов, содержат существенные различия.

Преимуществом языков низкого уровня является то, что можно создать эффективные и компактные программы, поскольку разработчик получает доступ ко всем возможностям процессора. Недостатки ЯПНУ:

- программист, работающий с языками низкого уровня, должен быть высокой квалификации, хорошо понимать устройство микропроцессорной системы, для которой создается программа. Так, если программа создается для компьютера, нужно знать устройство компьютера, особенно устройство и особенности работы его процессора;
- результирующая программа не может быть перенесена на компьютер или устройство с другим типом процессора;
- значительное время разработки больших и сложных программ;
- языки низкого уровня, как правило, используют для написания небольших системных программ, драйверов устройств, модулей стыков с нестандартным оборудованием, программирования специализированных микропроцессоров, когда важнейшими требованиями являются компактность, быстродействие и возможность прямого доступа к аппаратным ресурсам.

Языки программирования высокого уровня являются более понятными человеку, чем компьютеру. Особенности конкретных компьютерных архитектур в них не учитываются, поэтому созданные программы легко переносятся с компьютера на компьютер. В основном достаточно просто перекомпилировать программу под определенную компьютерную архитектурную и операционную систему. Разрабатывать программы на таких языках гораздо проще, и ошибок допускается меньше. Значительно сокращается время разработки программы, что особенно важно при работе над большими программными проектами. К языкам программирования высокого уровня относятся Фортран, Кобол, Алгол, Pascal, Java, C/C++, C#, Delphi и многие другие современные языки. Их недостатком является больший размер программ по сравнению с программами на языке низкого уровня. Поэтому языки высокого уровня в основном используются для разработок программного обеспечения компьютеров и устройств, которые имеют большой объем памяти. А разные подвиды ассемблера применяются для программирования других устройств, где критичным является размер программы.

Основным понятием для ЯПВУ является *абстракция*. В настоящее время уже существуют так называемые *языки программирования сверхвысокого уровня*, которые отличаются высоким уровнем абстракции. К языкам сверхвысокого уровня часто относят такие современные языки, как Python, Ruby и Haskell, а также Perl и предшествовавший ему мини-язык AWK.

Лекция 1.6

Структурное программирование

В основе структурного программирования лежит представление программы в виде иерархической структуры блоков. Предложена в 70-х гг. XX в. Э. Дейкстрой, разработана и дополнена Н. Виртом.

Теорема Бёма-Якопини (положение структурного программирования). Любой исполняемый алгоритм может быть преобразован к структурированному виду, т. е. такому виду, когда ход его выполнения определяется только при помощи трёх структур управления: следования, ветвления и цикла.

Структурное программирование делает текст программы более понятным – алгоритм решения ясно виден из исходного текста. Структурное программирование называют программированием без GOTO. Его методология основана на использовании подпрограмм и независимых структур данных, объединяющих связанные между собой совокупности данных. Подпрограммы позволяют заменять в тексте программ упорядоченные блоки команд, отчего программный код становится более компактным. Структурный подход обеспечивает создание

более понятных и легко читаемых программ, упрощает их тестирование и отладку.

Наиболее распространённые ЯПВУ, которые поддерживают приемы структурного программирования: Алгол, Фортран, Basic, Pascal, C/C++ и другие. В рамках данного курса мы с вами познакомимся с ЯПВУ C/C++.

Язык С был разработан в 1969-73 гг. Денисом Ритчи при создании операционной системы UNIX. Он был создан как инструмент для программистов-практиков. Язык программирования С++ произошёл от С (Бьёрн Страуструп). Однако в дальнейшем С и С++ развивались независимо, что привело к росту несовместимости между ними. Редакция С99 добавила в язык несколько конфликтующих с С++ особенностей.

Структура программы на С/С++

Схематично структура программы изображена на рис. 10. Программа на С/С++ состоит из одной главной функции или нескольких функций (главной и вспомогательных).

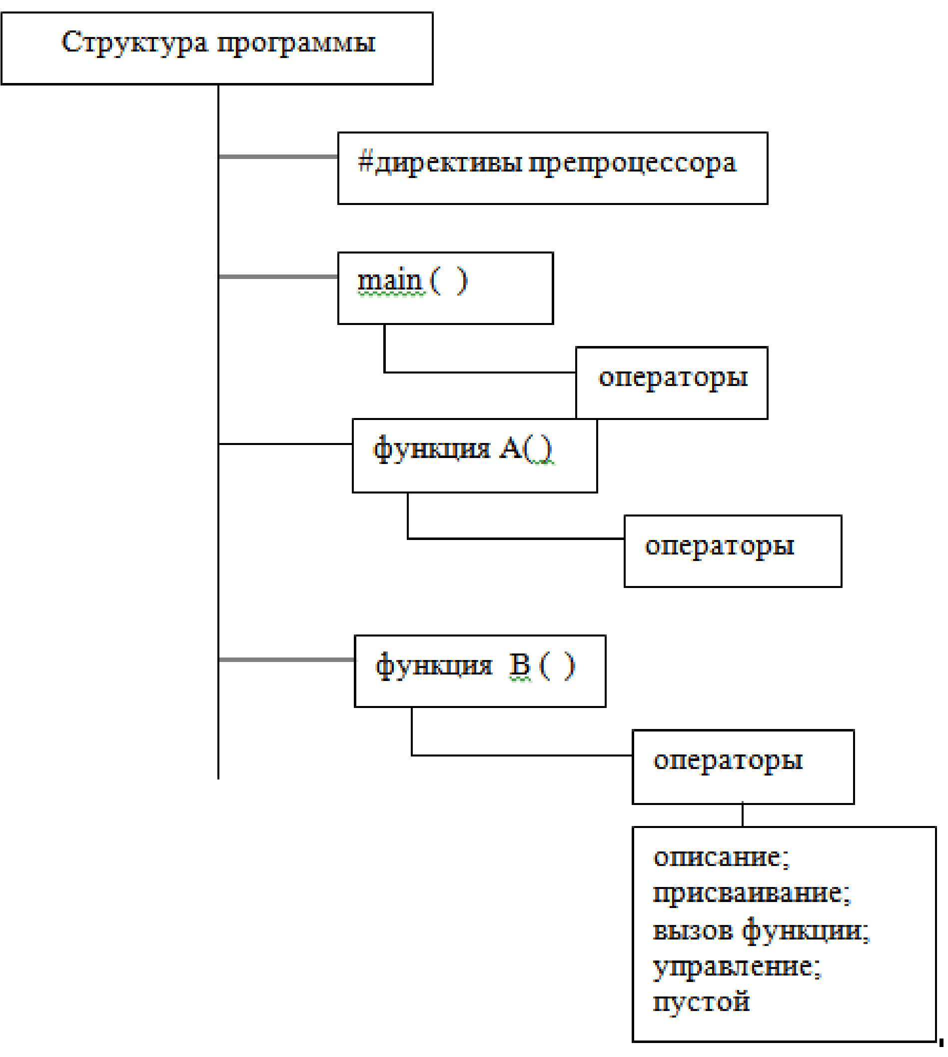

Рис. 10. Структура программы на С/С++

Главная функция (обязательная) называется main. Все остальные функции могут носить произвольные имена. В начале программы необходимо подключить необходимые библиотеки, которые обеспечивают программу стандартными функциями ввода, вывода, работы с файлами, математическими функциями и многими другими. Библиотеки подключаются с помощью директив препроцессора, которые представляют собой инструкции, записанные в тексте программы и выполняемые до трансляции программы.

Элементы языка С/С++

Символы (алфавит) – это основные неделимые знаки, с помощью которых пишутся все тексты.

Лексема – минимальная единица языка, имеющая самостоятельный смысл.

Выражение задаёт правило вычисления некоторого значения.

Оператор задаёт законченное описание некоторого действия.

Группа операторов составляет блок или *составной оператор*. Все операторы делятся на исполнимые и неисполнимые.

Символами могут выступать:

- большие и маленькие латинские буквы;
- цифры;
- специальные символы: “ { } , . | [ ] ( ) + - / % \ ; ‘ : ? < = > _ ! & # ~ ^ * пробел;
- комбинация символов: ++ -- == || << >> <= >= += -= *= |= ?: |* */ //.

Лексемами являются:

- идентификаторы (последовательность латинских букв, цифр, символов подчёркивания (_), начинающаяся с буквы или символа подчёркивания, например: KOM_6, __abc, A_B_C, MAX, Max);
- служебные (ключевые) слова (это идентификаторы, зарезервированные в языке программирования);
- константы;
- строковые константы;
- операции (знаки операций);
- разделители (знаки пунктуаций).

Данные и величины, типы данных

Программа на ЭВМ работает с данными. Данными называется совокупность величин. Данные бывают *входными* (начальными, исходными) и *выходными* (результатом).

Величины бывают *постоянными* (константы) и *переменными*. Постоянные величины во время работы программы не меняют своего значения, переменные величины могут менять свое значение во время работы программы несколько раз.

Основные свойства величин:

- имя (в программе уникальный идентификатор);
- значение (может быть начальным и меняться во время работы программы);
- тип (задается в начале программы при объявлении переменной или константы).

Минимальным необходимым набором типов данных для компьютерной программы являются следующие:

- целый (целые числа);
- вещественный (вещественные числа);
- логический («истина», «ложь» (1, 0));
- символьный (цифры, буквы и др.).

Свойства величины, зависящие от типа (табл. 10):

- множество допустимых значений;
- множество допустимых операций;
- форма внутреннего представления.

Типы данных в С/С++ бывают *скалярные* и *составные*. К скалярным типам данных относятся арифметические и указатели, среди которых есть целые и вещественные. К составным относятся массивы, структуры, объединения, перечисления. Все типы данных, используемые в С/С++, описаны в табл. 11.

Описание переменных в программах на языке С/С++ имеет вид

<имя_типа> <список_переменных>;

Пример:

```
int number, row;
float x, X, cc3;
```

Одновременно с описанием можно задать начальные значения переменным. Такое действие называется инициализацией переменных. Описание с инициализацией производится по следующей схеме:

<имя типа> <имя_переменной> = <начальное_значение>;

Таблица 10. Свойства величин основных типов

Тип	**Значения**	**Операции**	**Внутреннее представление**
Целый	Целые положительные и отрицательные числа в некотором диапазоне. *Примеры*: 23, -12, 387.	Арифметические операции с целыми числами: +, –, ×, целое деление и остаток от деления. Операции отношений	Формат с фиксированной точкой
Вещественный	Любые (целые и дробные) числа в некотором диапазоне. *Примеры*: 2.5, -0.01, 45.0, 3.6×10^9	Арифметические операции: +, –, ×, /. Операции отношений	Формат с плавающей точкой
Логический	True (истина) False (ложь)	Логические операции: И (and), ИЛИ (or), не (not). Операции отношений	1 бит: 1 – true; 0 – false
Символьный	Любые символы компьютерного алфавита. *Примеры*: ‘a’, ‘5’, ‘+’, ‘$’	Операции отношений	Коды таблицы символьной кодировки. 1 символ – 1 байт

Ниже приведен пример инициализации переменных при объявлении: год, число π и вещественного числа c:

```
unsigned int year=2000;
float pi=3.14159, c=1.23;
```

Таблица 11. Типы данных C/C++

Тип данных	Размер занимаемой памяти (байт)	Диапазон возможных значений	Эквивалентные названия типа
char	1	-128…+127	signed char
bool	1	False, True	
int	2 (4)	зависит от СП	signed, signed int
unsigned char	1	0…255	нет
unsigned int	2 (4)	зависит от СП	unsigned
short int	2	-32768…32767	short, signed short int
unsigned short	2	0…65535	unsigned short int
long int	4	-2147483648…2147483647	long, signed long int
unsigned long int	4	0…4294967295	
float	4	$\pm(3.4E-38...3.4E+38)$	нет
double	8	$\pm(1.7E-308...1.7E+308)$	нет
long double	10	$\pm(3.4E-4932...3.4E+4932)$	нет

Оператор typedef. В язык C/C++ введено специальное средство, позволяющее назначать имена типам данных (переименовывать). Таким средством является оператор typedef. Он записывается в следующем виде: typedef <тип> <имя>;

Здесь «тип» − любой разрешенный тип данных и «имя» − любой разрешенный идентификатор.

Например,

typedef int INTEGER;

После этого можно сделать объявление

INTEGER a, b;

Оно будет выполнять то же самое, что и привычное объявление int a,b;. INTEGER можно использовать как синоним ключевого слова int.

Запись констант производится следующим образом:

Целые десятичные числа, начинающиеся не с нуля. Например: 4, 356, -128.

Целые восьмеричные числа, запись которых начинается с нуля. Например: 016, 077.

Целые шестнадцатеричные числа, запись которых начинается с символов 0x или 0X. *Например*: 0x1A, 0X253, 0xFFFF.

Если в записи числовой константы присутствует десятичная точка (2.5) или экспоненциальное расширение (1E-8), то компилятор рассматривает ее как вещественное число и ставит ей в соответствие тип double. *Примеры вещественных констант*: 44. 3.14159 44E0 1.5E-4 Программист может явно задать тип константы, используя для этого суффиксы. Существует три вида суффиксов: F(f) – float; U(u) – unsigned; L(l) – long (для целых и вещественных). Допускается совместное использование суффиксов U и L в вариантах UL или LU. Примеры: 3.14159F – константа типа float, 3.14L – константа типа long double, 50000U – константа типа unsigned int, 0LU – константа типа unsigned long,24242424UL - константа типа unsigned long.

Именованные константы. Примеры: const float pi=3.14159; const int iMIN=1, iMAX=1000; Еще одной возможностью ввести *именованную константу* является использование препроцессорной директивы #define в следующем формате:

#define <имя константы> <значение константы>

Например, #define iMIN 1.

Тип констант явно не указывается. В конце директивы не ставится точка с запятой.

Константы перечисляемого типа. Определяет последовательность целочисленных именованных констант. Описание перечисляемого типа начинается со служебного слова enum, а последующий список констант заключается в фигурные скобки. *Например*:

enum {A, B, C, D};

Для любой константы можно явно указать значение. *Например*:

enum {A=10, B, C, D};

Возможен и такой вариант определения перечисления:

enum {A=10, B=20, C=35, D=100};

Если перечисляемому типу дать имя, то его можно использовать в описании переменных. *Например*:

enum metal {Fe, Co, Na, Cu, Zn};

metal Met1, Met2;

Здесь идентификатор metal становится именем типа. Возможны следующие операторы: Met1=Na; Met2=Zn;

Операции с данными

Выражение – операнды, объединенные знаками операций. Любое выражение, завершенное «;» – оператор. Одиночный символ «;», не относящийся ни к одному оператору, – *пустой оператор*.

Операндами могут быть:

- вызов метода;
- переменная;
- константа;
- выражение.

Операции имеют *приоритет*, который предписывает очередность выполнения операций в выражении. Операции одного приоритета выполняются слева направо. Исключения будут оговорены особо. Порядок операций можно регулировать скобками «()».

В зависимости от числа операндов различают:

- унарные;
- бинарные;
- тернарные.

Приоритеты операций по группам:

1. Вызов метода.
2. Унарные операции.
3. Арифметические операции.
4. Сдвиги.
5. Отношения.
6. Битовые.
7. Логические.
8. Тернарная.
9. Присваивание.

Операция присваивания. Присваивание бывает *простым* и *составным*. В простом присваивании имеет место только присваивание. Например, A=5; (переменной A присвоено значение 5). В составном присваивании присутствует комбинация присваивания и еще какой-либо операции. Например, A *=B (что значит A = A*B, переменной A присваивается выражение A*B).

Присваивание происходит по следующему алгоритму: вычисляется значение выражения; если тип переменной и тип выражения не совпадают, то значение выражения преобразуется к типу переменной; значение выражения заносится в область памяти, отведенную для переменной.

Условиями преобразования переменной по умолчанию являются совместимость типов выражения и переменной; преобразование не приведет к потере информации. Потеря информации возможна при присваивании целой перемен-

ной вещественного значения (теряется дробная часть) и при присваивании переменной с меньшим диапазоном значений значения переменной с большим диапазоном значений. Рассмотренная схема преобразования типов уникальна только для операции присваивания, во всех остальных операциях действует другая схема. Перед выполнением операции операнд младшего типа преобразуется к операнду старшего типа. Старшим считается тип, у которого больше диапазон значений.

Арифметические операции.

- -   вычитание или унарный минус;
- +   сложение или унарный плюс;
- *   умножение;
- /   деление;
- %   деление по модулю (аналог mod в Паскале);
- ++ унарная операция увеличения на единицу (инкремент);
- -- унарная операция уменьшения на единицу (декремент).

Примеры: A=B+5; A++; A=--B;

Операции инкремента и декремента могут использоваться как в префиксной форме (знак перед переменной), так и в постфиксной форме (знак после переменной). Отличия будут проявляться при использовании операции в выражении: префиксная – сначала изменяется значение переменной, потом используется новое значение; постфиксная – сначала используется старое значение переменной, потом изменяется значение переменной.

Пример показан ниже.

a = 3; b = 2;
c = a++ * b++;
Результат выполнения: a=4, b=3, c=6.
	a = 3; b = 2;
	c = ++a * ++b;
	Результаты: a=4, b=3, c=12.

Операция явного преобразования типа выглядит следующим образом:

(тип приемника) операнд.

Пример: (int) A;

Преобразование типов в С/С++ происходит по следующим правилам:

1) если один из операндов в выражении имеет тип long double, то остальные тоже преобразуются к типу long double;
2) в противном случае, если один из операндов в выражении имеет тип double, то остальные тоже преобразуются к типу double;
3) в противном случае, если один из операндов в выражении имеет тип float, то остальные тоже преобразуются к типу float;

4) в противном случае, если один из операндов в выражении имеет тип unsigned long, то остальные тоже преобразуются к типу unsigned long;
5) в противном случае, если один из операндов в выражении имеет тип long, то остальные тоже преобразуются к типу long;
6) в противном случае, если один из операндов в выражении имеет тип unsigned, то остальные тоже преобразуются. к типу unsigned;
7) в противном случае все операнды преобразуются к типу int. При этом тип char преобразуется в int со знаком; тип unsigned char в int, у которого старший байт всегда нулевой; тип signed char в int, у которого в знаковый разряд передается знак из char; тип short в int (знаковый или беззнаковый).

Логические операции – операции с логическими переменными.

! операция отрицания (НЕ);

&& конъюнкция, логическое умножение (И);

|| дизъюнкция, логическое сложение (ИЛИ).

Например:

bool A;

A = !(10>5 && 3 == 8);

Операции сдвига. Для знаковых типов – арифметический сдвиг, для беззнаковых – логический (>> – сдвиг вправо, << – сдвиг влево).

Пример: A = 14 << 2;

Переменная A имеет значение 56, поскольку при сдвиге значения 14 (00001110 в двоичном выражении) на два бита влево, получается значение 56 (00111000 в двоичном выражении).

Битовые операции. Для двух переменных одинакового скалярного типа определены битовые операции:

& битовое И (AND)

| битовое ИЛИ (OR)

^ битовое ИСКЛЮЧАЮЩЕЕ ИЛИ (XOR)

~ битовое ОТРИЦАНИЕ (NOT) – унарная операция.

Битовые операторы работают следующим образом. Берутся два операнда, и к каждой паре соответствующих битов для левого и правого операнда применяется данная операция, результатом будет переменная того же типа, каждый бит которой есть результат применения соответствующей логической операции к соответствующим битам двух операндов. Битовое отрицание числа (величина f в примере) – это число, полученное из исходного заменой всех нулей на единицы, и наоборот. Применение побитового отрицания к неотрицательному числу даст отрицательное число, что связано с особенностями представления отрицательных чисел в виде дополнительного кода.

Пример:

a = 5 (101)

b = 6 (110)

c = a & b (100 == 4)

d = a | b (111 == 7)

e = a ^ b (11 == 3)

f = ~ a (1...11111010 == –6)

Тернарная операция условие «?:» . Предназначена для реализации простейшего варианта ветвления

Операнд1 ? Операнд2 : Операнд3

Пример. Вычисление абсолютной величины переменной X можно организовать с помощью одной операции: X<0 ? –X : X;

Операции отношения:

<	меньше;
<=	меньше или равно;
>	больше;
>=	больше или равно;
==	равно;
!=	не равно.

Операция sizeof используется для нахождения количества байтов, занимаемых величиной явно указанного типа или величиной, полученной в результате вычисления выражения. Имеет две формы записи: sizeof(тип) и sizeof(выражение).

Пример:

sizeof(int)	результат – 2;	sizeof(1)	результат – 2;
sizeof(0.1)	результат – 8;	sizeof(1L)	результат – 4;
sizeof(char)	результат – 1;	sizeof('a')	результат – 2.

Операция запятая используется для связывания нескольких выражений в одно. Несколько выражений, разделенных запятыми «,» вычисляются последовательно слева направо. В качестве результата такого совмещенного выражения принимается значение самого правого выражения. Например, если переменная X имеет тип int, то значение выражения (X=3, 5*X) будет равно 15, а переменная X примет значение 3.

Организация ввода/вывода данных в C/C++

Кроме ранее рассмотренных операций над данными возникает необходимость ввода данных в программу и вывода результатов ее работы. Данные (входные) в программу можно ввести несколькими способами: ввод с клавиа-

туры, считывание с файла, непосредственное присваивание в тексте. Данные можно вывести на экран компьютера или (если есть необходимость сохранить их для дальнейшего использования) записать в файл. Для организации ввода и вывода данных на всех языках программирования созданы специальные функции. Рассмотрим функции ввода, вывода данных на языке программирования C/C++.

Форматированный ввод-вывод данных. Для организации форматированного ввода-вывода на языке C/C++ в программе должна быть подключена стандартная библиотека <stdio.h> с помощью директивы #include.

Для организации ввода используются функции:

int getchar(void); (чтение одного символа)

gets(); (чтение строки)

scanf(“управляющая строка”, &арг1, &арг2…);

Управляющая строка: спецификации преобразования;

Аргументы должны быть указателями, перед ними ставится &.

Пример: scanf(“%d”, &a);

Для организации вывода используются функции:

int putchar(int); (вывод одного символа)

puts(); (вывод строки)

printf(“управляющая строка”, арг1, арг2…);

Управляющая строка: текст, спецификации преобразования, управляющие символьные константы.

Пример: printf(“Number of group is %d”, n);

Спецификации преобразования функции scanf: %c_n

c – одиночный символ;

d или i – десятичное целое число (аргумент типа int);

D или l – десятичное целое число (аргумент типа long);

e или E – вещественное число с плавающей точкой;

f – вещественное число с плавающей точкой;

o – восьмеричное целое число (аргумент типа int);

O – восьмеричное целое число (аргумент типа long);

s – появление строки символов;

x – шестнадцатеричное целое число (аргумент типа int);

X – шестнадцатеричное целое число (аргумент типа long).

Модифицированные типы:

hd – short int;

ld – long int;

lf, le – double;

Lf, Le – long double.

