МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ
ВЫСШЕГО ОБРАЗОВАНИЯ

«БРАТСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»

Кафедра управления в технических системах

УТВЕРЖДАЮ:
Проректор по учебной работе
_____ Е.И. Луковникова
«31» мая 2019 г.

РАБОЧАЯ ПРОГРАММА ДИСЦИПЛИНЫ

ПРОГРАММИРОВАНИЕ И ОСНОВЫ АЛГОРИТМИЗАЦИИ

Б1.Б.14

НАПРАВЛЕНИЕ ПОДГОТОВКИ

27.03.04 Управление в технических системах

ПРОФИЛЬ ПОДГОТОВКИ

Управление и информатика в технических системах

Программа академического бакалавриата

Квалификация (степень) выпускника: бакалавр

Программа составлена в соответствии с федеральным государственным образовательным стандартом высшего образования по направлению подготовки 27.03.04 Управление в технических системах от 20.10.2015 г № 1171 и учебным планом ФГБОУ ВО «БрГУ» от 01.04.2019 г № 196 для заочной формы обучения набора 2019 года

СОДЕРЖАНИЕ ПРОГРАММЫ

1. ПЕРЕЧЕНЬ ПЛАНИРУЕМЫХ РЕЗУЛЬТАТОВ ОБУЧЕНИЯ ПО ДИСЦИПЛИНЕ, СООТНЕСЕННЫХ С ПЛАНИРУЕМЫМИ РЕЗУЛЬТАТАМИ ОСВОЕНИЯ ОБРАЗОВАТЕЛЬНОЙ ПРОГРАММЫ

Вид деятельности выпускника

Дисциплина охватывает круг вопросов, относящихся к научно-исследовательскому и проектно-конструкторскому видам профессиональной деятельности выпускника в соответствии с компетенциями и видами деятельности, указанными в учебном плане.

Цель дисциплины

Целью изучения дисциплины является формирование у студентов знаний и навыков по использованию современных технологий и методов разработки программных систем для решения практических задач с использованием современных инструментальных средств, необходимых в дальнейшем, при проектировании и эксплуатации систем управления и автоматизации.

Задачи дисциплины

Задачи данной дисциплины состоят в обучении свободному владению языком программирования как “средством выражения” алгоритмов применительно к традиционному кругу задач - арифметико-логическим, сортировки и поиска, приближенным вычислений, обработки текста.

Код компетенции	Содержание компетенций	Перечень планируемых результатов обучения по дисциплине
ОК-7	способность к самоорганизации и самообразованию	**знать:** - содержание процессов самоорганизации и самообразования, их особенностей и технологий реализации, исходя из целей совершенствования профессиональной деятельности; **уметь:** - планировать цели и устанавливать приоритеты при выборе способов принятия решений с учетом условий, средств, личностных возможностей и временной перспективы осуществления деятельности; **владеть:** - приемами саморегуляции эмоциональных и функциональных состояний при выполнении профессиональной деятельности.
ОПК-9	способность использовать навыки работы с компьютером, владеть методами информационных технологий, соблюдать основные требования информационной безопасности	**знать:** - базовое устройство персонального компьютера; основные информационные процессы, происходящие в персональном компьютере; **уметь:** - использовать персональный компьютер для самостоятельной работы; **владеть:** - достаточным уровнем использования универсальных пакетов прикладных компьютерных программ.

2. МЕСТО ДИСЦИПЛИНЫ В СТРУКТУРЕ ОБРАЗОВАТЕЛЬНОЙ ПРОГРАММЫ

Дисциплина Б1.Б.14 Программирование и основы алгоритмизации относится к базовой.

Дисциплина программирование и основы алгоритмизации базируется на знаниях, полученных при изучении дисциплин основных общеобразовательных программ.

Основываясь на изучении перечисленных дисциплин, программирование и основы алгоритмизации представляет основу для изучения дисциплин: Б1.В.15 Структуры и алгоритмы обработки данных, Б1.Б.13 Вычислительные машины, системы и сети, Б1.В.18 Технологии программирования, Б1.В.ДВ.4.1 Прикладное программирование.

Такое системное междисциплинарное изучение направлено на достижение требуемого ФГОС уровня подготовки по квалификации бакалавр.

3. РАСПРЕДЕЛЕНИЕ ОБЪЕМА ДИСЦИПЛИНЫ

3.1. Распределение объема дисциплины по формам обучения

Форма обучения	*Курс*	*Семестр*	*Трудоемкость дисциплины в часах*						*Курсовая работа*	*Вид промежуточной аттестации*
			Всего часов (с экз.)	*Аудиторных часов*	*Лекции*	*Лабораторные работы*	*Практические занятия*	*Самостоятельная работа*		
1	2	3	4	5	6	7	8	9	10	11
Очная	1	2	144	54	18	36	-	54	кр	Экзамен
Заочная	1	-	144	18	6	12	-	126	кр	Экзамен
Заочная (ускоренное обучение)	1	-	144	10	4	6	-	134	кр	Экзамен
Очно-заочная	-	-	-	-	-	-	-	-	-	-

3.2. Распределение объема дисциплины по видам учебных занятий и трудоемкости:

Вид учебных занятий	*Трудоемкость (час.)*	*в т.ч. в интерактивной, активной, инновационной формах, (час.)*	*Распределение по семестрам, час*
			2
I. Контактная работа обучающихся с преподавателем (всего)	54	12	54
Лекции (Лк)	18	6	18
Лабораторные работы (ЛР)	36	6	36

Контрольная работа (кр)	+	-	-
Индивидуальные (групповые) консультации	+	-	+
II.Самостоятельная работа обучающихся (СР)	54	-	54
Подготовка к лабораторным работам	12	-	12
Подготовка к экзамену в течение семестра	12	-	12
Выполнение курсовой работы	30	-	30
III. Промежуточная аттестация экзамен	36	-	36
Общая трудоемкость дисциплины ……. час.	144	-	144
зач. ед.	4	-	4

4. СОДЕРЖАНИЕ ДИСЦИПЛИНЫ

4.1. Распределение разделов дисциплины по видам учебных занятий - для очной формы обучения:

№ раз-дела и темы	*Наименование раздела и тема дисциплины*	*Трудоем-кость, (час.)*	*Виды учебных занятий, включая самостоятельную работу обучающихся и трудоемкость; (час.)*		
			учебные занятия		*самостоятельная работа обучаю-щихся*
			лекции	*лабораторные работы*	
1	2	3	4	5	6
1.	**Инструментарий технологии программирования**	**9**	**3**	**-**	**6**
1.1.	Программа как формализованное описание процесса обработки данных	1,5	0,5	-	1
1.2.	Технология программирования как инструмент разработки надежных программных средств....	1	-	-	1
1.3.	Специфика разработки программных средств	1	-	-	1
1.4.	Жизненный цикл программного средства	1,5	0,5	-	1
1.5.	Понятие качества программного средства	2	1	-	1
1.6.	Виды современных языков программирования	2	1	-	1
2.	**Основные принципы и подходы к разработке программных алгоритмов**	**18**	**2**	**8**	**8**
2.1.	Понятие алгоритма и его свойства	5,5	0,5	2	3
2.2.	Классы алгоритмов и способы их описания	5,5	0,5	3	2
2.3.	Основные алгоритмические конструкции	7	1	3	3

3.	**Основы программирования на языке высокого уровня Си++**	**17**	**3**	**8**	**6**
3.1.	Архитектура программного средства	4	1	2	1
3.2.	Проектирование программного обеспечения при структурном подходе	3	1	1	1
3.3.	Этапы работы с программой на языке си/си++ в системе программирования	4	-	2	2
3.4.	Разработка структуры программы и модульное программирование	4	1	2	1
3.5.	Синтаксис и семантика языка	2	-	1	1
4.	**Типы данных, выражения и операции в языке программирования с++**	**11**	**2**	**2**	**7**
4.1.	Концепция типов данных в языке с++	2,5	0,5	-	2
4.2.	Арифметические типы данных	2,5	0,5	-	2
4.3.	Константы и переменные	2,5	0,5	1	1
4.4.	Основные операции языка си	3,5	0,5	1	2
5.	**Операторы языка программирования Си++ и управление их исполнением**	**16**	**2**	**8**	**6**
5.1.	Операторы условия и передачи управления	4,5	0,5	2	2
5.2.	Операторы циклов	6	1	3	2
5.3.	Одномерные и многомерные массивы	5,5	0,5	3	2
6.	**Указатели и динамическое распределение памяти**	**16**	**2**	**7**	**7**
6.1.	Указатели, ссылки и адресная арифметика	4,5	0,5	2	2
6.2.	Операторы динамического распределения памяти	3,5	0,5	2	1
6.3.	Динамическое размещение массивов	3,5	0,5	1	2
6.4.	Типы строк в языке с++	4,5	0,5	2	2
7.	**Функции языка программирования Си++**	**11**	**2**	**3**	**6**
7.1.	Объявление и описание функций	2,5	0,5	1	1
7.2.	Функции с переменным количеством параметров	1,5	0,5		1
7.3.	Рекурсивные функции	3,5	0,5	1	2
7.4.	Передача параметров в функцию по ссылке	3,5	0,5	1	2
8.	**Статические и динамические структуры данных**	**10**	**2**	**-**	**8**
8.1.	Статические структуры данных	3,5	0,5	-	3
8.2.	Полустатические структуры данных	2,5	0,5	-	2
8.3.	Динамические структуры данных	4	1	-	3

	ИТОГО	**108**	**18**	**36**	**54**

- для заочной формы обучения:

№ раз-дела и темы	*Наименование раздела и тема дисциплины*	*Трудоем-кость, (час.)*	*Виды учебных занятий, включая самостоятельную работу обучающихся и трудоемкость; (час.)*		
			учебные занятия		*самостоя тельная работа обучаю-щихся*
			лекции	*лабораторные работы*	
1	2	3	4	5	6
1.	**Инструментарий технологии программирования**	**14**	**-**	**-**	**14**
1.1.	Программа как формализованное описание процесса обработки данных	2	-	-	2
1.2.	Технология программирования как инструмент разработки надежных программных средств....	2	-	-	2
1.3.	Специфика разработки программных средств	2	-	-	2
1.4.	Жизненный цикл программного средства	2	-	-	2
1.5.	Понятие качества программного средства	2	-	-	2
1.6.	Виды современных языков программирования	4	-	-	4
2.	**Основные принципы и подходы к разработке программных алгоритмов**	**18**	**2**	**2**	**14**
2.1.	Понятие алгоритма и его свойства	5	1	-	4
2.2.	Классы алгоритмов и способы их описания	6	1	1	4
2.3.	Основные алгоритмические конструкции	7	-	1	6
3.	**Основы программирования на языке высокого уровня Си++**	**18**	**-**	**4**	**14**
3.1.	Архитектура программного средства	3	-	-	3
3.2.	Проектирование программного обеспечения при структурном подходе	4	-	1	3
3.3.	Этапы работы с программой на языке си/си++ в системе программирования	4	-	1	3
3.4.	Разработка структуры программы и модульное программирование	4	-	1	3
3.5.	Синтаксис и семантика языка	**3**	-	1	2
4.	**Типы данных, выражения и операции в языке программирования с++**	**18**	**2**	**2**	**14**

4.1.	Концепция типов данных в языке с++	4	1	-	3
4.2.	Арифметические типы данных	4	1	-	3
4.3.	Константы и переменные	4	-	1	3
4.4.	Основные операции языка си	6	-	1	5
5.	**Операторы языка программирования Си++ и управление их исполнением**	**18**	**2**	**2**	**14**
5.1.	Операторы условия и передачи управления	5	1	-	4
5.2.	Операторы циклов	6	1	1	4
5.3.	Одномерные и многомерные массивы	7	-	1	6
6.	**Указатели и динамическое распределение памяти**	**16**	**-**	**2**	**14**
6.1.	Указатели, ссылки и адресная арифметика	4	-	1	3
6.2.	Операторы динамического распределения памяти	4	-	1	3
6.3.	Динамическое размещение массивов	3	-	-	3
6.4.	Типы строк в языке с++	5	-	-	5
7.	**Функции языка программирования Си++**	**19**	**-**	**-**	**19**
7.1.	Объявление и описание функций	5	-	-	5
7.2.	Функции с переменным количеством параметров	3	-	-	3
7.3.	Рекурсивные функции	6	-	-	6
7.4.	Передача параметров в функцию по ссылке	5	-	-	5
8.	**Статические и динамические структуры данных**	**14**	**-**	**-**	**14**
8.1.	Статические структуры данных	4	-	-	4
8.2.	Полустатические структуры данных	4	-	-	4
8.3.	Динамические структуры данных	6	-	-	6
	ИТОГО	**135**	**6**	**12**	**117**

- для заочной формы обучения (ускоренное обучение)**:**

№ раз-дела и темы	***Наименование раздела и тема дисциплины***	***Трудоем-кость, (час.)***	***Виды учебных занятий, включая самостоятельную работу обучающихся и трудоемкость; (час.)***		
			учебные занятия		***самостоя тельная работа обучаю-щихся***
			лекции	***лабораторные работы***	
1	**2**	**3**	**4**	**5**	**6**
1.	**Инструментарий технологии программирования**	**4**	**-**	**-**	**4**

1.1.	Программа как формализованное описание процесса обработки данных	0,5	-	-	0,5
1.2.	Технология программирования как инструмент разработки надежных программных средств....	0,5	-	-	0,5
1.3.	Специфика разработки программных средств	0,5	-	-	0,5
1.4.	Жизненный цикл программного средства	0,5	-	-	0,5
1.5.	Понятие качества программного средства	1	-	-	1
1.6.	Виды современных языков программирования	1	-	-	1
2.	**Основные принципы и подходы к разработке программных алгоритмов**	**10**	**1**	**2**	**7**
2.1.	Понятие алгоритма и его свойства	2	-	-	2
2.2.	Классы алгоритмов и способы их описания	2	-	-	2
2.3.	Основные алгоритмические конструкции	6	1	2	3
3.	**Основы программирования на языке высокого уровня Си++**	**10**	**1**	**2**	**7**
3.1.	Архитектура программного средства	1	-	-	1
3.2.	Проектирование программного обеспечения при структурном подходе	1	-	-	1
3.3.	Этапы работы с программой на языке си/си++ в системе программирования	1	-	-	1
3.4.	Разработка структуры программы и модульное программирование	3	-	1	2
3.5.	Синтаксис и семантика языка	4	1	1	2
4.	**Типы данных, выражения и операции в языке программирования с++**	**7**	**-**	**-**	**7**
4.1.	Концепция типов данных в языке с++	1	-	-	1
4.2.	Арифметические типы данных	2	-	-	2
4.3.	Константы и переменные	2	-	-	2
4.4.	Основные операции языка си	2	-	-	2
5.	**Операторы языка программирования Си++ и управление их исполнением**	**11**	**2**	**2**	**7**
5.1.	Операторы условия и передачи управления	2	-	-	2
5.2.	Операторы циклов	5	2	1	2
5.3.	Одномерные и многомерные массивы	4	-	1	3

6.	**Указатели и динамическое распределение памяти**	**7**	-	-	**7**
6.1.	Указатели, ссылки и адресная арифметика	1	-	-	1
6.2.	Операторы динамического распределения памяти	2	-	-	2
6.3.	Динамическое размещение массивов	2	-	-	2
6.4.	Типы строк в языке с++	2	-	-	2
7.	**Функции языка программирования Си++**	**7**	-	-	**7**
7.1.	Объявление и описание функций	1	-	-	1
7.2.	Функции с переменным количеством параметров	2	-	-	2
7.3.	Рекурсивные функции	2	-	-	2
7.4.	Передача параметров в функцию по ссылке	2	-	-	2
8.	**Статические и динамические структуры данных**	**7**	-	-	**7**
8.1.	Статические структуры данных	2	-	-	2
8.2.	Полустатические структуры данных	2	-	-	2
8.3.	Динамические структуры данных	3	-	-	3
	ИТОГО	**72**	**4**	**6**	**53**

4.2. Содержание дисциплины, структурированное по разделам и темам

1. ИНСТРУМЕНТАРИЙ ТЕХНОЛОГИИ ПРОГРАММИРОВАНИЯ

Лекция проводится в интерактивной форме с разбором конкретных примеров.

ПРОГРАММА КАК ФОРМАЛИЗОВАННОЕ ОПИСАНИЕ ПРОЦЕССА ОБРАБОТКИ ДАННЫХ.

Целью программирования является описание процессов обработки данных (в дальнейшем — просто процессов). Согласно ИФИПа [1]: данные — это представление фактов и идей в формализованном виде, пригодном для передачи и переработке в некоем процессе, а информация — это смысл, который придается данным при их представлении. Обработка данных — это выполнение систематической последовательности действий с данными. Данные представляются и хранятся на т.н. носителях данных. Совокупность носителей данных, используемых при какой- либо обработке данных, будем называть информационной средой. Набор данных, содержащихся в какой-либо момент в информационной среде, будем называть состоянием этой информационной среды. Процесс можно определить как последовательность сменяющих друг друга состояний некоторой информационной среды.

Описать процесс — означает определить последовательность состояний заданной информационной среды. Если мы хотим, чтобы по заданному описанию требуемый процесс порождался автоматически на каком-либо компьютере, необходимо, чтобы это описание было формализованным. Такое описание называется программой. С другой стороны, программа должна быть понятной и человеку, так как и при разработке программ, и при их использовании часто приходится выяснять, какой именно процесс она порождает. Поэтому программа составляется на удобном для человека формализованном языке программирования, с которого она автоматически переводится на язык соответствующего компьютера с помощью другой программы, называемой транслятором. Человеку (программисту), прежде чем составить программу на удобном для него языке программирования, приходится проделывать большую подготовительную работу по уточнению постановки задачи, выбору метода ее решения, выяснению специфики применения требуемой программы, прояснению общей организации разрабатываемой программы и многое другое. Использование этой информации может существенно упростить задачу понимания программы человеком, поэтому весьма полезно ее как-то фиксировать в виде отдельных документов (часто не формализованных, рассчитанных только для восприятия человеком).

Обычно программы разрабатываются в расчете на то, чтобы ими могли пользоваться люди, не участвующие в их разработке (их называют пользователями). Для освоения программы пользователем помимо ее текста требуется определенная дополнительная документация. Программа или логически связанная совокупность программ на носителях данных, снабженная программной документацией, называется программным средством (ПС). Программа позволяет осуществлять некоторую автоматическую обработку данных на компьютере. Программная документация позволяет понять, какие функции выполняет та или иная программа ПС, как подготовить исходные данные и запустить требуемую программу в процесс ее выполнения, а также: что означают получаемые результаты (или каков эффект выполнения этой программы). Кроме того, программная документация помогает разобраться в самой программе, что необходимо, например, при ее модификации.

ТЕХНОЛОГИЯ ПРОГРАММИРОВАНИЯ КАК ИНСТРУМЕНТ РАЗРАБОТКИ НАДЕЖНЫХПРОГРАММНЫХ СРЕДСТВ

В соответствии с обычным значением слова «технология» [6] под технологией программирования будем понимать совокупность производственных процессов, приводящую к созданию требуемого ПС, а также описание этой совокупности процессов. Другими словами, технологию программирования мы будем понимать здесь в широком

смысле как технологию разработки программных средств, включая в нее все процессы, начиная с момента зарождения идеи этого средства, и, в частности, связанные с созданием необходимой программной документации. Каждый процесс этой совокупности базируется на использовании каких-либо методов и средств, например, компьютер (в этом случае будем говорить об компьютерной технологии программирования).

В литературе имеются и другие, несколько отличающиеся, определения технологии программирования. Эти определения обсуждаются в работе [7]. Используется в литературе и близкое к технологии программирования понятие программной инженерии, определяемой как систематический подход к разработке, эксплуатации, сопровождению и изъятию из обращения программных средств [7]. Именно программной инженерии (SoftwareEngineering) посвящена упомянутая работа [3]. Главное различие между технологией программирования и программной инженерией как дисциплинами для изучения заключается в способе рассмотрения и систематизации материала. В технологии программирования акцент делается на изучении процессов разработки ПС (технологических процессов) и порядке их прохождения - методы и инструментальные средства разработки ПС используются в этих процессах (их применение и образуют технологические процессы). Тогда как в программной инженерии изучаются прежде всего методы и инструментальные средства разработки ПС с точки зрения достижения определенных целей — они могут использоваться в разных технологических процессах (и в разных технологиях программирования); как эти методы и средства образуют технологические процессы — здесь вопрос второстепенный.

Не следует также пугать технологию программирования с методологией программирования [8]. Хотя в обоих случаях изучаются методы, но в технологии программирования методы рассматриваются «сверху» (с точки зрения организации технологических процессов), а в методологии программирования методы рассматриваются «снизу» (с точки зрения основ их построения). В работе [9, стр. 25] методология программирования определяется как совокупность механизмов, применяемых в процессе разработки программного обеспечения и объединенных одним общим философским подходом.

Имея ввиду, что надежность является неотъемлемым атрибутом ПС, мы будем обсуждать технологию программирования как технологию разработки надёжных ПС. Это означает, что, во- первых, мы будем обсуждать все процессы разработки ПС, начиная с момента возникновения замысла ПС. Во-вторых, нас будут интересовать не только вопросы построения программных конструкций, но и вопросы описания функций и принимаемых решений с точки зрения их человеческого (неформального) восприятия, и, наконец, в качестве продукта технологии мы будем принимать надежную (а не правильную) ПС. Все это будет существенно влиять на выбор методов и инструментальных средств в процессах разработки ПС.

СПЕЦИФИКА РАЗРАБОТКИ ПРОГРАММНЫХ СРЕДСТВ

Разработке программных средств присущ ряд специфических особенностей [1]:

- Прежде всего, следует отметить некоторое противостояние: неформальный характер требований к ПС (постановки задачи) и понятия ошибки в нем, но формализованный основной объект разработки — программы ПС. Тем самым разработка ПС содержит определенные этапы формализации, а переход от неформального к формальному существенно неформален.
- Разработка ПС носит существенно творческий характер (на каждом шаге приходится делать какой-либо выбор, принимать какое-либо решение), а не сводится к выполнению какой- либо последовательности регламентированных действий. Тем самым эта разработка ближе к процессу проектирования каких-либо сложных устройств, но никак не к их массовому производству. Этот творческий характер разработки ПС сохраняется до самого ее конца.
- Следует отметить также особенность продукта разработки. Он представляет собой

некоторую совокупность текстов (т.е. статических объектов), смысл же (семантика) этих текстов выражается процессами обработки данных и действиями пользователей, запускающих эти процессы (т.е. является динамическим). Это предопределяет выбор разработчиком ряда специфичных приемов, методов и средств.

- Продукт разработки имеет и другую специфическую особенность: ПС при своем использовании (эксплуатации) не расходуется и не расходует используемых ресурсов.

ЖИЗНЕННЫЙ ЦИКЛ ПРОГРАММНОГО СРЕДСТВА

Под жизненным циклом ПС понимают весь период его разработки и эксплуатации (использования), начиная от момента возникновения замысла ПС и кончая прекращением всех видов его использования [1, 2, 3, 4]. Жизненный цикл включает все процессы создания и использования ПС (softwareprocess).

Различают следующие стадии жизненного цикла ПС: разработку ПС, производство программных изделий (ПИ) и эксплуатацию ПС.

Стадия разработки (development) ПС состоит из этапа его внешнего описания, этапа конструирования ПС, этапа кодирования (программирование в узком смысле) ПС и этапа аттестации ПС. Всем этим этапам сопутствуют процессы документирования и управление (management) разработкой ПС. Этапы конструирования и кодирования часто перекрываются, иногда довольно сильно. Это означает, что кодирование некоторых частей программного средства может быть начато до завершения этапа конструирования.

Внешнее описание (Requirementsdocument) ПС является описанием его поведения с точки зрения внешнего по отношению к нему наблюдателю с фиксацией требований относительно его качества. Внешнее описание ПС начинается с определения требований к ПС со стороны пользователей (заказчика).

Конструирование (design) ПС охватывает процессы: разработку архитектуры ПС, разработку структур программ ПС и их детальную спецификацию.

Кодирование (coding) — создание текстов программ на языках программирования, их отладку с тестированием ПС.

На этапе аттестации ПС производится оценка качества ПС, после успешного завершения, которого разработка ПС считается законченной.

Программное изделие (ПИ) — экземпляр или копия, снятая с разработанного ПС.

Изготовление ПИ — это процесс генерации и/или воспроизведения (снятия копии) программ и программных документов ПС с целью их поставки пользователю для применения по назначению. Производство ПИ — это совокупность работ по обеспечению изготовления требуемого количества ПИ в установленные сроки [1]. Стадия производства ПС в жизненном цикле ПС является, по существу, вырожденной (несущественной), так как представляет рутинную работу, которая может быть выполнена автоматически и без ошибок. Этим она принципиально отличается от стадии производства различной техники. В связи с этим в литературе эту стадию, как правило, не включают в жизненный цикл ПС.

Стадия эксплуатации ПС охватывает процессы хранения, внедрения и сопровождения ПС, а также транспортировки и применения (operation) ПИ по своему назначению. Она состоит из двух параллельно проходящих фаз: фазы применения ПС и фазы сопровождения ПС [4, 5].

Применение (operation) ПС — это использование ПС для решения практических задач на компьютере путем выполнения ее программ.

Сопровождение (maintenance) ПС —• это процесс сбора информации о его качестве в эксплуатации, устранения обнаруженных в нем ошибок, его доработки и модификации, а также извещения пользователей о внесенных в него изменениях [1,4, 5].

Модель жизненного цикла отражает различные состояния системы, начиная с момента возникновения необходимости в данной ИС и заканчивая моментом ее полного выхода из употребления. *Модель жизненного цикла* - структура, содержащая процессы,

действия и задачи, которые осуществляются в ходе разработки, функционирования и сопровождения программного продукта в течение всей жизни системы, от определения требований до завершения ее использования.

В настоящее время известны и используются следующие модели жизненного цикла:

- *Каскадная модель*(рис. 5.1) предусматривает последовательное выполнение всех этапов проекта в строго фиксированном порядке. Переход на следующий этап означает полное завершение работ на предыдущем этапе.
- *Поэтапная модель* с промежуточным контролем (рис. 5.21. Разработка ИС ведется итерациями с циклами обратной связи между этапами. Межэтапные корректировки позволяют учитывать реально существующее взаимовлияние результатов разработки на различных этапах; время жизни каждого из этапов растягивается на весь период разработки.
- *Спиральная модель* (рис. 5.31. На каждом витке спирали выполняется создание очередной версии продукта, уточняются требования проекта, определяется его качество и планируются работы следующего витка. Особое внимание уделяется начальным этапам разработки - анализу и

проектированию, где реализуемость тех или иных технических решений проверяется \ обосновывается посредством создания прототипов (макетирования).

Рис. 5.1. Каскадная модель ЖЦ ИС

Рис. 5.2. Поэтапная модель с промежуточным контролем

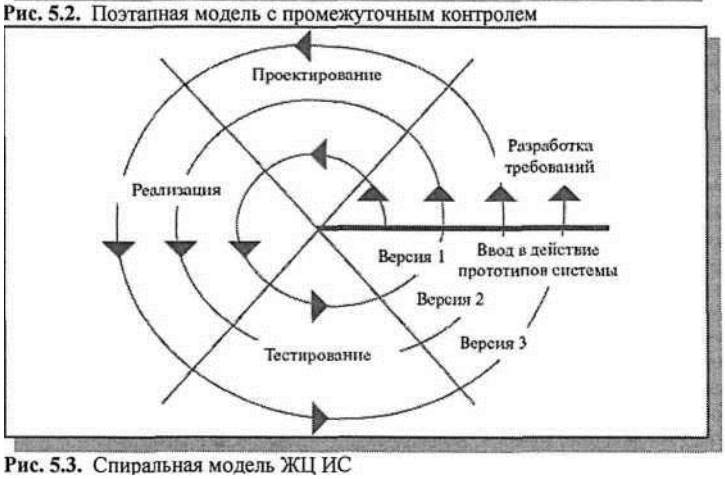


Рис. 5.3. Спиральная модель ЖЦ ИС

На практике наибольшее распространение получили две основные модели жизненного цикла:

- каскадная модель (характерна для периода 1970-1985 гг.);
- спиральная модель (характерна для периода после 1986.Г.).

В ранних проектах достаточно простых ИС каждое приложение представляло

собой единый, функционально и информационно независимый блок. Для разработки такого типа приложений эффективным оказался каскадный способ. Каждый этап завершался после полного выполнения и документального оформления всех предусмотренных работ.

Можно выделить следующие положительные стороны применения каскадного подхода:

- на каждом этапе формируется законченный набор проектной документации, отвечающий критериям полноты и согласованности;
- выполняемые в логической последовательности этапы работ позволяют планировать сроки завершения всех работ и соответствующие затраты.

Каскадный подход хорошо зарекомендовал себя при построении относительно простых ИС, когда в самом начале разработки можно достаточно точно и полно сформулировать все требования к системе. Основным *недостатком этого подхода является* то, что реальный процесс создания системы никогда полностью не укладывается в такую жесткую схему, постоянно возникает потребность в возврате к предыдущим этапам и уточнении или пересмотре ранее принятых решений. В результате реальный процесс создания ИС оказывается соответствующим поэтапной модели с промежуточным контролем.

Однако и эта схема не позволяет оперативно учитывать возникающие изменения и уточнения требований к системе. Согласование результатов разработки с пользователями производится только в точках, планируемых после завершения каждого этапа работ, а общие требования к ИС зафиксированы в виде технического задания на все время ее создания. Таким образом, пользователи зачастую получают систему, не удовлетворяющую их реальным потребностям.

Спиральная модель ЖЦ была предложена для преодоления перечисленных проблем. На этапах анализа и проектирования реализуемость технических решений и степень удовлетворения потребностей заказчика проверяется путем создания прототипов. Каждый виток спирали соответствует созданию работоспособного фрагмента или версии системы. Это позволяет уточнить требования, цели и характеристики проекта, определить качество разработки, спланировать работы следующего витка спирали. Таким образом углубляются и последовательно конкретизируются детали проекта и в результате выбирается обоснованный вариант, который удовлетворяет действительным требованиям заказчика и доводится до реализации.

Итеративная разработка отражает объективно существующий спиральный цикл создания сложных систем. Она позволяет переходить на следующий этап, не дожидаясь полного завершения работы на текущем и решить главную задачу - как можно быстрее показать пользователям системы работоспособный продукт, тем самым активизируя процесс уточнения и дополнения требований.

Основная проблема спирального цикла - определение момента перехода на следующий этап. Для ее решения вводятся временные ограничения на каждый из этапов жизненного цикла, и переход осуществляется в соответствии с планом, даже если не вся запланированная работа закончена. Планирование производится на основе статистических данных, полученных в предыдущих проектах, и личного опыта разработчиков.

Несмотря на настойчивые рекомендации компаний - вендоров и экспертов в области проектирования и разработки ИС, многие компании продолжают использовать каскадную модель вместо какого-либо варианта итерационной модели. Основные причины, по которым каскадная модель сохраняет свою популярность, следующие [3]:

1. **Привычка** - многие ИТ-специалисты получали образование в то время, когда изучалась только каскадная модель, поэтому она используется ими и в наши дни.
2. **Иллюзия снижения рисков** участников проекта (заказчика и исполнителя). Каскадная модель предполагает разработку законченных продуктов на каждом

этапе: технического задания, технического проекта, программного продукта и пользовательской документации. Разработанная документация позволяет не только определить требования к продукту следующего этапа, но и определить обязанности сторон, объем работ и сроки, при этом окончательная оценка сроков и стоимости проекта производится на начальных этапах, после завершения обследования. Очевидно, что если требования к информационной системе меняются в ходе реализации проекта, а качество документов оказывается невысоким (требования неполны и/или противоречивы), то в действительности использование каскадной модели создает лишь иллюзию определенности и на деле увеличивает риски, уменьшая лишь ответственность участников проекта. При формальном подходе менеджер проекта реализует только те требования, которые содержатся в спецификации, опирается на документ, а не на реальные потребности бизнеса. Есть два основных типа контрактов на разработку ПО. Первый тип предполагает выполнение определенного объема работ за определенную сумму в определенные сроки *{fixedprice).*Второй тип предполагает повременную оплату работы (*timework).*Выбор того или иного типа контракта зависит от степени определенности задачи. Каскадная модель с определенными этапами и их результатами лучше приспособлена для заключения контракта с оплатой по результатам работы, а именно этот тип контрактов позволяет получить полную оценку стоимости проекта до его завершения. Более вероятно заключение контракта с повременной оплатой на небольшую систему, с относительно небольшим весом в структуре затрат предприятия. Разработка и внедрение интегрированной информационной системы требует существенных финансовых затрат, поэтому используются контракты с фиксированной ценой, и, следовательно, каскадная модель разработки и внедрения. Спиральная модель чаще применяется при разработке информационной системы силами собственного отдела ИТ предприятия.

