Ю. Ю. Громов,
С. И. Татаренко

Языки СИ и С++ для решения инженерных и экономических задач

• Издательство ТГТУ •

Министерство образования Российской Федерации
Тамбовский государственный технический университет

Ю. Ю. Громов, С. И. Татаренко

Языки СИ и С++ для решения инженерных и экономических задач

Рекомендовано Ученым советом университета в качестве учебного пособия

Тамбов
• ИЗДАТЕЛЬСТВО ТГТУ •
2001

1 ОПИСАНИЕ ЯЗЫКА СИ

1.1 БАЗОВЫЕ ПОНЯТИЯ ЯЗЫКА СИ

1.1.1 Используемые символы (алфавит)

В алфавит языка СИ включены все символы присутствующие на клавиатуре персонального компьютера (не русифицированного) кроме символов @ и $. Их можно разделить на несколько групп.

1 *Алфавитно-цифровые символы* - прописные и строчные буквы латинского алфавита A - Z a - z, цифры 0 - 9 и символ подчеркивания _ , который может использоваться как буква. Следует отметить, что одинаковые прописные и строчные буквы считаются различными символами, т.е. f и F - это разные символы.

2 *Знаки препинания, знаки операций и специальные знаки* (табл. 1).

Таблица 1

Символ	Наименование	Символ	Наименование
,	запятая	)	круглая скобка правая
.	точка	(	круглая скобка левая
;	точка с запятой	{	фигурная скобка левая
?	вопросительный знак	}	фигурная скобка правая
‘	апостроф	<	меньше
!	восклицательный знак	>	больше
|	вертикальная черта	]	квадратная скобка правая
/	дробная черта	[	квадратная скобка левая
\	обратная черта	#	номер
~	тильда	%	процент
*	звездочка	&	амперсенд
+	плюс	^	логическое не
-	минус	=	равно
«	кавычки		

3 *Обобщенные пробельные символы* - кроме пробела, символы, не имеющие графического представления и служащие для отделения частей текста программ, - это возврат каретки, новая строка, табуляция, вертикальная табуляция, новая страница.

Все остальные представимые символы (прописные и строчные буквы русского алфавита, знаки @ и $, символы псевдографики и другие) не включены в алфавит языка СИ, но могут содержаться в тексте программы внутри данных, записанных в программе.

В данных, обрабатываемых СИ-программой, кроме обобщенных пробельных символов могут встречаться и другие символы, не имеющие графического представления; для их записи используются специальные последовательности, начинающиеся со знака \ . Эти последовательности представлены в табл. 2

Таблица 2

Управляющая последовательность	Наименование символа	Шестнадцатеричная замена
\a	Звонок	07
\b	Возврат на шаг	08
\t	Табуляция	09
\n	Новая строка	0a
\v	Вертикальная табуляция	0b
\r	Возврат каретки	0c
\f	Новая страница	0d
\»	Кавычки	22
\’	Апостроф	27
\0	Ноль-символ	00
\\	Обратная дробная черта	5c
\ddd	Символ набора кодов ПЭВМ в восмеричном представлении	
\xdd	Символ набора кодов ПЭВМ в шестнадцатеричном представлении	

Последовательности вида \ddd и \xdd (здесь d обозначает цифру) позволяет преставить символ из набора кодов ПЭВМ как последовательность восьмеричных или шестнадцатеричных цифр соответственно. Например, символ возврата каретки может быть представлен различными способами:

\r - общая управляющая последовательность;

\015 - восмеричная управляющая последовательность;

\x0D - шестнадцатеричная управляющая последовательность.

Кроме определения управляющей последовательности, символ обратной дробной черты (\) используется также как символ продолжения. Если за (\) следует (\n), то оба символа игнорируются, а следующая строка является продолжением предыдущей. Это свойство может быть использовано для записи длинных строк.

1.1.2 Ключевые слова

Ключевые слова - зарезервированные слова, использование которых строго регламентировано. Использовать ключевые слова для обозначения объектов программы запрещено.

Список ключевых слов:

do auto enum break short double continue typedef
if case else floan struct return register default
int char long union extern switch unsigned

for goto void while signed sizeof volatile

Кроме того, в большинстве версий СИ, зарезервированными являются слова: near, far, huge, interrupt , const, volatile.

1.1.3 Идентификаторы

Идентификатором называется последовательность цифр и букв, а также специальных символов при условии, что первой стоит буква или специальный символ. Для образования идентификаторов могут быть использованы строчные или прописные буквы латинского алфавита. В качестве специального символа может использоваться символ подчеркивание (_). Два идентификатора, для образования которых используются совпадающие строчные и прописные буквы, считаются различными. Например: abc, ABC, A128B, a128b.

Важной особенностью является то, что компилятор допускает любое количество символов в идентификаторе, хотя значимыми являются первые 31 символ. Идентификатор создается на этапе объявления переменной, функции, структуры и т.п.; после этого его можно использовать в последующих операторах разрабатываемой программы. Следует отметить важные особенности при выборе идентификатора.

Во-первых, идентификатор не должен совпадать с ключевыми словами, с зарезервированными словами и именами функций библиотеки компилятора языка СИ.

Во-вторых, следует обратить особое внимание на использование символа _ (подчеркивание) в качестве первого символа идентификатора. Во многих компиляторах СИ с одного или двух таких символов начинаются имена системных функций и (или) переменных, поэтому при использовании таких идентификаторов программы могут оказаться непереносимыми, т.е. их нельзя использовать на компьютерах других типов.

В-третьих, на идентификаторы, используемые для определения внешних переменных, должны быть наложены ограничения, формируемые используемым редактором связей (отметим, что использование различных версий редактора связей или различных редакторов накладывает различные требования на имена внешних переменных).

1.1.4 Использование комментариев в тексте программы

Комментарий - это набор символов, начинающийся символами /* и заканчивающийся символами */, которые игнорируются компилятором. Внутри набора символов, который представляет комментарий, не может быть специальных символов, определяющих начало и конец комментариев, соответственно (/* и */). Отметим, что комментарии могут заменить как одну строку, так и несколько. Например:

```
/* комментарии к программе */
/* начало алгоритма */
          или
/* комментарии можно записать в следующем виде, однако надо
   быть осторожным, чтобы внутри последовательности, которая игнорируется компилятором, не попались операторы программы, которые также будут игнорироваться */
```

Неправильное определение комментариев:

```
/* комметарии к алгоритму /* решение краевой задачи */ */
          или
/* комметарии к алгоритму решения */ краевой задачи */
```

Некоторые трансляторы СИ поддерживают и второй метод записи комментариев – строка, начинающаяся с символов //, считается комментарием.

```
// -  однострочный комментарий.
```

1.2 БАЗОВЫЕ ТИПЫ ДАННЫХ И ОБЪЯВЛЕНИЕ ПЕРЕМЕННЫХ

Данные - объекты, над которыми возможно выполнение некоторых действий, в результате чего получаются некоторые значения. В качестве таких объектов выступают переменные и константы. Данные могут иметь три характеристики: имя, тип, значение. Имя - это имя, под которым данное известно в программе. В качестве имени используется идентификатор. Тип данного определяет область допустимых значений и возможность применения к нему тех или иных операций. Значение - это текущее значение конкретного данного.

Данные, значение которых может быть изменено во время выполнения программы, - это переменные. Все переменные, используемые в программе, должны быть предварительно объявлены; при этом каждой переменной ставится в соответствие имя и тип. Объявления переменной имеют следующую форму:

[спецификатор_класса_памяти] спецификатор_типа

описатель [= инициатор] [,описатель [= инициатор]]...

Описатель - идентификатор переменной либо более сложная конструкция с квадратными скобками, круглыми скобками или звездочками.

Спецификатор_типа - одно или несколько ключевых слов или более сложная конструкция, содержащая звездочки, круглые и квадратные скобки, определяющая тип объявляемой переменной. В языке СИ имеется стандартный набор типов данных, используя который можно сконструировать свои собственные типы данных.

Инициатор - задает начальное значение или список начальных значений, которые (которое) присваиваются переменной при объявлении.

Спецификатор_класса_памяти - определяется одним из четырех ключевых слов языка СИ: auto, extern, register, static, и, с одной стороны, указывает, каким образом будет распределяться память под объявляемую переменную, а с другой, - область видимости этой переменной, т.е. из каких частей программы можно к ней обратиться.

Данные, значение которых невозможно изменить во время выполнения программы, - это константы, и, поскольку они имеют всегда одно и то же значение, им не требуется имя. В программе константа - это некоторое значение, записанное в той или иной форме.

Константы, имеющие имя, называются именованными константами и их, как и переменные, нужно предварительно объявлять. При объявлении именованной константы ей ставится в соответствие имя, тип и значение. Объявление именованной константы можно выполнить двумя разными способами. Первый способ аналогичен объявлению переменной, но перед именем именованной константы записывается модификатор const, а после имени обязательно записывается инициализатор. Второй способ может быть применен только для целых констант и связан с использованием специального перечислимого типа.

В языке СИ имеются арифметические данные целые и с плавающей точкой, символьные данные и указатели. Из этих данных можно образовывать различные совокупности данных: массивы, структуры и объединения. Подмножествами целых данных являются также данные перечислимого типа и битовые поля.

В СИ не реализован логический тип данных, но имеется большой набор логических операций. В этих операциях вместо логических величин можно использовать любую арифметическую величину или указатель. При этом любое значение отличное от нуля считается истиной, а нулевое значение - ложью.

1.2.1 Целые типы данных

В СИ имеется несколько целых типов данных, отличающихся размером памяти, отводимой для хранения значения данного, и способом записи знака значения. Для

объявления целых переменных используются спецификаторы_типа представленные в табл. 3.

Заметим, что ключевые слова signed и unsigned необязательны. Они указывают как интерпретируется нулевой бит объявляемой переменной, т.е. если указано ключевое слово unsigned, то нулевой бит интерпретируется как часть числа, в противном случае нулевой бит интерпретируется как знаковый. В случае отсутствия ключевого слова unsigned целая переменная считается знаковой. В том случае, если спецификатор_типа состоит из ключевого типа signed или unsigned и далее следует идентификатор переменной, то она будет рассматриваться как переменная типа int.

Таблица 3

Тип	Размер в байтах	Диапазон значений
char	1	от -128 до 127
int	2 или 4	как long или short
short	2	от -32768 до 32767
long	4	от -2 147 483 648 до 2 147 483 647
unsigned shar	1	от 0 до 255
unsigned int	2 или 4	как unsigned long или unsigned short
unsigned short	2	от 0 до 65535
unsigned long	4	от 0 до 4 294 967 295

Спецификатор_типа char требуется для представления символьных переменных, которые в СИ могут использоваться как арифметические данные. Значением объекта типа char является код (размером 1 байт), соответствующий представляемому символу. Для представления символов русского алфавита необходимо использовать тип unsigned char, так как коды русских букв превышают величину 127. Например:

```
unsigned int n;
unsigned int b;
signed char lat;
unsigned char rus;
int c;            (подразумевается signed int c )
unsigned d;               (подразумевается unsigned int d )
signed f;         (подразумевается signed int f )
```

Следует сделать следующее замечание: в языке СИ не определено представление в памяти и диапазон значений для переменных со спецификаторами_типа int и unsigned int. Размер памяти для переменной типа int определяется длиной машинного слова, которое имеет различный размер на разных машинах. Так, на 16-ти разрядных машинах размер слова равен двум байтам, на 32-х разрядных машинах, соответственно, четырем байтам, т.е. тип int эквивалентен типам short int, или long int в зависимости от архитектуры используемой

ПЭВМ. Таким образом, одна и та же программа может правильно работать на одном компьютере и неправильно на другом.

Целая константа - это число, записанное в программе в десятичной, восьмеричной или шестнадцатеричной форме. Десятичная целая константа может состоять из десятичных цифр, причем первая цифра не должна быть нулем.

Восьмеричная константа состоит из обязательного нуля и одной или нескольких восьмеричных цифр (среди цифр должны отсутствовать восьмерка и девятка, так как эти цифры не входят в восьмеричную систему счисления).

Шестнадцатеричная константа начинается с обязательной последовательности 0x или 0X и содержит одну или несколько шестнадцатеричных цифр (цифры, представляющие собой набор цифр шеснадцатеричной системы счисления: 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F). Примеры целых констант:

Десятичная	Восьмеричная	Шестнадцатеричная
16	020	0x10
127	0117	0x2B
240	0360	0XF0

Если требуется сформировать отрицательную целую константу, то используют знак "-" перед записью константы (который будет называться унарным минусом). Например: -0x2A, -088, -16 .

Каждой целой константе присваивается тип, определяющий преобразования, которые должны быть выполнены, если константа используется в выражениях. Тип константы определяется следующим образом:

- десятичные константы рассматриваются как величины со знаком, и им присваивается тип int (целая) или long (длинная целая) в соответствие со значением константы. Если константа меньше 32768, то ей присваивается тип int, в противном случае long;

- восьмеричным и шестнадцатеричным константам присваивается тип int, unsigned int (беззнаковая целая), long или unsigned long в зависимости от значения константы согласно табл. 4.

Таблица 4

Тип	Диапазон шестнадцатеричных констант	Диапазон восьмеричных констант
int	0x0 - 0x7FFF	0 - 077777
unsigned int	0X8000 - 0XFFFF	0100000 - 0177777
long	0X10000 - 0X7FFFFFFF	0200000 - 017777777777
unsigned long	0X80000000 - 0XFFFFFFFF	020000000000 - 037777777777

Для того, чтобы любую целую константу определить типом long, достаточно в конце константы поставить букву "l" или "L"; для того, чтобы сделать константу беззнаковой, можно использовать букву u или U . Буквы L и U можно использовать одновременно. Например:

5l, 6l, 128L, 0105L, OX2A11L, 0x004000c2ul.

Символьная константа тоже относится к данным целого типа signed int, но записывается в символьной или кодовой форме. В символьной форме символьная константа - это любой символ (кроме обратного слеша \, апострофа ' и кавычек "), заключенный в апостофы. Например, 'ф' , '+' , '(' . В кодовой форме можно записать любую символьную константу, в том числе и неимеющую символьного представления. Кодовая форма начинается со знака \ и может содержать специальный символ или восьмеричное или шестнадцатеричное число (см. табл. 2). Например, ' '- пробел, 'Q'- буква Q, '\n' - символ новой строки, '\\' - обратная дробная черта, '\v' - вертикальная табуляция.

1.2.2 Данные плавающего типа

Значения данных плавающего типа хранятся в памяти в виде мантиссы и порядка, и поэтому могут быть как целыми так и дробными. Для переменных, представляющих число с плавающей точкой, используются спецификаторы_типа float или double.

Величина типа float занимает 4 байта. Из них 1 байт отводится для знака числа, 8 бит для порядка и 23 бита для мантиссы. Диапазон значений переменной от 3.14E-3 до 3.14E+3.

Величина типа double занимает 8 бит в памяти. Ее формат аналогичен формату float. Биты памяти распределяются следующим образом: 1 бит для знака, 11 бит для экспоненты и 52 бита для мантиссы. Диапазон значений от 1.7E-308 до 1.7E+308. Примеры: float f, a, b; double x,y;

Константа с плавающей точкой - десятичное число, представленное в виде действительной величины с десятичной точкой и (или) экспонентой. Формат имеет вид: [цифры].[цифры] [E|e [+|-] цифры].

Константы с плавающей точкой представляют положительные величины удвоенной точности (имеют тип double). Для определения отрицательной величины необходимо сформировать константное выражение, состоящее из знака минус и положительной константы. Например: 115.75, 1.5E-7, -0.025, .075, -0.85E0

1.2.3 Переменные перечислимого типа

Переменная, которая может принимать значение из некоторого списка целых значений, называется переменной перечислимого типа или перечислением.

Объявление перечисления начинается с ключевого слова enum и имеет следующую форму:

enum имя_тега_перечисления {список_перечисления}

описатель [,описатель...];

Объявление перечисления задает тип переменной перечисления, тег перечисления (шаблон) и дополнительно определяет список целых именованных констант, называемый списком_перечисления.

Один из компонентов объявления - либо имя_тега_перечисления, либо список_перечисления, либо описатель - может быть опущен. При этом можно объявить либо переменную (пропустить тег или список_перечисления), либо тег (пропустить описатель), либо и то, и другое (ничего не пропускать).

Объявленный тег как бы именует список_перечисления и его можно использовать в последующем для объявления переменных такого же типа без указания списка_перечисления.

Переменная типа перечисления может принимать только значения, указанные в списке_перечисления, состоящем из именованных констант. Именованные константы списка_перечисления имеют тип int. Таким образом, память, соответствующая переменной перечисления, - это память, необходимая для размещения значения типа int, и использовать переменную типа перечисления допускается также как переменную типа int.

Список-перечисления содержит одну или несколько конструкций вида

идентификатор [= константное_выражение]

Каждый идентификатор именует элемент перечисления. Все идентификаторы в списке enum должны быть уникальными. В случае отсутствия константного_выражения первому идентификатору соответствует значение 0, следующему идентификатору - значение 1 и т.д. Имя константы перечисления эквивалентно ее значению.

Константное_выражение - это выражение, состоящее из констант и знаков операций, например, 2*3 , 6+5.

Идентификатор, связанный с константным_выражением, принимает значение, задаваемое этим константным выражением. Константное_выражение должно иметь тип int и может быть как положительным, так и отрицательным. Следующему идентификатору в списке присваивается значение, равное константному выражению плюс 1, если этот идентификатор не имеет своего константного_выражения.

Использование элементов перечисления должно подчиняться следующим правилам:

1 Список_перечисления может содержать повторяющиеся значения.

2 Идентификаторы в списке перечисления должны быть отличны от всех других идентификаторов в той же области видимости, включая имена обычных переменных и идентификаторы из других списков перечислений.

3 Имена типов перечислений должны быть отличны от других имен типов перечислений, структур и объединений в этой же области видимости.

4 Значение может следовать за последним элементом списка перечисления.

Например:

```
enum week {     SUB = 0, /* 0 */
                VOS = 0, /* 0 */
                POND, /* 1 */
                VTOR, /* 2 */
                SRED, /* 3 */
                HETV, /* 4 */
                PJAT /* 5 */
           }    rab_ned
```

В этом примере обьявлен перечислимый тег week с соответствующим множеством значений и объявлена переменная rab_ned, имеющая тип week.

```
enum orig {     SET, /* 0 */ CUR, /* 1 */ END }; /* 2 */
```

Здесь объявлен тег orig со множеством значений 0, 1, 2. С помощью тегов week и orig можно объявить переменные перечислимого типа:

```
enum week ned_1, ned_2;     enum orig nachalo;
```

переменные ned_1, ned_2 будут такими же как и переменная rab_ned, переменная nachalo - это переменная перечислимого типа со множеством допустимых значений 0, 1, 2.

Заметим, имена именованных констант в языке СИ принято (но необязательно) записывать прописными буквами, что позволяет сразу узнавать их при чтении программ.

1.2.4 Указатели

Указатель - это переменная, предназначенная для хранения адреса данного определенного типа. При объявлении переменной как указателя перед именем переменной записывается символ *, а спецификатор_типа задает тип данного, адрес которого может быть значением объявленного указателя. Формат объявления указателя

спецификатор_типа * [модификатор] описатель.

Вместо спецификатора_типа можно использовать ключевое слово void. При этом такой указатель может содержать адрес любого объекта, но к такому указателю нельзя применять операции адресной арифметики, а к объекту, адрес которого содержит указатель, нельзя обратиться с помощью этого указателя. Заметим, что переменных типа void не существует.

В качестве модификаторов при объявлении указателя могут выступать ключевые слова near, far, huge; они влияют на размер переменной, объявленной как указатель, который также зависит от архитектуры компьютера и от используемой модели памяти. Указатели на одинаковые типы данных необязательно должны иметь одинаковую длину. К указателю можно применять модификатор const.

Модификатор, записанный после звездочки, относится к указателю, а записанный перед звездочкой - к адресуемому объекту. Например:

unsigned int * a; /*Переменная a представляет собой указатель на тип unsigned int (целые числа без знака).*/

double * x; /*Переменная x указывает на тип данных с плавающей точкой удвоенной точности.*/

char * buffer ; /*Объявляется указатель с именем buffer, который указывает на переменную типа char.*/

void * adr; /*Переменная adr - это указатель на тип void */

const * dr; /*Переменная dr объявлена как указатель на константное значение типа int, т.е. значение указателя может изменяться во время выполнения программы, а величина, на которую он указывает, нет.*/

unsigned char * const w = &obj. /*Переменная w объявлена как константный указатель на данные типа char unsigned. Это означает, что во время выполнения программы w будет указывать на одну и ту же область памяти. Содержание же этой области может быть изменено.*/

Значением указателя может быть не только адрес какого-либо данного, но и адрес некоторой подпрограммы. В языке СИ все подпрограммы называются функциями. Объявление указателя на функцию обязательно содержит две пары круглых скобок. Первая пара скобок содержит звездочку и имя указателя, вторая - типы и имена параметров функции, например,

void (* f1)(void);

переменная f1 - это указатель на функцию без параметров и без возвращаемого значения.

1.2.5 Массивы

Массивы - это совокупность элементов одинакового типа (базового типа, такого как double или float, или более сложного), занимающих смежные участки памяти. Из объявления массива компилятор должен получить информацию о типе элементов массива и их количестве. Объявление массива имеет формат:

спецификатор_типа описатель [константное_выражение];

Описатель - это идентификатор массива. Спецификатор_типа задает тип элементов объявляемого массива. Элементами массива не могут быть функции и элементы типа void. Константное_выражение в квадратных скобках задает количество элементов массива. Константное_выражение при объявлении массива может быть опущено (но сами скобки [] остаются), если при объявлении массив инициализируется или если массив объявлен как формальный параметр функции. Например:

int array[4]; /* array - массив из 4 элементов типа int */

double dd[10]; /* dd - масив из 10 элементов типа double */

float * ff [8] /* ff - масив из 8 указателей на float */

enum { False, True } bool [6];

/* bool - массив из 6 элементов перечислимого типа */

Для обращения к элементам массива следует указывать имя массива и индекс элемента, заключенный в квадратные скобки, например, array[2], dd[3]. Заметим, что в языке СИ первый элемент массива имеет индекс, равный 0, и обращение dd[3] обозначает обращение к четвертому элементу массива dd.

Элементы массивов могут участвовать в любых операциях как простые переменные соответствующего типа.

В языке СИ определены только одномерные массивы, но поскольку элементом массива может быть массив, можно определить и многомерные массивы. Они формализуются списком константных_выражений, следующих за идентификатором массива, причем каждое константное_выражение заключается в свои квадратные скобки.

Каждое константное_выражение в квадратных скобках определяет число элементов по данному измерению массива, так что объявление двухмерного массива содержит два константных_выражения, трехмерного - три и т.д. Например:

```
int w[2][3];          /* двумерный массив из двух строк и трех столбцов
             w[0][0] w[0][1] w[0][2]
             w[1][0] w[1][1] w[1][2] */
double f[10];  /* вектор из 10 элементов имеющих тип double */
```

Как у всякой переменной у переменной типа массив должно быть значение. В языке СИ значением переменной типа массив является адрес памяти используемой для хранения значений элементов массива. Тип значения переменной типа массив - это указатель на тип элементов массива, т.е. значение переменной f имеет тип указателя на double, значение переменной w имеет тип указателя на указатель на int. Кроме того, значение переменной типа массив нельзя изменять во время выполнения программы.

Константой типа массив символов char [] является строковая константа. Она записывается в виде последовательности символов, заключенной в кавычки. Каждый символ в строковой константе может быть представлен либо в виде символа, либо в виде специальной последовательности символов (см. табл. 2).

Следует отметить, что в строковых константах всегда обязательно задавать все три цифры в управляющей последовательности. Например, отдельную управляющую последовательность \n (переход на новую строку) можно представить как \010 или \xA, но в строковых константах необходимо задавать все три восьмеричные или две шестнадцатиричные цифры, в противном случае символ или символы, следующие за управляющей последовательностью будут рассматриваться как ее недостающая часть. Например: "ABCDE\0773GH" данная строка будет воспринята как слово ABCDE?3GH, поскольку \077 это символ ?. В случае, если указать неполную управляющую строку "ABCDE\773GH", то строка будет содержать ABCDE;GH, так как компилятор воспримет последовательность \773 как символ ; .

Отметим тот факт, что, если обратная дробная черта предшествует символу, не являющемуся управляющей последовательностью (т.е. не включенному в табл. 2) и не являющемуся цифрой, то эта черта игнорируется, а сам символ представляется как литеральный, например, символ \h представляется символом h в строковой или символьной константе.

Каждая строковая константа автоматически дополняется символом \0, помещаемым в конце строки. Например, константа "Язык СИ" состоит из восьми следующих символов Я,з,ы,к,пробел,С,И,\0.

Несмотря на то, что строковые константы - это массивы символов, значение строковой константы - не массив символов, образующих строку, а адрес памяти, начиная с которого расположены символы, образующие строку, т.е. тип значения строковой константы - это указатель на тип char.

1.2.6 Структуры

Структура - совокупность элементов, каждый из которых может иметь любой тип кроме функции. И в отличие от массива, который состоит из элементов одинакового типа,

структура может состоять из элементов разных типов. Объявление структуры похоже на объявление перечисления, но начинается со слова struct:

struct имя_тега_структуры { список_определений }
описатель [,описатель ...];

Один из компонентов объявления - либо имя_тега_структуры, либо список_определений, либо описатель - могут быть пропущены. При этом, как и для перечисления, объявляется либо тег, либо переменная типа структуры, либо и тег и переменная.

В списке_определений структуры перечисляются типы и имена элементов структуры и этот список должен содержать хотя бы один элемент. Синтаксис элемента списка_определений совпадает с синтаксисом объявления любого другого объекта. В простейшем случае - это описатели простых переменных и массивов. Примеры:

struct {double x;   float y; } s1, s2, sm[9];
struct kalendar {int year;   char moth, day; } date1, date2;

Переменные s1 и s2 объявлены как структуры, каждая из которых состоит из двух компонент x и y. Переменная sm - это массив из девяти элементов, каждый из которых является структурой. Каждая из двух переменных date1, date2 состоит из трех компонентов year, moth, day.

С помощью тега kalendar можно объявить структуру такую же как date1 и date2 и указатель на такую структуру:

struct kalendar date3, * adrdate;

Использование тегов структуры необходимо для описания рекурсивных структур. Ниже рассматривается использование рекурсивных тегов структуры:

struct node {int data;   struct node * next; } st1_node;

Тег структуры node действительно является рекурсивным, так как он используется в своем собственном описании, т.е. в формализации указателя next. Структуры не могут быть прямо рекурсивными, т.е. структура node не может содержать компоненту, являющуюся структурой node, но любая структура может иметь компоненту, являющуюся указателем на свой тип, как и сделано в приведенном примере.

Доступ к компонентам структуры осуществляется с помощью указания имени структуры и следующего через точку имени выделенного компонента, например s1.x, s2.y, date3.day, sm[3].x. В последнем примере происходит обращение к компоненте x четвертого элемента массива sm.

Другим способом обращения к элементам структур является использование указателей на структуры. В этом случае вместо точки используются знаки ->, а вместо имени структуры указатель на структуру, например, adrdate->day, adrdate->moth.

К элементам структур можно применять операции как к простым переменным соответствующего типа. Значением переменной типа структура является совокупность значений всех ее элементов, и такое значение может быть присвоено другой такой же структуре или передано функции как параметр. Других операций с типом данных структура нет.

1.2.7 Объединения (смеси)

Объединение подобно структуре, однако в каждый момент времени может использоваться (или другими словами быть ответным) только один из элементов объединения. Синтаксис объявления объединения полностью совпадает с синтаксисом объявления структуры за исключением первого слова. При объявлении объединения используется ключевое слово union:

union имя_тега_объединения { список_определений }
описатель [,описатель ...]

Главной особенностью объединения является то, что для всех элементов списка_определений выделяется одна и та же область памяти, т.е. они перекрываются. Хотя доступ к этой области памяти возможен с использованием любого из элементов, элемент для этой цели должен выбираться так, чтобы полученный результат не был бессмысленным. Доступ к элементам объединения осуществляется тем же способом, что и к структурам. Например, при объявлениях

```
union tu { long a;  int b[2];   char c[4]; } ;
union tu u1, *au, mu[5];
```

возможны обращения к элементам u1.a, u1.c[2], ua->b[1], mu[2].c[3].

Объединение применяется для следующих целей:

- инициализации используемого объекта памяти, если в каждый момент времени только один объект из многих является активным;

- интерпретации основного представления объекта одного типа, как если бы этому объекту был присвоен другой тип.

Память, которая соответствует переменной типа объединения, определяется величиной, необходимой для размещения наиболее длинного элемента объединения. Когда используется элемент меньшей длины, то переменная типа объединения может содержать неиспользуемую память. Все элементы объединения хранятся в одной и той же области памяти, начиная с одного адреса. Например:

```
union {    char fio[30];    char adres[80];
int vozrast;    int telefon; } inform;
union {    int ax;    char al[2]; } ua;
```

При использовании объекта infor типа union можно обрабатывать только тот элемент, который получил значение, т.е. после присвоения значения элементу inform.fio, не имеет смысла обращаться к другим элементам. Объединение ua позволяет получить отдельный доступ к младшему ua.al[0] и к старшему ua.al[1] байтам двухбайтного числа ua.ax .

1.2.8 Поля битов

Элементом структуры может быть битовое поле, обеспечивающее доступ к отдельным битам памяти. Вне структур битовые поля объявлять нельзя. Нельзя также организовывать массивы битовых полей и нельзя применять к полям операцию определения адреса. В общем случае тип структуры с битовым полем задается в следующем виде:

```
struct {    unsigned идентификатор_1 : длинна_поля_1;
            unsigned идентификатор_2 : длинна_поля_2;  }
```

длина поля задается целым выражением или константой, которая определяет число битов, отведенное соответствующему полю. Поле нулевой длинны обозначает выравнивание на границу следующего слова. Пример:

```
struct {    unsigned a1 : 1;    unsigned a2 : 2;
              unsigned a3 : 5;    unsigned a4 : 2;       } prim;
```

Структуры битовых полей могут содержать и знаковые компоненты. Такие компоненты автоматически размещаются на соответствующих границах слов, при этом некоторые биты слов могут оставаться неиспользованными. Обращения к битовым полям выполняются точно так же, как и компоненты общих структур, например prim.a2, prim.a3. Само же битовое поле рассматривается как целое число, максимальное значение которого определяется длиной поля.

1.2.9 Конструирование переменных с изменяемой структурой

Очень часто некоторые объекты программы относятся к одному и тому же классу, отличаясь лишь некоторыми деталями. Рассмотрим, например, представление геометрических фигур. Общая информация о фигурах может включать такие элементы, как площадь,

периметр. Однако соответствующая информация о геометрических размерах может оказаться различной в зависимости от их формы.

Рассмотрим пример, в котором информация о геометрических фигурах представляется на основе комбинированного использования структуры и объединения:

```
struct figure    { double area,perimetr;    /* общие компоненты */
                   int type;          /* признак компонента */
                   union              /* перечисление компонент */
                 { struct {   double l1,      /* сторона 1 */
                                     l2,      /* сторона 2 */
                                     l3;      /* сторона 3 */
                          } tri ;             /* треугольник */
                   struct {   double o1,      /* основание 1 */
                              o2,   /* основание 2 */
                               h;   /* высота */
                        } trap;               /* трапеция равнобокая */
                 struct {   double l1,        /* сторона 1 */
                             l2,    /* сторона 2 */
                             alf;   /* угол между сторонами */
                        } par;                /* параллелограмм */
               } geom_fig;          } fig1, fig2 ;
```

В этом случае переменные fig1 и fig2 будут состоять из трех общих компонентов - area, perimetr, type - и одного из уникальных компонентов - tri, trap, par, входящих в объединение gtom_fig. Компонент type называется меткой активного компонента, так как он используется для указания, какой из компонентов объединения geom_fig является активным в данный момент. Такая структура называется переменной структурой, потому что ее компоненты меняются в зависимости от значения метки активного компонента (значение type). Отметим, что вместо компоненты type типа int, целесообразно было бы использовать перечисляемый тип, например, такой

```
enum figure_chess {       TREYGOLNIK,   TRAPEZIA,   PARALLEL } ;  .
```

Константы TREYGOLNIK, TRAPEZIA, PARALLEL получат значения соответственно равные 0, 1, 2. Переменная type может быть объявлена как имеющая перечислимый тип:

```
enum figure_chess type;
```

В этом случае компилятор СИ предупредит программиста о потенциально ошибочных присвоениях, таких, например, как figure.type = 40; .