Спецификации преобразования функции printf: %[признаки][ширина поля][точность]c_n

c – символ;

d или i – десятичное целое число;

e или E – вещественное десятичное число в экспоненциальной форме вида 1.23e+2 или 1.23E+2;

f – значением аргумента является вещественное десятичное число с плавающей точкой;

g (или G) – используется как e или f и исключает вывод незначащих нулей;

o – восьмеричное целое число;

s – строка символов;

u – беззнаковое целое число;

x или X – шестнадцатеричное целое;

p – указатель.

Необязательные параметры:

признак минус (-) – преобразованный параметр должен быть выровнен влево в своем поле;

признак плюс (+) – вывод результата со знаком;

строка цифр, задающая минимальный размер поля (ширина поля). Здесь может также использоваться символ *, который задает минимальную ширину поля и точность представления выводимого числа;

точка (.), отделяющая размер поля от последующей строки цифр;

строка цифр, задающая максимальное число выводимых символов, или же количество цифр, выводимых справа от десятичной точки в значениях типов float или double (точность).

Управляющие символьные константы:

\a – для кратковременной подачи звукового сигнала;

\b – для перевода курсора влево на одну позицию;

\f – для подачи формата;

\n – для перехода на новую строку;

\r – для возврата каретки;

\t – горизонтальная табуляция;

\v – вертикальная табуляция;

\\ – вывод символа \;

\' – вывод символа ' ;

\» – вывод символа «;

\? – вывод символа ?.

Потоковый ввод-вывод данных. Для организации потокового ввода-вывода на языке С++ в программе должна быть подключена стандартная библиотека <iostream.h> с помощью директивы #include.

Для организации ввода используются потоки cin и cout.

Пример: cin>>a; cout<<" Number of group is "<<n;

Для форматирования ввода-вывода используются так называемые *манипуляторы* и функции:

endl – при выводе перейти на новую строку;

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

flush – немедленно вывести и очистить все промежуточные буферы;

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 n символов (n -целое);

setfill(int n) – установить символ-заполнитель; этим символом выводимое значение будет дополняться до заданной 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 (0 означает систему счисления по умолчанию, т.е. 10).

Программы с линейной структурой

Линейная структура программы представлена операторами ввода-вывода и операторами присваивания. В примере ниже реализована задача нахождения длины окружности по введённому с клавиатуры значению радиуса.

Пример:

```
#include<stdio.h>
#include<math.h>
#define pi 3.14
void main()
{
 float r, S;
 printf("\nвведите значение радиуса:");
 scanf("%f", &r);
 S=2*pi*r;
```

```
printf("\nЗначение длины окружности=%.2f", S);
}
```

Лекция 1.7
Программы, содержащие ветвление

Структура ветвления в программе C/C++ организуется несколькими способами.

– *операции условия «? :»*

```
<Условие> ? <Оператор 1> : <Оператор 2>;
```

– *условного оператора if … else*

```
if (<Условие>) <Оператор 1>; else  <Оператор 2>;
```

– *оператора выбора switch ()*

```
switch (<целочисленное_выражение>)
{
  case <константа 1>: <операторы 1>;
  case < константа 2>: <операторы 2>;
  …
  case < константа n >: <операторы n>;
  [default: <операторы>;]
}
```

Операция условия используется в случае простой альтернативы. Ниже в примере рассмотрена реализация поиска максимального из двух целых чисел, введенных с клавиатуры.

Пример:

```
#include<stdio.h>
void main()
{
 int a, b, max;
 printf("\nвведите два целых числа:");
 scanf("%d%d", &a,&b);
 (a>=b) ? max=a : max=b;
 printf(  "max=%d", max);
}
```

Эта же задача, реализованная с помощью условного оператора, приведена ниже.

Пример:

```
#include<stdio.h>
void main()
{
  int a, b, max;
  printf("\nвведите два целых числа:");
  scanf("%d%d", &a,&b);
  if (a>=b) max=a;
   else max=b;
  printf(  "max=%d", max);
}
```

Условный оператор может быть вложен в другой условный оператор. Кроме того, условный оператор может быть неполным, т. е. без *else*. Выражение условия может быть простым и составным, т. е. состоять из нескольких простых выражений, связанных логическими связками. Например, ниже реализована задача поиска минимального значения из трех целых чисел с помощью вложенных условных операторов и составных условий.

Пример:

```
#include<stdio.h>
void main()
{
  int a, b, c, min;
  printf("\nвведите три  целых числа:");
  scanf("%d%d%d", &a,&b, &c);
  if  (a<=b && a<=c)   min=a;
  else
    if  (b<a && b<=c) min=b;
    else
      if (c<a && c<b) min=c;
  printf(  "min=%d", min);
}
```

Особенностью оператора switch является то, что при целочисленном значении, равном какой-либо из констант, начинают выполняться соответствующие этой константе операторы и далее все последующие операторы ниже описанных констант. Если есть необходимость выполнять только операторы выбранной константы, необходимо последним оператором ставить оператор break. Ниже приведен пример фрагмента задачи, реализующей вывод оценки на экран. В каждом случае последним оператором является оператор break, чтобы выводилась только одна оценка. Если бы этого оператора не было, то при вы-

боре, например, случая 4 на экране было бы напечатано сначала «хорошо», потом «отлично» и «нет такой оценки».

Пример:

```
…
switch(ball)
{
  case 2: cout<<“не удовлетворительно\n”; break;
  case 3: cout<<“удовлетворительно\n”; break;
  case 4: cout<<“хорошо\n”; break;
  case 5: cout<<“отлично\n”; break;
  default: cout<<“нет такой оценки\n”;
}
…
```

Программы, содержащие циклы

Программировать циклы можно, используя приемы цикла по счетчику (с заданным числом повторений) и итерационного цикла (число повторений заранее неизвестно). Циклы в программе C/C++ бывают трех видов.

– *Цикл с параметрами*

```
for(<выражение_1>;< выражение_2>;< выражение_3>)
 <тело цикла>
```

Выражение 1 содержит инициализацию переменной цикла, выражение 2 является критерием окончания цикла (условие, при котором цикл выполняется), выражение 3 описывает правило изменения переменной цикла. Тело цикла содержит операторы, которые должны выполняться. Тело цикла может быть пустым.

Пример. Нахождение факториала 10.

```
…
F=1;
for(i=1; i<=10; i++)
{
   F=F*i;
}
…
```

– *Цикл с предусловием*

```
while(<условие>)
<тело цикла>
```

Условие представляет собой условия выполнения цикла. Тело цикла содержит операторы, которые должны выполняться. Тело цикла может быть пустым.

Пример. Нахождение факториала 10.

```
…
F=1;
i=1;
while(i<=10)
{
  F=F*i;
  i++;
}
…
```

– *Цикл с постусловием*

do <тело цикла> while(<условие>);

Условие также представляет собой условия выполнения цикла (в этом есть отличие C/C++ от других языков программирования, в которых в данном случае описано условие окончания цикла). Тело цикла содержит операторы, которые должны выполняться. Тело цикла может быть пустым.

Пример. Нахождение факториала 10.

```
…
F=1;
i=1;
do
{
 F=F*i;
 i++;
}
while(i<=10);
…
```

Особенностью цикла с параметрами является то, что каждое из выражений можно опускать, в этом случае ставится просто пустой оператор ; и, тогда его можно использовать так же, как циклы с постусловием и предусловием. Если пропущено первое выражение, в котором описывается инициализация переменной цикла, то эта инициализация должна быть описана до начала цикла, как в цикле с предусловием. Если пропущено второе выражение, в котором описан критерий окончания цикла, то это условие должно быть описано в теле цикла, и тогда цикл будет работать по принципу цикла с постусловием. Если пропущено выражение 3, то правило изменения переменной цикла описывается оператором

в теле цикла, как реализовано в циклах с постусловием и предусловием. Ниже приведены примеры опущенных параметров в цикле for при той же задаче вычисления факториала.

Примеры. Цикл без описания первого параметра:

```
…
F=1; i=1;
for( ; i<=10; i++)
{
  F=F*i;
}
…
```

Цикл без описания второго параметра:

```
…
F=1;
for(i=1; ; i++)
{
  if(i<=10)
    F=F*i;
  else return;
}
…
```

Цикл без описания третьего параметра:

```
…
F=1;
for(i=1; i<=10;  )
{
  F=F*i;
  i++;
}
…
```

Как и ветвления, циклы могут быть вложены друг в друга. Например, если необходимо совершить какие-либо действия по двум и более изменяющимся переменным. Особенно востребовано вложение циклов при работе с матрицами, которое будет подробно рассмотрено позже при изучении составных структур данных. Ниже приведен пример вложенных циклов for.

Пример:

```
…
for(i=1; i<=10; i++)
  for(j=1; j<=10; j++)
```

```
    printf (“рассмотрены пары чисел %d  и %d\n”, i, j);
…
```

На экран выведется текст:

рассмотрены пары чисел 1 и 1
рассмотрены пары чисел 1 и 2
…
рассмотрены пары чисел 1 и 10
рассмотрены пары чисел 2 и 1
…
рассмотрены пары чисел 2 и 10
…
рассмотрены пары чисел 10 и 10

То есть внутренний цикл является оператором внешнего цикла и сначала идет изменение переменной внутреннего цикла, а при достижении критерия окончания работы внутреннего цикла меняется переменная внешнего цикла.

Можно использовать в одном цикле несколько переменных цикла, в этом случае надо тщательно прописывать условия выполнения цикла и правило изменения переменных цикла. Ниже описан пример использования цикла for с двумя переменными цикла, похожий на предыдущий пример, однако результат его работы будет другим.

Пример:

```
…
for(i=1, j=1; i<=10 &&  j<=10; i++, j++)
  printf (“рассмотрены пары чисел %d  и %d\n”, i, j);
…
```

На экран выведется текст:

рассмотрены пары чисел 1 и 1
рассмотрены пары чисел 2 и 2
…
рассмотрены пары чисел 10 и 10

То есть в этом случае обе переменные цикла будут менять свои значения одновременно.

Аналогично можно использовать вложенные циклы и циклы с несколькими итерационными переменными в случае циклов с предусловием и постусловием. Таким образом, работа с циклами очень разнообразна и позволяет реализовывать достаточно сложные задачи.

Лекция 1.8

Программное обеспечение ЭВМ

Программное обеспечение компьютера – это совокупность программ, хранящихся в долговременной памяти компьютера и предназначенных для автоматизации решения различных задач на ЭВМ. Программное обеспечение делится на системное, прикладное и инструментальное.

Системное ПО – это набор программ, обеспечивающих работу компьютера, в том числе управление ресурсами ЭВМ, взаимосвязь ЭВМ и пользователя, создание копий используемой информации, проверка работоспособности устройств компьютера, выдача справочной информации и другие функции.

Инструментальное ПО – это набор программ для создания новых программ для ЭВМ.

Прикладное ПО – это набор программ, предназначенных для решения прикладных задач на ЭВМ.

Системное ПО в свою очередь делится на базовое и сервисное.

Базовое ПО поставляется пользователю вместе с компьютером и обеспечивает его работоспособность. В его состав входят:

- операционная система (запуск и работа компьютера, функционирование других программ на компьютере, диагностика и контроль работоспособности устройств компьютера, выполнение других технологических процессов. Самые распространенные ОС: Windows, Linux, Mac OS, NetWare, Solaris, MS DOS и др.);
- операционная оболочка (комфортное общение пользователя с ЭВМ, текстовый и графический интерфейсы. Например, Norton Commander, Windows Commander и др.);
- сетевые программные средства (работа компьютера в сети, поддержка сетевых служб. Сюда входят программы электронной почты, обозреватели).

Сервисное ПО расширяет возможности компьютера и может приобретаться за отдельную плату или в последующем поставляться через Интернет (программы диагностики работоспособности компьютера, антивирусные программы, программы обслуживания дисков, программы-архиваторы, программы обслуживания сети).

Характеристика антивирусных программ. Для обнаружения, удаления и защиты от компьютерных вирусов разработано несколько видов специальных программ, которые позволяют обнаруживать и уничтожать вирусы. Такие программы называются *антивирусными*.

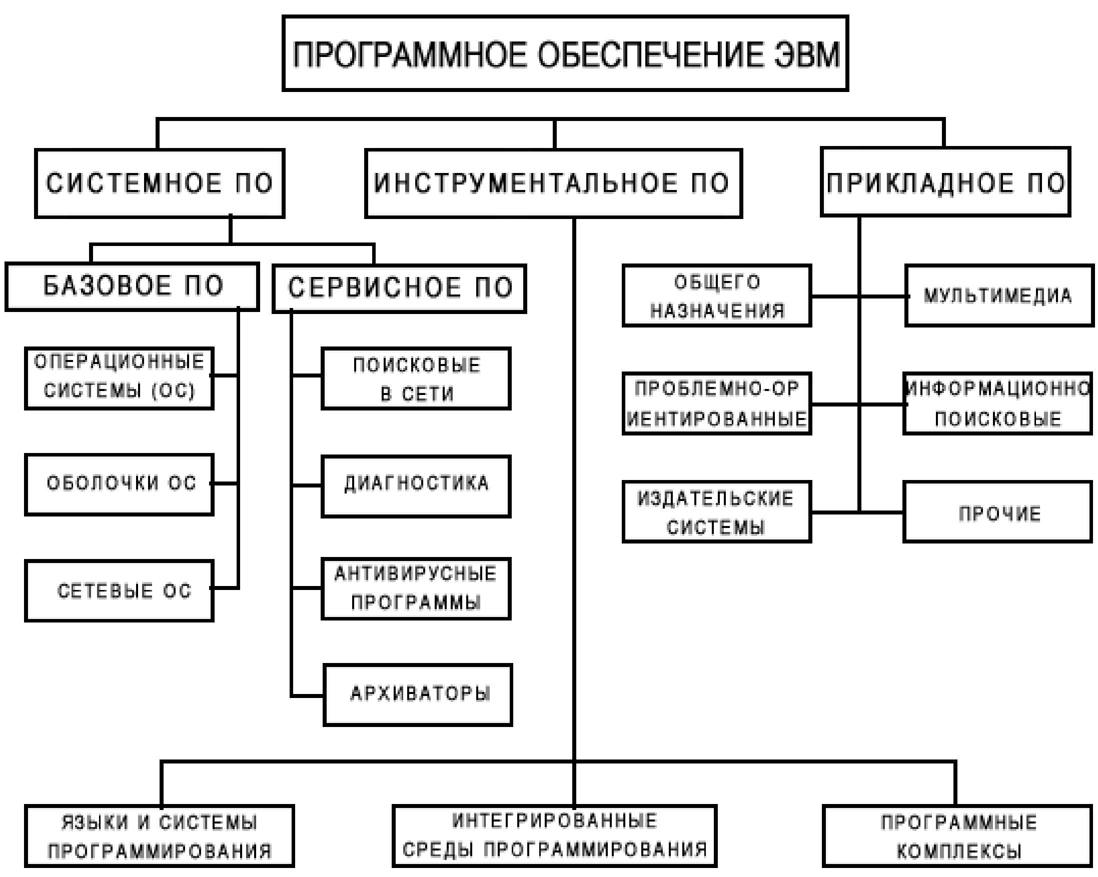

Рис. 11. Классификация программного обеспечения

Различают следующие виды антивирусных программ:

- программы-детекторы;
- программы-доктора или фаги;
- программы-ревизоры;
- программы-фильтры;
- программы-вакцины или иммунизаторы.

Программы-детекторы осуществляют поиск характерной для конкретного вируса последовательности байтов (сигнатуры вируса) в оперативной памяти и в файлах, и при обнаружении выдают соответствующее сообщение. Недостатком таких антивирусных программ является то, что они могут находить только те вирусы, которые известны разработчикам таких программ.

Программы-доктора или *фаги*, а также программы-вакцины не только находят зараженные вирусами файлы, но и «лечат» их, т.е. удаляют из файла тело программы вируса, возвращая файлы в исходное состояние. В начале своей работы фаги ищут вирусы в оперативной памяти, уничтожая их, и только затем переходят к «лечению» файлов. Среди фагов выделяют полифаги, т.е. программы-доктора, предназначенные для поиска и уничтожения большого количества вирусов. Наиболее известными полифагами являются программы

Aidstest, Scan, Norton Anti Virus и Doctor Web. Учитывая, что постоянно появляются новые вирусы, программы-детекторы и программы-доктора быстро устаревают, и требуется регулярное обновление их версий.

Программы-ревизоры относятся к самым надежным средствам защиты от вирусов. Ревизоры запоминают исходное состояние программ, каталогов и системных областей диска тогда, когда компьютер не заражен вирусом, а затем периодически или по желанию пользователя сравнивают текущее состояние с исходным. Обнаруженные изменения выводятся на экран видеомонитора. Как правило, сравнение состояний производят сразу после загрузки операционной системы. При сравнении проверяются длина файла, код циклического контроля (контрольная сумма файла), дата и время модификации, другие параметры. Программы-ревизоры имеют достаточно развитые алгоритмы, обнаруживают степс-вирусы и могут даже отличить изменения версии проверяемой программы от изменений, внесенных вирусом. К числу программ-ревизоров относится широко распространенная в России программа ADinf фирмы «Диалог-Наука».

Программы-фильтры, или «сторожа», представляют собой небольшие резидентные программы, предназначенные для обнаружения подозрительных действий при работе компьютера, характерных для вирусов (попытки коррекции файлов с расширениями COM и EXE; изменение атрибутов файлов; прямая запись на диск по абсолютному адресу; запись в загрузочные сектора диска; загрузка резидентной программы).

Вакцины, или *иммунизаторы,* – это резидентные программы, предотвращающие заражение файлов. Вакцины применяют, если отсутствуют программы-доктора, «лечащие» этот вирус. Вакцинация возможна только от известных вирусов. Вакцина модифицирует программу или диск таким образом, чтобы это не отражалось на их работе, а вирус будет воспринимать их зараженными и поэтому не внедрится. В настоящее время программы-вакцины имеют ограниченное применение.

Своевременное обнаружение зараженных вирусами файлов и дисков, полное уничтожение обнаруженных вирусов на каждом компьютере позволяют избежать распространения вирусной эпидемии на другие компьютеры.

Инструментальное ПО используется для создания программных продуктов в любой области, включая и системные программы. В настоящее время для создания программных продуктов используются мощные системы визуального программирования, которые включают в себя обширные библиотеки стандартных программ, специальные средства отладки и тестирования.

Системы программирования предназначены для разработки новых программ на конкретном языке программирования и включают в себя компилято-

ры, интерпретаторы, диалоговую среду, редакторы текстов, библиотеки стандартных подпрограмм, компоновщики, отладчики, справочные службы и т.д.

Компилятор (составитель, собиратель) выполняет преобразование исходного текста программы, написанного на языке высокого уровня, в машинный язык, язык близкий к машинному, или в объектный модуль. Он создает законченный вариант программы на машинном языке (exe-файл), который потом и выполняется ЭВМ. Процесс работы компилятора называется компиляцией.

Интерпретатор (толкователь, устный переводчик) переводит и выполняет программу строка за строкой.

Диалоговая среда – средство взаимодействия пользователя и ЭВМ.

Редактор текста – программа, выполняющая набор, корректировку и печать текстов.

Библиотеки стандартных подпрограмм – это совокупность программ, составленных на одном из языков программирования и предназначенных для выполнения узкого класса задач.

Отладчики – программные средства, выполняющие отладку и проверку готовых программ, поиск алгоритмических и семантических ошибок в программе и тестирование программ.

Компоновщик (также редактор связей, линкер – от англ. link editor, linker) – программа, которая производит компоновку – принимает на вход один или несколько объектных модулей и собирает по ним исполняемый модуль.

Справочная служба – набор программ, хранящих справочную и пояснительную информацию.

SDK (от англ. Software Development Kit) или «devkit» – комплект средств разработки, который позволяет специалистам по программному обеспечению создавать приложения для определённого пакета программ, программного обеспечения базовых средств разработки, аппаратной платформы, компьютерной системы, видеоигровых консолей, оперативных систем и прочих платформ. Программист, как правило, получает SDK непосредственно от разработчика целевой технологии или системы. Часто SDK распространяется через Интернет. Многие SDK распространяются бесплатно для того, чтобы поощрить разработчиков использовать данную технологию или платформу. Поставщики SDK иногда подменяют термин Software в словосочетании Software Development Kit на более точное слово. Например, «Microsoft» и «Apple» предоставляют Driver Development Kits (DDK) для разработки драйверов устройств, а «PalmSource» называет свой инструментарий для разработки «PalmOS Development Kit (PDK)».

Ассемблер (от англ. assembler – рабочий-сборщик) – компьютерная программа, компилятор исходного текста программы, написанной на языке ассемблера, в программу на машинном коде.

Генератор документации – программа или пакет программ, позволяющих получать документацию, предназначенную для программистов и/или для конечных пользователей системы, особым образом комментированному исходному коду и, в некоторых случаях, по исполняемым модулям (полученным на выходе компилятора).

Средства анализа покрытия кода – мера, используемая при тестировании программного обеспечения. Она показывает процент тестирования исходного кода программы.

Средства непрерывной интеграции. Непрерывная интеграция (англ. Continuous Integration) – это практика разработки программного обеспечения, которая заключается в выполнении частых автоматизированных сборок проекта для скорейшего выявления и решения интеграционных проблем. В обычном проекте, где над разными частями системы разработчики трудятся независимо, стадия интеграции является заключительной. Она может непредсказуемо задержать окончание работ. Переход к непрерывной интеграции позволяет снизить трудоёмкость интеграции и сделать её более предсказуемой за счет наиболее раннего обнаружения и устранения ошибок и противоречий.

Средства автоматизированного тестирования, системы управления версиями, парсеры и генераторы парсеров и др. В настоящее время для создания программных продуктов используются интегрированные среды разработки.

Интегрированная среда разработки программного обеспечения (англ. IDE, Integrated development environment) – система программных средств, используемая программистами для разработки программного обеспечения. Обычно среда разработки включает в себя текстовый редактор, компилятор и/или интерпретатор, средства автоматизации сборки и отладчик. Иногда также содержит систему управления версиями и разнообразные инструменты для упрощения конструирования графического интерфейса пользователя. Многие современные среды разработки также включают браузер классов, инспектор объектов и диаграмму иерархии классов – для использования при объектно-ориентированной разработке ПО. Хотя существуют среды разработки, предназначенные для нескольких языков – такие как Eclipse или Microsoft Visual Studio – обычно среда разработки предназначается для одного определённого языка программирования – как, например, Visual Basic. Примеры сред разработки: Eclipse, Sun Studio, Turbo Pascal, Borland C++, GNU toolchain, DrPython, Borland Delphi, Dev-C++, Lazarus, KDevelop, QDevelop, QNX Momentics IDE,

XCode. Частный случай ИСР – среды визуальной разработки, которые включают в себя возможность визуального редактирования интерфейса программы.

Прикладное ПО делится на программы общего и специального назначения, которые также можно классифицировать по применению: проблемно-ориентированные программы, программы автоматизированного проектирования, общие, методо-ориентированные программы, офисные, настольные издательские системы, мультимедиа, системы искусственного интеллекта.