3. **Проблемы внедрения** при использовании итерационной модели. В некоторых областях спиральная модель не может применяться, поскольку невозможно использование/тестирование продукта, обладающего неполной функциональностью (например, военные разработки, атомная энергетика и т.д.). Поэтапное итерационное внедрение информационной системы для бизнеса возможно, но сопряжено с организационными сложностями (перенос данных, интеграция систем, изменение бизнес-процессов, учетной политики, обучение пользователей). Трудозатраты при поэтапном итерационном внедрении оказываются значительно выше, а управление проектом требует настоящего искусства. Предвидя указанные сложности, заказчики выбирают каскадную модель, чтобы "внедрять систему один раз".

ПОНЯТИЕ КАЧЕСТВА ПРОГРАММНОГО СРЕДСТВА

Каждое ПС должно выполнять определенные функции, т.е. делать то, что задумано. Хорошее ПС должно обладать еще целым рядом свойств, позволяющим успешно его использовать в течении длительного периода, т.е. обладать определенным качеством. Качество ПС — это совокупность его черт и характеристик, которые влияют на его способность удовлетворять заданные потребности пользователей [6]. Это не означает, что разные ПС должны обладать одной и той же совокупностью таких свойств в их высшей возможной степени. Этому препятствует тот факт, что повышение качества ПС по одному из таких свойств часто может быть достигнуто лишь ценой изменения стоимости, сроков завершения разработки и снижения качества этого ПС по другим его свойствам. Качество ПС является удовлетворительным, когда оно обладает указанными свойствами в такой степени, чтобы гарантировать успешное его использование.

Совокупность свойств ПС, которая образует удовлетворительное для пользователя качество ПС, зависит от условий и характера эксплуатации этого ПС, т.е. от позиции, с которой должно рассматриваться качество этого ПС. Поэтому при описании качества ПС

должны быть, прежде всего, фиксированы критерии отбора требуемых свойств ПС. В настоящее время критериями качества ПС принято считать [6,7, 8, 9, 10]:

- функциональность,
- надёжность,
- лёгкость применения,
- эффективность,
- сопровождаемость,
- мобильность.

Функциональность — это способность ПС выполнять набор функций, удовлетворяющих заданным или подразумеваемым потребностям пользователей. Набор указанных функций определяется во внешнем описании ПС.

Надежность подробно обсуждалась в первой лекции.

Лёгкость применения — это характеристики ПС, которые позволяют минимизировать усилия пользователя по подготовке исходных данных, применению ПС и оценке полученных результатов, а также вызывать положительные эмоции определённого или подразумеваемого пользователя.

Эффективность — это отношение уровня услуг, предоставляемых ПС пользователю при заданных условиях, к объему используемых ресурсов.

Сопровождаемость — это характеристики ПС, которые позволяют минимизировать усилия по внесению изменений для устранения в нём ошибок и по его модификации в соответствии с изменяющимися потребностями пользователей.

Мобильность — это способность ПС быть перенесенным из одной среды (окружения) в другую, в частности, с одной ЭВМ на другую.

Функциональность и надёжность являются обязательными критериями качества ПС, причём обеспечение надёжности будет красной нитью проходить по всем этапам и процессам разработки

ПС. Остальные критерии используются в зависимости от потребностей пользователей в соответствии с требованиями к ПС — их обеспечение будет обсуждаться в подходящих разделах курса.

ВИДЫ СОВРЕМЕННЫХ ЯЗЫКОВ ПРОГРАММИРОВАНИЯ

В большинстве случаев в качестве платформы используется персональный компьютер IBMPCкакой-либо системой UNIX, либо SPARCstation20 с системой Solaris2 (тоже UNIXsvr4), но многие примеры без каких-либо изменений (либо с минимумом таковых) могут быть перенесены в среду MSDOS[**], либо на другой тип машины с системой UNIX.

Но появился язык C++, который развивается довольно динамично. Еще есть Objective-C. Во-вторых потому, что есть библиотеки и системные вызовы, которые развиваются вслед за развитием UNIXи других операционных систем. Следующими вашими (настольными) книгами должны стать "Справочное руководство": тап2 (по системным вызовам), тап3 (по библиотечным функциям).

Мощь языка Си - в существующем многообразии библиотек.

Прошу вас с первых же шагов следить за стилем оформления своих программ. Делайте отступы, пишите комментарии, используйте осмысленные имена переменных и функций,отделяйте логические части программы друг от друга пустыми строками. Помните, что "лишние" пробелы и пустые строки в Си допустимы везде, кроме изображений констант и имен. Программы на Си, набитые в одну колонку (как на FORTRAN-е) очень тяжело читать и понимать. Из-за этого бывает трудно находить потерянные скобки { и }, потерянные символы 'и другие ошибки.

Существует несколько "школ" оформления программ - приглядитесь к примерам в этой книге и в других источниках - и выберите любую! Ничего страшного, если вы будете смешивать эти стили. Но - ПОДАЛЬШЕ ОТ FORTRAN-а!!!

Программу можно автоматически сформатировать к "каноническому" виду при

помощи,
например, программы cb.

cb< НашФайл.c >/tmp/$$ mv
/tmp/$$ НашФайл.c

но лучше сразу оформлять программу правильно.

Выделяйте логически самостоятельные ("замкнутые") части программы в функци(даже если они будут вызываться единственный раз). Функции - не просто средство избежать повторения одних и тех же операторов в тексте программы, но и средство структурирования процесса программирования, делающее программу более понятной.

Во-первых, вы можете в другой программе использовать текст уже написанной вами ранее функции вместо того, чтобы писать ее заново. Во-вторых, операцию, оформленную в виде функции, можно рассматривать как неделимый примитив (от довольно простого по смыслу,вроде strcmp, strcpy, до довольно сложного - qsort, malloc, gets) и забыть о его внутреннем устройстве (это хорошо - надо меньше помнить).

Не гонитесь за краткостью в ущерб ясности. Си позволяет порой писать такие выражения, над которыми можно полчаса ломать голову. Если же их записать менее мудрено,но чуть длиннее - они самоочевидны (и этим более защищены от ошибок).

2.ОСНОВНЫЕ ПРИНЦИПЫ И ПОДХОДЫ К РАЗРАБОТКЕ ПРОГРАММНЫХ АЛГОРИТМОВ.

ПОНЯТИЕ АЛГОРТМА И ЕГО СВОЙСТВА

Алгоритм - упорядоченная совокупность системы правил, определяющая содержание и порядок действий над некоторыми объектами, строгое выполнение которых приводит к решению любой задачи из рассматриваемого класса задач за конечное число шагов.

Слово «*алгоритм*» появилось в IX веке, когда европейцы познакомились с работами выдающегося узбекского математика Муххамеда бен аль-Хорезми («algorithmi» - латинская форма записи имени аль-Хорезми). В своей книге «Об индийском счете» он сформулировал правила выполнения арифметических действий над десятичными числами. В дальнейшем понятие «алгоритм» стали использовать для обозначения .любой последовательности действий, приводящей к решению поставленной задачи.

При всем разнообразии алгоритмов можно выделить общие для них свойства*:* дискретность, массовость, определенность и результативность.

Дискретность (разрывность) - это свойство алгоритма, характеризующее его структуру: каждый алгоритм состоит из отдельных законченных действий [4, с. 292].

Массовость - применимость алгоритма ко всем задачам рассматриваемого типа, при любых исходных данных [4, с. 292].

Определенность - свойство алгоритма, указывающее на то, что каждый шаг алгоритма должен быть строго определен и не допускать различных толкований; также строго должен быть определен порядок выполнения отдельных шагов [4, с. 292].

Результативность конечность действий алгоритма решения задач, позволяющая получить желаемый результат при допустимых исходных данных за конечное число шагов.

КЛАССЫ АЛГОРИТМОВ И СПОСОБЫ ИХ ОПИСАНИЯ ***Способы описания алгоритмов***

Разработанный алгоритм можно записать несколькими способами:

- на естественном языке (словесное описание);
- с помощью псевдокода;
- в виде блок-схемы;
- в виде программы.

Словесное описание представляет структуру алгоритма на естественном языке. Примерами такой записи алгоритма являются инструкции по эксплуатации приборов бытовой техники, кулинарные рецепты, правила дорожного движения и т.д.

Словесная форма имеет ряд недостатков:

- строго не формализуема;
- страдает многословностью записей;
- допускает неоднозначность толкования отдельных предписаний.

Эта форма обычно используется на начальных стадиях разработки алгоритма.

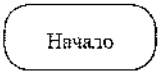
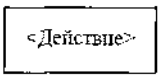
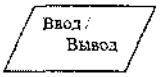
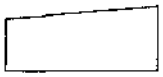
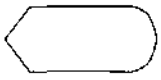
Псевдокод - пошагово-словесная запись алгоритма по определенным правилам или соглашениям. В псевдокоде используется общепринятая математическая символика и конструкции. Он занимает промежуточное место между естественным и формальным языками.

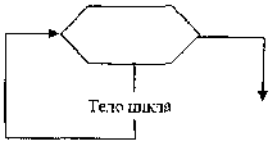
Блок-схема - это наглядное графическое представление алгоритма с помощью геометрических фигур, соединенных линиями-связями, показывающими порядок выполнения инструкций.

Рассмотрим основные фигуры, которые используются для построения блок-схем (табл. 2).

Таблица 2

Графические объекты блок-схем

Вид графического объекта	Название блока. Комментарии
	Начало алгоритма
	Конец алгоритма
	Процесс. Внутри блока записывается действие, вычислительная операция или группа
	Ввод/вывод данных с неопределенного носителя. Надпись внутри блока: ввод (вывод) и список вводимых (выводимых) переменных
	Ручной ввод (ввод с клавиатуры)
	Дисплей (вывод на монитор)
	Документ (вывод на печатающее устройство)
	Решение (условный блок). Условие записывается внутри блока. В результате проверки условия осуществляется выбор одного из возможных путей вычислительного процесса

1	2
	Модификатор. Используется для описания цикла с параметром
	Границы цикла. Описывает циклические процессы: «цикл с предусловием» и «цикл с постусловием»
	Внутристраничный соединитель
	Межстраничный соединитель
	Предопределенный процесс (подпрограмма)

Программа - описание структуры алгоритма на языке программирования.

ОСНОВНЫЕ АЛГОРИТМИЧЕСКИЕ КОНСТРУКЦИИ

Существует три типа алгоритмических конструкций: *линейная* (последовательная), разветвляю!цаяся и циклическая.

Линейная алгоритмическая конструкция

Линейный алгоритм - это такой алгоритм, в котором действия выполняются однократно в заданном порядке (рис. 1).

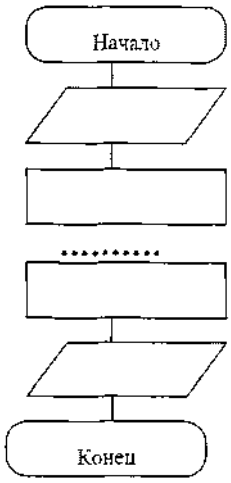


Рис. 1. Размещение блоков в линейном алгоритме

Разветвляющаяся алгоритмическая конструкция

Разветвляющийся алгоритм - такой алгоритм, в котором выполняется либо одна, либо другая последовательность действий, в зависимости от условия.

Различают две формы разветвляющейся алгоритмической структуры: *полное ветвление* (ЕСЛИ - ТО - ИНАЧЕ) и *неполное ветвление* (ЕСЛИ - ТО).

Полное ветвление (рис. 3) позволяет организовать в алгоритме две ветви (ТО или ИНАЧЕ). Неполное ветвление (рис. 4) предполагает наличие действий только на одной ветви (ТО), вторая ветвь отсутствует.

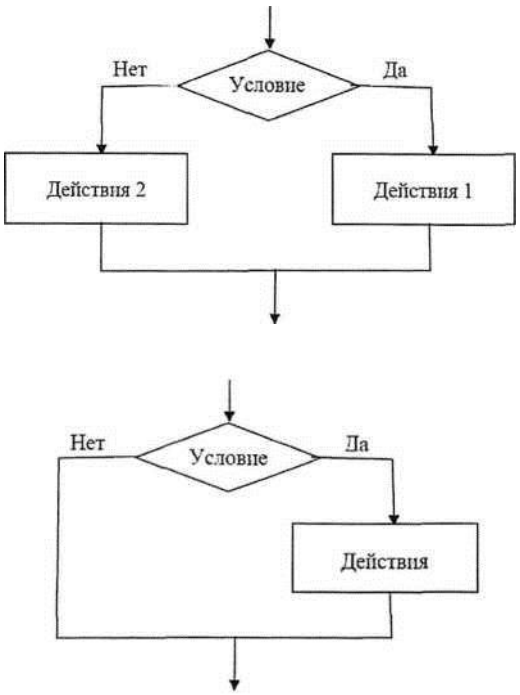


Рис. 4 Неполное ветвление

Алгоритмическая структура «выбор»

Для выбора из нескольких альтернативных действий используется алгоритмическая структура «выбор». Перед выполнением команды «выбор» вычисляется значение некоторого выражения X, а затем начинается проверка условий Yi(X), Y2(X).--Y_n(X). Проверка продолжается до тех пор, пока не встретится условие, принимающее значение ИСТИНА при данном X (рис. 7).

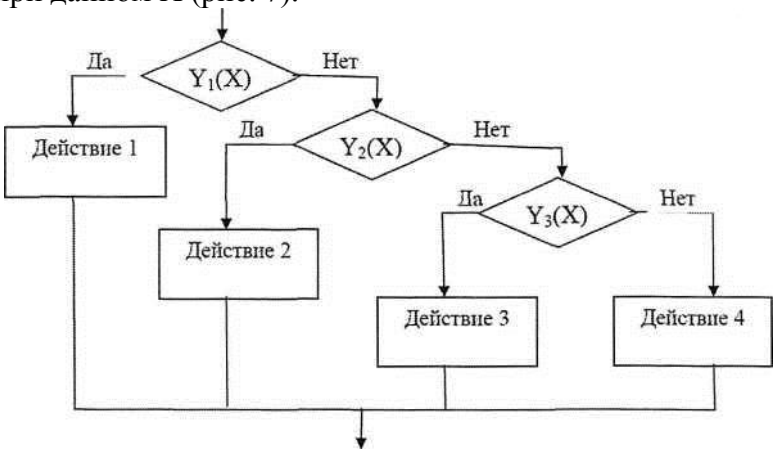


Рис. 7. Алгоритмическая структура «выбор»

Структуру «выбор» можно также изобразить в форме, представленной на рис. 8.

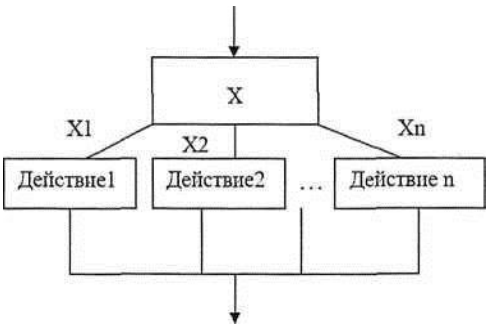


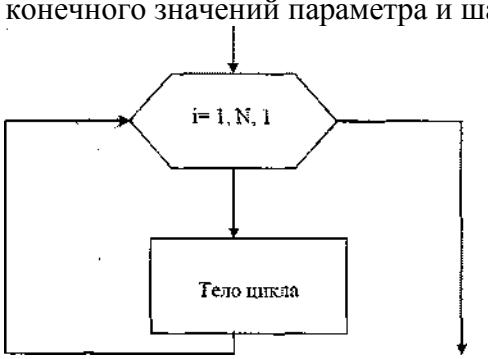
Рис. 8. Вариант оформления
алгоритмической структуры «выбор»

Циклическая алгоритмическая конструкция

Циклической называют алгоритмическую конструкцию, в которой действие выполняется указанное число раз, или - пока не выполнится условие. Группа повторяющихся действий цикла называется *телом цикла.* Существует три типа циклов: *цикл с параметром (арифметический), цикл с предусловием и цикл с постусловием.*

Цикл с параметром

В цикле с параметром число повторений цикла однозначно определено и задается с помощью начального, конечного значений параметра и шагом его изменения.



Цикл с предусловием

Действия внутри этого цикла повторяются, пока выполняется условие блоке ветвления, причем сначала проверяется условие, а затем выполняете действие.

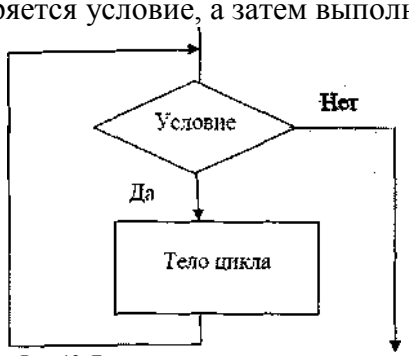


Рис. 13. Блок-схема цикла с предусловием,
представленная спомощью условного блока

Рис. 14. Границы
никла

Цикл с постусловием

Тело цикла с постусловием всегда будет выполнено хотя бы один раз. Оно будет выполняться до тех пор, пока значение условного выражения ЛОЖНО. Как только условное выражение принимает значение ИСТИНА, цикл завершается.

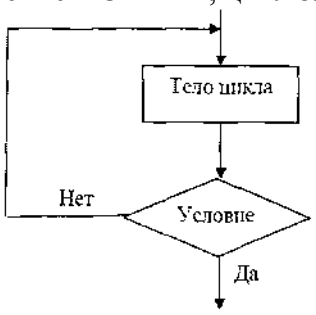


Рис. 16. Блок-схема цикла с постусловием, представленная с помощью условного блока

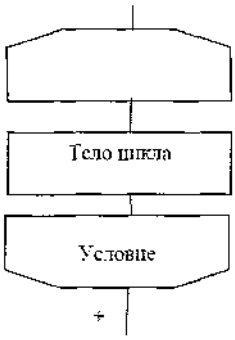


Рис. 17. Второй способ представления цикла с постусловием

3.ОСНОВЫ ПРОГРАММИРОВАНИЯ НА ЯЗЫКЕ ВЫСОКОГО УРОВНЯ СИ++ АРХИТЕКТУРА ПРОГРАММНОГО СРЕДСТВА

Лекция проводится в интерактивной форме с разбором конкретных примеров.

Понятие архитектуры программного средства

Архитектура ПС — это его строение как оно видно (или должно быть видно) извне его, т.е. представление ПС как системы, состоящей из некоторой совокупности взаимодействующих подсистем. В качестве таких подсистем выступают обычно отдельные программы. Разработка архитектуры является первым этапом борьбы со сложностью ПС, на котором реализуется принцип выделения относительно независимых компонент.

Основные задачи разработки архитектуры ПС:

- выделение программных подсистем и отображение на них внешних функций (заданных во внешнем описании) ПС;

- определение способов взаимодействия между выделенными программными подсистемами.

Основные классы архитектур программных средств

Различают следующие основные классы архитектур программных средств [1]:

- цельная программа;
- комплекс автономно выполняемых программ;
- слоистая программная система;
- коллектив параллельно выполняемых программ.

Цельная программа представляет вырожденный случай архитектуры ПС: в состав ПС входит только одна программа. Такую архитектуру выбирают обычно в том случае, когда ПС должно выполнять одну какую-либо ярко выраженную функцию и её реализация не представляется слишком сложной. Естественно, что такая архитектура не требует какого-либо описания (кроме фиксации класса архитектуры), так как отображение внешних функций на эту программу тривиально, а определять способ взаимодействия не требуется (в силу отсутствия какого-либо внешнего взаимодействия программы, кроме как взаимодействия её с пользователем, а последнее описывается в документации по применению ПС).

Комплекс автономно выполняемых программ состоит из набора программ, такого, что:

- любая из этих программ может быть активизирована (запущена) пользователем;
- при выполнении активизированной программы другие программы этого набора не могут быть активизированы до тех пор, пока не закончит выполнение активизированная программа;
- все программы этого набора применятся к одной и той же информационной среде.

Таким образом, программы этого набора по управлению никак не взаимодействуют — взаимодействие между ними осуществляется только через общую информационную среду.

Слоистая программная система состоит из некоторой упорядоченной совокупности программных подсистем, называемых слоями, такой, что:

- на каждом слое ничего не известно о свойствах (и даже существовании) последующих (более высоких) слоёв;
- каждый слой может взаимодействовать по управлению (обращаться к компонентам) с непосредственно предшествующим (более низким) слоем через заранее определенный интерфейс, ничего не зная о внутреннем строении всех предшествующих слоёв;
- каждый слой располагает определенными ресурсами, которые он либо скрывает от других слоев, либо предоставляет непосредственно последующему слою (через указанный интерфейс) некоторые их абстракции.

Таким образом, в слоистой программной системе каждый слой может реализовать некоторую абстракцию данных. Связи между слоями ограничены передачей значений параметров обращения каждого слоя к смежному снизу слою и выдачей результатов этого обращения от нижнего слоя верхнему. Недопустимо использование глобальных данных несколькими слоями.

В качестве примера рассмотрим использование такой архитектуры для построения операционной системы. Такую архитектуру применил Дейкстра при построении операционной системы THE[2]. Эта операционная система состоит из четырех слоев (см. Рис. 1). На нулевом слое производится обработка всех прерываний и выделение центрального процессора программам (процессам) в пакетном режиме. Только этот уровень осведомлен о мультипрограммных аспектах системы. На первом слое осуществляется управление страничной организацией памяти. Всем вышестоящим слоям предоставляется виртуальная непрерывная (не страничная) память. На втором слое осуществляется связь с консолью (пультом управления) оператора. Только этот слой

знает технические характеристики консоли. На третьем слое осуществляется буферизация входных и выходных потоков данных и реализуются так называемые абстрактные каналы ввода и вывода, так что прикладные программы не знают технических характеристик устройств ввода-вывода.

Коллектив параллельно действующих программ представляет собой набор программ, способных взаимодействовать между собой, находясь одновременно в стадии выполнения. Это означает, что такие программы, во-первых, вызваны в оперативную память, активизированы и могут попеременно разделять по времени один или несколько центральных процессоров, а во- вторых, осуществлять между собой динамические (в процессе выполнения) взаимодействия, на базе которых производиться их синхронизация. Обычно взаимодействие между такими процессами производится путём передачи друг другу некоторых сообщений.

Простейшей разновидностью такой архитектуры является конвейер, средства, для организации которого имеются в операционной системе UNIX[3]. Конвейер представляет собой последовательность программ, в которой стандартный вывод каждой программы, кроме самой последней, связан со стандартным вводом следующей программы этой последовательности (см. Рис. 2). Конвейер обрабатывает некоторый поток сообщений. Каждое сообщение этого потока поступает на ввод первой программе, которая, обработав его, передаёт переработанное сообщение следующей программе, а сама начинает обработку очередного сообщения потока. Таким же образом действует каждая программа конвейера: получив сообщение от предшествующей программы и обработав его, она передаёт переработанное сообщение следующей программе, а последняя программа конвейера выводит результат работы всего конвейера (результирующее сообщение). Таким образом, в конвейере, состоящим из п программ, может одновременно находиться в обработке до п сообщений. Конечно, в силу того, что разные программы конвейера могут затратить на обработку очередных сообщений разные отрезки времени, необходимо обеспечить каким-либо образом синхронизацию этих процессов (некоторые процессы могут находиться в стадии ожидания либо возможности передать переработанное сообщение, либо возможности получить очередное сообщение).

Порт сообщений представляет собой программную подсистему, обслуживающую некоторую очередь сообщений: она может принимать на хранение от программы какое-либо сообщение, ставя его в очередь, и может выдавать очередное сообщение другой программе по её требованию. Сообщение, переданное какой-либо программой некоторому порту, уже не будет принадлежать этой программе (и использовать её ресурсы), но оно не будет принадлежать и никакой другой программе, пока в порядке очереди не будет передано какой-либо программе по её запросу. Таким образом, программа, передающая сообщение не будет находиться в стадии ожидания пока программа, принимающая это сообщение, не будет готова его обрабатывать (если только не будет переполнен принимающий порт).

Пример программной системы с портами сообщений приведен на Рис. 3. Порт Uможет рассматриваться как порт вводных сообщений для представленного на этом рисунке коллектива параллельно действующих программ, а порт W— как порт выводных сообщений для этого коллектива программ.

Программные системы с портами сообщений могут быть как жесткой конфигурации, так и гибкой конфигурации. В системах с портами жесткой конфигурации с каждой программой могут быть жестко связаны один или несколько входных портов. Для передачи сообщения такая программа должна явно указать адрес передачи: имя программы и имя ее входного порта. В этом случае при изменении конфигурации системы придется корректировать используемые программы: изменять адреса передач сообщений. В системах с портами гибкой конфигурации с каждой программой связаны как входные, так и выходные виртуальные порты. Перед запуском такой

системы должна производиться ее предварительная настройка с помощью специальной программной компоненты, осуществляющая совмещение каждого выходного виртуального порта с каким-либо входным виртуальным портом на основании информации, задаваемой пользователем. Тем самым при изменении конфигурации системы в этом случае не требуется какой-либо корректировки используемых программ — необходимые изменения должны быть отражены в информации для настройки. Однако в этом случае требуется иметь специальную программную компоненту, осуществляющую настройку системы.

ПРОЕКТИРОВАНИЕ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ ПРИ СТРУКТУРНОМ ПОДХОДЕ

Процесс проектирования сложного ПО начинают с уточнения его структуры, т.е. определения структурных компонентов и связей между ними. Результат уточнения структуры может быть представлен в виде структурной и/или функциональной схем.

Структурная схема разрабатываемого ПО. *Структурной* называют схему, отражающую *состав и взаимодействие по управлению* частей разрабатываемого ПО.

Структурные схемы пакетов программ не информативны, поскольку организация программ в пакеты не предусматривает передачи управления между программами. Поэтому структурные схемы разрабатывают для каждой программы пакета, а список программ пакета определяют, анализируя функции, указанные в техническом задании.

Самый простой вид ПО - программа в качестве структурных компонентов может включать только подпрограммы и библиотеки ресурсов. Разработка структурной схемы программы обычно выполняется методом пошаговой детализации (см. § 4.2).

Структурными компонентами программной системы или программного комплекса могут служить программы, подсистемы, базы данных, библиотеки ресурсов и т. п.

Структурная схема программного комплекса демонстрирует передачу управления от программы-диспетчера соответствующей программе (рис. 4. 1).

Рис. 4.1. Пример структурной схемы программного комплекса

Структурная схема программной системы, как правило, показывает наличие подсистем или других структурных компонентов. В отличие от программного комплекса отдельные части (подсистемы) программной системы интенсивно обмениваются данными между собой и, возможно, с основной программой. Структурная же схема программной системы этого обычно не показывает (рис. 4.2).

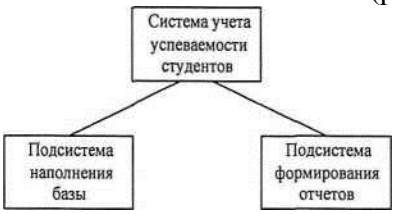


Рис. 4.2. Пример структурной схемы программной системы

Функциональная схема. *Функциональная схема* или *схема данных* (ГОСТ 19.701-90) - схема взаимодействия компонентов ПО с описанием информационных потоков, состава данных в потоках и указанием используемых файлов и устройств. Для изображения функциональных схем используют специальные обозначения, установленные стандартом.

Функциональные схемы более информативны, чем структурные. Так функциональные схемы программных комплексов и систем наглядно демонстрируют различие между ними (рис. 4.3). Все компоненты структурных и функциональных схем должны быть описаны. При структурном подходе особенно тщательно необходимо прорабатывать спецификации межпрограммных интерфейсов, так как от качества их описания зависит количество самых дорогостоящих ошибок. К самым дорогим при структурном подходе относятся ошибки, обнаруживаемые при комплексном тестировании, так как для их устранения могут потребоваться серьезные изменения уже отлаженных текстов.

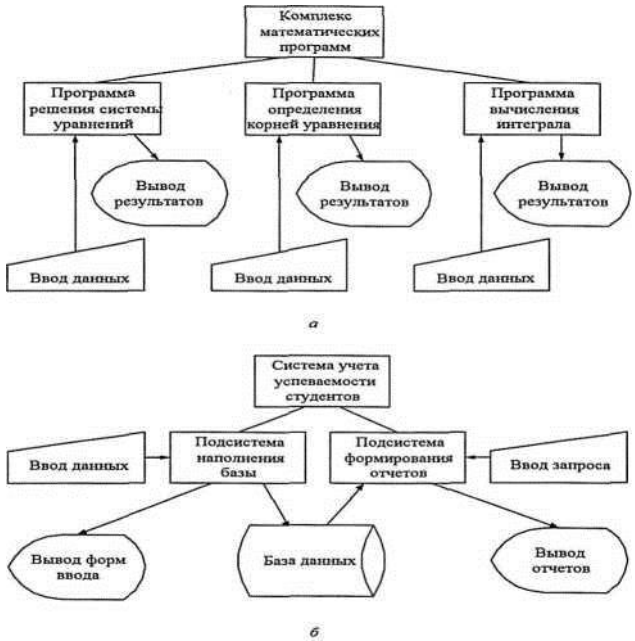


Рис. 4.3. Примеры функциональных схем: *а* - комплекс программ; *б* - программная система

Использование метода пошаговой детализации для проектирования структуры программного обеспечения

Структурный подход предлагает осуществлять декомпозицию программ методом пошаговой детализации. Результат декомпозиции - структурная схема п р о г р а м м ы - представляет собой многоуровневую иерархическую схему взаимодействия подпрограмм по управлению. Минимально такая схема отображает два уровня иерархии, т.е. показывает общую структуру программы. Однако тот же метод позволяет получить структурные схемы с большим количеством уровней.

Метод пошаговой детализации реализует нисходящий подход и базируется на основных конструкциях структурного программирования. Он предполагает пошаговую разработку алгоритма. Каждый шаг при этом включает разложение функции на подфункции. Так на первом этапе описывают решение поставленной задачи, выделяя общие подзадачи. На следующем аналогично описывают решение подзадач, формулируя уже подзадачи следующего уровня. Таким образом, на каждом шаге происходит уточнение функций проектируемого ПО. Процесс продолжают, пока не доходят до подзадач, алгоритмы решения которых очевидны.

Декомпозируя программу методом пошаговой детализации следует придерживаться основного п р а в и л а структурной декомпозиции, следующего из

принципа вертикального управления: в первую очередь детализировать *управляющие процессы* декомпозируемого компонента, оставляя уточнение операций с данными напоследок.

Кроме этого целесообразно придерживаться следующих рекомендаций:

* не отделять операции инициализации и завершения от соответствующей обработки, так как модули инициализации и завершения имеют плохую связность (временную) и сильное сцепление (по управлению);
* не проектировать слишком специализированных или слишком универсальных модулей, так как проектирование излишне специальных модулей увеличивает их количество, а проектирование излишне универсальных модулей - увеличивает их сложность;
* избегать дублирования действий в различных модулях, так как при их изменении исправления придется вносить во все места, где они выполняются - в этом случае целесообразно просто реализовать эти действия в отдельном модуле;
* группировать сообщения об ошибках в один модуль по типу библиотеки ресурсов, тогда будет легче согласовать формулировки, избежать дублирования сообщений, а также перевести сообщения на другой язык.

При этом, описывая решение каждой задачи, желательно использовать не более одной-двух структурных управляющих конструкций, таких как цикл-пока или ветвление, что позволяет четче представить себе структуру организуемого вычислительного процесса.

Структурные карты Константайна

На структурной карте отношения между модулями представляют в виде графа, вершинам которого соответствуют модули и общие области данных, а дугам - межмодульные вызовы и обращения к общим областям данных.

Различают четыре типа вершин:

* модуль - подпрограмма;
* подсистема - программа;
* библиотека - совокупность подпрограмм, размещенных в отдельном модуле;
* область данных - специальным образом оформленная совокупность данных, к которой возможно обращение извне.

Обозначения соответствующих вершин приведены на рис. 4.8. (Обозначения даны в соответствии со стандартами IBM, ISOи ANSI.)