В общем случае переменная структуры будет состоять из трех частей: набор общих компонент, метки активного компонента и части с меняющимися компонентами. Общая форма переменной структуры имеет следующий вид:

```
struct {  общие компоненты;
          метка активного компонента;
union {     описание компоненты 1 ;
            описание компоненты 2 ;
            :::
            описание компоненты n ;
            } идентификатор-объединения ;
       } идентификатор-структуры ;
```

Пример определения переменной структуры с именем helth_record:

```
struct {    char name [25];         /* имя */
            int age;         /* возраст */
            char sex;               /* пол */
            enum merital_status ins; /* метка активного компонента */
                                    /* (семейное положение) */
            /* переменная часть */
```

```
            union {          /* холост нет компонент */
                     struct { char marripge_date[8]; /* состоит в браке */
                              char spouse_name[25];
                             int no_children;    } marriage_info;
                     char date_divorced[8]; /* разведен */
        } marital_info;      } health_record;

    enum marital_status {           SINGLE, /* холост */
                             MARRIGO, /* женат */
                             DIVOREED /* разведен */     } ;
```

Обращаться к компонентам струкруры можно при помощи сложных имен:

```
helth_record.neme,      helth_record.ins,
helth_record.marital_info.marriage_info.marriage_date ,
helth_record.marital_info.date_divorced.
```

1.2.10 Инициализация данных

При объявлении любой переменной ей можно присвоить некоторое начальное значение, присоединяя инициализатор к описателю. Инициализатор начинается со знака "=", после которого следует константное выражение или список константных выражений в фигурных скобках:

Формат 1: = константное выражение;

Формат 2: = { список константных выражений };

формат 1 используется при инициализации переменных основных типов и указателей, а формат 2 - при инициализации составных объектов: массивов, структур и более сложных объектов. Например:

char tol = 'N';   переменная tol инициализируется символом 'N';

const long megabute = (1024 * 1024); именованная константа megabute инициализируется константным выражением, после чего она не может быть изменена;

int b[2][2] = {1,2,3,4}; инициализируется двухмерный массив b целых величин, элементам массива присваиваются значения из списка; эта же инициализация может быть выполнена следующим образом:

int b[2][2] = { { 1,2 }, { 3,4 } };   .

При инициализации массива можно опустить его первую размерность, в таком случае число элементов определяется по числу используемых инициализаторов: int b[][2] = { { 1,2 }, { 3,4 } };

Если при инициализации указано меньше значений, чем элементов массива, то оставшиеся элементы инициализируются 0, т.е. при описании

int k[3][2] = { { 1,2 }, { 3, }, };

элементы первой строки получат значения 1 и 2, второй - 3 и 0, третьей - 0 и 0.

При инициализации составных объектов, нужно быть внимательным к использованию скобок и списков инициализаторов.   Например:

```
struct complex { double real,imag; } comp [2][3] =
               { { {1,1}, {2,3}, {4,5} },
               { {6,7}, {8,9}, {10,11} } };
```

в данном примере инициализируется массив структур comp из двух строк и трех столбцов, где каждая структура состоит из двух элементов real и imag.

struct complex comp2 [2][3] = { {1,2},{3,4},{5,6}, {7,8},{9,10},{11,12} };

в этом примере компилятор интерпретирует рассматриваемые фигурные скобки следующим образом:

- первая левая фигурная скобка - начало составного инициатора для массива comp2;
- вторая левая фигурная скобка - начало инициализации первой строки массива comp2[0]. Значения 1,2 присваиваются двум элементам первой структуры;
- первая правая скобка (после 2) указывает компилятору, что список инициаторов для строки массива окончен, и элементы оставшихся структур в строке comp[0] автоматически инициализируются нулем;
- аналогично список {3,4} инициализирует первую структуру в строке comp[1], а оставшиеся структуры массива обращаются в нули;
- на следующий список инициализаторов {4,5} компилятор будет сообщать о возможной ошибке, так как строка 3 в массиве comp2 отсутствует.

При инициализации объединения инициализируется первый элемент объединения в соответствии с его типом. Например:

```
union tab {     unsigned char name[10];
            int tab1;     } pers = {'A','H','T','O','H'};
```

инициализируется переменная pers.name, и так как это массив, для его инициализации требуется список значений в фигурных скобках. Первые пять элементов массива инициализируются значениями из списка, остальные нулями.

Инициализацию массива символов можно выполнить путем использования строкового литерала: char stroka[ ] = "привет";

Инициализируется массив символов из 7 элементов, последним элементом (седьмым) будет символ '\0', которым завершаются все строковые литералы. В том случае, если задается размер массива, а строковый литерал длиннее, чем размер массива, то лишние символы отбрасываются.

Следующее объявление инициализирует переменную stroka строкой, состоящей из семи элементов: char stroka[5] = "привет";

В переменную stroka попадают первые пять элементов литерала, а символы 'т' и '\0' отбрасываются. Если строка короче, чем размер массива, то оставшиеся элементы массива заполняются нулями.

Отметим, что инициализация переменной типа tab может иметь следующий вид: union tab pers1 = "Антон"; и, таким образом, в символьный массив попадут символы: 'А','Н','Т','О','Н','\0', а остальные четыре элемента будут инициализированы нулем.

Инициализацию указателей можно выполнить с помощью адресов существующих к моменту объявления данных. Например, при объявлениях

```
int q;     int *b=&q;
```

переменная b получит значение адреса переменной q;

1.2.11 Конструирование переменных сложных типов

Как уже говорилось выше, все переменные, используемые в программах на языке СИ, должны быть объявлены. Тип объявляемой переменной зависит от того, какое ключевое слово используется в качестве спецификатора типа и является ли описатель простым идентификатором или же комбинацией идентификатора с модификатором указателя (звездочка), массива (квадратные скобки) или функции (круглые скобки).

При объявлении простой переменной, структуры, смеси или объединения, а также перечисления, описатель - это простой идентификатор. Для объявления указателя, массива или функции идентификатор модифицируется соответствующим образом: звездочкой слева, квадратными или круглыми скобками справа.

Отметим важную особенность языка СИ: при объявлении можно использовать одновременно более одного модификатора, что дает возможность создавать множество различных сложных описателей типов. Однако надо помнить, что некоторые комбинации модификаторов недопустимы:

- элементами массивов не могут быть функции;

- возвращаемым значением функции не могут быть массивы или функции.

При инициализации сложных описателей квадратные и круглые скобки (справа от идентификатора) имеют приоритет перед звездочкой (слева от идентификатора). Квадратные или круглые скобки имеют один и тот же приоритет и раскрываются слева направо. Спецификатор типа рассматривается на последнем шаге, когда описатель уже полностью проинтерпретирован. Можно использовать круглые скобки, чтобы поменять порядок интерпретации на необходимый.

Для интерпретации сложных описаний предлагается простое правило, которое звучит как "изнутри наружу", и состоит из четырех шагов:

1) начать с идентификатора и посмотреть вправо, есть ли квадратные или круглые скобки;

2) если они есть, то проинтерпретировать эту часть описателя и затем посмотреть налево в поиске звездочки;

3) если на любой стадии справа встретится закрывающая круглая скобка, то вначале необходимо применить все эти правила внутри круглых скобок, а затем продолжить интерпретацию;

4) интерпретировать спецификатор типа.

Примеры:

```
char * var [6] ;    Переменная var (1) - это массив (2) из шести указателей
  4    3  1   2      (3)   на  char (4):
```

```
int (* ua) [10] ;
 4   2 1    3
```

переменная ua (1) - это указатель (2) на массив (3) из 10 элементов типа int (4).

```
long ( * mas [5] ) [6] ;
  5    3   1   2     4
```

переменная mas (1) - это массив (2) из 5 указателей (3) на массив (4) из шести элементов типа long (5).

```
int * ( * comp [10]) (int);
 6  5  3   1     2      4
```

переменная comp (1) объявляется как массив из десяти (2) указателей (3) на функции (4) с одним параметром типа int, возвращающие указатели (5) на целые значения (6).

```
char * ( * ( * var ) () ) [10];
  7  6  4  2   1     3     5
```

переменная var (1) объявлена как указатель (2) на функцию (3), возвращающую указатель (4) на массив (5) из 10 элементов, которые являются указателями (6) на значения типа char (7).

1.2.12 Объявление типов и абстрактные типы

Кроме объявлений переменных различных типов, имеется возможность объявить типы. Это можно сделать двумя способами: 1) указать имя тега при объявлении структуры, объединения или перечисления, а затем использовать это имя в объявлении переменных и функций в качестве ссылки на этот тег; 2) использовать для объявления типа ключевое слово typedef.

При объявлении с ключевым словом typedef идентификатор, стоящий на месте описываемого объекта, является именем вводимого в рассмотрение типа данных, и далее этот тип может быть использован для объявления переменных.

Отметим, что любой тип может быть объявлен с использованием ключевого слова typedef, включая типы указателя, функции или массива. Имя с ключевым словом typedef для типов указателя, структуры, объединения может быть объявлено прежде чем эти типы будут определенны, но в пределах видимости объявителя. Например,

typedef double (* MATH) (double);   MATH fun;
MATH - новое имя типа, представляющее указатель на функцию с одним параметром типа double, возвращающую значения типа double, fun - переменная типа MATH, ее можно объявить с помощью эквивалентного объявления   double (* fun) (double) .

typedef char FIO[40];   FIO person;
здесь FIO - это тип массива из сорока символов, а person - переменная такого типа.

В приведенных примерах имена типов MATH и FIO были использованы для объявления переменных. Помимо этого, имена типов могут еще использоваться в трех случаях: в объявлении функций (в списке формальных параметров и при указании типа возвращаемого значения), в операции приведения типа и в операции sizeof (операция вычисления размера памяти). В таких операциях как приведение типа и sizeof операндом может являться не какое-либо данное, а тип данных, в этом случае можно использовать абстрактный описатель типа.

Абстрактный описатель - это описатель без идентификатора, состоящий из одного или более модификаторов указателя, массива или функции. Модификатор указателя (*) всегда задается перед подразумеваемым идентификатором в описателе, а модификаторы массива [] и функции () - после него. Таким образом, чтобы правильно интерпретировать абстрактный описатель, нужно начать интерпретацию с подразумеваемого идентификатора.

Абстрактные описатели могут быть сложными. Скобки в сложных абстрактных описателях задают порядок интерпретации подобно тому, как это делалось при интерпретации сложных описателей в объявлениях, например,

int (*) [5];   int * (*) [6];   char (* [4])(void);

В первом примере задан абстрактный тип указатель на массив из пяти элементов типа int; во втором - указатель на массив из шести указателей на int; в третьем - массив из четырех указателей на функцию без параметров, возвращающую значение типа char.

1.3 ВЫРАЖЕНИЯ И ПРИСВАИВАНИЯ

1.3.1 Операнды и операции

Комбинация знаков операций и операндов, результатом которой является определенное значение, называется выражением. Знаки операций определяют действия, которые должны быть выполнены над операндами. В простейшем случае операнд - это переменная или константа. В общем случае операндом может быть выражение. Значение выражения зависит от
расположения знаков операций и круглых скобок в выражении, а также от приоритета выполнения операций. Операции с более высоким приоритетом выполняются до операций с более низким приоритетом, если часть выражения заключена в скобки, то вначале вычисляется выражение внутри скобок.

В языке СИ выражения могут входить в состав различных операторов, либо используются как самостоятельный оператор. Оператор выражение - это выражение, после которого записан символ конца оператора - точка с запятой. Действие такого оператора заключается в вычислении выражения, причем полученное значение никак не используется. Например:

```
(a + 5)/7;   /* бессмыслица */  b = 5;       /* b присвоить 5 */
t++;       /* к t прибавить 1 */  printf("\n %s",mes); /* печать сообщения */
```

Смысл использования операторов выражений заключается в действиях выполняемых во время вычисления значения (изменение значений некоторых переменных, вызов функций и т.п.). В частности, первый из приведенных операторов не имеет смысла, но тем не менее и сложение и деление будут выполнены, второй и третий операторы изменяют значение переменных b и t, третий оператор выполняет функцию вывода сообщения на экран.

В зависимости от используемых операций выражения различают на первичные, унарные, бинарные, тернарные, выражения присваивания и выражения приведения типа. Список операций языка СИ приведен в табл. 5.

Таблица 5

Знак операции	Операция	Тип операции	Приоритет	Порядок выполнения
1	2	3	4	5
( ) [ ] . и –>	Вызов функции Индексная Выбор элемента	Первичная	1	Слева направо
– ! ~ * & ++ –– (тип) sizeof	Отрицание арифметическое Отрицание логическое Двоичное дополнение Косвенная адресация Вычисление адреса Инкремент Декремент Приведение типа Вычисление Размера	Унарная	2	Слева направо
* / %	Умножение Деление Остаток	Мультипликативные	3	Слева направо
+ –	Сложение Вычитание	Аддитивные	4	Слева направо
>> <<	Сдвиг вправо Сдвиг влево	Сдвиги	5	
> < >= <= == !=	Больше Меньше Больше или равно Меньше или равно Равно Не равно	Отношения	6 6 6 6 7 7	
& ^ |	Поразрядное И Поразрядное исключающее ИЛИ Поразрядное ИЛИ	Поразрядные	8 9 10	
&&	Логическое И	Логичес	11	

Продолжение табл. 5

<table>
<tr><td>||</td><td>Логическое ИЛИ</td><td>кие</td><td>12</td><td></td></tr>
<tr><td>? :</td><td>Условная</td><td>Условна
я</td><td>13</td><td></td></tr>
<tr><td>=</td><td>Присваивание</td><td>Присваи
вание</td><td>14</td><td>Справа
налево</td></tr>
<tr><td>*=
/=
%=
+=
–=
&=
|=
>>=
<<=
^=</td><td>Составное
присваивание</td><td>Присваи
вание</td><td>14</td><td>Справа
налево</td></tr>
<tr><td>,</td><td>Последовательное
вычисление</td><td>Последо
вательно
е
вычисле
ние</td><td>15</td><td>Справа
налево</td></tr>
</table>

К первичным выражениям относятся вызовы функций, индексные выражения и выбор элементов. Унарная операция имеет один операнд, бинарная - два, тернарная - три. В языке СИ реализовано несколько различных операций присваивания, и все они являются выражениями; значением выражения присваивания является величина, которая присваивается.

При вычислении выражений тип каждого операнда может быть предварительно преобразован к другому типу, в соответствии с требованиями выполняемой операции. Такое преобразование называется преобразованием по умолчанию. При выполнении операции присваивания тип результата должен быть преобразован к типу первого операнда, это может быть сделано неявно или явно с использованием операций приведения типов.

Операнд - это константа, литерал, идентификатор, вызов функции, индексное выражение, выражение выбора элемента или более сложное выражение, сформированное комбинацией операндов, знаков операций и круглых скобок. Операнд, имеющий константное значение, называется константным выражением. Каждый операнд имеет тип.

Если в качестве операнда используется константа, то ему соответствует значение и тип представляющей его константе. Напомним, что целая константа может иметь тип int или long и быть знаковой (signed) или беззнаковой (unsigned). Символьная константа имеет тип int. Константа с плавающей точкой всегда имеет тип double. Строковая константа состоит из последовательности символов, заключенных в кавычки, и представляется в памяти как массив элементов типа char, инициализируемый указанной последовательностью символов с добавлением в конце символа \0. Значением строковой константы является адрес первого элемента строки и синтаксически строковый литерал является немодифицируемым указателем на тип char. Константы и константные выражения не могут быть первым операнд дом операций присваивания.

Если операндом является идентификатор переменной, то значением операнда является значение соответствующей переменной. Заметим, что для простой переменной, указателя, структуры, объединения и перечисления имя переменной обозначает значение этой переменной, а для массивов и функций имя переменной - это адрес массива или функции, и поскольку это значение невозможно изменить, то имя массива или функции не может быть первым операндом операции присваивания.

Порядок вычисления сложных выражений

Значением вызова функции является возвращаемое значение этой функции. Вызов функции состоит из имени функции (или выражения, значением которого является адрес функции) и списка выражений в круглых скобках. Например, sin(x), sqrt(3), pow(x,y). Круглые скобки при вызове функции нельзя опускать, даже если список выражений передаваемых функции пуст. Например, clrscr() или kbhit().

Индексное выражение задает элемент массива и имеет вид

выражение_1 [выражение_2]

Тип индексного выражения является типом элементов массива, а значение представляет величину, адрес которой вычисляется с помощью значений выражение_1 и выражение_2.

Обычно выражение_1 - это идентификатор массива, а выражение_2 - это целая величина, задающая номер элемента в массиве, но необязательно. Синтаксис языка требует, чтобы одно из выражений имело тип указателя, а второе - целочисленной величины, и в любом случае в квадратные скобки заключается второй по следованию операнд индексного выражения. Например, пусть объявлены массив и указатель int array[4], *adr = array ;. Тогда к третьему элементу массива array можно обращаться с помощью следующих индексных выражений: array[2], 2[array], 2[adr], adr[2].

Выражение с несколькими индексами ссылается на элементы многомерных массивов. Многомерный массив - это массив, элементами которого являются массивы. Например, первым элементом трехмерного массива является массив с двумя измерениями.

Для обращения к элементу многомерного массива индексное выражение должно иметь несколько индексов, заключенных в квадратные скобки:

выражение-1 [выражение-2][выражение-3] ...

Такое индексное выражение интерпретируется слева направо, т.е. вначале рассматривается первое индексное выражение: выражение-1 [выражение-2]. Результатом этого выражения должно быть адресное выражение, с которым складывается выражение-3 и т.д. Например,

int mass [2][5][3];

Рассмотрим процесс вычисления индексного выражения mass[1][2][3]:

1 Вычисляется выражение mass[1]. Значение индекса 1 умножается на размер элемента этого массива, элементом же этого массива является двухмерный массив, содержащий 5 × 3 элементов, имеющих тип int. Получаемое значение складывается со значением указателя mass. Результатом является указатель на второй двухмерный массив размером 5 × 3 в трехмерном массиве mass.

2 Второй индекс 2 умножается на размер массива из трех элементов типа int и складывается с адресом, соответствующим mass [1]. При этом получаем адрес одномерного массива mass[1][2] из трех элементов типа int.

3 Так как каждый элемент трехмерного массива - это величина типа int, то индекс 3 умножается на размер типа int перед сложением с адресом mass [1][2].

4 Наконец, выполняется разадресация полученного указателя. Результирующим выражением будет элемент типа int.

Если было бы указано mass [1][2], то результатом был бы указатель на массив из трех элементов типа int. Соответственно значением индексного выражения mass[1] является указатель на двухмерный массив.

Выражение выбора элемента применяется, если в качестве операнда надо использовать элемент структуры или объединения. Такое выражение имеет значение и тип выбранного элемента. Рассмотрим две формы выражения выбора элемента:

выражение.идентификатор , выражение->идентификатор .

В первой форме выражение представляет величину типа struct или union, а идентификатор - это имя элемента структуры или объединения. Во второй форме выражение должно иметь значение адреса структуры или объединения, а идентификатор – имя выбираемого элемента структуры или объединения. Обе формы выражения выбора элемента дают одинаковый результат. Действительно, запись, включающая знак операции выбора (->), является сокращенной версией записи с точкой для случая, когда выражению, стоящему перед точкой, предшествует операция разадресации (*), примененная к указателю, т.е. запись выражение -> идентификатор эквивалентна записи (* выражение) . идентификатор в случае, если выражение является указателем. Например:

```
struct tree  {  float     num;  int  spisoc[5];  struct tree  *left; } tr[5] , elem ;
elem.left = & elem;
```

В приведенном примере используется операция выбора (.) для доступа к элементу left структурной переменной elem. Таким образом элементу left структурной переменной elem присваивается адрес самой переменной elem, т.е. переменная elem хранит ссылку на себя саму.

Приведение типов - это изменение (преобразование) типа объекта. Для выполнения явного преобразования необходимо перед объектом записать в скобках нужный тип:

(имя-типа) операнд.

Приведение типов используется для преобразования объектов одного скалярного типа в другой скалярный тип, причем результат такой операции может быть отличен от значения операнда. Например,

```
int i;    bouble x;    b = (double)i+2.0;
```

в этом примере целая переменная i с помощью операции приведения типов приводится к плавающему типу, а затем уже участвует в вычислении выражения.

Выражения со знаками операций могут участвовать в выражениях как операнды, быть унарными (с одним операндом), бинарными (с двумя операндами) и тернарными (с тремя операндами).

Круглые скобки в выражениях изменяют последовательность выполнения операций таким образом, что вначале будет вычисляться выражение, заключенное в скобки. Скобки могут быть вложенными, причем уровень вложенности скобок не ограничен.

Унарное выражение состоит из операнда и предшествующего ему знаку унарной операции и имеет следующий формат:

знак_унарной_операции операнд .

В языке СИ имеются следующие унарные операции:

- арифметическае отрицание (отрицание и дополнение); ~ побитовое логическое отрицание (дополнение); ! логическое отрицание; * разадресация (косвенная адресация); & вычисление адреса; ++ увеличение (инкремент); -- уменьшение (декремент); sizeof вычисление размера.

Унарные операции выполняются справа налево.

Операции увеличения и уменьшения увеличивают или уменьшают значение операнда на единицу и могут быть записаны как справа, так и слева от операнда. Если знак операции записан перед операндом (префиксная форма), то изменение операнда происходит до его использования в выражении. Если знак операции записан после операнда (постфиксная форма), то операнд вначале используется в выражении, а затем происходит его изменение.

Бинарное выражение состоит из двух операндов, разделенных знаком бинарной операции: операнд_1 знак_бинарной_операции операнд_2 .

К бинарным операциям относятся: аддитивные и мультипликативные: операции сдвига и отношения, поразрядные и логические, операции последовательного вычисления и присваивания.

Бинарные операции выполняются слева направо, за исключением присваиваний, которые выполняются справа налево.

Левый операнд операции присваивания должен быть выражением, ссылающимся на область памяти (но не объектом, объявленным с ключевым словом const), такие выражения называются леводопустимыми и к ним относятся:

- идентификаторы переменных целого и плавающего типов, типов указателя, структуры, объединения;
- индексные выражения, исключая выражения, имеющие тип массива или функции;
- выражения выбора элемента -> и . , если выбранный элемент является леводопустимым;
- выражения унарной операции разадресации * , за исключением выражений, ссылающихся на массив или функцию;
- выражение приведения типа, если результирующий тип не превышает размера первоначального типа.

Тернарное выражение состоит из трех операндов, разделенных знаками тернарной операции ? и : , и имеет формат

операнд_1 ? операнд_2 : операнд_3.

При записи выражений следует помнить, что символы *,&,! могут обозначать унарную или бинарную операцию.

1.3.2 Преобразования при вычислении арифметических выражений

При выполнении большинства бинарных операций производится автоматическое преобразование типов, чтобы привести операнды выражений к общему типу или чтобы расширить короткие величины до размера целых величин, используемых в машинных командах. Выполнение преобразования зависит от специфики операций и от типа операндов. Рассмотрим общие арифметические преобразования:

1 Операнды типа float преобразуются к типу double.
2 Если один операнд double, то второй также преобразуется к типу double.
3 Любые операнды типа char и short преобразуются к типу int.
4 Любые операнды unsigned char или unsigned short преобразуются к типу unsigned int.
5 Если один операнд типа unsigned long, то второй преобразуется к типу unsigned long.
6 Если один операнд типа long, то второй преобразуется к типу long.
7 Если один операнд типа unsigned int, то второй операнд преобразуется к этому же типу.

Таким образом, можно отметить, что при вычислении выражений операнды преобразуются к типу того операнда, который имеет наибольший размер. Например,

```
double   ft,sd;     unsigned char ch;     unsigned long in;     int   i;
....
sd = ft*(i + ch/in);
```

При выполнении оператора присваивания правила преобразования будут использоваться следующим образом. Операнд ch преобразуется к unsigned int (правило 4). Затем он преобразуется к типу unsigned long (правило 5). По этому же правилу i преобразуется к unsigned long и результат операции, заключенной в круглые скобки будет иметь тип unsigned long. Затем он преобразуется к типу double (правило 2) и результат всего выражения будет иметь тип double.

1.3.3 Операции отрицания и дополнения

Операция арифметического отрицания (-) формирует отрицание своего операнда. Операнд должен быть целой или плавающей величиной. При выполнении осуществляются обычные арифметические преобразования. Например:

```
double u = 5;
u = -u;     /* переменной u присваивается ее отрицание,
```

т.е. u принимает значение -5 */

Если операнд отрицания беззнаковый, то выполнение операции не приводит к прерыванию программы, а результатом операции -x будет значение 2^n - x, где n длина операнда в битах. Например, если значение переменной t типа unsigned short равно 10, то значением выражения -t будет 65526 (т.е. 65536-10).

Операция логического отрицания "НЕ" (!) вырабатывает значение 0 (ложь), если операнд отличен от нуля (истина), и значение 1 (истина), если операнд равен нулю. Результат имеет тип int. Операнд должен быть целого или плавающего типа или типа указатель. Например, int t, z=0; t=!z; переменная t получит значение, равное 1, так как переменная z имела значение, равное 0 (ложно).

Операция двоичного дополнения (~) вырабатывает двоичное дополнение своего операнда. Операнд должен быть целого типа. Осуществляется обычное арифметическое преобразование, результат имеет тип операнда после преобразования. Например, unsigned char f='9', r; r = ~f;

Двоичное представление символа '9' равно 00111001. В результате операции ~f будет получено двоичное значение 11000110, что соответствует символу 'ц'.

1.3.4 Операции разадресации и адреса

Эти операции используются для работы с переменными типа указатель. Операция разадресации * осуществляет косвенный доступ к адресуемой величине через указатель. Операнд должен быть указателем. Результатом операции является величина, на которую указывает операнд. Типом результата является тип величины, адресуемой указателем. Результат не определен, если указатель содержит нулевое значение или значение не выровненное на границу адресуемого объекта (значение указателя на short должно указывать на четный адрес, значение указателей на long и на float должно делиться на 4 и т.п.). При выполнении разадресации контроль за существованием какого либо объекта по указанному адресу осуществляется только программистом и некорректное использование этой операции может привести к непредсказуемым последствиям. Например,

```
int * s;        *s=7;
```

в примере объявлен указатель s, но выполнение присваивания *s=7 приводит к непредсказуемым последствиям, поскольку значение указателя s к моменту выполнения присваивания не определено.

Операция адрес & формирует адрес своего операнда. Операндом может быть любая переменная или элемент массива, структуры или объединения. Имя функции или массива также может быть операндом операции адрес, хотя в этом случае операция лишена смысла, так как имена массивов и функций сами по себе являются адресами. Результат операции имеет тип указателя на тип операнда.

Операция адрес не может применяться к элементам структуры, являющимися полями битов, и к объектам с классом памяти register.

Примеры:

```
  int t, f=56, * adress;
adress = &t; /* переменной adress, объявленной как указатель, присваивается */
 /* адрес переменной t                */
* adress = f;  /* переменной находящейся по адресу, содержащемуся в переменной */
 /* adress, присваивается значение переменной f, т.е. 56 , что */
 /* эквивалентно  t=f; или  t=56;            */
```

1.3.5 Операция sizeof

Операция sizeof вычисляет размер памяти занимаемой операндом. Операция sizeof имеет следующий формат:

sizeof(выражение) .

В качестве выражения может быть использован любой идентификатор, либо имя типа. Отметим, что нельзя использовать тип void, а идентификатор не может быть именем битового поля или функции. Если в качестве выражения указанно имя массива, то результатом является размер всего массива (т.е. произведение числа элементов на длину типа), а не размер указателя, соответствующего идентификатору массива. Когда sizeof применяются к имени типа структуры или объединения или к идентификатору имеющему тип структуры или объединения, то результатом является фактический размер структуры или объединения, который может включать участки памяти, используемые для выравнивания элементов структуры или объединения. Таким образом, этот результат может не соответствовать размеру, получаемому путем сложения размеров элементов структуры. Например,

```
struct  { char  h;  int  b;    double f;  } str;
int a1;    a1 = sizeof(str);
```

переменная a1 получит значение, равное 12, в то же время, если сложить длины всех используемых в структуре типов, то получим, что длина структуры str равна 7.

1.3.6 Мультипликативные операции

К классу мультипликативных операций относятся операции умножения (*), деления (/) и получение остатка от деления (%). Операндами операции (%) должны быть целые числа. Отметим, что типы операндов операций умножения и деления могут отличаться, и для них справедливы правила преобразования типов. Типом результата является тип операндов после преобразования. Операция умножения (*) выполняет умножение операндов. Например:

```
int i=5;    float f=0.2;    double g,z;
g=f*i;
```

Тип произведения i и f преобразуется к типу double, затем результат присваивается переменной g. Операция деления (/) выполняет деление первого операнда на второй. Если две целые величины не делятся нацело, то результат округляется в сторону нуля. При попытке деления на ноль выдается сообщение во время выполнения.

```
int i=49, j=10;    float  n, m;
n = i/j;    /* результат 4 так как оба операнда целые */
m = i/10.0;  /* результат 4.9 потому что второй операнд */
                /* имеет тип double */
```

Операция остаток от деления (%) дает остаток от деления первого операнда на второй. Знак результата зависит от конкретной реализации, но в большинстве случаев совпадает со знаком делимого. Если второй операнд равен нулю, то выдается сообщение об ошибке:

```
int n = 49, m = 10, i, j;
i = n % m;    /* результат  9  */    j = n % 8;    /* результат  1  */
```

1.3.7 Аддитивные операции

К аддитивным операциям относятся сложение + и вычитание -. Операнды могут быть целого или плавающего типов. В некоторых случаях аддитивные операции могут выполняться над указателями. Над операндами аддитивных операций выполняются общие арифметические преобразования. Однако эти преобразования не обеспечивают обработку ситуаций переполнения и потери значимости. Информация теряется, если результат

аддитивной операции не может быть представлен типом операндов после преобразования. При этом сообщение об ошибке не выдается. Например,

```
int  i=30000, j=30000, k;            k=i+j;
```

в результате сложения k получит значение, равное –5536, потому что сумма 60000 не может быть значением типа int. Операндами операции сложения могут быть указатель и целое число. При этом целое число предварительно умножается на размер адресуемого указателем объекта. В результате получается указатель, адресующий область памяти, расположенную на целое число объектов дальше от первоначального указателя.

Операция вычитания (-) вычитает второй операнд из первого. Возможна следующая комбинация операндов:

- оба операнда целого или плавающего типа;
- оба операнда являются указателями на один и тот же тип;
- первый операнд является указателем, а второй - целым.

При вычитании из указателя целого числа производится такое же масштабирование, что и при сложении, и в результате получается адрес сдвинутый на целое число объектов перед первоначальным указателем.

При вычитании одного указателя из другого получается число объектов данного типа, которые можно разместить между двумя указателями. Например:

```
double d[10],* u;      int i;
u = d+2;   /* u указывает на третий элемент массива */
i = u-d;     /* i принимает значение равное 2  */
```

Отметим, что операции сложения и вычитания над адресами в единицах отличных от длины типа, могут привести к непредсказуемым результатам. В фрагменте

```
float h[7];   int * w;
h[0]=0;   w=h;   w++;   *w=3000;
```

значение указателя w после операции w++ не будет равно адресу элемента h[1], а будет указывать на адрес внутри поля h[0]. Поэтому присваивание *w=3000 изменит значение h[0] на значение -6.338253e+31.

1.3.8 Операции сдвига

Операции сдвига осуществляют смещение операнда влево (<<) или вправо (>>) на число битов, задаваемое вторым операндом. Оба операнда должны быть целыми величинами. Выполняются обычные арифметические преобразования. При сдвиге влево правые освобождающиеся биты устанавливаются в нуль. При сдвиге вправо метод заполнения освобождающихся левых битов зависит от типа первого операнда. Если тип unsigned, то свободные левые биты устанавливаются в нуль. В противном случае они заполняются копией знакового бита. Результат операции сдвига не определен, если второй операнд отрицательный.

Преобразования, выполненные операциями сдвига, не обеспечивают обработку ситуаций переполнения и потери значимости. Информация теряется, если результат операции сдвига не может быть представлен типом первого операнда после преобразования. Отметим, что сдвиг влево соответствует умножению первого операнда на степень числа 2, равную второму операнду, а сдвиг вправо соответствует делению первого операнда на 2 в степени, равной второму операнду. Например:

```
int i=0x1234, j, k ;
k = i<<4 ;     /*  k = 0x0234  */   j = i<<8 ;     /*  j = 0x3400  */
i = j>>8 ;     /*  i = 0x0034  */
```

1.3.9 Поразрядные операции

К поразрядным операциям относятся: операция поразрядного логического "И" (&), операция поразрядного логического "ИЛИ" (|), операция поразрядного "исключающего ИЛИ" (^). Операнды поразрядных операций могут быть любого целого типа. При необходимости над операндами выполняются преобразования по умолчанию, тип результата - это тип операндов после преобразования.