Прикладные программы общего назначения – это программы, предназначенные для решения каких-либо задач, не привязанных к конкретной предметной области и используемых большинством пользователей компьютеров. К таким программам относятся:

- текстовые редакторы и процессоры;
- электронные таблицы;
- графические редакторы;
- обозреватели;
- программы для просмотра изображений, воспроизведения аудио- и видеофайлов;
- системы управления БД;
- компьютерные игры;
- переводчики;
- и другие.

Прикладные программы специального назначения – это программы, предназначенные для решения задач из каких-либо предметных областей. К таким программам относятся:

- настольные издательские системы;
- электронные энциклопедии;
- видео, аудио и другие редакторы;
- экспертные системы;
- программы, реализующие математические, статические и другие методы решения задач;
- САПР;
- геоинформационные системы;
- программы для организаций;
- и другие.

Рассмотрим подробнее *программы, реализующие математические, статические и другие методы решения задач*. Среди программ так называемой «компьютерной математики» (математических пакетов) наиболее распространены следующие:

– Maple (ориентирован на широкий круг пользователей, алгоритм вычислений задается записью на входном языке пакета соответствующих математических формул, при вводе сложных выражений это вызывает определенные затруднения);
– MathCad (мощная система компьютерной математики, сочетающая в себе визуально ориентированный входной язык, удобный редактор текста и формул, численный и символьный процессоры);
– MatLab (одна из старейших систем компьютерной математики, является одним из наиболее мощных математических пакетов, сочетающих удобную оболочку, редактор, вычислитель и графический программный процессор);
– Matematica (позволяет производить численные и аналитические вычисления, объединять последовательности вычислений в программы, создавать диалоговые документы, объединяющие активные формулы, тексты, живые графики и звук);
– SciLab (система компьютерной математики для выполнения инженерных и научных вычислений, по возможностям пакет Scilab практически не уступает Mathcad, а по интерфейсу близок к Matlab);
– и другие.

Наиболее популярными программными пакетами для статистического анализа и построения графиков являются:

– Statistica (реализует функции анализа данных, управления данных, добычи данных, визуализации данных с привлечением статистических методов, обладает широкими графическими возможностями, позволяет выводить информацию в виде различных типов графиков (включая научные, деловые, трёхмерные и двухмерные графики в различных системах координат, специализированные статистические графики – гистограммы, матричные, категорированные графики и др.), все компоненты графиков настраиваются);
– Origin Pro (предназначен для численного анализа данных и научной графики; для выполнения операций можно спользовать как инструмент графического интерфейса пользователя (диалоги/меню), так и вызывать их в программах; включён собственный компилятор C/C++ с поддержкой и оптимизацией векторных и матричных вычислений; есть возможность создания двумерной, трёхмерной научной графики, которая создаётся с помощью готовых шаблонов, доступных для редактирования пользователем (создания новых); можно проводить численный анализ данных, включая различные статистические операции, обработку сигналов и т. п.);

- RLPlot (открытый удобный и очень компактный пакет для научной 2D презентации данных, имеет интерактивный графический пользовательский интерфейс, работает с таблицами, позволяет свободно редактировать графики, поскольку включает векторный редактор);
- и другие.

Стоит еще отметить on-line базу знаний и набор команд Wolfram|Alpha, которая обладает способностью решать очень широкий спектр математических задач: решение уравнений и неравенств и их систем, построение графиков, дифференцирование и интегрирование и др.

Лекция 1.9
Функции на C/C++

Как говорилось ранее, алгоритм состоит из различных базовых структур, которые могут быть представлены вспомогательными алгоритмами. Вспомогательные алгоритмы на определенных языках программирования описываются с помощью функций (или процедур), к которым потом можно обращаться в тексте основной функции.

Функция – это совокупность объявлений и операторов, обычно предназначенных для решения определенной задачи. Каждая функция имеет имя, которое используется при объявлении, определении и вызове функции. Имя главной функции программы C/C++ main.

Имя функции имеет структуру

<имя функции>([<параметры>]).

Функция может возвращать некоторое (одно!) значение. Перед именем функции необходимо указывать тип значения, которое функция возвращает. Оператор возврата return <выражение>; Если функции не требуется возврата какого-либо значения, то необходимо указать тип void. Ниже приведены примеры функций. Первая функция находит максимальное значение из двух целых чисел и поэтому возвращает значение целого типа. Вторая функция совершает обмен значениями двух переменных, возврата при этом не требуется и, следовательно, функция типа void. К параметрам второй функции обращение происходит по ссылке для того, чтобы при выходе из функций переменные x и y сохранили свои измененные значения. Подробнее работа с ссылками будет рассмотрена позже в одной из следующих лекций.

Примеры:

```
int max( int x, int y)          void swap( int &x, int &y)
{                               {
  if (x>=y) return x;             int r=x;
  else return y;                  x=y; y=r;
}                               }
```

Прежде чем обратиться к функции, она должна быть описана (определена). В некоторых языках программирования (в том числе и C/C++) функцию можно описывать после ее использования, в этом случае в начале программы необходимо указать ее прототип (объявление функции). *Объявление* функции выглядит следующим образом:

<имя функции>([параметры]);

Прототипы библиотечных функций находятся в заголовочных файлах (#include <*.h>). Прототипы функций, разработанных в отдельных файлах, – (#include "*.h").

При *определении* функции задается тип возвращаемого значения, имя функции, типы и количество формальных параметров, класс памяти. В тело функции включено объявление переменных и операторы, определяющие действие функции. Например, описана функция поиска минимального целого числа из двух:

```
int max (int a, int b)
{
  if (a>=b) return a;
  else return b;
}
```

Вызов функции осуществляется использованием ее имени в операторе присваивания (если функция возвращает значение) или с помощью <имя функции>([<фактические параметры>] (если функция ничего не возвращает).
Примеры:

```
M = max( a, b);          swap( x, y);
M= max( 2*x, 3*y);       clear();
N=a*max( a, b)-1;        printf("a=%d\n", a);
```

Рекурсия

Рекурсивной функцией является функция, которая вызывает сама себя. Если вызов функции происходит из нее же самой (непосредственно), то такую рекурсию называют простой. Если вызов функции происходит через другие функции, то такую рекурсию называют сложной или косвенной рекурсией. Количество вложенных вызовов функции или процедуры называется глубиной рекурсии.

Можно сказать, что все задачи, решаемые итеративно, можно решить рекурсивно. Только это не всегда эффективно, т. е. в каждом отдельном случае решение остается за программистом. Ранее были рассмотрены примеры реализации задачи нахождения факториала с использованием циклов, ниже приведен пример нахождения факториала с использованием рекурсии.

Пример. Нахождение факториала числа n.

```
long factorial( int n)
{
  if (n==0 || n==1) return 1;
  return n*factorial(n-1);
}
```

Каждый раз функция вызывает саму себя с параметром на 1 меньше, таким образом, получается

n*factorial(n-1) -> n*(n-1)*factorial(n-2) -> n*(n-1)*(n-2)*factorial(n-3)->…->

n*(n-1)*(n-2)*…*factorial(0) -> n*(n-1)*(n-2)*…*1 – это есть n!.

С помощью рекурсивной функции эффективно реализуются рекуррентные соотношения. Рекуррентные соотношения – это формулы для вычисления элемента какой-либо последовательности, в которых используются значения одного или нескольких предыдущих элементов этой же последовательности. Ярким примером являются числа Фибоначчи: 1, 1, 2, 3, 5, 8, … Каждый следующий элемент последовательности Фибоначчи вычисляется как сумма двух предыдущих элементов. Рассмотрим реализацию этого примера рекурсивно.

Пример:

```
int fib(int n)
{
   if (n==0 || n==1) return 1;
   else return fib(n-1)+fib(n-2);
}
```

Придумайте самостоятельно рекуррентное соотношение, задающее следующие числовые последовательности и реализуйте его с помощью рекурсивной функции C/C++:

а) 1, 2, 3, 4, …

б) 0, 5, 10, 15, …

в) 1, 1, 1, 1, …

г) 1, -1, 1, -1, …

д) 1, -2, 3, -4, 5, -6, …

е) 2, 4, 8, 16, …

ж) 2, 4, 16, 256, …

з) 0, 1, 2, 3, 0, 1, 2, 3, 0, …

и) 1!, 3!, 5!, 7!, …

к) 1, a, a^2, a^3, …

Математические функции

В C/C++ разработана библиотека встроенных математических функций <math.h>, которую необходимо подключать в начале программы. Ниже в таблице приведены основные математические функции из данной библиотеки.

Ниже приведен пример реализации задачи нахождения площади треугольника по введенным с клавиатуры значениям двух сторон и угла между ними. Используется математическая функция sin.

Пример:

```
#include<stdio.h>
#include< math.h >
void main()
{
  float a, b, alfa, S;
  printf("\nвведите значения двух сторон треугольника (см) и угла между ними (рад):");
  scanf("%f%f%f", &a,&b,&alfa);
 S=(a*b*sin(alfa))/2;
  printf(  "S=%.2f ", S);
}
```

Таблица 12. Основные математические функции библиотеки math.h

Обращение	Тип арг-та	Тип рез-та	Функция
abs(x)	int	int	абсолютное значение целого числа
acos(x)	double	double	арккосинус (радианы)
asin(x)	double	double	арксинус (радианы)
atan(x)	double	double	арктангенс (радианы)
ceil(x)	double	double	ближайшее целое не меньше x
cos(x)	double	double	косинус (x в радианах)
exp(x)	double	double	e^x – экспонента от x
fabs(x)	double	double	абсолютное значение вещественного x
log(x)	double	double	логарифм натуральный – ln x
log10(x)	double	double	логарифм десятичный – lg x
pow(x,y)	double double	double	x в степени y – x^y
sin(x)	double	double	синус (x в радианах)
sinh(x)	double	double	гиперболический синус
sqrt(x)	double	double	корень квадратный
tan(x)	double	double	тангенс (x в радианах)
tanh(x)	double	double	гиперболический тангенс

Классы памяти

Все переменные в программе характеризуются не только типом, но и классом памяти. В языке С существует четыре класса памяти для переменных: автоматический (automatic), регистровый (register), статический (static) и внешний (external).

Автоматические переменные в программе можно описать следующим образом:

auto char c; auto int x = 0;

Если мы этим не пользовались, то только потому что опущенный описатель auto используется по умолчанию. Зона действия автоматической переменной ограничена блоком или функцией, где она описана. Она начинает существовать после обращения к функции и исчезает после выхода из нее. Таким образом, автоматические переменные не занимают область в памяти. Значение автоматической переменной не может быть изменено другими функциями и в этих функциях могут находиться переменные с таким же именем.

Регистровые переменные объявляются в программе с помощью ключевого слова register и по замыслу автора языка С должны храниться в сверхбыстрой памяти ЭВМ – регистрах.

В целом операции с регистровыми переменными выполняются гораздо быстрее, чем с переменными, сохраненными в памяти. Фактически, когда значение переменной содержится в процессоре, не требуется доступа к памяти для определения или модификации значения. Это делает регистровые переменные идеальным средством для управления циклами. Ниже приведен *пример* объявления регистровой переменной типа int и дальнейшего ее использования для управления циклом:

```
int factorial (register int m)
{
register int temp;
temp = 1 ;
for( m; m>=1;m--) temp *= m;
return temp;
}
```

В данном примере m и temp объявлены как регистровые переменные, поскольку они используются в цикле. Обычно регистровые переменные используются там, где они принесут наибольшую пользу, т. е. в местах, где делается много ссылок на одну и ту же переменную. Это важно, поскольку не все переменные можно оптимизировать по времени доступа. Однако использование таких переменных крайне редко в последнее время.

Статические переменные, подобно автоматическим, локальны в той функции или блоке, где они описаны. Разница заключается в том, что статические переменные не исчезают, когда функция (блок) завершает работу, и их значения сохраняются для последующих вызовов функции. Описание статических переменных выглядит так:

static char c; static int a=1;

Рассмотрим *пример*, в котором переменная объявлена как статическая:

```
#include <stdio.h>
plus1()
{
static int x=0;
x=x+1;
printf("x=%d\n",x);
}
  main()
{
plus1();
plus1();
plus1();
}
```

Начальное значение, равное нулю, присваивается переменной x только один раз. Затем в программе main функция plus1() несколько раз запускается, так как при каждом запуске функции аргумент x не изменяется, а оставляет значение из предыдущей функции. Таким образом, повторение функции plus1 обеспечивает увеличение переменной x на 1 при каждом запуске 1, 2, 3 ...

Внешние переменные вводятся как нечто противоположное автоматическим. Это глобальные переменные, и к ним можно обращаться с помощью имен из любой функции. Поскольку внешние переменные доступны везде, их можно использовать для связи между функциями, не пренебрегая механизмом формальных параметров. Внешние переменные могут определяться вне какой-либо функции; при этом выделяется фактическая память. В любой другой функции, обращающейся к этим переменным, они должны описываться; делается это явно с помощью описателя extern. Все внешние переменные размещают в начале исходного модуля (вне всяких функций!), опуская дополнительные описания со словом extern внутри функций. Конечно, если внешняя переменная и функция, которая ее использует, размещены в разных файлах, описывать эту переменную в функции необходимо. Но самым важным способом является описание каждой внешней переменной с ключевого слова extern в любой функции, которая ее использует. А еще лучше избегать применения внешних переменных, так как они часто служат источником трудно обнаруживаемых ошибок.

Пример:

```
int var                    //описана внешняя переменная var
main()
{extern int var;    // та же внешняя переменная var
…}
```

```
func1()
{extern int var1;   //новая внешняя переменная var1
…      }                          //внешняя var здесь также видна
func2()
{…             //здесь переменная var видна, а переменная
}                        // var1 – не видна
int var1;      //глобально описана переменная var1
func3()
{ int var;     //здесь var – локальная переменная,
…      }                //видна внешняя переменная var1
func4()
{auto int var1;         //здесь var1 – локальная переменная,
…                               //видна внешняя глобальная var
```

Начиная с версии C++ 11 появился еще один описатель класса памяти для переменных – thread_local. Переменная, объявленная с описателем thread_local, доступна только в том блоке, в котором она была создана. Переменная создается при создании блока и уничтожается при его уничтожении. Каждый блок имеет свою собственную копию переменной. Описатель thread_local можно использовать совместно с static или extern. Его можно применять только в объявлениях и определениях данных и нельзя использовать в объявлениях и определениях функций.

Классы памяти для функций: extern и static. По умолчанию все функции являются внешними, т. е. доступными всем элементам программы (проекта). static делает функцию видимой только внутри своего модуля.

Лекция 1.10

Структуры данных в C/C++

Основные простые типы данных были рассмотрены в лекции 1.6 данного пособия. Рассмотрим основные структуры данных. Самой распространенной и присутствующей во всех языках программирования структурой данных являются *массивы*.

Массив – это регулярная (все его компоненты одного типа) структура со случайным доступом (каждый компонент может выбираться произвольно и доступен одинаково с остальными компонентами). Компоненты массива обычно называют *элементами* массива. Доступ к элементу массиву обеспечивается *индексом*. Для использования индекса используется переменная, обычно целого типа. Например, чтобы обратиться к пятому элементу массива, необходимо

написать A[5] или A[4] (если нумерация элементов начинается с нуля). Если надо обратиться к *i*-му элементу массива, то

```
int i;
float A[50];
i = 5;
A[i] = 15.5;
```

Массивы могут быть одномерными, двумерными, трехмерными и т. д. Подробнее работа с массивами будет рассмотрена при изучении языка программирования С/С++.

Еще одной распространенной структурой данных, которая встречается в большинстве языков программирования, является *строка*. В некоторых языках программирования под строку отводится тип, в некоторых строка рассматривается как массив символов. В С/С++ со строками можно работать и как с массивом символов:

char S[50]; (объявлена строка из 50 символов),

и как с объектом шаблонного класса:

string s; (объявлен объект класса – строка, заранее не известно, сколько памяти он будет занимать).

Работа со строками похожа на работу с массивами. В С/С++ для работы со строками созданы специальные функции (для сравнения, копирования и др.), которые находятся в библиотеках среды программмирования.

Еще одной важной структурой, которая также встречается в большинстве языков программирования, является файл. *Файл* – это специально организованная структура данных, распознаваемая компьютером как единое целое. В основном речь идет о текстовых файлах (расширение *.txt). Однако в разных языках программирования организована работа с файлами и других типов. Например, в С++ есть возможность работать с бинарными файлами (имеют расширение *.dat).

В разных языках программирования встречаются еще такие структуры данных, как записи, множества, структуры, динамические структуры, объединения, перечисления и другие. В рамках данного пособия при изучении языка программирования С/С++ будут подробно изучены структуры (в том числе и динамические), объединения и перечисления. Пока ограничимся определениями и простыми примерами.

Структурой называется набор элементов, которые могут быть разного типа.

Пример: структура студент.

```
struct student {
char name[50];
```

```
int group;
};
```

Из структур можно организовать массив. Например, student st[50]; (объявлен массив из 50 структур типа student).

Объединением называется набор элементов одного или разного типа, каждый раз доступ возможен только к одному из элементов объединения. Например, с помощью объединения можно задать способы измерения расстояния и каждый раз использовать только один из способов:

```
union distance
{
  int miles;
  long meters;
};
```

Перечислением называется набор элементов обычно одного типа, означающий варианты возможных значений для переменных типа этого перечисления. Например, с помощью перечислимого типа можно задать варианты цветов обуви:

```
enum color {black, white, red, blue, green, yellow};
struct shoes {
   char type[50];
   color col;
   float size;
};
```

Все описанные структуры рассмотрены подробно в лекциях 1.12, 2.1 и 2.2.

Указатели и ссылки в C/C++

Указатель – это переменная, которая содержит адрес некоторого объекта. Здесь имеется в виду адрес в памяти компьютера. В программе указатели объявляются следующим образом:

тип *имя_переменной;

С указателями связаны два оператора:

& – взятие адреса,

* – возврат значения по указанному адресу.

Пример:

```
int x=10;
int *p, *g; //определение указателей
```

p = &x; //берём адрес переменной x

g = p; //создаём ещё один указатель который тоже указывает на x

cout << *g; //печатаем значение переменной x

Возможна ситуация, когда указатели ссылаются на указатели. *Например*,

int **point;

Указатель Указатель Переменная

Адрес >Адрес >Значение

Соответственно, чтобы получить значение переменной, необходимо написать: **point.

После того как указатель был объявлен, но до того, как ему было присвоено какое-либо значение, указатель содержит неизвестное значение. Поэтому попытка использовать указатель до присвоения ему значения является неприятной ошибкой, так как она может нарушить работу не только программы, но и системы. Принято считать, что указатель, который указывает в никуда, должен иметь значение NULL или 0.

Операции над указателями. Кроме операций присваивания над указателями можно производить операции сложения и вычитания, но они имеют свои специфические особенности.

Рассмотрим следующий *пример*:

int *p, x;

p = &x;

p++; //увеличиваем значение, расположенное по адресу p

В этом примере p увеличится не на 1, а на 2, так как x носит переменную типа int. И это правильно, поскольку новое значение должно указывать не на следующий адрес в памяти, а на адрес следующего целого.

Над указателем можно применять основные операции сравнения.

Например, p < g означает, что адрес, находящийся в p меньше адреса, находящегося в g.

Общая формула для вычисления значения указателя после выполнения операции сложения следующая:

p = p+n*количество_байт_памяти_базового_типа_указателя,

где p – указатель, n – число, на которое происходит увеличение.

Преобразование типов в указателях. Если указатель определён одним типом, а переменная, на которую он будет указывать, другого типа, то необходимо перед присваиванием адреса обязательно выполнять преобразование типов.

float x;

int *p;

p = (int*)&x;

При подобных присвоениях необходимо следить за размерами переменных. Если переменная с более длинного типа будет преобразовываться на тип с меньшими размерами, то значение переменной будет урезаться. *Например*, предыдущий код будет неверен, так как истинное значение переменной теряется (float = 4 байтам, а int = 2 байта).

Тип void и указатели. Нельзя создавать переменную типа void, но можно создавать указатель на тип void. Указателю на void можно присвоить указатель любого другого типа. Однако при обратном присваивании необходимо использовать явное преобразование указателя на void.

Пример:

```
void *p;
float f, *pf;
pf = &f;
p = pf;
pf = (float*)p;
```

Массивы и указатели. Существует связь между указателями и массивами. Имя массива – это адрес памяти, с которого расположен массив, т.е. адрес первого элемента массива. *Например*,

```
int plus[10];
```

Здесь переменная plus является указателем на массив, точнее, на первый элемент массива. Следующие две операции идентичны:

```
p = plus;
p = &plus[0];
```

Арифметические операции над указателями выполняются быстрее, чем над массивами, если мы работаем с подряд идущими элементами. *Например*, в следующих двух идентичных записях, где происходит ссылка на 6-й элемент массива, второй вариант предпочтительнее:

```
plus[5];
*(plus+5);
```

Для динамического выделения памяти во время работы программы под массив используется оператор new, например:

```
int *a = new int[N];
if (a == 0) {cout << “out of memory” << endl; rturn 0;}
```

Аналогично память можно выделять и под многомерные массивы, *например*:

```
int **malloc2d (int r, int c){
int **t= new int*[r];
for (int i = 0; i < r; i++)
t[i] = new int[c];
```

```
return t;
}
int *a = malloc2d (M*N*sizeof(int));
```
приведённый выше код создаст массив как массив указателей на массивы. После создания переменной с помощью созданной функции будет возможность ссылаться на элемент массива как a[i][j].

Ссылка – это простой ссылочный тип, по сути тот же указатель, но менее мощный и более безопасный. Ссылки как тип можно использовать только в C++. Объявление ссылки производится следующим образом:

```
int &x;
```

Отличием от указателей является то, что ссылку не надо разыменовывать.

Рассмотрим два примера, которые совершают одни и те же действия, только в первом используются указатели, а во втором ссылки.

Примеры:

```
…
int x=5;
int *y = &x;
int z = *y;
…
```

```
…
int x=5;
int &y = x;
int z = y;
…
```

В обоих примерах создана переменная целого типа x со значением 5. В первом примере указатель *y (адрес) принимает значение адреса переменной x. Во втором примере создана ссылка &y, которой присваивается значение переменной x. Далее в первом примере переменной целого типа z присваивается значение, находящееся по адресу, на который ссылается указатель y (значение переменной x); во втором примере переменной целого типа z присваивается значение, находящееся по адресу ссылки y (значение переменной x).

Указатели и ссылки как параметры функции

В лекции 1.9 рассмотрено использование функций в C/C++. Нам уже известно, что функция может иметь входные параметры. Существуют три способа создания входных параметров функции: параметры как обычные перемен-

ные, параметры – указатели и параметры ссылки. Если в функции используются параметры как обычные переменные, то при вызове функции в нее подаются только значения, хранящиеся во входных переменных без указания адресов, и функция не способна поменять значения самих переменных. А если входными параметрами являются указатели или ссылки, то функция работает не со значениями, а с адресами и способна поменять значения переменных, находящихся по входным адресам. Особенно это востребовано в случае, когда нужен возврат более одного значения в функции. Например, при работе с массивами, когда функция должна поменять несколько или все элементы массива. Ниже приведен пример реализации разных вариантов входных параметров функции.