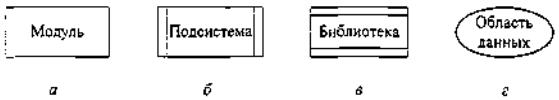


Рис. 4.8. Обозначения вершин: *а* - модуль; *б* - подсистема; *в* - библиотека; г - область данных

Различают также типы вызова: последовательный (рис. 4.9, *а),* параллельный (рис, 4.9, б) и вызов сопрограммы (рис. 4.9, *в)*

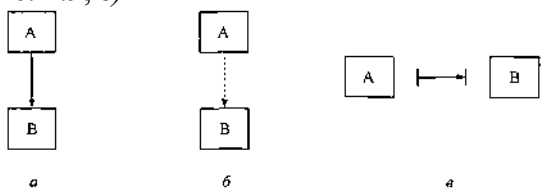


Рис. 4.9. Обозначения типа вызова: а-последовательный вызов; *б* - параллельный вызов; в - вызов сопрограммы

Под вызовом сопрограммы понимают возможность поочередного выполнения фрагментов двух одновременно запущенных программ, например, если одна программа подготовила порцию данных для вывода, то вторая может ее вывести, а затем перейти в состояние ожидания следующей порции (рис. 4.10). Причем в мультипрограммных

системах основная программа, передав данные, продолжает работать, а не переходит в состояние ожидания, как изображено на рисунке.

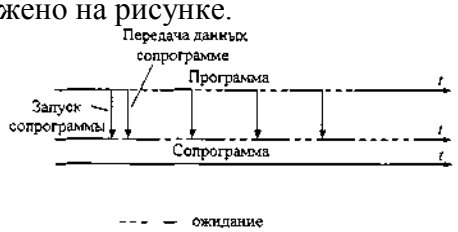
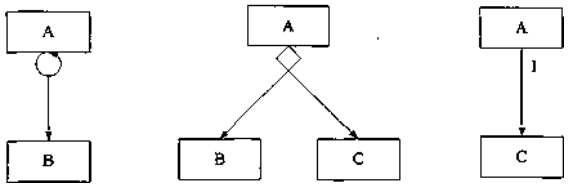


Рис. 4.10. Диаграмма вызова сопрограммы

Если стрелка, изображающая вызов касается блока, то обращение происходит к модулю целиком, а если входит в блок, то - к элементу внутри модуля.

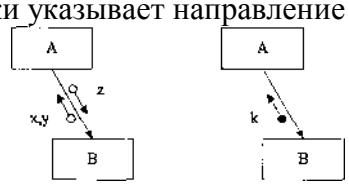
При необходимости на структурной карте можно уточнить особые условия вызова: циклический вызов (рис. 4.11, *а),* условный вызов (рис. 4.11, *б)* и однократный вызов - при повторном вызове основного модуля однократно вызываемый модуль не активизируется (рис. 4.11, в).



а б в

Рис. 4.11. Обозначения особых условий вызова: *а* - циклический вызов; *б* - условный вызов; *в* - однократный вызов

Связи по данным и управлению обозначают стрелками, параллельными дуге вызова, причем Направление стрелки указывает направление связи (рис. 4.12).



а б

Рис. 4.12. Обозначение типа связи: *а* - связи по данным; *б* - связь по управлению

Структурные карты Константайна позволяют наглядно представить результат декомпозиции программы на модули и оценить ее качество, т.е. соответствие рекомендациям структурного программирования.

ЭТАПЫ РАБОТЫ С ПРОГРАММОЙ НА ЯЗЫКЕ СИ/СИ++ В СИСТЕМЕ ПРОГРАММИРОВАНИЯ

Все программы на языке Си и Си++ строятся по модульному принципу, т.е. состоят из множества модулей. Под модулем здесь имеется в виду библиотеки или ранее написанные части кода программы. Согласно принципам скрытия информации текст модуля обычно разделяют на заголовочный файл с расширением **.h**или **.hpp**и файл реализации с одним из расширений **.c, .cpp,** ■obj, .lib, .asm.

После того, как исходный текст программы сформирован, на основе его должен быть выполнен или создан исполняемый файл с расширением **.exe.** Этот процесс осуществляется в несколько этапов, структура такого процесса представлена на рис.1.

Вначале с помощью текстового редактора формируется и сохраняется в файлах с расширением **.cpp** исходный текст программы. Затем осуществляется этап препроцессорной обработки, содержание которого определяется директивами препроцессора, расположенными в начале исходного кода. В частности, по директиве ^includeпрепроцессор подключает к тексту программы заголовочные файлы стандартных

библиотек. Результатом работы препроцессорной обработки является формирование полного текстового кода программы, который в явном виде не доступен программисту и достаточно объемен из-за подключаемых заголовочных файлов. Далее происходит компиляция программы, в ходе которой могут быть обнаружены синтаксические ошибки, которые должны быть устранены. В случае успешной компиляции получается объектный код программы в файле с расширением .obj. После компиляции выполняется этап компоновки и редактирования связей, осуществляемый с помощью встроенных системных программ языка. В результате компоновки создается исполняемый двоичный код программы в файле с расширением **.exe,** который запускается на выполнение и который является результатом написания программы.

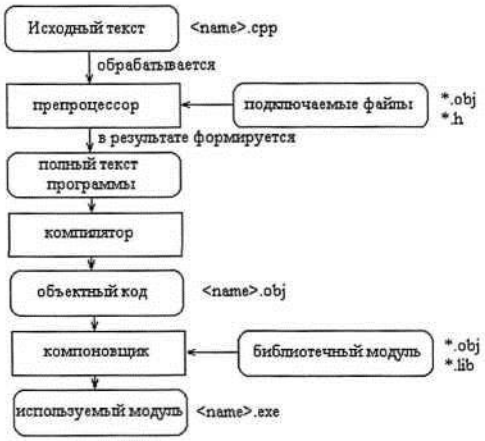


Рис.1 Структура процесса компиляции

Особенности выполнения перечисленных действий зависят от конкретного компилятора языка Си/Си++ и той операционной системы, в которой он работает. Технические подробности можно изучить по документации для конкретного программного продукта. Например, при работе с интегрированными средами фирмы Borlandнеобходимая информация может быть получена из руководств пользователя.

В простейшем случае программа на языке Си и Си++ представляет собой набор описаний и определений, и состоит из набора функций. Среди этих функций всегда должна быть функция с именем main(для консольных программ) или WinMain(для программ в 32 - разрядных операционных систем). Данные функции являются точками входа и выхода, и без них программа не может быть выполнена. Если функция не возвращает никакого значения в результате своего

42

выполнения, то перед именем функции помещается служебное слово void, обозначающее тип отсутствия значения. Также каждая функция должна иметь набор параметров, если параметры отсутствуют, то скобки оставляют пустыми или в них указывается void.

За заголовком размещается тело функции. Тело функции - это последовательность определений, описаний и исполняемых операторов, заключенных в фигурные скобки. Ниже приведен листинг простейшей программы на языке Си/Си++, вычисляющая объем цилиндра по радиусу и высоте, значения которых вводятся с клавиатуры.

```
        // подключение средств консольного потокового
ввода/вывода //include<iostream.h>
        //подключение математических функций
//include<math.h>
        /* главная функция программы *1
voidmain()
         {
        // объявление переменных intr,v,h;
```

```
//вывод на экран
cout«’ЛпВведите радиус и высоту цилиндра:";
    //вывод на экран cin» r » h;
    //ввод значений с клавиатуры
v=M_PI*r*r*h;
cout<<"\побъем цилиндра = "« v;
     }
```

РАЗРАБОТКА СТРУКТУРЫ ПРОГРАММЫ И МОДУЛЬНОЕ ПРОГРАММИРОВАНИЕ

Цель модульного программирования

Приступая к разработке каждой программы ПС, следует иметь ввиду, что она, как правило, является большой системой, поэтому мы должны принять меры для её упрощения. Для этого такую программу разрабатывают по частям, которые называются программными модулями [1, 2]. А сам такой метод разработки программ называют модульным программированием [3]. Программный модуль — это любой фрагмент описания процесса, оформляемый как самостоятельный программный продукт, пригодный для использования в описаниях процесса. Это означает, что каждый программный модуль программируется, компилируется и отлаживается отдельно от других модулей программы, и тем самым, физически разделён от других модулей программы. Более того, каждый разработанный программный модуль может включаться в состав разных программ, если выполнены условия его использования, декларированные в документации по этому модулю. Таким образом, программный модуль может рассматриваться и как средство борьбы со сложностью программ, и как средство борьбы с дублированием в программировании (т.е. как средство накопления и многократного использования программистских знаний).

Модульное программирование является воплощением в процессе разработки программ обоих общих методов борьбы со сложностью (см. лекцию «Замок дракона. Лекция 3 — Методы борьбы со сложностью»), и обеспечения независимости компонент системы, и использования иерархических структур. Для воплощения первого метода формулируются определенные требования, которым должен удовлетворять программный модуль, т.е. выявляются основные характеристики «хорошего» программного модуля. Для воплощения второго метода используют древовидные модульные структуры программ (включая деревья со сросшимися ветвями).

Основные характеристики программного модуля

Не всякий программный модуль способствует упрощению программы [2]. Выделить хороший с этой точки зрения модуль является серьезной творческой задачей. Для оценки приемлемости выделенного модуля используются некоторые критерии. Так, Хольт [4] предложил следующие два общих таких критерия:

- хороший модуль снаружи проще, чем внутри;
- хороший модуль проще использовать, чем построить.

Майерс [5] предлагает использовать более конструктивные характеристики программного модуля для оценки его приемлемости: размер модуля; прочность модуля; сцепление с другими модулями; рутинность модуля (независимость от предыстории обращений к нему).

Размер модуля измеряется числом содержащихся в нем операторов (строк). Модуль не должен быть слишком маленьким или слишком большим. Маленькие модули приводят к громоздкой модульной структуре программы и могут не окупать накладных расходов, связанных с их оформлением. Большие модули неудобны для изучения и изменений, они могут существенно увеличить суммарное время повторных трансляций программы при отладке программы. Обычно рекомендуются программные модули размером от нескольких десятков до нескольких сотен операторов.

Прочность модуля — это мера его внутренних связей. Чем выше прочность

модуля, тем больше связей он может спрятать от внешней по отношению к нему части программы и, следовательно, тем больший вклад в упрощение программы он может внести. Для оценки степени прочности модуля Майерс [5] предлагает упорядоченный по степени прочности набор из семи классов модулей. Самой слабой степенью прочности обладает модуль, прочный по совпадению. Это такой модуль, между элементами которого нет осмысленных связей. Такой модуль может быть выделен, например, при обнаружении в разных местах программы повторения одной и той же последовательности операторов, которая и оформляется в отдельный модуль. Необходимость изменения этой последовательности в одном из контекстов может привести к изменению этого модуля, что может сделать его использование в других контекстах ошибочным. Такой класс программных модулей не рекомендуется для использования. Вообще говоря, предложенная Майерсом упорядоченность по степени прочности классов модулей не бесспорна. Однако, это не очень существенно, так как только два высших по прочности класса модулей рекомендуются для использования. Эти классы мы и рассмотрим подробнее.

Функционально прочный модуль — это модуль, выполняющий (реализующий) одну какую- либо определенную функцию. При реализации этой функции такой модуль может использовать и другие модули. Такой класс программных модулей рекомендуется для использования.

Информационно прочный модуль — это модуль, выполняющий (реализующий) несколько операций (функций) над одной и той же структурой данных (информационным объектом), которая считается неизвестной вне этого модуля. Для каждой из этих операций в таком модуле имеется свой вход со своей формой обращения к нему. Такой класс следует рассматривать как класс программных модулей с высшей степенью прочности. Информационно-прочный модуль может реализовывать, например, абстрактный тип данных.

В модульных языках программирования как минимум имеются средства для задания функционально прочных модулей (например, модуль типа FUNCTIONв языке ФОРТРАН). Средства же для задания информационно прочных модулей в ранних языках программирования отсутствовали — они появились только в более поздних языках. Так в языке программирования Ада средством задания информационно прочного модуля является пакет [6].

Сцепление модуля — это мера его зависимости по данным от других модулей. Характеризуется способом передачи данных. Чем слабее сцепление модуля с другими модулями, тем сильнее его независимость от других модулей. Для оценки степени сцепления Майерс предлагает [5] упорядоченный набор из шести видов сцепления модулей. Худшим видом сцепления модулей является сцепление по содержимому. Таким является сцепление двух модулей, когда один из них имеет прямые ссылки на содержимое другого модуля (например, на константу, содержащуюся в другом модуле). Такое сцепление модулей недопустимо. Не рекомендуется использовать также сцепление по общей области — это такое сцепление модулей, когда несколько модулей используют одну и ту же область памяти. Такой вид сцепления модулей реализуется, например, при программировании на языке ФОРТРАН с использованием блоков COMMON. Единственным видом сцепления модулей, который рекомендуется для использования современной технологией программирования, является параметрическое сцепление (сцепление по данным по Майерсу [5]) — это случай, когда данные передаются модулю либо при обращении к нему как значения его параметров, либо как результат его обращения к другому модулю для вычисления некоторой функции. Такой вид сцепления модулей реализуется на языках программирования при использовании обращений к процедурам (функциям).

Рутинность модуля — это его независимость от предыстории обращений к нему. Модуль будем называть рутинным, если результат (эффект) обращения к нему зависит

только от значений его параметров (и не зависит от предыстории обращений к нему). Модуль будем называть зависящим от предыстории, если результат (эффект) обращения к нему зависит от внутреннего состояния этого модуля, хранящего следы предыдущих обращений к нему. Майерс [5] не рекомендует использовать зависящие от предыстории (непредсказуемые) модули, так как они провоцируют появление в программах хитрых (неуловимых) ошибок. Однако такая рекомендация является неконструктивной, так как во многих случаях именно зависящий от предыстории модуль является лучшей реализаций информационно прочного модуля. Поэтому более приемлема следующая (более осторожная) рекомендация:

- всегда следует использовать рутинный модуль, если это не приводит к плохим (не рекомендуемым) сцеплениям модулей;
- зависящие от предыстории модули следует использовать только в случае, когда это необходимо для обеспечения параметрического сцепления;
- в спецификации зависящего от предыстории модуля должна быть четко сформулирована эта зависимость таким образом, чтобы было возможно прогнозировать поведение (эффект выполнения) данного модуля при разных последующих обращениях к нему.

В связи с последней рекомендацией может быть полезным определение внешнего представления (ориентированного на информирование человека) состояний зависящего от предыстории модуля. В этом случае эффект выполнения каждой функции (операции), реализуемой этим модулем, следует описывать в терминах этого внешнего представления, что существенно упростит прогнозирование поведения данного модуля.

Методы разработки структуры программы

Как уже отмечалось выше, в качестве модульной структуры программы принято использовать древовидную структуру, включая деревья со сросшимися ветвями. В узлах такого дерева размещаются программные модули, а направленные дуги (стрелки) показывают статическую подчиненность модулей, т.е. каждая дуга показывает, что в тексте модуля, из которого она исходит, имеется ссылка на модуль, в который она входит. Другими словами, каждый модуль может обращаться к подчиненным ему модулям, т.е. выражается через эти модули. При этом модульная структура программы, в конечном счете, должна включать и совокупность спецификаций модулей, образующих эту программу. Спецификация программного модуля содержит, во-первых, синтаксическую спецификацию его входов, позволяющую построить на используемом языке программирования синтаксически правильное обращение к нему (к любому его входу), и, во-вторых, функциональную спецификацию модуля (описание семантики функций, выполняемых этим модулем по каждому из его входов). Функциональная спецификация модуля строится так же, как и функциональная спецификация ПС.

В процессе разработки программы её модульная структура может по-разному формироваться и использоваться для определения порядка программирования и отладки модулей, указанных в этой структуре. Поэтому можно говорить о разных методах разработки структуры программы. Обычно в литературе обсуждаются два метода [1, 7]: метод восходящей разработки и метод нисходящей разработки.

Метод восходящей разработки заключается в следующем. Сначала строится модульная структура программы в виде дерева. Затем поочередно программируются модули программы, начиная с модулей самого нижнего уровня (листья дерева модульной структуры программы), в таком порядке, чтобы для каждого программируемого модуля были уже запрограммированы все модули, к которым он может обращаться. После того, как все модули программы запрограммированы, производится их поочередное тестирование и отладка в принципе в таком же (восходящем) порядке, в каком велось их программирование. На первый взгляд такой порядок разработки программы кажется вполне естественным: каждый модуль при программировании выражается через уже запрограммированные непосредственно подчиненные модули, а при тестировании

использует уже отлаженные модули. Однако, современная технология не рекомендует такой порядок разработки программы. Во-первых, для программирования какого- либо модуля совсем не требуется текстов используемых им модулей — для этого достаточно, чтобы каждый используемый модуль был лишь специфицирован (в объёме, позволяющем построить правильное обращение к нему), а для тестирования его возможно (и даже, как мы покажем ниже, полезно) используемые модули заменять их имитаторами (заглушками). Во- вторых, каждая программа в какой-то степени подчиняется некоторым внутренним для нее, но глобальным для ее модулей соображениям (принципам реализации, предположениям, структурам данных и т.п.), что определяет её концептуальную целостность и формируется в процессе ее разработки. При восходящей разработке эта глобальная информация для модулей нижних уровней ещё не ясна в полном объеме, поэтому очень часто приходится их перепрограммировать, когда при программировании других модулей производится существенное уточнение этой глобальной информации (например, изменяется глобальная структура данных). В-третьих, при восходящем тестировании для каждого модуля (кроме головного) приходится создавать ведущую программу (модуль), которая должна подготовить для тестируемого модуля необходимое состояние информационной среды и произвести требуемое обращение к нему. Это приводит к большому объему «отладочного» программирования и в то же время не дает никакой гарантии, что тестирование модулей производилось именно в тех условиях, в которых они будут выполняться в рабочей программе.

Метод нисходящей разработки заключается в следующем. Как и в предыдущем методе сначала строится модульная структура программы в виде дерева. Затем поочередно программируются модули программы, начиная с модуля самого верхнего уровня (головного), переходя к программированию какого-либо другого модуля только в том случае, если уже запрограммирован модуль, который к нему обращается. После того, как все модули программы запрограммированы, производится их поочередное тестирование и отладка в таком же (нисходящем) порядке. При таком порядке разработки программы вся необходимая глобальная информация формируется своевременно, т.е. ликвидируется весьма неприятный источник просчетов при программировании модулей. Существенно облегчается и тестирование модулей, производимое при нисходящем тестировании программы. Первым тестируется головной модуль программы, который представляет всю тестируемую программу и поэтому тестируется при «естественном» состоянии информационной среды, при котором начинает выполняться эта программа. При этом все модули, к которым может обращаться головной модуль, заменяются их имитаторами (так называемые заглушки [5]). Каждый имитатор модуля представляется весьма простым программным фрагментом, сигнализирующим, в основном, о самом факте обращения к имитируемому модулю с необходимой для правильной работы программы обработкой значений его входных параметров (иногда с их распечаткой) и с выдачей, если это необходимо, заранее запасенного подходящего результата. После завершения тестирования и отладки головного и любого последующего модуля производится переход к тестированию одного из модулей, которые в данный момент представлены имитаторами, если таковые имеются. Для этого имитатор выбранного для тестирования модуля заменяется на сам этот модуль и добавляются имитаторы тех модулей, к которым может обращаться выбранный для тестирования модуль. При этом каждый такой модуль будет тестироваться при «естественных» состояниях информационной среды, возникающих к моменту обращения к этому модулю при выполнении тестируемой программы. Таким образом, большой объем «отладочного» программирования заменяется программированием достаточно простых имитаторов используемых в программе модулей. Кроме того, имитаторы удобно использовать для подыгрывания процессу подбора тестов путем задания нужных результатов, выдаваемых имитаторами.

Некоторым недостатком нисходящей разработки, приводящим к определенным затруднениям при её применении, является необходимость абстрагироваться от базовых возможностей используемого языка программирования, выдумывая абстрактные операции, которые позже нужно будет реализовать с помощью выделенных в программе модулей. Однако способность к таким абстракциям представляется необходимым условием разработки больших программных средств, поэтому её нужно развивать.

В рассмотренных методах восходящей и нисходящей разработок (которые мы будем называть классическими) модульная древовидная структура программы должна разрабатываться до начала программирования модулей. Однако такой подход вызывает ряд возражений: представляется сомнительным, чтобы до программирования модулей можно было разработать структуру программы достаточно точно и содержательно. На самом деле этого делать не обязательно. Например, модульная структура при конструктивном и архитектурном подходах к разработке программ [7] формируется в процессе программирования модулей.

Конструктивный подход к разработке программы представляет собой модификацию нисходящей разработки, при которой модульная древовидная структура программы формируется в процессе программирования модуля. Сначала программируется головной модуль, исходя из спецификации программы в целом, причем спецификация программы является одновременно и спецификацией ее головного модуля, так как последний полностью берет на себя ответственность за выполнение функций программы. В процессе программирования головного модуля, в случае, если эта программа достаточно большая, выделяются подзадачи (внутренние функции), в терминах которых программируется головной модуль. Это означает, что для каждой выделяемой подзадачи (функции) создаётся спецификация реализующего ее фрагмента программы, который в дальнейшем может быть представлен некоторым поддеревом модулей. Важно заметить, что здесь также ответственность за выполнение выделенной функции берёт головной (может быть, и единственный) модуль этого поддерева, так что спецификация выделенной функции является одновременно и спецификацией головного модуля этого поддерева. В головном модуле программы для обращения к выделенной функции строится обращение к головному модулю указанного поддерева в соответствии с созданной его спецификацией. Таким образом, на первом шаге разработки программы (при программировании её головного модуля) формируется верхняя начальная часть дерева, например, такая, которая показана на рис. Рис. 1.

Аналогичные действия производятся при программировании любого другого модуля, который выбирается из текущего состояния дерева программы из числа специфицированных, но пока еще не запрограммированных модулей. В результате этого производится очередное доформирование дерева программы, например, такое, которое показано на рис. Рис. 2.

Архитектурный подход к разработке программы представляет собой модификацию восходящей разработки, при которой модульная структура программы формируется в процессе программирования модуля. Но при этом ставится существенно другая цель разработки: повышение уровня используемого языка программирования, а не разработка конкретной программы. Это означает, что для заданной предметной области выделяются типичные функции, каждая из которых может использоваться при решении разных задач в этой области. Далее специфицируются, а затем и программируются отдельные программные модули, выполняющие эти функции. Так как процесс выделения таких функций связан с накоплением и обобщением опыта решения задач в заданной предметной области, то обычно сначала выделяются и реализуются отдельными модулями более простые функции, а затем постепенно появляются модули, использующие ранее выделенные функции. Такой набор модулей создаётся в расчёте на то, что при разработке той или иной программы заданной предметной области в рамках конструктивного подхода могут оказаться приемлемыми некоторые из этих модулей. Это

позволяет существенно сократить трудозатраты на разработку конкретной программы путём подключения к ней заранее заготовленных и проверенных на практике модульных структур нижнего уровня. Так как такие структуры могут многократно использоваться в разных конкретных программах, то архитектурный подход может рассматриваться как путь борьбы с дублированием в программировании. В связи с этим программные модули, создаваемые в рамках архитектурного подхода, обычно параметризуются для того, чтобы усилить применимость таких модулей путем настройки их на параметры.

В классическом методе нисходящей разработки рекомендуется сначала все модули разрабатываемой программы запрограммировать, а уж затем начинать нисходящее их тестирование [5]. Однако такой порядок разработки не представляется достаточно обоснованным: тестирование и отладка модулей может привести к изменению спецификации подчиненных модулей и даже к изменению самой модульной структуры программы, так что в этом случае программирование некоторых модулей может оказаться бесполезно проделанной работой. Нам представляется более рациональным другой порядок разработки программы, известный в литературе как метод нисходящей реализации. В этом методе каждый запрограммированный модуль начинают сразу же тестировать до перехода к программированию другого модуля.

Все эти методы имеют ещё различные разновидности в зависимости от того, в какой последовательности обходятся узлы (модули) древовидной структуры программы в процессе её разработки [1]. Это можно делать, например, по слоям (разрабатывая все модули одного уровня, прежде чем переходить к следующему уровню). При нисходящей разработке дерево можно обходить также в лексикографическом порядке (сверху-вниз, слева-направо). Возможны и другие варианты обхода дерева. Так, при конструктивной реализации для обхода дерева программы целесообразно следовать идеям Фуксмана, которые он использовал в предложенном им методе вертикального слоения [8]. Сущность такого обхода заключается в следующем. В рамках конструктивного подхода сначала реализуются только те модули, которые необходимы для самого простейшего варианта программы, которая может нормально выполняться только для весьма ограниченного множества наборов входных данных, но для таких данных эта задача будет решаться до конца. Вместо других модулей, на которые в такой программе имеются ссылки, в эту программу вставляются лишь их имитаторы, обеспечивающие, в основном, контроль над выходом за пределы этого частного случая. Затем к этой программе добавляются реализации некоторых других модулей (в частности, вместо некоторых из имеющихся имитаторов), обеспечивающих нормальное выполнение для некоторых других наборов входных данных. И этот процесс продолжается поэтапно до полной реализации требуемой программы. Таким образом, обход дерева программы производится с целью кратчайшим путём реализовать тот или иной вариант (сначала самый простейший) нормально действующей программы. В связи с этим такая разновидность конструктивной реализации получила название метода целенаправленной конструктивной реализации. Достоинством этого метода является то, что уже на достаточно ранней стадии создаётся работающий вариант разрабатываемой программы. Психологически это играет роль допинга, резко повышающего эффективность разработчика. Поэтому этот метод является весьма привлекательным.

Подводя итог сказанному, на Рис. 3 представлена общая схема классификации рассмотренных методов разработки структуры программы.

Контроль структуры программы

Для контроля структуры программы можно использовать три метода [5]:

- статический контроль,
- смежный контроль,
- сквозной контроль.

Статический контроль состоит в оценке структуры программы с точки зрения, хорошо ли программа разбита на модули с учётом значений рассмотренных выше

основных характеристик модуля.

Смежный контроль сверху — это контроль со стороны разработчиков архитектуры и внешнего описания ПС. Смежный контроль снизу — это контроль спецификации модулей со стороны разработчиков этих модулей.

Сквозной контроль — это мысленное прокручивание (проверка) структуры программы при выполнении заранее разработанных тестов. Является видом динамического контроля так же, как и ручная имитация функциональной спецификации или архитектуры ПС.

Следует заметить, что характер осуществления этих методов контроля зависит от выбранного метода разработки структуры программы: при классическом подходе они применяются до начала программирования модулей, а при конструктивном и архитектурном подходах — в процессе программирования модулей (в подходящие моменты времени

СИНТАКСИС И СЕМАНТИКА ЯЗЫКА.

Основные элементы синтаксиса языка Си/Си++

Основные синтаксические правила и записи программ на языке Си/Си++ сводятся к следующему:

■ Прописные и строчные буквы считаются разными символами. При записи идентификатора могут использоваться латинские буквы, цифры и символ подчеркивания (_). Идентификатор не может начинаться с цифры и содержать в себе пробельный символ. Длина идентификатора не ограничена.

* Пробельные символы могут размещаться в любом месте текста программы, но не внутри идентификатора.

- Комментарии заключаются в скобки вида: *Г***текст комментария** **1* и могут вводиться в любом месте программы и занимать любое количество строк. Еще один способ введения комментария - размещение его после символов двойного слэша *{II)*. Такой комментарий должен занимать конец строки и не должен переходить на следующую строку.
- Каждое (почти каждое) предложение языка кончается символом точка с запятой (;).
- В строке могут размещаться несколько операторов.

« Фигурные скобки выделяют составной оператор ({}). Все операторы, заключенные между такими скобками воспринимаются как один оператор.

- Все используемые типы, переменные, константы, функции должны быть описаны или объявлены до их первого использования. Объявления могут встречаться в любом месте программы.

Также ряд слов в языке Си/Си-н- имеет особое значение и не может использоваться в качестве идентификаторов. Такие зарезервированные слова называются служебными. Полный список служебных слов зависит от реализации языка, однако существует список основных служебных слов, определенный стандартом языка Си/Си++:

В таблице 1 приведены служебные слова основных арифметических типов данных, их размеры в памяти и диапазоны допустимых значений.

Пользователь также может вводить в программу свои собственные типы. Объявления пользовательских типов могут делаться в различных местах кода программы. Их место объявления влияет на область видимости вводимого типа. Синтаксис объявления пользовательского типа является следующий:

typedefопределение_типа идентификатор;

Где идентификатор - это вводимое пользователем имя нового типа, а определение_типа - описание этого типа. Например, запись typedefdoubleАг[ю];

объявляет тип пользователя с именем Аг как массив из 10 действительных чисел. В дальнейшем на этот тип можно ссылаться при объявлении переменных.

Описание переменных и констант

Константа - это лексема, представляющая изображение фиксированного числового, строкового или символьного значения. Константы могут использоваться непосредственно в тексте программы в любых операторах и выражениях. В языке Си/Си++ имеется несколько различных видов записи констант.

Запись целых констант. Целые константы могут быть десятичные, восьмеричные и шестнадцатеричные. Целые десятичные числа начинаются не с нуля. Например: 4,-128. Восьмеричные начинаются с символа нуля, после которого следуют восьмеричные цифры (от 0 до 7). Например: 032. Восьмеричные константы не могут превышать значения 037777777777. Шестнадцатеричные константы начинаются с символов нуля и X или **x,** после которых следуют шестнадцатеричные цифры (от 0 до F, которые можно записывать и в верхнем, и в нижнем регистрах). Например: 0XF01. Шестнадцатеричные константы не могут превышать значения OxFFFFFFFF. Тип целой константы компилятор определяет по диапазону, в котором лежит значение константы.

Запись вещественных констант. Если в записи числовой константы присутствует десятичная точка (например: 2.5) или экспоненциальное расширение (например: 1E-8), то компилятор рассматривает ее как вещественное число и ставит ей в соответствии тип double.

Запись символьных и строковых констант. Символьные константы должны заключаться в одинарные кавычки. Эти константы хранятся как char, signedcharили unsignedchar. Например: ‘A’, ‘d’. Строковые константы заключаются в двойные кавычки. Например: “Введите исходные данные”.

Особую разновидность символьных констант представляют *управляющие символы* или *эскейп - последовательности*(ESC- sequence), с помощью которых можно задать символьную константу указанием ее кода и управлять консольным выводом на экран. Последовательности символов, начинающихся со знака обратной косой черты 'V называют эскейп - последовательностью. В таблице 2 представлены основные используемые значения таких последовательностей.

Например, константа

ТИмяУЧАдресЩИвановММосква”

будет при консольном отображении на экране выглядеть как

"Имя" Адрес Иванов Москва

Для использования внутренних кодов символов на IBM- совместимых компьютерах применяется таблица кодов ASCII(Приложение 1). Выбирая из кодовой таблицы подходящее значение, можно использовать их в программе вместо явных изображений символов. Например, константа, соответствующая заглавной латинской букве А, может быть представлена тремя способами: 'A', '\101 ‘\x41’.

Перечисляемые константы. Язык Си/Си-н- позволяет определять последовательность целочисленных именованных констант. Описание констант перечисляемого типа начинается со служебного слова enum, а последующий список значений констант заключается в фигурные скобки.

enumимя *{значения* };

Например, запись

enumcolor{red,yellow,green *};*

объявляет переменную с именем color, которая может принимать константные значения red, yellowили green. Эти значения в дальнейшем можно использовать как константы для присваивания переменной colorили для проверки ее значения. Этим константам соответствуют целые значения, определяемые их местом в списке объявления: red- 0, yellow- 1, green- 2. Эти значения можно изменить, если инициализировать константы явным образом. Например, объявление

enum color {red,yellow = 3,green= red + *l};*

приведет к тому, что значения констант будут равны: red- 0, yellow- 3, green- 1.

При этом не обязательно должна соблюдаться уникальность значений. Несколько констант в списке могут иметь одинаковые значения.

Именованные константы. Именованная константа - это константа, которой присвоен некоторый идентификатор. Объявление именованной константы является указателем для компилятора заменить во всем тексте этот идентификатор значением константы. Такая замена производится только в процессе компиляции и не отражается на исходном тексте. Цель объявления именованной константы - сделать текст более осмысленным и облегчить при необходимости изменение значения константы во всем тексте.