Операция поразрядного логического И (&) сравнивает каждый бит первого операнда с соответствующим битом второго операнда. Если оба сравниваемых бита единицы, то соответствующий бит результата устанавливается в 1, в противном случае в 0.

Операция поразрядного логического ИЛИ (|), сравнивает каждый бит первого операнда с соответствующим битом второго операнда. Если любой (или оба) из сравниваемых битов равен 1, то соответствующий бит результата устанавливается в 1, в противном случае результирующий бит равен 0.

Операция поразрядного исключающего ИЛИ (^) сравнивает каждый бит первого операнда с соответствующими битами второго операнда. Если один из сравниваемых битов равен 0, а второй бит равен 1, то соответствующий бит результата устанавливается в 1, в противном случае, т.е. когда оба бита равны 1 или 0, бит результата устанавливается в 0. Например:

```
int i=0x45FF,   /*   i= 0100 0101 1111 1111  */
j=0x00FF;       j= 0000 0000 1111 1111  */
char r;          r = i^j;   /* r=0x4500 = 0100 0101 0000 0000  */
               r = i|j;   /* r=0x45FF = 0100 0101 0000 0000  */
               r = i&j;   /* r=0x00FF = 0000 0000 1111 1111  */
```

1.3.10 Логические операции

К логическим операциям относятся операция логического И (&&) и операция логического ИЛИ (||). Операнды логических операций могут быть целого типа, плавающего или указателя, при этом в каждой операции могут участвовать операнды различных типов. Операнды логических выражений вычисляются слева направо. Если значения первого операнда достаточно, чтобы определить результат операции, то второй операнд не вычисляется. Логические операции не вызывают стандартных арифметических преобразований. Они оценивают каждый операнд с точки зрения его эквивалентности нулю. Результатом логической операции является 0 или 1, тип результата int.

Операция логического И (&&) вырабатывает значение 1, если оба операнда имеют нулевые значения. Если один из операндов равен 0, то результат также равен 0. Если значение первого операнда равно 0, то второй операнд не вычисляется.

Операция логического ИЛИ (||) выполняет над операндами операцию включающего ИЛИ. Она вырабатывает значение 0, если оба операнда имеют значение 0, если какой-либо из операндов имеет ненулевое значение, то результат операции равен 1. Если первый операнд имеет ненулевое значение, то второй операнд не вычисляется.

1.3.11 Операция последовательного вычисления

Операция последовательного вычисления обозначается запятой (,) и вычисляет свои операнды слева направо. При выполнении операции последовательного вычисления, преобразование типов не производится. Операнды могут быть любых типов. Результат операции имеет значение и тип второго операнда. Эту операцию можно использовать для вычисления двух и более выражений там, где по синтаксису допустимо только одно выражение. Отметим, что запятая может использоваться также как символ разделитель, поэтому необходимо по контексту различать, запятую, используемую в качестве разделителя и в качестве знака операции.

1.3.12 Условная операция

В языке СИ имеется одна тернарная операция - условная операция, которая имеет следующий формат: операнд_1 ? операнд_2 : операнд_3

Операнд_1 должен быть целого или плавающего типа или быть указателем. Он оценивается с точки зрения его эквивалентности 0. Если операнд_1 не равен 0, то вычисляется операнд_2 и его значение является результатом операции. Если операнд_1 равен 0, то вычисляется операнд_3 и его значение является результатом операции. Следует отметить, что вычисляется либо операнд_2, либо операнд_3, но не оба.

Тип результата зависит от типов операнда_2 и операнда_3 . Если их типы одинаковы, то и результат будет такого же типа, если типы операнда_2 и операнда_3 различны, то тип результата определяется следующим образом.

1 Если операнд_2 или операнд_3 имеет целый или плавающий тип, то выполняются обычные арифметические преобразования. Типом результата является тип операндов после преобразования.

2 Если один операнд является указателем на объект любого типа, а другой операнд является указателем на vold, то указатель на объект преобразуется к указателю на vold, который и будет типом результата.

3 Если один из операндов является указателем, а другой константным выражением со значением 0, то типом результата будет тип указателя.

Пример: max = (d<=b) ? b : d; Переменной max присваивается максимальное значение переменных d и b.

1.3.13 Простое присваивание

Операция простого присваивания используется для замены значения левого операнда, значением правого операнда. При присваивании производится преобразование типа правого операнда к типу левого операнда по правилам, упомянутым раньше. Левый операнд должен быть модифицируемым. Например:

```
int t;    char f;    long z;
t=f+z;
```

значение переменной f преобразуется к типу long, вычисляется f+z,, результат преобразуется к типу int и затем присваивается переменной t.

Заметим, что кроме замены значения переменной, присваивание является операцией, результатом которой является присвоенное значение и эту операцию можно использовать в качестве операнда в других операциях. т.е. в одном выражении можно выполнить несколько присваиваний. Например,

```
f=t=8;    s=f+(g=t*4);
```

в первом выражении вначале произойдет присваивание t=8, а затем результат этой операции, т.е. число 8, будет участвовать во второй операции присваивания и переменная f тоже получит значение 8. Во втором выражении вначале будет выполнено присваивание g=t*4, в результате чего будет получено значение 32, а затем это значение будет участвовать в операции сложения и в присваивании. В результате переменная s получит значение 40.

1.3.14 Операции увеличения и уменьшения

Операции увеличения (++) и уменьшения (--) являются унарными операциями присваивания. Они соответственно увеличивают или уменьшают значения операнда на единицу. Операнд может быть целого или плавающего типа или типа указатель (кроме указателя на void) и должен быть модифицируемым. Операнд целого или плавающего типа увеличиваются (уменьшаются) на единицу. Тип результата соответствует типу операнда. Операнд адресного типа увеличивается или уменьшается на размер объекта, который он адресует. В языке допускается префиксная или постфиксная формы операций увеличения (уменьшения), поэтому значения выражения, использующего операции увеличения

(уменьшения) зависит от того, какая из форм указанных операций используется. Если знак операции стоит перед операндом (префиксная форма записи), то изменение операнда происходит до его использования в выражении и результатом операции является увеличенное или уменьшенное значение операнда. В том случае если знак операции стоит после операнда (постфиксная форма записи), то операнд вначале используется для вычисления выражения, а затем происходит изменение операнда. Например,

int t=1, s=5, z, f;   z=(t++)*5;

вначале происходит умножение t*5 , а затем увеличение t. В результате получится z=5, t=2.

f=(++s)/3;

вначале значение s увеличивается, а затем используется в операции сложения. В результате получим s=6, f=2.

В случае, если операции увеличения и уменьшения используются как самостоятельные операторы, префиксная и постфиксная формы записи становятся эквивалентными: z++; /* эквивалентно */ ++z; .

1.3.15 Составное присваивание

Кроме простого присваивания, имеется целая группа операций присваивания, которые объединяют простое присваивание с одной из бинарных операций. Такие операции называются составными операциями присваивания и имеют вид: операнд_1 бинарная_операция = операнд_2

Составное присваивание по результату эквивалентно следующему простому присваиванию: операнд_1 = операнд_1 бинарная_операция операнд_2

Отметим, что выражение составного присваивания с точки зрения реализации неэквивалентно простому присваиванию, так как в последнем операнд_1, вычисляется дважды. Каждая операция составного присваивания выполняет преобразования, которые осуществляются соответствующей бинарной операцией. Левым операндом операций += и -= может быть указатель, в то время как правый операнд должен быть целым числом. Например:

```
double arr[4];   double b=3.0;
b+=arr[2];       /* эквивалентно  b=b+arr[2]       */
arr[b*3]/=b+1; /* эквивалентно  arr[b*3]=arr[b*3]/(b+1)  */
```

Заметим, что при втором присваивании использование составного присваивания дает более заметный выигрыш во времени выполнения, так как левый операнд является индексным выражением.

1.3.16 Побочные эффекты

Операции присваивания в сложных выражениях могут вызывать побочные эффекты, если они изменяют значение переменных, входящих в это выражение. Порядок вычисления операндов некоторых операций зависит от реализации, и поэтому могут возникать разные побочные эффекты, если в одном из операндов используется операция увеличения или уменьшения, а также другие операции присваивания. Например, выражение i*j+(j++)+(--i) может принимать различные значения при обработке разными компиляторами. Чтобы избежать недоразумений, вызываемых побочными эффектами, не нужно использовать операции присваивания переменной, если эта переменная используется в выражении более одного раза.

Побочный эффект может возникать и при вызове функции, если в параметрах содержится прямое или косвенное присваивание (через указатель). Это связано с тем, что аргументы функции могут вычисляться в любом порядке. Например, побочный эффект имеет место в следующем вызове функции: prog (a,a=k*2); .

В зависимости от того, какой аргумент вычисляется первым, в функцию могут быть переданы различные значения. Во избежание недоразумений не нужно использовать

операции присваивания переменной в вызове функции, если эта переменная участвует в формировании других аргументов функции.

1.3.18 Преобразование типов

При выполнении операций происходят неявные преобразования типов: а) при выполнении операций осуществляются обычные арифметические преобразования (которые были рассмотрены выше); б) при выполнении операций присваивания, если значение одного типа присваивается переменной другого типа; в) при передаче аргументов функции. Кроме того, в СИ есть возможность явного приведения значения одного типа к другому. В операциях присваивания тип значения, которое присваивается, преобразуется к типу переменной, получающей это значение. Допускается преобразование целых и плавающих типов, даже если оно ведет к потере информации.

Преобразование целых типов со знаком. Целое со знаком преобразуется к более короткому целому со знаком посредством усечения старших битов. Целое со знаком преобразуется к более длинному целому со знаком, путем размножения знака. При преобразовании целого со знаком к целому без знака целое со знаком преобразуется к размеру целого без знака и результат рассматривается как значение без знака. Преобразование целого со знаком к плавающему типу происходит без потери информации, за исключением случая преобразования значения типа long int или unsigned long int к типу float, когда точность часто может быть потеряна.

Преобразование целых типов без знака. Целое без знака преобразуется к более короткому целому без знака или со знаком путем усечения старших битов. Целое без знака преобразуется к более длинному целому без знака или со знаком путем дополнения нулей слева. Когда целое без знака преобразуется к целому со знаком того же размера, битовое представление не изменяется. Поэтому значение, которое оно представляет, изменяется, если знаковый бит установлен (равен 1), т.е. когда исходное целое без знака больше чем максимальное положительное целое со знаком, такой же длины. Целые значения без знака преобразуются к плавающему типу, путем преобразования целого без знака к значению типа signed long, а затем значение signed long преобразуется в плавающий тип. Преобразования из unsigned long к типу float, double или long double производятся с потерей информации, если преобразуемое значение больше, чем максимальное положительное значение, которое может быть представлено для типа long.

Преобразования плавающих типов. Величины типа float преобразуются к типу double без изменения значения. Величины double и long double преобразуются к float с некоторой потерей точности. Если значение слишком велико для float, то происходит потеря значимости, о чем сообщается во время выполнения. При преобразовании величины с плавающей точкой к целым типам она сначала преобразуется к типу long (дробная часть плавающей величины при этом отбрасывается), а затем величина типа long преобразуется к требуемому целому типу. Если значение слишком велико для long, то результат преобразования неопределен. Преобразование из float, double или long double к типу unsigned long производится с потерей точности, если преобразуемое значение больше, чем максимально возможное положительное значение, представленное типом long.

Преобразование типов указателя. Указатель на некоторый тип может быть преобразован к указателю на другой тип. Однако результат может быть не определен из-за отличий в требованиях к выравниванию и размерах для различных типов.

Указатель на тип void может быть преобразован к указателю на любой тип, и указатель на любой тип может быть преобразован к указателю на тип void без ограничений. Значение указателя может быть преобразовано к целой величине. Метод преобразования зависит от размера указателя и размера целого типа: а) если размер указателя меньше размера целого типа или равен ему, то указатель преобразуется точно так же, как целое без

знака; б) если указатель больше, чем размер целого типа, то указатель сначала преобразуется к указателю с тем же размером, что и целый тип, и затем преобразуется к целому типу.

Целый тип может быть преобразован к адресному типу: а) если целый тип того же размера, что и указатель, то целая величина просто рассматривается как указатель (целое без знака); б) если размер целого типа отличен от размера указателя, то целый тип сначала преобразуется к размеру указателя (используются способы преобразования, описанные выше), а затем полученное значение трактуется как указатель.

Преобразования при вызове функции. Преобразования, выполняемые над аргументами при вызове функции, зависят от того, был ли задан прототип функции (объявление "вперед") со списком объявлений типов аргументов. Если задан прототип функции и он включает объявление типов аргументов, то над аргументами в вызове функции выполняются только обычные арифметические преобразования. Эти преобразования выполняются независимо для каждого аргумента. Величины типа float преобразуются к double, величины типа char и short преобразуются к int, величины типов unsigned char и unsigned short преобразуются к unsigned int. Могут быть также выполнены неявные преобразования переменных типа указатель. Задавая прототипы функций, можно переопределить эти неявные преобразования и позволить компилятору выполнить контроль типов.

Преобразования при приведении типов. Явное преобразование типов, может быть осуществлено посредством операции приведения типов, которая имеет формат: (имя-типа) операнд. В приведенной записи имя-типа задает тип, к которому должен быть преобразован операнд. Например:

```
int  i=2;    long  l=2;    double  d;    float   f;
d=(double)i * (double)l;     f=(float)d;
```

значения величин i, l, d будут явно преобразованы к указанным в круглых скобках типам. Заметим, что ни тип, ни значение самих переменных i,j,d при этом не изменяются.

1.4 ОПЕРАТОРЫ

Операторы языка программирования управляют процессом выполнения программы. В составе языка СИ имеются операторы обеспечивающие выполнение всего набора конструкций структурного программирования. Все операторы языка СИ могут быть условно разделены на следующие категории:

- условные операторы, к которым относятся оператор условия if и оператор выбора switch;
- операторы цикла (for,while,do while);
- операторы перехода (break, continue, return, goto);
- другие операторы (оператор выражение, пустой оператор).

Операторы в программе могут объединяться в составные операторы с помощью фигурных скобок. Любой оператор в программе может быть помечен меткой, состоящей из имени и следующего за ним двоеточия. Все операторы языка СИ, кроме составных операторов, заканчиваются точкой с запятой ; .

1.4.1 Оператор выражение

Любое выражение, которое заканчивается точкой с запятой, является оператором выражением. Выполнение оператора выражения заключается в вычислении выражения. Полученное значение выражения никак не используется, поэтому, как правило, такие выражения вызывают побочные эффекты. Заметим, что вызов функции не возвращающей значения можно осуществить только при помощи оператора выражения. Например:

```
clrscr(); /* - вызов функции */
a=b+c; /* - выражения i++; /* с присваиванием */
```

1.4.2 Пустой оператор

Пустой оператор состоит только из точки с запятой. При выполнении этого оператора ничего не происходит. Он необходим для двух случаев. Во-первых, он используется в составе других операторов в том месте, где по синтаксису необходим оператор, но по смыслу программы он не требуется; во-вторых, при необходимости пометить закрывающуюся фигурную скобку.

Синтаксис языка СИ требует, чтобы после метки обязательно следовал оператор. Фигурная же скобка оператором не является. Поэтому, если надо передать управление на фигурную скобку, необходимо использовать пустой оператор. Например:

```
int main ( )
{…  { if (...) goto a;        /* переход на скобку */
            { ... }
     a:; }                    /* помеченый пустой оператор */
return 0;  }
```

1.4.3 Составной оператор

Составной оператор представляет собой один или несколько операторов и объявлений заключенных в фигурные скобки:

{ [объявление] … оператор; [оператор]; … }

Заметим, что в конце составного оператора точка с запятой не ставятся. Обычно составным оператором называют оператор не содержащий объявлений, а оператор с объявлениями называется блок, но синтаксических отличий между ними нет. Выполнение составного оператора заключается в последовательном выполнении составляющих его операторов, например, { r=s; s=t; t=r; } Этот составной оператор меняет местами значения переменных s и t, используя вспомогательную переменную r.

```
int main ()
{ int q,b;  double t,d;
        :
        if (...)
        { int e,g;    double f,q;   /* начало блока */
       e=b+t-d;
       g=++e;
       f=q=e/g;    } /* конец блока */
          :
          return 0;     }
```

В отмеченном блоке объявлены переменные e,g,f,q, которые будут уничтожены после выполнения составного оператора. Отметим, что переменная q является локальной в составном операторе, т.е. она никоим образом не связана с переменной q объявленной вначале функции main с типом int.

1.4.4 Оператор if

Оператор if позволяет разветвить вычислительный процесс на два варианта в зависимости от значения некоторого условия. В состав этого оператора могут входить один или два любых других оператора. Формат оператора

if (выражение) оператор_1; [else оператор_2;]

Выполнение оператора if начинается с вычисления выражения. Далее выполнение осуществляется по схеме:

- если выражение истинно (т.е. отлично от 0), то выполняется оператор_1.
- если выражение ложно (т.е. равно 0), то выполняется оператор_2.
- если выражение ложно и отсутствует конструкция со словом else (в квадратные скобки заключена необязательная конструкция), то выполнение оператора if завершается.

После выполнения оператора if значение передается на следующий оператор программы, если последовательность выполнения операторов программы не будет принудительно нарушена использованием операторов перехода, например,

```
if ( i<j ) { j = 0; i--; }    else { j = i-3; i++; }
```

Этот пример иллюстрирует также и тот факт, что на месте оператора_1, так же как и на месте оператора_2 могут находиться сложные конструкции.

Допускается использование вложенных операторов if. Оператор if может быть включен в конструкцию if или в конструкцию else другого оператора if. Чтобы сделать программу более читабельной, рекомендуется группировать операторы и конструкции во вложенных операторах if, используя фигурные скобки. Если же фигурные скобки опущены, то компилятор связывает каждое ключевое слово else с наиболее близким if, для которого нет else. Например:

```
int main ( )
{   int t=2, b=7, r=3;
    if (t>b)  {   if (b<r) r=b;   }
   else r=t;
   return (0);   }
```

В результате выполнения этой программы r станет равным 2. Если же в программе опустить фигурные скобки, стоящие после оператора if, то программа будет иметь следующий вид:

```
int main ( )
{  int t=2,b=7,r=3;
if ( a>b )
if ( b<c ) t=b;
else r=t;
return (0);   }
```

В этом случае r получит значение, равное 3, так как ключевое слово else относится ко второму оператору if, который не выполняется, поскольку не выполняется условие, проверяемое в первом операторе if.

Следующий фрагмент иллюстрирует вложенные операторы if:

```
char ZNAC;   int x,y,z;
if (ZNAC == '-') x = y - z;    else if (ZNAC == '+') x = y + z;
else if (ZNAC == '*') x = y * z;   else if (ZNAC == '/') x = y / z;  else ...
```

Из рассмотрения этого примера можно сделать вывод, что конструкции использующие вложенные операторы if, являются довольно громоздкими и не всегда достаточно надежными. Другим способом организации выбора из множества различных вариантов является использование специального оператора выбора switch.

1.4.5 Оператор switch

Оператор switch предназначен для организации выбора из множества различных вариантов. Формат оператора следующий

```
switch ( выражение )
{ [объявление]
        [ case константное_выражение_1]: [ список_операторов_1]
        [ case константное_выражение_2]: [ список_операторов_2]
        :
        [ default: [ список_операторов_default ] ]     }
```

Выражение, следующее за ключевым словом switch в круглых скобках, может быть любым выражением, допустимым в языке СИ, значение которого должно быть целым. Отметим, что можно использовать явное приведение к целому типу, однако необходимо помнить о тех ограничениях и рекомендациях, о которых говорилось выше. Значение этого выражения является ключевым для выбора из нескольких вариантов. Тело оператора smitch состоит из нескольких операторов, помеченных ключевым словом case с последующим константным выражением. Следует отметить, что использование целого константного выражения является существенным недостатком, присущим рассмотренному оператору. Так как константное выражение вычисляется во время трансляции, оно не может содержать переменные или вызовы функций. Обычно в качестве константного выражения используются целые или символьные константы.

Все константные выражения в операторе switch должны быть уникальны. Кроме операторов, помеченных ключевым словом case, один из фрагментов может быть помечен ключевым словом default. Списки операторов, следующих за двоеточиями, могут быть пустыми, либо содержать один или более операторов. Причем эти списки не требуется заключать в фигурные скобки. Отметим также, что внутри оператора switch можно использовать свои локальные переменные, объявления которых находятся перед первым ключевым словом case, однако в объявлениях не должна использоваться инициализация. Схема выполнения оператора switch:

- вычисляется выражение в круглых скобках;

- вычисленные значения последовательно сравниваются с константными_выражениями, следующими за ключевыми словами case;

- если одно из константных_выражений совпадает со значением выражения, то управление передается на оператор, помеченный соответствующим ключевым словом case;

- если ни одно из константных_выражений не равно выражению, то управление передается на оператор, помеченный ключевым словом default, а в случае его отсутствия управление передается на следующий после switch оператор.

Отметим интересную особенность использования оператора switch: конструкция со словом default может быть не последней в теле оператора switch. Ключевые слова case и default в теле оператора switch существенны только при начальной проверке, когда определяется начальная точка выполнения тела оператора switch. Все операторы, между начальным оператором и концом тела, выполняются вне зависимости от ключевых слов, если только какой-то из операторов не передаст управления из тела оператора switch. Таким образом, программист должен сам позаботится о выходе из case, если это необходимо. Чаще всего для этого используется оператор break. Для того, чтобы выполнить одни и те же действия для различных значений выражения, можно пометить один и тот же оператор несколькими ключевыми словами case. Например:

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

Выполнение оператора switch начинается с оператора, помеченного case 2. Таким образом, переменная i получает значение, равное 6, далее выполняется оператор, помеченный ключевым словом case 0, а затем case 4, переменная i примет значение 3, а затем значение -2. Оператор, помеченный ключевым словом default, не изменяет значения переменной.

Пример, приведенный ранее и иллюстрирующий использование вложенных операторов if, можно теперь записать с использованием оператора switch:

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

```
        case '*':     x = y * z;     break;
        case '/':     x = u / z;          break;
        default :     ;              }
```

Использование оператора break позволяет в необходимый момент прервать последовательность выполняемых операторов в теле оператора switch и передать управление оператору, следующему за switch. Отметим, что в теле оператора switch можно использовать вложенные операторы switch, при этом в ключевых словах case можно использовать одинаковые константные выражения. Например,

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

1.4.6 Оператор break

Оператор break может быть использован только внутри операторов switch, for, while или do while и обеспечивает прекращение выполнения самого внутреннего из содержащих его операторов. После выполнения оператора break управление передается оператору, следующему за прерванным. Формат оператора: break; Заметим, что оператор break нельзя использовать для выхода из нескольких вложенных циклов, а составной оператор, состоящий из двух операторов break, эквивалентен одному оператору break, т.е. в следующем фрагменте

```
for (i=0; i<100; i++)
for (j=0; j<100; j++)
{  …  if (a[i][j]<0) { break; break; }   …  }
```

при выполнении условия a[i][j]<0 завершится выполнение только внутреннего цикла по переменной j, а выполнение внешнего цикла по переменной i продолжится, несмотря на то, что оператор break повторен дважды. По своей сути оператор break является оператором перехода и операторы, записанные после него, выполняться не будут, если только им не будет передано управление с помощью других операторов перехода.

Для выхода из обоих циклов можно использовать дополнительную переменную, которая будет принимать значение отличное от нуля только при необходимости выхода из внешнего цикла.

```
for (i=0; i<100; i++)
          {      for (br2=j=0; j<100; j++)
          {    …    if  ( a[i][j]<0 )   {   br2=1;   break;}   … }
if  (br2)  break;   …   }
```

В приведенном примере сразу после завершения внутреннего цикла проверяется условие окончания внешнего цикла.

1.4.7 Оператор for

Оператор for - наиболее общий способ организации цикла, имеет следующий формат: for (выражение_1 ; выражение_2 ; выражение_3) тело

Выражение_1 обычно используется для установления начального значения переменных, управляющих циклом. Выражение_2 - определяет условие, при котором тело цикла будет выполняться. Выражение_3 определяет изменение переменных, управляющих циклом после каждого выполнения тела цикла. В качестве тела может быть использован любой оператор, в

том числе пустой или составной. Любое из выражений, а также все сразу, могут быть опущены, при этом разделяющие их символы ; пропускать нельзя. Схема выполнения оператора for:

1) вычисляется выражение_1;

2) вычисляется выражение_2;

3) если значения выражения_2 отлично от нуля (истина), выполняется тело цикла, вычисляется выражение_3 и осуществляется переход к пункту 2, если выражение_2 равно нулю (ложь), выполнение оператора for завершается и управление передается на оператор, следующий за оператором for. При отсутствии выражения_2 оно подразумевается истинным.

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

```
int main ( )
{   int i,b;
    for  ( i=1;  i<10;  i++)    b=i*i;
    return 0;    }
```

в примере вычисляются квадраты чисел от 1 до 9.

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

```
int main ( )
{   int top, bot;   char string[100], temp;
    for  (  top=0,  bot=98 ;   top<bot ;    top++,   bot--)
    {          temp =            string[top];
               string[top] =    string[bot];
               string[bot] =    temp;      }
return 0;       }
```

В этом примере, реализующем запись строки символов в обратном порядке, для управления циклом используются две переменные top и bot, значения которых движутся навстречу друг другу. Отметим, что на месте выражения_1 и выражения_3 здесь используется несколько выражений, записанных через запятую и выполняемых последовательно.

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

```
for   ( ; ; )
{ ...   if  ( некоторое условие )   break;   ...   }
```

Согласно синтаксису языка СИ тело цикла должно иметься у любого цикла, но если по смыслу оно не требуется, то в качестве тела можно использовать пустой оператор. Такой цикл может быть использован например, для организации поиска:

```
for   (i=0;   t[i]=>0;   i++)   ;
```

в примере переменная цикла i примет значение номера первого по порядку отрицательного элемента массива t.

1.4.8 Оператор while

Оператор цикла while называется циклом с предусловием и имеет следующий формат: while (выражение) тело ;

В качестве выражения допускается использовать любое выражение языка СИ, а в качестве тела - любой оператор, в том числе пустой или составной. Схема выполнения оператора while следующая:

1) вычисляется выражение;

2) если выражение ложно, то выполнение оператора while заканчивается и выполняется следующий по порядку оператор. Если выражение истинно, то выполняется тело оператора while;

3) процесс повторяется с пункта 1.

Оператор цикла вида

```
for ( выражение-1; выражение-2; выражение-3) тело ;
```

может быть заменен оператором while следующим образом:

```
выражение-1;
while ( выражение-2 )
{ тело
  выражение-3; }
```

Так же как и при выполнении оператора for, в операторе while вначале происходит проверка условия. Поэтому оператор while удобно использовать в ситуациях, когда тело оператора не всегда нужно выполнять.

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

1.4.9 Оператор do while

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

```
do тело while ( выражение );
```

Схема выполнения оператора do while:

1) выполняется тело цикла (которое может быть составным оператором);

2) вычисляется выражение;

3) если выражение ложно, то выполнение оператора do while заканчивается и выполняется следующий по порядку оператор. Если выражение истинно, то выполнение оператора продолжается с пункта 1.

Чтобы прервать выполнение цикла до того, как условие станет ложным, можно использовать оператор break. Операторы while и do while могут быть вложенными. Например:

```
int i,j,k,a[30];
...
i=0; j=0; k=0;
do {      i++;
          j--;
          while ( a[k]<i ) k++;     }
while ( i<30 && j<-30 );
```

1.4.10 Оператор continue

Оператор continue, как и оператор break, используется только внутри операторов цикла for, while или do while. Этот оператор позволяет продолжить выполнение цикла, пропустив операторы оставшиеся в теле цикла. Обычно этот оператор входит в состав оператора if. Формат оператора: continue; например:

```
int main ( )
{   int a,b;
    for ( a=1, b=0 ; a<100;  b+=a, a++)
    {       if (b%2) continue;
            ... /* обработка четных сумм */   }
```

```
return 0;     }
```

Когда сумма чисел от 1 до а становится нечетной, оператор continue передает управление на очередную итерацию цикла for, не выполняя операторы обработки четных сумм. Оператор continue, как и оператор break, прерывает самый внутренний из объемлющих его циклов.

1.4.11 Оператор return

Оператор return завершает выполнение функции, в которой он задан, и возвращает управление в вызывающую функцию, в точку, непосредственно следующую за вызовом. Оператор return в функции main передает управление операционной системе. Формат оператора: return [выражение];

Значение выражения, если оно задано, возвращается в вызывающую функцию в качестве значения вызываемой функции. Если выражение опущено, то возвращаемое значение не определено. Выражение может быть заключено в круглые скобки, хотя их наличие необязательно. Если в какой-либо функции отсутствует оператор return, то пе редача управления в вызывающую функцию происходит после выполнения последнего оператора вызываемой функции. При этом возвращаемое значение не определено. Если функция не должна иметь возвращаемого значения, то ее нужно объявлять с типом void.

Таким образом, использование оператора return необходимо либо для немедленного выхода из функции, либо для передачи возвращаемого значения, например,

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

Функция sum имеет два формальных параметра а и b типа int и возвращает значение типа int, о чем говорит описатель стоящий перед именем функции. Возвращаемое оператором return значение равно сумме фактических параметров. Например:

```
int  prov  ( int a,  double b )
{   double c;
    if  ( a<3 ) return 1;
    else     if (b>10) return 2;
               else  { c=a+b;
                        if  ( (2*c-b)==11) return 3;      }      }
```

Оператор return используется для выхода из функции в случае выполнения одного из проверяемых условий.

1.4.12 Оператор goto

Использование оператора безусловного перехода goto в практике программирования на языке СИ настоятельно не рекомендуется, так как он затрудняет понимание программ и возможность их модификаций. Формат этого оператора следующий: goto метка;

Оператор goto передает управление на оператор, помеченный меткой метка. Помеченный оператор должен находиться в той же функции, что и оператор goto, а метка должна быть уникальной, т.е. нельзя помечать одной меткой более одного оператора.

Любой оператор в составном операторе может иметь свою метку. Используя оператор goto, можно передавать управление внутрь составного оператора, условного оператора, операторов цикла и оператора переключателя . Но в таких случаях следует быть осторожным, так как при таком входе инициализация объявленных переменных не выполняется и их значения будут не определены, неопределенными также останутся и переменные, определяемые в заголовке цикла for.

1.5 ОПРЕДЕЛЕНИЕ И ВЫЗОВ ФУНКЦИЙ

Мощность языка СИ во многом определяется легкостью и гибкостью в определении и использовании функций в СИ-программах. В отличие от других языков программирования высокого уровня в языке СИ нет деления на процедуры, подпрограммы и функции, здесь вся программа строится только из функций.

Функция - это совокупность объявлений и операторов, обычно предназначенная для решения определенной задачи. Каждая функция должна иметь имя, которое используется для ее объявления, определения и вызова. В любой программе на СИ должна быть функция с именем main (главная функция), именно с этой функции, в каком бы месте программы она не находилась, начинается выполнение программы.

При вызове функции ей при помощи аргументов (формальных параметров) могут быть переданы некоторые значения (фактические параметры) используемые во время выполнения функции. Функция может возвращать некоторое (одно !) значение. Это возвращаемое значение и есть результат выполнения функции, который при выполнении программы подставляется в точку вызова функции, где бы этот вызов не встретился. Допускается также использовать функции без аргументов и функции не возвращающие никаких значений. Действие таких функций может состоять, например, в изменении значений некоторых переменных, выводе на печать некоторых текстов и т.п. С использованием функций в языке СИ связаны три понятия - определение функции (описание действий, выполняемых функцией), объявление функции (задание формы обращения к функции) и вызов функции (выполнение).

Определение функции задает тип возвращаемого значения, имя функции, типы и число формальных параметров, а также объявления переменных и операторы, называемые телом функции и определяющие действие функции. В определении функции также может быть задан класс памяти. Например:

```
int rus (unsigned char r)
{ if (r>='А' && r<='ё')   return 1;   else   return 0;   }
```

- определена функция с именем rus, имеющая один параметр с именем r и типом unsigned char. Функция возвращает целое значение, равное 1, если параметр функции является буквой русского алфавита, или 0 в противном случае.