Пример:

```
#include <conio.h>
#include <iostream.h>

int sum_x1(int x)
{
  x++;
  return x;
 }

 int sum_x2(int &x)
 {
   x=x++;
   return x;
  }

 int sum_x3(int *x)
{
   *x=*x++;
    return *x;
 }

void main()
 {
    int X = 10;
    cout<<sum_x1(X)<<endl;
    cout<<X<<endl;
    cout<<sum_x2(X)<<endl;
    cout<<X<<endl;
```

```
    cout<<sum_x3(&X)<<endl;
    cout<<X<<endl;
  }
```

При первом обращении в функцию sum_x1 передается параметр, равный X. Внутри функции этот параметр принимает переменная x, которая является локальной, так как объявлена внутри функции, а значит, всё, что происходит с ней, происходит только в пределах этой функции. При выводе на экран значения sum_x1(X) будет выведено 11, а для значения X будет выведено 10.

При втором обращении в функцию sum_x2(X) в качестве параметра передается ссылка на X. Обращение к ссылке точно такое же, как обращение к переменной, но работа функции при таком вызове происходит по-другому. Так как функция принимает параметр в качестве ссылки, то X теперь как входной, так и выходной параметр. При выводе на экран значения sum_x2(X) и самого значения X выведется 11.

Третий вариант работы функции такой же, как второй, только вызов происходит по-другому: так как указатель – это адрес переменной, то в вызове функции идет обращение к адресу параметра. При выводе на экран значения sum_x3(X) и самого значения X выведется 12.

Массивы. Основные алгоритмы обработки массивов

Вернемся к подробному изучению массивов в C/C++. Как уже говорилось, массив объявляется следующим образом:

тип_данных имя_массива[количество элементов в массиве];

Причем, в C/C++ нумерация элементов массива начинается с нуля, поэтому если количество элементов в массиве равно 50, то индекс последнего элемента равен 49.

Размер массива может явно не указываться. *Например*:

int p[]={2, 4, 6, 10, 1};

p[0]=2, p[1]=4, p[2]=6, p[3]=10, p[4]=1.

Количество элементов в массиве в этом случае можно определить так:

n=sizeof(p)/sizeof(int);

В результате следующего объявления массива

int M[6]={5, 3, 2};

будет создан массив из шести элементов. Первые три элемента получат инициализированные значения. Значения остальных будут либо неопределенными, либо равны нулю, если массив внешний или статический.

При таких объявлениях массива память выделяется статически. Если число элементов массива заранее неизвестно и требуется экономия памяти компь-

ютера, следует выделять память под массив динамически. *Примеры* выделения динамической памяти для одномерного массива:

```
#include <stdlib.h>
#include <stdio.h>
void main()
{int *X;
int i=0,n,A;
scanf("%d",&n);//количество элементов
// выделение динамической памяти
X=(int*)malloc(n*sizeof(int));
for(i=0;i<n;i++)
  scanf("%f",&X[i]);
…
free(X); //освобождение памяти из-под массива
}
```

или

```
#include <iostream.h>
…
float *ptrarray = new float [10];
…
delete [] ptrarray;
```

Примеры выделения динамической памяти для двумерного массива:

```
…
float **a;
int n,m,i,j;
scanf('%d%d",&n,&m);
a=(float**)malloc(n*sizeof(float*));
for (i=0;i<n;i++)
a[i]=(float*)malloc(m*sizeof(float));
…
for (i=0;i<n;i++)
  free(a[i]);
    free(a);
…
```

или

```
…
int n,m,i,j;
cin>>n>>m;
```

```
int **a;
a=new int *[n];
for (i=0;i<n;i++)
a[i]=new int[m];
…
for(i=0;i<n;i++)
    delete[] a[i];
 delete []a;
…
```

Ввод/вывод элементов массива производится поэлементно с использованием циклов. *Например*:

```
…
int X[10], N, i;
cout<<”N=”;
cin>>N; //ввод размера массива
for (i=0; i<N; i++)
{
cout<<”\n X[“<<i<<”]=”; //сообщение о вводе элемента
cin>>X[i]; //ввод элементов массива в цикле
}
…
или
scanf(“%d”, &N);
for (i=0; i<N; i++)
{
  printf(“\n X[%d]=”, i); //сообщение о вводе элемента
scanf(“%d”, &X[i]);   или scanf(“%d”, X+i]);   //ввод элементов массива в цикле
}
…
```

Рассмотрим основные алгоритмы обработки массивов. К таким алгоритмам относят поиск элемента в массиве, удаление элемента из массива, вставка элемента в массив после элемента с заданным номером, сортировка массива.

Поиск элемента массива можно осуществлять по какому-либо критерию (критериям). *Например*, поиск минимального и максимального элемента в массиве выглядит следующим образом:

```
#include <iostream.h>
#include<conio.h>
```

```
void main()
{ int X[]={6,4,9,3,2,1,5,7,8,10};
  int i=0, n, max=0, min=a[0];
  n=sizeof(X)/sizeof(X[0]);
 for(i=0; i<n; i++)
{
   if(X[i]>=max) max=X[i];
   if(X[i]<=min) min=X[i];
}
cout<<“max=“<<max<<endl<<“min=“<<min<<endl;
}
```

Или, *например*, поиск элементов в массиве, кратных числу 5:

```
#include <iostream.h>
#include<conio.h>
void main()
{ int X[]={6,4,9,3,2,1,5,7,8,10};
  int i=0, n, k=0;
  n=sizeof(X)/sizeof(X[0]);
 for(i=0; i<n; i++)
{
   if(X[i]%5==0) k++;
}
cout<<“количество элементов, кратных 5,=“<<k<<endl;
}
```

Удаление элемента из массива производится по принципу смещения значений элементов массива влево начиная со следующего элемента за удаляемым и удаления из памяти последнего элемента (если память под массив выделяется динамически) или его обнуления.

Пример:

```
#include <iostream>
using namespace std;
int main()
{
   int X[10], N, i, m;
   cin>>N; //ввод реального размера массива
   for (i=0; i<N; i++)
      cin>>X[i]; //ввод элементов массива в цикле
```

```
  cin>>m; //ввод номера элемента, подлежащего удалению
  for (i=m; i<N-1; i++)
    X[i]=X[i+1]; //удаление m-го элемента
  for (i=0; i<N-1; i++)
  cout<<"\n X["<<i<<"]="<<X[i]; //вывод массива
  X[N-1]=0;
  N--;
  return 0;
 }
```

Вставка элемента в массив после элемента с заданным номером производится по принципу смещения значений элементов вправо начиная со следующего за заданным элементом и присваивания значения вставляемого элемента элементу, следующему непосредственно за заданным.

Пример:

```
#include <iostream>
using namespace std;
int main ()
{
   int X[10], N, i, m, Y;
  cin>>N; //ввод реального размера массива
  for (i=0; i<N; i++)
    cin>>*(X+i); //ввод элементов массива
  cin>>m; //ввод номера элемента, после которого вставка
  cin>>Y; //вставляемый элемент
  for (i=N; i>m+1; i--)
      *(X+i)=*X+i-1); //сдвиг элементов массива
  *(X+m+1)=Y;
  for (i=0; i<N+1; i++)
      cout<<"\n X["<<i<<"]="<<*(X+i); //вывод
  return 0;
 }
```

Сортировка массива – это процесс упорядочения множества объектов по заданному признаку. Бывает внутренней и внешней. При *внутренней* сортировке все элементы хранятся в оперативной памяти, таким образом, как правило, это сортировка массивов. При *внешней* сортировке элементы хранятся на внешнем запоминающем устройстве, это сортировка содержимого файлов. Одно из

основных требований к методам сортировки – экономное использование памяти.

Сортировку массива можно производить:

- по возрастанию: каждый следующий элемент больше предыдущего: $a[1]<a[2]<\ldots<a[n]$;
- по неубыванию: каждый следующий элемент не меньше предыдущего: $a[1]\leq a[2] \leq\ldots\leq a[n]$;
- по убыванию: каждый следующий элемент меньше предыдущего: $a[1]>a[2]>\ldots>a[n]$;
- по невозрастанию: каждый следующий элемент не больше предыдущего: $a[1]\geq a[2] \geq\ldots\geq a[n]$.

Наиболее популярными являются следующие сортировки:

- метод «пузырька» – метод простого обмена;
- модифицированный «пузырек»;
- шейкер-сортировка;
- сортировка вставками (метод прямого включения);
- сортировка бинарными вставками;
- сортировка методом простого выбора;
- сортировка слияниями;
- быстрая сортировка (сортировка Хоара, 1962);
- пирамидальная сортировка.

Лекция 1.11
Сортировки массивов

Рассмотрим алгоритмы некоторых сортировок массивов.

Метод «пузырька» – метод простого обмена. Производится последовательное упорядочение смежных пар элементов массива: x[1] и x[2], x[2] и x[3], …x[N-1] и x[N]. Если в паре первый элемент больше второго, то элементы меняются местами. Далее таким же образом обрабатываем следующую пару. В итоге после N–1 сравнения максимальное значение переместится на место элемента X[N], т.е. «вверху» окажется самый «легкий» элемент – отсюда аналогия с пузырьком. Следующий проход делается аналогичным образом до второго сверху элемента (X[N–1]), в результате второй по величине элемент поднимется на правильную позицию и т.д. Для сортировки всего массива нужно выполнить N–1 проход по массиву. При первом прохождении нужно сравнить N–1 пар элементов, при втором – N–2 пары, при k-м прохождении – N k пар.

Программа сортировки методом «пузырька»:

```
#include <iostream>
#include <locale.h>
using namespace std;

void input(int *m, int &n)
{
	cout<<”Введите количество элементов массива “;
	cin>>n;
	for (int i=0;i<n;i++)
	{
		cout<<”a[“<<i<<”]=”;
		cin>>m[i];
	}

}
void print (int *m, int n)
{
	for (int i=0;i<n;i++)
		cout<<m[i]<<” “;
	cout<<”\n”;

}
void sort_bubble (int *m, int n)
{
	int i, j, t;
	for (i=0;i<n-1;i++)
		for(j=0;j<n-i-1;j++)
			if (m[j]>m[j+1])
			{
				t=m[j];
				m[j]=m[j+1];
				m[j+1]=t;
			}
}
void main()
{
	setlocale(LC_ALL, “rus”);
	const int nmax=20;
```

```
        int n, a[nmax];
        input(a,n);
        cout<<"Исходный массив:\n";
        print(a,n);
        sort_bubble(a,n);
        cout<<"Отсортированный массив:\n";
        print(a,n);
}
```

Среднее число сравнений и обменов имеют квадратичный порядок роста: O(n2), отсюда можно заключить, что алгоритм пузырька очень медленен и малоэффективен.

Модифицированный «пузырек». Можно улучшить эффективность метода «пузырька», просматривая массив только до тех пор, пока существует обмен между элементами. Как только при очередном просмотре не происходит ни одного обмена между элементами, это означает, что массив отсортирован. При таком подходе может потребоваться один просмотр массива, если массив уже был отсортирован. Поэтому лучше использовать «модифицированный пузырек», который отсортирует за один просмотр.

Программа:

```
void modif_bubble(int *m, int n)
{
 int i=n,t;
 int f;
 do {
  f=0;
  for (int k=0;k<i-1;k++)
   if (m[k]>m[k+1])
       {
    t=m[k]; m[k]=m[k+1]; m[k+1]=t;
    f=1;
       }
  i--;
 } while (f && i>1);
}
```

Шейкер-сортировка. Этот алгоритм уменьшает количество перемещений, действуя следующим образом: за один проход из всех элементов выбирается минимальный и максимальный, минимальный элемент помещается в начало массива, а максимальный соответственно в конец. Далее алгоритм выполняется для остальных данных. Таким образом, за каждый проход два элемента поме-

щаются на свои места, а значит, понадобится N/2 проходов, где N – количество элементов.

Реализация алгоритма шейкер-сортировки:

```
void sort_sheiker (int *m, int n)
{
 int left=1;
 int right=n-1;
 int j,t,k=n-1;
 do
 {
  for (j=right;j>=left; j--)
   if (m[j-1]>m[j])
        {
                t=m[j-1]; m[j-1]=m[j]; m[j]=t;
                k=j;
            }
  left=k+1;
  for (j=left; j<=right; j++)
   if (m[j-1]>m[j])
        {
     t=m[j-1]; m[j-1]=m[j]; m[j]=t;
     k=j;
        }
  right=k-1;
 } while (left<=right);
}
```

Фиксирование индексов последнего обмена при проходах слева направо и справа налево ускоряет «шейкер»-сортировку.

Сортировка вставками (метод прямого включения):

1. Начиная со 2-го элемента, каждый текущий элемент с номером i запоминается в промежуточной переменной.
2. Затем просматриваются предыдущие i–1 элемент и ищется место вставки i-го элемента таким образом, чтобы не нарушить упорядоченности, при этом все элементы, превышающие i-й, сдвигаются вправо.
3. На освободившееся место вставляется i-й элемент.

Программа сортировки вставками:

```
void sort_insert (int *m, int n)
{
    int j,r;
```

```
        for (int i=1;i<n;i++)
        {
                r=m[i];
                j=i-1;
                while (j>=0 && m[j]>r)
                { m[j+1]=m[j]; j--;}
                m[j+1]=r;
        }
}
```

Сортировка бинарными вставками. Алгоритм сортировки простыми включениями можно легко улучшить, пользуясь тем, что готовая последовательность a[1],..., a[i-1], в которую нужно включить новый элемент, уже упорядочена. Поэтому место включения можно найти значительно быстрее, применив бинарный поиск, который исследует средний элемент готовой последовательности и продолжает деление пополам, пока не будет найдено место включения. Модифицированный алгоритм сортировки называется сортировкой бинарными включениями.

Реализация алгоритма бинарными вставками:

```
void sort_bin_insert (int *a, int n)
{ int x,left,right,sred;
  for (int i=1; i<n; i++)
  {
   if (a[i-1]>a[i])
        {
    x=a[i];
    left=0;
    right=i-1;
         do {
      sred = (left+right)/2;
      if (a[sred]<x ) left= sred+1;
       else right=sred-1;
         } while (left<=right);
    for (int j=i-1; j>=left;j--) a[j+1]= a[j];
    a[left]= x;
        }
  }
}
```

Сортировка методом простого выбора. Обычно применяется для массивов, не содержащих повторяющихся элементов.

Алгоритм:

1) выбрать максимальный элемент массива;

2) поменять его местами с последним элементом (после этого самый большой элемент будет стоять на своём месте);

3) повторить пункты 1-2 с оставшимися n–1 элементами.

Программа сортировки методом простого выбора:

```
void sort_vybor (int *a, int n)
{
        int k,m;
        for (int i=n-1;i>=0;i--)
        {
                k=i;
                m=a[i];
                for (int j=0;j<i-1;j++)
                        if (a[j]>m) {k=j; m=a[k];}
      if (k!=i)
           {
       a[k]=a[i];
       a[i]=m;
           }
        }
}
```

Сортировка слияниями. Метод слияний – один из первых в теории алгоритмов сортировки. Он предложен Дж. фон Нейманом в 1945 г. Рассмотрим следующую задачу. Объединить («слить») упорядоченные фрагменты массива A A[k],...A[m] и A[m+1],..., A[q] в один A[k],..., A[q], естественно, тоже упорядоченный ($k < m < q$). Основная идея решения состоит в сравнении очередных элементов каждого фрагмента, выяснении, какой из элементов меньше, переносе его во вспомогательный массив D (для простоты) и продвижении по тому фрагменту массива, из которого взят элемент. При этом следует не забыть записать в D оставшуюся часть того фрагмента, который не успел себя «исчерпать».

Реализация слияния фрагментов:

```
#include <iostream>
#include <locale.h>
using namespace std;
```

```
void Sl(int m[],int k,int q)
{
  int i,j,t,mid,d[20];
  i=k;
  mid=k+(q-k)/2;
  j=mid+1;
  t=1;
  while (i<=mid && j<=q)
  {
        if (m[i]<=m[j])  {d[t]=m[i]; i++;}
        else { d[t]=m[j]; j++;}
   t++;
  }
  while (i<=mid)
  { d[t]=m[i]; i++; t++;}
  while (j<=q)
  {  d[t]=m[j]; j++; t++;}
  for (i=1;i<=t-1;i++) m[k+i-1]=d[i];
 }
void Sort_Sl(int *m, int i,int j)
{
 int  t;
 if (i<j)
   if (j-i==1) {
     if (m[j]<m[i])
          { t=m[i]; m[i]=m[j]; m[j]=t;};}
          else {
     Sort_Sl(m,i,i+(j-i)/2);
     Sort_Sl(m,i+(j-i)/2+1,j);
     Sl(m,i,j);
          }
}
void input(int *m, int &n)
{
        cout<<"Введите количество элементов массива ";
        cin>>n;
        for (int i=0;i<n;i++)
        {
                cout<<"a["<<i<<"]=";
```

```
            cin>>m[i];
        }

}
void print (int *m, int n)
{
    for (int i=0;i<n;i++)
            cout<<m[i]<<” “;
    cout<<”\n”;

}
void main()
{
    setlocale(LC_ALL,»rus»);
    const int nmax=20;
    int n, a[nmax];
    input(a,n);
    cout<<”Исходный массив:\n”;
    print(a,n);
    Sort_Sl(a,0,n-1);
    cout<<”Отсортированный массив:\n”;
    print(a,n);
}
```

Первый вызов процедуры – Sort_Sl(a,0,n-1);, где a – исходный массив из N элементов.

Данный алгоритм работает быстрее, чем, например, «пузырьковый», а недостаток метода – в требовании довольно большого объема дополнительной памяти.

Быстрая сортировка (сортировка Хоара, 1962). Рассмотрим элемент, находящийся посередине массива M (назовем его X). Затем начнем два встречных просмотра массива: двигаемся слева направо, пока не найдем элемент M[i]>X и справа налево, пока не найдем элемент M[j]<X. Когда указанные элементы будут найдены, поменяем их местами и продолжим процесс «встречных просмотров с обменами», пока два идущих навстречу друг другу просмотра не встретятся. В результате массив разделится на две части: левую, состоящую из элементов меньших или равных X, и правую, в которую попадут элементы, большие или равные X. После такого разделения массива M, чтобы завершить его сортировку, достаточно осуществить то же самое с обеими полу-

ченными частями, затем с частями этих частей и т.д., пока каждая часть не будет содержать ровно один элемент.

Реализация указанного метода приводит к следующему (рекурсивному!) решению задачи сортировки:

```
void Qsort(int a[],int L,int R)
{ int i=L,j=R,w,x;
  x=a[(L+R)/2];
  do { while (a[i]<x) i++;
         while (a[j]>x) j--;
         if(i<=j)
        {w=a[i]; a[i]=a[j]; a[j]=w;
               i++; j--;}
  } while (i<j);
  if (L<j) Qsort(a,L,j);
  if (i<R) Qsort(a,i,R);
}
void main()
{
        setlocale(LC_ALL,»rus»);
        const int nmax=20;
        int n, a[nmax];
        input(a,n);
        cout<<'Исходный массив:\n";
        print(a,n);
        Qsort(a,0,n-1);
        cout<<"Отсортированный массив:\n";
        print(a,n);
}
```

Основная программа должна содержать вызов процедуры в виде Qsort(a,0,N-1);

Лекция 1.12
Строки в C/C++

В этой лекции мы рассмотрим строки как массивы символов char и как шаблонный класс string.

Строкой называется массив символов, который заканчивается пустым символом '\0'. Строка объявляется как обычный символьный массив, *например*,

```
char  str1[10];         // строка длиной в девять символов
```

char *str2;   // указатель на строку

Строка при объявлении может быть инициализирована начальным значением, *например*, так:

char string[10] =”abcdefghf”;

Если подсчитать количество символов в двойных кавычках после символа равно их окажется 9, а размер строки 10 символов, последнее место отводится под нуль-терминатор, причём компилятор сам добавит его в конец строки.

Посимвольная инициализация строки:

char string[10] = {'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'f', '\0'};

При объявлении строки необязательно указывать её размер, но при этом обязательно нужно её инициализировать начальным значением. Тогда размер строки определится автоматически и в конец строки добавится нуль-терминатор:

char string[ ] = “abcdefghf”;

Строка может содержать символы, цифры и специальные знаки:

char str1[ ] = “This is a string №1.”;

В языке программирования С для работы со строками существует большое количество функций, прототипы которых описаны в заголовочных файлах stdlib.h и string.h.

Строки и указатели. Строчная константа в языке С ассоциируется с адресом начала строки в памяти, тип строки получается char*(указатель на тип char). Поэтому возможно использовать следующее присваивание:

char *pc;

pc = «Hello world»;

Ввод-вывод строк. Для ввода строки с консоли служит функция

char* gets (char *str);

которая записывает строку по адресу str и возвращает адрес введенной строки. Функция прекращает ввод, если встретит символ ‘\n’ или EOF (конец файла). Символ перехода на новую строку не копируется. В конец прочитанной строки помещается нулевой байт. В случае успеха функция возвращает указатель на прочитанную строку, а в случае неудачи NULL. Для вывода строки на консоль служит стандартная функция

int puts (const char *s);

которая в случае удачи возвращает неотрицательное число, а в случае неудачи – EOF. Прототипы функций gets и puts описаны в заголовочном файле stdio.h.

Пример:

…

char str[80];

```
printf(“Input string: “);
gets(str);
puts(str);
…
```

Существуют варианты функций scanf и printf, которые предназначены для форматирования строк и называются соответственно sscanf и sprintf. Функция

int sscanf (const char *str, const char *format, …);

читает данные из строки, заданной параметром str, в соответствии с форматной строкой, заданной параметром format. В случае удачи возвращает количество прочитанных данных, а в случае неудачи – EOF.

Функция

int sprintf (char *buffer, const char *format, …);

форматирует строку в соответствии с форматом, который задан параметром format и записывает полученный результат в символьный массив buffer. Возвращает функция количество символов, записанных в символьный массив buffer, исключая завершающий нулевой байт.

Функции для работы со строками

Основные функции для работы со строками приведены в табл.13. Для использования этих функций необходимо подключить библиотеки stdlib.h и string.h.