Объявление именованных констант происходит с помощью служебного слова constследующим образом:

consttип имя_константы = значение;

Например;

constfloatPi= 3.14159;

В качестве значения константы можно указывать и константное выражение, содержащее ранее объявленные константы. Например, если объявлена константа Pi, то далее можно объявить константы

constfloatPi2 = 2 * Pi; //удвоенное число Пи

Если тип константы не указан, то по умолчанию он определяется как **int.**

Например: **constmaxint= 12345;**

Попытка где-то в тексте изменить значение именованной константы приведет к ошибке компиляции с выдачей соответствующего сообщения.

Переменная является идентификатором, обозначающим некоторую область в памяти, в которой хранится значение переменной. Это значение может измениться во время выполнения программы. Объявление переменной может происходить в любом месте программы и имеет вид:

тип список_идентификаторов_переменных;

Например: **intA;**

doubleB1;

Список идентификаторов может состоять из нескольких идентификаторов переменных, разделенных запятыми. Например:

intx1, x2;

Так как, при объявлении переменной ее значение неопределенно (оно является случайным), то одновременно с объявлением переменные могут быть инициализированы, т.е. им присвоены начальные значения. Например:

intx1 = 1, x2 = 2;

Для инициализации можно использовать не только константы, но и произвольные выражения, содержащие объявленные ранее константы и переменные. Например:

intx1 = 1, x2 = 2 * x1;

Так же следует отметить, что константы и переменные должны быть объявлены до их первого использования.

4.ТИПЫ ДАННЫХ, ВЫРАЖЕНИЯ И ОПЕРАЦИИ В ЯЗЫКЕ ПРОГРАММИРОВАНИЯ С++

КОНЦЕПЦИЯ ТИПОВ ДАННЫХ В ЯЗЫКЕ С++

Типы данных

Концепция типов данных является важнейшей стороной любого языка программирования. С типом величины связаны три ее свойства: форма внутреннего представления, множество принимаемых значений и множество допустимых операций. Особенность языка Си состоит в большом разнообразии типов, схематически представленном на рис. 2.

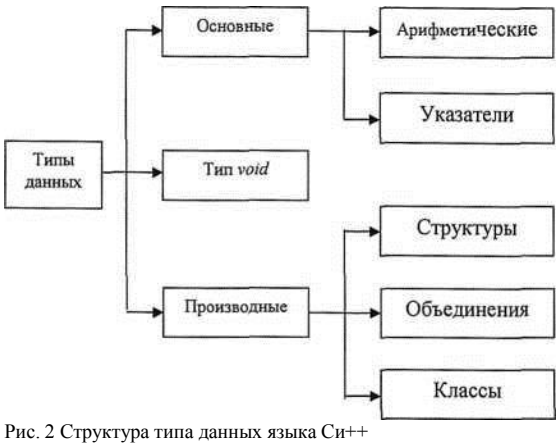


Рис. 2 Структура типа данных языка Си++

В таблице 1 приведены служебные слова основных арифметических типов данных, их размеры в памяти и диапазоны допустимых значений.

Пользователь также может вводить в программу свои собственные типы. Объявления пользовательских типов могут делаться в различных местах кода программы. Их место объявления влияет на область видимости вводимого типа. Синтаксис объявления пользовательского типа является следующий:

typedef определение_типа идентификатор;

Где идентификатор - это вводимое пользователем имя нового типа, а определение_типа - описание этого типа. Например, запись typedef double Аг[ю];

объявляет тип пользователя с именем Аг как массив из 10 действительных чисел. В дальнейшем на этот тип можно ссылаться при объявлении переменных. Например:

Аг А = {1,2,3,4,5,6,7,8,9,10}.

АРИФМЕТИЧЕСКИЕ ТИПЫ ДАННЫХ

Таблица 1

тип данных	*название*	*размер,бит*	*Диапазон значений*
unsigned char	беззнаковый малый целый или коды символа	8	0..255
char	малый целый или код символа	8	-128 .. 127
unsigned int	беззнаковый целый	16	0..65535
short int (short)	короткий целый	16	-32768 .. 32767
unsigned short	беззнаковый короткий целый	16	0..65535
int	целый	16	-32768 .. 32767
unsigned long	беззнаковый длинный целый	32	0..4294967295
long	длинный целый	32	-214748348 . . 2147483647
float	Вещественный одинарной точности	32	3.4E-38 .. 3.4E+38
double	Вещественный двойной точности	64	1.7E-308 .. 1.7E+308
long double	Вещественный максимальной точности	80	3.4E-4932 .. 1.1E+4932

Пр

иведение типов при вычислении выражений

В арифметических выражениях, содержащих элементы различных арифметических типов, в процессе вычислений автоматически осуществляет преобразование типов. Это стандартное преобразование всегда осуществляется по принципу: если операция имеет операнды разных типов, то тип операнда «младшего» типа приводится к типу операнда «старшего» типа. Иначе говоря, менее точный тип приводится к более точному. Например, если в операции участвует короткое целое и длинное целое, то короткое приводится к длинному; если участвует целый и действительный операнды, то целый приводится к действительному и т.д. Таким образом, после подобного приведения типов оба операнда оказываются одного типа. И результат применения операции имеет тот же тип.

Все это относиться к арифметическим операциям, но не относиться к операции присваивания. Присваивание сводиться к приведению типа результата выражения к типу левого операнда. Если тип левого операнда «младше», чем тип результата выражения, возможна потеря точности или вообще неправильный результат.

Рассмотрим примеры неявного автоматического преобразования типов. В результате действия следующих операторов

double a = 5.4, **b =** 2;

int c = a*b;

переменная с получит значение 10, хотя истинное значение должно быть равно 10.8. Это значение действительно будет вычислено в результате умножения а * Ь, но затем дробная часть будет отброшена, поскольку с - целая переменная.

Результатом выполнения операторов

int m = 1, n = 2; double A = m
/ n;

будет значение А = 0. Поскольку т и п - целые переменные, то деление т / п сведется к целочисленному делению с отбрасыванием дробной части, результат которого равен нулю.

Как видно из приведенных примеров, неявное автоматическое приведение типов не всегда дает желаемый результат. Это можно исправить, применив операцию явного приведение типов, которая имеет следующие конструкции:

(тип) операнд; тип
(операнд);

Если в предыдущем примере, который давал неверное значение переменной А, применить во втором операторе явное приведение типа:

double A = (double) m / n ;

или

double A = double (m) / n;

то получим правильный результат А = 0.5. В этом случае переменная т, которой применяется операция приведения типа, рассматривается как действительная величина типа double, тогда и переменная п неявно приводиться к типу double, так что деление осуществляется уже не с целыми, а с действительными числами.

КОНСТАНТЫ И ПЕРЕМЕННЫЕ

Константа - это лексема, представляющая изображение фиксированного числового, строкового или символьного значения. Константы могут использоваться непосредственно в тексте программы в любых операторах и выражениях. В языке Си/Си++ имеется несколько различных видов записи констант.

Запись целых констант. Целые константы могут быть десятичные, восьмеричные и шестнадцатеричные. Целые десятичные числа начинаются не с нуля. Например: 4,-128. Восьмеричные начинаются с символа нуля, после которого следуют восьмеричные цифры (от 0 до 7). Например: 032. Восьмеричные константы не могут превышать значения 037777777777. Шестнадцатеричные константы начинаются с символов нуля и X или x, после которых следуют шестнадцатеричные цифры (от 0 до F, которые можно записывать

и в верхнем, и в нижнем регистрах). Например: 0XF01. Шестнадцатеричные константы не могут превышать значения OxFFFFFFFF. Тип целой константы компилятор определяет по диапазону, в котором лежит значение константы.

Запись вещественных констант. Если в записи числовой константы присутствует десятичная точка (например: 2.5) или экспоненциальное расширение (например: 1E-8), то компилятор рассматривает ее как вещественное число и ставит ей в соответствии тип double.

Запись символьных и строковых констант. Символьные константы должны заключаться в одинарные кавычки. Эти константы хранятся как char, signed char или unsigned char. Например: A', 'd\ Строковые константы заключаются в двойные кавычки. Например: "Введите исходные данные".

Особую разновидность символьных констант представляют *управляющие символы* или э*скейп - последовательности*(ESC - sequence), с помощью которых можно задать символьную константу указанием ее кода и управлять консольным выводом на экран. Последовательности символов, начинающихся со знака обратной косой черты 'V называют эскейп - последовательностью. В таблице 2 представлены основные используемые значения таких последовательностей.

Например, константа

ТИмяУПАдреЩпИвановШосква"

будет при консольном отображении на экране выглядеть как

"Имя" Адрес

Иванов Москва

Для использования внутренних кодов символов на IBM - совместимых компьютерах применяется таблица кодов ASCII (Приложение 1). Выбирая из кодовой таблицы подходящее значение, можно использовать их в программе вместо явных изображений символов. Например, константа, соответствующая заглавной латинской букве А, может быть представлена тремя способами: 'A', ЛЮТ, 4x41'.

Перечисляемые константы. Язык Си/Си++ позволяет определять последовательность целочисленных именованных констант. Описание констант перечисляемого типа начинается со служебного слова enum, а последующий список значений констант заключается в фигурные скобки.

enum имя *{значения* };

Например, запись

enum color {red,yellow,green *};*

объявляет переменную с именем color, которая может принимать константные значения red, yellow или green. Эти значения в дальнейшем можно использовать как константы для присваивания переменной color или для проверки ее значения. Этим константам соответствуют целые значения, определяемые их местом в списке объявления: red - 0, yellow - 1, green - 2. Эти значения можно изменить, если инициализировать константы явным образом. Например, объявление

enum color {red,yellow = *3*,green= red *+1};*

приведет к тому, что значения констант будут равны: red - 0, yellow - 3, green - 1. При этом не обязательно должна соблюдаться уникальность значений. Несколько констант в списке могут иметь одинаковые значения.

Именованные константы. Именованная константа - это константа, которой присвоен некоторый идентификатор. Объявление именованной константы является указателем для компилятора заменить во всем тексте этот идентификатор значением константы. Такая замена производится только в процессе компиляции и не отражается на исходном тексте. Цель объявления именованной константы - сделать текст более осмысленным и облегчить при необходимости изменение значения константы во всем тексте.

Объявление именованных констант происходит с помощью служебного слова

const следующим образом:

const тип имя_константы = значение;

Например:

const float Pi = 3.14159;

В качестве значения константы можно указывать и константное выражение, содержащее ранее объявленные константы. Например, если объявлена константа Pi, то далее можно объявить константы

const float Pi2 = 2 * Pi; //удвоенное число Пи

Если тип константы не указан, то по умолчанию он определяется как int.
Например: const maxint = 12345;

Попытка где-то в тексте изменить значение именованной константы приведет к ошибке компиляции с выдачей соответствующего сообщения.

Переменная является идентификатором, обозначающим некоторую область в памяти, в которой хранится значение переменной. Это значение может измениться во время выполнения программы. Объявление переменной может происходить в любом месте программы и имеет вид:

тип список_идентификаторов_переменных;

Например: int A;

double B1;

Список идентификаторов может состоять из нескольких идентификаторов переменных, разделенных запятыми. Например:

int x1, x2;

Так как, при объявлении переменной ее значение неопределенно (оно является случайным), то одновременно с объявлением переменные могут быть инициализированы, т.е. им присвоены начальные значения. Например:

int x1 = 1, x2 = 2;

Для инициализации можно использовать не только константы, но и произвольные выражения, содержащие объявленные ранее константы и переменные. Например:

int x1 = 1, x2 = 2 *x1;

Так же следует отметить, что константы и переменные должны быть объявлены до их первого использования.

ОСНОВНЫЕ ОПЕРАЦИИ ЯЗЫКА СИ

Арифметические операции и операции присваивания Операции подобны встроенным функциям языка. Они применяются к выражениям - *операндам.* Большинство операций имеют два операнда, один из которых помещается перед знаком операции, а другой - после. Такие операции называются бинарными. Существуют и унарные операции, имеющие только один операнд, как правило, помещаемый после знака операции.

В сложных выражениях последовательность выполнения операций определяется скобками, старшинством операций, а при одинаковом старшинстве - ассоциативностью операций.

Арифметические операции. Арифметические операции применяются к действительным и целым числам, существуют следующие арифметические операции:

Таблица 3

Бинарные арифметические операции

Обозначение	*Операция*	*Пример*
+	сложение	X + Y
-	вычитание	X - Y
*	умножение	X* Y
/	деление	X/Y
%	Остаток целочисленного деления	X%Y

Таблица 4

Унарные арифметические операции

Обозначение	*Операция*	*Пример*
+	Унарный плюс (подтверждение знака)	+7
-	Унарный минус (изменение знака)	-X
++	инкремент	i++
--	декремент	i--

Для арифметических операций действуют следующие правила:

* Бинарные операции сложения (+) и вычитания (-) применимы к целым и действительным числам.
- В операциях умножения (*) и деления (/) операнды могут быть любых арифметических типов. При разных типах операндов применяются стандартные правила автоматического приведения типов.
- В операции вычисления остатка от деления (%) оба операнда должны быть целыми числами.
* В операциях деления и вычисления остатка второй операнд не может быть равен нулю. Если оба операнда в этих операциях целые, а результат деления является не целым числом, то знак результата вычисления остатка совпадет со знаком первого операнда. Округление всегда осуществляется по направлению к нулю.

" Унарные операции инкремента (++) и декремента (—) сводятся к увеличению (++) или уменьшению (—) операнда на единицу. Операции применимы к операндам, представляющим собой выражения любых арифметических типов. Причем выражение должно быть модифицируемым значением, т.е. должно допускать изменение. Например, ошибочным является выражение (a+Ь)++, поскольку (a+b) не является переменной, которую можно модифицировать.

* Операции инкремента и декремента выполняются быстрее, чем обычное сложение и вычитание. Поэтому, если переменная a должна быть увеличена на 1, лучше применить операцию (++), чем выражения a=a+1.

Операции присваивания. В Си++ определен ряд операций присваивания.

Помимо простой операции присваивания (=) все прочие являются составными операциями. Они присваивают первому операнду результат применения соответствующей простой операции, указанной перед символом (=), к первому и второму операндам. Например, выражение X+=Y эквивалентно выражению X=X+Y, но записывается компактнее и может выполнятся быстрее. Аналогично определяются и другие операции присваивания: X%=Y эквивалентно X=X%Y и т.д.

При записи составных операций присваивания между символом операции и знаком

равенства пробел не допускается.

В операциях присваивания первый операнд не может быть нулевым указателям.

Операции отношения и эквивалентности

Операции отношения и эквивалентности используются при сравнении двух операндов. Они возвращают **true** - истина, если указанное соотношение операндов выполняется, и **false** - ложь, если соотношение не выполняется. Определены следующие операции отношения:

Таблица 8

Основные операции отношения

Обозначен	*Операция*	*Пример*
==	**Равно**	
!=	**Не**	**X!=Y**
<	**Меньше**	**X<Y**
>	**Больше**	**00**
<=	**Меньше**	**A<=|**
>=	**Больше**	**X>=Y**

Операнды должны иметь совместимые типы, за исключением целых и дейстшяельншс типов, которые могут сравниваться друг с другом.

Логические операции

Логические операции принимают в качестве операндов выражения скалярных типов и возвращают результат булева типа: true или false. В С++ любое выражение, имеющее некоторое

значение, может использоваться в логических операциях. Так если значение выражения 0, то оно трактуется как false, любое другое значение трактуется как true.

Таблица 9

Основные логические операции

Обозначение	*Операция*	*Пример*
!	**Отрицание**	**!A**
&&	**Логическое И**	**A&&B**
II	**Логическое**	**A||B**

Унарная операция логического отрицания (!) возвращает true, если операнд возвращает ненулевое значение. Таким образом, выражение !А эквивалентно выражению А==0.

Операция логического И (&&) возвращает true, если оба ее операнда возвращают ненулевые значения. Если хотя бы один операнд возвращает 0 (false), то операция И также возвращает false.

Операция логического ИЛИ (||) возвращает true, если хотя бы один ее операнд возвращает ненулевое значение. Если оба операнда возвращают 0 (false), то операция ИЛИ также возвращает false.

5.ОПЕРАТОРЫ ЯЗЫКА ПРОГРАММИРОВАНИЯ СИ++ И УПРАВЛЕНИЕ ИХ ИСПОЛНЕНИЕМ

ОПЕРАТОРЫ УСЛОВИЯ И ПЕРЕДАЧИ УПРАВЛЕНИЯ

Условный оператор выбора

Оператор **if** предназначен для выполнения тех или иных действий в зависимости от истинности или ложности некоторого условия. Условие задается выражением, имеющим результат булева типа.

Оператор имеет две формы:

1) if(yслoBMe)oпepaTop;

Скобки, обрамляющие условие, обязательны.

Условием может быть выражение, преобразуемое в булев тип. Если условие истинно, то указанный в конструкции оператор выполняется. В противном случае

управление сразу передается следующему за конструкцией if оператору.

2) **If (условие) оператор1; else**
оператор2;

Если условие возвращает true, то выполняется первый из указанных операторов, в противном случае выполняется второй оператор. Обратите внимание, что в конце первого оператора перед ключевым словом else ставится точка с запятой.

При вложенных конструкциях if могут возникнуть неоднозначности в понимании того, к какой из вложенных конструкций if относится элемент else. Компилятор всегда считает, что else относится к последней из конструкций if, в которой не было раздела else. Например,

if(ycnoBHel) if(ycnoBwe2)
оператор"!; else оператор2;

else будет отнесено компилятором ко второй конструкции if, т.е. оператор2 будет выполняться в случае, если первое условие истинно, а второе ложно.

Если же вы хотите отнести else к первому if, это надо записать в явном виде с помощью фигурных скобок:

```
if(ycnoBne1)
{
    if(ycnoBwe2) оператор"!;
}
    else оператор2;
```

В конструкциях в качестве оператор, оператор"! и оператор2 понимается использование одного оператора или выражения. Если необходимо выполнение нескольких операторов, то следует использовать составной оператор.

Поскольку в Си++ любое арифметическое значение может преобразовываться к булеву типу, т.е. если значение выражения 0, то оно трактуется как false, а любое другое значение трактуется как true. То в условии можно использовать практически любые арифметические выражения. Например:

int a,b,c; if(a—b/c) ...;

В данном случае условие if(a—Ь/с) будет false, если результат выражения a-b/c буде нуль, и условие будет true при всех остальных результатах выражения.

Так же, следует предостеречь от довольно распространенной ошибки: случайного применения вместо операции эквивалентности (==) операции присваивания (=). Например, если по ошибке вместо оператора

if (A==2)

используется оператор

if (A=2)

то это не будет расценено как синтаксическая ошибка. Результат операции A=2 будет трактоваться как true независимо от того, чему было равно значение переменной A до выполнения этого ошибочного оператора. К тому же выполнение этого оператора приведет к изменению переменной.

Условный оператор множественного выбора

Оператор **switch** позволяет провести анализ значения некоторого выражения и в зависимости от его значения выполнить те или иные действия. В общем случае формат записи оператора является следующим:

```
switch (выражение_выбора){ case
значение_1 : оператор_1;
  break; // не обязательно
case значение_п : оператор_п;
  break; // не обязательно default: оператор;
// не обязательно }
```

В этой конструкции выражение выбора должно иметь порядковый тип - целый,

перечислимый и т.д. Поэтому, например, нельзя использовать выражения, возвращающие действительные числа или строки.

Значения, указываемые в метках **case,** должны быть константными выражениями, соответствующими возможным значениям выражения выбора. После значения ставится двоеточие <:>, а затем пишется оператор (может писаться составной оператор), который должен выполняться, если выражение приняло указанное в метке значение.

Если значение выражения выбора совпало со значением, указанным в одной из меток case, то выполняется оператор, записанный после этой метки, после чего, если не принять соответствующих мер, будут выполняться все последующие операторы остальных меток. Поскольку это обычно нежелательно, то, как правило, после оператора, который должен выполняться, записывают оператор break, который прерывает выполнение структуры switch и управление передается следующему за ней оператору.

Если значение выражения выбора не соответствует ни одному из перечисленных в метках, то выполняется оператор, следующий за меткой default. Впрочем, метка default является не обязательной.

Значения в метках могут содержать константы и константные выражения, которые совместимы по типу с объявленным выражением и которые компилятор может вычислить заранее, до выполнения программы. Недопустимо использование переменных и функций. В метках не допускается повторение одних и тех же значений, поскольку в этом случае выбор был бы неоднозначным.

Приведенный ниже пример анализирует переменную Key типа char, содержащую символ, введенный пользователем.

```
switch (Key) {
      case ‘y’: case ‘Y’: cout« “Вы нажали клавишу У
или y’’; break;
      case ‘n’: case 'N': cout« “Вы нажали клавишу N
или n”; break;
default;
    cout«“Bbi не нажали клавишу Y/y и N/n”;
        }
```

При необходимости выполнять одинаковые действия при нескольких значениях выражения выбора, надо размещать подряд несколько меток case.

Оператор передачи управления

Оператор **goto** позволяет прервать обычный поток управления и передать управление в произвольную точку кода, помеченную специальной меткой и имеет следующую форму:

goto метка;

Метка в тексте программы обозначается идентификатором с последующим двоеточием. Например

Lbegin:

Метка отмечает точку, в которую передается управление оператором goto. Метка может располагаться в любом месте блока, как после оператора goto передающего на нее управление, так и до этого оператора. При этом нужно иметь в виду, что передача управления извне или внутрь цикла может приводить к непредсказуемым последствиям.

Метки можно использовать внутри всей функции, в которой они указаны, но на них нельзя ссылаться вне тела функции.

Следует помнить, что чрезмерно широкое применение оператора goto делает структуру программы крайне запутанной и плохо читаемой, что затрудняет ее дальнейшее ее сопровождение.

ОПЕРАТОРЫ ЦИКЛОВ

Оператор цикла с постусловием

Структура **do...while** используется для организации циклического выполнения

оператора или совокупности операторов, называемых телом цикла, до тех пор, пока не окажется нарушенным некоторое условие. Синтаксис данного оператора является следующим:

do оператор while (условие);

Структура работает следующим образом: Выполняется оператор тела цикла. Затем вычисляется условие - выражение, которое должно возвращать результат булева типа. Если выражение возвращает true (не нулевое значение), то повторяется выполнение тела цикла и после этого снова вычисляется условие - выражение. Такое циклическое повторение цикла продолжается до тех пор, пока проверяемое условие - выражение не возвратит false (нуль). После этого цикл завершается и управление передается оператору, следующему за структурой do...while.

Поскольку проверка выражения осуществляется после выполнения тела цикла, то цикл будет заведомо выполнен хотя бы один раз, даже если выражение сразу ложно.

В конструкции в качестве оператор понимается использование одного оператора или выражения. Если необходимо выполнение нескольких операторов, то следует использовать составной оператор.

Ниже приведен пример, в котором осуществляется проверка вводимых значений пользователем с клавиатуры. Ввод значений будет продолжаться до тех пор, пока пользователь не введет нужные значения:

```
#include<iostream.h>
main()
{
double A,B;
        do {
        cout «"Введите значения больше нуля:"«епб1;
        cout «"A=";
        cin »A;
        cout <<"B=";
        cin »B;
        if ((A<=0)||(B<=0))
cout «"Неправильные значения"«епб1;
  } whiie((A<=0)||(B<=0));
        }
```

Оператор цикла с предусловием

Оператор **while** используется для организации циклического выполнения тела цикла, пока выполняется некоторое условие. Синтаксис структуры:

while (условие) оператор;

Структура работает следующим образом: Сначала вычисляется условие, которое должно возвращать результат булева типа. Если выражение возвращает true (ненулевое значение), то выполняется оператор тела цикла, после чего опять вычисляется выражение, определяющее условие. Такое циклическое повторение выполнения оператора и проверки условия продолжается до тех пор, пока условие не вернет false (нуль). После этого цикл завершается и управление передается оператору, следующему за структурой while.

Если необходимо выполнение нескольких операторов, то следует использовать составной оператор.

Поскольку проверка выражения осуществляется перед выполнением оператора тела цикла, то, если условие сразу ложно, оператор не будет выполнен ни одного раза.

Ниже приведен пример, в котором осуществляется вывод натуральных чисел, не превышающих заданного значения.

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

```
        {
int N,k=0; cout«"N="; cin »N;
        while (k<=N){ cout«k«", k++;
        }
        }
```

Универсальный оператор цикла

Оператор **for** обеспечивает циклическое повторение некоторого оператора (в частности, составного оператора) заданное число раз. Повторяемый оператор называется телом цикла. Повторение цикла обычно определяется некоторой управляющей переменной, которая изменяется при каждом выполнении тела цикла. Повторение завершается, когда управляющая переменная достигает заданного значения.

Синтаксис структуры for:

for (выражение-!; выражение2; выражение3) оператор;

где выражение) задает начальное значение переменной, управляющей циклом, выражение2 является условием продолжения цикла, а выражение3 изменяет управляющую переменную.

Структура for работает следующим образом: Сначала выполняется выражение) (оно может состоять и из ряда выражений, разделенных запятой т.е. может использоваться операция последования). Это выражение задает начальные значения переменной (или переменных) цикла.

Затем проверяется **выражение2 - условие** продолжения цикла. Если условие истинно (возвращает **true** - ненулевое значение), то выполняется тело цикла - **оператор,** записанный в структуре **for.** После завершения тела цикла выполняется **выражение3,** определяющее обычно изменение переменной цикла. Затем опять проверяется условие, записанное как **выражение2,** и при истинности этого условия выполнение цикла продолжается. Как только в каком-нибудь цикле **выражение2** вернет **false** (нулевое значение), цикл прерывается и управление передается оператору, расположенному следом за структурой **for.**

Приведем примеры использования цикла **for.**

.V-1 |

Найти сумму ряда .

.=o г + I

```
#include<iostream.h> main()
{
int N; cout <<"N="; cin »N; double S=0;
for(int i=0;i<N; i++) S+=1/doub!e(i+1); cout
«"S="«S;
}
```

Здесь первое выражение в структуре **for** вводит целую переменную **i,** являющуюся счетчиком циклов, и инициализирует ее значением **0.** Второе выражение проверяет условие завершения цикла. В данном случае цикл должен завершиться, когда переменная **i,** используемая в теле, примет значение, равное значению переменной **N,** введенное пользователем с клавиатуры. Третье выражение структуры **for** увеличивает после каждого выполнения цикла значение **i** на **1** с помощью операции инкремента.

Теперь рассмотрим этот же приме пример, но с использованием в структуре **for** операции запятая. Если объявить переменные **i** и **S** до начала цикла, собственно цикл можно весь разместить в заголовке структуры **for:**

```
#include<iostream.h>
main()
{
```

int N,i; cout«"N="; cin »N; double S;
for(i=0,S=0; i<N; S+=1/double(i+1),i++); cout
<<"S="«S;
}

В этом примере первое выражение структуры **for** включает в себя два оператора, разделенных операцией запятая и задающих начальные значения переменной **S,** накапливающей сумму, и переменной цикла **i.** Третье выражение структуры **for** включает также два оператора - формирование суммы и постфиксный инкремент переменной цикла **i.** После структуры **for** стоит точка с запятой, что означает пустое тело цикла.

В приведенных примерах переменная цикла увеличивалась на единицу при каждом цикле. Можно организовывать циклы с изменением переменной на любое значение.

Выражения в структуре **for** являются необязательными. Иногда может отсутствовать первое выражение, если начальное значение управляющей переменной задано где-то в другом месте программы. Если отсутствует второе выражение, предполагается, что условие продолжения цикла всегда истинно и таким образом создается бесконечно повторяющийся цикл. Выйти из такого цикла можно, проверив в теле цикла какие-то условия и прервав выполнение передачей управления за пределы цикла оператором **goto** или применить другие способы прерывания.

Может отсутствовать в структуре for и третье выражение, если приращение переменной осуществляется операторами в теле структуры или если приращение не требуется.

При пропуске какого-то из выражений, точка с запятой после пропущенного выражения (кроме третьего) должна писаться. Например, в заголовке

for (; i<10;) ...;

пропущено первое условие и третье.

Если условие продолжения цикла не удовлетворяется с самого начала, то операторы тела структуры for не выполняются ни разу.

Операторы цикла for могут быть вложенными.

Операторы прерывания цикла

В некоторых случаях желательно прервать повторение цикла, проанализировав какие-то условия внутри него. Это может потребоваться в тех случаях, когда проверки условия окончания цикла громоздкие, требуют многоэтапного сравнения и сопоставления каких-то данных и все эти проверки просто невозможно разместить в выражении условия операторов for, do...while или while.

Один из возможных вариантов решения этой задачи - использование оператора break. Оператор break прерывает выполнение тела любого цикла for, do или while и передает управление следующему за циклом выполняемому оператору.

Например, пусть в цикле осуществляется ввод с клавиатуры, который будет продолжаться до тех пор, пока пользователь не введет нужные значения:

#include<iostream.h> main()
{
double A,B; while(1){
cout «"Введите значения больше Нуля:"«endl;
cout «"A="; cin »A; cout «"B="; cin »B;
if ((A<=0)||(B<=0)) cout «"Неправильные значения"«endl; else break; // Прерывание цикла }
}

Еще один способ прерывания цикла - использование оператора goto, передающего управление какому-то оператору, расположенному вне тела цикла.

Описанные способы прерывали выполнение цикла. Имеется еще процедура continue, которая прерывает только выполнение текущей итерации, текущего выполнения

тела цикла и передает управление на следующую итерацию.

Чтобы продемонстрировать применение continue, рассмотрим пример нахождения

N j

произведения ГГ , которое можно организовать следующим образом:

/=1 ' - 5

i*5

```
#include<iostream.h>
main()
{
int N;
cout «"N="; cin » N;
double P=1;
for (int i=1; i<N; i++)
if (i==5) continue; // Прерывание текущей итерации цикла
else P*=1/double(i-5); cout «"P="«P;
}
```

В этом варианте при i равном 5 текущая итерация прерывается и произведение не вычисляется, но цикл не прекращается.

ОДНОМЕРНЫЕ И МНОГОМЕРНЫЕ МАССИВЫ

Одномерные массивы

Массив представляет собой структуру данных, позволяющую хранить под одним именем совокупность данных любого, но только одного какого-то типа. Массив характеризуется своим именем, типом хранимых элементов и размерностью (количеством хранимых элементов). Объявление переменной как одномерного массива имеет вид:

тип имя_массива [размерность]

В качестве размерности в объявлении массива разрешено использовать только константные выражения, ввиду чего такой массив будет являться статическим. Тип массива может быть любым.

Например,

int A [10];

объявляет массив с именем А, содержащий 10 целых чисел.

Доступ к элементам массива осуществляется выражением

имя_массива [номер_элемента]

где номер_элемента - индекс, являющийся целочисленным значением в диапазоне от 0 до размерность - 1. То есть номер первого элемента равен 0, а номер последнего элемента на 1 меньше размерности массива. Для предыдущего примера, $A[0]$ - значение первого элемента, Л[1] - второго, $A[9]$ - последнего.

Работа с массивами, как правило, непосредственно связана о использованием операторов цикла.

Ниже приведен пример заполнения массива числами Фибоначчи, первые 2 из которых равны 1, а каждое последующее равно сумме двух предыдущих.

```
#include<iostream.h>
main()
{
int B[20];
B[0]=1;
B[1]=1;
for (int i=2; i<20; i++)
B[i]=B[i-1]+B[i-2]; for (i=0; i<20; i++)
cout «"B["«i«"]="«B[i]«endl;
}
```

При объявлении массива значения элементов носят случайный характер, поэтому

Таким образом, после данных объявлений A[2] будет содержать значение 3, а S[3] значение

иногда желательно совместить объявление массива с заданием элементам начальных значений, т.е. инициализацией. Эти значения перечисляются в списке инициализации после знака равенства, разделяются запятыми и заключаются в фигурные скобки.

Если начальных значений меньше, чем элементов в массиве, оставшиеся элементы автоматически получают нулевые начальные значения. Например, оператор

int A[10] ={1,2,3};

задает значения первым трем элементам, а остальные будут равны 0.

Оператор int A[10] ={0};

присваивает нулевые значения всем элементам массива.

В объявлении массива со списком инициализации размерность массива можно не указывать. Тогда количество элементов массива будет равно количеству элементов в списке начальных значений. Например, объявление

double S[] ={0.1,2.0,-3.2,0,6.5};

создает массив из пяти элементов.