В языке СИ нет требования, чтобы определение функции обязательно предшествовало ее вызову. Определения используемых функций могут следовать за определением функции main, перед ним, или находится в другом файле. Однако для того, чтобы компилятор мог осуществить проверку соответствия типов передаваемых фактических параметров типам формальных параметров и в случае необходимости выполнить нужные преобразования, до вызова функции требуется поместить объявление (прототип) функции.

Объявление функции имеет такой же вид, что и определение функции, с той лишь разницей, что тело функции отсутствует, и имена формальных параметров тоже могут быть опущены. Для функции, определенной в последнем примере, прототип может иметь вид

int rus (unsigned char r); или rus (unsigned char); .

В программах на языке СИ широко используются так называемые библиотечные функции, т.е. функции, предварительно разработанные и записанные в библиотеки. Прототипы библиотечных функций находятся в специальных заголовочных файлах, поставляемых вместе с библиотеками в составе систем программирования, и включаются в программу с помощью директивы #include.

Если объявление функции не задано, то по умолчанию строится прототип функции на основе анализа первой ссылки на функцию, будь то вызов функции или определение. Однако такой прототип не всегда согласуется с последующим определением или вызовом функции. Рекомендуется всегда задавать прототип функции. Это позволит компилятору либо выдавать диагностические сообщения, при неправильном использовании функции, либо корректным образом регулировать несоответствие аргументов, устанавливаемое при выполнении программы.

Объявление параметров функции при ее определении может быть выполнено в так называемом "старом стиле", при котором в скобках после имени функции следуют только имена параметров, а после скобок объявления типов параметров. Например, функция rus из предыдущего примера может быть определена следующим образом:

```
int rus (r)
unsigned char r;
{ ... /* тело функции */ ... }
```

В соответствии с синтаксисом языка СИ определение функции имеет следующую форму:

[спецификатор_класса_памяти] [спецификатор_типа] имя_функции
([список_формальных_параметров])
{ тело_функции }

Необязательный спецификатор_класса_памяти задает класс памяти функции, который может быть static или extern. Подробно классы памяти будут рассмотрены в следующем разделе.

Спецификатор_типа функции задает тип возвращаемого значения и может задавать любой тип. Если спецификатор_типа не задан, то предполагается, что функция возвращает значение типа int. Функция не может возвращать массив или функцию, но может возвращать указатель на любой тип, в том числе и на массив и на функцию. Тип возвращаемого значения, задаваемый в определении функции, должен соответствовать типу в объявлении этой функции. Функция возвращает значение, если ее выполнение заканчивается оператором return, содержащим некоторое выражение. Указанное выражение вычисляется, преобразуется, если необходимо, к типу возвращаемого значения и возвращается в точку вызова функции в качестве результата. Если оператор return не содержит выражения или выполнение функции завершается после выполнения последнего ее оператора (без выполнения оператора return), то возвращаемое значение не определено. Для функций, не использующих возвращаемое значение, должен быть использован тип void, указывающий на отсутствие возвращаемого значения. Если функция определена как функция, возвращающая некоторое значение, а в операторе return при выходе из нее отсутствует выражение, то поведение вызывающей функции после передачи ей управления может быть непредсказуемым.

Список_формальных_параметров - это последовательность объявлений формальных параметров, разделенная запятыми. Формальные параметры - это переменные, используемые внутри тела функции и получающие значение при вызове функции путем копирования в них значений соответствующих фактических параметров. Список_формальных_параметров может заканчиваться запятой (,) или запятой с многоточием (,...) , это означает, что число аргументов функции переменно. Однако предполагается, что функция имеет, по крайней мере, столько обязательных аргументов, сколько формальных параметров задано перед последней запятой в списке параметров. Такой функции может быть передано большее число аргументов, но над дополнительными аргументами не проводится контроль типов. Если функция не использует параметров, то наличие круглых скобок обязательно, а вместо списка параметров рекомендуется указать слово void.

Порядок и типы формальных параметров должны быть одинаковыми в определении функции и во всех ее объявлениях. Типы фактических параметров при вызове функции должны быть совместимы с типами соответствующих формальных параметров. Тип формального параметра может быть любым основным типом, структурой, объединением, перечислением, указателем или массивом. Если тип формального параметра не указан, то этому параметру присваивается тип int.

Для формального параметра можно задавать класс памяти register, при этом для величин типа int спецификатор типа можно опустить.

Идентификаторы формальных параметров используются в теле функции в качестве ссылок на переданные значения. Эти идентификаторы не могут быть переопределены в блоке, образующем тело функции, но могут быть переопределены во внутреннем блоке внутри тела функции.

При передаче параметров в функцию, если необходимо, выполняются обычные арифметические преобразования для каждого формального параметра и каждого фактического параметра независимо. После преобразования формальный параметр не может быть короче, чем int, т.е. объявление формального параметра с типом char равносильно его объявлению с типом int. А параметры, представляющие собой действительные числа, имеют тип double.

Преобразованный тип каждого формального параметра определяет, как интерпретируются аргументы, помещаемые при вызове функции в стек. Несоответствие типов фактических аргументов и формальных параметров может быть причиной неверной интерпретации.

Тело_функции - это составной оператор, содержащий операторы, определяющие действие функции.

Все переменные, объявленные в теле функции без указания класса памяти, имеют класс памяти auto, т.е. они являются локальными. При вызове функции локальным переменным отводится память в стеке и производится их инициализация. Управление передается первому оператору тела функции и начинается выполнение функции, которое продолжается до тех пор, пока не встретится оператор return или последний оператор тела функции. Управление при этом возвращается в точку, следующую за точкой вызова, а локальные переменные становятся недоступными. При новом вызове функции для локальных переменных память распределяется вновь, и поэтому старые значения локальных переменных теряются.

Параметры функции передаются по значению и могут рассматриваться как локальные переменные, для которых выделяется память при вызове функции и производится инициализация значениями фактических параметров. При выходе из функции значения этих переменных теряются. Поскольку передача параметров происходит по значению, в теле функции нельзя изменить значения переменных в вызывающей функции, являющихся фактическими параметрами. Однако, если в качестве параметра передать указатель на некоторую переменную, то, используя операцию разадресации можно изменить значение этой переменной. Например:

```
void change (int x, int y) /* Неправильное использование параметров */
{ int k=x;
x=y;    y=k;  }
```

В данной функции значения переменных x и y, являющихся формальными параметрами, меняются местами, но поскольку эти переменные существуют только внутри функции change, значения фактических параметров, используемых при вызове функции, останутся неизменными.

Для того, чтобы менялись местами значения фактических аргументов, можно использовать функцию, приведенную в следующем примере. Например:

```
void change (int *x, int *y) /* Правильное использование параметров */
{ int k=*x;
*x=*y;    *y=k;    }
```

При вызове такой функции в качестве фактических параметров должны быть использованы не значения переменных, а их адреса change (&a,&b);

Если требуется вызвать функцию до ее определения в рассматриваемом файле, или определение функции находится в другом исходном файле, то вызов функции следует предварять объявлением этой функции. Объявление (прототип) функции имеет следующий формат: [спецификатор_класса_памяти] [спецификатор_типа] имя_функции ([список_формальных_параметров]) [,список_имен_функций];

В отличие от определения функции, в прототипе за заголовком сразу же следует точка с запятой, а тело функции отсутствует. Если несколько разных функций возвращают значения одинакового типа и имеют одинаковые списки формальных параметров, то эти функции можно объявить в одном прототипе, указав имя одной из функций в качестве имени-функции, а все другие поместить в список_имен_функций, причем каждая функция должна сопровождаться списком формальных параметров. Правила использования остальных элементов формата такие же, как при определении функции. Имена формальных параметров при объявлении функции можно не указывать, а если они указаны, то их область действия распространяется только до конца объявления.

Прототип - это явное объявление функции, которое предшествует определению функции. Тип возвращаемого значения при объявлении функции должен соответствовать типу возвращаемого значения в определении функции. Если прототип функции не задан, а встретился вызов функции, то строится неявный прототип из анализа формы вызова функции. Тип возвращаемого значения создаваемого прототипа int, а список типов и числа параметров функции формируется на основании типов и числа фактических параметров, используемых при данном вызове.

Таким образом, прототип функции необходимо задавать:

1) если функция возвращает значение типа, отличного от int;

2) если требуется проинициализировать некоторый указатель на функцию до того, как эта функция будет определена.

Наличие в прототипе полного списка типов аргументов параметров позволяет выполнить проверку соответствия типов фактических параметров при вызове функции типам формальных параметров, и, если необходимо, выполнить соответствующие преобразования. В прототипе можно указать, что число параметров функции переменно, или, что функция не имеет параметров.

Если прототип задан с классом памяти static, то и определение функции должно иметь класс памяти static. Если спецификатор класса памяти не указан, то подразумевается класс памяти extern. Вызов функции имеет следующий формат: адресное_выражение ([список_выражений]).

Поскольку синтаксически имя функции является адресом начала тела функции, в качестве обращения к функции может быть использовано адресное_выражение (в том числе и имя функции или разадресация указателя на функцию), имеющее значение адреса функции.

Список_выражений представляет собой список фактических параметров, передаваемых в функцию. Этот список может быть и пустым, но наличие круглых скобок обязательно.

Фактический параметр может быть величиной любого основного типа, структурой, объединением, перечислением или указателем на объект любого типа. Массив и функция не могут быть использованы в качестве фактических параметров, но можно использовать указатели на эти объекты.

Выполнение вызова функции происходит следующим образом:

1 Вычисляются выражения в списке выражений и подвергаются обычным арифметическим преобразованиям. Затем, если известен прототип функции, тип полученного фактического аргумента сравнивается с типом соответствующего формального параметра. Если они не совпадают, то либо производится преобразование типов, либо формируется сообщение об ошибке. Число выражений в списке выражений должно совпадать с числом формальных параметров, если только функция не имеет переменного числа параметров. В последнем случае проверке подлежат только обязательные параметры. Если в прототипе функции указано, что ей не требуются параметры, а при вызове они указаны, формируется сообщение об ошибке.

2 Происходит присваивание значений фактических параметров соответствующим формальным параметрам.

3 Управление передается на первый оператор функции.

4 Выполнение оператора return в теле функции возвращает управление и, возможно, значение в вызывающую функцию. При отсутствии оператора return управление возвращается после выполнения последнего оператора тела функции, а возвращаемое значение не определено.

Адресное выражение, стоящее перед скобками определяет адрес вызываемой функции. Это значит, что функция может быть вызвана, через указатель на функцию, например,

```
int (*fun)(int x, int *y);
```

здесь объявлена переменная fun как указатель на функцию с двумя параметрами: типа int и указателем на int. Сама функция должна возвращать значение типа int. Круглые скобки, содержащие имя указателя fun и признак указателя *, обязательны, иначе запись int *fun (intx,int *y); будет интерпретироваться как объявление функции fun возвращающей указатель на int.

Вызов функции возможен только после инициализации значения указателя fun и имеет вид: (*fun)(i,&j); в этом выражении для получения адреса функции, на которую ссылается указатель fun, используется операция разадресации *.

Указатель на функцию может быть передан в качестве параметра функции. При этом разадресация происходит во время вызова функции, на которую ссылается указатель на функцию. Присвоить значение указателю на функцию можно в операторе присваивания, употребив имя функции без списка параметров. Например:

```
double (*fun1)(int x, int y);    double fun2(int k, int l);
fun1=fun2;     /* инициализация указателя на функцию */
(*fun1)(2,7);   /* обращение к функции */
```

- указатель на функцию fun1 описан как указатель на функцию с двумя параметрами, возвращающую значение типа double, и также описана функция fun2. В противном случае, т.е. когда указателю на функцию присваивается функция, описанная иначе чем указатель, произойдет ошибка.

Рассмотрим пример использования указателя на функцию в качестве параметра функции вычисляющей производную от функции cos(x). Например:

```
double proiz(double x, double dx, double (*f)(double x) );
double fun(double z);
int main()
{  double x; /* точка вычисления производной */
double dx; /* приращение */
double z; /* значение производной */
scanf("%f,%f",&x,&dx); /* ввод значений x и dx */
z=proiz(x,dx,fun); /* вызов функции */
printf("%f",z); /* печать значения производной */
return 0;    }

double proiz(double x,double dx, double (*f)(double z) )
{ /* функция, вычисляющая производную */
double xk,xk1,pr;
xk=fun(x);   xk1=fun(x+dx);   pr=(xk1/xk-1e0)*xk/dx;
return pr;    }

double fun( double z) { /* функция, от которой вычисляется производная */
return (cos(z));   }
```

Для вычисления производной от какой-либо другой функции можно изменить тело функции fun или использовать при вызове функции proiz имя другой функции. В частности,

для вычисления производной от функции cos(x) можно вызвать функцию proiz в форме z=proiz(x,dx,cos); а для вычисления производной от функции sin(x) в форме z=proiz(x,dx,sin);

Любая функция в программе на языке СИ может быть вызвана рекурсивно, т.е. она может вызывать саму себя. Компилятор допускает любое число рекурсивных вызовов. При каждом вызове для формальных параметров и переменных с классом памяти auto и register выделяется новая область памяти, так что их значения из предыдущих вызовов не теряются, но в каждый момент времени доступны только значения текущего вызова.

Переменные, объявленные с классом памяти static, не требуют выделения новой области памяти при каждом рекурсивном вызове функции и их значения доступны в течение всего времени выполнения программы.

Классический пример рекурсии - это математическое определение факториала:

$$n! = \begin{cases} 1 & \text{при } n=0; \\ n*(n-1)! & \text{при } n\geq 1 . \end{cases}$$

Функция, вычисляющая факториал, будет иметь вид

```
long fakt(int n)
{   return ( (n==1) ? 1 : n*fakt(n-1) );   }
```

Хотя компилятор языка СИ не ограничивает число рекурсивных вызовов функций, это число ограничивается ресурсом памяти компьютера, и при слишком большом числе рекурсивных вызовов может произойти переполнение стека.

1.6 СТРУКТУРА ПРОГРАММЫ И КЛАССЫ ПАМЯТИ

1.6.1 Исходные файлы и объявление переменных

Обычная СИ-программа представляет собой определение функции main, которая для выполнения необходимых действий вызывает другие функции. Приведенные выше примеры программ представляли собой один исходный файл, содержащий все необходимые для выполнения программы функции. Связь между функциями осуществлялась по данным посредством передачи параметров и возврата значений функций. Но компилятор языка СИ позволяет также разбить программу на несколько отдельных частей (исходных файлов), оттранслировать каждую часть отдельно, и затем объединить все части в один выполняемый файл при помощи редактора связей.

При такой структуре исходной программы функции, находящиеся в разных исходных файлах могут использовать глобальные внешние переменные. Все функции в языке Си по определению внешние и всегда доступны из любых файлов. Например, если программа состоит из двух исходных файлов, как показано на рис. 2, то функция main может вызывать любую из трех функций fun1, fun2, fun3, а каждая из этих функций может вызывать любую другую.

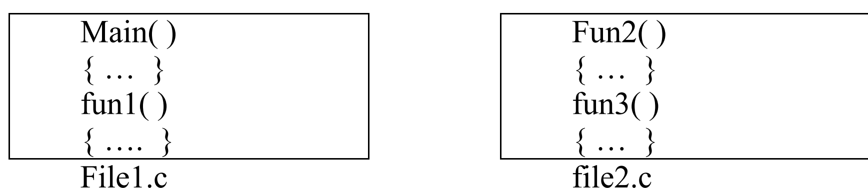


Рис. 2 Пример программы из двух файлов

Для того, чтобы определяемая функция могла выполнять какие-либо действия, она должна использовать переменные. В языке СИ все переменные должны быть объявлены до их использования. Объявления устанавливают соответствие имени и атрибутов переменной, функции или типа. Определение переменной вызывает выделение памяти для хранения ее значения. Класс выделяемой памяти определяется спецификатором класса памяти, и

определяет время жизни и область видимости переменной, связанные с понятием блока программы.

В языке СИ блоком считается последовательность объявлений, определений и операторов, заключенная в фигурные скобки. Существуют два вида блоков - составной оператор и определение функции, состоящее из составного оператора, являющегося телом функции, и предшествующего телу заголовка функции (в который входят имя функции, типы возвращаемого значения и формальных параметров). Блоки могут включать в себя составные операторы, но не определения функций. Внутренний блок называется вложенным, а внешний блок - объемлющим.

Время жизни - это интервал времени выполнения программы, в течение которого программный объект (переменная или функция) существует. Время жизни переменной может быть локальным или глобальным. Переменная с глобальным временем жизни имеет распределенную для нее память и определенное значение на протяжении всего времени выполнения программы, начиная с момента выполнения объявления этой переменной. Переменная с локальным временем жизни имеет распределенную для него память и определенное значение только во время выполнения блока, в котором эта переменная определена или объявлена. При каждом входе в блок для локальной переменной распределяется новая память, которая освобождается при выходе из блока.

Все функции в СИ имеют глобальное время жизни и существуют в течение всего времени выполнения программы.

Область видимости - это часть текста программы, в которой может быть использован данный объект. Объект считается видимым в блоке или в исходном файле, если в этом блоке или файле известны имя и тип объекта. Объект может быть видимым в пределах блока, исходного файла или во всех исходных файлах, образующих программу. Это зависит от того, на каком уровне объявлен объект: на внутреннем, т.е. внутри некоторого блока, или на внешнем, т.е. вне всех блоков. Если объект объявлен внутри блока, то он видим в этом блоке и во всех внутренних блоках. Если объект объявлен на внешнем уровне, то он видим от точки его объявления до конца данного исходного файла. Объект может быть сделан глобально видимым с помощью соответствующих объявлений во всех исходных файлах, образующих программу.

Спецификатор класса памяти в объявлении переменной может быть auto, register, static или extern. Если класс памяти не указан, то он определяется по умолчанию из контекста объявления. Объекты классов auto и register имеют локальное время жизни. Спецификаторы static и extern определяют объекты с глобальным временем жизни.

При объявлении переменной на внутреннем уровне может быть использован любой из четырех спецификаторов класса памяти, а если он не указан, то подразумевается класс памяти auto. Переменная с класом памяти auto имеет локальное время жизни и видна только в блоке, в котором объявлена. Память для такой переменной выделяется при входе в блок и освобождается при выходе из блока. При повторном входе в блок этой переменной может быть выделен другой участок памяти. Переменная с классом памяти auto автоматически не инициализируется. Она должна быть проинициализирована явно при объявлении путем присвоения ей начального значения. Значение неинициализированной переменной с классом памяти auto считается неопределенным.

Спецификатор класса памяти register предписывает компилятору распределить память для переменной в регистре, если это представляется возможным. Использование регистровой памяти обычно приводит к сокращению времени доступа к переменной. Переменная, объявленная с классом памяти register, имеет ту же область видимости, что и переменная auto. Число регистров, которые можно использовать для значений переменных, ограничено возможностями компьютера, и в том случае, если компилятор не имеет в распоряжении свободных регистров, то переменной выделяется память как для класса auto.

Класс памяти register может быть указан только для переменных с типом int или указателей с размером, равным размеру int.

Переменные, объявленные на внутреннем уровне со спецификатором класса памяти static, обеспечивают возможность сохранить значение переменной при выходе из блока и использовать его при повторном входе в блок. Такая переменная имеет глобальное время жизни и область видимости внутри блока, в котором она объявлена. В отличие от переменных с классом auto, память для которых выделяется в стеке, для переменных с классом static память выделяется в сегменте данных, и поэтому их значение сохраняется при выходе из блока. Например:

```
/* объявления переменной i на внутреннем уровне
   с классом памяти static. */
/* исходный файл file1.c */
main ( )   { ...   }
fun1 ( )    {          static int i=0; ...   }
/* исходный файл file2.c */
fun2()   {  static int i=0; ...   }
fun3 ( )   {static int i=0; ...   }
```

В приведенном примере объявлены три разные переменные с классом памяти static, имеющие одинаковые имена i. Каждая из этих переменных имеет глобальное время жизни, но видима только в том блоке (функции), в которой она объявлена. Эти переменные можно использовать для подсчета числа обращений к каждой из трех функций.

Переменные класса памяти static могут быть инициализированы константным выражением. Если явной инициализации нет, то такой переменной присваивается нулевое значение. При инициализации константным адресным выражением можно использовать адреса любых внешних объектов, кроме адресов объектов с классом памяти auto, так как адрес последних не является константой и изменяется при каждом входе в блок. Инициализация выполняется один раз при первом входе в блок.

Переменная, объявленная локально с классом памяти extern, является ссылкой на переменную с тем же самым именем, определенную глобально в одном из исходных файлов программы. Цель такого объявления состоит в том, чтобы сделать определение переменной глобального уровня видимым внутри блока. Например:

```
/* объявления переменной i, являющейся именем внешнего
 массива длинных целых чисел, на локальном уровне */
/* исходный файл file1.c */
main ( )   { ...   }
fun1 ( )    {          extern long i[]; ...   }
/* исходный файл file2.c */
long i[MAX]={0};
fun2 ( )   { ...   }
fun3 ( )   { ...   }
```

Объявление переменной i[] как extern в приведенном примере делает ее видимой внутри функции fun1. Определение этой переменной находится в файле file2.c на глобальном уровне и должно быть только одно, в то время как объявлений с классом памяти extern может быть несколько.

Объявление с классом памяти extern требуется при необходимости использовать переменную, описанную в текущем исходном файле, но ниже по тексту программы, т.е. до выполнения ее глобального определения. Следующий пример иллюстрирует такое использование переменной с именем st:

```
main()   { extern int st[]; ...   }
static int st[MAX]={0};
```

fun1 ( ) { ... }

Объявление переменной со спецификатором extern информирует компилятор о том, что память для переменной выделять не требуется, так как это выполнено где-то в другом месте программы.

При объявлении переменных на глобальном уровне может быть использован спецификатор класса памяти static или extern, а так же можно объявлять переменные без указания класса памяти. Классы памяти auto и register для глобального объявления недопустимы.

Объявление переменных на глобальном уровне - это или определение переменных, или ссылки на определения, сделанные в другом месте программы. Объявление глобальной переменной, которое инициализирует эту переменную (явно или неявно), является определением переменной. Определение на глобальном уровне может задаваться в следующих формах:

1 Переменная объявлена с классом памяти static. Такая переменная может быть инициализирована явно константным выражением или по умолчанию нулевым значением. То есть объявления static int i=0 и static int i эквивалентны, и в обоих случаях переменной i будет присвоено значение 0.

2 Переменная объявлена без указания класса памяти, но с явной инициализацией. Такой переменной по умолчанию присваивается класс памяти static, т.е. объявления int i=1 и static int i=1 будут эквивалентны.

Переменная, объявленная глобально, видима в пределах остатка исходного файла, в котором она определена. Выше своего описания и в других исходных файлах эта переменная невидима (если только она не объявлена с классом extern). Глобальная переменная может быть определена только один раз в пределах своей области видимости. В другом исходном файле может быть объявлена другая глобальная переменная с таким же именем и с классом памяти static, конфликта при этом не возникает, так как каждая из этих переменных будет видимой только в своем исходном файле.

Спецификатор класса памяти extern для глобальных переменных используется, как и для локального объявления, в качестве ссылки на переменную, объявленную в другом месте программы, т.е. для расширения области видимости переменной. При таком объявлении область видимости переменной расширяется до конца исходного файла, в котором сделано объявление.

В объявлениях с классом памяти extern не допускается инициализация, так как эти объявления ссылаются на уже существующие и определенные ранее переменные.

Переменная, на которую делается ссылка с помощью спецификатора extern, может быть определена только один раз в одном из исходных файлов программы.

1.6.2 Объявления функций

Функции всегда определяются глобально. Они могут быть объявлены с классом памяти static или extern. Объявления функций на локальном и глобальном уровнях имеют одинаковый смысл.

Правила определения области видимости для функций отличаются от правил видимости для переменных и состоят в следующем:

1 Функция, объявленная как static, видима в пределах того файла, в котором она определена. Каждая функция может вызвать другую функцию с классом памяти static из своего исходного файла, но не может вызвать функцию определенную с классом static в другом исходном файле. Разные функции с классом памяти static, имеющие одинаковые имена, могут быть определены в разных исходных файлах, и это не ведет к конфликту.

2 Функция, объявленная с классом памяти extern, видима в пределах всех исходных файлов программы. Любая функция может вызывать функции с классом памяти extern.

3 Если в объявлении функции отсутствует спецификатор класса памяти, то по умолчанию принимается класс extern.

Все объекты с классом памяти extern компилятор помещает в объектном файле в специальную таблицу внешних ссылок, которая используется редактором связей для разрешения внешних ссылок. Часть внешних ссылок порождается компилятором при обращениях к библиотечным функциям СИ, поэтому для разрешения этих ссылок редактору связей должны быть доступны соответствующие библиотеки функций.

1.6.3 Время жизни и область видимости программных объектов

Время жизни переменной (глобальной или локальной) определяется по следующим правилам:

1 Переменная, объявленная глобально (т.е. вне всех блоков), существует на протяжении всего времени выполнения программы.

2 ЛОКАЛЬНЫЕ ПЕРЕМЕННЫЕ (Т.Е. ОБЪЯВЛЕННЫЕ ВНУТРИ БЛОКА) С КЛАССОМ ПАМЯТИ REGISTER ИЛИ AUTO, ИМЕЮТ ВРЕМЯ ЖИЗНИ ТОЛЬКО НА ПЕРИОД ВЫПОЛНЕНИЯ ТОГО БЛОКА, В КОТОРОМ ОНИ ОБЪЯВЛЕНЫ. ЕСЛИ ЛОКАЛЬНАЯ ПЕРЕМЕННАЯ ОБЪЯВЛЕНА С КЛАССОМ ПАМЯТИ STATIC ИЛИ EXTERN, ТО ОНА ИМЕЕТ ВРЕМЯ ЖИЗНИ НА ПЕРИОД ВЫПОЛНЕНИЯ ВСЕЙ ПРОГРАММЫ.

Видимость переменных и функций в программе определяется следующими правилами:

1 Переменная, объявленная или определенная глобально, видима от точки объявления или определения до конца исходного файла. Можно сделать переменную видимой и в других исходных файлах, для чего в этих файлах следует ее объявить с классом памяти extern.

2 Переменная, объявленная или определенная локально, видима от точки объявления или определения до конца текущего блока. Такая переменная называется локальной.

3 Переменные из объемлющих блоков, включая переменные объявленные на глобальном уровне, видимы во внутренних блоках. Эту видимость называют вложенной. Если переменная, объявленная внутри блока, имеет то же имя, что и переменная, объявленная в объемлющем блоке, то это разные переменные, и переменная из объемлющего блока во внутреннем блоке будет невидимой.

4 Функции с классом памяти static видимы только в исходном файле, в котором они определены. Всякие другие функции видимы во всей программе.

Метки в функциях видимы на протяжении всей функции. Имена формальных параметров, объявленные в списке параметров прототипа функции, видимы только от точки объявления параметра до конца объявления функции.

1.6.4 Инициализация глобальных и локальных переменных

При инициализации необходимо придерживаться следующих правил:

1 Объявления, содержащие спецификатор класса памяти, extern не могут содержать инициаторов.

2 Глобальные переменные всегда инициализируются, и если это не сделано явно, то они инициализируются нулевым значением.

3 Переменная с классом памяти static может быть инициализирована константным выражением. Инициализация для них выполняется один раз перед началом программы. Если явная инициализация отсутствует, то переменная инициализируется нулевым значением.

4 Инициализация переменных с классом памяти auto или register выполняется всякий раз при входе в блок, в котором они объявлены. Если инициализация переменных в объявлении отсутствует, то их начальное значение неопределенно.

5 Начальными значениями для глобальных переменных и для переменных с классом памяти static должны быть константные выражения. Адреса таких переменных являются константами, и эти константы можно использовать для инициализации объявленных глобально указателей. Адреса переменных с классом памяти auto или register не являются константами, и их нельзя использовать в инициаторах.

Пример:

```
int global_var;
int func(void)
{   int local_var;                    /* по умолчанию auto */
static int *local_ptr=&local_var; /* так неправильно */
static int *global_ptr=&global_var;       /* а так правильно */
register int *reg_ptr=&local_var;   }       /* и так правильно */
```

В приведенном примере глобальная переменная global_var имеет глобальное время жизни и постоянный адрес в памяти, и этот адрес можно использовать для инициализации статического указателя global_ptr. Локальная переменная local_var, имеющая класс памяти auto, размещается в памяти только на время работы функции func, адрес этой переменной не является константой и не может быть использован для инициализации статической переменной local_ptr. Для инициализации локальной регистровой переменной reg_ptr можно использовать неконстантные выражения и, в частности, адрес переменной local_ptr.

1.7 УКАЗАТЕЛИ И АДРЕСНАЯ АРИФМЕТИКА

1.7.1 Методы доступа к элементам массивов

В языке СИ между указателями и массивами существует тесная связь. Например, когда объявляется массив в виде int array[25], то этим определяется не только выделение памяти для двадцати пяти элементов массива, но и для указателя с именем array, значение которого равно адресу первого по счету (нулевого) элемента массива, т.е. сам массив остается безымянным, а доступ к элементам массива осуществляется через указатель с именем array. С точки зрения синтаксиса языка указатель arrey является константой, значение которой можно использовать в выражениях, но изменить это значение нельзя.

Поскольку имя массива является указателем, допустимо, например, такое присваивание:

```
int arrey[25];    int *ptr;
ptr = array;
```

здесь указатель ptr устанавливается на адрес первого элемента массива, причем присваивание ptr=arrey можно записать в эквивалентной форме

```
ptr=&arrey[0].
```

Для доступа к элементам массива существует два различных способа. Первый способ связан с использованием обычных индексных выражений в квадратных скобках, например, array[16]=3 или array[i+2]=7.

При таком способе доступа записываются два выражения, причем второе выражение заключается в квадратные скобки. Одно из этих выражений должно быть указателем, а второе - выражением целого типа. Последовательность записи этих выражений может быть любой, но в квадратных скобках записывается выражение, следующее вторым. Поэтому записи arrey[16] и 16[arrey] будут эквивалентными и обозначают элемент массива с номером шестнадцать.

Указатель, используемый в индексном выражении, не обязательно должен быть константой, указывающей на какой-либо массив, это может быть и переменная. В частности, после выполнения присваивания ptr=array доступ к шестнадцатому элементу массива можно получить с помощью указателя ptr в форме ptr[16] или 16[ptr].

Второй способ доступа к элементам массива связан с использованием адресных выражений и операции разадресации в форме

*(array+16)=3 или *(array+i+2)=7.

При таком способе доступа адресное выражение, равное адресу шестнадцатого элемента массива, тоже может быть записано разными способами *(array+16) или *(16+array).

При реализации на компьютере первый способ приводится ко второму, т.е. индексное выражение преобразуется к адресному. Для приведенных примеров array[16] и 16[array] преобразуются в *(array+16).

Для доступа к начальному элементу массива (т.е. к элементу с нулевым индексом) можно использовать просто значение указателя array или ptr. Любое из присваиваний

*array = 2; array[0] = 2; *(array+0) = 2; *ptr = 2; ptr[0] = 2; *(ptr+0) = 2;

присваивает начальному элементу массива значение 2, но быстрее всего выполнются присваивания *array=2 и *ptr=2 , так как в них не требуется выполнять операции сложения.

1.7.2 Указатели на многомерные массивы

Указатели на многомерные массивы в языке СИ - это массивы массивов, т.е. такие массивы, элементами которых являются массивы. При объявлении таких массивов в памяти компьютера создается несколько различных объектов. Например, при выполнении объявления двумерного массива int arr2[4][3] в памяти выделяется участок для хранения значения переменной arr, которая является указателем на массив из четырех указателей. Для этого массива из четырех указателей тоже выделяется память. Каждый из этих четырех указателей содержит адрес массива из трех элементов типа int, и, следовательно, в памяти компьютера выделяется четыре участка для хранения четырех массивов чисел типа int, каждый из которых состоит из трех элементов. Такое выделение памяти показано на схеме на рис. 3.