Таблица 13. Функции для работы со строками

Функция	**Описание**
strlen(имя_строки)	Определяет длину указанной строки, без учёта нуль-символа
Копирование строк	
strcpy(s1,s2)	Выполняет побайтное копирование символов из строки s2 в строку s1
strncpy(s1,s2, n)	Выполняет побайтное копирование n символов из строки s2 в строку s1, возвращает значения s1
Конкатенация строк	
strcat(s1,s2)	Объединяет строку s2 со строкой s1. Результат сохраняется в s1
strncat(s1,s2,n)	Объединяет n символов строки s2 со строкой s1. Результат

Функция	Описание
	сохраняется в s1
Сравнение строк	
strcmp(s1,s2)	Сравнивает строку s1 со строкой s2 и возвращает результат типа int: 0 –если строки эквивалентны, >0 – если s1<s2, <0 — если s1>s2, с учётом регистра
strncmp(s1,s2)	Сравнивает n символов строки s1 со строкой s2 и возвращает результат типа int: 0 –если строки эквивалентны, >0 – если s1<s2, <0 — если s1>s2, с учётом регистра
stricmp(s1,s2)	Сравнивает строку s1 со строкой s2 и возвращает результат типа int: 0 –если строки эквивалентны, >0 – если s1<s2, <0 — если s1>s2, без учёта регистра
strnicmp(s1,s2)	Сравнивает n символов строки s1 со строкой s2 и возвращает результат типа int: 0 –если строки эквивалентны, >0 – если s1<s2, <0 — если s1>s2, без учёта регистра
Обработка символов	
isalnum(c)	Возвращает значение true, если c является буквой или цифрой, и false в других случаях
isalpha(c)	Возвращает значение true, если c является буквой, и false в других случаях
isdigit(c)	Возвращает значение true, если c является цифрой, и false в других случаях
islower(c)	Возвращает значение true, если c является буквой нижнего регистра, и false в других случаях
isupper(c)	Возвращает значение true, если c является буквой верхнего регистра, и false в других случаях
isspace(c)	Возвращает значение true, если c является пробелом, и false в других случаях

Функция	**Описание**
toupper(c)	Если символ c, является символом нижнего регистра, то функция возвращает преобразованный символ c в верхнем регистре, иначе символ возвращается без изменений
Функции поиска	
strchr(s,c)	Поиск первого вхождения символа c в строке s. В случае удачного поиска возвращает указатель на место первого вхождения символа c. Если символ не найден, то возвращается ноль
strcspn(s1,s2)	Определяет длину начального сегмента строки s1, содержащего те символы, которые не входят в строку s2
strspn(s1,s2)	Возвращает длину начального сегмента строки s1, содержащего только те символы, которые входят в строку s2
strprbk(s1,s2)	Возвращает указатель первого вхождения любого символа строки s2 в строке s1
Функции преобразования	
atof(s1)	Преобразует строку s1 в тип double
atoi(s1)	Преобразует строку s1 в тип int
atol(s1)	Преобразует строку s1 в тип long int
Функции стандартной библиотеки ввода/вывода <stdio>	
getchar(c)	Считывает символ c со стандартного потока ввода, возвращает символ в формате int
gets(s)	Считывает поток символов со стандартного устройства ввода в строку s до тех пор, пока не будет нажата клавиша ENTER

Основные алгоритмы обработки строк

Так как строка – это массив символов, то и все алгоритмы обработки массивов подходят для строк с некоторой корректировкой. Если производить сортировки строк, то в случае сортировки по возрастанию будет производиться сортировка в алфавитном порядке, в случае сортировки по убыванию – сортировка в порядке, обратном алфавитному.

Пример. Сортировка строки методом «пузырька».

```
#include <locale.h>
#include<stdio.h>
#include<string.h>
int main()
{
        setlocale(LC_ALL,"rus");
         const int nmax=20;
          int n, i, j;
        char t;
        char  a[nmax];
        printf("введите строку\n");
        gets(a);
        n = strlen(a);
        for (i=0; i<n-1; i++)
                for(j=0; j<n-i-1; j++)
                        if (a[j]>a[j+1])
                        {
                                t=a[j];
                                a[j]=a[j+1];
                                a[j+1]=t;
                        }
        printf("Отсортированная строка:\n");
      puts(a);
}
```

Класс string

string – это STL'евский класс, основанный на шаблонах, который входит в стандартную библиотеку C++. Для использования данного класса в ваших приложениях нужно подключить директиву <string>. Класс string предназначен, естественно, для работы со строками. Класс string уступает массивам символов в эффективности. Основные действия со строками выполняются в нем с помо-

щью операций и методов, а длина строки изменяется динамически в соответствии с потребностями. Чтобы объявить объект класса string, необходимо в программе написать следующее:

```
#include <string>
…
string str1;
или
…
string str1(“Hello!”);
или
…
string str1(S);
```

где S – строковая константа.

Чтобы ввести строку с клавиатуры, необходимо воспользоваться потоком cin или функций getline(cin, s);

Пример:

```
#include <string>
#include <iostream>
using namespace std;
int main()
{
  string str1, str2;
  cout<<”enter string: ”
  cin>>str1;
  getline(cin, str2);
  ….
}
```

Основные операции и методы, для работы со строками string приведены в табл. 14.

Таблица 14. Операции и методы класса string

Операция или метод	**Описание**
=	Присваивание
+	Конкатенация
+=	Присваивание с конкатенацией
==	Равенство
!=	Неравенство
<	Меньше

Операция или метод	Описание
<=	Меньше или равно
>	Больше
>=	Больше или равно
[ ]	Индексация
<<	Вывод
>>	Ввод
string &assign (const string &strob, size_type start, size_type num); string &assign(const char *str, size_type num);	Присваивание одной строки другой. Первый формат позволяет присвоить вызывающему объекту num символов из строки, заданной параметром strob начиная с индекса start. При использовании второго формата вызывающему объекту присваиваются первые num символов строки с завершающим нулем, заданной параметром str. В каждом случае возвращается ссылка на вызывающий объект
string &append (const string &strob, size_type start, size_type num); string &append (const char *str, size_type num);	Присоединение части одной строки в конец другой. Здесь при использовании первого формата num символов из строки, заданной параметром strob, начиная с индекса start, будет присоединено в конец вызывающего объекта. Второй формат позволяет присоединить в конец вызывающего объекта первые num символов из строки с заверша-

Операция или метод	Описание
	ющим нулем, заданной параметром str. В каждом случае возвращается ссылка на вызывающий объект
string &insert(size_type start, const string &strob); string &insert(size_type start, const string &strob, size_type insStart, size_type num);	Вставка символов. Первый формат функции insert() позволяет вставить строку, заданную параметром strob, в позицию вызывающей строки, заданную параметром start. Второй формат функции insert() предназначен для вставки num символов из строки, заданной параметром strob начиная с индекса insStart, в позицию вызывающей строки, заданную параметром start
string &replace (size_type start, size_type num, const string &strob); string &replace (size_type start, size_type orgNum, const string &strob, size_type replaceStart, size_type replaceNum);	Замена символов. Первый формат функции replace() служит для замены num символов в вызывающей строке начиная с индекса start, строкой, заданной параметром strob. Второй формат позволяет заменить orgNum символов в вызывающей строке начиная с индекса start replaceNum символами строки, заданной параметром strob начиная с индекса replaceStart. В каждом случае возвраща-

Операция или метод	Описание
	ется ссылка на вызывающий объект
string &erase (size_type start = 0, size_type num = npos);	Удаление символов. Эта функция удаляет num символов из вызывающей строки начиная с индекса start. Функция возвращает ссылку на вызывающий объект.
size_type find (const string &strob,size_type start=0) const; size_type rfind (const string &strob,size_type start = npos) const;	Поиск. Функция find() начиная с позиции start просматривает вызывающую строку на предмет поиска первого вхождения строки, заданной параметром strob. Если поиск успешен, функция find() возвращает индекс, по которому в вызывающей строке было обнаружено совпадение. Если совпадения не обнаружено, возвращается значение npos. Функция rfind() выполняет то же действие, но с конца. Начиная с позиции start она просматривает вызывающую строку в обратном направлении на предмет поиска первого вхождения строки, заданной параметром strob. Если поиск прошел удачно, функция rfind() возвращает индекс, по которому в вызывающей строке было

Операция или метод	Описание
	обнаружено совпадение. Если совпадения не обнаружено, возвращается значение npos
int compare (size_type start, size_type num,const string &strob) const;	Сравнение. Функция compare() сравнивает с вызывающей строкой num символов строки, заданной параметром strob начиная с индекса start. Если вызывающая строка меньше строки strob, функция compare() возвратит отрицательное значение. Если вызывающая строка больше строки strob, она возвратит положительное значение. Если строка strob равна вызывающей строке, compare() возвратит нуль

Лекция 1.13

Понятие и оценка сложности алгоритма

Практически для каждой задачи существует алгоритм решения и чаще всего не один. Если у задачи существует два и более алгоритмов решения, встает вопрос, какой алгоритм лучше? По каким критериям определить этот «наилучший» алгоритм. В настоящее время алгоритм характеризуется своей эффективностью. Эффективность алгоритма (программы) рассматривается в двух аспектах: объем используемой компьютерной памяти при его выполнении и временные затраты. Конечно, кроме написания эффективных алгоритмов, есть и другие пути решения этих проблем: в последнее время компьютерная память стоит дешевле, можно использовать также память чужих компьютеров (работать удаленно); хорошо развита работа суперкомпьютеров, которые за счет большого количества процессоров решают задачи очень быстро. Лучший

способ сравнения эффективностей алгоритмов состоит в сопоставлении сложностей алгоритма.

Сложность алгоритма оценивается функцией зависимости объема работы, выполняемой алгоритмом, от размера входных данных.

Временная сложность алгоритма (в худшем случае) – это функция от размера входных данных, равная максимальному количеству элементарных операций, проделываемых алгоритмом для решения экземпляра задачи указанного размера. Аналогично понятию временной сложности в худшем случае определяется понятие временная сложность алгоритма в наилучшем случае. Также рассматривают понятие среднее время работы алгоритма, т. е. математическое ожидание времени работы алгоритма

По аналогии с временной сложностью определяют *пространственную сложность алгоритма*, только говорят не о количестве элементарных операций, а об объёме используемой памяти.

Время зависит от того, кто является исполнителем (насколько быстро он способен делать операции). Объективная оценка фактического времени, затраченного на выполнение алгоритма и не зависящая от исполнителя, выглядит следующим образом:

$$T_f = \bar{\tau} \cdot k \,,$$

где $\bar{\tau}$ – среднее время одного шага исполнителя, k – количество шагов в алгоритме.

Память S, используемая алгоритмом, удовлетворяет следующей оценке (очень грубой):

$$S \le \mu \cdot k \,,$$

где μ – средний объем памяти, необходимый для выполнения одной операции (шага).

Несмотря на то, что функция временной сложности алгоритма может быть определена точно, в большинстве случаев искать ее точное значение бессмысленно. Дело в том, что, во-первых, точное значение временной сложности зависит от определения элементарных операций (например, сложность можно измерять в количестве арифметических операций, битовых операций или операций на машине Тьюринга), а во-вторых, при увеличении размера входных данных вклад постоянных множителей и слагаемых низших порядков, фигурирующих в выражении для точного времени работы, становится крайне незначительным. Рассмотрение входных данных большого размера и оценка порядка роста времени работы алгоритма приводят к понятию *асимптотической сложности алгоритма*. При этом алгоритм с меньшей асимптотической сложностью

является более эффективным для всех входных данных, за исключением лишь, возможно, данных малого размера.

Порядок сложности алгоритма выражает его эффективность обычно через количество обрабатываемых данных. Например, некоторый алгоритм может существенно зависеть от размера обрабатываемого массива. Если, скажем, время обработки удваивается с удвоением размера массива, то порядок временной сложности алгоритма определяется как размер массива.

Порядок алгоритма – это функция, доминирующая над точным выражением временной сложности. Функция $f(n)$ имеет порядок $O(f(n))$. O-функции выражают относительную скорость алгоритма в зависимости от некоторой переменной (или переменных). Существуют три важных правила для определения сложности.

1. $O(kf) = O(f)$, k – константа.
2. $O(fg) = O(f)O(g)$.
3. $O(f + g)$ равна доминанте $O(f)$ и $O(g)$.

Наиболее часто встречающиеся классы сложности алгоритмов:

- $O(1)$ – большинство операций в программе выполняются только раз или только несколько раз алгоритмами константной сложности. Любой алгоритм, всегда требующий независимо от размера данных одного и того же времени, имеет *константную сложность*. Пример: очищение фиксированной части массива независимо от его размеров.
- $O(\log_2 n)$ – логарифмическая сложность, возникает в алгоритмах, в которых неоднократно подразделяет данные на подсписки. Такое время работы встречается обычно в программах, которые делят большую проблему на маленькие и решают их по отдельности. Пример: работа с бинарными деревьями.
- $O(n \log_2 n)$ – логарифмическая сложность. Такое время работы имеют те алгоритмы, которые делят большую проблему на маленькие, а затем, решив их, соединяют их решения. Примеры: алгоритмы быстрой сортировки, сортировки слиянием.
- $O(N)$ – время работы программы линейно, обычно, когда каждый элемент входных данных требуется обработать лишь линейное число раз, имеет место *линейная сложность.* Пример: для каждого входного данного производится одна операция.
- $O(N^2)$, $O(N^3)$, $O(N^a)$ – *полиномиальная сложность* (квадратичная, кубическая и т.д.). Примеры: алгоритм Дейкстры (нахождение кратчайших путей в графе), алгоритм Прима (построение минимального связывающего дерева), алгоритм Флойда-Уорелла (динамический алгоритм поиска крат-

чайших расстояний между всеми вершинами взвешенного ориентированного графа).

- $O(2^N)$ – *экспоненциальная сложность*. Такие алгоритмы чаще всего возникают в результате подхода, именуемого метод грубой силы. Работа таких алгоритмов очень медленная и применима только при малых N. Эта сложность обычно связана с задачами, требующими неоднократного поиска дерева решений.
- $O(N!)$ – *факториальная сложность*. Пример: определения числа сочетаний, перестановок в комбинаторике.

Алгоритмы без циклов и рекурсивных вызовов имеют константную сложность. Если нет рекурсии и циклов, все управляющие структуры могут быть сведены к структурам константной сложности. Следовательно, и весь алгоритм также характеризуется константной сложностью. Определение сложности алгоритма в основном сводится к анализу циклов и рекурсивных вызовов.

Пример. Рассмотрим алгоритм обработки элементов массива.

```
for (i = 1; i <= N; i++)
{ ... }
```

Сложность этого алгоритма $O(N)$, так как тело цикла выполняется N раз и сложность тела цикла равна $O(1)$.

Если один цикл вложен в другой и оба цикла зависят от размера одной и той же переменной, то вся конструкция характеризуется квадратичной сложностью $O(N^2)$. Эта информация может использоваться программистом для построения более эффективной программы следующим образом. Прежде всего можно попытаться сократить глубину вложенности повторений. Затем следует рассмотреть возможность сокращения количества операторов в циклах с наибольшей глубиной вложенности. Если 90% времени выполнения составляет выполнение внутренних циклов, то 30%-ное сокращение этих небольших секций приводит к 90%*30%=27%-му снижению времени выполнения всей программы.

Понятие оптимизации алгоритмов

Под *оптимизацией алгоритма* понимается его переработка с целью повышения его эффективности (по времени и/или по памяти). Алгоритм, эффективность которого невозможно повысить, называется оптимальным. При решении конкретной задачи оптимальная тактика поведения может состоять в том, чтобы не оптимизировать метод решения, а подключиться к стандартной программе или воспользоваться простейшим методом, составление программы для которого не потребует много усилий. При выборе метода решения задачи исследователь ориентируется на некоторые ее свойства и выбирает алгоритм ре-

шения в зависимости от них, причем алгоритм будет применимым и при решении других задач, обладающих этими свойствами.

Исследование характеристик оптимальных алгоритмов решения задач складывается из двух этапов: построение конкретных методов решения с возможно лучшими характеристиками и получение оценок снизу характеристик вычислительных. По существу, первый этап является основной проблемой теории численных методов и в большинстве случаев рассматривается независимо от проблемы оптимизации. Получение оценок снизу обычно сводится к оценке снизу e-энтропии или поперечников соответствующих пространств; иногда оно проводится независимо, но с использованием техники, аналогичной технике получения указанных оценок.

Вычислительные алгоритмы условно делят на *пассивные* и *активные*. В первом случае алгоритм решения задачи не зависит от получаемой в ходе решения задачи информации, а во втором – зависит.

Понятие сложности задачи, классы сложности задач

Под *задачей* понимается некоторый вопрос (проблема), на который нужно найти (вычислить) ответ. Задачи бывают *общими* и *частными*.

Общая задача определяется:

- списком параметров – свободных переменных, конкретные значения которых неопределенны;
- формулировкой условий – свойств, которыми должен обладать ответ (решение задачи).

Частная задача получается из общей, если всем параметрам общей задачи придать конкретные значения – здесь исходные данные.

Под *сложностью* задачи принято понимать минимальную из сложностей алгоритмов, решающих эту задачу. При разработке алгоритмов можно наблюдать, что для некоторых задач можно построить алгоритм полиномиальной сложности. Такие задачи называют *полиномиальными*. Полиноминально разрешимые задачи можно успешно решать на компьютере и даже в тех случаях, когда они имеют большую размерность.

Для других задач не удается найти полиномиальный алгоритм. поэтому их называют *трудноразрешимыми*. К классу трудноразрешимых задач относится большое число задач алгебры, математической логики, теории графов, теории автоматов и других разделов дискретной математики. В большинстве своем это так называемые *переборные* задачи. Переборная задача характеризуется экспоненциальным множеством вариантов, среди которых нужно найти решение, и может быть решена алгоритмом полного перебора. Переборный алго-

ритм имеет экспоненциальную сложность и может хорошо работать только для небольших размеров задачи. С ростом размера задачи число вариантов быстро растет, и задача становится практически неразрешимой методом перебора. Многие из переборных задач являются экспоненциальными по постановке. Экспоненциальные по постановке задачи не представляют особого интереса для теории алгоритмов, поскольку для них, очевидно, невозможно получить алгоритм полиномиальной сложности. Возникает вопрос: если известно, что некоторая задача алгоритмически разрешима, то неудача в разработке для нее полиномиального разрешающего алгоритма является следствием неумения конкретного разработчика или следствием каких-то свойств самой задачи? Ответ на этот вопрос дает классическая теория алгоритмов, которая классифицирует задачи по сложности. При этом классифицируются лишь распознавательные задачи – задачи, имеющие распознавательную форму. В распознавательной форме суть задачи сводится к распознаванию некоторого свойства, а ее решение – один из двух ответов: «да» или «нет». Существуют задачи, которые изначально имеют распознавательную форму. Например, являются ли два графа изоморфными? Другой пример – задача о выполнимости булевой функции, которая является исторически первой распознавательной задачей, глубоко исследованной в теории алгоритмов. Многие задачи, которые в исходной постановке представлены в иной форме (к ним относятся задачи дискретной оптимизации), довольно просто приводятся к распознавательной форме. Например, если требуется найти хроматическое число графа, то формулируют вопрос: верно ли, что заданный граф является k-раскрашиваемый, где k – заданное целое положительное число. Между тем имеются задачи, которые нельзя привести к распознавательной форме. Это в первую очередь *конструктивные* задачи – задачи на построение объектов дискретной математики, обладающих заданными свойствами: генерация всех подмножеств конечного множества; генерация всех n! различных перестановок; построение плоской укладки графа; построение остовного дерева графа и т. п. Такие задачи могут быть как трудноразрешимыми, так и полиноминально разрешимыми. Они пока не попадают под существующую в теории алгоритмов классификацию.

Под классом сложности задачи понимается множество вычислительных задач, примерно одинаковых по сложности вычисления. В рамках классической теории алгоритмические задачи различаются по следующим основным классам сложности:

- P-сложные (задачи, которые могут быть решены за время, полиномиально зависящее от объема исходных данных, с помощью детерминированной вычислительной машины);

- *NP*-сложные (задачи, которые могут быть решены за полиноминально выраженное время с помощью недетерминированной машины (следующее состояние которой не всегда однозначно определяется предыдущими состояниями); решение таких задач можно проверить с помощью дополнительной информации полиномиальной длины за полиномиальное время; работу недетерминированной машины можно представить как разветвляющийся на каждой неоднозначности процесс и задача считается решенной, если хотя бы одна ветвь процесса пришла к ответу);
- экспоненциально сложные (задачи, которые могут быть решены за время, превышающее полиномиальное).

Класс *P* содержится в классе *NP*, и нельзя однозначно утверждать, что для *NP*-задачи не существует *P*-решения. Вопрос о возможной эквивалентности этих двух классов в настоящее время остается открытым. В этом контексте вводится понятие *NP*-полной задачи – задачи из класса *NP*, для которой может быть найдено *P*-решение.

Лекция 2.1

Структуры, объединения, перечисления в C/C++

Как уже говорилось, *структура* в C/C++ – это группа элементов различного типа. В программе структура определяется так:

```
struct <имя> {
<тип> <имя>;
…
<тип> <имя>;
};
```

Например,

```
struct adress {
        char city[50];
        char street[50];
        int number_house;
        int number_flat;
       };
```

Описание структуры определяет новый тип, имя которого можно использовать в дальнейшем наряду со стандартными типами данных, *например*:

```
struct Worker
{
string fio;
int code;
float salary;
```

```
};
Worker stuff[100], *ps;
```

В примере описана структура рабочий и далее двумя способами создаются массивы структур, в первом случае массив из 100 элементов, во втором случае указатель на начало динамического массива.

Инициализация элементов структуры может происходить явно в программе, вводиться с клавиатуры, считываться с файла (этот способ будет рассмотрен позже при изучении темы «файлы в C/C++»). Можно инициализировать сразу все поля структуры или некоторые из них. Пример инициализации в программе приведен ниже.

Пример:

```
Worker worker = {“Страусенко”, 215, 34001.55};
```

или

```
Worker worker;
cin>>worker.fio;
cin>>worker.code;
cin>>worker. salary;
```

Доступ к отдельным полям структуры выполняется с помощью операций выбора «.» (точка) при обращении к полю через имя структуры и «->» при обращении через указатель, *например*:

```
Worker worker, stuff[100], *ps;
int i, n;
cout<<”Enter n”<<endl;
cin>>n;
ps = new Worker[n];
…
worker.fio = “Страусенко”;
stuff[8].code = 215;
ps[5]->salary = 0.12;
…
```

Ввод и вывод информации при работе со структурами осуществляются поэлементно. *Например*:

```
#include <iostream>
#include <string>
using namespace std;

struct Students
{
   unsigned int num;
```

```
    string fio;
    unsigned short int kurs;
};
int main()
{
    Students stud[5];
    int i;
    for (i = 0; i < 5; i++)
  {
       cout << “enter information about the student “ << i << '\n';
       cout << “Please enter the number of the student: “ << '\n';
       cin >> stud[i].num;
       cout << “Please enter the name of the student: “ << '\n';
       cin >> stud[i].fio;
       cout << “Please enter the kurs of the student: “ << '\n';
       cin >> stud[i].kurs;
  }
    for (i = 0; i < 5; i++)
    {
      cout << stud[i].num <<”  “<< stud[i].fio<<”  “ << stud[i].kurs<<’\n’;
    }
}
```

В структурах есть возможность создавать *битовые поля*. Битовое поле – это поле, которому можно выделить память с точностью до бита. *Например*, следующая структура хранит в компактной форме дату и время дня с точностью до секунды.

```
struct TimeAndDate
{
unsigned hours :5; // часы от 0 до 24
unsigned mins :6; // минуты
unsigned secs :6; // секунды от 0 до 60
unsigned weekDay :3; // день недели
unsigned monthDay :5; // день месяца от 1 до 31
unsigned month :4; // месяц от 1 до 12
unsigned year :12 // год от 0 до 4096
};
```

Структуру можно передавать в функцию в качестве параметра и возвращать в качестве значения функции. Другие операции со структурами могут

быть определены пользователем. В качестве параметра функции использование структуры возможно таким образом:

```
…
void print(Worker worker)
{
  cout>>”Фамилия Имя Отчество”>>worker.fio>>endl;
  cout>>”Код”>>worker.code>>end;
  cout>>”Оклад”>>worker. salary >>endl;
}
…
void main()
{
  Worker w[10];
  int i;
…
   for (i = 0; i < 10; i++)
   {
      cout << “enter information about the workers “ << i <<’\n’;
      cout << “Please name: “ << ‘\n’;
      cin >> worker.fio;
      cout << “Please code: “ << ‘\n’;
      cin >> worker.code;
      cout << “Please salary: “ << ‘\n’;
      cin >> worker. salary;
   }
…
   for (i = 0; i < 10; i++)
       print(w[i]);
}
```

Ниже приведен пример ввода информации с помощью функции.