В объявлении массива в качестве размерности лучше всегда использовать именованные константы. Ниже приведен пример объявление массива, заполнение его случайными значениями от -10 до 10 и подсчет суммы его элементов:

```
#include<iostream.h>
#include<stdlib.h>
main()
{
randomize(); double c[15];
for (int i=0; i<15; i++)
c[i]=10-double(random(201 ))/10; for (int
i=0; i<15; i++)
cout «"c["«i«"]="«c[i]«endl; double S=0;
for (int i=0; i<15; i++) S+=c[i];
cout«"S="«S;
}
```

Если в дальнейшем потребуется массив не из 15 элементов, а, например, из 100, нужно будет изменить размерность массива и в объявлении, и во всех операторах, работающих с этим массивом (в данном случае в операторах for. Таких операторов в разных программах может быть очень много и поэтому она плохо масштабируется. Грамотнее будет реализовать этот пример следующим образом:

```
#include<iostream.h>
#include<stdlib.h>
main()
{
const int N=15; // объявление константы
randomize();
double c[N];
for (int i=0; i<N; I++)
c[i]=10-double(random(201 ))/10; for (int
i=0; i<N; i++)
cout <<"c["«i«"]="«c[i]«endl; double S=0;
for (int i=0; i<N; i++) S+=c[i];
cout«"S="«S;
}
```

В этом случае мы вводим именованную константу **N** и используем ее во всех операторах, **в** которых требуется указать размерность массива. Тогда при необходимости изменения размерности массива, достаточно будет изменить его только в объявлении константы N.

const int N=100;

Программа сразу становится масштабируемой. А объявление N как константы гарантирует, что объявленное значение не будет случайно изменено где-то в программе.

Аналогичный результат можно получить, если заменить объявление константы директивой компилятора #define

#define N 10

В ряде случаев требуются константные массивы, данные из которых программа может только читать. Такие массивы обязательно должны инициализироваться в момент объявления. Например, требуется неизменяемый массив простых чисел, значения которых не больше 30:

const primeQ = {2,3,5,7,11,13,17,19,23,29};

Двумерные массивы

По аналогии с одномерными массивами, можно объявлять и многомерные статические массивы, т.е. массивы, элементами которых являются массивы. Например, двумерный массив можно объявить следующим образом:

int A2 [10] [3];

Такое объявление описывает двумерный массив, который можно представить себе как матрицу, состоящую из 10 строк и 3 столбцов.

6.УКАЗАТЕЛИ И ДИНАМИЧЕСКОЕ РАСПРЕДЕЛЕНИЕ ПАМЯТИ

УКАЗАТЕЛИ, ССЫЛКИ И АДРЕСНАЯ АРИФМЕТИКА

Указатели

Указатель - это переменная, значение которой равно значению адреса памяти, по которому лежит значение некоторой другой переменной. В этом смысле имя этой другой переменной отсылает к ее значению прямо, а указатель — косвенно. Ссылка на значение посредством указателя называется косвенной адресацией.

Указатели в языке Си делятся на два вида: указатели на объекты и указатели на функции. Свойства и правила их использования различны.

Указатели, подобно любым другим переменным, перед своим использованием должны быть объявлены. Объявление указателя на объект имеет вид:

тип *имя_указателя;

где **тип** - один из предопределенных или определенных пользователем типов, а **имя_указателя** - указатель.

Например,

int *APtr;

объявляет переменную **APtr** типа **int** * (т.е указатель на целое число) и читается следующем образом: “ **APtr** является указателем на объект целочисленного типа”. Каждая переменная, объявляемая как указатель, должна иметь перед собой символ (*), который обозначает операцию косвенной адресации.

При объявлении указателя возможна его инициализация. Имеется две формы инициализации указателя:

1) **тип *имя_указателя=инициализирующее_значение**
2) **тип *имя_указателя (инициализирующее_значение)**

В качестве **инициализирующего_значения** может использоваться:

- явно заданный участок памяти

double *cc=0x1047;

- указатель, уже имеющий значение

float *ca;

float *cb=ca;

- выражение, позволяющее получить адрес объекта с помощью операции ссылки (&) **int a;**

int *ce=&a;

Указатели должны инициализироваться либо при своем объявлении, либо с помощью оператора присваивания. Использовать указатель без присвоения ему какого-нибудь участка памяти нельзя. Указатель может получить в качестве начального значения **0** или **NULL.** Указатель с начальным значением **0** или **NULL** ни на что не указывает. **NULL** - это символическая константа, определенная специально для цели показать, что данный указатель ни на что не указывает. Пример объявления указателя о его инициализацией:

int *countPtr = NULL;

Для присваивания указателю адреса некоторой переменной используется операция адресации & (операция ссылки), которая возвращает адрес своего операнда. Например, если имеются объявления

int y = 5; int *yPtr, x;

то оператор yPtr = &y;

присваивает адрес переменной y указателю yPtr.

Присвоив указателю адрес конкретного участка памяти, можно с помощью операции разыменования не только получать, но и изменять содержимое этого участка памяти. Для этого используется операция разыменования * (операция косвенной адресации). Она возвращает значение объекта, на который указывает ее операнд (т.е. указатель).

Если присвоить указателю адрес конкретного объекта или значение уже инициализированного указателя, то это превратит запись ***имя_указателя** в синоним уже имеющегося имени объекта. Запись,

double z; double *A=&z;

double *C,*D;

C=&z;

D=A;

превратит *A, *C, *D в синонимы переменной z. Изменяя значение, лежащее по адресу на которое указывают указатели, автоматически изменяется значение переменной, так как указатели A, C, D и переменная z связаны между собой одним участком памяти. Продолжая начатый пример

z=6;

double A1, C1, D1;

A1 =*A;

C1=*C;

D1=*D;

присвоит переменным A1, C1, D1 значение 6, т.е. значение переменной Z, на которую указывают указатели A, C, D.

ОПЕРАТОРЫ ДИНАМИЧЕСКОГО РАСПРЕДЕЛЕНИЯ ПАМЯТИ.

Операции new и delete

Чтобы связать неинициализированный участок памяти, еще не занятым никаким объектом программы, используется оператор new:

указатель=пе«г тип (инициализирующее_значение);

Операция возвращает указатель на динамически размещенный в памяти объект, т.е. предоставляет указателю память под объект заданного типа. Пример

double *A=new double; int *B;

B=new int;

объявляет указатели и выделяет им свободное место, не занятое никаким объектом.

Инициализирующее_значение задает начальные значения создаваемого объекта. Запись

double *C=new double (-1.8);

предоставляет место указателю C под объект вещественного типа и инициализирует его, т.е. записывает начальное значение -1.8.

Если операция new возвратила нулевое значение адреса - NULL, то это значит, что операционная система не может выделить память под данный объект, поэтому после использования оператора new желательно всегда проверять адрес выделенного участка памяти

```
double *A;
   A=new double; if
(A==NULL) ...
```

Динамически распределенную память следует освобождать, когда отпадает необходимость в размещенных в ней объектах. Освобождение памяти осуществляется с помощью операции delete, которая имеет следующий синтаксис

delete имя_указателя;

Пример

```
int *B=new int;
delete B;
```

объявляет указатель, выделяя и освобождая место под объект в адресном пространстве.

Освобождение памяти уничтожает сам объект, а не указатель. В дальнейшем, указателю можно заново выделить участок памяти под новый объект и освободить эту память, что позволяет более гибко использовать оперативное адресное пространство компьютера. Поэтому, объекты созданные с помощью операции new называются динамическими.

```
long *P;
   P=new long; delete P;
   P=new long; delete P;
```

Операция delete освобождает память, но сама не задает указателю значение NULL, поэтому во избежание использования в дальнейшем неинициализированного указателя желательно это делать программно.

```
delete P;
P=NULL;
```

Операции над указателями

Указатели могут применяться как операнды в арифметических выражениях, выражениях присваивания и выражениях сравнения. Однако, не все операции, обычно используемые в этих выражениях, разрешены применительно к переменным указателям.

С указателями может выполняться ограниченное количество арифметических операций. Указатель можно увеличивать (++), уменьшать (--), складывать с указателем целые числа (+ или +=), вычитать из него целые числа (- или -=) или вычитать один указатель из другого.

Сложение указателей с целыми числами отличается от обычной арифметики. Прибавить к указателю 1 означает сдвинуть его на число байтов, содержащихся в переменной, на которую он указывал. Обычно подобные операции применяются к указателям на массивы. Например, запись

```
int *Pt;
Pt+=2;
```

увеличит значение Pt (т.е. адрес в памяти, на который указывает Pt), не на два, а на четыре байта, так как один объект целочисленного типа int в памяти занимает два байта.

Аналогичные правила действуют и при вычитании из указателя целого значения.

Однако при использовании арифметических операций над указателями нельзя

полагать, чтоб две переменные - указатели одинакового типа будут находится в памяти вплотную друг к другу, если только они не соседствуют в массиве.

Сравнение указателей операциями >, <, >=, <= также имеют смысл только для указателей на один и тот же массив. Однако, операции отношения == и != имеют смысл для любых указателей. При этом указатели равны, если они указывают на один и тот же адрес в памяти.

Указатель можно присваивать другому указателю, если оба указателя имеют одинаковый тип. В противном случае нужно использовать операцию приведения типа, чтобы преобразовать значение указателя в правой части присваивания к типу указателя в левой части присваивания. Исключением из этого правила является указатель на void (т.е. void*), который является общим указателем, способным представлять указатели любого типа. Указателю на void можно присваивать все типы указателей без приведения типа. Однако указатель на void не может быть присвоен непосредственно указателю другого типа - указатель на void сначала должен быть приведен к типу соответствующего указателя.

Ссылки

Ссылки - это специальный тип указателя, который позволяет работать с указателем как с объектом. Объявление ссылки делается с помощью операции ссылки, обозначаемой амперсантом (&) - тем же символом, который используется для адресации. Например, если имеется объявление

```
double*P = new double;
```

то можно создать ссылку на этот объект оператором:

```
double& Ref = *P;
```

Объявленная таким образом переменная Ref является ссылкой на объект вещественного типа. Она может рассматриваться как псевдоним объекта. Эта переменная является именно указателем, а не самим объектом. Но работа с ней производится как с объектом, в отличие от указателя.

```
*P=*P+0.5;
Ref=Ref+0.5;
```

Чаще всего ссылки используются при передаче в функции параметров по ссылке.

ДИНАМИЧЕСКОЕ РАЗМЕЩЕНИЕ МАССИВОВ

Связь массивов и указателей

Массивы и указатели Си++ тесно связаны и могут быть использованы почти эквивалентно. При определении массива, имя массива является указателем константой, значением которой служит адрес первого элемента массива (с индексом 0), а запись

```
имя_массива[индекс]
```

является выражением с двумя операндами. Первый из них, т.е. имя_массива - константный указатель - адрес начала массива в основной памяти, а индекс - это выражение целого типа, определяющее смещение от начала массива.

Используя операцию обращения по адресу * (операция разыменования), действие бинарной операции можно представить следующим образом:

```
*(имя_массива + индекс)
```

т.е. операндами для операции [] служат имя_массива и индекс. Из этого становится очевидным, почему индекс первого элемента массива равен нулю. Таким образом, запись *имя_массива - обращение к первому элементу массива, *(имя_массива+1) - обращение ко второму, и т.д.

Поскольку сложение *(имя_массива+индекс) коммутативно, то возможна такая эквивалентная запись *(индекс+имя_массива) и, следовательно, индекс [имя_массива] именует тот же элемент массива, что имя_массива [индекс].

В некоторых конструкциях можно использовать выражение имя_массива [индекс] с отрицательным значением индекса. В этом случае имя_массива должен указывать не на начало массива, т.е. не на его нулевой элемент.

Одномерные динамические массивы

Такая операция позволяет выделить в динамической памяти участок для размещения массива соответствующего типа, но не позволяет его инициализировать. В результате выполнения этой операции возвращается адрес первого элемента массива.

При выделении динамической памяти размер массива должен быть определен явно, т.к. операция new получает информацию о выделяемом участке из размерности и типе_массива. Оператор

double *A=new double [30];

создает динамический массив из 30 элементов вещественного типа.

В отличие от объявления статических массивов, при выделении динамической памяти под указатель в качестве размерности могут использоваться переменные целочисленного типа, что позволяет задавать количество элементов массива программно.

```
float *e; int N;
cout«“N=”; cin »N; e=new
float [N];
```

Доступ к значениям элементов динамического массива может быть осуществлен таким же образом, как и при работе со статическими массивами. Продолжая пример выше, можно записать

```
for(int i=0; i<N; i++)
e[i]=double(random(41 )-20)/10;
```

Изменять значение указателя на динамический массив нужно с осторожностью, так как указатель, значение которого определяется при выделении памяти, используется затем для освобождения этой памяти.

Освободить память, выделенную под динамический массив можно используя операцию delete, которая в этом случае имеет следующий синтаксис

delete □ имя_указателя;

Таким образом, оператор delete

```
[]e;
```

освободит целиком всю память, выделенную для определенного выше массива, если указатель e адресует его начало.

Массивы указателей или многомерные динамические массивы

Аналогично одномерным массивам, можно с помощью указателя и операции new выделить место под двумерный динамический массив. Для этого используются массивы указателей, объявление которых имеет следующий вид;

тип_массива ** имя_указателя;

Данная запись может быть интерпретирована как объявление указателя на объект типа тип_массива *.

Выделение участка памяти указателю на объект типа тип_массива * с помощью операции new позволяет получить массив нужного размера, элементами которого являются также указатели, только на объекты типа тип_массива.

Например

double **B;

B=new double* [20];

выделяет место под массив указателей из 20 элементов.

Так как, каждый элемент массива указателей тоже является указателем, то ему соответственно также можно выделить некоторый участок памяти в виде массива, только уже вещественного типа

B[3]=new double [10];

Доступ к элементам созданного массива осуществляется через индекс элемента массива указателей и индекс массива, на который указывает элемент - указатель, а именно

B[3][5]=10;

Данная запись обращения к элементам аналогична доступу к элементам в двумерном статическом массиве.

Приведенный ниже пример создает двумерный динамический массив из *n* - строк и *m* - столбцов, введенных пользователем

```
int n,m; double **matr;
cout«“n=”; cin»n; cout«“m=”;
cin»m;
//выделение места под массив указателей из //п
элементов, т.е. выделение места под п строк matr=new
double*[n];
//выделение места каждому указателю из массива //указателей по m вещественных элементов,
//т.е. выделение места под m столбцов for (int
i=0; i<n; i++)
   matr[i]=new double[m];
```

Количество элементов, выделяемых всем указателям из массива указателей, не обязательно должно быть одинаковым. Пример

```
int п;
   double **matr; cout«“n=”; cin»n;
   matr=new double*[n]; for (int i=0; i<n; i++)
      matr[i]=new double[i+1];
```

реализует двумерный динамический массив в виде “лесенки со ступеньками”, т.е. первый указатель matr[0] будет содержать массив только из одного элемента, указатель matr[1] будет содержать массив из двух элементов, а последний указатель matr[n-1] будет содержать *n* элементов.

Освобождение памяти, выделенное под двумерный динамический массив, осуществляется в обратном порядке по отношению к выделению памяти, т.е. вначале освобождается память, выделенная под каждый элемент - указатель, а затем освобождается память, выделенная под массив указателей. Следующая запись демонстрирует освобождение памяти, выделенное в предыдущем примере

```
for (int i=0; i<n; i++) delete [] matr[i]; delete Qmatr;
```

ТИПЫ СТРОК В ЯЗЫКЕ С++

В отличие от других языков программирования, стандарт языка Си не содержит специального типа строк. Строки представляются одномерными массивами символов, т.е. массивами элементов типа char. Но если длина массивов чисел имеет определенный значимый характер, то длина строк постоянно меняется и не всегда, получается, иметь информацию об ее длине. Поэтому, для определения длины строки был введен специальный нулевой символ '\0’, который обозначает конец строки в массиве в не зависимости от начальной выделенной для него памяти. В этом случае, число хранимых символов в строке всегда на 1 больше числа значащих символов. Если нулевой символ отсутствует, то обработка строки будет продолжаться до тех пор, пока в памяти не встретится случайный нулевой символ.

Таким образом, строка в языке Си рассматривается, как массив символов, оканчивающийся нулевым символом ('\0'), в не зависимости от длины.

Строка может быть объявлена либо как просто массив символов

```
char St[] = “строка”;
```

либо как переменная указатель типа char*

```
char *Sp = “строка";
```

Каждое из этих приведенных эквивалентных объявлений присваивает строковой переменной начальное значение “строка”, а длина при этом определяется автоматически компилятором и равна 7 элементам, содержащих соответственно символы: 'о’, ‘т’, ‘р’, ‘о’, ’к’, ‘а’ и ’\0'. Можно объявлять строковые переменные заданной длины. Например,

операторы char Str[100]; char *Stp=new char [100];

объявляют переменные, которые могут содержать строки до 99 значащих символов плюс заключительный нулевой символ.

Доступ к отдельным символам строки осуществляется по индексам, начинающимся с нуля. Например, St[0] и Sp[0] - первые символы объявленных выше строк, St[5] и Sp[5] - последние, а St[6] и Sp[6] содержат нулевые символы.

Так как, строка представляется массивом, то доступ к ней осуществляется через указатель или указатель - константу, который является адресом ее первого символа, поэтому при работе со строками необходимо учитывать общие правила работы с указателями и с массивами, так запись

Str=”nepeMeHHan“;

или

Str=St;

вызовет ошибку компилятора, потому что, во первых - нельзя присваивать значение указателю - константе, а во вторых - St, Str, Sp и Stp содержат лишь адрес на участок памяти по которому располагается набор символов, а не сам этот набор. Присваивание литералы - константы возможно лишь только при объявлении строки.

Так же, следует помнить, что резервирование памяти полностью лежит на программисте, а при работе с указателем на строку этот указатель должен указывать на выделенный ранее участок памяти.

Ввод - вывод массивов символов

Имеется несколько возможностей для ввода и вывода строк, рассмотрим только основные из

них.

С помощью форматируемого ввода-вывода. Форматируемый ввод и вывод можно осуществить с помощью функций scant и printf из стандартной библиотеки stdio.h, которые были рассмотрены в лабораторной работе №1. Эти функции имеют существенный недостаток, так как останавливают ввод и вывод строки, не только если встретится нулевой символ, но и если встретится пробел. Поэтому являются не удобными при использовании.

С помощью неформатируемого ввода-вывода. Такой ввод и вывод осуществляется функциями gets и puts без указания формата из библиотеки stdio.h и имеют следующие прототипы:

char *gets(char ’string); int

*puts(char ‘string);

Функция gets осуществляет ввод набора символов с клавиатуры в строку string, до тех пор, пока не будет нажата клавиша Enter. Возвращает адрес начала введенной строки или значение NULL, если произошла ошибка при вводе. Следует помнить, что функция gets не осуществляет выделение памяти под введенный набор символов и не отслеживает размер зарезервированной под него памяти.

Функция puts выводит строку string на монитор и переводит курсор на новую строку. Возвращает последний выводимый символ, которым обычно является символ новой строки '\n'.

Следующий пример иллюстрирует работу данных функций

```
#include <stdio.h>
#include <conio.h> main()
{
char A[100];
char *B=new char [100];
char C[]="Ввод строки:";
char 0[]="Вывод строки:";
puts(C);
```

```
gets(A);
puts(C);
gets(B);
puts(D);
puts (A);
puts(D);
puts(B);
getch();
}
```

С *помощью потокового ввода-вывода.* Наиболее распространенным способом ввода и вывода массивов символов является потоковый ввод-вывод, осуществляемый стандартной библиотекой классов, подключаемой к программе с помощью заголовочного файла iostream.h. Некоторые элементы потокового ввода - вывода были рассмотрены в лабораторной работе №1.

Если необходимо прочитать из входного потока (с клавиатуры) строку символов, содержащую пробелы, то можно воспользоваться перегруженными компонентными функциями get и getline объекта cin, которые имеют следующие прототипы istream& get(char *string, int maxjen, char='\n'); istream& getline(char'string, int maxjen, char='\n');

Каждая из этих функций выполняет чтение последовательности символов с клавиатуры и перенос их в символьный массив string, задаваемый первым параметром. Второй параметр maxjen определяет максимально допустимое количество вводимых символов. Третий параметр определяет ограничивающий символ при появлении, которого следует завершить ввод. Второй и третий параметры могут быть опущены, в этом случае размер вводимых символов не ограничен, а конец ввода будет осуществляться при нажатии клавиши Enter. Если из входного потока извлечены ровно maxjen - 1 символов, а ограничивающий символ не встретился, то нулевой символ помещается после введенной последовательности автоматически, при этом массив, в который выполняется чтение, должен иметь длину не менее maxjen символов. Различие между get и getline заключается в том, что в последней функции в конец строки помещается ограничивающий символ, который прервал ввод.

Потоковый вывод массива символов можно осуществить стандартным методом, а именно с помощью стандартного потока вывода на экран cout« string;

Ниже представлен пример потокового ввода - вывода строк, подобный примеру, который был представлен выше с помощью функций gets и puts #include <iostream.h>

```
#include <conio.h> main()
{
char A[100]; char *B=new char
[100]; char C[]="Ввод строки:"; char
D[]="BbiBOfl строки:"; cout« C;
cin.getline(A, 100); cout« C;
cin.getline(B,100); cout « D « endl;
cout « A « endl; cout« D « endl;
cout« B « endl; getch();}
```

Также библиотека классов iostream.h содержит множество других удобных и интересных способов ввода и вывода, рассмотрении которых лежит вне данного учебного пособия.

7.ФУНКЦИИ ЯЗЫКА ПРОГРАММИРОВАНИЯ Си++
ОБЪЯВЛЕНИЕ И ОПИСАНИЕ ФУНКЦИЙ

Вызов функций

Функция вызывается при вычислении выражений. При вызове ей передаются определенные аргументы, функция выполняет необходимые действия и возвращает результат.

Программа на языке Си++ состоит, по крайней мере, из одной функции - функции main. С нее всегда начинается выполнение программы. Встретив имя функции в выражении, программа вызовет эту функцию, т.е. передаст управление на ее начало и начнет выполнять операторы. Достигнув конца функции или оператора return - выхода из функции, управление вернется в ту точку, откуда функция была вызвана, подставив вместо нее вычисленный результат.

Прежде всего, функцию необходимо объявить. Объявление функции, аналогично объявлению переменной, определяет имя функции и ее тип - типы и количество ее аргументов и тип возвращаемого значения.

```
    // функция sqrt с одним аргументом - //
вещественным числом двойной точности,
    // возвращает результат типа double
    double sqrt(double x);
    // функция sum от трех целых аргументов //
возвращает целое число int sum(int a, int b, int c);
```

Объявление функции называют иногда прототипом функции. После того, как функция объявлена, ее можно использовать в выражениях: double x = sqrt(3) + 1; sum(k, 1, m) /15;

Если функция не возвращает никакого результата, т.е. она объявлена как void, ее вызов не может быть использован как операнд более сложного выражения, а должен быть записан сам по

себе:

```
func(a,b,c);
```

Определение функции описывает, как она работает, т.е. какие действия надо выполнить, чтобы получить искомый результат. Для функции sum, объявленной выше, определение может выглядеть следующим образом: int

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

Первая строка - это заголовок функции, он совпадает с объявлением функции, за исключением того, что объявление заканчивается точкой с запятой. Далее в фигурных скобках заключено тело функции - действия, которые данная функция выполняет.

Аргументы a, b и c называются формальными параметрами. Это переменные, которые определены в теле функции (т.е. к ним можно обращаться только внутри фигурных скобок). При написании определения функции программа не знает их значения. При вызове функции вместо них подставляются фактические параметры - значения, с которыми функция вызывается. Выше, в примере вызова функции sum, фактическими параметрами (или фактическими аргументами) являлись значения переменных k, 1 и т.

Формальные параметры принимают значения фактических аргументов, заданных при вызове, и функция выполняется.

Первое, что мы делаем в теле функции — объявляем внутреннюю переменную result типа целое. Переменные, объявленные в теле функции, также называют локальными. Это связано с тем, что переменная result существует только во время

выполнения тела функции sum. После завершения выполнения функции она уничтожается - ее имя становится неизвестным, и память, занимаемая этой переменной, освобождается.

Вторая строка определения тела функции - вычисление результата. Сумма всех аргументов присваивается переменной result. Отметим, что до присваивания значение result было неопределенным (то есть значение переменной было неким произвольным числом, которое нельзя определить заранее).

Последняя строчка функции возвращает в качестве результата вычисленное значение. Оператор return завершает выполнение функции и возвращает выражение, записанное после ключевого слова return, в качестве выходного значения. В следующем фрагменте программы переменной s присваивается значение 10: int к = 2; int 1 = 3; int m = 5;

```
int s = sum(k, 1, m);
```

Имена функций

В языке Си++ допустимо иметь несколько функций с одним и тем же именем, потому что функции различаются не только по именам, но и по типам аргументов. Если в дополнение к определенной выше функции sum мы определим еще одну функцию с тем же именем double

```
sum(double a, double b, double c)
{
    double result; result = a + b + c;
return result;
}
```

это будет считаться новой функцией. Иногда говорят, что у этих функций разные подписи. В следующем фрагменте программы в первый раз будет вызвана первая функция, а во второй раз - вторая:

```
int x, y, z, ires; double p,q,s,
dres;
// вызвать первое определение функции sum
ires = sum(x,y,z);
// вызвать второе определение функции sum
dres = sum(p,q,s);
```

При первом вызове функции sum все фактические аргументы имеют тип int. Поэтому вызывается первая функция. Во втором вызове все аргументы имеют тип double, соответственно, вызывается вторая функция.

Важен не только тип аргументов, но и их количество. Можно определить функцию sum, суммирующую четыре аргумента: int

```
sum(int xl, int x2, int x3, int x4)
{
    return xl + x2 + x3 + x4;
}
```

Отметим, что при определении функций имеют значение тип и количество аргументов, но не тип возвращаемого значения. Попытка определения двух функций с одним и тем же именем, одними и теми же аргументами, но разными возвращаемыми значениями, приведет к ошибке компиляции:

```
int foo(int x); double foo(int
x);
// ошибка - двукратное определение имени
```

Необязательные аргументы функций

При объявлении функций в языке Си++ имеется возможность задать значения аргументов по умолчанию. Первый случай применения этой возможности языка - сокращение записи. Если функция вызывается с одним и тем же значением аргумента в

99% случаев, и это значение достаточно очевидно, можно задать его по умолчанию. Предположим, функция expnt возводит число в произвольную целую положительную степень. Чаще всего она используется для возведения в квадрат. Ее объявление можно записать так: double expnt (double x, unsigned int e = 2);

Определение функции: double
expnt (double x, unsigned int e)
{
double result = 1; for (int i = 0; i
< e; i++) result *= x; return result;
}
int main()
{
double y = expnt(3.14); double x =
expnt(2.9, 5); return 1;
}

Использовать аргументы по умолчанию удобно при изменении функции. Если при изменении программы нужно добавить новый аргумент, то для того чтобы не изменять все вызовы этой функции, можно новый аргумент объявить со значением по умолчанию. В таком случае старые вызовы будут использовать значение по умолчанию, а новые - значения, указанные при вызове.

Необязательных аргументов может быть несколько. Если указан один необязательный аргумент, то либо он должен быть последним в прототипе, либо все аргументы после него должны также иметь значение по умолчанию.

Если для функции задан необязательный аргумент, то фактически задано несколько подписей этой функции. Например, попытка определения двух функций double expnt (double x, unsigned int e = 2); double expnt (double x);

приведет к ошибке компиляции - неоднозначности определения функции. Это происходит потому, что вызов

double x = expnt(4.1);

подходит как для первой, так и для второй функции.

ФУНКЦИИ С ПЕРЕМЕННЫМ КОЛИЧЕСТВОМ ПАРАМЕТРОВ

Рассмотрим подкласс алгоритмов, которые, хотя и не всегда дают оптимальное решение, тем не менее широко используются на практике.

Определение 2.7

Эвристический алгоритм —- основанный на опыте или неких интуитивных предположениях метод решения задачи, дающий, как правило, хороший результат.

Данное определение эвристического алгоритма содержит много неконкретных понятий и в этом заключается особенность эвристики. Эвристическими могут называться и алгоритмы, которые не могут быть полностью проанализированы, не доказана их сложность или оптимальность, но из опыта или из общих рассуждений и предположений известно, что они дают хорошее приближение к требуемому результату. Что считать хорошим результатом, зависит от конкретной задачи, условий и ограничений.

Подразумевается, что эвристический алгоритм обладает небольшой, реализуемой сложностью, и это компенсирует возможную неточность результата. К примеру, для задачи коммивояжера, которую рассматривали в подразд. 2.1, предложен способ решения «полным перебором», имеющий сложность порядка 0(п!). С помощью этого метода всегда находим оптимальный путь, платя за это большой трудоемкостью, которая явно нереализуема уже на величинах *п* порядка нескольких десятков.

В то же время легко указать эвристический способ нахождения маршрута, не обязательно оптимального (можно построить примеры, когда этот способ не будет

находить кратчайший объезд), но, как правило, достаточно короткого. Этот способ называется «правилом ближайшего соседа» и, как следует из названия, заключается в том, чтобы из каждого города все время выезжать в ближайший непосещенный город. Такие методы называются иногда «жадными», так как они всегда выбирают локальный оптимум, стараясь достичь глобального. Очевидно, сложность правила ближайшего соседа составляет $0(n^2)$ (для каждого из *n* городов нужно выбрать выезд минимальной стоимости из п — 1 направлений), что значительно меньше, чем сложность полного перебора. Для этого примера метод ближайшего соседа дает объезд *ABCDA* со стоимостью 15. т.е. в данном случае полученный объезд совпадает с оптимальным. Однако можно привести примеры, в которых применение данного метода будет неоптимально. Для следующей задачи также может быть получен простой эвристический алгоритм.

Задача 2.15(задача о расписании процессоров). Пусть нам даны *n* параллельных (одинаковых по производительности) процессоров *P\, P2,...Pn,* и *m* заданий *1\,Д,...,1m,* где время выполнения каждого задания на любом из процессоров известно и составляет Ti*, To,...,Tm.* Требуется составить расписание, распределяющее задания по процессорам так, чтобы время выполнения всех заданий было минимальным.

Решение. Можно легко видеть, что различные расписания приводят к различному времени выполнения всех задач. Рассмотрим пример для трех процессоров *Г*, РЬ, *P;* и четырех заданий *1\, 1%* /3,14 со временем выполнения *Т* = 10, *Т%* = 8, *Т%* = 2, Т4 = 3. Предположим, что задания стоят в очереди, при этом освободившийся процессор берет первое задание в очереди для обработки. Если одновременно свободны несколько процессоров, они берут задания в порядке своих номеров.

Таким образом, расписание может быть задано просто последовательностью заданий Ij_1,, Ij_m, и, следовательно, можно задать *ml* различных расписаний. Например, подавая задания в порядке /4, /3, *I2, h,* т.е. первый процессор берет четвертое задание, второй — третье и т.д., получим время выполнения, равное 12 (рис. 2.17, *а)*. Очевидно, такое расписание неоптимально: пока одним процессором обрабатывается задание со временем выполнения 10, остальные два успеют обработать оставшиеся задания (рис. 2.17, *б)*. Расписание, изображенное на рис. 2.17. *б,* является оптимальным, так как время обработки всех заданий не может быть меньше, чем время обработки самого трудоемкого задания. Расписание на рис. 2.17, *б* дает именно такое время, т.е. - 10.