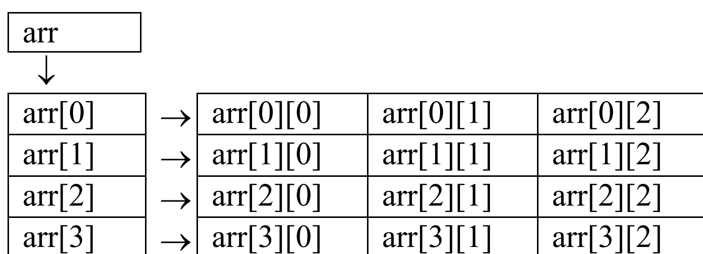


Рис. 3 Распределение памяти для двумерного массива

Таким образом, объявление arr2[4][3] порождает в программе три разных объекта: указатель с идентификатором arr, безымянный массив из четырех указателей и безымянный массив из двенадцати чисел типа int. Для доступа к безымянным массивам используются адресные выражения с указателем arr. Доступ к элементам массива указателей осуществляется с указанием одного индексного выражения в форме arr2[2] или *(arr2+2). Для доступа к элементам двумерного массива чисел типа int должны быть использованы два индексных выражения в форме arr2[1][2] или эквивалентных ей *(*(arr2+1)+2) и (*(arr2+1))[2]. Следует учитывать, что с точки зрения синтаксиса языка СИ указатель arr и указатели arr[0], arr[1], arr[2], arr[3] являются константами, и их значения нельзя изменять во время выполнения программы.

Размещение трехмерного массива происходит аналогично и объявление float arr3[3][4][5] порождает в программе, кроме самого трехмерного массива из шестидесяти чисел типа float, массив из четырех указателей на тип float, массив из трех указателей на массив указателей на float и указатель на массив массивов указателей на float.

При размещении элементов многомерных массивов они располагаются в памяти подряд по строкам, т.е. быстрее всего изменяется последний индекс, а медленнее - первый. Такой

порядок дает возможность обращаться к любому элементу многомерного массива, используя адрес его начального элемента и только одно индексное выражение. Например, обращение к элементу arr2[1][2] можно осуществить с помощью указателя ptr2, объявленного в форме int *ptr2=arr2[0] как обращение ptr2[1*4+2] (здесь 1 и 2 - это индексы используемого элемента, а 4 - это число элементов в строке) или как ptr2[6]. Заметим, что внешне похожее обращение arr2[6] выполнить невозможно, так как указателя с индексом 6 не существует.

Для обращения к элементу arr3[2][3][4] из трехмерного массива тоже можно использовать указатель, описанный как float *ptr3=arr3[0][0] с одним индексным выражением в форме ptr3[3*2+4*3+4] или ptr3[22].

Далее приведена функция, позволяющая найти минимальный элемент в трехмерном массиве. В функцию передается адрес начального элемента и размеры массива, возвращаемое значение - указатель на структуру, содержащую индексы минимального элемента:

```
struct INDEX {   int i,
                 int j,
                 int k } min_index ;

struct INDEX * find_min (int *ptr1, int l, int m int n)
{   int min, i, j, k, ind;
    min=*ptr1;
    min_index.i=min_index.j=min_index.k=0;
    for (i=0; i<l; i++)
        for (j=0; j<m; j++)
            for (k=0; k<n; k++)
                { ind=i*(m*n)+j*n+k;
 if (min>*(ptr1+ind)
   { min=*(ptr1+ind);
    min_index.i=i;
    min_index.j=j;
    min_index.k=k;   }    }
return &min_index;   }
```

1.7.3 Операции с указателями

Над указателями можно выполнять унарные операции инкремент и декремент. При выполнении операций ++ и -- значение указателя увеличивается или уменьшается на длину типа, на который ссылается используемый указатель. Например:

```
int *ptr, a[10];
ptr=&a[5];
ptr++;        /* равно адресу элемента a[6] */
ptr--;        /* равно адресу элемента a[5] */
```

В бинарных операциях сложения и вычитания могут участвовать указатель и величина типа int. При этом результатом операции будет указатель на исходный тип, а его значение будет на указанное число элементов больше или меньше исходного. Например:

```
int *ptr1, *ptr2, a[10];   int i=2;
ptr1=a+(i+4);      /* равно адресу элемента a[6] */
ptr2=ptr1-i;       /* равно адресу элемента a[4] */
```

В операции вычитания могут участвовать два указателя на один и тот же тип. Результат такой операции имеет тип int и равен числу элементов исходного типа между уменьшаемым

и вычитаемым, причем если первый адрес младше, то результат имеет отрицательное значение. Например:

```
int *ptr1, *ptr2, a[10];    int i;
ptr1=a+4;    ptr2=a+9;
i=ptr1-ptr2;      /* равно 5 */
i=ptr2-ptr1;      /* равно -5 */
```

Значения двух указателей на одинаковые типы можно сравнивать в операциях ==, !=, <, <=, >, >=; при этом значения указателей рассматриваются просто как целые числа, а результат сравнения равен 0 (ложь) или 1 (истина). Например:

```
int *ptr1, *ptr2, a[10];
ptr1=a+5;    ptr2=a+7;
if (prt1>ptr2) a[3]=4;
```

В данном примере значение ptr1 меньше значения ptr2 и поэтому оператор a[3]=4 не будет выполнен.

1.7.4 Массивы указателей

В языке СИ элементы массивов могут иметь любой тип, и, в частности, могут быть указателями на любой тип. Рассмотрим несколько примеров с использованием указателей.

Следующие объявления переменных

```
int a[]={10,11,12,13,14,};       int *p[]={a, a+1, a+2, a+2, a+3, a+4};
int **pp=p;
```

порождают программные объекты, представленные на схеме рис. 4.

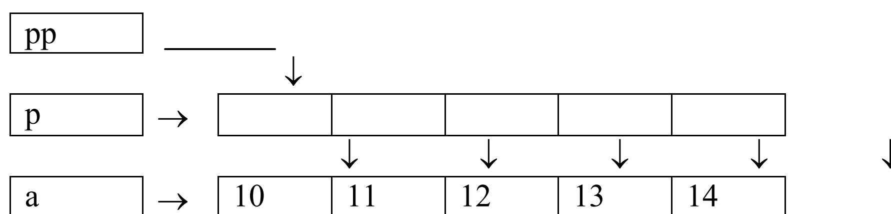


Рис. 4 Схема размещения переменных при объявлении

При выполнении операции pp-p получим нулевое значение, так как ссылки pp и p равны и указывают на начальный элемент массива указателей, связанного с указателем p (на элемент p[0]).

ПОСЛЕ ВЫПОЛНЕНИЯ ОПЕРАЦИИ PP+=2 СХЕМА ИЗМЕНИТСЯ И ПРИМЕТ ВИД, ИЗОБРАЖЕННЫЙ НА РИС. 5.

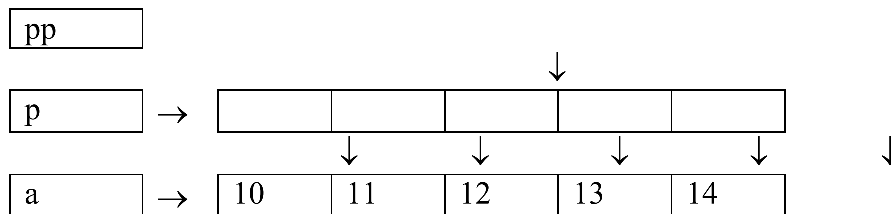


Рис. 5 Схема размещения переменных после выполнения операции pp+=2

Результатом выполнения вычитания pp-p будет 2 , так как значение pp есть адрес третьего элемента массива p. Ссылка *pp-a тоже дает значение 2, так как обращение *pp есть адрес третьего элемента массива a, а обращение a есть адрес начального элемента массива a.

При обращении с помощью ссылки **pp получим 12 - это значение третьего элемента массива a. Ссылка *pp++ даст значение четвертого элемента массива p, т.е. адрес четвертого элемента массива a.

Если считать, что pp=p, то обращение *++pp - это значение первого элемента массива a (т.е. значение 11), операция ++*pp изменит содержимое указателя p[0], таким образом, что он станет равным значению адреса элемента a[1].

Сложные обращения раскрываются изнутри. Например, обращение *(++(*pp)) можно разбить на следующие действия: *pp дает значение начального элемента массива p[0], далее это значение инкременируется ++(*p) в результате чего указатель p[0] станет равен значению адреса элемента a[1], и последнее действие - это выборка значения по полученному адресу, т.е. значение 11.

В предыдущих примерах был использован одномерный массив, рассмотрим теперь пример с многомерным массивом и указателями. Следующие объявления переменных

```
int a[3][3]={   { 11,12,13 },
                { 21,22,23 },
                { 31,32,33 } };
int *pa[3]={ a,a[1],a[2] };
int *p=a[0];
```

порождают в программе объекты, представленные на схеме рис. 6.

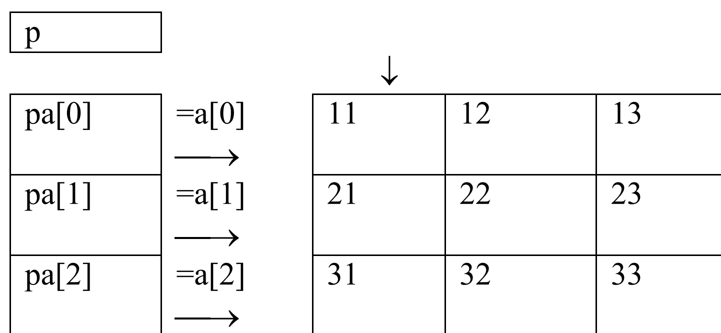


Рис. 6 Схема размещения указателей на двумерный массив

Согласно этой схеме доступ к элементу a[0][0] получить по указателям a, p, pa при помощи следующих обращений: a[0][0], *a, **a[0], *p, **pa, *p[0].

Рассмотрим теперь пример с использованием строк символов. Объявления переменных

```
char *c[]={ "lim", "dx", "yes", "no" };
char **cp[]={ c+3, c+1 , c+2 , c };
char ***cpp=cp;
```

можно изобразить схемой, представленной на рис. 7.

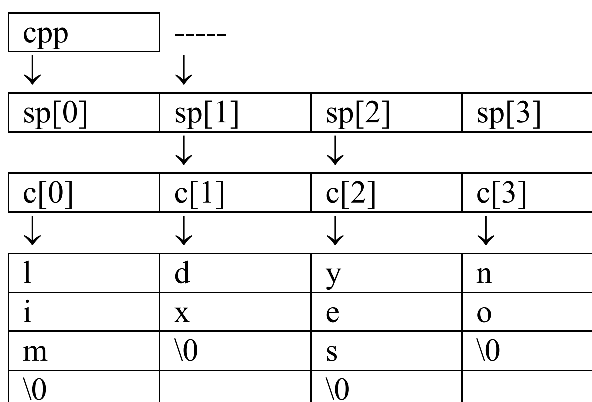


Рис. 7 Схема размещения указателей на строки

Значением выражения cpp[1][1][1] будет символ ‘x’, выражения cpp[0][1][0] и cp[0][0] дадут значение ‘n’. Выполнение операции *(*(*(++cpp)+1)), показанное на рис. 7 пунктиром, увеличит значение указателя cpp на 1 и значением этой операции будет символ ‘e’.

1.7.5 Динамическое размещение массивов

При динамическом распределении памяти для массивов следует описать соответствуюший указатель и присваивать ему значение при помощи функции calloc. Одномерный массив a[10] из элементов типа float можно создать следующим образом:

```
float *a;
a=(float*)(calloc(10,sizeof(float));
```

Для создания двумерного массива вначале нужно распределить память для массива указателей на одномерные массивы, а затем распределять память для одномерных массивов. Пусть, например, требуется создать массив a[n][m], это можно сделать при помощи следующего фрагмента программы:

```
#include <alloc.h>
main ( )
{   double **a;    int n,m,i;
    scanf("%d %d",&n,&m);
    a=(double **)calloc(m,sizeof(double *));
    for ( i=0; i<=m; i++)
    a[i]=(double *)calloc(n,sizeof(double));      }
```

Аналогичным образом можно распределить память и для трехмерного массива размером n,m,l. Следует только помнить, что ненужную для дальнейшего выполнения программы память следует освобождать при помощи функции free:

```
#include <alloc.h>
main ( )
{   long ***a;
    int n,m,l,i,j;
    scanf("%d %d %d",&n,&m,&l);
    a=(long ***)calloc(m,sizeof(long **));   /*   распределение памяти  */
for ( i=0; i<=m; i++)
{   a[i]=(long **)calloc(n,sizeof(long *));
    for ( j=0; i<=l; j++)
    a[i][j]=(long *)calloc(l,sizeof(long));    }
for ( i=0; i<=m; i++)     /*  освобождение памяти  */
    {         for (j=0; j<=l; j++) free (a[i][j]);
              free (a[i]);       }
free (a);      }
```

Рассмотрим еще один интересный пример, в котором память для массивов распределяется в вызываемой функции, а используется в вызывающей. В таком случае в вызываемую функцию требуется передавать указатели, которым будут присвоены адреса выделяемой для массивов памяти:

```
#include <alloc.h>
main ( )
{   int vvod(double ***, long **);
```

```
    double **a; /* указатель для массива a[n][m] */
    long *b; /* указатель для массива b[n] */
    vvod (&a,&b);  /* в функцию vvod передаются адреса указателей, а не
                        их значения */    }
int vvod ( double ***a, long **b)
{   int n, m, i, j;
    scanf (" %d %d ",&n,&m);
    *a=(double **)calloc(n,sizeof(double *));
    *b=(long *)calloc(n,sizeof(long));
    for ( i=0; i<=n; i++)
    *a[i] = (double *)calloc(m,sizeof(double));    }
```

Отметим также то обстоятельство, что указатель на массив не обязательно должен показывать на начальный элемент некоторого массива. Он может быть сдвинут так, что начальный элемент будет иметь индекс отличный от нуля, причем он может быть как положительным так и отрицательным. Например:

```
#include <alloc.h>
int main()
{   float *q, **b;
    int i, j, k, n, m;
    scanf("%d %d",&n,&m);
    q=(float *)calloc(m,sizeof(float)); /* сейчас указатель q показывает на начало массива */
    q[0]=22.3;
q-=5;       /* теперь начальный элемент массива имеет индекс 5, а конечный элемент индекс
   n-5 */
q[5]=1.5;   /* сдвиг индекса не приводит к перераспределению массива в памяти и изменится
   начальный элемент */
    q[6]=2.5;   /* - это второй элемент */
    q[7]=3.5;   /* - это третий элемент */
    q+=5;       /* теперь начальный элемент вновь имеет индекс 0, а значения элементов
q[0], q[1], q[2] равны соответственно 1.5, 2.5, 3.5 */
    q+=2;       /* теперь начальный элемент имеет индекс -2, следующий -1, затем 0 и т.д.
по порядку */
    q[-2]=8.2;
    q[-1]=4.5;
     q-=2; /* возвращаем начальную индексацию, три первых элемента массива q[0], q[1],
q[2], имеют значения 8.2, 4.5, 3.5 */
  q--;   /* вновь изменим индексацию . */
  /* Для освобождения области памяти в которой размещен массив q используется
функция free(q), но поскольку значение указателя q смещено, то выполнение функции free(q)
приведет к непредсказуемым последствиям. Для правильного выполнения этой функции
указатель q должен быть возвращен в первоначальное положение */
  free(++q);
  /* Рассмотрим возможность изменения индексации и освобождения памяти для
двумерного массива */
  b=(float **)calloc(m,sizeof(float *));
  for (i=0; i<m; i++)
  b[i]=(float *)calloc(n,sizeof(float));
  /* После распределения памяти начальным элементом массива будет элемент b[0][0] */
  /* Выполним сдвиг индексов так, чтобы начальным элементом стал элемент b[1][1] */
  for (i=0; i<m ; i++) --b[i];
```

```
b--;
/* Теперь присвоим каждому элементу массива сумму его индексов */
for (i=1; i<=m; i++)
for (j=1; j<=n; j++)
b[i][j]=(float)(i+j);
/* Обратите внимание на начальные значения счетчиков циклов i и j, они начинаются с 1 а не с 0 */
/* Возвратимся к прежней индексации */
for (i=1; i<=m; i++) ++b[i];
b++;   /* Выполним освобождение памяти */
for (i=0; i<m; i++) free(b[i]);
free(b);
return 0;   }
```

В качестве последнего примера рассмотрим динамическое распределение памяти для массива указателей на функции, имеющие один входной параметр типа double и возвращающие значение типа double:

```
#include <alloc.h>
#include <stdio.h>
double cos(double), sin(double), tan(double);
int main( )
{   double (*(*masfun))(double);
    double x=0.5, y;   int i;
    masfun=(double(*(*))(double))
    calloc(3,sizeof(double(*(*))(double)));
    masfun[0]=cos;     masfun[1]=sin;    masfun[2]=tan;
    for (i=0; i<3; i++);
    {         y=masfun[i](x);
    printf("\n x=%g y=%g",x,y);    }
return 0;     }
```

1.8 ДИРЕКТИВЫ ПРЕПРОЦЕССОРА

Директивы препроцессора представляют собой инструкции, записанные в тексте программы на СИ и выполняемые до трансляции программы. Директивы препроцессора позволяют изменить текст программы, например, заменить некоторые лексемы в тексте, вставить текст из другого файла, запретить трансляцию части текста и т.п. Все директивы препроцессора начинаются со знака #. После директив препроцессора точка с запятой не ставится.

1.8.1 Директива #include

Директива #include включает в текст программы содержимое указанного файла. Эта директива имеет две формы:

```
#include "имя файла"          #include <имя файла>
```

Имя файла должно соответствовать соглашениям операционной системы и может состоять либо только из имени файла, либо из имени файла с предшествующим ему маршрутом. Если имя файла указано в кавычках, то поиск файла осуществляется в соответствии с заданным маршрутом, а при его отсутствии в текущем каталоге. Если имя файла задано в угловых скобках, то поиск файла производится в стандартных директориях операционной системы, задаваемых командой PATH.

Директива #include может быть вложенной, т.е. во включаемом файле тоже может содержаться директива #include, которая замещается после включения файла, содержащего эту директиву. Она широко используется для включения в программу, так называемых, заголовочных файлов, содержащих прототипы библиотечных функций, и поэтому большинство программ на СИ начинаются с этой директивы.

1.8.2 Директива #define

Директива #define служит для замены часто использующихся констант, ключевых слов, операторов или выражений некоторыми идентификаторами. Идентификаторы, заменяющие текстовые или числовые константы, называют именованными константами. Идентификаторы, заменяющие фрагменты программ, называют макроопределениями, причем макроопределения могут иметь аргументы. Директива #define имеет две синтаксические формы: #define идентификатор текст, #define идентификатор (список параметров) текст. Эта директива заменяет все последующие вхождения идентификатора на текст. Такой процесс называется макроподстановкой. Текст может представлять собой любой фрагмент программы на СИ, а также может и отсутствовать. В последнем случае все экземпляры идентификатора удаляются из программы, например,

#define WIDTH 80 #define LENGTH (WIDTH+10)

Эти директивы изменят в тексте программы каждое слово WIDTH на число 80, а каждое слово LENGTH на выражение (80+10) вместе с окружающими его скобками.

Скобки, содержащиеся в макроопределении, позволяют избежать недоразумений, связанных с порядком вычисления операций. Например, при отсутствии скобок выражение t=LENGTH*7 будет преобразовано в выражение t=80+10*7 , а не в выражение t=(80+10)*7 , как это получается при наличии скобок, и в результате получится 780, а не 630.

Во второй синтаксической форме в директиве #define имеется список формальных параметров, который может содержать один или несколько идентификаторов, разделенных запятыми. Формальные параметры в тексте макроопределения отмечают позиции на которые должны быть подставлены фактические аргументы макровызова. Каждый формальный параметр может появиться в тексте макроопределения несколько раз.

При макровызове вслед за идентификатором записывается список фактических аргументов, количество которых должно совпадать с количеством формальных параметров, например,

#define MAX(x,y) ((x)>(y))?(x):(y)

Эта директива заменит фрагмент t=MAX(i,s[i]); на фрагмент t=((i)>(s[i])?(i):(s[i]); .

Как и в предыдущем примере, круглые скобки, в которые заключены формальные параметры макроопределения, позволяют избежать ошибок, связанных с неправильным порядком выполнения операций, если фактические аргументы являются выражениями. Например, при наличии скобок фрагмент t=MAX(i&j,s[i]||j); будет заменен на фрагмент t=((i&j)>(s[i]||j)?(i&j):(s[i]||j);, а при отсутствии скобок - на фрагмент t=(i&j>s[i]||j)?i&j:s[i]||j;, в котором условное выражение вычисляется в совершенно другом порядке.

1.8.3 Директива #undef

Директива #undef используется для отмены действия директивы #define. Синтаксис этой директивы следующий: #undef идентификатор. Она отменяет действие текущего определения #define для указанного идентификатора. Не является ошибкой использование директивы #undef для идентификатора, который не был определен директивой #define, например, #undef WIDTH #undef MAX. Эти директивы отменяют определение именованной константы WIDTH и макроопределения MAX.

2 ОПИСАНИЕ ЯЗЫКА C++

Язык C++ разработан в начале 80-х гг., как объектно-ориентированное расширение языка СИ. Благодаря совместимости с СИ снизу вверх и эффективной реализации на большинстве типов компьютеров, язык C++ стал основным языком разработки как системных, так и прикладных программ.

Объектно-ориентированное программирование (ООП) предполагает использование отличных от традиционных методов разработки и реализации программ. Хорошо развитые традиционные методы процедурного программирования базируются на модели построения программы, как совокупности функций. Приемы программирования объясняют, как разрабатывать, организовывать и реализовать функции, составляющие программу.
Функциональная декомпозиция определяет функцию как некоторую абстрактную операцию, которая решает часть задачи, а приемы структурного программирования делают реализацию функции логичной и понятной.

В объектно-ориентированной модели разработки программ используется абстракция данных, состоящая в том, что структура данных определяется операциями над этой структурой в большей степени, чем особенностями реализации самой структуры. При абстракции данных структура данных включается внутрь создаваемого абстрактного типа данных, называемого объектом (инкапсюлируется), а доступ к данным возможен с помощью набора операций, которые тоже являются частью объекта. Абстракция данных углубляет процедурный подход к программированию, рассматривая функцию как абстрактную операцию.

ООП базируется на модели построения программы, как набора объектов абстрактного типа, соответствующих объектам решаемой задачи. Операции в объектных типах, как и функции при процедурном программировании, представляют собой абстрактные операции, решающие задачу.

2.1 Новые возможности C++, не связанные с ООП

В C++ кроме дополнений, связанных с ООП, имеются дополнения, расширяющие возможности языка как при использовании приемов ООП, так и без них. Рассмотрим вначале эти дополнения.

2.1.1 КОММЕНТАРИИ

В C++ кроме комментариев принятых в С, т.е. выделенных символами /* и */, можно использовать однострочные комментарии, начинающиеся с символов // и продолжающихся до конца строки.

К однострочным комментариям понятие вложенных комментарием неприменимо, так как последовательность символов между // и концом строки не анализируется.

2.1.2 ОБЪЯВЛЕНИЕ ДАННЫХ ВНУТРИ БЛОКА

Язык СИ требует, чтобы все описания переменных внутри блока помещались вначале блока до первого исполняемого оператора. В C++ можно объявлять переменные в любой точке блока, но до их использования. Очень удобно использовать, например, такое объявление

```
for ( int i=0; i<n;  i++)   {.............}
```

Но следует помнить, что объявление, сделанное в заголовке цикла, будет действовать до конца блока, содержащего цикл, а не только в теле цикла.

В приведенном примере объявление используется как выражение, но не следует использовать подобный прием для ссылочных выражений. Например, выражения k = int i =0; printf (“\n%d”, int i=6); будут восприниматься как ошибочные.

Модификатор const как в СИ так и в С++ запрещает изменять значение переменной и требует ее инициализации. В С++, кроме того, этот модификатор делает недоступной переменную из других исходных файлов.

2.1.3 ОПЕРАТОР ОБЛАСТИ ВИДИМОСТИ

В С++ , как и в СИ, локальные и глобальные переменные могут иметь одинаковые имена. В этом случае в СИ невозможно обратиться к глобальной переменной, если в рассматриваемом блоке имеется локальная переменная с таким же именем. В С++ для доступа к такой переменной используется специальный оператор разрешения области видимости :: (двойное двоеточие), записанный перед именем переменной. Например:

```
int sum;          //  это глобальная переменная sum
int f()           //  начало блока f
{  int sum;              //  это локальная переменная sum в блоке f
   sum++;                //  увеличение локальной переменной sum
   ::sum++;   }   //  увеличение глобальной переменной sum
```

2.1.4 Операторы new и delete

Операции выделения и освобождения памяти при программировании на СИ и С++ используются довольно часто, поэтому в С++ для этих целей существуют специальные операторы new и delete, которые работают аналогично функциям malloc и free. Оператор new выделяет блок памяти, необходимый для размещения переменной или массива и возвращает указатель на этот блок. В операторе new необходимо указывать тип переменной, а для массива еще и число элементов. Например:

```
int  *t, *s;
t=new int;  s=new int [10];
```

указатель t будет указывать на переменную типа int, а указатель s на массив из 10 элементов типа int.

Вновь созданной переменной можно присвоить начальное значение или список значений, этот список указывается в скобках после типа:

```
int  *p,  *r;    p=new int (28);     r=new int [3] (12,45,9);  .
```

В случае, если имеющейся памяти недостаточно, оператор new возвращает указатель на нулевое значение (void *)0. Оператор delete освобождает ранее занятую память. Например, delete p; delete r; .

2.1.5 ПРОТОТИПЫ ФУНКЦИЙ И АРГУМЕНТЫ ПО УМОЛЧАНИЮ

В СИ наличие прототипов функций необязательно, в С++ до обращения к функции следует поместить либо определение функции, либо ее прототип. Причем, если в СИ пустые скобки в прототипе обозначают, что функция может иметь любое число аргументов, то в С++ это значит отсутствие аргументов у функции.

В С++ в прототипах функций можно задавать значения переменных присваиваемые по умолчанию, т.е. в случае отсутствия значений при вызове функции. Например, пусть функция circ рисует окружность заданного радиуса, с центром в заданной точке. Если записать прототип этой функции в виде void circ (int x=100, int y=50, int r=20);, то вызовы

этой функции будут восприниматься по-разному, в зависимости от количества передаваемых аргументов,

вызов функции	полный набор аргументов
circ()	circ(100,50,30)
circ(200)	circ(200,50,30)
circ(200,150)	circ(200,150,30)
circ(, , 40)	ошибка

Значения по умолчанию можно задавать и опускать для всех аргументов, но только слева направо, поэтому в последнем примере вызов circ (, , 40) считается ошибкой. Прототип вида circ (int x, int y=50, int r=40) тоже будет ошибочным.

2.1.6 ПРЕОБРАЗОВАНИЕ ТИПОВ

В С++ явное преобразование типов можно записать и в операторной форме, как в СИ, и в функциональной. В последнем случае имя типа синтаксически используется как имя функции. Операторная форма (int)a эквивалентна функциональной форме int (a). В некоторых случаях функциональная форма упрощает выражение, уменьшая количество скобок. Например, double (long (a+3)) выглядит лучше, чем (double)((long) a+3).

2.1.7 ССЫЛКИ

В языке СИ параметры функций передаются только по значению, если же в вызываемой функции требуется доступ к переменной из вызывающей функции, то в качестве параметра необходимо передавать адрес переменной, а параметр должен быть указателем. В С++ реализованы два метода передачи параметров: и по значению (как в СИ), и по адресу. В последнем случае соответствующий параметр функции должен быть объявлен как ссылка, для чего используется модификатор &, действующий аналогично модификатору * при объявлении указателя. Объявление int & a читается как «a есть ссылка на int». Например:

```
void    swap ( int & x,  int & y)
        {   int z=y;          y=x;    x=z;    }
        main()
        {   int a=3,b=5;
            swap(a,b);    }
```

Поскольку параметры функции swap объявлены как ссылки, при вызове функции изменятся значения переменных a и b в функции main.

При использовании ссылок необходимо следить за соответствием типа ссылки при объявлении функции и типа используемого фактического параметра. При их несовпадении компилятор создает временную переменную нужного типа, копирует в нее значение фактического параметра, а вызываемая функция работает с этой временной переменной. Поэтому вызов float t=3, r=8; swap(t,r); не изменит значений переменных t и r, а будет эквивалентен следующему фрагменту: float t=3, r=8; int temp1=t, temp2=r; swap(temp1,temp2);

Передача аргументов по ссылке полезна, когда параметры представляют собой "объемистые" структуры данных, копирование которых может занимать длительное время. В таких случаях можно исключить возможность неумышленного изменения параметра внутри функции, если описать его с модификатором const:

```
Struct s1 {     //  большая структура    };
void f(const s1 &par);
```

При попытке внутри функции f изменить какое-нибудь поле структуры s1 компилятор выдаст сообщение об ошибке.

Возвращаемое значение функции тоже может быть ссылкой, причем такую функцию можно использовать в левой части операции присваивания:

```
int & ptintint (int &x)
{    printf ("\n%d",x);
     return x;   }
viod main()
{    int n=2,m=8;
     printint(m)=n;     }
```

Поскольку printint(m) возвращает ссылку на m, а присвоение значения ссылке эквивалентно присвоению значения тому объекту, на который она ссылается, то m будет иметь такое же значение, что и n.

Использование ссылки как возвращаемого значения особенно полезно при "конвейерных" вызовах f1(f2(f3(par))), когда каждая последующая функция должна обрабатывать сам объект, а не его копию.

Ссылочный тип можно использовать и для создания псевдонимов переменных. При объявлении переменной ссылочного типа обязательна инициализация объектом соответствующего типа. Использование переменной ссылочного типа дает такой же эффект, что и прямое использование переименованного объекта, например, int i;

```
int &i1=i;       i1=5;  // то же что и i=5;
```

Если тип ссылки и тип переменной, на которую она ссылается различен, то создается анонимная переменная с указанным псевдонимом, которой после преобразования присваивается значение существующей переменной. Например:

```
int   r = 128;
char  &t = r;        // создается анонимная переменная типа char
                 //  доступная с помощью ссылки t, значением
                 //  этой переменной будет 128 преобразованное
                 /  к типу char т.е. 0
t=10;        // значение r не изменится
```

2.1.8 ПОДСТАВЛЯЕМЫЕ ФУНКЦИИ

В обычном СИ для уменьшения расходов на вызовы небольших функций используются макросы с параметрами, однако их применение зачастую приводит к труднообнаруживаемым ошибкам. В С++ есть другое средство для того, чтобы вместо вызова функции осуществить ее подстановку. Если функцию описать с ключевым словом inline, то такая функция будет называться подставляемой, и если это возможно, то везде вместо ее вызова будет подставлено ее тело. Определение подставляемой функции должно быть выполнено до ее первого вызова. Например:

```
inline int quadro (int x)
{       return  x*x;      }
b=2;    a=quadro(b);   c=quadro(b++);
```

Сравним с макроопределением вида

```
#define quadro1(x)     ((x)*(x))
  b=2;    a=quadro1(b);           c=quadro1(b++);
```

ЗАМЕТИМ, ЧТО ВЫЗОВ QUADRO1(B++) В ИТОГЕ ДАСТ ((B++)*(B++)), Т.Е. ЗНАЧЕНИЕ В УВЕЛИЧИТЬСЯ НА 2, А НЕ НА 1, КАК ЭТО ДЕЛАЕТСЯ ПРИ ВЫЗОВЕ QUADRO(B++).

Кроме того подставляемая функция, как и всякая другая в С++, обеспечивает контроль типов, а макрос этой возможности не имеет. Сделать функцию подставляемой не всегда возможно, в частности, функции, содержащие операторы if, case, for, while, goto, не могут быть подставляемыми, но объявление их как inline не приводит к ошибке. Такая функция просто рассматривается компилятором как статическая.

2.1.9 ПЕРЕГРУЖАЕМЫЕ ФУНКЦИИ

В С++ можно создавать несколько различных функций с одинаковыми именами, но разными количеством и типами параметров. Обычно эта возможность широко используется для выполнения одинаковых по смыслу действий над объектами различных типов. Например, для печати значений разного типа можно разработать функции

```
void print ( int i )         { printf ("\n%d",i); }
void print ( double d ) { printf ("\n%lf",d); }
void print ( char * s )  { printf ("\n%s",s); }
```

В вызывающей функции компилятор сам выберет одну из функций print основываясь на анализе типов аргументов

```
int i=5;double pi=3.141592;
print (i);      print(pi);      print ("перегрузка функций");
```

При разработке перегружаемых функций следует помнить, что различия только в типе возвращаемого значения недостаточно.