Пример:

```
…
Worker In_Data(int i)
{
   Worker w;
   cout << “Please name: “ << ‘\n’;
   cin >> w.fio;
   w.code=i;
   cout << “Please salary: “ << ‘n’;
```

```
   cin >> worker. salary;
   return w;
}
…
void main()
{
   Worker *pc;
   int i, n;
   cout<<"Enter n"<<endl;
   cin>>n;
   ps = new Worker[n];
  for (i=0; i<n; i++)
    ps[i]= In_Data (i);
  …
}
```

Объединением в C/C++ называется набор элементов одного типа или разных типов, каждый раз доступ возможен только к одному из элементов объединения. При объявлении объединения память выделяется в объеме одного из полей, которое занимает максимальное количество байтов, т. е. если в объединении присутствуют два поля типов int и double, памяти выделится как для переменной типа double.

Пример:

```
…
union size_of_sh
 {
  int rus;
  float USA;
 };

struct sh
 {
  …
  size_of_sh size;
  …
 };

sh sh1;
cout << "Enter rus size"<<endl;
cin >> sh1.size.rus;
```

```
cout << "Enter USA size"<<endl;
cin >> sh1.size.USA;
…
```

В приведенном примере, если мы обратимся к полю sh1.size.rus после последней строки, программа выдаст ошибку, так как из памяти она удалится при вводе данных в поле sh1.size.USA.

При написании программ часто возникает потребность определить несколько именованных констант, для которых требуется, чтобы все они имели различные значения. Для этого удобно воспользоваться перечисляемым типом данных:

enum [имя_типа] { список_констант };

Пример:

```
enum season {winter, spring, summer, autumn};
```

Причем, по умолчанию, к константам из перечисляемого типа можно обращаться как к целым числам, начиная с нуля.

Например:

```
…
enum season {winter, spring, summer, autumn};
strust birthday
{
 …
season seas;
} b1;
…
if (birthday.seas == 0) cout<< "Winter day of birth!"<<endl;
…
```

Если необходимо начать с другого начального значения, то его необходимо задать при объявлении перечисления вот так:

```
enum season {winter=1, spring, summer, autumn};
```

Динамические структуры данных

На практике часто возникают ситуации, когда в программе заранее не известно, сколько данных потребуется для работы. Ддинамические массивы не помогают, так как непонятно, сколько элементов будет использовано. В этом случае прибегают к динамическим структурам данных, которые отличаются от обычных структур наличием у их элементов дополнительных полей (минимум одного) – использую ссылки на другие элементы структуры. С помощью динамических структур можно организовать в программе различные списки, дере-

вья и др. Каждый элемент динамической структуры называют *узлом*. Узел состоит из двух блоков памяти: блок данных и блок ссылок.

Ссылки – это адреса других узлов этого же типа, с которыми данный элемент логически связан. В языке C/C++ для организации ссылок используются переменные-указатели. При добавлении нового узла в такую структуру выделяется новый блок памяти и (с помощью ссылок) устанавливаются связи этого элемента с уже существующими. Для обозначения конечного элемента в цепи используются нулевые ссылки (NULL).

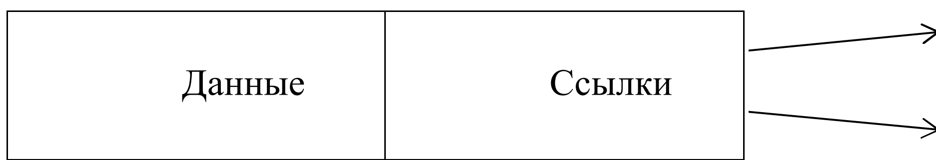

Рис. 12. Узел динамической структуры

Списки

В простейшем случае каждый узел содержит всего одну ссылку – ссылку на следующий за ним элемент. У последнего в списке элемента поле ссылки содержит NULL. *Списком* называется структура данных, каждый элемент которой с помощью указателя связывается со следующим элементом. В списке обычно выделяют две части – голову Head (первый элемент) и хвост Tail (список, состоящий из всех элементов, кроме первого). Ссылка на голову рассматривается как ссылка на список в целом.

В списке каждый элемент, как минимум, состоит из двух различных по назначению полей: содержательного поля (поле данных) и служебного поля (поля указателя), где хранится адрес следующего элемента списка. Поле указателя последнего элемента списка содержит нулевой указатель (NULL), свидетельствующий о конце списка.

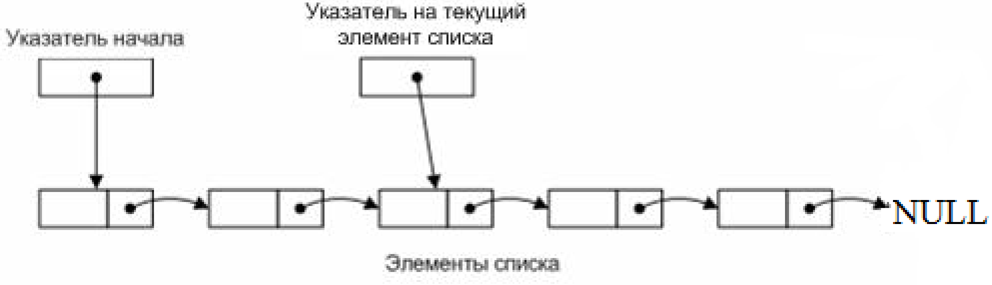

Рис. 13. Схематическое изображение списка

В программе C/C++ допустим следующий способ создания списка:

…

```
struct Node
{ int x;
  Node *next;
};
typedef Node *PNode;
PNode Head = NULL;
```

Если указатель ссылается только на следующее звено списка (как показано на рис.13 и в объявленной выше структуре), то такой список называют *однонаправленным* (односвязным), если на следующее и предыдущее звенья – *двунаправленным* (двусвязным) списком. Если указатель в последнем звене установлен не в NULL, а ссылается на заглавное звено списка, то такой список называется *кольцевым*. Кольцевыми могут быть и однонаправленные, и двунаправленные списки. Среди возможных списковых структур выделяют некоторые специальные списки – стек и очередь.

Стек – это линейный список, в котором все включения и исключения (и обычно всякий доступ) делаются в одном конце списка (в голове списка). Стеки используются в работе алгоритмов, имеющих рекурсивный характер. Конец стека называется вершиной стека. Принцип работы стека – «последний пришел – первый вышел». Внизу находится наименее доступный элемент. Часто говорят, что элемент опускается в стек.

Очередь – это линейный список, в один конец которого добавляются элементы, а с другого конца исключаются, элементы удаляются из начала списка, а добавляются в конце списка – как обыкновенная очередь в магазине. Принцип работы очереди: «первый пришел - первый вышел».

Пример:

Ввести с клавиатуры 10 чисел, записав их в стек. Вывести содержимое стека и очистить память.

```
#include <iostream>
using namespace std;
struct Node
{
	int x;
	Node *next;
};
typedef Node *PNode;

void Add(int x, PNode &Head)
{
	PNode MyNode;
```

```
        if (Head==NULL)
        {
                Head=new(Node);
                MyNode=Head;
                Head->next=NULL;
        }
        else
        {
                MyNode=new(Node);
              MyNode->next=Head;
               Head=MyNode;
        }
        MyNode->x=x;
}
void Show(PNode &Head)
{
        PNode MyNode;
        MyNode=Head;
        while (MyNode!=NULL)
        {
                cout<<MyNode->x<<" ";
                MyNode=MyNode->next;
        }
}
void ClearNode(PNode &Head)
{
        PNode MyNode;
        while (Head!=NULL)
        {
                MyNode=Head->next;
                delete Head;
                Head=MyNode;
        }
}
void main()
{
        PNode Head;
        Head=NULL;
        for (int i=0;i<10;i++)
```

```
        Add(i,Head);
    Show(Head);
    ClearNode(Head);
}
```

Пример:

Ввести с клавиатуры 10 чисел, записав их в очередь. Вывести содержимое очереди и очистить память. Для решения этой задачи достаточно в предыдущем примере изменить процедуру добавления элемента.

```
void Add(int x, PNode &Head, PNode &MyNode)
{
  PNode Temp;
    if (Head==NULL)
    {
        Head=new(Node);
        MyNode=Head;
        Head->next=NULL;
    }
    else
    {
        Temp=new(Node);
        MyNode->next=Temp;
        MyNode=Temp;
        MyNode->next=NULL;
    }
    cin>>MyNode->x;
}
…
void main()
{
    PNode Head, MyNode;
    Head=NULL;
   MyNode=NULL;
    for (int i=0;i<10;i++)
        Add(i,Head,MyNode);
    Show(Head);
    ClearNode(Head);
}
```

Многие проблемы при работе с односвязным списком вызваны тем, что в них невозможно перейти к предыдущему элементу. Возникает естественная

идея – хранить в памяти ссылку не только на следующий, но и на предыдущий элемент списка. Двусвязный список может быть организован следующим образом:

```
struct Node
 {
   int x;
   Node *next, *prev;
  };
typedef Node *PNode;
```

Понятно, что при добавлении узлов в двусвязный список необходимо создавать ссылки на предыдущий элемент и Head->prev = NULL.

Для того чтобы пройти весь список и сделать какие-либо действия с его элементами, надо начать с головы и, используя указатель next, продвигаться к следующему узлу:

```
PNode p = Head;
while ( p != NULL )
{
  …
  p = p->next;
}
```

Аналогично можно пройти весь список с конца, двигаясь по указателям prev.

Алгоритмы работы со списками

К основным алгоритмам работы со списками относятся поиск узла по какому-либо критерию, добавление и удаление узла, сортировка. Рассмотрим алгоритмы и примеры их реализации для линейных односвязных и двусвязных списков.

Поиск узла. В задачах часто требуется найти в списке нужный элемент по какому-либо критерию (его адрес или данные). Необходимо предусмотреть, что требуемого элемента может и не быть, тогда просмотр заканчивается при достижении конца списка. Алгоритм для односвязного списка:

1) начать с головы списка;
2) пока текущий элемент существует (указатель – не NULL), проверить нужное условие и перейти к следующему элементу;
3) закончить, когда найден требуемый элемент или все элементы списка просмотрены.

Например, поиск поля со значением x:

```
PNode Find (PNode Head,int x)
{
```

```
  PNode q = Head;
  while (q!=NULL  &&  q->data !=x)
    q = q->next;
  return q;
}
```

Функция вернет либо указатель на NULL (если элемент не найден), либо указатель на найденный элемент. В двусвязном списке можно двигаться в обратном направлении.

Добавление узла после заданного. Для односвязного списка. Предположим, что мы нашли определенный элемент списка с адресом p.. Требуется вставить в список новый узел после узла с адресом p. Алгоритм:

1) установить ссылку нового узла на узел, следующий за данным;
2) установить ссылку данного узла p на NewNode.

Последовательность операций менять нельзя, потому что если сначала поменять ссылку у узла p, будет потерян адрес следующего узла:

```
void AddAfter (PNode p, PNode NewNode)
{
  NewNode->next = p->next;
  p->next = NewNode;
}
```

Алгоритм для двусвязного списка. Если узел p является последним, то операция сводится к добавлению в конец списка (рассмотрен ниже). Если узел p – не последний, то операция вставки выполняется в два этапа:

1) установить ссылки нового узла на следующий за данным (next) и предшествующий ему (prev);
2) установить ссылки соседних узлов так, чтобы включить NewNode в список.

```
 void AddAfter (PNode &Head, PNode &Tail, PNode p, PNode NewNode)
{
  if ( p->next==NULL )
    AddLast (Head, Tail, NewNode);
  else
  {
    NewNode->next = p->next;
    NewNode->prev = p;
    p->next->prev = NewNode;
    p->next = NewNode;
  }
}
```

Добавление узла перед заданным. Для односвязного списка сложность состоит в том, что нет ссылки на предыдущий элемент. Таким образом, необходимо в списке найти элемент, за которым будет следовать узел p и воспользоваться алгоритмом вставки после заданного, а также предусмотреть случай, если заданный узел является головой списка.

```
void AddBefore(PNode &Head, PNode p, PNode NewNode)
{
  PNode q = Head;
  if (Head == p)
  {
         NewNode->Next=Head;
             Head=NewNode;
    return;
  }
  while (q!=NULL  && q->next!=p)
    q = q->next;
  if ( q!=NULL  )
    AddAfter(q, NewNode);
}
```

Для двусвязного списка все проще, алгоритм аналогичен вставке после заданного.

Добавление узла в начало списка. Для односвязного списка добавление узла в начало списка происходит по алгоритму, приведенному выше при написании алгоритма добавления перед заданным. Для двусвязного списка алгоритм добавления узла NewNode в начало списка выглядит так:

1) установить ссылку next узла NewNode на голову существующего списка и его ссылку prev в NULL;
2) установить ссылку prev бывшего первого узла (если он существовал) на NewNode;
3) установить голову списка на новый узел;
4) если в списке не было ни одного элемента, хвост списка также устанавливается на новый узел.

```
void AddFirst(PNode &Head, PNode &Tail, PNode NewNode)
{
  NewNode->next = Head;
  NewNode->prev = NULL;
  if ( Head!=NULL  ) Head->prev = NewNode;
  Head = NewNode;
```

```
  if ( Tail==NULL ) Tail = Head;
}
```

Добавление узла в конец списка. Для односвязного списка надо сначала найти последний узел, у которого ссылка равна NULL, а затем воспользоваться процедурой вставки после заданного узла. Необходимо предусмотреть случай, когда список пуст.

```
void AddLast(PNode &Head, PNode NewNode)
{
  PNode q = Head;
  if (Head == NULL)
        {
          Head=new(Node);
               NewNode=Head;
               Head->Next=NULL;
          return;
    }
  while (q->next!=NULL  ) q = q->next;
  AddAfter(q, NewNode);
}
```

Для двусвязного списка аналогично:

```
void AddLast(PNode &Head, PNode &Tail, PNode NewNode)
{
  NewNode->next = NULL;
  NewNode->prev = Tail;
  if ( Tail !=NULL  ) Tail->next = NewNode;
  Tail= NewNode;
  if ( Head==NULL) Head = Tail
}
```

Удаление узла. Эта процедура также связана с поиском заданного узла по всему списку, так как нам надо поменять ссылку у предыдущего узла, а перейти к нему непосредственно невозможно. Если мы нашли узел, за которым идет удаляемый узел, надо просто переставить ссылку. При удалении узла освобождается память, которую он занимал. Отдельно рассматриваем случай, когда удаляется первый или последний (для двусвязного) элемент списка. Для односвязного списка:

```
void DeleteNode(PNode &Head, PNode OldNode)
{
  PNode q = Head;
  if (Head == OldNode)
```

```
  Head = OldNode->next;
 else
  {
   while (q !=NULL  && q->next != OldNode)
     q = q->next;
   if ( q == NULL ) return;
   q->next = OldNode->next;
  }
 delete OldNode;
}
```

Для двусвязного списка:

```
void Delete(PNode &Head, PNode &Tail, PNode OldNode)
{
 if (Head == OldNode)
 {
  Head = OldNode->next;
  if ( Head ) Head->prev = NULL;
  else Tail = NULL;
 }
 else
 {
  OldNode->prev->next = OldNode->next;
  if ( OldNode->next )
   OldNode->next->prev = OldNode->prev;
  else
   {
    Tail = OldNode->prev;
    Tail->next=NULL;
   }
 }
 delete OldNode;
}
```

Лекция 2.2

Работа с файлами в C/C++

Файлом называют способ хранения информации на физическом устройстве. В языке C/C++ все необходимые действия по работе с файлами выполняются с помощью функций, включенных в стандартную библиотеку stdio.h (C) и

библиотеки fstream, ifstream, ofstream (C++). Файл представляется в виде потока. В С/С++ существуют два типа потоков: текстовые (text) и двоичные (binary).

Текстовый поток – это последовательность символов. При передаче символов из потока на экран часть из них не выводится (например, символ возврата каретки, перевода строки).

Двоичный поток – это последовательность байтов, которые однозначно соответствуют тому, что находится на внешнем устройстве.

Прежде чем читать или записывать информацию в файл, он должен быть открыт и тем самым связан с потоком. После использования файла его необходимо закрыть.

Логическое имя – это указатель на требуемый файл, который необходимо определить.

Рассмотрим работу с файлами с использованием стандартной библиотеки stdio.h. Определение логического имени потока выглядит следующим образом:

FILE *f;

Здесь FILE – имя типа, описанное в стандартном заголовочном файле stdio.h, f – указатель на файл. Для открытия файла необходимо использовать функцию fopen( )

f = fopen(<спецификация файла>, «<способ использования файла>«);

Спецификация файла – имя файла и путь к нему. *Например*, «c:\\my_prog.txt».

Способ использования файла задается символами. Укажем некоторые из них:

r – открыть существующий файл для чтения;

w – создать новый файл для записи (если файл с указанным именем существует, то он будет переписан);

a – дополнить файл (открыть существующий файл для записи информации начиная с конца файла или создать файл, если он не существует);

r+ – открыть существующий файл для чтения и записи;

w+ – создать новый файл для чтения и записи;

a+ – дополнить или создать файл с возможностью чтения и записи;

rb – открыть двоичный файл для чтения;

wb – создать двоичный файл для записи;

ab – дополнить двоичный файл;

Если в результате обращения к функции fopen( ) возникает ошибка, то она возвращает указатель на константу NULL.

Рекомендуется использовать следующий способ открытия файла:

FILE *fp;

if (fp=fopen("1.txt","r")==NULL

```
{ puts("Не могу открыть файл\n");
 return; }
```

Закрытие файла производится с помощью библиотечной функции fclose().

При успешном завершении операции функция fclose( ) возвращает значение нуль. Любое другое значение свидетельствует об ошибке. Другие библиотечные функции, используемые для работы с файлами из библиотеки stdio.h, приведены в табл. 15.

Таблица 15. Функции работы с файлами (stdio.h)

Функция	**Описание**
int putc(int c, FILE *f);	Запись символа в файл. Здесь f – указатель на файл, возвращенный функцией fopen(), c – символ для записи (переменная c имеет тип int, но используется только младший байт). При успешном завершении putc() возвращает записанный символ, в противном случае возвращается константа EOF. Она определена в файле stdio.h и имеет значение -1
int getc(FILE *f);	Чтение символа из файла. Здесь f – указатель на файл, возвращенный функцией fopen(). Эта функция возвращает прочитанный символ. Соответствующее значение имеет тип int, но старший байт равен нулю. Если достигнут конец файла, то getc() возвращает значение EOF
int fputc(char *, FILE *f);	Запись строки в файл. При возникновении ошибки возвращается значение EOF
int fgetc(char *, int n, FILE *f);	Чтение строки из файла. В качестве второго параметра должно быть указано максимальное число вводимых символов плюс единица, а в качестве третьего – указатель на переменную файлового типа. Строка считывается целиком, если ее длина не превышает указанного числа символов, в противном случае функция возвращает только заданное число символов. Функция возвращает указатель на строку при успешном завершении и константу NULL в случае ошибки либо достижения конца файла. При возникновении ошибки возвращается значение EOF

Функция	Описание
int fprintf(FIE *f, ‘тескт, спецификаторы', переменные);	Выполняет те же действия, что и функция printf( ), но работает с файлом
int fscanf(FIE *f, “спецификаторы”, &переменные);	Выполняет те же действия, что и функция scanf(), но работает с файлом. При достижении конца файла возвращается значение EOF
int fseek(FILE *f, long count, int access);	Чтение и запись с произвольным доступом. Здесь f – указатель на файл, возвращенный функцией fopen( ), count - номер байта относительно заданной начальной позиции, начиная с которого будет выполняться операция, access – способ задания начальной позиции.Переменная access может принимать следующие значения: SEEK_SET (имеет значение 0) – начало файла; SEEK_CUR (имеет значение 1) – начальная позиция считается текущей; SEEK_END (имеет значение 2) – конец файла. При успешном завершении возвращается нуль, при ошибке - ненулевое значение
unsigned fread(void *ptr, unsigned size, unsigned n, FILE *f);	Чтение блоков данных из потока. Читает n элементов данных, длиной size байт каждый, из заданного входного потока f в блок, на который указывает указатель ptr. Общее число прочитанных байтов равно произведению n*size. При успешном завершении функция fread( ) возвращает число прочитанных элементов данных, при ошибке – 0
unsigned fwrite(void *ptr, unsigned size, unsigned n, FILE *f);	Запись блоков данных в поток. Добавляет n элементов данных, длиной size байт каждый, в заданный выходной файл f. Данные записываются с позиции, на которую указывает указатель ptr. При успешном завершении операции функция возвращает число записанных элементов данных, при ошибке – неверное число элементов данных

В языке С имеются пять стандартных файлов со следующими логическими именами:
stdin – для ввода данных из стандартного входного потока (по умолчанию - с клавиатуры);
stdout – для вывода данных в стандартный выходной поток (по умолчанию - на экран дисплея);
stderr – файл для вывода сообщений об ошибках (всегда связан с экраном дисплея);
stdprn – для вывода данных на принтер;
stdaus – для ввода и вывода данных в коммуникационный канал.

Пример:

Программа, по которой в файл запишется последовательность целых чисел, вводимых с клавиатуры, а затем эта последовательность будет прочитана и выведена на экран. Признаком конца ввода пусть будет число 0.

```
#include<stdio.h>
#include<conio.h>
void main()
{
   FILE *f;
   int x;
   if((f=fopen(“1.txt”,”w”))==NULL)
   {
       puts(“Ошибка открытия файла для записи!\n”);
       return ;
   }
 puts(“Введите числа. Признак конца – 0”);
 scanf(“%d”,&x);
 while(x!=0)
 {
    fprintf(f,”%d”,x);
    scanf(“%d”,&x);
 }
 fclose(f);
 if((f=fopen(“1.txt”,”r”))==NULL)
 {
    puts(“Ошибка открытия файла для чтения!\n”);
    return ;
  }
```

```
else
{
  while((fscanf(f,"%d",&x)))!=EOF)
     printf("\n%d",x);
  fclose(f);
}
}
```

В рассмотренном примере условие while((fscanf(f,”%d”,&x)))!=EOF) проверяет на считываемость числа с файла, т. е. пока считывается, то можно выполнять какие-либо действия. Так необходимо делать, когда неизвестно, сколько элементов в файле и сколько придется считывать данных. Кроме такого способа можно еще воспользоваться функцией проверки конца файла:

```
int feof(FILE *f);
```

Функция feof() возвращает не 0, если достигнут конец файла, в противном случае она возвращает 0.