Однако для перебора всех возможных расписаний, как и в задаче коммивояжера, требуется решать задачу экспоненциальной сложности. Эвристический подход, который сформулируем, весьма прост: будем считать расписанием последовательность 1_{71}, /;-2, ..., lj_m такую, что $Tj_1 < Tj_2 < \ldots < Tj_m$, отсортированную по

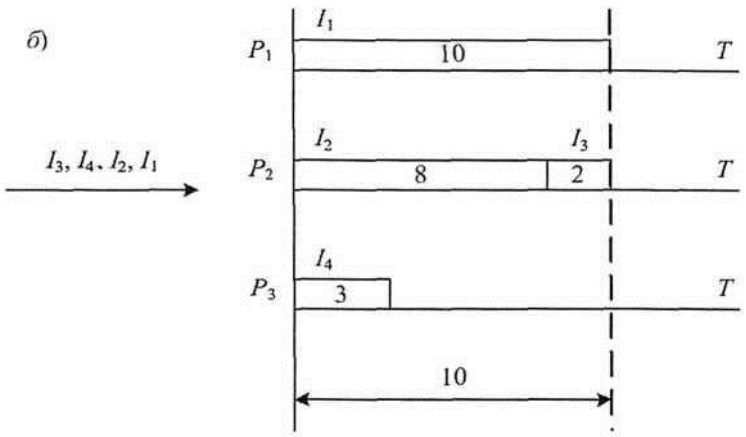


Рис. 2.17. Задача о расписании процессоров: а - произвольное расписание; б - оптимальное расписание

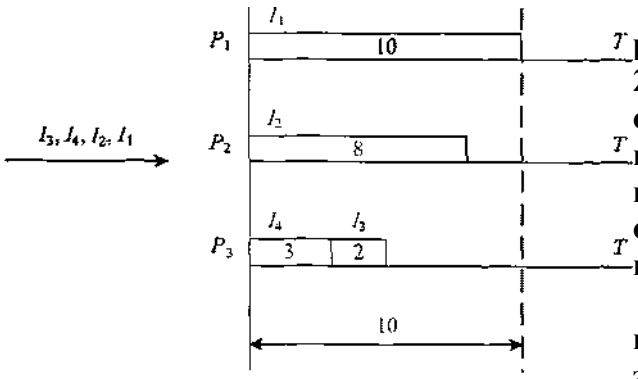


Рис. 2.18. Эвристическое расписание

времени обработки. Такое расписание изображено на рис. 2.18. В данном случае время обработки расписания, полученного эвристическим путем, снова совпадает с оптимальным и дает общее время обработки — 10.

Однако легко указать пример, для которого эвристическое расписание не даст оптимального решения. Такой пример приведен на рис. 2.19.

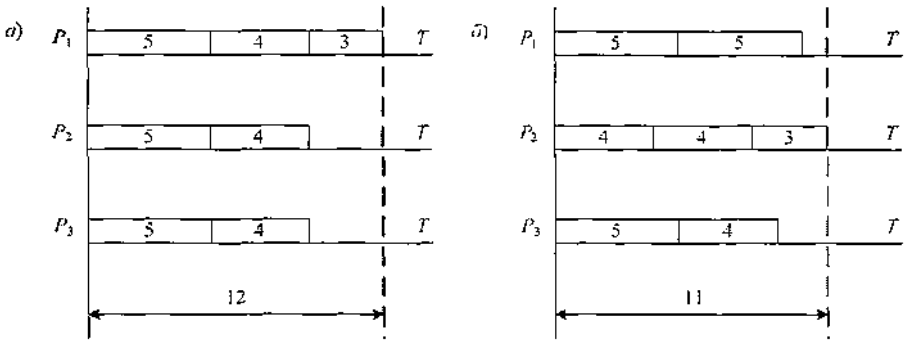


Рис. 2.19. Проигрыш эвристического алгоритма: а - эвристическое расписание; б - оптимальное расписание

Можно доказать, что если обозначить через То оптимальное время обработки, а через $T_э$ — полученное с помощью описанного

Таким образом, не только сформулирован эвристический алгоритм, но и в некотором смысле оценена его эффективность относительно оптимального, т.е. получена оценка максимального проигрыша.

РЕКУРСИВНЫЕ ФУНКЦИИ

При создании или описании разного рода алгоритмов часто можно заметить, что решение той или иной задачи может быть выражено как комбинация или модификация решений той же задачи (или другой, но с известным решением), с другими входными данными, другой размерностью, дополнительными условиями и т.п. Выявление подобных закономерностей позволяет либо найти решение данной задачи через решения уже известных задач, либо получить дополнительную информацию о «структуре» решаемой проблемы, ее свойствах, и эта информация может быть использована при разработке алгоритмов решения. Одним из случаев такой закономерности является возможность описать задачу с помощью рекурсивной последовательности действий, или, проще говоря, рекурсии.

Определение 2.4

Рекурсивный алгоритм — решение задачи в ходе выполнения обращающееся само к себе.

Одна и та же задача может быть решена как с применением рекурсии, так и без нее. Вообще, любой рекурсивный алгоритм может быть описан нерекурсивно, но не наоборот. Рекурсия часто является удобным алгоритмическим решением задачи, однако не всегда удачна с точки зрения реализации встроенными методами рекурсии на языке программирования: при рекурсивном вызове функции требуется сохранение значений всех локальных переменных, точки возврата, что может

привести к переполнению выделенной программе памяти даже при формально корректной записи алгоритма и реализации.

Рекурсивный алгоритм обязательно должен содержать две части:

1. **Шаг рекурсии. Указание, каким образом производится рекурсивный вызов.**
2. **База рекурсии. Условие выхода из рекурсии.**

Распространенной ошибкой является отсутствие базы рекурсии — это приводит к зацикливанию алгоритма или его некорректному завершению. Далее рассмотрим несколько задач, которые могут быть решены с помощью рекурсии.

ПЕРЕДАЧА ПАРАМЕТРОВ В ФУНКЦИЮ ПО ССЫЛКЕ

Каждая программа на С и С++ должна иметь функцию main; причем ваше дело, где вы ее поместите. Некоторые программисты помещают ее в начале файла, некоторые в конце. Однако независимо от ее положения необходимо помнить следующее.

Запускающая процедура Borland C++ посылает функции main три параметра (аргумента): argc, argv и env.

- argc, целое, - это число аргументов командной строки, посылаемое функции main,
- argv это массив указателей на строки (char * []).

J Под версией DOS 3.x и более поздними argv[0] определяется как полный маршрут запускаемой программы.

J При работе под более ранними версиями DOS argv указывает на нулевую строку (""). argv[l] указывает на первую после имени программы строку командной строки.

J argv[2] указывает на вторую после имени программы строку командной строки.

J argv[argc-l] указывает на последний аргумент, посылаемый функции main.

J argv[argc] содержит NULL.

- env также является массивом указателей на строки. Каждый элемент env[] содержит строку вида ENVVARr^HaneHHe.

J ENVVAR - это имя переменной среды, типа PATH или 87.

Заметим, однако, что если вы описываете некоторые из этих аргументов, то вы должны описывать их в таком порядке: argc, argv, env. Например, допустимы следующие объявления аргументов:

```
main()
main(int argc) /* допустимо но не очень хорошо */
main(int argc, char *argvQ)
main(int argc, char *argv[], char *env[])
```

Объявление main(int argc) не очень удобно тем, что зная количество параметров, вы не имеете доступа к ним самим. Аргумент env всегда доступен через глобальную переменную environ. Смотрите описание переменной environ и функции putenv и getenv.

Параметры **argc** и **argv** также доступны через **nepeMeHHbie_argc** и **_argv.**

Пример программы, использующей argc, argv и env.

Это пример программы ARGS.EXE, которая демонстрирует простейший путь использования аргументов, посылаемых функции main.

```
/* программа ARGS.C */
ffinclude <stdio.h>
#include <stdlib.h>
void main(int argc, char *argv[|, char *env[|)
```

```
{
int i;
рптЮЗначение argc равно %d \n\n",argc);
printf("В командной строке содержится %d параметров \n\n",argc);
                    ПЕРЕДАЧА ПАРАМЕТРОВ В ФУНКЦИЮ ПО ССЫЛКЕ
 for (i=0; i<=argc; i++) printf(" argv[%d]:
%s\n",i,argv[i]);
 printf("Среда содержит следующие строки:\n”);
 for (i=0; env[i] != NULL; i++) printf(" env[%d]:
%s\n",i,env[i]);
 return 0;
 }
```

Предположим, что вы запускаете программу ARGS.EXE со следующей командной строкой:

```
C:>args first_arg "arg with blanks" 3 4 "last but one" stop!
```

Заметим, что вы можете послать аргумент с пробелами, заключив его в двойные кавычки, как показано на примере "argument with blanks" и "last but one" в примере вызова программы.

В результате работы программы вы получите примерно следующее:

```
Значение argc равно 7
    В командной строке содержится 7 параметров argv[0]: c:\turboc\testargs.exe
argv[l]: first_arg argv[2]: arg with blank argv[3]: 3 argv [4]: 4
    argv[5]: last but one argv[6]: stop!
Среда содержит следующие строки:
    env[0]: COMSPEC=C:\COMMAND.COM
    env[l]: PROMPT=$p $g
    env[2]: PATH=C:\SPRINT;C:\DOS;C:\BC .
```

Максимальная общая длина командной строки, посылаемая функции main (включая пробелы и имя самой программы), не може превышать 128 символов; это ограничения DOS.

Аргументы командной строки могут содержать символы маскирования. При этом они могут расширяться для всех имен файлов, которые совпадают с аргументом так, как это делается, например, с командой DOS copy. Для использования символов маскирования необходимо при связывании вашей программы редактором связей подсоединить к ней объектный файл WILDARGS.OBJ, который поставляется с Borland C++.

Если файл WILDARGS.OBJподсоединен к вашей программе, то вы можете в командной строке использовать аргументы типа При этом имена всех файлов, подходящих к данной маске, заносятся в массив argv. Максимальный размер массива argv зависит только от объема динамической области памяти.

Если под данную маску не нашлось подходящих файлов, то аргумент передается в том виде, в каком он был набран в командной отроке.

Аргументы, заключенные в двойные кавычки ("..."), нрасширяются.

Пример. Следующие команды компилируют файл ARGS.C и связывают его с модулем WILDARGS.OBJ, а затем запускают получившуюся программу ARGS.EXE:

```
bcc args wildarg.obj
args C:\BORLANDC\INCLUDE\*.H "*.C"
```

При запуске ARGS.EXE первый аргумент расширяется до имен всех файлов с расширением H в директории Borland C++INCLUDE.OTMeraM, что все строки включают полный маршрут (к примеру C:\TC\INCLUDE\ALLOC.H). Аргумент *.C не расширяется, т.к. он заключен в кавычки.

Если вы работаете в Интегрированном Окружении (BC.EXE), то вам просто

нужно указать в меню проекта имя файла проекта, который должен содержать следующие строки:

ARGS

W1LDARGS.OBJ

Затем с помощью команд "Run/Arguments" следует установить параметры командной строки.

8.СТАТИЧЕСКИЕ И ДИНАМИЧЕСКИЕ СТРУКТУРЫ ДАННЫХ

СТАТИЧЕСКИЕ СТРУКТУРЫ ДАННЫХ

Статические структуры относятся к разряду непримитивных структур, которые, фактически, представляют собой структурированное множество примитивных, базовых, структур. Например, вектор может быть представлен упорядоченным множеством чисел. Поскольку по определению статические структуры отличаются отсутствием изменчивости, память для них выделяется автоматически - как прави- ло, на этапе компиляции или при выполнении - в момент активизации того программного блока, в котором они описаны. Ряд языков программирования (PL/1, ALGOL-60) допускают размещение статических структур в памяти на этапе выполнения по явному требованию программиста, но и в этом случае объем выделенной памяти остается неизменным до уничтожения структуры. Выделение памяти на этапе компиляции является столь удобным свойством статических структур, что в ряде задач программисты используют их даже для представления объектов, обладающих изменчивостью. Например, когда размер массива неизвестен заранее, для него резервируется максимально возможный размер.

Каждую структуру данных характеризуют еЯ логическим и физическим представлениями. Очень часто говоря о той или иной структуре данных, имеют в виду еЯ логическое представление. Физическое представление обычно не соответствует логическому, и кроме того, может существенно различаться в разных программных системах. Нередко физической структуре ставится в соответствие дескриптор, или заголовок, который содержит общие сведения о физической структуре. Дескриптор необходим, например, в том случае, когда граничные значения индексов элементов массива неизвестны на этапе компиляции, и, следовательно, выделение памяти для массива может быть выполнено только на этапе выполнения программы (как в языке PL/1, ALGOL-60). Дескриптор хранится как и сама физическая структура, в памяти и состоит из полей, характер, число и размеры которых зависят от той структуры, которую он описывает и от приня- тых способов ее обработки. В ряде случаев дескриптор является совершенно необходимым, так как выполнение операции доступа к структуре требует обязательного знания каких-то ее параметров, и эти параметры хранятся в дескрипторе. Другие хранимые в дескрипторе параметры не являются совершенно необходимыми, но их использование позволяет сократить время доступа или обеспечить контроль правильности доступа к структуре. Дескриптор структуры данных, поддерживаемый языками программирования, является "невидимым" для программиста; он создается компилятором и компилятор же, формируя объектные коды для доступа к структуре, включает в эти коды команды, обращающиеся к дескриптору.

Статические структуры в языках программирования связаны со структурированными типами. Структурированные типы в языках программирования являются теми средствами интеграции, которые позволяют строить структуры данных сколь угодно большой сложности. К таким типам относятся: массивы, записи (в некоторых языках - структуры) и множества (этот тип реализован не во всех языках).

Векторы

Вектор (одномерный массив) - структура данных с фиксированным числом элементов одного и того же типа типа. Каждый элемент вектора имеет уникальный в рамках заданного вектора номер. Обращение к элементу вектора выполняется по имени вектора и номеру требуемого элемента.

Элементы вектора размещаются в памяти в подряд расположенных ячейках памяти. Под элемент вектора выделяется количество байт памяти, определяемое базовым типом элемента этого вектора. Необходимое число байтов памяти для хранения одного элемента вектора называется слотом. Размер памяти для хранения вектора определяется произведением длины слота на число элементов.

где @ Имя -адрес вектора или, что тоже самое, адрес первого элемента вектора, Sizeof(Tnn)-ра3Мер слота (количество байтов памяти для записи одного элемента вектора), (к- n)*Sizeof(THn) - относительный адрес элемента с номером к, или, что тоже самое, смещение элемента с номером к.

Например:

var ml:array[-2..2] of real; представление данного вектора в памяти

В языках, где память распределяется до выполнения программы на этапе компиляции (C, PASCAL, FORTRAN), при описании типа вектора граничные значения индексов должны определены. В языках, где память может распределяться динамически (ALGOL, PL/1), значения индексов могут быть заданы во время выполнения программы.

Количество байтов непрерывной области памяти, занятых одновременно вектором, определяется по формуле:

ByteSise = (к - n + 1) * Sizeof (тип)

Обращение к i-тому элементу вектора выполняется по адресу вектора плюс смещение к данному элементу. Смещение i-oro элемента вектора определяется по формуле:

ByteNumer = (i- n) * Sizeof (тип), а адрес его: @ ByteNumber = @ имя + ByteNumber. где @ имя - адрес первого элемента вектора.

При доступе к вектору задается имя вектора и номер элемента вектора. Таким образом, адрес i-ro элемента может быть вычислен как:

@Имя|1] = @Имя + i*Sizeof(Tnn) - n*Sizeof(mn) (3.1)

Это вычисление не может быть выполнено на этапе компиляции, так как значение переменной i в это время еще неизвестно. Следовательно, вычисление адреса элемента должно производиться на этапе выполнения программы при каждом обращении к элементу вектора. Но для этого на этапе выполнения, во-первых, должны быть известны параметры формулы (3.1): @Имя Sizeof(™n), п, а во-вторых, при каждом обращении должны выполняться две операции умножения и две - сложения. Преобразовав формулу (3.1) в формулу (3.2),

@Имя[1] = AO + i*Sizeof(nm) — (3.2)

AO = @Имя - n*Sizeof(THn) - сократим число хранимых параметров до двух, а число операций - до одного умножения и одного сложения, так как значение AO может быть вычислено на этапе компиляции и сохранено вместе с Sizeoffnm) в дескрипторе вектора. Обычно в дескрипторе вектора сохраняются и граничные значения индексов. При каждом обращении к элементу вектора заданное значение сравнивается с граничными и программа аварийно завершается, если заданный индекс выходит за допустимые пределы.

Таким образом, информация, содержащаяся в дескрипторе вектора, позволяет, во-первых, сократить время доступа, а во-вторых, обеспечивает проверку правильности обращения. Но за эти преимущества приходится платить, во-первых, быстродействием, так как обращения к дескриптору - это команды, во-вторых, памятью как для размещения самого дескриптора, так и команд, с ним работающих.

Можно ли обойтись без дескриптора вектора?

В языке С, например, дескриптор вектора отсутствует, точнее, не сохраняется на этапе выполнения. Индексация массивов в С обязательно начинается с нуля. Компилятор каждое обращение к элементу массива заменяет на последовательность команд,

реализующую частный случай формулы (3.1) при п = 0:

Смещение элемента относит. адреса m1 (байт)	+0	+6	+12	+18	+24
Значения элем. массива	m1[-2]	m1[-1]	m1[0]	m1[1]	m1[2]

@Имя[1] = @Имя + i*Sizeof(THn)

Программисты, привыкшие работать на С, часто вместо выражения вида: ИмяЩ употребляют выражение вида: *(Имя+1).

Но во-первых, ограничение в выборе начального индекса само по себе может являться неудобством для программиста, во-вторых, отсутствие граничных значений индексов делает невозможным контроль выхода за пределы массива. Программисты, работающие с С, хорошо знают, что именно такие ошибки часто являются причиной "зависания" С-программы при ее отладке.

Представление разреженных матриц методом связанных структур.

Методы последовательного размещения для представления разреженных матриц обычно позволяют быстрее выполнять операции над матрицами и более эффективно использовать память, чем методы со связанными структурами. Однако последовательное представление матриц имеет определенные недостатки. Так включение и исключение новых элементов матрицы вызывает необходимость перемещения большого числа других элементов. Если включение новых элементов и их исключение осуществляется часто, то должен быть выбран описываемый ниже ме- тод связанных структур.

Метод связанных структур, однако, переводит представляемую структуру данных в другой раздел классификации. При том, что логическая структура данных остается статической, физическая структура становится динамической.

Для представления разреженных матриц требуется базовая структура вершины (рис.3.6), называемая MATRIX_ELEMENT ("элемент матрицы"). Поля V, R и C каждой из этих вершин содержат соответственно значение, индексы строки и столбца элемента матрицы. Поля LEFT и UP являются указателями на следующий элемент для строки и столбца в циклическом списке, содержащем элементы матрицы. Поле LEFT указывает на вершину со следующим наименьшим номером строки.

На рис. 3.7 приведена многосвязная структура, в которой используются вершины такого типа для представления матрицы A, описанной ранее в данном пункте. Циклический список представляет все строки и столбцы. Список столбца может содержать общие вершины с одним списком строки или более. Для того, чтобы обеспечить использование более эффективных алгоритмов включения и исключения элементов, все списки строк и столбцов имеют головные вершины. Головная вершина каждого списка строки содержит нуль в поле C; аналогично каждая головная вершина в списке столбца имеет нуль в поле R. Строка или столбец, содержащие только нулевые элементы, представлены головными вершинами, у которых поле LEFT или UP указывает само на себя.

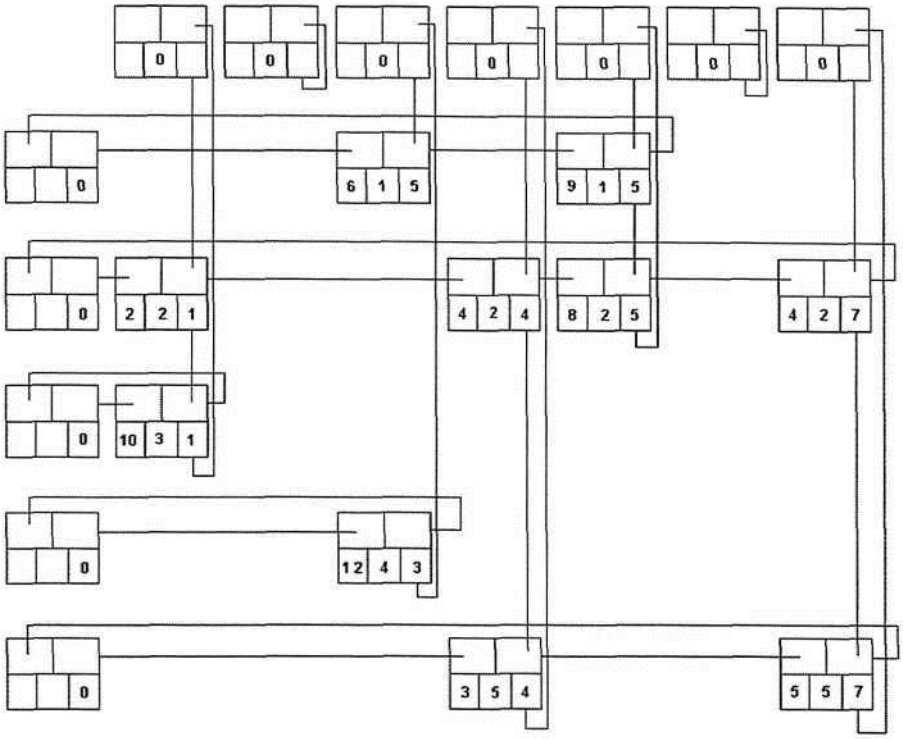


Рис. 3.7. Многосвязная структура для нредставления матрицы А

Может показаться странным, что указатели в этой многосвязной структуре направлены вверх и влево, вследствие чего при сканировании циклического списка элементы матрицы встречаются в порядке убывания номеров строк и столбцов. Такой метод представления используется для упрощения включения новых вершин в структуру. Предполагается, что новые вершины, которые должны быть добавлены к матрице, обычно располагаются в порядке убывания индексов строк и индексов столбцов. Если это так, то новая вершина всегда добавляется после головной и не требуется никакого просмотра списка.

Множества

Логическая структура.

Множество - такая структура, которая представляет собой набор неповторяющихся данных одного и того же типа. Множество может принимать все значения базового типа. Базовый тип не должен превышать 256 возможных значений. Поэтому базовым типом множества мотут быть byte, char и производные от них типы.

Физическая структура.

Множество в памяти хранится как массив битов, в котором каждый бит указывает является ли элемент принадлежащим объявленному множеству или нет. Т.о. максимальное число элементов множества 256, а данные типа множество могут занимать не более 32-ух байт.

Число байтов, выделяемых для данных типа множество, вычисляется по формуле: ByteSize = (max div 8)-(min div 8) + 1, где max и min - верхняя и нижняя границы базового типа данного множества.

Номер байта для конкретного элемента E вычисляется по формуле:

ByteNumber = (E div 8)-(min div 8),

Стандартный числовой тип, который может быть базовым для формирования множества - тип byte.

где @S - адрес данного типа множество.

ПОЛУСТАТИЧЕСКИЕ СТРУКТУРЫ ДАННЫХ

Характерные особенности полустатических структур

Полустатические структуры данных характеризуются следующими признаками:

- они имеют переменную длину и простые процедуры ее изменения;
- изменение длины структуры происходит в определенных пределах, не превышая какого-то максимального (предельного) значения.

Если полустатическую структуру рассматривать на логическом уровне, то о ней можно сказать, что это последовательность данных, связанная отношениями линейного списка. Доступ к элементу может осуществляться по его порядковому номеру.

Физическое представление полустатических структур данных в памяти - это обычно последовательность слотов в памяти, где каждый следующий элемент расположен в памяти в следующем слоте (т.е. вектор). Физическое представление может иметь также вид однонаправленного связного списка (цепочки), где каждый следующий элемент адресуется указателем находящемся в текущем элементе. В пос- леднем случае ограничения на длину структуры гораздо менее строгие.

Стеки

Логическая структура стека

Стек - такой последовательный список с переменной длиной, включение и исключение элементов из которого выполняются только с одной стороны списка, называемого вершиной стека. Применяются и другие названия стека - магазин и очередь, функционирующая по принципу L1FO (Last - In - First- Out - "последним пришел - первым исключается"). Примеры стека: винтовочный патронный магазин, тупиковый железнодорожный разъезд для сортировки вагонов.

Основные операции над стеком - включение нового элемента (английское название push - заталкивать) и исключение элемента из стека (англ. pop - выскакивать).

Полезными могут быть также вспомогательные операции:

- определение текущего числа элементов в стеке;
- очистка стека;
- неразрушающее чтение элемента из вершины стека, которое может быть реализовано, как комбинация основных операций:

pop(stack); push(stack,x).

Некоторые авторы рассматривают также операции включения/исключения элементов для середины стека, однако структура, для которой возможны такие операции, не соответствует стеку по определению.

Для наглядности рассмотрим небольшой пример, демонстрирующий принцип включения элементов в стек и исключения элементов из стека. На рис. 4.1 (а,б,с) изображены состояния стека:

- а), пустого;
- б-г). после последовательного включения в него элементов с именами 'A', 'B', 'C[1];
- д, е). после последовательного удаления из стека элементов 'C' и 'B';
- ж), после включения в стек элемента 'D'.

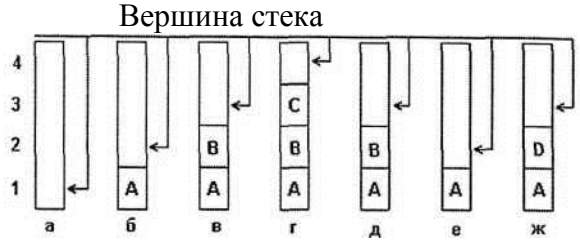


Рис 4.1. Включение и исключение элементов из стека.

Как видно из рис. 4.1, стек можно представить, например, в виде стопки книг (элементов), лежащей на столе. Присвоим каждой книге свое название, например A,B,C,D... Тогда в момент времени, когда на столе книг нет, про стек аналогично можно сказать, что он пуст, т.е. не содержит ни одного элемента. Если же мы начнем

последовательно класть книги одну на другую, то получим стопку книг (допустим, из п книг), или получим стек, в котором содержится п элементов, причем вершиной его будет являться элемент п+1. Удаление элементов из стека осуществляется аналогичным образом т. е. удаляется последовательно по одному элементу, начиная с вершины, или по одной книге из стопки.

Машинное представление стека и реализация операций

При представлении стека в статической памяти для стека выделяется память, как для вектора. В дескрипторе этого вектора кроме обычных для вектора параметров должен находиться также указатель стека - адрес вершины стека. Указатель стека может указывать либо на первый свободный элемент стека, либо на последний записанный в стек элемент. (Все равно, какой из этих двух вариантов выбрать, важно в последствии строго придерживаться его при обработке стека.) В дальнейшем мы будем всегда считать, что указатель стека адресует первый свободный элемент и стек растет в сторону увеличения адресов.

При занесении элемента в стек элемент записывается на место, определяемое указателем стека, затем указатель модифицируется таким образом, чтобы он указывал на следующий свободный элемент (если указатель указывает на последний записанный элемент, то сначала модифицируется указатель, а затем производится запись элемента). Модификация указателя состоит в прибавлении к нему или в вычитании из него единицы (помните, что наш стек растет в сторону увеличения адресов.

Операция исключения элемента состоит в модификации указателя стека (в направлении, обратном модификации при включении) и выборке значения, на которое указывает указатель стека. После выборки слот, в котором размещался выбранный элемент, считается свободным.

Операция очистки стека сводится к записи в указатель стека начального значения - адреса начала выделенной области памяти.

Определение размера стека сводится к вычислению разности указателей: указателя стека и адреса начала области.

Стеки в вычислительных системах

Стек является чрезвычайно удобной структурой данных для многих задач вычислительной техники. Наиболее типичной из таких задач является обеспечение вложенных вызовов процедур.

Предположим, имеется процедура А, которая вызывает процедуру В, а та в свою очередь - процедуру С. Когда выполнение процедуры А дойдет до вызова В, процедура А приостанавливается и управление передается на входную точку процедуры В. Когда В доходит до вызова С, приостанавливается В и управление передается на процедуру С. Когда заканчивается выполнение процедуры С, управление должно быть возвращено в В, причем в точку, следующую за вызовом

С. При завершении В управление должно возвращаться в А, в точку, следующую за вызовом В. Правильную последовательность возвратов легко обеспечить, если при каждом вызове процедуры записывать адрес возврата в стек. Так, когда процедура А вызывает процедуру В, в стек заносится адрес возврата в А; когда В вызывает С, в стек заносится адрес возврата в В. Когда С заканчивается, адрес возврата выбирается из вершины стека - а это адрес возврата в В. Когда заканчивается В, в вершине стека находится адрес возврата в А, и возврат из В произойдет в А.

В микропроцессорах семейства Intel, как и в большинстве современных процессорных архитектур, поддерживается аппаратный стек. Аппаратный стек расположен в ОЗУ, указатель стека содержится в паре специальных регистров - SS:SP, доступных для программиста. Аппаратный стек расширяется в сторону уменьшения адресов, указатель его адресует первый свободный элемент. Выполнение команд CALL и INT, а также аппаратных прерываний включает в себя запись в аппаратный стек адреса возврата. Выполнение команд RET и IRET включает в себя выборку из аппаратного стека

адреса возврата и передачу управления по этому адресу. Пара команд - PUSH и POP - обеспечивает использование аппаратного стека для программного решения других задач.

Системы программирования для блочно-ориентированных языков (PASCAL, C и др.) используют стек для размещения в нем локальных переменных процедур и иных программных блоков. При каждой активизации процедуры память для ее локальных переменных выделяется в стеке; при завершении процедуры эта память освобождается. Поскольку при вызовах процедур всегда строго соблюдается вложенность, то в вершине стека всегда находится память, содержащая локальные переменные активной в данный момент процедуры.

Этот прием делает возможной легкую реализацию рекурсивных процедур. Когда процедура вызывает сама себя, то для всех ее локальных переменных выделяется новая память в стеке, и вложенный вызов работает с собственным представлением локальных переменных. Когда вложенный вызов завершается, занимаемая его переменными область памяти в стеке освобождается и актуальным становится представление локальных переменных предыдущего уровня. За счет этого в языках PASCAL и C любые процедуры/функции могут вызывать сами себя. В языке PL/1, где по умолчанию приняты другие способы размещения локальных переменных, рекурсивная процедура должна быть определена с описателем RECURSIVE - только тогда ее локальные переменные будут размешаться в стеке.

Рекурсия использует стек в скрытом от программиста виде, но все рекурсивные процедуры могут быть реализованы и без рекурсии, но с явным использованием стека. В программном примере 3.17 приведена реализация быстрой сортировки Хоара в рекурсивной процедуре.

Один проход сортировки Хоара разбивает исходное множество на два множества. Границы полученных множеств запоминаются в стеке. Затем из стека выбираются границы, находящиеся в вершине, и обрабатывается множество, определяемое этими границами. В процессе его обработки в стек может быть записана новая пара границ и т.д. При начальных установках в стек заносятся границы исходного множества. Сортировка заканчивается с опустошением стека.

Очереди FIFO

Логическая структура очереди

Очередью FIFO (First - In - First- Out - "первым пришел - первым исключается"), называется такой последовательный список с переменной длиной, в котором включение элементов выполняется только с одной стороны списка (эту сторону часто называют концом или хвостом очереди), а исключение - с другой стороны (называемой началом или головой очереди). Те самые очереди к прилавкам и к кассам, которые мы так не любим, являются типичным бытовым примером очереди FIFO.

Основные операции над очередью - те же, что и над стеком - включение, исключение, определение размера, очистка, неразрушающее чтение.

Машинное представление очереди FIFO и реализация операций

При представлении очереди вектором в статической памяти в дополнение к обычным для дескриптора вектора параметрам в нем должны находиться два указателя: на начало очереди (на первый элемент в очереди) и на ее конец (первый свободный элемент в очереди). При включении элемента в очередь элемент записывается по адресу, определяемому указателем на конец, после чего этот указатель увеличивается на единицу. При исключении элемента из очереди выбирается элемент, адресуемый указателем на начало, после чего этот указатель уменьшается на единицу.

Очевидно, что со временем указатель на конец при очередном включении элемента достигнет верхней границы той области памяти, которая выделена для очереди. Однако, если операции включения чередовались с операциями исключения элементов, то в начальной части отведенной под очередь памяти имеется свободное место. Для того, чтобы места, занимаемые исключенными элементами, могли быть повторно

использованы, очередь замыкается в кольцо: указатели (на начало и на конец), достигнув конца выделенной области памяти, переключаются на ее начало. Такая организация очереди в памяти называется кольцевой очередью. Возможны, конечно, и другие варианты организации: например, всякий раз, когда указатель конца достигнет верхней границы памяти, сдвигать все непустые элементы очереди к началу области памяти, но как этот, так и другие варианты требуют перемещения в памяти элементов очереди и менее эффективны, чем кольцевая очередь.