Очевидно, что три приведенные функции print в объектном модуле будут соответствовать трем функциям с различными именами, модифицированные транслятором имена функций содержат информацию о количестве и типах параметров, причем модификацию имен компилятор выполняет для всех функций, а не только перегружаемых. Для того, чтобы функцию С++ можно было бы вызвать из программы на СИ, необходимо запретить модификацию имен, для чего функцию нужно описать с описателем extern "СИ":

```
extern "СИ" { int fun2(int);   double fun3(double);    }
```

Очевидно, что функции, описанные как extern "СИ", не могут быть перегружаемыми.

2.1.10 ПЕРЕГРУЗКА ОПЕРАТОРОВ

При определении новых типов данных возникает необходимость выполнять некоторые действия над такими данными. В СИ для этой цели используются функции, в С++ кроме этой возможности можно разработать свои операции над введенными типами данных, используя для обозначения таких операций привычные знаки, такие как +, - и т.п. Для перегрузки оператора следует описать функцию с именем operator@, где @ знак любой операции кроме . ., :: ?; . Объявленная функция должна описывать необходимые действия, а вызов этой функции осуществляется не как функции, а как операции обозначаемой знаком @. Например, введем тип данных complex для комплексных чисел:

```
typedef struct {  double re, im;  } complex;
```

и введем операции над комплексными числами

```
complex  operator+ (complex a, complex b)  //  сложение
{        return complex ( a.re+b.re,  a.im+b.im);   }
complex  operator*  (complex a, complex b) //  умножение
{        return complex  (       a.re*b.re - a.im*b.im,
                                 a.re*b.im+a.im*b.re  );     }
complex operator/   (complex a, complex b)  //   деление
{        double r=b.re,   i=b.im,   ti,  tr;
         tr=fabs(r);       ti=fabs(i);
         if  ( tr<=ti )    {        ti=r/i;   tr=i*(1+ti*ti);
                                 r=a.re;  i=a.im; }
         else      {        ti=-i/r;  tr=r*(1+ti*ti);  r=-a.im;         i=a.re; }
         return complex ( (r*ti+i)/tr,  (i*ti-r)/tr  );
```

Далее для вызова этих операций можно использовать выражения

```
complex a,b,c;
a=b+a/c;        b=a+12;
```

В ПОСЛЕДНЕЙ СТРОКЕ ОПЕРАНДЫ СЛОЖЕНИЯ РАЗЛИЧНОГО ТИПА, НО ПОСКОЛЬКУ ДЛЯ ОДНОГО ИЗ ОПЕРАНДОВ ИМЕЕТСЯ ПЕРЕГРУЖЕННЫЙ ОПЕРАТОР, ДЛЯ ВЫПОЛНЕНИЯ ОПЕРАЦИИ СОЗДАЕТСЯ АНОНИМНЫЙ ОБЪЕКТ ТИПА COMPLEX, КОТОРОМУ ПРИСВАИВАЕТСЯ ЗНАЧЕНИЕ (12,0).

Одну и ту же операцию можно перегружать для переменных различного типа; очевидно, что для переменных тех типов, для которых эта операция не перегружена, будут использованы обыкновенные операции.

2.2 ИСПОЛЬЗОВАНИЕ КЛАССОВ

Классы в C++ поддерживают несколько путей организации программ и управляющих зависимостей между структурами данных и функциями. Приемы программирования с использованием классов широко используются в C++, поскольку классы являются основным понятием объектно-ориентированного программирования.

2.2.1 Классовые типы

Классы - это сложные структуры данных, определяемые пользователями. Они могут содержать как элементы данных, представляющих тип, так и функциональные элементы, реализующие операции над типом. Части класса могут быть скрыты или сделаны общедоступными при помощи деклараций private и public. Элементы в секции private доступны только через функциональные элементы этого класса или через другие функции, объявленные как дружественные (friend) классу. Используя этот механизм сокрытия информации, класс может инкапсулировать структуру данных так, что только специфицированные классом функции могут ее использовать.

Представим реализацию сортировки строк с использованием классового типа String для представления строк. Признак класса String - это имя типа класса, и может использоваться также как имя, заданное при помощи typedef.

```
#include <stdio.h>
#include <string.h>
    class String
    {  char *str;
       public:    String () { str =new char[1] ;*str=0;}
                  String (char *);
                  void  print ( )  {  printf ("%s",str);  }
                 friend int operator < (String s1,String s2)
                 {   return strcmp (s1.str, s2.str ) < 0;   }     };
```

Структура данных, использованная для представления строки, не изменилась; это указатель на завершающийся нулем массив символов. Тем не менее, сейчас этот указатель скрыт как элемент str в приватной части класса String. Только конструкторы String (элемент-функция print и дружественная функция operator <) имеет доступ к представлению строки. Все элементы-функции объявлены в секции public класса и, соответственно, общедоступны.

Имена первых двух функциональных элементов совпадают с именем класса, это значит, что данные функции являются конструкторами для класса. Когда создается переменная типа String, конструктор используется для ее инициализации. Первый конструктор String не имеет аргументов и инициализирует str как указатель на пустую строку. Оператор new используется для выделения в памяти места под строку. Так как эта функция коротка, ее тело

определяется внутри определения класса. Это простейший способ сделать элемент или friend-функцию inline-вида.

Второй конструктор только объявлен внутри класса, поэтому его определение должно быть дано где-то в другом месте.

```
String::String(char *s)
{ str=new char[strlen(s)+1]; strcpy(str,s); }
```

Для элемента-функции, определяемого вне класса, имя функции должно быть квалифицировано, чтобы показать, что упоминаемая функция находится в области видимости класса. Здесь квалификатор области видимости String:: указывает, что это определение элемента String.

Функции-элементы всегда находятся в области видимости их класса. Поэтому элементы объекта класса, с которыми оперируют элементы-функции, доступны без использования оператора доступа к элементам. Обратите внимание, что String и print обращаются к str без квалификации. Напротив, дружественная функция

```
friend int operator < (String s1, String s2)
{ return strcmp(s1.str,s2.str)<0;}
```

должна использовать оператор доступа к элементу для получения str своих аргументов. Дружественный статус разрешает доступ к приватному элементу str, но не изменяет область видимости функции.

Операторные функции - это способ перегрузки предопределенных в языке операторов для применения к операндам типа классов. Здесь оператор < определяется для сравнения двух ссылочных элементов String. Инфиксный оператор < теперь можно использовать для двух операндов типа String. Завершая пример с сортировкой строк, мы увидим, как можно использовать тип String. В функции, читающей строки в список, конструктор вызывается для создания объекта класса String из символов во входном буфере. Затем объект типа String присваивается слоту в массиве-списке:

```
void   input(String *a, int limit, int &i)
{   static char buffer [100];
    for (i=0; i<limit; i++)
            if (scanf("%s", buffer) == EOF) break;
            else a[i] = String(buffer);  }
```

В процедуре, выдающей список, функция print, которая используется для выдачи String-ов, вызывается с каждым элементом списка с применением . (точки) - оператора доступа к элементу:

```
void output(String *a, int size)
{   printf("\n list:");
    for (int i =0; i <size; i++) a[i].print();
    printf("\n ");   }
```

В функции, сортирующей список, operator < вызывается для сравнения элементов списка в инфиксной нотации, и выглядит точно так же, как сравнение целых чисел:

```
void sort(String *a, int n)
{   int changed;
    do{     changed = 0;
            for (int i = 0; i<n -1; i++)
            if (a[i+1]< a[i])
        {        String temp = a[i];
                 a[i] = a[i+1];   a[i+1] = temp;   changed = 1;  }
      } while(changed);    }
```

И в завершение, функция main читает, сортирует и выводит список:

```
const int max = 10;   String list[max];
int size = 0;
```

```
main ()
{    input(llst, max, size);    sort(list, size);    output(list, size);    }
```

Массив списка изначально состоит из String-ов, а не инициализирован нулями. Хотя для этой программы это не имеет значения, однако автоматическая инициализация объектов класса через конструкторы делает программы корректнее, так как структуры данных, инкапсулированные в объекте, с самого начала автоматически находятся в корректном состоянии.

2.2.2 Компоненты данных

Определение классового типа, как в ситуации со String в предыдущем разделе, служит шаблоном для объектов классового типа. Когда создается объект классового типа, выделяется пространство для объекта, а также создаются и инициализируются элементы данных как компоненты объекта. Также как и другие объекты данных в C++, объекты классового типа могут быть созданы в разных местах и существовать разные периоды времени. Внутрифайловые или локальные статические объекты создаются перед началом выполнения программы и существуют до ее окончания; параметры и локальные объекты создаются в случаях, когда выполнение программы достигнет точки их объявления и существуют до тех пор, пока блок, в котором они объявлены, существует; динамически создаваемые с помощью оператора new объекты существуют, пока не будут явно уничтожены оператором delete. Временные объекты создаются также для получения промежуточных результатов вычисления выражения и возвращаемых функциями значений.

Доступ к компонентам класса возможен при помощи операторов доступа . (точка) и ->. Оператор . (точка) используется с классовыми объектами.

```
String s;    s.sti;
```

Оператор **->** используется с указателями на классовые объекты.

```
String *sp =new String;
sp->str;
```

Компоненты данных класса могут быть любого предопределенного в языке типа, предварительно определенного классового типа, указателями на классовый тип или ссылкой на классовый тип. Для указателей и ссылок на классовый тип не нужно, чтобы такой класс был определен, нужно только, чтобы имя класса было объявлено. Например:

```
class Node; // объявляется имя;
Node *n ; //оно используется для объявления указателя.
```

Используя указатель или ссылку на классовый тип как элемент класса, можно строить рекурсивные классовые структуры. Например, вершина бинарного дерева может содержать указатели на правого и левого потомка:

```
class Node
{    Node *left, *right;    // и т.д.    };
```

Каждый компонент данных в объекте класса занимает определенное место, необходимое для представления его типа. Между элементами классового объекта могут быть буферные промежутки, зависящие от необходимого для разных машин выравнивания.

Один из способов экономии места при использовании объекта - использование одного и того же пространства для размещения более чем одного компонента. Есть классы, элементы данных в которых размещаются непоследовательно, но с перекрытием, имея начальный адрес, совпадающий с начальной позицией в объекте. Такой тип класса называется union. Спецификатор union заменяет class в объявлении типа. Элементы объединенного типа union все общедоступны (public):

```
union un_type
{   int i;    double d;    char *p:    };
```

Когда объект объединенного типа реализуется, его размер и выравнивание

устанавливается так, что все элементы занимают в объекте одно и то же место. На элементы union можно сослаться, используя оператор доступа к элементу:

```
un_type u;
u.i=1;   // в u запомнили значение типа int;
u.d-3.1415; // перекрыли значением типа double;
char c=*u.p // Опасность! Может быть неверное значение указателя.
```

Достаточно опасно запоминать элементы union как данные одного типа, а использовать как данные другого. Допустимое значение одного элемента может не быть допустимым для другого. Для корректного использования union обычно необходимо обладать информацией, определяющей, какой элемент используется. Union можно сделать элементами классов, если другие элементы содержат информацию о том, как интерпретировать элементы union.

```
enum {    ISINT,
          ISDOUBLE,
          ISCHARSTAR };
class Node
{   Node  *left,  *rlght;
    public: int code;
    un_type info;          } *np;
```

в этом примере code в объекте Node определяет способ интерпретации элемента союза.

```
int i=0; double d=0;  char *p=0;
switch(np->code)
{   case ISCHARSTAR: p=np->info.p; break;
    case ISDOUBLE: d=np->nfo.d', break;
    case ISINT: i=np->inf o.i; break;      }
```

Так как Node содержит одновременно информацию только одного типа, используем союз для наложения объектов разных типов, экономя пространство, занимаемое объектом Node.

Совершенно аналогично элементы класса можно сделать перекрывающимися, не добавляя явного доступа к элементу союза, но вводя в класс анонимный союз. Анонимный союз не имеет признака и не объявляет имя элемента.

```
class Node
{   Node *left,*right;
    public:   int code;
    union
    {       int i;
            double d;
            char *p;    };       } *np;
```

рдесь i, d и p перекрываются, но к ним можно получить прямой доступ как к элементам Node.

```
int i=0;  double d=0;  char *p=0;
switch(np->code)
{   case ISCHARSTAR: p=np->p; break;
    case ISDOUBLE: d=np->d; break;
    case ISINT: i=np->i; break;    }
```

Элементы данных, объявленные как static, используются всеми объектами классового типа. Элемент, объявленный static, создается при определении класса и существует раньше любого объекта класса. На него можно ссылаться как на элемент данных или, не используя объект класса, с помощью оператора доступа (двойное двоеточие). Так как элементы данных, объявленные static, не зависят ни от какого конкретного объекта класса, можно брать их адрес и заводить указатели на них как на любой другой static- объект.

Как пример использования элемента данных static, предположим, что нам необходимо проследить, какое количество объектов класса String уже создано. Для подсчета static

элемент String будет увеличиваться каждым конструктором на 1:

```
class String
{    char *str;  static int count;
     public:   String () { count++; str=new char; *str=0;}
               String(char *s) [count ++; str=new char [strlen(s)+1];
strcpy(str,s);  }
friend void report();    //и т.д.    }
```

Для каждого объекта типа String, инициализируемого конструктором, один и тот же статический элемент count увеличивается на 1. Для определения количества созданных объектов типа String в любой точке программы, функция с соответствующим разрешением доступа может быть построена с использованием ссылки String: :count, как показано ниже в report:

```
void report()
{   printf("Доклад об использовании String:");
printf("%d String-ов создано \n",String::count);          }
```

Удобство использования классов состоит в том, что списки enum можно сгруппировать так, что символьные значения, определяемые ими, находятся в области видимости класса. Сгруппированные значения enum доступны как элементы класса или с использованием оператора видимости/

```
class Node
{   Node *left,*right;
    public:    enum {ISINT, ISDOUBLE, ISCHARSTAR};
    int code;
    union    {       int i;    double d;    char *p;    }     } *np;
```

в этом примере значения enum локальны по отношению к классу. Они не конфликтуют с именами в разных областях видимости, и доступ к ним можно защитить, поместив их в приватную часть. На значения элементов enum класса можно ссылаться также, как и на статические элементы.

```
switch(np->code)
{ case Node::  ISCHARSTAR; p=np->p; break;
  case Node::  ISDOUBLE; d=np->d; break;
  case Node::  ISINT; i=np->i; break;    }
```

2.2.3 Функциональные компоненты

Функция, объявленная внутри определения класса и не специализированная как friend, является функциональной компонентой класса, функциональная компонента (элемент функция) может быть объявлена внутри класса; в этом случае она безоговорочно является inline. Для вызова элементов функций используются операторы доступа . и ->. Оператор . (точка) используется с классовыми объектами:

```
String s;
s.print();
```

Оператор -> используется с указателями на классовые объекты:

```
String *sp=new String;
*sp->print();
```

Как и для простых функций, для функциональных компонент осуществляется проверка вызова при компиляции. Для идентификации функции используется тип классового объекта; фактические аргументы сопоставляются с типом параметров в объявлении функции.

Элементы-функции оперируют объектами того классового типа, с которыми они вызываются. Указатель на этот объект является скрытым аргументом всех функциональных компонент, на них можно напрямую сослаться в определении функции как на this. В

функциональной компоненте не нужно использовать this. Тем не менее, this косвенно используется для ссылок на элементы.

```
class String
{   char *str; public:
    void  print() { printf("%s",str);}       };
```

ссылка на элемент str в print представляет собой то же самое, что и this->str. При вызове print содержимому this присваивается адрес объекта, использованного в вызове. Например, s.print() устанавливает this в значение &s, поэтому внутри print str представляет собой то же, что и (&s)->str или s.str. В вызове sp->print(), this равно sp и доступный элемент определен как sp->str.

Все функциональные элементы логически помещаются внутри области видимости их класса. Если определение элемента функции находится вне описания класса, область видимости функции надо указывать при помощи оператора видимости с именем класса:

```
class X { int f(); };
int X:: f() {/* и т.д. */ }
```

Вложенность элементов-функций внутри класса ведет к различиям в правилах доступа для функциональных компонент и функций, не являющихся элементами. В функциональной компоненте объявление идентификатора сначала рассматривается в пределах видимости блока, затем класса и, наконец, файла. Таким образом, объявление элемента могут перекрывать объявление в файле. В следующем примере идентификатор x в функции X::f() относится к элементу, а не к переменной файла x:

```
class X       { int x; int f();      };
int x;
int  X::f() { return x: }// возвращает this->x
```

Для доступа к объявлению идентификатора файла, перекрытому локальным объявлением или объявлением в классе, можно использовать оператор видимости. Для изменения нашего примера так, чтобы X::f() возвращала значение переменной файла, а не элемента, используем :: для индикации, что x является переменной файла:

```
int X::f() { return ::x; }// возвращает x из файла
```

Области видимости класса - это подобие блока вокруг функциональной компоненты. Объявление функции внутри блока может перекрыть объявление компоненты точно так же, как объявление внутреннего блока может перекрыть объявление внешнего. В следующем случае X::f() возвращает значение локальной переменной:

```
int X::f() { int x=3; return x; }
```

Используя оператор видимости, можно изменить пример так, чтобы возвращался элемент X::x или переменная файла ::x.

2.2.4 Операторные функции

Предопределенные в языке операторы могут быть перегружены для работы с операндами классового типа. Это делается при помощи введения операторной функции, которая берет всего один аргумент классового типа. Объявления и определения операторных функций синтаксически такие же, как и для других функций, за исключением имени, имеющего форму operator x, где x – символ перегружаемого оператора. Определяемые пользователем операторы можно вызывать в инфиксном синтаксисе – обычном для символа x операнды выражения являются аргументами для вызова функции. Так как синтаксис их вызова не меняется, операторные функции должны иметь то же число операндов, что и в предопределенной версии оператора. Перегружаемые операторы имеют тот же приоритет, что и соответствующие встроенные.

Взаимосвязи предопределенных операторов, такие как эквивалентность a+=b и a=a+b или a[b] и *(a+b), или (&p)->x и p.x не сохраняются для определенных пользователем

операторов, если только операторные функции не реализованы таким образом. Результат операторной функции может быть абсолютно не связан с результатом предопределенных версий. В частности, определенные пользователем ++ и -- не могут вести себя полностью аналогично предопределенным версиям. Нет способа определить другие версии этих операторов в префиксной и постфиксной формах, так как один и тот же результат будет выработан независимо от того, как используются операторы, разработанные пользователем.

Предположим, что добавлены операторы к классу complex:

```
Class complex {     // и т.д.
complex operator++();
complex operator +=(complex);        };
```

Одна и та же версия ++ будет использована для префиксной и постфиксной формы, вырабатывая один результат:

```
Complex c1,c2;
c2=c1++; // то же. Что и c2 = ++c1;
```

Оператор += можно определить так, что c1 += c2 будет эквивалентно c1=c1+c2, а можно и иначе, хотя, возможно, это и будет выглядеть странным. Так как класс complex должен работать как числовой тип, лучше реализовывать += так, чтобы он действовал аналогично встроенным арифметическим операторам.

Операторные функции могут быть, а могут и не быть элементами класса, за исключением операторов (), [] и ->, которые могут быть только функциональными компонентами. Если операторная функция является элементом класса, неявный аргумент this является ее первым операндом. В классе String operator <, который реализован как friend, т.е.:

```
Class String
{   char *str;
    public:    friend int operator < (String s1, String s2)
    { return strcmp(sl.str, s2.str)<0;}     // и т.д.    };
```

может также быть реализован как элемент.

```
Class String
{   char *str;
    public:
    int operator < (String s2) { return strcmp(str, s2.str)<0;}    //и т.д.    };
```

Если операторная функция является элементом, первый операнд должен всегда соответствовать типу класса, преобразования типа при этом не допускаются. Операторные функции часто предполагаются работающими аналогично предопределенным в языке версиям, в которых преобразования применяются к каждому операнду. В связи с этим операторные функции часто реализуются как friend.

Операторные функции – это функции со специфическими именами, оперирующие аргументами классового типа. Они необязательно должны вызываться в инфиксной нотации, можно вызывать их и как любую другую функцию. Для функций, не являющихся элементами класса, все аргументы передаются в списке аргументов:

```
Class String
{   friend int operator < (String, String);    //ит.д.     };
void sort(String *a, int n)
{  …  if (operator < (a[i+1],a[i]))          }
```

Для функциональных компонент используется оператор доступа к элементу.

```
Class String
{    int operator < (String);   // ит.д.     };
void sort(String *a, int n)
{   if (a[i+1].operator < (a[i]))       }
```

2.2.5 Защита доступа и дружественные функции

Элементы класса могут быть как public (общедоступными), так и private (доступ к элементам и дружественным функциям защищен). Доступность элементов определяется их объявлением в секции определений класса, озаглавленной меткой public или private. Эти метки могут появляться в определении класса любое число раз и в любом порядке. Первая секция определения класса начинается как private, пока метка не укажет другой уровень защиты.

Есть и третий уровень защиты в C++, играющий роль при использовании наследования классов. Элемент может специфицироваться как protected, точно также как public или private.

Функции, объявленные дружественными в определении класса, не являются элементами класса, но имеют разрешение на доступ к приватным элементам объектов классового типа. Один класс также может быть объявлен дружественным другому, указывая, что все функциональные компоненты дружественного класса дружественны:

```
class Y;
class X
{   friend Y;   int i;   void f();   };
class Y
{   int f1(X&);   void f2(X&);   // и т.д.   };
```

в примере приватные элементы объектов типа X, такие, как элементы i и j() аргументов X&, доступны внутри Y::fl и Y::f2. Операторы доступа к элементам обязательно должны использоваться дружественными функциями, так как только элементы класса имеют this.

Мы уже упоминали, что в союзах элементы общедоступны, если не указано противное. Структуры - это другой тип классов, которые отличаются от классов предопределенным уровнем доступа к элементам. В определении структуры вместо class используется struct. Первая секция структуры будет общедоступной, пока при помощи метки не будет указан другой уровень защиты:

```
struct String
{   String ();
    String( char * );
    void print ();
    friend int operator < (String si, String s2);   // и т.д.
    private:   char *str;   };
```

в версии String все функциональные компоненты общедоступны, тогда как элемент данных str - приватный.

2.2.6 Инициализация и преобразования

Если класс содержит конструкторы, то конструктор всегда используется для инициализации объектов классового типа при их создании. В приведенном выше примере с сортировкой строк конструктор без аргументов инициализирует массив String. Для классовых объектов - не массивов - аргументы конструктора могут использоваться для инициализации объектов в точке объявления:

```
String s1( "hi" );
String s2="hi";
```

в этих двух объявлениях используются обе формы инициализаторов как аргументы конструктора для инициализации объектов s1 и s2 типа String.

Аргументы конструктора также могут использоваться при создании объектов оператором new:

```
String *sp;
sp = new String( "hello" );
```

здесь sp указывает на String, инициализированную конструктором с аргументом "hello". Если

аргументы не используются при объявлении или создании объекта класса, используется конструктор без аргументов или с предопределенными аргументами.

Классовые объекты, являющиеся компонентами других классовых объектов, также инициализируются конструктором. Аргументы для конструктора могут быть даны в списке инициализации элементов для конструктора объемлющего класса. Начальные значения для любого элемента (а не только аргументы конструктора) могут задаваться в списке инициализации элементов конструктора. Элементы, требующие инициализации, такие как константы или элементы ссылочного типа, должны иметь свои инициализаторы в списке. В определении конструктора список инициализации элементов отделяется от списка аргументов конструктора двоеточием. Список инициализации содержит список имен элементов, каждый из которых сопровождается заключенным в скобки списком аргументов конструктора или начальным значением:

```
enum {A, B, C};
class Node
{   public:     int code;
    String str;
    Node(int,char *);    };
Node::Node(int c, char *s)
{   code( c );    str( s );      }
```

Конструктор Node в этом примере инициирует элемент code значением аргумента c и передает второй аргумент конструктору String для инициализации элемента str из Node. С учетом данного определения Node, фрагмент программы

```
Node *np = new Node(A, "hello");
np->str.print();
```

печатает hello.

Может показаться, что использование списка инициализации элементов можно заменить присваиваниями в теле конструктора:

```
Node::Node(int c, char *s)
{   code = c;    str = s;    }
```

Для целого элемента code нет большой разницы между первым присваиванием в этой новой версии конструктора Node и указанием начального значения в списке инициализации. Элемент str имеет классовый тип String, имеющий конструкторы:

```
class String
{   char *str;
    public:
    String ()
    {         stp = new char;      *str = 0;    }
    String( char *s)
    {         str = new char [strlen(s}+1];    strcpy(str,s):     } // …     }
```

Когда бы не создавался объект String, он всегда инициализируется. Так как не указан никакой инициализатор, для инициализации str при его создании используется предопределенный конструктор String. Присваивание внутри тела конструктора заменяет начальное значение str на результат преобразования в правой части присваивания. Будут произведены два вызова конструкторов String: один для инициализации str, другой - для преобразования s перед присваиванием. Присваивание, таким образом, это не то же самое, что инициализация. Аргументы конструкторов и инициализирующие элементы должны быть указаны в списке инициализации.

Конструктор создает значение классового объекта из его аргументов, и поэтому преобразует аргументы в классовый тип. Конструкторы - не только инициализаторы, но и операторы преобразования. Это показано в программе сортировки строк, в которой указатель на массив символов преобразуется в String с использованием преобразования в

функциональном стиле и присваивается элементу массива String:

a[i] = String (buffer); .

Когда конструктор принимает один аргумент, он также может быть вызван для преобразования с использованием оператора преобразования в форме приведения типов:

a[i] = (String) buffer; .

Преобразования по типу вызова функции и приведением типов вообще-то взаимозаменяемы, но преобразование при помощи конструктора с многими аргументами требует вызова в функциональном стиле, а спецификации преобразования синтаксически сложного типа требуют следующего приведения типов:

```
x = X(i,j);          // преобразует i и j в X
fprint (*) ()) g;          // преобразует g в указатель на функцию.
```

Операнды выражений и аргументы вызова функции автоматически преобразуются в нужный тип, если доступны необходимые предопределенные и/или определенные пользователем операции преобразования. Самое большее одно предопределенное и одно определенное пользователем преобразование производятся автоматически. Двусмысленный выбор возможных преобразований является ошибкой. Для вызова преобразования, которое не может быть произведено автоматически, необходима явная операция преобразования.

Так как преобразования могут происходить автоматически, явная операция преобразования входного буфера char* в String в примере с сортировкой строк не нужна. Наличие конструктора String (char*) разрешает определенное пользователем преобразование из типа в правой части в тип левой части. Присваивание, заботящееся об автоматическом преобразовании, a[i] = buffer; работает точно также, как присваивание с явным преобразованием. В обоих случаях для преобразования правой части используется конструктор String.

Конструкторы производят преобразования в тип класса. Можно производить преобразования из классового типа с помощью операторных функций преобразования. Эти функции должны быть элементами преобразуемого класса. Они имеют имена в форме operator T, где T - имя или спецификация типа, являющегося результатом преобразования. Добавим к классу String оператор обратного преобразования String в char *:

```
ClassString
{   char *str;
    public:    operator char*();   …   };
String::operator char*()
{   char *p = new char[strlen(str)-1];
    strcpy(p,str);
    return p;      }
```

Необходимо учитывать, что operator char* создает копию массива символов, на который указывает str и возвращает указатель на этот новый массив. Простой возврат значения str производит необходимое преобразование, но это дает возможность доступа к структуре данных, защищенных String, оставляя ее открытой для неправильного использования. Создание копии оставляет массив, на который указывает элемент str, доступным только через объект типа String и функции, имеющие доступ к String.

Следующий пример демонстрирует использование оператора преобразования и показывает, каким образом результат char* может быть использован без влияния на исходный String. Мы также ввели в наш пример новый элемент ~String(). Это деструктор для класса String. Он выполняется, как только String покидает область видимости. Деструктор используется для освобождения места, занимаемого массивом символов, созданного конструктором String. Необходимо учитывать, что предопределенный конструктор изменен. Оператор delete освобождает память, захваченную new и может применяться только к объекту, созданному с помощью new. Оператор delete вызывает деструктор для объекта класса перед тем, как освободить память.

```
#include <stdio.h>
#include <string.h>
class String
{   char *str; public:
    String()  {  str = new char; *str=0; }
    String( char * );
    void print(){ printf("%s", str);}
    operator char*();
    ~String() { delete str; }       };
String::String(char*s)
{   str = new char[strlen(s)+1];      strcpy(str,s);         }
String::operator char*()
{   char *p = new char[strlen(str)+1];
    strcpy(p, str);     return p;       }
main()
{   String *sp = new String( "hello world?" );
    sp->print();
    char *cp;    cp = (char *)*sp;     cp[11]='!';
    printf("\n %s \n",cp);
    sp->print();
    delete sp;     cp[0]='H';       cp[6]='W':
    printf("\n %s \n",cp);       }
```

В этой программе явное преобразование String в char* cp-(char *)*sp: не является необходимым, так как преобразование может быть произведено автоматически. Это преобразование включено только для того, чтобы отметить место использования operator char*. Программа выдает:

```
hello world?
hello world!
hello world?
Hello World!
```

2.2.7 Указатели на компоненты класса

Элементы класса, отличные от элементов данных static, являются компонентами класса. Адреса компонентов класса являются "смещениями" относительно конкретного объекта. Относительные адреса компонентов класса имеют тип указателя на элемент класса.

Модификатор типа для индикации указателя на элемент в объявлении записывается как X::*, где X - имя класса. Для разыменования используются операторы .* и ->*. Как и связанные с ними операторы доступа к элементам, операторы разыменования надо использовать с левым операндом классового типа.

В следующих примерах объявляются, создаются и используются указатели на элемент простого класса:

```
class Node
{      public:
     int code;     int num;
     void print();    void report();
};
     int Node::*pi;
     Node  n,*np  =  new  Node;
int i,J;
```

```
pi=&Node::code;
i=n,*pi;      //          достаем n.code
j=np->*pi:  //   достаем   np->code
pi=&Node::num;
i=n,*pi;      //достаем n.num
j=np->*pi;   //   достаем   np->num
```

Сначала указатель pi на элемент данных Node типа int объявляется как указатель на элемент int из Node. Он создан для указания на Node: :code и Node:: num и используется для доступа к этим элементам объектов типа Node.

Спецификация типа указателя на функциональный элемент - это нечто сложное, поэтому мы используем typedef для присвоения типу указателя на функциональный элемент Node имени pftype, не используя аргументов и возвращая void. Затем объявим pf как указатель этого типа:

```
typedef void (Node::*Pftype)();
Pftype pf;
pf=&Node::print;
(n.*pf)();      // вызывает n.print()
(np->*pf)();  // вызывает np->print()
pf=&Node::report;
(n.pf)();  // вызывает n.report()
(np->*pf)();  //  вызывает  np->report()
```

Указатель pf использован для вызова сначала Node::print, а затем Node::report для разных объектов Node. При использовании n.*pf и np->*pf следует указывать скобки для получения нужной компоновки операторов разыменования относительно оператора вызова. Вызов функциональных компонент таким образом - это основное использование указателей на элементы.

Для демонстрации использования указателя на функциональный элемент мы расширим функцию сортировки строк для расположения списка по возрастанию или убыванию в зависимости от значения дополнительного третьего параметра. Используются варианты операторов-элементов для сравнения значений String:

```
class String   {     // скрытая реализация
               public: // и т.д.
               int operator < (String s);
               int operator > (String s);    };
```

Функция sort объявляет переменную compare как указатель на элемент-функцию String, имеющую аргумент типа String и возвращающую int. Поскольку переменная compare представляет собой указатель на тип функций, одну из которых мы хотим выбрать, то он устанавливается на String::operator > или на String::.operator <:

```
void sort(Strlng *a, int n, int descending)
{   int changed;
    typedef int (Strlng::*Ftype) (String);
    Ftype compare - descending ?
    Strlng::operator >;     String::operator<;
    do {     changed=0;
             for (int i=0; i<n-1; i++)
             if ((a[i+1].*compare)(a[i]))
             { // поменять местами элементы массива}
    } while (changed);       }
```

Вызов функции через указатель, (a[i+1].*compare)(a[i]) - это то же, что и инфиксный вызов операторной функции a[i+1]>a[i] или a[i+1]<a[i] в зависимости от значения compare.

2.3 АБСТРАКЦИЯ ДАННЫХ

Абстрактный тип данных - это инкапсулированный тип данных, доступ к которому осуществляется только через интерфейс, скрывающий детали реализации. Свойства абстрактного типа данных определяются его интерфейсом, а не его внутренней структурой или реализацией. Один и тот же абстрактный тип данных, таким образом, можно

реализовывать по-разному, не влияя на фрагменты кода, его использующие. Именно в этом состоит смысл абстрактности типа данных: свойства типа определяются интерфейсом, а от деталей реализации абстрагируются.