Например, пока не конец файла, считывать целые числа:

```
…
while(!feof(f))
{
   fscanf(f,"%d",&x);
}
…
```

Теперь рассмотрим работу с файлами с помощью библиотек fstream (файлы для чтения и записи), ifstream (файлы для чтения), ofstream (файлы для записи). Для работы с файлом объявляется поток соответствующего типа, который называется так же, как и подключаемая библиотека. Следовательно, если необходимо создать файл для записи данных, то в программе пишут:

```
#include <ofstream>
…
ofstream f;
…
```

Перед тем, как работать с файлом, его необходимо открыть

```
f.open(<спецификация файла, <способ использования файла>);
```

Спецификация файла – имя файла и путь к нему. *Например*, “1.txt”.

Способ использования файла задается следующими константами (приведены основные):

ios_base::in	открыть файл для чтения
ios_base::out	открыть файл для записи
ios_base::ate	при открытии переместить указатель в конец файла

ios_base::app открыть файл для записи в конец файла
ios_base::trunc удалить содержимое файла, если он существует
ios_base::binary открытие файла в двоичном режиме

Т. е., чтобы открыть файл для записи, необходимо написать в программе:

```
f.open("1.txt", ios_base::out);
```

Режимы открытия файлов можно комбинировать с помощью поразрядной логической операции или |. *Например*, ios_base::out | ios_base::trunc – открытие файла для записи, предварительно очистив его.

Чтобы проверить правильность открытия файла, нужно использовать функции is_open или fail. *Например*:

```
…
if (!f.is_open())
      cout << "Файл не может быть открыт!\n";
else
   {
      …
   }
…
```

или

```
…
if (f.fail())
      cout<<"\n Ошибка открытия файла";
else
   {
      …
   }
…
```

Файловый ввод/вывод аналогичен стандартному вводу/выводу, единственное отличие – это то, что ввод/вывод выполняется не на экран, а в файл. Если ввод/вывод на стандартные устройства выполняется с помощью объектов cin и cout, то для организации файлового ввода/вывода достаточно создать собственные объекты, которые можно использовать аналогично операторам cin и cout. *Например*, чтобы считать целое число из файла и присвоить его значение переменной *a*, необходимо написать

```
…
f>>a;
…
```

Аналогично можно записать значение переменной в файл:

…

```
f<<a;
…
```
Если нужно считать из файла символьную строку, то необходимо воспользоваться функцией getline:

```
…
f.getline(str, 50);
…
```
После окончания работы с файлом его необходимо закрыть с помощью функции close:

```
…
f.close();
…
```
Пример:

Программа, по которой в файл запишется последовательность целых чисел, вводимых с клавиатуры, а затем эта последовательность будет прочитана и выведена на экран. Признаком конца ввода пусть будет число 0.

```
#include<iostream.h>
#include<fstream>
using namespace std;
void main()
{
    ofstream *f;
    int x;
    f.open("1.txt", ios_base::out);
    if((f.fail()
    {
        cout<<"Ошибка открытия файла для записи!\n";
     }
else
{
  cout<<"Введите числа. Признак конца – 0";
  cin>>x;
  while(x!=0)
  {
     f<<x;
     cin>>x;
  }
  f.close();
  f.open("1.txt", ios_base::in);
```

```
if((f.fail()
  {
      cout<<"Ошибка открытия файла для записи!\n";
   }
else
{
while((!f.eof())
  {
     f>>x;
     cout<<x;
  }
f.close();
}
```

В рассмотренном примере присутствует функция проверки конца файла f.eof(), которую можно использовать при необходимости.

Лекция 2.3

Объектно-ориентированное программирование

Объектно-ориентированное программирование – это парадигма программирования (метод, прием), при использовании которого главными элементами программ являются *объекты.* Изучение ООП целесообразно начать на примере объектно-ориентированного языка программирования С++, как наиболее теоретически выдержанного в этой части.

Объект – это совокупность свойств, методов их обработки и событий, на которые данный объект может реагировать и которые приводят, как правило, к изменению свойств объекта.

Большинство концепций объектно-ориентированного программирования были развиты Аланом Кэйем и Дэном Ингаллсом в языке Smalltalk. Именно он стал первым широко распространённым объектно-ориентированным языком программирования. В настоящее время количество прикладных языков программирования, реализующих объектно-ориентированную парадигму, является наибольшим по отношению к другим парадигмам.

Основные принципы, заложенные в ООП, по книге Т. Бадда:

1. Всё является объектом.
2. Вычисления осуществляются путём взаимодействия (обмена данными) между объектами, при котором один объект требует, чтобы другой объект выполнил некоторое действие. Объекты взаимодействуют, посылая и

получая сообщения. Сообщение – это запрос на выполнение действия, дополненный набором аргументов, которые могут понадобиться при выполнении действия.

3. Каждый объект имеет независимую память, которая состоит из других объектов.
4. Каждый объект является представителем класса, который выражает общие свойства объектов (таких, как целые числа или списки).
5. В классе задаётся поведение (функциональность) объекта. Тем самым все объекты, которые являются экземплярами одного класса, могут выполнять одни и те же действия.
6. Классы организованы в единую древовидную структуру с общим корнем, называемую иерархией наследования. Память и поведение, связанное с экземплярами определённого класса, автоматически доступны любому классу, расположенному ниже в иерархическом дереве.

В ООП основной идеей является *абстракция* – это придание объекту характеристик, которые отличают его от всех других объектов, четко определяя его концептуальные границы. Основная идея – отделить способ использования составных объектов данных от деталей их реализации в виде более простых объектов. Такой подход является основой объектно-ориентированного программирования. Это позволяет работать с объектами, не вдаваясь в особенности их реализации.

Объктно-ориентированный подход основан на представлении любого объекта в виде набора взаимодействующих частей, которые в свою очередь также являются объектами. Такая модель представления объекта и взаимодействия с другими объектами называется *объектной моделью*. Каждый объект характеризуется набором свойств, состояний и поведений. Объекты с одинаковыми наборами свойств и поведений объединяются в классы. Следует отметить, что существуют отношения между объектами и классами, такие как ассоциации, обобщения и отношение зависимости. *Ассоциацией* называется отношение, показывающее, что объекты одного типа (класса) связаны с объектами другого типа. *Обобщением* называется отношения общего класса и его вариаций. *Зависимость* заключается в том, что изменение одного типа объектов ведет к изменениям другого типа.

Основные принципы ООП

Основными принципами ООП являются *инкапсуляция*, *наследование* и *полиморфизм*.

Инкапсуляция позволяет скрывать внутреннюю реализацию. В классе могут быть реализованы внутренние вспомогательные методы, поля, к которым

доступ для пользователя необходимо запретить, тогда и используется инкапсуляция.

Наследование позволяет создавать новый класс на базе другого класса. В данном случае возникают такие понятия, как базовый и производный классы. *Производный класс* – класс-потомок. *Базовый класс* – класс-родитель. Производный класс может быть родителем для других производных классов. Таким образом, организована иерархия классов.

Полиморфизм – это способность объектов с одним интерфейсом иметь различную реализацию.

Подробнее эти приницпы будет изучены в лекциях 2.4 и 2.5.

Основные понятия ООП

Базовыми понятиями ООП являются *класс* и *объект*.

Класс – это абстрактный тип данных (шаблон). С помощью класса описывается некоторая сущность (ее характеристики и возможные действия). *Например*, класс может описывать студента, автомобиль и т.д.

Описав класс, мы можем создать его экземпляр – *объект*. *Объект* – это уже конкретный представитель класса.

Пример:

```
class student
{
  public:
  string name;
  string special;
  int kurs;
  show() {…}
};
student st;
```

Ключевое слово public означает открытый доступ ко всем элементам класса. Все виды доступа к элементам класса будут рассмотрены в конце данной лекции, а пока будем пользоваться только открытым доступом к данным и методам класса. Члены класса можно разделить на информационные члены (данные) и функции-члены (методы) класса. Информационные члены описывают внутреннюю структуру информации, хранящейся в объекте, который создается на основе класса. Методы класса описывают алгоритмы обработки этой информации. Данные, хранящиеся в информационных членах, описывают состояние объекта, созданного на основе класса. Состояние объекта изменяется на основе изменения хранящихся данных с помощью методов класса. Алгорит-

мы, заложенные в реализации методов класса, определяют поведение объекта, т. е. реагирование объекта на поступающие внешние воздействия в виде входных данных. Среди методов класса существуют основные – *конструктор* и *деструктор*, которые будут рассмотрены подробно в лекции 2.4. Конструктор предназначен для инициализации элементов класса некоторыми начальными значениями. Деструктор служит для уничтожения элементов класса.

Каждый метод класса должен быть определен в программе. Определить метод класса можно либо непосредственно в классе (если тело метода не слишком сложно и громоздко), либо вынести определение вне класса, а в классе только объявить соответствующий метод, указав его прототип. При определении метода класса вне класса для указания области видимости соответствующего имени метода используется операция «::».

Пример:

```
class x
{
  public:
  int ia1;
  …
  int func1();
 };
int x::func1()
{
  …
  return ia1;
 }
```

Это позволяет повысить наглядность текста, особенно в случае значительного объема кода в методах. При определении метода вне класса с использованием операции «::» прототипы объявления и определения функции должны совпадать.

Как уже говорилось, экземплярами класса являются объекты. В программе можно создать один и несколько объектов класса, в том числе массив объектов. По сути, объекты – это те же структуры, рассмотренные нами ранее, с разницей в том, что по умолчанию в структуре доступ ко всем данным и методам открытый, а в классе – закрытый. Работа с объектами схожа на работу со структурами. Рассмотрим ранее приведенный пример, реализовавший структуру студент, на применение объектов.

Пример:

```
#include <iostream>
#include <string>
```

```
using namespace std;

class Students
{
   public:
   unsigned int num;
   string fio;
   unsigned short int kurs;
   void show();
};

void Students::show()
{
   cout << num <<”  “<< fio<<”  “ <<kurs<<’\n’;
}

int main()
{
   Students stud[5];
   int i;
   for (i = 0; i < 5; i++)
   {
      cout << “enter information about the student “ << i << ‘\n’;
      cout << “Please enter the number of the student: “ << ‘\n’;
      cin >> stud[i].num;
      cout << “Please enter the name of the student: “ << ‘\n’;
      cin >> stud[i].fio;
      cout << “Please enter the kurs of the student: “ << ‘\n’;
      cin >> stud[i].kurs;
   }
   for (i = 0; i < 5; ++i)
      stud[i].show();
   }
```

Объекты также могут быть полями других классов. Например, если создать класс группа, то одним из полей можно включить массив объектов класса Students.

Пример:

```
…
class group
```

```
{
  public:
  Students st[30];
  string special;
};
…
```

Уровни доступа к членам класса

По уровню доступа все члены (данные и методы) класса делятся:
на открытые (public),
закрытые (private),
защищённые (protected).

Перед объявлением членов внутри класса ставятся соответствующие ключевые слова. Как уже говорилось ранее, если такое слово не поставлено, то по умолчанию член объявлен с уровнем private. Члены, объявленные как private, доступны только внутри класса. Члены, объявленные как protected, доступны внутри класса и внутри всех его потомков. Члены, объявленные как public, доступны как внутри, так вне класса (в том числе в потомках). Подробнее с различными уровнями доступа к элементам класса познакомимся при изучении инкапсуляции, наследования и полиморфизма.

Лекция 2.4

Конструкторы и деструкторы

Как уже говорилось выше, конструкторы и деструкторы являются специальными методами класса. Конструкторы вызываются при создании объектов класса для отведения памяти под них и инициализации. Деструкторы вызываются при уничтожении объектов для освобождения отведенной для них памяти. В большинстве случаев конструкторы и деструкторы вызываются автоматически (неявно) соответственно при описании объекта (в момент отведения памяти под него) и при уничтожении объекта. Конструктор (как и деструктор) может вызываться и явно, например, при создании объекта в динамической области памяти с помощью операции new.

Так как конструкторы и деструкторы неявно входят в интерфейс объекта, их следует располагать в открытой области класса. Конструкторы и деструкторы могут располагаться и в закрытой области для блокирования возможности неявного создания объекта. Но в этом случае явное создание объекта возможно только при использовании статических методов, являющихся частью класса, а не конкретного объекта. Статические методы описываются далее.

Отличия *конструктора* от обычной функции:

1) имя конструктора совпадает с именем класса;
2) при описании конструктора не указывается тип возвращаемого значения.

В описании конструктора тип возвращаемого значения не указывается не потому, что возвращаемого значения нет. Оно как раз есть. Ведь результатом работы конструктора в соответствии с его названием является созданный объект того типа, который описывается данным классом. Страуструп отмечал, что конструктор – это то, что область памяти превращает в объект.

Конструкторы можно классифицировать разными способами:

1) по наличию параметров:
   - без параметров,
   - с параметрами;
2) по количеству и типу параметров:
   - конструктор умолчания,
   - конструктор преобразования,
   - конструктор копирования,
   - конструктор с двумя и более параметрами.

Набор и типы параметров зависят от того, на основе каких данных создается объект. В классе может быть несколько конструкторов. В соответствии с правилами языка С++ все они имеют одно имя, совпадающее с именем класса, что является одним из проявлений статического полиморфизма. Компилятор выбирает тот конструктор, который в зависимости от ситуации, в которой происходит создание объекта, удовлетворяет ей по количеству и типам параметров. Естественным ограничением является то, что в классе не может быть двух конструкторов с одинаковым набором параметров.

Конструктор умолчания, если он явно не прописан, вызывается неявно и просто выделяет память под объект. Конструктор преобразования и конструктор с параметрами создаются для инициализации объекта. Конструктор копирования вызывается, когда объект является параметром или возвращаемым значением какой-либо функции, а также для явного создания копий объекта.

Деструкторы применяются для корректного уничтожения объектов. Часто процесс уничтожения объектов включает в себя действия по освобождению выделенной для них по операциям new памяти. Имя деструктора: ~имя_класса(). У деструкторов нет параметров и возвращаемого значения. В отличие от конструкторов деструктор в классе может быть только один.

Пример. Описание класса с несколькими конструкторами и создание объектов.

```
class box
{
  int len, wid, hei;
```

```
public:
box(int l, int w, int h)
{
  len = l;
  wid = w;
  hei = h;
}
box(int s)
{
  len = s;
  wid = s;
  hei = s;
}
box()
{
 len = 2;
 wid =1;
 hei = 1;
}
int volume()
{
  return len*wid*hei;
}
~box()
{
}
};
box cube(1), b1(1,2,3), b2;
```

В рассмотренном примере при создании объекта cube вызовется конструктор от одного параметра и будет создан короб со всеми сторонами, равными единице; при создании объекта b1 вызовется конструктор от трех параметров и будет создан короб со сторонами 1, 2 и 3 соответственно; при создании объекта b2 вызовется конструктор по умолчанию и будет создан короб со сторонами 2, 1 и 1.

Автоматически могут генерироваться только конструкторы умолчания, конструкторы копирования и деструкторы. Если в классе явно не описано ни одного конструктора, то автоматически генерируется конструктор умолчания с пустым телом. Если в классе явно описан хотя бы один конструктор, например, конструктор копирования, то конструктор умолчания не будет автоматически

генерироваться, даже если он необходим в соответствии с постановкой задачи. В случае отсутствия в классе явно описанного конструктора копирования он всегда генерируется автоматически и обеспечивает поверхностное копирование. Если в классе не описан деструктор, то всегда автоматически генерируется деструктор, который не производит никаких действий. Таким образом, даже если в классе не описаны конструкторы и деструктор, они все равно неявно в нем присутствуют.

При создании объекта конструкторы вызываются в следующем порядке:

1. Конструкторы базовых классов, если класс для создаваемого объекта является наследником других классов, в порядке их появления в описании класса. Если в списке инициализации описываемого класса присутствует вызов конструктора преобразования (или конструктора с двумя и более параметрами) базового класса, то вызывается конструктор преобразования (или конструктор с двумя и более параметрами), иначе вызывается конструктор умолчания базового класса.
2. Конструкторы умолчания всех вложенных информационных членов, которые не перечислены в списке инициализации, и конструкторы преобразования, копирования и конструкторы с двумя и более параметрами всех вложенных информационных членов, которые перечислены в списке инициализации. Все перечисленные в данном пункте конструкторы (умолчания, преобразования, копирования, с двумя и более параметрами) вызываются в порядке описания соответствующих информационных членов в классе.
3. Собственный конструктор.

Такая последовательность вызова конструкторов логически обосновывается тем, что в момент выполнения собственного конструктора все информационные поля должны быть уже проинициализированы.

Деструкторы вызываются в обратном порядке:

1. Собственный деструктор. В момент начала его работы поля класса еще не очищены, и их значения могут быть использованы в теле деструктора.
2. Деструкторы вложенных объектов в порядке, обратном порядку их описания.
3. Деструкторы базовых классов в обратном порядке их задания.

Инкапсуляция

Как уже говорилось в предыдущей лекции, инкапсуляция позволяет скрывать внутреннюю реализацию. В классе могут быть реализованы внутренние вспомогательные методы, поля, к которым доступ для пользователя необ-

ходимо запретить. Принцип инкапсуляции обеспечивается вводом в класс областей доступа:

- private (закрытый, доступный только собственным методам);
- public (открытый, доступный любым функциям);
- protected (защищенный, доступный только собственным методам и методам производных классов).

Члены класса, находящиеся в закрытой области (private), недоступны для использования со стороны внешнего кода. Напротив, члены класса, находящиеся в открытой секции (public), доступны для использования со стороны внешнего кода. При описании класса каждый член класса помещается в одну из перечисленных выше областей доступа следующим образом:

```
class Имя_класса {
private:
определение_закрытых_членов_класса
public:
определение_открытых_членов_класса
protected:
определение_защищенных_членов_класса
…
};
```

Порядок следования областей доступа и их количество в классе произвольны. Служебное слово, определяющее первую область доступа, может отсутствовать. По умолчанию эта область считается private. В закрытую (private) область обычно помещаются информационные члены, а в открытую (public) область – методы класса, реализующие интерфейс объектов класса с внешней средой. Если какой-либо метод имеет вспомогательное значение для других методов класса, являясь подпрограммой для них, то его также следует поместить в закрытую область. Это обеспечивает логическую целостность информации.

Пример. Класс – сфера, для которой данные – радиус – скрыт, а метод вычисления объема сферы открыт.

```
class sphere
{
 private:
 float r;
 public:
 float volume()
 {    return 4*3.14*pow(r,3)/3;   }
};
```

Лекция 2.5
Наследование

Наследование позволяет создавать производный класс на базе уже существующего. *Производный класс* – класс-потомок. *Базовый класс* – класс-родитель. Производный класс может быть родителем для других производных классов. Таким образом, организована *иерархия классов*. Производные классы являются более мощными по отношению к базовым классам, так как, включая поля и методы базового класса (кроме конструктора, деструктора и компонентной функции-оператора (=)), они обладают еще и своими компонентами. Доступ к полям и функциям базового класса при наследовании ограничивается с помощью специальных описателей, определяющих вид наследования:

class <Имя пр_класса >:<Вид наследования><Имя баз_класса>
{<Тело класса>};

где вид наследования определяется ключевыми словами: private, protected, public. Если вид наследования явно не указан, то по умолчанию принимается private. Видимость полей и функций базового класса из производного определяемая видом наследования и типом доступа полей и функций, представлена в табл. 16.

Таблица 16. Видимость компонентов базового класса в производном классе

Вид наследования	**Объявление компонентов в базовом классе**	**Видимость компонентов в производном классе**
private	private	не доступны
	protected	private
	public	private
protected	private	не доступны
	protected	protected
	public	protected
public	private	не доступны
	protected	protected
	public	public

Пример. Базовый класс А и наследный класс В. Вид наследования public. Так как в базовом классе методы с public-доступом, в производном классе можно свободно ими пользоваться. Прямого доступа к закрытому полю х производного класса нет, только через открытые методы.

class A

```
{
  int x;
  public:
  void set_a(int ax)
   {
     x=ax;
   }
  print_a()
   {
      cout<<x;
   }
};
class B:public A
{
  int y;
  public:
  void set_b(int ax,int bx)
  {
     set_a(ax); // доступ к скрытому полю x
     b=bx;
  }
  print_b()
 {
    print_a(); // доступ к скрытому полю x
    cout<<y;
  }
};
```

Если базовый класс содержит хотя бы один конструктор и деструктор, то производный класс должен включать собственные конструктор и деструктор. При создании объектов производного класса предусмотрен автоматический вызов конструктора базового класса для инициализации его полей. По умолчанию осуществляется вызов конструктора базового класса без параметров (если такого конструктора нет, то компилятор выдает сообщение об ошибке error C2512). Чтобы передать конструктору производного аргументы для инициализации полей базового класса, следует:

- добавить соответствующие параметры к собственным параметрам конструктора производного класса;
- вызвать конструктор базового класса в списке инициализации конструктора производного класса, передав ему соответствующие аргументы.

Пример:

```
class A
{
  int x;
  public:
  A(int ax):x(ax) {} // конструктор базового класса
};
class B:public A
{
  int y;
  public:
  B(int ax,int ay):A(ax),y(ay){} // конструктор производного класса
};
```

Язык С++ позволяет осуществлять наследование не только от одного, но и одновременно от нескольких классов. Такое наследование получило название *множественного наследования*. Описание производного класса при множественном наследовании выглядит следующим образом:

class <Имя производного класса>:
<Вид наследования><Имя базового класса 1>,
<Вид наследования><Имя базового класса 2>,
...
<Вид наследования><Имя базового класса n> {...};

Вид наследования, как и в случае простого наследования, определяет режим доступа к компонентам соответствующего базового класса. Если конструкторы базовых классов не имеют аргументов, то производный класс может не иметь конструктора. При наличии у конструкторов базового класса аргументов производный класс обязан иметь конструктор со списком инициализации следующего вида:

<Имя конструктора производного класса>(<Список аргументов>):
<Имя конструктора базового класса 1>(<Список аргументов 1>),
......
<Имя конструктора базового класса n>(<Список аргументов n>)
{<Тело конструктора производного класса>}

Пример:

```
class first
{
  public: int num_f;
  first(int va):num_f(va)
   {
```

```
      cout<<"Конструктор 1\n";
    }
};
class second
{
  public: char c_s; int num_s;
  second(int vn):num_s(vn)
    {
      cout<<"Конструктор 2\n";
    }
};
class third:private first, public second
{
   public:
   third(int nm,char vc,int nfx):first(nfx), second(nm)
     {
       cout<<"Конструктор third\n";
       c_s=vc;
     }
};
```

В рассмотренном примере создан третий класс на базе первого и второго классов. Наследование от первого класса закрытого типа, от второго класса – открытого типа. При создании конструктора наследного класса организован вызов конструкторов обоих классов-родителей.

Полиморфизм

Ранее было определено, что *полиморфизм* – это способность объектов с одним интерфейсом иметь различную реализацию. Рассмотрим основные виды полиморфизма.