В исходном состоянии указатели на начало и на конец указывают на начало области памяти. Равенство этих двух указателей (при любом их значении) является признаком пустой очереди. Если в процессе работы с кольцевой очередью число операций включения превышает число операций исключения, то может возникнуть ситуация, в которой указатель конца "догонит" указатель начала. Это ситуация заполненной очереди, но если в этой ситуации указатели сравняются, эта ситуация будет неотличима от ситуации пустой очереди. Для различения этих двух ситуаций к кольцевой очереди предъявляется требование, чтобы между указателем конца и указателем начала оставался "зазор" из свободных элементов. Когда этот "зазор" сокращается до одного элемента, очередь считается заполненной и дальнейшие попытки записи в нее блокируются. Очистка очереди сводится к записи одного и того же (не обязательно начального) значения в оба указателя. Определение размера состоит в вычислении разности указателей с учетом кольцевой природы очереди.

Очереди с приоритетами

В реальных задачах иногда возникает необходимость в формировании очередей, отличных от FIFO или LIFO. Порядок выборки элементов из таких очередей определяется приоритетами элементов. Приоритет в общем случае может быть представлен числовым значением, которое вычисляется либо на основании значений каких-либо полей элемента, либо на основании внешних факторов. Так, и FIFO, и LIFO-очереди могут трактоваться как приоритетные очереди, в которых приоритет элемента зависит от времени его включения в очередь. При выборке элемента всякий раз выбирается элемент с наибольшим приоритетом.

Очереди с приоритетами могут быть реализованы на линейных списковых структурах - в смежном или связном представлении. Возможны очереди с приоритетным включением - в которых последовательность элементов очереди все время поддерживается упорядоченной, т.е. каждый новый элемент включается на то место в последовательности, которое определяется его приоритетом, а при исключении всегда выбирается элемент из начала. Возможны и очереди с приоритетным исключением - новый элемент включается всегда в конец очереди, а при исключении в очереди ищется (этот поиск может быть только линейным) элемент с максимальным приоритетом и после выборки удаляется из последовательности. И в том, и в другом варианте требуется поиск, а если очередь размещается в статической памяти - еще и перемещение элементов.

Наиболее удобной формой для организации больших очередей с приоритетами является сортировка элементов по убыванию приоритетов частично упорядоченным деревом, рассмотренная нами в п.3.9.2.

Очереди в вычислительных системах

Идеальным примером кольцевой очереди в вычислительной системы является буфер клавиатуры в Базовой Системе Ввода-Вывода ПЭВМ IBM PC. Буфер клавиатуры занимает последовательность байтов памяти по адресам от 40:1E до 40:2D включительно. По адресам 40:1A и 40:1C располагаются указатели на начало и конец очереди соответственно. При нажатии на любую клавишу генерируется прерывание 9. Обработчик этого прерывания читает код нажатой клавиши и помещает его в буфер клавиатуры - в конец очереди. Коды нажатых клавиш могут накапливаться в буфере клавиатуры, прежде чем они будут прочитаны программой. Программа при вводе данные с клавиатуры обращается к прерыванию 16H. Обработчик этого прерывания выбирает

код клавиши из буфера - из начала очереди - и передает в программу.

Очередь является одним из ключевых понятий в многозадачных операционных системах (Windows NT, Unix, OS/2, EC и др.). Ресурсы вычислительной системы (процессор, оперативная память, внешние устройства и т.п.(используются всеми задачами, одновременно выполняемыми в среде такой операционной системы. Поскольку многие виды ресурсов не допускают реально одновременного использования разными задачами, такие ресурсы предоставляются задачам поочередно. Таким образом, задачи, претендующие на использование того или иного ресурса, выстраиваются в очередь к этому ресурсу. Эти очереди обычно приоритетные, однако, довольно часто применяются и FIFO-очереди, так как это единственная логическая организация очереди, которая гарантированно не допускает постоянного вытеснения задачи более приоритетными. LIFO- очереди обычно используются операционными системами для учета свободных ресурсов.

Также в современных операционных системах одним из средств взаимодействия между параллельно выполняемыми задачами являются очереди сообщений, называемые также почтовыми ящиками. Каждая задача имеет свою очередь - почтовый ящик, и все сообщения, отправляемые ей от других задач, попадают в эту очередь. Задача-владелец очереди выбирает из нее сообщения, причем может управлять порядком выборки - FIFO, LIFO или по приоритету.

Деки

Логическая структура дека

Дек - особый вид очереди. Дек (от англ. deq - double ended queue,т.е очередь с двумя концами) - это такой последовательный список, в котором как включение, так и исключение элементов может осуществляться с любого из двух концов списка. Частный случай дека - дек с ограниченным входом и дек с ограниченным выходом. Логическая и физическая структуры дека аналогичны логической и физической структуре кольцевой FIFO-очереди. Однако, применительно к деку целесообразно говорить не о начале и конце, а о левом и правом конце.

Операции над деком:

- включение элемента справа;
- включение элемента слева;
- исключение элемента справа;
- исключение элемента слева;
- определение размера;
- очистка.

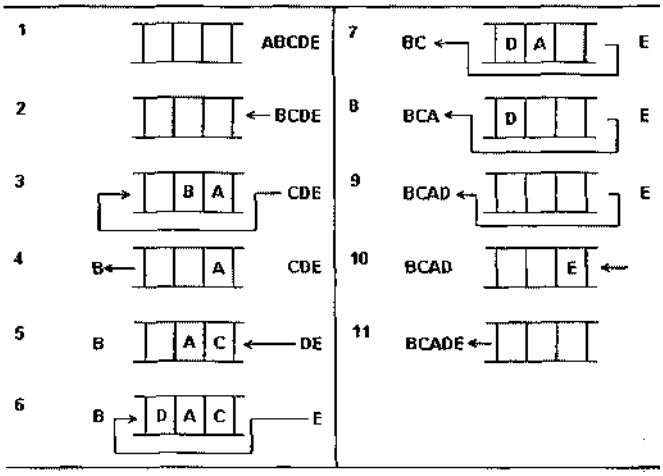


Рис. 4.2. Состояния дека в процессе изменения.

На рис. 4.2 в качестве примера показана последовательность состояний дека при включении и удалении пяти элементов. На каждом этапе стрелка указывает с какого конца дека (левого или правого) осуществляется включение или исключение элемента.

Элементы соответственно обозначены буквами A, B, C, D, E.

Физическая структура дека в статической памяти идентична структуре кольцевой очереди.

Деки в вычислительных системах

Задачи, требующие структуры дека, встречаются в вычислительной технике и программировании гораздо реже, чем задачи, реализуемые на структуре стека или очереди. Как правило, вся организация дека выполняется программистом без каких-либо специальных средств системной поддержки.

Однако, в качестве примера такой системной поддержки рассмотрим организацию буфера ввода в языке REXX. В обычном режиме буфер ввода связан с клавиатурой и работает как FIFO- очередь. Однако, в REXX имеется возможность назначить в качестве буфера ввода программный буфер и направить в него вывод программ и системных утилит. В распоряжении программиста имеются операции QUEUE - запись строки в конец буфера и PULL - выборка строки из начала буфера. Дополнительная операция PUSH - запись строки в начало буфера - превращает буфер в дек с ограниченным выходом. Такая структура буфера ввода позволяет программировать на REXX весьма гибкую конвейерную обработку с направлением выхода одной программы на вход другой и модификацией перенаправляемого потока.

ДИНАМИЧЕСКИЕ СТРУКТУРЫ ДАННЫХ

Связное представление данных в памяти

Динамические структуры по определению характеризуются отсутствием физической смежности элементов структуры в памяти непостоянством и непредсказуемостью размера (числа элементов) структуры в процессе ее обработки. В этом разделе рассмотрены особенности динамических структур, определяемые их первым характерным свойством. Особенности, связанные со вторым свойством рассматриваются в последнем разделе данной главы.

Поскольку элементы динамической структуры располагаются по непредсказуемым адресам памяти, адрес элемента такой структуры не может быть вычислен из адреса начального или предыдущего элемента. Для установления связи между элементами динамической структуры используются указатели, через которые устанавливаются явные связи между элементами. Такое представление данных в памяти называется связным. Элемент динамической структуры состоит из двух полей:

- информационного поля или поля данных, в котором содержатся те данные, ради которых и создается структура; в общем случае информационное поле само является интегрированной структурой - вектором, массивом, записью и т.п.;
- поле связок, в котором содержатся один или несколько указателей, связывающий данный элемент с другими элементами структуры;

Когда связное представление данных используется для решения прикладной задачи, для конечного пользователя "видимым" делается только содержимое информационного поля, а поле связок используется только программистом-разработчиком.

Достоинства связного представления данных - в возможности обеспечения значительной изменчивости структур;

- размер структуры ограничивается только доступным объемом машинной памяти;
- при изменении логической последовательности элементов структуры требуется не перемещение данных в памяти, а только коррекция указателей.

Вместе с тем связное представление не лишено и недостатков, основные из которых:

- работа с указателями требует, как правило, более высокой квалификации от

программиста;

- на поля связок расходуется дополнительная память;
- доступ к элементам связной структуры может быть менее эффективным по времени.

Последний недостаток является наиболее серьезным и именно им ограничивается применимость связного представления данных. Если в смежном представлении данных для вычисления адреса любого элемента нам во всех случаях достаточно было номера элемента и информации, содержащейся в дескрипторе структуры, то для связного представления адрес элемента не может быть вычислен из исходных данных. Дескриптор связной структуры содержит один или несколько указателей, позволяющих войти в структуру, далее поиск требуемого элемента выполняется следованием по цепочке указателей от элемента к элементу. Поэтому связное представление практически никогда не применяется в задачах, где логическая структура данных имеет вид вектора или массива - с доступом по номеру элемента, но часто применяется в задачах, где логическая структура требует другой исходной информации доступа (таблицы, списки, деревья и т.д.).

Связные линейные списки

Списком называется упорядоченное множество, состоящее из переменного числа элементов, к которым применимы операции включения, исключения. Список, отражающий отношения соседства между элементами, называется линейным. Если ограничения на длину списка не допускаются, то список представляется в памяти в виде связной структуры. Линейные связные списки являются простейшими динамическими структурами данных.

Графически связи в списках удобно изображать с помощью стрелок. Если компонента не связана ни с какой другой, то в поле указателя записывают значение, не указывающее ни на какой элемент. Такая ссылка обозначается специальным именем - nil.

Машинное представление связных линейных списков

На рис. 5.1 приведена структура односвязного списка. На нем поле INF - информационное поле, данные, NEXT - указатель на следующий элемент списка. Каждый список должен иметь особый элемент, называемый указателем начала списка или головой списка, который обычно по формату отличен от остальных элементов. В поле указателя последнего элемента списка находится специальный признак nil, свидетельствующий о конце списка.

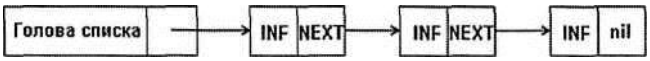


Рис.5.1. Структура односвязного списка

Однако, обработка односвязного списка не всегда удобна, так как отсутствует возможность продвижения в противоположную сторону. Такую возможность обеспечивает двухсвязный список, каждый элемент которого содержит два указателя: на следующий и предыдущий элементы списка. Структура линейного двухсвязного списка приведена на рис. 5.2, где поле NEXT - указатель на следующий элемент, поле PREV - указатель на предыдущий элемент. В крайних элементах соответствующие указатели должны содержать nil, как и показано на рис. 5.2.

Для удобства обработки списка добавляют еще один особый элемент - указатель конца списка. Наличие двух указателей в каждом элементе усложняет список и приводит к дополнительным затратам памяти, но в то же время обеспечивает более эффективное выполнение некоторых операций над списком.

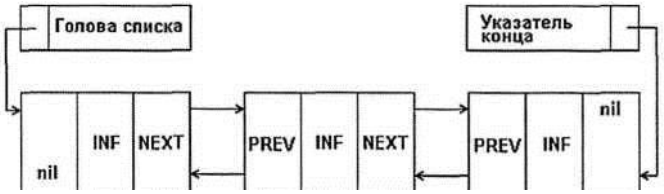


Рис.5.2. Структура двухсвязного списка

Разновидностью рассмотренных видов линейных списков является кольцевой список, который может быть организован на основе как односвязного, так и двухсвязного списков. При этом в односвязном списке указатель последнего элемента должен указывать на первый элемент; в двухсвязном списке в первом и последнем элементах соответствующие указатели переопределяются, как показано на рис.5.3.

При работе с такими списками несколько упрощаются некоторые процедуры, выполняемые над списком. Однако, при просмотре такого списка следует принять некоторых мер предосторожности, чтобы не попасть в бесконечный цикл.

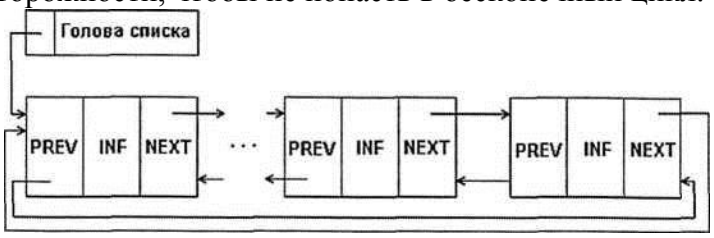


Рис.5.3. Структура кольцевого двухсвязного списка

В памяти список представляет собой совокупность дескриптора и одинаковых по размеру и формату записей, размещенных произвольно в некоторой области памяти и связанных друг с другом в линейно упорядоченную цепочку с помощью указателей. Запись содержит информационные поля и поля указателей на соседние элементы списка, причем некоторыми полями информационной части могут быть указатели на блоки памяти с дополнительной информацией, относящейся к элементу списка. Дескриптор списка реализуется в виде особой записи и содержит такую информацию о списке, как адрес начала списка, код структуры, имя списка, текущее число элементов в списке, описание элемента и т.д., и т.п. Дескриптор может находиться в той же области памяти, в которой располагаются элементы списка, или для него выделяется какое-нибудь другое место.

Реализация операций над связными линейными списками

Ниже рассматриваются некоторые простые операции над линейными списками. Выполнение операций иллюстрируется в общем случае рисунками со схемами изменения связей и программными примерами.

На всех рисунках сплошными линиями показаны связи, имевшиеся до выполнения и сохранившиеся после выполнения операции. Пунктиром показаны связи, установленные при выполнении операции. Значком 'x' отмечены связи, разорванные при выполнении операции. Во всех операциях чрезвычайно важна последовательность коррекции указателей, которая обеспечивает корректное изменение списка, не затрагивающее другие элементы. При неправильном порядке коррекции легко потерять часть списка. Поэтому на рисунках рядом с устанавливаемыми связями в скобках показаны шаги, на которых эти связи устанавливаются.

Перебор элементов списка.

Эта операция, возможно, чаще других выполняется над линейными списками. При ее выполнении осуществляется последовательный доступ к элементам списка - ко всем до конца списка или до нахождения искомого элемента.

Обработка может состоять в:

- печати содержимого инф.части;
- модификации полей инф.части;
- сравнения полей инф.части с образцом при поиске по ключу;
- подсчете итераций цикла при поиске по номеру;
- и т.д., и т.п.}

В двухсвязном списке возможен перебор как в прямом направлении (он выглядит точно так же, как и перебор в односвязном списке), так и в обратном. В последнем случае параметром процедуры должен быть tail - указатель на конец списка, и переход к следующему элементу должен осуществляться по указателю назад:

В кольцевом списке окончание перебора должно происходить не по признаку последнего элемента - такой признак отсутствует, а по достижению элемента, с которого начался перебор. Алгоритмы перебора для двусвязного и кольцевого списка мы оставляем читателю на самостоятельную разработку.

Вставка элемента в список.

Вставка элемента в середину односвязного списка показана на рис.5.4 и в примере 5.2. prev

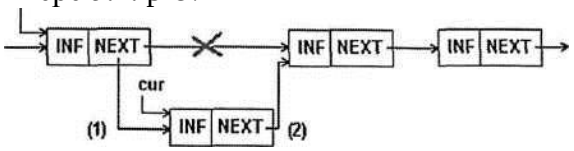


Рис.5.4. Вставка элемента в середину 1-связного списка

Рисунок 5.5 представляет вставку в двухсвязный список.

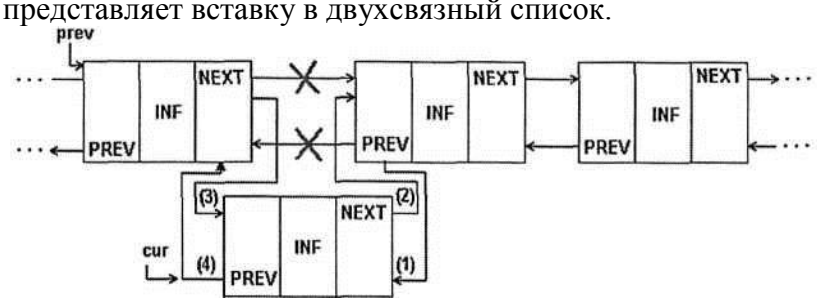


Рис.5.5. Вставка элемента в середину 2-связного списка

Приведенные примеры обеспечивают вставку в середину списка, но не могут быть применены для вставки в начало списка. При последней должен модифицироваться указатель на начало списка, как показано на рис.5.6.

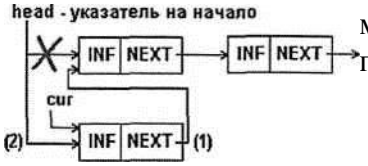
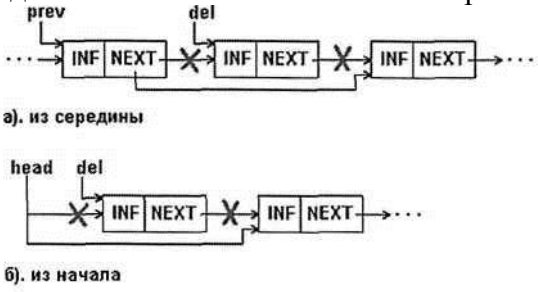


Рис.5.6. Вставка элемента в начало 1-связного списка Удаление элемента из списка.

Удаление элемента из односвязного списка показано на рис.5.7.



Очевидно, что процедуру удаления легко выполнить, если известен адрес элемента, предшествующего удаляемому (prev на рис.5.7.а).

Удаление элемента из двухсвязного списка требует коррекции большего числа указателей, как показано на рис.5.8.

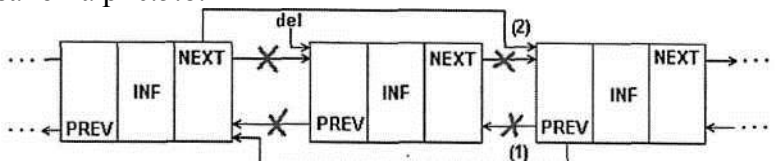


Рис.5.8. Удаление элемента из 2-связного списка

Рис.5.8. Удаление элемента из 2-связного списка

Процедуру удаления элемента из двухсвязного списка окажется даже проще, чем для односвязного, так как в ней не нужен поиск предыдущего элемента, он выбирается по указателю назад.

Перестановка элементов списка.

Изменчивость динамических структур данных предполагает не только изменения размера структуры, но и изменения связей между элементами. Для связных структур изменение связей не требует пересылки данных в памяти, а только изменения указателей в элементах связной структуры. В качестве примера приведена перестановка двух соседних элементов списка. В алгоритме перестановки в односвязном списке (рис.5.9) исходили из того, что известен адрес элемента, предшествующего паре, в которой производится перестановка. В приведенном алгоритме также не учитывается случай перестановки первого и второго элементов.

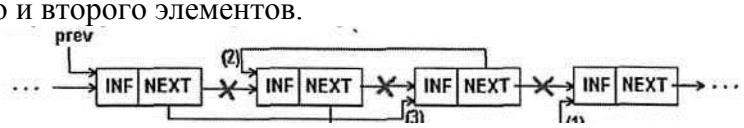


Рис.5.9. Перестановка соседних элементов 1-связного списка

В процедуре перестановки для двухсвязного списка (рис.5.10.) нетрудно учесть и перестановку в начале/конце списка.

Копирование части списка.

При копировании исходный список сохраняется в памяти, и создается новый список. Информационные поля элементов нового списка содержат те же данные, что и в элементах старого списка, но поля связок в новом списке совершенно другие, поскольку элементы нового списка расположены по другим адресам в памяти. Существенно, что операция копирования предполагает дублирование данных в памяти. Если после создания копии будут изменены данные в исходном списке, то изменение не будет отражено в копии и наоборот.

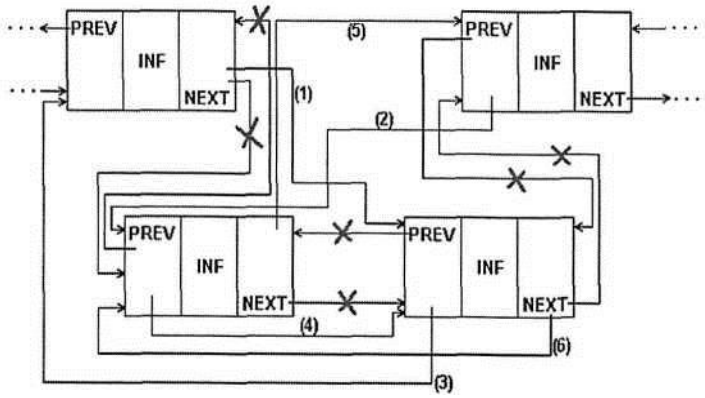


Рис.5.10. Перестановка соседних элементов 2-связного списка

Слияние двух списков.

Операция слияния заключается в формировании из двух списков одного - она аналогична операции сцепления строк.

Применение линейных списков

Линейные списки находят широкое применение в приложениях, где непредсказуемы требования на размер памяти, необходимой для хранения данных; большое число сложных операций над данными, особенно включений и исключений. На базе линейных списков могут строится стеки, очереди и деки. Представление очереди с помощью линейного списка позволяет достаточно просто обеспечить любые желаемые дисциплины обслуживания очереди. Особенно это удобно, когда число элементов в очереди трудно предсказуемо.

Стек представляется как линейный список, в котором включение элементов всегда производятся в начала списка, а исключение - также из начала. Для представления его нам достаточно иметь один указатель - top, который всегда указывает на последний записанный в стек элемент. В исходном состоянии (при пустом стеке) указатель top - пустой. Процедуры StackPush и StackPop сводятся к включению и исключению элемента в начало списка. Обратите внимание, что при включении элемента для него выделяется память, а при исключении - освобождается. Перед включением элемента проверяется доступный объем памяти, и если он не позволяет выделить память для нового элемента, стек считается заполненным. При очистке стека последовательно просматривается весь список и уничтожаются его элементы. При списковом представлении стека оказывается непросто определить размер стека. Эта операция могла бы потребовать перебора всего списка с подсчета числа элементов.

Линейные связные списки иногда используются также для представления таблиц - в тех случаях, когда размер таблицы может существенно изменяться в процессе ее существования. Однако, то обстоятельство, что доступ к элементам связного линейного списка может быть только последовательным, не позволяет применить к такой таблице эффективный двоичный поиск, что существенно ограничивает их применимость. Поскольку упорядоченность такой таблицы не может помочь в организации поиска, задачи сортировки таблиц, представленных линейными связными списками, возникают значительно реже, чем для таблиц в векторном представлении. Однако, в некоторых случаях для таблицы, хотя и не требуется частое выполнение поиска, но задача генерации отчетов требует расположения записей таблицы в некотором порядке. Для упорядочения записей такой таблицы применимы любые алгоритмы из описанных нами в разделе 3.9. Некоторые алгоритмы, возможно, потребуют каких-либо усложнений структуры, например, быструю сортировку Хоара целесообразно проводить только на двухсвязном списке, в цифровой сортировке удобно создавать промежуточные списке для цифровых групп и т.д.

Мультисписки

В программных системах, обрабатывающих объекты сложной структуры, могут решаться разные подзадачи, каждая из которых требует, возможно, обработки не всего множества объектов, а лишь какого-то его подмножества. Так, например, в автоматизированной системе учета лиц, пострадавших вследствие аварии на ЧАЭС, каждая

запись об одном пострадавшем содержит более 50 полей в своей информационной части. Решаемые же автоматизированной системой задачи могут потребовать выборки, например:

- участников ликвидаций аварии;
- переселенцев из зараженной зоны;
- лиц, состоящих на квартирном учете;
- лиц с заболеваниями щитовидной железы;
- и т.д., и т.п.

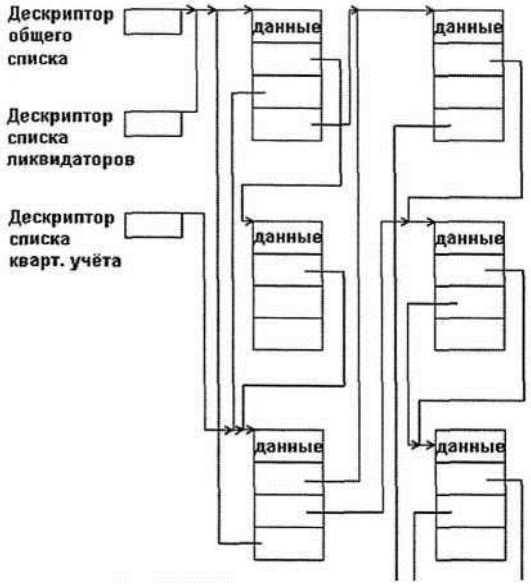


Рис.5.11. Пример мультисписка

Для того, чтобы при выборке каждого подмножества не выполнять полный просмотр с отсеиванием записей, к требуемому подмножеству не относящихся, в каждую запись включаются дополнительные поля ссылок, каждое из которых связывает в линейный список элементы соответствующего подмножества. В результате получается многосвязный список или мультиепиеок, каждый элемент которого может входить одновременно в несколько односвязных списков. Пример такого мультисписка для названной нами автоматизированной системы показан на риК&Мтоинствам мультисписков помимо экономии памяти (при множестве списков информационная часть существует в единственном экземпляре) следует отнести также целостность данных - в том смысле, что все подзадачи работают с одной и той же версией информационной части и изменения в данных, сделанные одной подзадачей немедленно становятся доступными для другой подзадачи.

Каждая подзадача работает со своим подмножеством как с линейным списком, используя для этого определенное поле связок. Специфика мультисписка проявляется только в операции исключения элемента из списка. Исключение элемента из какого-либо одного списка еще не означает необходимости удаления элемента из памяти, так как элемент может оставаться в составе других списков. Память должна освобождаться только в том случае, когда элемент уже не входит ни в один из частных списков мультисписка. Обычно задача удаления упрощается тем, что один из частных списков является главным - в него обязательно входят все имеющиеся элементы. Тогда исключение элемента из любого неглавного списка состоит только в переопределении указателей, но не в освобождении памяти. Исключение же из главного списка требует не только освобождения памяти, но и переопределения указателей как в главном списке, так и во всех неглавных списках, в которые удаляемый элемент входил.

4.3. Лабораторные работы

№ п/п	*Номер раздела дисциплины*	*Наименование лабораторной работы*	*Объем (час.)*	*Вид занятия в интерактивной, активной, инновационной формах, (час.)*
1	**2,3,4.**	Программирование алгоритмов линейной структуры	3	-
2	**2,3,5.**	Программирование алгоритмов разветвляющейся структуры	3	-
3	**2,3,5.**	Программирование алгоритмов циклической структуры	6	-
4	**2,5,6.**	Программирование алгоритмов над одномерными массивами	6	-
5	**2,5,6.**	Программирование алгоритмов над многомерными массивами	6	Работа в малых группах (3 часа)
6	**3,6.**	Программирование алгоритмов над массивами символов	6	-
7	**3,7.**	Программирование алгоритмов при помощи функций	6	Работа в малых группах (3 часа)
		ИТОГО	**36**	**6**

4.4. Семинары/ практические занятия

Учебным планом не предусмотрено

4.5. Контрольные мероприятия: контрольная работа

Методы организации, хранения и обработки линейных двухсвязных списков

Цель: разработка программы, включающая в себя алгоритмы по обработки линейных двухсвязных списков.

Структура: Каждое индивидуальное задание предполагает разработку алгоритмов и программы по созданию: нового списка, добавление элементов в список, вывод списка на дисплей, сохранение данных списка в файл, чтение данных списка из файла, удаление списка из памяти компьютера, поиск элемента в списке, сортировка списка и удаление элемента списка. Разрабатываемая программа должна обеспечивать “диалоговый режим” работы пользователя с программой и содержать информацию о ее назначении (название или краткое описание задачи, на решение которой она направлена).

Основная тематика: Разработка алгоритма и программного обеспечения задачи предметной области.

Рекомендуемый объем: Пояснительная записка объемом 30 – 40 с должна содержать титульный лист, задание, описание разработанных алгоритмов и программы, блок схему и листинг программы.

График контрольных мероприятий

Выдача заданий защита кр проводится в соответствии с календарным учебным графиком

Оценка	**Критерии оценки**
отлично	соответствие требованиям по структурному содержанию и объему работы; правильность выполнения задания, сопровождающегося схемами, таблицами, формулами; самостоятельность выполнения;

	оформление работы и списка использованных источников соответствует требованиям; грамотность, отсутствие стилистических ошибок; уверенное владение материалом при устной защите.
хорошо	соответствие требованиям по структурному содержанию и объему работы; правильность выполнения задания, сопровождающегося схемами, таблицами, переходными характеристиками; самостоятельность выполнения; оформление работы и списка использованных источников не полностью соответствует требованиям; грамотность, отсутствие стилистических ошибок; хорошее владение материалом при устной защите.
удовлетворительно	не полное соответствие требованиям по структурному содержанию и объему работы; неточность выполнения задания, сопровождающегося схемами, таблицами, формулами; частичная самостоятельность выполнения; оформление работы и списка использованных источников не полностью соответствует требованиям; наличие некоторых стилистических ошибок; не уверенное владение материалом при устной защите.
неудовлетворительно	несоответствие требованиям по структурному содержанию и объему работы; неправильность выполнения задания, сопровождающегося схемами, таблицами, формулами; отсутствие самостоятельности выполнения; оформление работы и списка использованных источников не соответствует требованиям; грамотность, наличие стилистических ошибок; отсутствие владения материалом при устной защите.

5. МАТРИЦА СООТНЕСЕНИЯ РАЗДЕЛОВ УЧЕБНОЙ ДИСЦИПЛИНЫ К ФОРМИРУЕМЫМ В НИХ КОМПЕТЕНЦИЯМ И ОЦЕНКЕ РЕЗУЛЬТАТОВ ОСВОЕНИЯ ДИСЦИПЛИНЫ

Компетенции / *№, наименование разделов дисциплины*	*Кол-во часов*	*Компетенции*		*Σ комп.*	$t_{ср}$, *час*	*Вид учебных занятий*	*Оценка результатов*
		ОК	*ОПК*				
		7	*9*				
1	2	3	4	5	6	7	8
1. Инструментарий технологии программирования	9	+	+	2	4,5	Лк, СРС	ЭКЗАМЕН
2. Основные принципы и подходы к разработке программных алгоритмов	18	+	+	2	9	Лк, ЛР, СРС	ЭКЗАМЕН
3. Основы программирования на языке высокого уровня Си++.	17	+	+	2	8,5	Лк, ЛР, СРС	ЭКЗАМЕН
4. Типы данных, выражения и операции в языке программирования с++	11	+	+	2	5,5	Лк, ЛР, СРС	ЭКЗАМЕН
5. Операторы языка программирования Си++ и управление их исполнением	16	+	+	2	8	Лк, ЛР, СРС	ЭКЗАМЕН
6. Указатели и динамическое распределение памяти	16	+	+	2	8	Лк, ЛР, СРС	ЭКЗАМЕН
7. Функции языка программирования Си++	11	+	+	2	5,5	Лк, ЛР, СРС	ЭКЗАМЕН
8. Статические и динамические структуры данных	10	+	+	2	5	Лк, СРС	ЭКЗАМЕН, КР
всего часов	**108**	**54**	**54**	**2**	**54**		

6. ПЕРЕЧЕНЬ УЧЕБНО-МЕТОДИЧЕСКОГО ОБЕСПЕЧЕНИЯ ДЛЯ САМОСТОЯТЕЛЬНОЙ РАБОТЫ ОБУЧАЮЩИХСЯ ПО ДИСЦИПЛИНЕ

1. С++. Объектно-ориентированное программирование. Практикум : учебное пособие для вузов / Т. А. Павловская, Ю. А. Щупак. - Санкт-Петербург : Питер, 2004. - 265 с. - (Учебное пособие).
2. Хусаинов, Б. С. Структуры и алгоритмы обработки данных. Примеры на языке Си : учеб. пособие для вузов / Б. С. Хусаинов. - М. : Финансы и статистика, 2004. - 464 с.