Для абстракции данных в C++ используются классы, скрывающие реализацию типа в приватной части своего определения и предлагающие интерфейс из общедоступных операций. В этой главе представлено несколько примеров реализации абстрактных типов данных с помощью классов C++ и обсуждаются вопросы разработки классов для абстракции данных.

2.3.1 Комплексные числа

Класс комплексных чисел является хорошим примером абстрактного типа данных.

```
class complex
{   double re, im;
    public:    friend complex operator + (complex, complex):
friend complex operator / (complex, complex);
complex (double=0.0, double=0.0;     };
```

В данном случае комплексное число представлено его координатами в прямоугольной системе на комплексной плоскости.

В будущем, тем не менее, можно изменить реализацию, чтобы использовать полярную систему координат. Представление, при помощи которого обеспечивается реализация комплексных чисел, скрыто от общего доступа в приватной части определения класса; поэтому изменение реализации не будет влиять на программы пользователя. Общедоступный интерфейс определяет свойства типа complex. Пока этот интерфейс поддерживается, никакие изменения реализации не влияют на программы пользователя:

```
class complex
{   double theta; double r:
    public:
    friend complex operator* (complex a, complex b)
    {         complex result;
              result.r=-a.r+b.r:     result.theta=a.theta+b.theta;
              return result;     }     }
```

Так как абстрактный интерфейс определяет для пользователя семантику типа, основной задачей при создании абстрактного типа данных является разработка интерфейса, которая должна учитывать абстрактные свойства типа, а не просто предохранять пользователя от изменения реализации. В этом разница между защитой и абстракцией данных. Интерфейс, защищающий реализацию типа от несанкционированного доступа, не скрывая при этом его структуры, провоцирует создание программы, использующей его с установлением зависимостей от реализации.

Например, если интерфейс нашего типа complex открывает, что реализация - пара чисел double, то пользователи complex могут написать программу, предполагающую, что комплексное число - пара прямоугольных координат:

```
class complex
{   double first*() { return re; }
    double second(){ return im; }     };
extern complex a;
complex b ( a.first(), a.second() );   .
```

Если complex реализован как пара координат, то a равно b. Если же изменить реализацию, введя полярные координаты, complex все также представляется парой чисел типа double, но a уже не обязательно равно b.

Для комплексных чисел дополнительная семантика типа встраивается в множество арифметических операций и преобразований из других и в другие числовые типы.

Комплексные числа можно складывать, вычитать, перемножать и делить, поэтому эти свойства реализуются перегрузкой операторов +,-,* и / для поддержки операндов и выработки результатов типа complex. Совсем необязательно для реализации этого интерфейса перегружать операторы; "обыкновенные" функции с подходящим разрешением доступа тоже можно использовать для реализации абстрактного интерфейса:

```
class complex
{   public:   friend complex add(complex,complex);
    friend complex mul(complex,complex);
    friend complex div(complex,complex);
    complex(double=0.0.double=0.0);     };  .
```

Применение таких функций для абстрактного типа данных, тем не менее, не приводит к естественному расширению числовых типов и операций, представляемых языком. Сравните:

```
Z = add(add(R,mul(mul(j, omega), L)), div(1,mul(mul(j,omega),C)));
```

с намного более читабельным выражением

```
Z=R+j * omega * L+ 1/(j * omega * C);  .
```

Предопределенные арифметические операторы предлагают интуитивный интерфейс для класса complex, но перегрузка операторов может быть осуществлена неправильно. Так как семантика перегруженного оператора определяется программистом, реализующим класс, он может определить + для вычитания и - для сложения, но при отсутствии злого умысла это маловероятно.

Более частой ошибкой при перегрузке операторов является их неправильное использование. Например, другое свойство комплексных чисел, которое может быть отражено в абстрактном интерфейсе - это экспоненциальность. Так как в С++ нет оператора возведения в степень, для реализации последнего необходимо использовать один из существующих операторов, как, например, ^ (исключающее или), который не имеет смысла для комплексных чисел. Хотя это и возможно, однако такое использование перегрузки операторов вызовет ошибки и неправильное прочтение. Причина - несоответствующий возведению в степень приоритет ^. Пользователь класса complex может записать выражение $1+e^{i\square}$ как l+e^(i*pi), предполагая, что оператор возведения в степень более сильный, чем сложение, как это сделано в большинстве языков со встроенным экспоненциальным оператором. Вспомним, что перегрузка не изменяет существующих приоритетов и ассоциативности операторов, поэтому выражение l+e^(i*pi) интерпретируется как $(1+e)^{i\square}$, так как ^ имеет приоритет ниже +. В этом случае лучше заменить перегрузку оператора более прозаическим, но и более понятным использованием неоператорной функции 1 + pow(e,i*pi).

Перегрузка операторов может использоваться, если существующий приоритет и семантика оператора поддерживает интуитивное понимание его нового использования.

Как уже отмечалось ранее, к приватным частям определения класса имеют доступ два типа функций: функции, являющиеся элементами класса, и функции, специально объявленные как дружественные. Почему же арифметические операции над комплексными числами реализованы как дружественные функции, а не как элементы класса?

```
class complex
{    public: // операторы-элементы
    complex operator +(complex); //бинарный
    complex operator -(complex); // бинарный
    complex operator-(): //унарный          };  .
```

Причина существует в способе взаимодействия комплексных чисел с другими арифметическими типами в смешанных выражениях.

Конструктор для класса complex (который, как и операторные функции, является частью интерфейса) играет двоякую роль. Кроме инициализации каждого объекта типа complex, он также специфицирует преобразование значения типа double в значение типа complex. Существуют предопределенные преобразования других арифметических типов в double;

поэтому конструктор также определяет преобразование в complex других предопределенных арифметических типов. Конструктор вызывается, если необходимо произвести преобразование при присваивании или инициализации объекта complex предопределенным арифметическим типом.

```
complex x = 12.34;        // complex(12.34,0)
x=12;                 // complex((double)12,0)
```

В этом случае конструктор, если необходимо, вызывается для преобразования при инициализации формальных аргументов функции фактическими аргументами вызова.

```
complex add(complex, complex);
complex a,b: double c,d:
add(a,b);
add(a,d);  // add(a,complex(d));
add(c,b);  // add(complex(c),b);  .
```

Операторная функция не сильно отличается от неоператорной и может быть вызвана и как инфиксный оператор, и как неоператорная функция:

```
complex operator +(complex, complex);
a + b;  operator +(a,b);   // то же, что и выше
a+d;    operator+(a,d); c+b ;  operator +(c,b);  .
```

Попробуем написать ту же последовательность выражений с использованием операторной функции-элемента класса:

```
a+b;   a.operator +(b);   //отлично...
a+d;   a.operator +(d);   //отлично...
c+b;  // ошибка!   c.operator+(b);    //ошибка!
```

Затруднения со сложением b и c в приведенном примере состоит в том, что мы пытаемся вызвать операторную функцию элемент c, однако c - не классового типа и не имеет элементов! И в записи c+b, и как c.operator+(b), выражение не имеет смысла. Если мы реализуем комплексные операторные функции как элементы класса, пользователи нашего типа никогда не смогут написать выражение, в котором первым операндом комплексного оператора будет некомплексный элемент, не используя при этом явного преобразования.

Реализация операций с комплексными числами перегрузкой существующих операторов и поддержка инициализации и преобразований с помощью конструктора, позволяет нам расширить систему арифметических типов C++ за счет включения комплексных чисел, которые можно также легко и естественно использовать, как и встроенные арифметические типы.

2.3.2 Строки

В качестве следующего примера абстракции данных рассмотрим модифицированную версию типа данных String.

```
class String_rep
{char *str;    int refs;
Strlng_rep( char *);
friend class String;    };
class String
{ Strlng_rep *r,
public:    friend String operator+(String,String);
           friend int operator (String.String);
String &operator = (String);
operator char *();
String(char *= "");
String(String &);
~String();      };
```

в этой версии класса String мы решили расширить фактические строки символов настолько, насколько это возможно, и создали класс String_rep, объединяющий символьную строку и счетчик обращений. Класс String теперь ссылается на эту структуру данных вместо того, чтобы ссылаться напрямую на строку символов.

Первый конструктор очень прост; он специфицирует способ инициализации String символьной строкой (и, как и конструктор для класса complex, определяет преобразование символьной строки в String).

```
String::String(char *s)      { r = new String_rep(s);      }
```

Класс String_rep сам позаботится о своей инициализации.

```
String_rep::Strlng_rep(char *s)
{ str = new char[strlen(s)+1];      strcpy(str.s);      refs=1;    }
```

Когда один String инициализируется другим, вместо создания второй копии символьной строки, мы ссылаемся на существующее представление и увеличиваем его счетчик обращений. Мы определяем такую семантику конструктором, получающим аргумент String&:

```
String::String(String &init)    {  r=lnit.r;     r=refs++;      }
```

Семантика присваивания аналогична, но необходимо позаботиться о том, чтобы String_rep, на который ссылается левый операнд (цель) присваивания, получил свое новое значение. Счетчик обращений String_rep уменьшается на 1, потому что присвоенный String на него больше не ссылается. Если не осталось ссылок, String_rep удаляется:

```
String & String::operator =(String str)
{   if (!--r-refs)      delete r;
    r=str.r;      r-refs++;
    return *this;     }
```

Для симуляции поведения встроенного оператора присваивания оператор присваивания String не только изменяет объект, которому происходит присваивание, но и возвращает значение объекта. Указатель this используется для доступа к объекту, чтобы вернуть его значение.

Следует иметь в виду существующую разницу в семантике присваивания и инициализации для String и комплексных чисел. В случае нашей реализации комплексных чисел, предопределенная семантика присваивания и инициализации является достаточной, и представление одного комплексного числа покомпонентно копируется в другое. Для правильной реализации Siring, счетчик обращений в String_rep, на который ссылается String, необходимо при присваивании модифицировать. Поэтому для класса String необходимо реализовать operator =.

В общем случае, если необходимо для абстрактного типа данных реализовать присваивание, неплохо реализовать также инициализацию и наоборот. В типе String предоставление одного без другого влечет такие же трудности как и отсутствие того и другого. Вместе эти операции определяют, как объекты данного классового типа должны копироваться во всех ситуациях.

Необходимо также позаботиться об установке счетчиков обращений для String-оф, которые удаляются или выходят из области видимости. Для этого определим деструктор:

```
String::~String()
{ if (!--r-refs)  delete r;       }
```

Как и в случае комплексных чисел, организуем перегружаемые операторы и конструктор так, чтобы они хорошо сочетались с существующей в C++ системой типов. Один из конструкторов реализует преобразование символьных строк в String, а операция конкатенации реализуется переопределением оператора + как дружественного для реализации комплексных арифметических операций:

```
extern char *home_dir,*path,*file;
String home=home_dir;
String fpath = home +"/"+ path + "/" + file;  .
```

Для завершения связи типа String с существующей системой типов реализуем оператор, определяющий преобразования из String в char*. В этой версии оператора преобразования, в отличие от предложенной в предыдущей главе, выбран более эффективный, но потенциально более опасный подход с возвращением адреса строки символов вместо адреса копии строки.

```
String::operator char*()
{return r-str;     }
char *newfile = fpath;       // fpath.operator char *()
```

преобразование по сути реализует функцию, обратную конструктору. Конструкторы класса String определяют как создавать форму Siring из **char*** или из другого String, тогда как оператор преобразования operator char* определяет как создать char* из String. Эта возможность важна для полноты связи с существующей системой типов C++; она предоставляется для связи с существующими программами для объектов типа char*:

```
extern FILE *fopen(const char *,const char*);
FILE *fp=fopen(fpath,"r+");  .
```

Как и конструктор, оператор преобразования при необходимости вызывается в выражениях и инициализаторах.

В отличие от перегружаемого оператора + для конкатенации и в отличие от операторов комплексной арифметики, присваивание для String реализовано как функциональная компонента класса. Причина в том, что объекты, порождаемые элементами-функциями с учетом автоматического преобразования аргументов, есть в точности то, что нужно для присваивания; т.е. не нужно применять преобразования к левому аргументу присваивания.

Если operator= реализован как дружественная функция, будет возможным присваивать String-u символьным строкам, так как char* в левой части оператора присваивания будет автоматически преобразован в String конструктором String, принимающим аргумент char *: "/usr/bin" = pathname Вызов инфиксного оператора, который здесь использован - это только принятая запись, и он полностью эквивалентен вызову функции operator -("/usr/bin", pathname); поэтому, даже если мы, возможно, не хотим позволить присваивания символьной строке типа "/usr/bin", это тем не менее корректно, если operator= не является элементом класса String. Если сделать = функциональным элементом, присваивание, приведенное выше, будет иметь тот же смысл, что и "/usr/bin".operator-(pathname), т.е. никакого. Реализация присваивания String как функциональной компоненты предотвращает применение таких преобразований к левой части оператора присваивания. Более разумная семантика присваивания или инициализации char* элементами элементов типа String уже производится оператором преобразования.

Правильное использование операторных функций, конструкторов и операторов преобразования позволяет разрабатывать абстрактные типы данных, соответствующие системе предопределенных типов C++ и расширяющие ее.

2.3.3 Упорядоченные последовательности

Абстракция данных дает две важных возможности тем, кто ее использует. Во-первых, она упрощает семантику использования типа, ограничивая операции теми, которые представлены в общедоступном интерфейсе. Пользователи типа не должны бороться с семантическими особенностями, зависящими от реализации и не имеющими значения для абстрактной функции типа. Например, пользователи типа String не должны заботиться об обновлении счетчиков обращений при присваивании или о переполнении буферов при конкатенациях. "Вторичная" семантика - забота скрытой реализации, и ответственность за это возлагается на реализатора типа. По этой же причине реализатор волен изменить реализацию, не боясь повлиять на программы пользователя до тех пор, пока сохраняется

семантика. Также важна предоставляемая абстракцией данных возможность сделать язык программирования ближе к конкретной проблемной области. Возможность определения типов данных complex и String дает использующему эти типы возможность писать понятные, компактные программы, базирующиеся на комплексной арифметике и манипулировании строками, в результате чего язык C++ расширяется, сочетая возможности думать и программировать в этих проблемных областях. Использование абстракции данных для поддержки концептуальной абстракции для разработки программ наиболее важно.

Комплексные числа и строки - типы данных с широкой областью использования и на самом деле являются встроенными типами во многих языках программирования. Хорошо разработанные абстрактные типы данных, тем не менее, могут дать аналогичные возможности и в более специализированных прикладных областях. Например, для некоторых приложений может потребоваться тип с семантикой упорядоченной последовательности целых. Используя ту же технику, что и для complex и String, определим общедоступный интерфейс для такого типа.

```
typedef int ETYPE;
class sorted_collection
{   public:   sorted_collection();
    void insert(ETYPE):
    void apply(void (*)ETYPE));     };
```

класс представляет простую концепцию и поэтому имеет простой интерфейс: есть операции создания sorted_collection, вставки нового целого в последовательность, и применения функции к каждому элементу последовательности в определенном порядке.

После разработки интерфейса пользователи типа могут начать разработку и программирование прикладных программ:

```
#include "collection.h"
printint()   // считывает, сортирует и печатает целые числа
{   extern void print(int);    extern int read(int &);
    sorted_collection sc;
    int i;
    while (read(i)) sc.insert(i);    sc.apply(prlnt);    }
```

Пока пользователи работают с интерфейсом sorted_collection, можно откомпилировать его первую реализацию:

```
typedef int ETYPE;
class sorted_collection
{   ETYPE ary[100];    int free;
    public:   sorted_collection ()   { free = 0; }
    void insert (ETYPE);
    void apply (void (ETYPE));      };
voidsorted_collection::apply (void (*f) (ETYPE) )
{   for (int i=0; i!=free; i++)   f (ary[i]);    }
void sorted_collection::lnsert(ETYPE el)
{   for ( int i =free++;  i && ary[i-1] el;  i--)   ary[i]=ary[i-1];  ary[i]=el;   }
```

Реализация, понятно, не слишком хороша. Алгоритм вставки неэффективен для больших последовательностей. Эта неэффективность, правда, вряд ли вызовет проблемы, ибо, как ранее упоминалось, большая последовательность целых перескочит массив для последовательности фиксированного размера и разрушит программу. Единственный позитивный момент данной реализации - ее можно написать за несколько минут. Этого достаточно, так как теперь пользователи типа могут компилировать и начинать отлаживать свои прикладные программы в то время, пока мы работаем над получением лучшей реализации. Быстрые реализации, похожие на эту, также полезны на начальных этапах разработки программного проекта, когда общедоступный интерфейс абстрактного типа

данных еще не зафиксирован. Таким образом, пользователи могут экспериментировать с типом и модифицировать его интерфейс, не опасаясь свести на нет дорогостоящую программу. Изменим теперь реализацию:

```
typedef  int  ETYPE;
class tree
{   ETYPE el;
    tree *lchild,  *rchild;
    tree (ETYPE i)  { el = i;  lchild = rchild = 0; }
    void  insert (ETYPE);
    void  apply (void (RETYPE));
    friend sorted_collection;   };
void tree:: insert (ETYPE i)
{   if (i)   if (lchild)          lchild=insert(i);
              else      lchild = new tree(i);
    else   if (rchild)          rchild =insert(i); else
    rchild = new tree(i);    };
void  tree:: apply (void (*f) ETYPE))
{   if (lchild)        lchild=apply(f);
    f(el);
    if (rchild)        rchild = apply(f);    }
class sorted_collection
{   tree *root;
    public:
    sorted_collection()  { root=0.0;}
    void  insert (ETYPE el)
    {         if (root)          root=insert(el);
              else      root = new tree(el);    }
    void apply(void (*f)(ETYPE))
    { if (root) root=apply(f); }    };
```

Вторая реализация лучше первой, так как может оперировать большими последовательностями и вставка элементов реализована гораздо более эффективно. Каждая реализация - представление одного и того же абстрактного типа, причем абстрактная семантика не меняется от одной реализации к другой.

Это не значит, что "вторичная" или зависящая от реализации семантика не влияет значительно на поведение программ, использующих тип. Начальная реализация sorted_collection использует неявное допущение о том, что ни одна программа пользователя не попытается вставить в данную последовательность более 100 элементов. Так как это ограничение не представлено в абстрактном интерфейсе, пользователи типа не знают об этом и могут разрушить защиту.

Более хитрыми являются ситуации, в которых программы обходят зависящую от реализации семантику, не являющуюся явной частью абстрактной семантики типа. Например, можно создать тип данных последовательность (чьи элементы не обязаны быть упорядоченными) из упорядоченной последовательности: typedef sorted_collection collection;

К несчастью, пользователь типа может написать программу, учитывающую факт, что последовательность оказалась упорядоченной. Позже можно реализовать collection для большей эффективности вставок по сравнению с sorted_collection по-другому, без упорядочения, но при этом эффективно повреждая программы, зависящие от конкретной семантики реализации. Поэтому часто полезно рассматривать общедоступный интерфейс абстрактного типа данных как договоренность между реализатором и пользователем типа.

От реализатора требуется предоставление корректной абстрактной семантики, специфицированной интерфейсом, не делая никаких дополнительных предположений, а пользователи типа должны учитывать только явно представленную в общем интерфейсе семантику.

2.3.4 Общность

Упорядоченная последовательность целых - полезный для конкретных приложений тип данных, но таковыми являются и упорядоченные последовательности double, String и собственно упорядоченные последовательности какого-то типа. Хотелось бы параметризировать реализацию sorted_collection и затем конкретизировать или создавать ее варианты для элементов конкретных типов. Таким образом, единственная параметризованная реализация sorted_collection может служить общим представлением семантики упорядоченной последовательности. Каждая версия произведет новый тип упорядоченной последовательности для конкретного типа элемента. Пользователи затем смогут объявлять и использовать объекты этих конкретизированных типов.

Язык C++ поддерживает концепцию родовых или параметризованных типов. Тем не менее, можно получить большинство потенциальных возможностей параметризуемых типов с использованием других черт языка. Рассмотрим вторую реализацию sorted_collection. Мы фактически реализовали тип для работы с ETYPE-элементами, где ETYPE определен при помощи typedef как int. Какие особенности ETYPE использует наша реализация? Просматривая программу, находим, что для ETYPE должны быть определены операторы = и < и должна быть возможность инициализации одного ETYPE другим (чтобы инициализировать формальный аргумент ETYPE указателя на функцию в sorted_collection::apply).

Просто изменив typedef для определения некоторого другого типа ETYPE с этими свойствами, получим версию sorted_collection для данного ETYPE:

```
typedef  String ETYPE;
class sorted_collection    {       //...     };
printString()    //  сортирует и печатает вводимые предложения, удаляя нежелательный
                     текст
{   extern void print(char *);
    extern void censor(char *);
    extern char *readstr();
    sorted_collection list;
    char *s;
    while (s = readstr())    list.insert(s);
    list.apply(censor);         list.apply(print);      }
```

Обратите внимание, как работают описанные ранее преобразования из String в char* и обратно.

Эта схема работает, если необходима упорядоченная последовательность не более, чем одного типа, но ее нельзя использовать для нескольких реализаций. ETYPE не может представлять два типа одновременно! Единственное, что может помочь, это копирование реализации sorted_collection и редактирование каждой копии для получения разных версий для элементов различных типов. Обычно для редактирования используется препроцессор. Один из стандартных заголовочных файлов C++ generic.h, содержит препроцессор макроопределений для проведения таких операций редактирования. Макросы предназначены для связывания имен и для объявления и определения родовых типов. Проблема при определении различных примеров родового типа по единому образцу состоит в создании уникальных имен для каждого конкретного типа. Для решения данной задачи будем использовать макрос name2 из generic.h.

```
#define     sorted_collection(ETYPE) \
```

```
                    name2(ETYPE.sorted_collection)
    #define tree(ETYPE) name2(ETYPE.tree)
```

Здесь созданы уникальные имена для наших классовых типов при помощи связывания параметра типа с именами родовых типов. Например, текст tree (String) будет с помощью макроса tree преобразован в Stringtree. Реализация макроса name2 проста, но может отличаться у разных препроцессоров. Для препроцессора ANSI C, name2 может быть реализовано оператором конкатенации: #define name2(a,b) a##b

Для конкретизации новой версии родового типа generic.h предлагает макрос declare: #define declare(a,t) name2(a,declare) (t)

Пользователь, создавая тип "упорядоченная последовательность String-оф", может использовать его следующим образом: declare(sorted_collection, String); После расширения получим: sorted_collectiondeclare(String); здесь sorted_collectiondeclarc - еще один макрос. Как поставщики родового типа sorted_collection, мы должны определить этот макрос:

```
#define sorted_collectiondeclare(ETYPE)
class tree(ETYPE)
{   ETYPEel;
    tree(ETYPE) *l,*r;
    tree(ETYPE)(ETYPE i) { el=i;  l=r=0; }
    void insert (ETYPE);
    void apply(void (RETYPE));
    friend sorted_collection(ETYPE); \    };
class sorted_collection(ETYPE)\
{   tree(ETYPE) *root,
    public:
    sorted_collection(ETYPE)() { root=0; }
    void  insert (ETYPE el)    {    if (root)  root=insert(el); }
    void  apply (void (*f) (ETYPE) )    {root=apply(f); }    };
```

Когда макрос sorted_collectiondeclare расширяется своим аргументом-типом, аргумент замещает ETYPE и макросы, определенные нами ранее, на sorted_collection и tree; по необходимости вызываются для создания уникальных имен классов для родовых sorted_collection и tree. Пользователи теперь могут порождать версии родового класса упорядоченной последовательности и объявлять объекты конкретизированных типов:

```
#include generic.h
#include "collection.h"
#include "string.h"
declare(sorted_collection,int);
declare(sorted_collection.String);
printint()
{   sorted_collection(int) sc;
    int i;
    while (read(i))  sc.insert(i);
    sc.apply(prit);     }
printString()
{   sorted_collection(String) list;
    char *s;
    while (s =readstr())  list.inset(s);
    list.apply(censor);
    list. apply(print);     }
```

В добавление к определениям класса для каждой конкретизированной версии sorted_collection, сгенерированной однажды для каждого файла, в котором они используются, соответствующие определения функциональных компонент tree: :insert и tree:

:app1y должны быть сгенерированы ровно один раз в программе для каждого конкретизированного типа. Для этого можно поступить также, как и с определениями классов, однако нужно убедиться, что эти функции определяются в программе только однажды для каждого типа параметра. Чтобы упростить определение таких параллельных операций, generic.h предоставляет макрос implement, функция которого идентична declare.

Поступая аналогично, можно определить и использовать родовые типы, требующие больше одного параметра-типа для конкретизации.

2.3.5 Абстракция управления

Функция apply класса sorted_collection фактически является композицией двух различных концепций: отслеживание любой структуры данных, использованной для хранения элементов sorted_colleclion, и применение функции к элементу последовательности. Обе эти концепции, как и их композиция (как в функции apply), принадлежат к общему классу абстракций, которые называются абстракциями управления. Отслеживание скрытого представления структуры данных без всяких сопровождающих операций используется, конечно, нечасто. Если отслеживание можно разделить на стадии, например, каждая из которых дает некоторое интересное значение, тогда пользователи типа могут получать значения, связанные некоторым образом с типом так, что предохраняется приватность скрытой реализации. Управляющая абстракция такого типа называется итератор.

Обратимся к реализации типа список. У списка есть голова, хвост и последовательность элементов между ними. Мы хотим разработать механизм доступа к значениям элементов списка по порядку:

```
struct node
{   node *next;     ETYPE el;
    node (ETYPE i, node *n)  { el = i; next=n; }      };
class list
{   node *hd;
    public:     list (node *n=0)  { hd=n;}
    list(list &seq)      { hd=seq.hd;}
    void lnsert (ETYPE i)  { hd = new node(i.hd); }
    friend ETYPE head (list seq) { return seq.hd-el;}
    friend list tall(list seq) { return list(seq.hd-next);}
    friend int isempty(llst seq) { return seq.hd ===0; }     };
void print_llst(list seq)
{   extern void print(ETYPE);
    for (list s=seq; lisempty(s); s=tail(s))  print(head(s));     }
```

Это полезная абстракция, если вы привыкли рассматривать списки рекурсивно, то есть список - пустой, либо же это элемент, за которым следует список. В нашем случае мы хотим рассматривать списки итеративно, то есть как последовательность элементов списка, а не как рекурсивно определенную последовательность списков. Хотя приведенная выше абстракция эффективна, она позволяет получить доступ к элементам только последовательно, не поддерживая наш способ рассмотрения списков, поэтому пробуем другую версию:

```
class list
{   ETYPE el; list *link;
    public:    list (ETYPE  i,  list *next) { el=i; link=next;}
    list *next()  { return link; }
    ETYPE value()  { return el; }     };
void print_list(list *seq)
{   extern void print(ETYPE);
```

```
        for (list *p=seq;  p; p=p-next() )  print(p-value() );    }
```
Эта вторая реализация, конечно, итеративна по своей природе, но мы как-то потеряли целостность нашей абстракции, а принимаем во внимание только элементы списка. Вот другая версия:

```
    class node
    {   node *next; ETYPE el;
        node(ETYPE, node *);
        friend list;     friend Iter;      };
    class list
    {   node *hd;
        public:     list()
        void insert (ETYPE);
        friend iter;     };
    class iter
    {   node *currenf,
        public:     iter(list);
        ETYPE *operator()();     };
    void prlnt_list (list lst)
    {   extern void print(ETYPE);
        iter next=lst;
        ETYPE *p;
        while (p=next())     print(*p);    }
```
Здесь создан тип итератор, связанный с типом список, точно также включающий в себя абстракцию управления (последовательный просмотр элементов списка), как и тип список включает абстракцию структуры данных списка. Для перебора элементов списка мы создали итерационный объект и встроили его в списочный объект в инициализации:

```
    iter::lter (list Hist) { current=ilist.hd;}
```
Итерационный объект сам по себе сохраняет состояние итерации от обращения к обращению:

```
    ETYPE * iter::operator()()
    {   if (current)
            {   node *tmp = current;   current = current-next;  return &tmp-el;  }
        return 0;    }
```
Этот итератор имеет единственную операцию: он получает следующий элемент списка. В связи с чем мы решили использовать перегружаемый оператор () для выражения этой операции, но мы также могли перегрузить ++ или использовать неоператорную элемент-функцию, или даже дружественную функцию, не являющуюся элементом:

```
    class iter
    {   public: //...
        ETYPE operator ()();  //next()
        ETYPE *operator ++();          // next++  или  ++next
        ETYPE *next ();                // next.next()
        friend ETYPE *nxt(iter);       // nxt(next)    };  .
```
Основная идея состоит в том, что итераторный объект имеет концепцию сохранения от обращения к обращению. Такая точка зрения на список - как раз то, что нам нужно. Мы сохранили концепцию списка как "вещи в себе", но имеем возможность естественным образом проследовать по элементам списка. Итератор в результате - расширение класса список, предоставляющее абстрактную операцию поточного управления. Это создание абстрактных типов поточного управления, использующего вместе с абстрактными типами данных мощный механизм для работы с абстрактными типами со сложной внутренней структурой в реализационно-независимом стиле.

Итератор для нашего класса список очень простой; с каждым вызовом он дает следующий элемент списка до тех пор, пока элементов не останется. Можно, конечно же, реализовать более сложную семантику. Например, создание итератора для списка может иметь побочный эффект блокировки списка, так что пока итерация не завершилась, новые элементы не могут быть вставлены. Мы могли определить более общее продвижение по списку. Например, добавить в класс iter сканирующую функцию, возвращающую следующий элемент списка, удовлетворяющий как аргумент предикатной функции,

```
class employee
{   public:
    int ismgr();
    int istrue() { return 1; }     };
typedef employee ETYPE;
class list   {        //...   };
class iter
{   public:
    ETYPE *operator ()();
    ETYPE *operator () int (ETYPE::*) ();     };
void clean_up (list alist, int all)
{   extern void print(employee);
    extem void fire(employee);
    iter next = alist;     employee *e;
    while (e=next())   print(*e);
    int (employee::*predicate)() = all ? employee::lstrue : employee::ismgr;
    iter next_vlctim = alist;
    while (e= next_victim(predicate))  fire(*e);    }
```

или мы могли добавить возможность возврата к предыдущему элементу, или к голове списка. Необходимо помнить, что итераторный объект может быть присвоен другому итераторному объекту или передаваться функции как аргумент, и что для одного списка может быть несколько одновременно активных итераторных объектов. Для одного и того же абстрактного типа, (как и нескольких объектов одного итераторного типа), можно создать несколько типов итераторов. Тип данных дерево может определять различные итераторы для обхода слева, справа, сначала вширь и сначала вглубь.

Некоторые приемы так очевидны, что их можно не заметить, как, например, возможность поддержки элементов класса только для чтения. В некоторых ситуациях реализатор может хотеть представить окно в приватной части реализации типа, или реализатор может хотеть, чтобы пользователи типа проэкзаменовали часть общедоступного интерфейса, не изменяя ее. Эта возможность может быть реализована подставляемыми (inline) функциями:

```
class node
{   node *n;     ETYPE val;
    public:    node *next() { return n; }
    ETYPE value()              { return val; }      };
for (node *p=head; p; p=p-next() ) print (p-value());
```

В данном случае реализация - это абстракция, но пользователь типа Node не может изменять структуру списка или значения элементов.

Другой простой прием управления представляется аппликаторами. Аппликатор - это функция, применяющая один из своих аргументов к другим: void apply (char *s, void(*f)(char *)) { f(s); } Это простой аппликатор, применяющий свой аргумент функцию (указатель) к своему аргументу - символьной строке:

```
String pathname-"/usr/cmd";
extern void exec(char *);   extern char *cmd;
```

```
apply(pathname+"/"+cmd, exec);
```

Конечно, мы могли реализовать это более эффективно, вызывая функцию напрямую, зачем же нужны аппликаторы.

Мы уже видим, как используются аппликаторы в классе sorted_collection, хотя и не в явной форме. Функциональный элемент apply-это композиция отслеживания и аппликатора, позволяющая пользователям sorted_collection применять функцию к каждому элементу последовательности.

2.4 НАСЛЕДОВАНИЕ

Абстракция данных - эффективный метод расширения предопределенной системы типов, если можно определить одну, ясно определенную концепцию, как это было сделано в предыдущей главе. Если же необходимо иметь абстрактный тип данных, который похож на уже имеющийся, но не совсем такой, то можно использовать наследование.

Наследование - это механизм для построения класса из других классовых типов путем добавления новых свойств к уже имеющимся классам.