Простой (статический) полиморфизм реализуется с помощью механизма переопределения (перегрузки) функций. Поэтому такие полиморфные функции называются в С++ переопределяемыми. В соответствии с общими правилами переопределения функций они должны отличаться сигнатурой, т. е. количеством, типом и порядком следования передаваемых параметров. *Сложный (динамический)* полиморфизм основан на механизме использования виртуальных функций.

Для перегрузки операций используется ключевое слово operator. Прототип перегруженной операции:

Значения Тип_возвращаемого
operator оператора
Символ (операнды){ тело_функции };

Перегружать операции можно с помощью:

- функции-члена;
- функции-друга;
- глобальной функции (как правило, менее эффективно).

Можно перегружать любые операции языка С++, кроме:

. (операции выбора члена класса);
:: (операции разрешения области видимости);
? : (условной операции);
.* (операции разыменования указателя на член класса);
(директивы препроцессора;
sizeof;
typeid.

Пример:

```
class complex
{
   double re, im;
   public:
  complex(double r=0, double i=0):re(r),im(i){}
  complex operator +(const complex& y);
};
complex complex::operator +(const complex & y)
{
  complex t(re + y.re, im + y.im);
  return t;
}
```

Известны три случая, когда при переопределении методов возникают ошибки некорректного определения типа объекта при компиляции:

1) если наследуемый метод для объекта производного класса вызывает метод, переопределенный в производном классе;
2) если объект производного класса через указатель базового класса обращается к методу, переопределенному производным классом;
3) если процедура вызывает переопределенный метод для объекта производного класса, переданного в процедуру через параметр-переменную, описанный как объект базового класса («процедура с полиморфным объектом»).

Возникает необходимость использовать сложный (динамический) полиморфизм, основанный на механизме использования виртуальных функций.

Виртуальными называют функции, которые объявлены с использованием ключевого слова virtual в базовом классе и переопределяются (замещаются) в одном или нескольких производных классах. При этом прототипы функций в разных классах должны совпадать не только по именам, но и по сигнатуре, хотя алгоритмы, реализуемые такими функциями, различны. Если прототипы функций не совпадают, то механизм виртуальности для них не включается.

Пример:

```
class A
{
  public:
  virtual void f (int x)
  {
    cout << "A::f" << '\n';
   }
};

class C: public A
{
  public:
  void f (int x)
  {
     cout << "C::f" << '\n';
   }
};
```

Друзья класса

Имеется ряд ситуаций, когда объекту одного класса необходимо иметь прямой доступ к закрытым членам объекта другого класса без использования методов-селекторов. Для этого в языке C++ введена концепция друзей и специальное ключевое слово friend..

Друг класса – это функция, не являющаяся членом класса, но имеющая доступ к его закрытым и защищенным членам. Друзья класса объявляются в самом классе с помощью служебного слова friend. в любой области доступа. Другом класса может быть обычная функция, метод другого класса или другой класс (при этом каждый его метод становится другом класса). Другом класса может быть не только метод другого класса, но и внешняя функция. Кроме того, возможна дружественность сразу для нескольких классов. Это необходимо,

например, в случае организации взаимодействия нескольких объектов разных классов, когда функция, обеспечивающая взаимодействие, должна иметь доступ к закрытым компонентам одновременно нескольких объектов. Объявить функцию методом одновременно нескольких классов невозможно, поэтому в стандарте языка C++ предусмотрена возможность объявлять внешнюю по отношению к классу функцию дружественной данному классу. Для этого необходимо в теле класса объявить некоторую внешнюю по отношению к классу функцию с использованием ключевого слова friend:

friend имя_функции (список_формальных_параметров);

Пример:

```
class B;
class D
{
  int x;
  . . .
  friend void func(B &, D &);. . .
};
class B
{
  int y;
  . . .
  friend void func(B &, D &);. . .
};
void func(B & b1, D & d1)
{
  cout << d1.x + b1.y;
}
```

Лекция 2.6

Библиотека STL

STL (Standard Template Library) является частью стандарта C++. Основные компоненты этой библиотеки – иерархии шаблонов классов и функций. Библиотека STL является важной составной частью стандартной библиотеки. STL состоит из четырех основных компонентов:

- контейнеры;
- итераторы;
- алгоритмы;
- распределители памяти (аллокаторы).

Контейнер – тип данных (класс), предназначенный для хранения объектов какого-либо типа (возможна реализация контейнера, который хранит объекты разных типов: в этом случае в нем хранятся указатели на базовый тип для всех желаемых типов, т. е. формально хранятся объекты одного типа, а фактически указатели ссылаются на элементы разных типов из одной иерархии классов).

Итератор – это класс, объекты которого по отношению к контейнерам играют роль указателей. Итераторы поддерживают абстрактную модель совокупности данных как последовательности объектов (что и представляет собой любой контейнер).

Алгоритмы STL (их около 60) реализуют некоторые распространенные операции с контейнерами, которые не реализуются методами каждого из контейнеров (например, просмотр, сортировка, поиск, удаление элементов и прочие). Такие операции являются универсальными для любого из контейнеров, поэтому находятся вне этих контейнеров. Зная, как устроены алгоритмы, можно писать необходимые дополнительные алгоритмы обработки, которые не будут зависеть от контейнера.

Распределитель памяти обеспечивает стандартные способы выделения и перераспределения памяти, а также стандартные имена типов для указателей и ссылок.

Стандартные контейнеры библиотеки STL:

- Vector < T > (динамический массив);
- List < T > (линейный список);
- Stack < T > (стек);
- Queue < T > (очередь);
- Deque < T > (двусторонняя очередь);
- Priority_queue < T > (очередь с приоритетами);
- Set < T > (множество);
- Bitset < N > (множество битов (массив из N бит));
- Multiset < T > (набор элементов, возможно, одинаковых);
- Map < key, val > (ассоциативный массив);
- Multimap < key, val > (ассоциативный массив для хранения пар «ключ–значение», где с каждым ключом может быть связано).

Строго говоря, стек, очередь, очередь с приоритетами не считаются стандартными контейнерами. Они построены с ограничениями функциональности на базе других контейнеров. Тем не менее включены в библиотеку STL наряду с другими стандартными контейнерами.

В каждом классе-контейнере определен набор функций для работы с этим контейнером, причем все контейнеры поддерживают стандартный набор базо-

вых операций (функции, одинаково называющиеся, имеющие одинаковый профиль и семантику, их примерно 15-20). Например, функция push_back() помещает элемент в конец контейнера, функция size() выдает текущий размер контейнера. Основные операции включаются в следующие группы:

- доступ к элементу;
- вставка элемента;
- удаление элемента;
- итераторы.

Операции, которые не могут быть эффективно реализованы для всех контейнеров, не включаются в набор общих операций. Например, обращение по индексу введено для контейнера vector, но не для list.

Каждый контейнер в своей открытой области содержит набор определений стандартных имен типов. Среди них есть следующие имена:

- value_type - тип элемента,
- allocator_type - тип распределителя памяти,
- size_type - тип, используемый для индексации,
- iterator, const_iterator - итератор,
- reverse_iterator, const_reverse_iterator - обратный итератор,
- pointer, const_pointer - указатель на элемент,
- reference, const_reference - ссылка на элемент.

Эти имена определяются внутри каждого контейнера так, как это необходимо для соответствующего контейнера. При этом реальные типы инкапсулированы. Это позволяет писать программы с использованием контейнеров, не зависящие от типов данных, реально используемых в контейнерах.

Пример. Использование контейнера vector. Создается контейнер из целых чисел, заполняется числами 22, 11, 4 и выводится на экран.

```
#include <iostream>
#include <vector>
using namespace std;

void main()
{
   vector <int> k;
   k.push_back(22);
   k.push_back(11);
   k.push_back(4);
   for (int i = 0; i<k.size(); i++)
      cout<<k[i]<<”\n”;
}
```

Каждый контейнер содержит итераторы, поддерживающие стандартный набор итерационных операций со стандартными именами и смыслом. Итератор – это класс, объекты, которого по отношению к контейнерам играют роль указателей. Итераторы поддерживают абстрактную модель совокупности данных как последовательности объектов. Обычно основное действие с последовательностью элементов – перебор. Он организуется с помощью итераторов. Итератор – это класс, чьи объекты выполняют ту же роль по отношению к контейнеру, которую выполняют указатели по отношению к массиву. Указатель может использоваться в качестве средства доступа к элементам массива, а итератор – в качестве средства доступа к элементам контейнера.

Классы итераторов и функции, предназначенные для работы с ними, находятся в библиотечном файле <iterator>. Каждый контейнер содержит ряд ключевых методов, позволяющих найти концы последовательности элементов в виде соответствующих значений итераторов. Это:

- iterator begin() – возвращает итератор, который указывает на первый элемент последовательности;
- const_iterator begin() const iterator end() – возвращает итератор, который указывает на элемент, следующий за последним элементом последовательности (используется при оформлении циклов);
- const_iterator end () const reverse_iterator rbegin() - возвращает итератор, указывающий на первый элемент в обратной последовательности (используется для работы с элементами последовательности в обратном порядке);
- const_reverse_iterator rbegin() const reverse_iterator rend() - возвращает итератор, указывающий на элемент, следующий за последним в обратной последовательности.

Пример:

```
#include <iostream>
#include <vector>
using namespace std;
void main()
{
   vector <int> k;
   k.push_back(22);
   k.push_back(11);
   k.push_back(4);
   vector <int>::iterator p;
   for (p = k.begin(); p < k.end(); p++)
   {
```

```
        cout<<*p<<"\n";
    }
}
```

Алгоритмы STL – это некоторые универсальные алгоритмы, которые можно использовать для большинства контейнеров. Каждый алгоритм представлен шаблоном функции или набором шаблонов функций. Все стандартные алгоритмы находятся в пространстве имен std, а их объявления – в библиотечном файле <algorithm>.

Можно выделить три основные группы алгоритмов:

1. Немодифицирующие алгоритмы, те, которые извлекают информацию из контейнера (о его устройстве, об элементах, которые там есть и т. д.), но никак не модифицируют сам контейнер (ни элементы, ни порядок их расположения). Например, find() – поиск первого вхождения элемента с заданным значением; count() – количество вхождений элемента с заданным значением; for_each() – для применения некоторой операции к каждому элементу, не изменяющей элементы контейнера.
2. Модифицирующие алгоритмы, которые каким-либо образом изменяют содержимое контейнера. Либо сами элементы меняются, либо их порядок, либо их количество. Например, transform() – для применения некоторой операции к каждому элементу, изменяющей элементы контейнера в отличие от алгоритма; for_each reverse() – переставляет элементы в последовательности; copy() – создает новый контейнер
3. Сортировка. Например, sort() – простая сортировка; stable_sort() – сохраняет порядок следования одинаковых элементов; merge() – объединяет две отсортированные последовательности.

STL-подход обладает достоинствами и недостатками. Среди достоинств можно отметить:

- каждый контейнер обеспечивает стандартный интерфейс в виде набора операций, так что один контейнер может использоваться вместо другого, причем это не влечет существенного изменения кода;
- дополнительная общность использования обеспечивается через стандартные итераторы;
- каждый контейнер связан с распределителем памяти (аллокатором), который можно переопределить с тем, чтобы реализовать собственный механизм распределения памяти;
- для каждого контейнера можно определить дополнительные итераторы и интерфейсы, что позволит оптимальным образом настроить его для решения конкретной задачи;

– контейнеры по определению однородны, т.е. должны содержать элементы одного типа, но возможно создание разнородных контейнеров как контейнеров, содержащих указатели на общий базовый класс;
– алгоритмы, входящие в состав STL, предназначены для работы с содержимым контейнеров. Все алгоритмы представляют собой шаблонные функции, следовательно, их можно использовать для работы с любым контейнером.

Недостатки таковы:

– контейнеры не имеют фиксированного стандартного представления. Они не являются производными от некоторого базового класса. Это же верно и для итераторов. Использование стандартных контейнеров и итераторов не подразумевает никакой явной или неявной проверки типов во время выполнения;
– каждый доступ к итератору приводит к вызову виртуальной функции. Эти затраты по сравнению с вызовом обычной функции могут быть значительными;
– предотвращение выхода за пределы контейнера по-прежнему возлагается на программиста, при этом каких-то специальных средств для такого контроля не предлагается.

Рассмотрим подробнее на примерах некоторые из контейнеров.

Контейнер vector. Для создания вектора вам необходимо подключить <vector>. Затем создание вектора почти ничем не отличается от создания переменной и/или массива:

vector <тип элементов> имя_вектора;

Для записи в вектор достаточно набрать:

имя вектора.push_back(значение).

При выполнении этой операции будет происходить динамическое выделение памяти под данный элемент вектора и занесение в эту область памяти указанного значения.

Например:

```
vector <int> test;
test.push_back(10);
test.push_back(20);
```

Обращение к n-му элементу ничем не отличается от обращения к элементу массива:

```
test[0]++;
cout<<test[1];
test[1]=222;
```

Для удаления последнего элемента вектора используется функция pop_back():

test.pop_back();

Еще ряд функций:

test.at(i) – равносильно записи test[i], но при этом, если i-го элемента не существует, программа не вылетит;

test.asign(n,m) – записывает в массив n элементов со значением m;

test.asign(start,end) – записывает в вектор значения от start до end (start и end - итераторы на элементы другого вектора);

test.front() – возвращает ссылку на первый элемент;

test.back() – возвращает ссылку на последний элемент;

test.begin() – возвращает итератор первого элемента вектора;

test.end() – возвращает итератор последнего элемента вектора;

test.clear() – очищает вектор;

test.erase(i) или test.erase(start,end) – удаляет элемент с итератором i или элементы с интераторами между start и end;

test.size() – возвращает количество элементов в векторе;

test.swap(test2) – меняет местами содержимое вектора test и вектора test2;

test.insert(a,b) – вставляет в вектор test переменную b перед элементом с итератором a и возвращает итератор вставленного элемента;

test.insert(a,n,b) – вставляет n копий b перед элементом с итератором a.

Контейнер deque. Подключение библиотеки:

deque <тип элементов> имя_дека;

Типовые операции:

front – возврат значения первого элемента;

back – возврат значения последнего элемента;

push_front – добавление элемента в начало;

push_back – добавление элемента в конец;

pop_front – удаление первого элемента;

pop_back – удаление последнего элемента;

size – возврат числа элементов дека;

clear – очистка дека.

Пример. Создать дек и вывести его на экран двумя способами (через указатель и через прямой доступ).

```
#include <iostream>
#include <deque>
#include <iterator>
using namespace std;
```

```
int main()
{
   setlocale(LC_ALL,»Rus»);
   int dequeSize = 0,k;
   cout << "Введите размер дека: ";
   cin >> dequeSize;
   deque <int> myDeque;
   deque <int>::iterator out;
       cout << "Введите элементы дека: ";
   for (int i = 0; i < dequeSize; i++) {
      cin >> k;
              myDeque.push_back(k);
   }
    cout << "\nВведенный дек: ";
       out=myDeque.begin();
   for (out = myDeque.begin(); out < myDeque.end(); out++)
          {
     cout<<*out;
        }
       cout<<endl;
       for (int i = 0; i < dequeSize; i++)
          {
     cout<<myDeque[i];
        }
   return 0;
}
```

Контейнер list (двусвязный список). По своей структуре списки сильно отличаются от векторов и деков, хотя они поддерживают почти весь набор операций, характерных для деков и векторов. Однако, кроме того, в списках есть набор специфических функций. Чтобы воспользоваться контейнером списков в С++, вам необходимо подключить следующий заголовочный файл:

```
#include <list>
```

Дополнительные функции: сцепка списков (splice) служит для перемещения элементов из одного списка в другой без перераспределения памяти, только за счет изменения указателей:

splice(позиция начала вставки, имя списка)

splice(позиция начала вставки, имя второго списка, итератор вставляемого элемента);

splice(позиция начала вставки, имя второго списка, итератор 1 вставляемого элемента, итератор 2 вставляемого элемента).

Оба списка должны содержать элементы одного типа.

Первая форма функции вставляет в вызывающий список перед элементом, позиция которого указана первым параметром, все элементы списка, указанного вторым параметром. Второй список остается пустым. Нельзя вставить список в самого себя. Вторая форма функции переносит элемент, позицию которого определяет третий параметр, из списка x в вызывающий список. Допускается переносить элемент в пределах одного списка. Третья форма функции аналогичным образом переносит из списка в список несколько элементов. Их диапазон задается третьим и четвертым параметрами функции. Если для одного и того же списка первый параметр находится в диапазоне между третьим и четвертым, результат не определен.

Пример. Исходный список 1 2 3 4 5 12 13. Список после сцепки 1 12 13 2 3 4 5.

```
#include <list>
#include <iostream>
using namespace std;

int main()
{
list <int> L1;
list <int>::iterator i, j, k;
for (int i = 0; i<5; i++) L1.push_back(i + 1);
for (int i = 12; i<14; i++) L1.push_back(i);
cout <<”Исходный список: “;
for (i = L1.begin() ; i!= L1.end(); ++i)
        cout <<*i<< “ “;
cout <<endl;
i = L1.begin();
i++;
k = L1.end();
j = --k;
k++;
j--;
L1.splice(i,L1,j,k);
cout <<”Список после сцепки: “;
for (i = L1.begin(); i != L1.end(); ++i)
cout <<*i <<” “;
```

}

Для *удаления* элемента по его значению используется функция remove(значение). Если в списке несколько повторяющихся значений, то они будут удалены все.

Для *упорядочения* списков используется метод sort():

L1.sort();

Метод unique() сортирует список, удаляя из него повторяющиеся элементы.

Для слияния списков используется метод merge(второй список). Оба сливаемых списка должны быть упорядочены.

Метод reverse() меняет порядок следования элементов на обратный.

Контейнер stack. Для создания стека нужно подключить <stack> и в коде программы его объявить:

stack <type> name, где type - тип стека, а name - имя стека.

У стека есть немного функций:

push() - добавить элемент

pop() - удалить верхний элемент

top() - получить верхний элемент

size() - размер стека

empty() - true, если стек пуст

Пример. Считываем слова с клавиатуры, пока не встретится слово «end» и выводим их в обратном порядке.

```
#include <iostream>
#include <string>
#include <stack>
using namespace std;

void main()
{
string s;
stack <string> st;
s=««;
while (s.compare(«end»))
{
cin>>s;
st.push(s);
}
while (!(st.empty()))
{
```

```
cout<<st.top()<<endl;
st.pop();
}
}
```

Контейнер queue (очередь). Очереди, как следует из названия, используют принцип first in first out (FIFO). Т. е. тот, кого мы первым «запихнули» в очередь, первым из нее и выйдет. Реализуются очереди также просто. Подключаем <queue> и создаем очередь:

queue <type> name;

Перечень функций почти тот же, что и у стека:

push() - добавить элемент

pop() - удалить первый элемент очереди

size() - размер очереди

empty() - true, если очередь пуста

front() - получить первый элемент

back() - получить последний элемент

Пример. Аналогичный предыдущему.

```
#include <iostream>
#include <string>
#include <queue>
using namespace std;

void main()
{
string s;
queue <string> st;
s="";
while (s.compare("end"))
{
cin>>s;
st.push(s);
}
while (!(st.empty()))
{
cout<<st.front()<<endl;
st.pop();
}
}
```

Список литературы

1. *Альчаков В. В.* Исследование оптимальных кодов. Код Шеннона-Фано. Код Хаффмена: методические указания к лаб.работам по дисциплине «Теория информации и кодирования». – Севастополь, 2014. – 8 с.
2. *Андрианова А. А., Исмагилов Л. Н., Мухтарова Т. М.* Объектно-ориентированное проггграммирование на С++: учебное пособие. – Казань, 2010. – 230 с.
3. *Волкова И. А., Иванов А. В., Карпов Л. Е.* Основы объектно-ориентированного программирования. Язык программирования С++: учебное пособие для студентов 2 курса. – М., 2011. – 112 с.
4. *Жданова Т. А., Бузыкова Ю. С.* Основные подходы к определению понятия «ИНФОРМАТИКА» // Вестник ХГАЭП. 2012. – № 3 (60). – С. 34-38.
5. *Иванова Г. С., Ничушкина Т. Н.* Объектно-ориентированное программирование на языке С++ в среде Visual Studio 2008: электронное учебное пособие. – М., 2012.
6. *Карасева О. А.* Информатика и программирование: курс лекций для направлений «Прикладная информатика» и «Бизнес-информатика». – Екатеринбург, 2012. – 143 с.
7. *Луковкин С.Б.* Теоретические основы информатики: учебное пособие. – Мурманск: Изд-во МГТУ, 2008. – 125 с.
8. *Малев В. В.* Общая методика преподавания информатики:учебное пособие. – Воронеж, 2005. – 272 с.
9. *Острейковский В. А., Полякова И. В.* Информатика. Теория и практика: учебное пособие. – М.: Издательство Оникс, 2008. – 608 с.
10. *Павловская Т. А.* С/С++. Программирование на языке высокого уровня. – СПб.: Питер, 2003. – 461 с.
11. *Поляков В.И., Скорубский В.И.* Основы теории алгоритмов. – СПб, 2012. – 51 с.
12. *Семакин И. Г., Хеннер Е. К.* Информатика и ИКТ. Базовый уровень: учебник для 10-11 классов. 8-е изд. –М.: БИНОМ, Лаборатория знаний, 2012.
13. *Семакин И. Г., Русакова О. Л., Тарунин Е.Л., Шкарапута А.П.* Программирование, численные методы и математическое моделирование [Электронный ресурс]: учебное пособие. – М.: Кнорус, 2017. – 298 с. – URL: https://www.book.ru/book/920222/
14. *Семакин И. Г., Шестаков А. П.* Основы программирования: учебник. – М.: Мастерство, 2002. – 432 с.
15. *Симонович С., Евсеев Г.* Практическая информатика: универсальный курс. – М.: АСТ-ПРЕСС, 2000.

16. *Струуструп Б.* Язык программирования C++. Специальное издание. Пер. с англ. – М.: Бином, 2011. – 1136 с.
17. *Рублев В.С.* Основы теории алгоритмов: учеб. пособие. – Ярославль, 2005. – 143 с.
18. *Чепкунова Е.Г.* Пособие к подготовке к экзамену по дисциплине «Теоретические основы информатики»: учеб.пособие. – Казань, 2012. – 92 с.
19. *Чернавский Д. С.* Синергетика и информация. Динамическая теория хаоса. М.: Наука, 2001. – 105 с.
20. URL: http://pmi34psu.blogspot.ru/p/blog-page_29.html

Учебное издание

Бузмакова Мария Михайловна

ИНФОРМАТИКА И ОСНОВЫ ПРОГРАММИРОВАНИЯ

КУРС ЛЕКЦИЙ

Учебное пособие

Редактор *Л. Г. Подорова*
Корректор *Л. И. Иванова*
Компьютерная верстка: *М. М. Бузмакова*

Подписано в печать 05.12.2017. Формат 60×84/16
Усл. печ. л. 10,46. Тираж 100 экз. Заказ ___

Издательский центр
Пермского государственного
национального исследовательского университета
614990, г. Пермь, ул. Букирева, 15

Отпечатано на ризографе
ООО «Учебный центр «Информатика».
614990, г. Пермь, ул. Букирева, 15