7. ПЕРЕЧЕНЬ ОСНОВНОЙ И ДОПОЛНИТЕЛЬНОЙ ЛИТЕРАТУРЫ, НЕОБХОДИМОЙ ДЛЯ ОСВОЕНИЯ ДИСЦИПЛИНЫ

№	*Наименование издания*	*Вид занятия*	*Количество экземпляров в библиотеке*, **шт.**	*Обеспеченность*, **(экз./ чел.)**
1	2	3	4	5
Основная литература				
1	Подбельский В.В. Язык СИ++ : учеб. пособие для вузов. – 5-е изд. – М. : Финансы и статистика, 2007. – 559 с. - (Бакалавр. Базовый курс).	Лк, КР, ЛР	21	1
2	Незнанов, А. А. Программирование и алгоритмизация : учебник / А. А. Незнанов. - М. : Академия, 2010. - 304 с.. - (Бакалавр. Академический курс)	Лк, КР, ЛР	10	0,7
Дополнительная литература				
3	С.А. Дьяконица, Д.С. Семенов. Основы программирования на языке Си/Си++: Лабораторный практикум. – Братск: Изд-во БрГУ, 2015. – 153 с.	ЛР	46	1

8. ПЕРЕЧЕНЬ РЕСУРСОВ ИНФОРМАЦИОННО - ТЕЛЕКОММУНИКАЦИОННОЙ СЕТИ «ИНТЕРНЕТ» НЕОБХОДИМЫХ ДЛЯ ОСВОЕНИЯ ДИСЦИПЛИНЫ

1.Электронный каталог библиотеки БрГУ http://irbis.brstu.ru/CGI/irbis64r_15/cgiirbis_64.exe?LNG=&C21COM=F&I21DBN=BOOK&P21DBN=BOOK&S21CNR=&Z21ID=.
2. Электронная библиотека БрГУ http://ecat.brstu.ru/catalog .
3. Электронно-библиотечная система «Университетская библиотека online» http://biblioclub.ru .
4. Электронно-библиотечная система «Издательство «Лань» http://e.lanbook.com .
5. Информационная система "Единое окно доступа к образовательным ресурсам" http://window.edu.ru .
6. Научная электронная библиотека eLIBRARY.RU http://elibrary.ru .
7. Университетская информационная система РОССИЯ (УИС РОССИЯ) https://uisrussia.msu.ru/ .
8. Национальная электронная библиотека НЭБ http://xn--90ax2c.xn--p1ai/how-to-search /.

9. МЕТОДИЧЕСКИЕ УКАЗАНИЯ ДЛЯ ОБУЧАЮЩИХСЯ ПО ОСВОЕНИЮ ДИСЦИПЛИНЫ

9.1. Методические указания для обучающихся по выполнению лабораторных работ

Лабораторная работа №1

Программирование алгоритмов линейной структуры

Цель работы:

Выработать практические навыки работы с системой программирования языка Си/Си++, научиться созда-вать, вводить в компьютер, выполнять и исправлять простейшие программы на языке Си/Си++ в режиме диалога, познакомиться с диагностическими сообщениями компилятора об ошибках при выполнении линейных программ.

Задание:

1. Написать программы и построить блок-схему согласно варианту.

Порядок выполнения:

Запустить программу C++ Builder. Выбрать из предложенных шаблонов «Console wizard». В открывшемся окне написать программный код. Создать все предложенные отчеты по этой программе. Построить блок-схему согласно алгоритму.

Форма отчетности:

Отчет набирается на компьютере и сдается в печатном виде. В отчете должны присутствовать:

1. Номер варианта
2. Цель работы
3. Задание
4. Поэтапное выполнение всех заданий варианта
5. Вывод.

Задания для самостоятельной работы:

Предусмотрены вариантом студента.

Рекомендации по выполнению заданий и подготовке к лабораторной работе

Ознакомиться с теоретическим материалом, представленным в четвертом разделе данной дисциплины.

Основная литература

1. Подбельский В.В. Язык СИ++ : учеб. пособие для вузов. – 5-е изд. – М. : Финансы и статистика, 2007. – 559 с. - (Бакалавр. Базовый курс)
2. Незнанов, А. А. Программирование и алгоритмизация : учебник / А. А. Незнанов. - М. : Академия, 2010. - 304 с.. - (Бакалавр. Академический курс)

Дополнительная литература

1. С.А. Дьяконица, Д.С. Семенов. Основы программирования на языке Си/Си++: Лабораторный практикум. – Братск: Изд-во БрГУ, 2015. – 153 с.

Контрольные вопросы для самопроверки

1. Какие элементы программы являются обязательными?
2. Как формируются отчеты?
3. Происходит процесс отладки?

Лабораторная работа №2

Программирование алгоритмов разветвляющейся структуры

Цель работы:

Выработать практические навыки работы с системой программирования языка Си/Си++, познакомиться с диагностическими сообщениями компилятора об ошибках при выполнении программ, реализующих алгоритмическую структуру “ветвление” (операторы *if, switch).*

Задание:

1. Для данных областей составить программу, которая определяет принадлежность точки с координатами (x,y) закрашенной области.
2. Вычислить значение функции.

Порядок выполнения:

Запустить программу C++ Builder. Выбрать из предложенных шаблонов «Console wizard». В открывшемся окне написать программный код. Создать все предложенные отчеты по этой программе. Построить блок-схему согласно алгоритму.

Форма отчетности:

Отчет набирается на компьютере и сдается в печатном виде. В отчете должны присутствовать:

1. Номер варианта
2. Цель работы
3. Задание
4. Поэтапное выполнение всех заданий варианта
5. Вывод.

Задания для самостоятельной работы:

Предусмотрены вариантом студента.

Рекомендации по выполнению заданий и подготовке к лабораторной работе

Ознакомиться с теоретическим материалом, представленным в четвертом разделе данной дисциплины.

Основная литература

1. Подбельский В.В. Язык СИ++ : учеб. пособие для вузов. – 5-е изд. – М. : Финансы и статистика, 2007. – 559 с. - (Бакалавр. Базовый курс)

Дополнительная литература

1. С.А. Дьяконица, Д.С. Семенов. Основы программирования на языке Си/Си++: Лабораторный практикум. – Братск: Изд-во БрГУ, 2015. – 153 с.

Контрольные вопросы для самопроверки

1. Как работает оператор *switch*?
2. Как работает оператор *if*?
3. Логические операции?

Лабораторная работа №3

Программирование алгоритмов циклической структуры

Цель работы:

Закрепить практические навыки работы с системой программирования языка Си/Си++, познакомиться с диагностическими сообщениями компилятора об ошибках при выполнении программ, содержащих операторы цикла *while, do ... while* и *for*.

Задание:

1. Вычислить функцию с помощью соответствующего ряда и точность вычислений Вычислить значение функции.
2. Определить значения функции в диапазоне изменения аргумента при заданном количестве значений.

Порядок выполнения:

Запустить программу C++ Builder. Выбрать из предложенных шаблонов «Console wizard». В открывшемся окне написать программный код. Создать все предложенные отчеты по этой программе. Построить блок-схему согласно алгоритму.

Форма отчетности:
Отчет набирается на компьютере и сдается в печатном виде. В отчете должны присутствовать:

1. Номер варианта
2. Цель работы
3. Задание
4. Поэтапное выполнение всех заданий варианта
5. Вывод.

Задания для самостоятельной работы:
Предусмотрены вариантом студента.

Рекомендации по выполнению заданий и подготовке к лабораторной работе
Ознакомиться с теоретическим материалом, представленным в четвертом разделе данной дисциплины.

Основная литература

1. Подбельский В.В. Язык СИ++ : учеб. пособие для вузов. – 5-е изд. – М. : Финансы и статистика, 2007. – 559 с. - (Бакалавр. Базовый курс)
2. Незнанов, А. А. Программирование и алгоритмизация : учебник / А. А. Незнанов. - М. : Академия, 2010. - 304 с.. - (Бакалавр. Академический курс)

Дополнительная литература

1. С.А. Дьяконица, Д.С. Семенов. Основы программирования на языке Си/Си++: Лабораторный практикум. – Братск: Изд-во БрГУ, 2015. – 153 с.

Контрольные вопросы для самопроверки

1. Отличия циклов с постусловием и с предусловием?
2. Как работает цикл с параметром?
3. Какие существуют операторы прерывания?

Лабораторная работа №4

Программирование алгоритмов над одномерными массивами

Цель работы:
Закрепить практические навыки работы с системой программирования языка Си/Си++ на примере реализации алгоритмов над статическими массивами.

Задание:

1. Написать программы и построить блок-схему согласно варианту.

Порядок выполнения:
Запустить программу C++ Builder. Выбрать из предложенных шаблонов «Console wizard». В открывшемся окне написать программный код. Создать все предложенные отчеты по этой программе. Построить блок-схему согласно алгоритму.

Форма отчетности:
Отчет набирается на компьютере и сдается в печатном виде. В отчете должны присутствовать:

1. Номер варианта
2. Цель работы
3. Задание
4. Поэтапное выполнение всех заданий варианта
5. Вывод.

Задания для самостоятельной работы:
Предусмотрены вариантом студента.

Рекомендации по выполнению заданий и подготовке к лабораторной работе

Ознакомиться с теоретическим материалом, представленным в четвертом разделе данной дисциплины.

Основная литература

1. Подбельский В.В. Язык СИ++ : учеб. пособие для вузов. – 5-е изд. – М. : Финансы и статистика, 2007. – 559 с. - (Бакалавр. Базовый курс)

Дополнительная литература

1. С.А. Дьяконица, Д.С. Семенов. Основы программирования на языке Си/Си++: Лабораторный практикум. – Братск: Изд-во БрГУ, 2015. – 153 с.

Контрольные вопросы для самопроверки

1. Как осуществляется обращение к элементам массива?
2. Как происходит работа с указателем?
3. Как происходит доступ по ссылке?

Лабораторная работа №5

Программирование алгоритмов над многомерными массивами

Цель работы:

Закрепить практические навыки работы с системой программирования языка Си/Си++ на примере реализации алгоритмов над динамическими многомерными массивами.

Задание:

1. Написать программы и построить блок-схему согласно варианту.

Порядок выполнения:

Запустить программу C++ Builder. Выбрать из предложенных шаблонов «Console wizard». В открывшемся окне написать программный код. Создать все предложенные отчеты по этой программе. Построить блок-схему согласно алгоритму.

Форма отчетности:

Отчет набирается на компьютере и сдается в печатном виде. В отчете должны присутствовать:

1. Номер варианта
2. Цель работы
3. Задание
4. Поэтапное выполнение всех заданий варианта
5. Вывод.

Задания для самостоятельной работы:

Предусмотрены вариантом студента.

Рекомендации по выполнению заданий и подготовке к лабораторной работе

Ознакомиться с теоретическим материалом, представленным в четвертом разделе данной дисциплины.

Основная литература

1. Подбельский В.В. Язык СИ++ : учеб. пособие для вузов. – 5-е изд. – М. : Финансы и статистика, 2007. – 559 с. - (Бакалавр. Базовый курс)

Дополнительная литература

1. С.А. Дьяконица, Д.С. Семенов. Основы программирования на языке Си/Си++: Лабораторный практикум. – Братск: Изд-во БрГУ, 2015. – 153 с.

Контрольные вопросы для самопроверки

1. Массивы указателей?

2. Как происходит объявление многомерного массива?
3. Как осуществляется доступ к элементам?

Лабораторная работа №6

Программирование алгоритмов над массивами символов

Цель работы:

Закрепить практические навыки работы с системой программирования языка Си/Си++ на примере реализации алгоритмов над массивами символов.

Задание:

1. Написать программы и построить блок-схему согласно варианту.

Порядок выполнения:

Запустить программу C++ Builder. Выбрать из предложенных шаблонов «Console wizard». В открывшемся окне написать программный код. Создать все предложенные отчеты по этой программе. Построить блок-схему согласно алгоритму.

Форма отчетности:

Отчет набирается на компьютере и сдается в печатном виде. В отчете должны присутствовать:

1. Номер варианта
2. Цель работы
3. Задание
4. Поэтапное выполнение всех заданий варианта
5. Вывод.

Задания для самостоятельной работы:

Предусмотрены вариантом студента.

Рекомендации по выполнению заданий и подготовке к лабораторной работе

Ознакомиться с теоретическим материалом, представленным в четвертом разделе данной дисциплины.

Основная литература

1. Подбельский В.В. Язык СИ++ : учеб. пособие для вузов. – 5-е изд. – М. : Финансы и статистика, 2007. – 559 с. - (Бакалавр. Базовый курс)

Дополнительная литература

1. С.А. Дьяконица, Д.С. Семенов. Основы программирования на языке Си/Си++: Лабораторный практикум. – Братск: Изд-во БрГУ, 2015. – 153 с.

Контрольные вопросы для самопроверки

1. Какие особенности работы с текстом в языке C++?
2. Варианты ввода-вывода?
3. Какие функции используются для поиска символов в строке?

Лабораторная работа №7

Программирование алгоритмов при помощи функций

Цель работы:

Закрепить практические навыки работы с системой программирования языка Си/Си++ на примере реализации алгоритмов при помощи функций .

Задание:

1. Написать программы и построить блок-схему согласно варианту.

Порядок выполнения:

Запустить программу C++ Builder. Выбрать из предложенных шаблонов «Console wizard». В открывшемся окне написать программный код. Создать все предложенные отчеты по этой

программе. Построить блок-схему согласно алгоритму.

Форма отчетности:
Отчет набирается на компьютере и сдается в печатном виде. В отчете должны присутствовать:

1. Номер варианта
2. Цель работы
3. Задание
4. Поэтапное выполнение всех заданий варианта
5. Вывод.

Задания для самостоятельной работы:
Предусмотрены вариантом студента.

Рекомендации по выполнению заданий и подготовке к лабораторной работе
Ознакомиться с теоретическим материалом, представленным в четвертом разделе данной дисциплины.

Основная литература

1. Подбельский В.В. Язык СИ++ : учеб. пособие для вузов. – 5-е изд. – М. : Финансы и статистика, 2007. – 559 с. - (Бакалавр. Базовый курс)
2. Незнанов, А. А. Программирование и алгоритмизация : учебник / А. А. Незнанов. - М. : Академия, 2010. - 304 с.. - (Бакалавр. Академический курс)

Дополнительная литература

1. С.А. Дьяконица, Д.С. Семенов. Основы программирования на языке Си/Си++: Лабораторный практикум. – Братск: Изд-во БрГУ, 2015. – 153 с.

Контрольные вопросы для самопроверки

1. Как осуществляется перегрузка функции?
2. Как осуществляется передача параметров функции?
3. Принцип работы рекурсивной функции?

9.2. Методические указания по выполнению курсовой работы

Работа посвящена разработке программы, включающая в себя алгоритмы по обработки линейных списков.

Каждое индивидуальное задание предполагает разработку алгоритмов и программы по созданию нового списка, добавление элементов в список, вывод списка на дисплей, сохранение данных списка в файл, чтение данных списка из файла, удаление списка из памяти компьютера, поиск элемента в списке, сортировка списка и удаление элемента списка. Разрабатываемая программа должна обеспечивать “диалоговый режим” работы пользователя с программой и содержать информацию о ее назначении (название или краткое описание задачи, на решение которой она направлена).

Проектирование производиться каждым студентом индивидуально, по вариантам.

10. ПЕРЕЧЕНЬ ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ, ИСПОЛЬЗУЕМЫХ ПРИ ОСУЩЕСТВЛЕНИИ ОБРАЗОВАТЕЛЬНОГО ПРОЦЕССА ПО ДИСЦИПЛИНЕ

1. ОС Windows 7 Professional
2. Microsoft Office 2007 Russian Academic OPEN No Level
3. Антивирусное программное обеспечение Kaspersky Security.
4. Visual Studio Community

11. ОПИСАНИЕ МАТЕРИАЛЬНО-ТЕХНИЧЕСКОЙ БАЗЫ, НЕОБХОДИМОЙ ДЛЯ ОСУЩЕСТВЛЕНИЯ ОБРАЗОВАТЕЛЬНОГО ПРОЦЕССА ПО ДИСЦИПЛИНЕ

Вид занятия	*Наименование аудитории*	*Перечень основного оборудования*	*№ ЛР или ПЗ*
1	3	4	5
ЛР	Дисплейные классы	Персональные компьютеры	ЛР 1-7
ЛК	Дисплейные классы	Персональные компьютеры	ЛР 1-7
КР	Дисплейные классы	Персональные компьютеры	
СР	ЧЗЗ	-	-

Приложение 1

ФОНД ОЦЕНОЧНЫХ СРЕДСТВ ДЛЯ ПРОВЕДЕНИЯ ПРОМЕЖУТОЧНОЙ АТТЕСТАЦИИ ОБУЧАЮЩИХСЯ ПО ДИСЦИПЛИНЕ

1. Описание фонда оценочных средств (паспорт)

№ компетенции	Элемент компетенции	Раздел	Тема	ФОС
ОК-7	Способность к самоорганизации и самообразованию.	**1.** Инструментарий технологии программирования	1.1 Программа как формализованное описание процесса обработки данных	Экзаменационный билет
			1.3 Специфика разработки программных средств	
			1.5 Понятие качества программного средства	
		2. Основные принципы и подходы к разработке программных алгоритмов	2.1 Понятие алгортма и его свойства	Экзаменационный билет
			2.3 Основные алгоритмические конструкции	
		3. Основы программирования на языке высокого уровня Си++	3.1 Архитектура программного средства	Экзаменационный билет
			3.3 Этапы работы с программой на языке си/си++ в системе программирования	
			3.5 Синтаксис и семантика языка	
		4. Типы данных, выражения и операции в языке программирования с++	4.1 Концепция типов данных в языке с++	Экзаменационный билет
			4.3 Константы и переменные	
		5. Операторы языка программирования Си++ и управление их исполнением	5.1 Операторы условия и передачи управления	Экзаменационный билет
			5.3 Одномерные и многомерные массивы	
		6.Указатели и динамическое распределение памяти	6.1 Указатели, ссылки и адресная арифметика	Экзаменационный билет
			6.3 Динамическое размещение массивов	
		7.Функции языка программирования Си++	7.1 Объявление и описание функций	Экзаменационный билет
			7.3 Рекурсивные функции	
		8.Статические и динамические структуры данных	8.1 Статические структуры данных	Экзаменационный билет
			8.3 Динамические структуры данных	
ОПК-9	Способность	**1.**	1.2 Технология программирования	Экзаменационный

	использовать навыки работы с компьютером, владеть методами информационных технологий, соблюдать основные требования информационной безопасности.	Инструментарий технологии программирования	как инструмент разработки надежных программных средств.	билет
			1.4 Жизненный цикл программного средства	
			1.6 Виды современных языков программирования	
		2.Основные принципы и подходы к разработке программных алгоритмов	2.2 Классы алгоритмов и способы их описания	Экзаменационный билет
		3.Основы программирования на языке высокого уровня Си++	3.2 Проектирование программного обеспечения при структурном подходе	Экзаменационный билет
			3.4 Разработка структуры программы и модульное программирование	
		4.Типы данных, выражения и операции в языке программирования с++	4.2 Арифметические типы данных	Экзаменационный билет
			4.4 Основные операции языка си	
		5.Операторы языка программирования Си++ и управление их исполнением	5.2 Операторы циклов	Экзаменационный билет
		6.Указатели и динамическое распределение памяти	6.2 Операторы динамического распределения памяти	Экзаменационный билет
			6.4 Типы строк в языке с++	
		7.Функции языка программирования Си++	7.2 Функции с переменным количеством параметров	Экзаменационный билет
			7.4 Передача параметров в функцию по ссылке	
		8.Статические и динамические структуры данных	8.2 Полустатические структуры данных	Экзаменационный билет

2. Экзаменационные вопросы

№ п/п	Компетенции		ЭКЗАМЕНАЦИОННЫЕ ВОПРОСЫ	№ и наименование раздела
	Код	Определение		
1	2	3	4	5
1	ОК-7	Способность к самоорганизации и самообразовани	1. Цели программирования.	**1.** Инструментарий технологии программирования
			2.Технология программирования.	
			3.Специфика разработки программных средств.	

		ю.	4.Алгоритм.	**2.** Основные принципы и подходы к разработке программных алгоритмов
			5.Классы алгоритмов.	
			6.Разветвляющиеся алгоритмические конструкции.	
			7.Классы архитектур программного стредства.	**3.** Основы программирования на языке высокого уровня Си++
			8.Структурная схема программного средства.	
			9.Синтаксис языка С++.	
			10.Арифметические типы данных.	**4.** Типы данных, выражения и операции в языке программирования с++
			11.Арифметические операции.	
			12.Операции присваивания.	
			13.Операторы передачи управления.	**5.** Операторы языка программирования Си++ и управление их исполнением
			14.Циклы с пред- и постусловием.	
			15.Многомерные массивы.	
			16.Указатели.	**6.** Указатели и динамическое распределение памяти
			17.Динамическое распределение многомерных массивов.	
			18.Ввод-вывод строк.	
			19. Функции языка С++ .	**7.** Функции языка программирования Си++
			20. Рекурсивные функции.	
			21. **Эвристический алгоритм.**	
			22.Динамические структуры данных.	**8.** Статические и динамические структуры данных
			23.Полустатические структуры данных.	
			24.Способы адресации.	
2	ОПК-9	Способность использовать навыки работы с компьютером, владеть методами информационных технологий, соблюдать основные требования информационной безопасности	1.Описательный процесс.	**1.** Инструментарий технологии программирования
			2.Методология программирования.	
			3.Жизненный цикл программного средства.	
			4.Свойства алгоритма.	**2.** Основные принципы и подходы к разработке программных алгоритмов
			5. Методы описания алгоритмов.	
			6.Циклические алгоритмические конструкции.	
			7.Архитектура программного средства.	**3.** Основы программирования на языке высокого уровня Си++
			8.Функциональная схема программного средства.	
			9.Этапы разработки программного средства.	
			10.Приведение типов.	**4.** Типы данных, выражения и операции в языке программирования с++
			11.Унарные и бинарные операции.	
			12.Логические операции.	
			13.Операторы условия.	**5.** Операторы языка программирования Си++ и управление их исполнением
			14.Цикл с параметром.	
			15.Одномерные массивы.	
			16.Операторы динамического	**6.** Указатели и

			распределения памяти.	динамическое распределение памяти
			17.Динамическое распределение одномерных массивов.	
			18.Работа со строками С++.	
			19.Вызов функции.	**7.** Функции языка программирования Си++
			20.Функции с переменным количеством параметров.	
			21.Передача параметров в функцию по ссылке.	
			22.Статические структуры данных.	**8.** Статические и динамические структуры данных
			23.Связные списки.	
			24.Мультисписки.	

3. Описание показателей и критериев оценивания компетенций

Показатели	Оценка	Критерии
Знать (ОК-7): - содержание процессов самоорганизации и самообразования, их особенностей и технологий реализации, исходя из целей совершенствования профессиональной деятельности; (ОПК-9): - базовое устройство персонального компьютера. Основные информационные процессы происходящие в персональном компьютере. **Уметь** (ОК-7): – планировать цели и устанавливать приоритеты при выборе способов принятия решений с учетом условий, средств, личностных возможностей и временной перспективы	**отлично**	Студент должен во время ответа показать знания: основных алгоритмических конструкций, синтаксис и семантика языка СИ, основных терминов используемые в научно-технической литературе по базам данных. Студент должен иметь навыки владения: использования универсальных программных продуктов на ПК, понимания материала и способности высказывания мыслей на научно-техническом языке. Студент во время ответа должен продемонстрировать умения: использования навыков анализа основных понятий в теории систем управления базами данных.
	хорошо	Ответ содержит неточности. Дополнительные вопросы требуется, но студент с ними справляется отлично.

осуществления деятельности; (ОПК-9): – использовать персональный компьютер для самостоятельной работы. **Владеть** (ОК-7): – приемами саморегуляции эмоциональных и функциональных состояний при выполнении профессиональной деятельности; (ОПК-9): – достаточным уровнем использования универсальных пакетов прикладных компьютерных программ	**удовлетворительно**	Ответил только на один вопрос, либо слабо ответил на оба вопроса. На дополнительные вопросы отвечает неуверенно.
	неудовлетворительно	На оба вопроса студент отвечает неубедительно. На дополнительные вопросы преподавателя также не может ответить.

4. Методические материалы, определяющие процедуры оценивания знаний, умений, навыков и опыта деятельности

Дисциплина программирование и основы алгоритмизации направлена на ознакомление с основными алгоритмическими структурами, и их практическим применением в современных языках программирования; на получение теоретических знаний и практических навыков использования различных алгоритмов, и их дальнейшего использования в практической деятельности.

Изучение дисциплины системы управления базами данных предусматривает:

- лекции,
- лабораторные работы,
- курсовую работу,
- самостоятельную работу студента,
- экзамен.

В ходе освоения раздела 1 «Инструментарий технологии программирования» студенты должны изучить: основные понятия и определения дисциплины и способы их применения.

В ходе освоения раздела 2 «Основные принципы и подходы к разработке программных алгоритмов» студенты должны изучить: понятия алгоритма и основные алгоритмические конструкции, и способы их применения.

В ходе освоения раздела 3 «Основы программирования на языке высокого уровня Си++» студенты должны изучить: архитектуру программного средства, основные функции языка, синтаксис и семантику языка.

В ходе освоения раздела 4 «Типы данных, выражения и операции в языке программирования с++» студенты должны изучить: типы данных, используемые в языке .

В ходе освоения раздела 5 «Операторы языка программирования Си++ и управление их исполнением» студенты должны изучить: операторы условия и передачи управления, одномерные и многомерные массивы.

В ходе освоения раздела 6 «Указатели и динамическое распределение памяти»

студенты должны изучить: указатели и ссылки, динамическое размещение массивов.

В ходе освоения раздела 7 «Функции языка программирования Си++» студенты должны изучить: понятие, объявление и описание функции, рекурсивные функции.

В ходе освоения раздела 8 «Статические и динамические структуры данных» студенты должны изучить: статические, полустатические и динамические структуры данных.

В процессе проведения лабораторных работ происходит закрепление знаний, формирование умений и навыков реализации представления о работе с алгоритмическими структурами и использование языка Си++.

При подготовке к экзамену рекомендуется особое внимание уделить следующим вопросам: основные алгоритмические структуры, типы данных применяемые в языке Си++, функции и передача параметров в функции.

Работа с литературой является важнейшим элементом в получении знаний по дисциплине. Прежде всего, необходимо воспользоваться списком рекомендуемой по данной дисциплине литературой. Дополнительные сведения по изучаемым темам можно найти в периодической печати и Интернете.

Приложение 2

АННОТАЦИЯ
рабочей программы дисциплины
Программирование и основы алгоритмизации

1. Цель и задачи дисциплины

Целью изучения дисциплины является формирование у студентов знаний и навыков по использованию современных технологий и методов разработки программных систем для решения практических задач с использованием современных инструментальных средств, необходимых в дальнейшем, при проектировании и эксплуатации систем управления и автоматизации.

Задачей изучения дисциплины является: обучении свободному владению языком программирования как “средством выражения” алгоритмов применительно к традиционному кругу задач - арифметико-логическим, сортировки и поиска, приближенным вычислений, обработки текста.

2. Структура дисциплины

2.1 Распределение трудоемкости по отдельным видам учебных занятий, включая самостоятельную работу: Лк – 18 часов, ЛР – 36 часов, СРС – 54 часов.

Общая трудоемкость дисциплины составляет 144 часов, 4 зачетных единиц

2.2 Основные разделы дисциплины:

1. Инструментарий технологии программирования
2. Основные принципы и подходы к разработке программных алгоритмов
3. Основы программирования на языке высокого уровня Си++
4. Типы данных, выражения и операции в языке программирования с++
5. Операторы языка программирования Си++ и управление их исполнением
6. Указатели и динамическое распределение памяти
7. Функции языка программирования Си++
8. Статические и динамические структуры данных

3. Планируемые результаты обучения (перечень компетенций)

Процесс изучения дисциплины направлен на формирование следующих компетенций:

ОК-7 - Способность к самоорганизации и самообразованию;

ОПК-9 - Способность использовать навыки работы с компьютером, владеть методами информационных технологий, соблюдать основные требования информационной безопасности

4. Вид промежуточной аттестации: экзамен, контрольная работа

Приложение 3

Протокол о дополнениях и изменениях в рабочей программе
на 20___-20___ учебный год

1. В рабочую программу по дисциплине вносятся следующие дополнения:
_____
_____

2. В рабочую программу по дисциплине вносятся следующие изменения:

_____
_____
_____

Протокол заседания кафедры №____ от «___» _____ 20 ____ г.,
(разработчик)

Заведующий кафедрой _____ _____
(подпись) *(Ф.И.О.)*

Приложение 4

ФОНД ОЦЕНОЧНЫХ СРЕДСТВ ДЛЯ ТЕКУЩЕГО КОНТРОЛЯ УСПЕВАЕМОСТИ ПО ДИСЦИПЛИНЕ

1. Описание фонда оценочных средств (паспорт)

№ компетенции	Элемент компетенции	Раздел	Тема	ФОС
ОК-7	Способность к самоорганизации и самообразованию.	**2.** Основные принципы и подходы к разработке программных алгоритмов	Понятие алгоритма и его свойства.	*Отчеты по лабораторным работам.*
			Основные алгоритмические конструкции.	
		3. Основы программирования на языке высокого уровня Си++	Архитектура программного средства.	*Отчеты по лабораторным работам.*
			Этапы работы с программой на языке си/си++ в системе программирования.	
			Синтаксис и семантика языка.	
		4. Типы данных, выражения и операции в языке программирования с++.	Константы и переменные.	*Отчеты по лабораторным работам.*
		5. Операторы языка программирования Си++ и управление их исполнением	Операторы условия и передачи управления.	*Отчеты по лабораторным работам.*
			Одномерные и многомерные массивы.	
		6. Указатели и динамическое распределение памяти	Указатели, ссылки и адресная арифметика.	*Отчеты по лабораторным работам.*
			Динамическое размещение массивов.	
		7. Функции языка программирования Си++	Объявление и описание функций.	*Отчеты по лабораторным работам.*
			Рекурсивные функции.	
		8.Статические и динамические структуры данных	Динамические структуры данных.	*Курсовая работа.*

ОПК-9	Способность использовать навыки работы с компьютером, владеть методами информационных технологий, соблюдать основные требования информационной безопасности.	**2**. Основные принципы и подходы к разработке программных алгоритмов	Классы алгоритмов и способы их описания.	*Отчеты по лабораторным работам.*
		3. Основы программирования на языке высокого уровня Си++	Проектирование программного обеспечения при структурном подходе.	*Отчеты по лабораторным работам.*
			Разработка структуры программы и модульное программирование.	
		4. Типы данных, выражения и операции в языке программирования с++.	Основные операции языка си.	*Отчеты по лабораторным работам.*
		5. Операторы языка программирования Си++ и управление их исполнением	Операторы циклов.	*Отчеты по лабораторным работам.*
		6.Указатели и динамическое распределение памяти	Операторы динамического распределения памяти.	*Отчеты по лабораторным работам.*
			Типы строк в языке с++.	
		7. Функции языка программирования Си++	Передача параметров в функцию по ссылке.	*Отчеты по лабораторным работам.*
		8.Статические и динамические структуры данных	Динамические структуры данных.	*Курсовая работа.*

2. Описание показателей и критериев оценивания компетенций

Показатели	Оценка	Критерии

Знать (ОК-7): - Содержание процессов самоорганизации и самообразования, их особенностей и технологий реализации, исходя из целей совершенствования профессиональной деятельности; (ОПК-9): - Базовое устройство персонального компьютера. Основные информационные процессы происходящие в персональном компьютере. **Уметь** (ОК-7): – Планировать цели и устанавливать приоритеты при выборе способов принятия решений с учетом условий, средств, личностных возможностей и временной перспективы осуществления деятельности; (ОПК-9): – Использовать персональный компьютер для самостоятельной работы. **Владеть** (ОК-7): – Базовое устройство персонального компьютера. Основные информационные процессы происходящие в персональном компьютере; (ОПК-9): Достаточным уровнем использования универсальных пакетов прикладных компьютерных программ	**зачтено**	Во время защиты лабораторных работ и практических работ студент ответил на поставленные преподавателем вопросы.
	не зачтено	Во время защиты лабораторных работ и практических работ студент не смог дать ответы на поставленные преподавателем вопросы. Либо отчет имеет ряд замечаний.