2.4.1 Базовые и производные классы

Класс может наследовать черты другого класса как производный от него. Класс, полученный из исходного или базового типа, может добавить или приспособить черты своего родителя для получения специализации или расширения базового типа, или просто повторного использования реализации базового класса.

Например, можно рассматривать двусвязный список как односвязный список с добавочными указателем и операцией доступа к предыдущему элементу так же, как и к следующему. Рассмотрим класс список из предыдущей главы:

```
class list
{   ETYPE el;
    list *link;
    public:     list (ETYPE, ist *);
    list *next();
    ETYPE value();      };
```

Используем этот класс для определения двусвязного списка

```
class list2 : public list
{   list2 *link;
    public:     list2(ETYPE,list2 *,list2 *);
    list2 *previous();     };
```

Запись class list2 : public list определяет list2 как новый класс, выведенный из класса list. Мы говорим также, что класс list является базовым для класса list2. list2 наследует все элементы list (данные и функции) и добавляет свои: связь с предыдущим элементом списка, функцию для доступа к этой связи и конструктор.

Реализация функционального элемента previous аналогична функции next класса list:

```
list2 * list2::         previous () { return link;}
```

Класс list2 содержит два различных элемента с именем link: исходная связь вперед, унаследованная от базового класса, и обратная связь, объявленная в теле list2. Значение link, возвращаемое previous, - это link из list2, вследствие влияния наследования на видимость классов.

Производные классы образуют иерархию областей видимости. Область видимости производного класса содержится внутри области видимости его базового блока, во многом аналогично внутреннему блоку функции внутри охватывающего блока. Поэтому когда мы

ссылаемся на link в элементе-функции list2, компилятор сначала будет искать link, принадлежавший классу list2, и проверит область видимости базового класса только тогда, когда это имя не найдено в производном классе. В случае, если необходимо получить доступ к элементу базового класса, скрытого элементом производного класса, запись base::member позволяет специфицировать просмотр в поисках этого имени, начиная с области видимости base.

Функция whatsbefore получает доступ к наследуемым и обычным элементам совершенно аналогично:

```
list2 * whatsbefore (list2 *lst,  ETYPE e)
{   for (list2 *p = lst; p; p = (list2 *)p->next() )
            if (p->value()== e) break;
if (p)      return p->previous0;
else        return 0;    }
```

Вложенность областей видимости объясняет причину, почему работает этот код. Только вызов previous дает доступ к элементу, явно объявленному в list2. Вызовы next и value ссылаются на функциональные элементы расширенного базового класса.

Напротив, можно снабдить list2 полным набором функциональных элементов для передвижения по списку и доступа к значению элементов:

```
class list2 : public list
{   list2 *link;
    public:     list *next();
    list2 *previous();
    ETYPE value();     };
```

Здесь возникают некоторые проблемы. Рассмотрим реализацию list2::next.

```
list*  list2::next()     { return list::link; // ошибка!    }
```

Попытка получить значения обратной связи из базового класса - это ошибка, так как list::link - приватный. Если производный класс не объявлен как дружественный, он не имеет никаких специальных привилегий доступа к приватным элементам своего базового класса. Единственная возможность получить это значение - использовать общедоступный интерфейс list в виде   return list::next();   который будет работать, но вряд ли представляется ценной функция, которая ничего не делает. Напоминаем, что определение класса list2 начинается с class list2 : public list.

Ключевое слово public в этом контексте определяет, что общедоступные элементы list будут общедоступными и для пользователей list2. Если использовано ключевое слово private, или если не использованы ни private, public, то общедоступные элементы list будут приветными при доступе через list2. Например, в функции whatsbefore, все ссылки на value и next будут ошибочными ссылками на приватные элементы. Помните, что элементы и дружественные функции list2 имеют доступ к общедоступным элементам list, несмотря на то, является ли он общедоступным или приватным базовым классом.

Хорошее разделение доступа к общедоступным элементам данных класса может быть достигнуто объявлениями общедоступных базовых элементов в производном классе. Объявление общедоступного базового элемента имеет вид Base::member; где Base - имя базового класса, а member - общедоступный элемент этого базового класса. Объявление должно находиться в общедоступной части определения производного класса:

```
    class otherlist
{   public:   otherlist *forw;    ETYPE el;     };
    class otherlist2 : private otherlist
    {   public:     otherlist :: forw;     otherlist2 *back:     };
```

Хотя otherlist - приватный базовый класс otherlist2, объявление в базовом классе общедоступного forw позволяет получать доступ к нему как к общедоступному методу через otherlist2. Так как otherlist: :e1 не объявлен как общедоступный базовый элемент, при доступе через otherlist2 он будет приватным.

Конструкторы производных классов могут включать явную инициализацию базового класса в списке инициализации элементов. Конструктор для list2 использует список инициализации элементов для инициализации своего базового класса. как будто они инициализирует элемент. Инициализация базового класса полностью аналогична инициализации элемента класса, за исключением того, что базовый класс всегда (явно или неявно) инициализируется раньше любого элемента, даже если инициализатор элемента появляется до инициализатора базового класса в списке инициализации. Если у базового класса нет конструктора, его не нужно инициализировать, а если его конструктор можно вызвать без аргументов, его не нужно инициализировать явно:

```
list2 :: list2 (ETYPE e,  list2 *fl,  list2 *bl)  : list (e, fl)  { link =bl; }
```

Так же, как и для инициализации элементов, инициализация базового класса может быть реализована любым возможным инициализатором объекта базового классового типа, а не только инициализатором конструктора. Например, можно определить второй конструктор list2, инициализирующий его базовый класс копированием значения существующего объекта list:

```
extern list &exlist;
    list2 :: list2 (list2 * fl) : list(exlist)  { link = bl; }
```

Аргументы, передаваемые конструктору базового класса в списке инициализации элементов исходного конструктора list2, имеют типы ETYPE и list2* соответственно, тогда как типы, требуемые конструктором list - это ETYPE и list*. Почему не является ошибкой инициализация list* как list2*? Причина в том, что объект класса list2 является одновременно объектом класса list. Это было определено при выводе одного класса из другого. Кроме того, если класс Base - общедоступный базовый класс другого класса Derived, то существует предопределенное преобразование из Derived в Base, из указателя на Derived в указатель на Base, и из ссылки на Derived в ссылку на Base. Эти преобразования не существуют, если Base - приватный базовый класс для Derived.

Концепция "то же, что и" - мощный механизм абстракции, так как она позволяет во многих контекстах пользоваться производным классом как базовым. Например, так как list - общедоступный базовый класс list2, можно применить функцию print_list из предыдущей главы как к списку из компонентов типа list, так и к списку из компонентов типа list2:

```
typedef int ETYPE;
void print_list(list *lst)
{       extern void print(int);
        for (list *p=ist; p; p=p->next() )   print(p->value());    }
main()
{       list *lp = new list2(1,0);
        lp = new list(2,lp);          lp = new list(3,lp);
        print_list(lp);
    list2*l2p = new list2(1,0,0);
    l2p = new list2(2,l2p,0);        l2p = new list2(3,12p,0);
    print_lst(l2p);  }
```

В некоторых случаях производный класс может что-то добавлять или модифицировать поведение своего базового класса. Например, в предыдущей главе использован тип данных String для создания полного имени файла и его открытия. Можно инкапсулировать это поведение в класс, производный по отношению к String. Абстрактные операции, которые требуется выполнять над именами - это создание, конкатенация и сравнение строк, представляющих имена, а также операция открытия файлов, на которые они ссылаются. Многие из этих операций предлагаются типом String, и имя файла может быть рассмотрено и как расширение, и как особый случай String. Когда говорят "имя файла - это String со следующими дополнительными свойствами"., подразумевают расширение String. Когда говорят: "Имя файла - это String, который...", рассматривают специальный случай String. Обе

эти концепции выражаются наследованием классов:

```
class Pathname : public String
{   public:
    friend Pathname &operator+(Pathname &,Pathname &);
    FILE *open();     };
```

В рассматриваемом случае добавляются не элементы данных к расширению Pathname класса String, а добавляется поведение. В этом смысле Pathname - просто String, рассмотренный с другой точки зрения.

Абстрактные операции создания, уничтожения, сравнения и так далее наследуются от String без изменений. Для конкатенации же необходимо разделять две последовательности имен директорий при помощи слэша "\":

```
Pahname & operator +(Pathname &p1. Pathname &p2)
{    return (String &)p1+"\"+ (String &)p2; }
```

Pathname::open пытается открыть файл, на который ссылается представление полного имени при помощи String..

```
FILE* Pathname::open() { return fopen(*this,"r+");}
```

Текущий аргумент типа Pathname для fopen преобразуется в char* при помощи operator char*, унаследованного из класса String:

```
extern char  *home_dir,  *path,  *fiie;
Pathname home = homedir;
Pathname file = home + path + file;
FILE *fp=file->open();
```

Рассмотренный пример представляет одну из центральных идей при использовании производных классов: унаследовать основное поведение класса от базового и добавить к поведению базового класса что-то необходимое. Наследование, используемое таким образом, представляет эффективный прием для распространения и повторного использования кода.

2.4.2 Иерархии классов

Многие задачи не так легко моделировать единственным типом, их более естественно представить как совокупность связанных типов. Например, абстрактное синтаксическое дерево для компилятора может представлять несколько конструкций языка программирования. Каждый из этих типов вершин имеет "ядро" общих с другими типами свойств (те свойства, которые делают их элементами дерева), но каждый тип вершины имеет дополнительные свойства, отличающие его от других. Использование только абстракции данных для представления всех таких типов дает возможность создать либо единый тип, объединяющий свойства всех остальных, либо множество различных типов, не отражающее их общность. Наилучшим подходом будет использование производных классов для созданий множества типов, связанных наследованием.

Проблема решается при помощи создания иерархии типов. Базовые классы в иерархии представляют общую структуру и функционирование, а производные классы, наоборот, предлагают специализированные версии своих базовых. Пусть требуется создать абстрактное синтаксическое дерево для программы калькулятора. Каждая внутренняя вершина дерева представляет операцию, которую необходимо выполнить, а каждый лист представляет значение. Например, выражение -5+12*4 может быть представлено как при условии, что операторы +, * и унарный - имеют в языке нашего калькулятора такой же приоритет, как в C++.

```
            * □12
           □
        + □ - □ 5
```

Начнем с определения общего типа вершины, который служит базовым типом для других типов вершин:

```
class Node
{   public:    enum {         PLUS, TIMES, UMINUS, INT};
    const int code; int eval();
    Node(int c) : code (c) { }    };
```

Node содержит код, определяющий текущий тип вершины: +, *, унарный - или целое, конструктор и функциональный элемент для вычисления абстрактного синтаксического дерева.

Будем использовать производные классы для создания отдельных типов вершин из Node для операторов *, + и унарного - и для целых значений. Конструкторы производных классов явно вызывают конструктор базового класса и предоставляют соответствующий код вершины:

```
class Plus : public Node
{   public:    Node *left, *righf;
Plus (Node *l, Node *r) : Node(PLUS)     {        left =l; right = r; }     };
class Times : public Node
{   public:    Node *left, *rlght;
Tlmes (Node *l, Node *r) : Node(TIMES) { left = l; right = r; }          };
class int : public Node
{   public:    int value;
int(int v) : Node (INT)  { value = v; }    };
```

Решение не слишком плохое, хотя для двух бинарных операций многое дублируется. Если к языку нашего калькулятора добавить другие бинарные операции, это дублирование становится неуклюжим и может превратиться в потенциальный источник ошибок. Лучше ввести еще один уровень наследования, отражающий общность бинарных операций:

```
class Binop : public Node
{   public:     Node *left, *rlght;
    Binop (int c, Node *l, Node *r) : Node(c)     { left = l; right =r; }    };
class Times : public Binop
{   public:    Times (Node *l, Node *r) : Binop (TIMES,l,r)  {}   };
```

Результирующая иерархия классов показывает, как типы вершин связаны наследованием:

<table>
<tr><td colspan="4" align="center">Node</td></tr>
<tr><td colspan="2" align="center">Binop</td><td align="center">Uminus</td><td align="center">Int</td></tr>
<tr><td align="center">Plus</td><td align="center">Times</td><td></td><td></td></tr>
</table>

Таким образом, Plus - это Binop и Node, но не Uminus. Int - это Node, но не Binop и так далее. Если при создании иерархии быть внимательным, эти наследственные связи между типами будут отражать наше интуитивное представление о концепциях, которые они определяют в проблемной области.

```
int Node::eval()
{    swltch(code)
    { case INT:      return((lnt *)thls)->value;
      case UMINUS: return -((Uminus *)this)->operand->eval0;
      case PLUS:  return((Blnop *)thls)->left->eval0
                    +((Binop*)this)->right->eval();
```

```
    case TIMES: return ((Binop*)this)->left->eval()
                 *((Binop*)this)->right->eval();
    default: error();  return 0;    }     }
```

Здесь реализован Node::eval выбором Node::code и рекурсивным вычислением поддеревьев. Чтобы вызвать правильную элемент-функцию eval нужно привести указатель this к соответствующему типу производного объекта, как это сделано в коде для Node. Так как любой объект класса, производного от Node, тоже имеет тип Node, то указатель на Node - это и указатель на объект производного класса, определяемый как Node::code. Доступ к элементам производного класса можно получить используя приведение указателя this к типу соответствующего указателя на производный тип.

Несмотря на то, что приведенная конструкция работает и может строить и вычислять абстрактные синтаксические деревья, в реализации есть существенный дефект. Здесь реализована иерархия типов вершин, но при использовании явного кода вершин class Node в основном определяет, какие типы вершин могут быть выведены из нее.

Проблема очевидна, когда требуется добавить новый тип вершины в нашу иерархию, например, добавить оператор деления: не так просто объявить класс, производный для Binop. В таком случае должно также изменить реализацию Node::eval (и, вероятно, добавить новый элемент в перечисление типов операций класса Node). Если существующая иерархия вершин была частью библиотеки, наши пользователи вынуждены будут копировать и редактировать ее для получения своих собственных версий. Действуя так, они будут вынуждены либо вносить изменения в библиотеку в своей версии, либо не использовать стандартную библиотеку. В обоих случаях, наша цель - создание библиотеки - не будет достигнута.

Лучше будет инкапсулировать специфическую информацию вершины внутри соответствующей вершины, не записывая никакой специфической информации в типы вершин, находящихся в иерархии выше. Чтобы делать это максимально эффективно, нужно иметь возможность более обобщенно различать типы вершин в ходе выполнения программы, чем это сделано в текущей реализации Node::eval, где использовано явное приведение типов. Это можно выполнить при помощи виртуальных функций.

2.4.3 Виртуальные функции

Оставим ненадолго наш пример с абстрактным синтаксическим деревом и рассмотрим более простой. Предположим, мы работаем с набором типов фруктов, связанных наследованием:

```
class fruit {    public:   char *identify() { return "fruit"; } };
class apple : public fruit { public:   char *identify0{ return "apple"; } };
class orange : public fruit { public:   char *identify0{ return "orange"; } };
```

Пусть необходимо создать список разнородных "фруктовых" объектов, в котором каждый элемент идентифицируется отдельно. Будем использовать для этого односвязный список, описанный ранее:

```
typedef fruit *ETYPE;
#include "list.h"
void print_list(list *lst){    extern void print(char *);
while (lst)     { print (lst->value()->identify());   lst  = lst->next(); }    };
main()
{ list *lst = new list(new fruit());
  lst = new list(new apple, lst);     lst = new list(new orange,lst);
  print_list* lst);    }
```

Программа печатает слово "фрукт" трижды, даже когда список содержит orange, apple и fruit. Это поведение корректно, так как выражение lst->value0 имеет тип fruit*, и fruit::identify

возвращает указатель на строку символов "fruit". Естественно, элемент-функция вызывается в зависимости от типа указателя или ссылки, использованной для доступа к ней, а не от текущего типа объекта, на который ссылаются указатель или ссылка:

```
fruit *fp = new apple;
fp->ldentify();              // возвращает "fruit"
((apple *)fp)->identify(); // возвращает "apple"
apple *ap =(apple *)fp;
ap->identlfy();              // возвращает "apple"
```

Сравните это с реализацией Node::eval в этой главе ранее.

Для программы идентификации фруктов требуется, чтобы соответствующая идентифицирующая функция определялась типом текущего объекта, а не типом ссылки или указателя, использованных для доступа к нему. Для этого определим виртуальные функции: class fruit { public: virtual char *identlfy() { return "fruit"; } };

Виртуальные функции позволяют производным классам предлагать альтернативные версии функций базового класса. В объявлении fruit::identify как виртуальной функции, утверждается, что классы, выведенные из fruit, могут иметь собственные версии identify, и что эти функции будут вызываться на основе текущих типов объектов. Так, apple или orange будут иметь собственные версии виртуальной функции, вызываемой для них, даже если она рассматривается как fruit.

С добавлением ключевого слова virtual в объявление fruit::indentify, программа будет работать так, как требуется, печатая "orange", "apple" и "fruit".

Обратите внимание, что apple::identify и orange::identify являются виртуальными, хотя и не объявлены такими явно. Правило для определения, когда функция виртуальная, простое: функция является виртуальной, если она объявлена виртуальной, или если есть функция из базового класса с той же сигнатурой, которая является виртуальной. Сигнатура функции состоит из ее имени и типов формальных аргументов. Например, если объявление apple::identify, было char *identify(int=0) , то оно не будет виртуальным, так как тогда оно будет иметь сигнатуру, отличную от fruit::identify. Выходом программы в этом случае будет "orange", "fruit", "fruit". Если сигнатура функционального элемента производного класса совпадает с сигнатурой виртуального элемента базового класса, тогда возвращаемый тип тоже должен соответствовать типу результата функции базового класса. Это гарантирует, что динамическое (в ходе выполнения) связывание имен, производимое виртуальными функциями, сохраняет тип.

Комбинация связи "то же, что и" производного класса со своим общедоступным базовым и динамического связывания имен виртуальных функций - очень полезный метод абстракции. Например, функция print_list работает только со списками указателей fruit. Можно построить иерархию типов произвольной сложности и глубины, базируясь на классе fruit, не изменяя реализации print_list. Например, можно расширить иерархию, добавив тип banana - производного от fruit, или создать новый уровень, выводя macintosh и delicious из apple. Наследование, использованное таким образом, позволяет выделить общность из группы связанных типов и писать общие программы на базе абстракции. Специфические детали типа инкапсулированы внутри типа.

Используя эти концепции, можно теперь заново разработать реализацию абстрактного синтаксического дерева при помощи виртуальных функций:

```
class Node {    public:    Node() {}
                virtual ~ Node(){}
                virtual int eval() { error(); return (); }            };
class Binop :  public Node { public:    Node *left,*right;
            ~BlnopQ{ delete left; delete right; }
            Binop(Node *l, Node *r) { left = l;right = r; }       };
class Plus :     public Binop { public:    Plus(Node *I.Node *r) : Binop(l,r){}
```

```
            int eval() { return left->eval() + right >eval(); }     };
class Times:   public Binop { public: Times(Node *l,Node *r) : Binop(l,r){}
            int eval() { return left->eval() * right >eval(); }     };
class Uminus : public Node {    Node *operand;
            public:   Uminus(Node *o) { operand = o; }
            ~Uminus(){ delete operand;}
            int eval(); {return =operand->eval(); }    };
class int : public Node {    int value;
            public:    int(int v) { value = v; }
            int eval(){ return value; }     };
```

Функция eval объявлена в Node как виртуальная, поэтому eval в Plus, Times, Uminus и Int также являются виртуальными. Binop не объявляет функции eval, поэтому объект типа Binop вызывает Node::eval, унаследованную от базового класса Node. Теперь можно создавать и вычислять абстрактные синтаксические деревья:

```
int limited_use(){           // вычисляет -5+12*4
Node *np = new Plus( new Uminus ( new int(5)),
         new Times ( new int(12), new int(4) ) );
int result = np->eval();
delete np;  return result; }
```

Виртуальный вызов eval вызывает Plus::eval, так как вершина Plus - в корне абстрактного синтаксического дерева, на которое указывает np. Plus::eval вызывает виртуальные функции eval для своих левого и правого поддеревьев, что в этом случае приводит к вызову Uminus::eval и Times::eval соответственно, и т.д. На каждом шаге текущая функция для вызова определяется типом вершины в корне вычисляемого поддерева.

Класс Node также объявляет виртуальный деструктор. Хотя деструктор нельзя вызвать явно как элемент-функцию, его можно вызвать в виртуальном стиле. Строка delete np; вызывает виртуальный деструктор для корня абстрактного синтаксического дерева, на который указывает np. Это вершина типа Plus, которая не определяет деструктора, поэтому вызывается унаследованный деструктор Binop::~Binop. Binop::~Binop вызывает виртуальные деструкторы для своих левого и правого поддеревьев и так далее. В результате единственное удаляющее выражение освобождает целое абстрактное синтаксическое дерево.

Необходимо дополнительно учитывать, что конструкторы не могут быть виртуальными. Конструктор создает объект, тогда как виртуальная функция требует, чтобы объект уже существовал и по нему определяется, какая функция будет вызвана.

Использование виртуальных функций позволяет инкапсулировать специфические операции для вершины внутри объявления спецификатора типа вершины. Теперь можно добавлять новые типы вершин в иерархию, не влияя на существующие программы:

```
class Div : public Binop { public:   Div(Node *l,Node *r) : Binop(l,r){}
                   int eval() {return left->eval() / right >eval(); }    };
```

Объявив класс Div, можно создавать и вычислять (объекты типа Div как вершины абстрактного синтаксического дерева, не меняя реализаций других типов вершин.

2.4.4 Защищенные элементы

Node::eval играет важную роль: с одной стороны, "корневая" виртуальная функция, которая заставляет все стальные элементы eval с той же сигнатурой быть виртуальными, и с другой - нечто вроде стоп-черты. При отсутствии объявления Div::eval, любая попытка вызова eval для вершины типа Div приведет к вызову Node::eval и к ошибке. Никакие объекты Binop или Node не должны создаваться или вычисляться, а только объекты производных по отношению к ним классов. Отсекающая функция в корне иерархии - эффективный путь отлавливания ошибок такого типа, но, к сожалению, ошибка не

проявляется до момента выполнения, когда такая вершина вычисляется.

Предпочтительнее (и безопаснее) оградить пользователей иерархии вершин от создания объектов этих типов. Один из путей - сделать конструкторы для Node и Binop приватными. К сожалению, это также не позволит использовать конструкторы производным классам.

```
class Node{   Node() public:    };
class lnt : public Node {    int value;
        public: lnt(lntv){ // Ошибка! Неявный вызов приватного Node::Node
```

Аналогичная проблема возникает с элементами left и right класса Binop. Нам бы хотелось, чтобы они были приватными, чтобы предохранить их от неправильного употребления неаккуратными пользователями иерархии вершин, но они должны быть общедоступны для того, чтобы производные от Binop классы (Plus, Times и Div) могли их использовать.

Проблема решается при помощи защищенных элементов класса. Последовательность защищенных элементов вводится в определение класса при помощи protected: точно также, как общедоступные элементы предваряются public: , а приватные - private:. Защищенный элемент похож на приватный, за исключением того, что он доступен элементам и дружественным функциям классов, производных по отношению к классу, в котором он объявлен. Аналогично общедоступным элементам защищенные элементы базового класса являются защищенными в производных классах в случаях, если база общедоступна, и приватными - в противном случае.

```
class Node {   protected: Node()
               public: virtual ~Node(){}
               virtual int eval();   };
class Binop:   public Node {  protected: Node *left,*right;
               Binop(Node *l,Node *r) { left = l; right =r; }
               ~Binop(){ delete left; delete right; }     };
```

Теперь только элементы и дружественные функции Node, Binop и производных от них классов могут вызывать их конструкторы, поэтому другие фрагменты программы не могут размещать вершины этих типов. Кроме того, Binop::left и Binop::right будут доступны только для элементов и дружественных функций производных для них бинарных операторов.

2.4.5 Наследование как инструмент проектирования

Наследование - это метод проектирования, поскольку он позволяет абстрагировать проблему в более общем виде в базовом классе в корне иерархического дерева, а также моделировать и писать программы, работающие только с абстракцией. Специальные случаи (которые сами могут быть абстракциями еще более специальных случаев) могут выражаться производными по отношению к базовому классами.

Например, в случае с абстрактным синтаксическим деревом, проектирование было начато с общей концепции вершины и успешно продолжено иерархией типов вершин при помощи специализации. Но исходная абстракция, тем не менее, осталась. В программе типа

```
Void   print_value(Node *np){ extern void print(int);
print(np->eval0); }
```

используется только Node, и сложность специализированных типов, выведенных из Node, скрыта от программ общего назначения.

Использование наследования, как инструмента проектирования, аналогично пошаговому уточнению в парадигме функциональной декомпозиции. Пошаговое уточнение разделяет процедурные аспекты задачи на иерархию процедур, тогда как наследование преобразует аспекты типов задачи в иерархию типов.

Хотя результат развития программы пошаговой детализацией - строгая иерархия процедур, к этому результату не обязательно приходят напрямую разработкой сверху вниз. Иногда сначала пишутся низкоуровневые процедуры, чтобы посмотреть, как их реализация

влияет на структуру процедур более высокого уровня.

Аналогичным образом, в сложной программе, работающей со многими различными типами, может быть не ясно, как типы связаны друг с другом, если они вообще связаны. Иногда явную общность можно понять только после программирования реализаций нескольких, предположительно разных, типов. Наследование можно тогда использовать для расширения общих частей их итерфейса и реализации. Часто эта общность - повод для более глубокой абстракции, и "открытая" иерархия наследования становится рабочей абстракцией проектирования.

2.4.6 Наследование для расширения интерфейса

В некоторых рассматриваемых примерах производные классы являются расширением или специализацией базовых, которые сами по себе вполне полезны, list2 выведен из list и Pathname - из String, но и list, и String - отличные функционирующие типы. С другой стороны, базовый тип Node иерархии вершин синтаксического дерева не является полностью функциональным, и требует, чтобы его определение было дополнено наследованием классов. Поэтому Node формирует некий частичный тип, наполовину реализацию, наполовину шаблон, который дополняется в производных классах.

Можно расширить эту концепцию и определить базовый класс, служащий только шаблоном для производных классов, для того, чтобы гарантировать, что производный тип удовлетворяет конкретному интерфейсу, т.е. к производному классу можно относится не как к расширению или специализации базового, но как к реализации интерфейса, специализируемого базовым классом.

Рассмотрим задачу специфицирования интерфейса между ядром и драйвером устройства в операционной системе UNIX. Для выражения системного вызова на открытие, закрытие, чтение, запись и т.д. данному устройству, ядро индексирует двумерную таблицу переключателей устройства с главным номером устройства и кодом, соответствующим типу системного вызова (открыть, закрыть и т.д.). Главный номер устройства определяет вид устройства (диск или терминал и т.п.), а младший номер устройства используется для определения конкретного устройства определенного типа (из нескольких дисков или терминалов). Таблица содержит адреса функций, соответствующих устройству; операционная пара используется для индексации таблицы.

Ядро выполняет системный вызов, обращаясь к соответствующей программе косвенно через таблицу переключателе* устройства. Эта таблица определяет интерфейс между ядром UNIX и программами драйвера устройства. В схеме есть недостатки. Во-первых, размеры и содержание таблицы фиксированы после построения (компиляции) ядра и добавление нового типа устройства требует перестройки ядра. Во-вторых, нет гарантии, что функции, адреса которых содержатся в таблице, имеют нужные типы. В третьих, информация, относящаяся к специфическому типу устройств, рассосредоточена в нескольких местах и не помогает нам рассматривать устройство в целом.

Альтернативный способ реализации интерфейса состоит в определении типа драйвера устройства, инкапсулирующего поведение обобщенного устройства:

```
class Device {    public: virtual int open(char *,int,int);
virtual int close(int);
virtual int read(int,char * ,unsigned);
virtual int write(int,char * ,unsigned);
virtual int ioctl(int.int...);
Device();    ~Device();    };
```

Функциональные элементы этого обобщенного драйвера реализуют стандартное поведение:

```
int Devlce::open(char *path,int oflag,int mode)
{ return nulldev(); }
```

Теперь ядро может работать со всеми устройствами через этот обобщенный тип

драйвера точно так же, как выражение всех типов вершин абстрактного синтаксического дерева через обобщенный тип Node.

Создаем тип для драйвера конкретного вида устройств при помощи наследования из этого шаблона:

```
class Terminal : public Device {
        public:   int open(char *,int,int): int close(int);
        int read(int,char *,unsigned);
        int write(int,char *,unsigned);
        int ioctl(int,int...);
        Terminal();    ~Terminal();    };
```

Любая часть интерфейса, не переопределенная явно в производном типе устройства, по умолчанию совпадает с программой в базовом типе.

```
class Mem_mapped_io : public Device {
            public:    int read(int,char *,unsigned);
            int write(int,char *,unsigned);
            int ioctl(int,int...);   };
```

Иерархия драйверов устройств заменяет таблицу переключателей устройства. Можно рассматривать конкретные производные типы устройств как замену главных номеров устройств, а динамическое связывание, производимое виртуальными функциями, как аналог таблицы коммутации. Для замены младших номеров устройств создадим объект устройство для каждого физического устройства данного типа. Например, если к каналу подключены пять дисководов, существует пять соответствующих дисковых объектов под управлением ядра; при каждом входе пользователя в систему создается соответствующий терминальный объект, который уничтожается при выходе.

2.4.7 Множественное наследование

Производный класс может иметь любое число базовых классов. Использование двух или более классов называется множественным наследованием. Хотя использование множественного наследования встречается реже, чем простого, множественное наследование полезно для создания классов, комбинирующих поведение двух или более других классов.

Например, можно создать тип, отслеживающий данное условие и отражающий его статус на экране. Предположим, у нас есть библиотека растровой графики с абстрактным типом данных циферблат, отражающим изменяющееся значение:

```
class Dial {   // несущественные детали реализации
protected: double value;
Dial(char *,double,double);    ~Dial();   };
```

Конструктор Dial отражает связь с меткой и с диапазоном измерений между значениями второго и третьего аргументов. После создания объект Dial отображает текущее значение Dial::value. Пусть также доступен некоторый тип модель.

```
class Sampler {   // детали реализации
        protected: double freq;
        virtual void sample();
        Samplei (*double);    ~Sampler();   };
```

Объект типа Sampler вызывает свою виртуальную функцию sample каждые Sampler::freg секунд. Монитор представляет собой одновременно Dial и Sampler. Выразим это с использованием множественного наследования для получения черт обоих классов:

```
class Monitor :  public Sampler, public Dial {
        void sample() { value = get_value(); }
        protected:   virtual double get_value();
        Monitor(char *lab,double, I,double, h.double, f)
        : Dial(lab,l,h),Sampler(f){}    };
```

Запись class Monitor public Sampler, public Dial объявляет Monitor как новый классовый тип, производный и от Dial, и от Sampler. Использование нескольких базовых классов - естественное расширение случая с наследованием от одного базового. Любое число имен классов может быть представлено через запятую в списке базовых классов, но ни одно из имен не должно появляться в одном списке дважды.

Конструктор Monitor использует список инициализации элементов для явного вызова конструкторов классов. Порядок инициализации определяется списком инициализации элементов, при этом любые неявные инициализации базовых классов вызываются в том порядке, в котором базовые классы появляются в списке в определении классов. Напоминаем, что базовые классы всегда инициализируются раньше элементов, даже если имя элемента появляется раньше имени базового класса в списке инициализации.

Область видимости класса Monitor помещается внутри области видимости всех его базовых классов. Поэтому Monitor::sample - виртуальная функция, переопределяющая Sampler::sample. Ссылка на value в Monitor::sample является ссылкой на Dial: :value.

Множественное помещение может быть источником двусмысленностей, которые не возникают при простом наследовании. Для иллюстрации некоторых из этих проблем вернемся к иерархии фруктов и подключим иерархию деревьев:

```
class fruit { public:   virtual char *identify(){ return "fruit"; }   };
class tree { public:   virtual char *identify() { return "tree"; }   };
```

Некоторые типы являются одновременно fruit и tree.

```
class apple : public fruit, public tree {};
apple *ap = new apple;
ap->identify();     // ошибка!
```

Этот фрагмент не компилируется, поскольку вызов функции двусмысленен. Так как контекст apple находится в области видимости как fruit, так и tree, вызов может относиться и к fruit::identify, и к tree::identify. Нужно сделать этот вызов однозначным, указав базовый класс при помощи оператора видимости: ap->fruit::identify();   или используя приведения типов: ((fruit *) ap)->identify();

Лучшим решением будет определение функции identify для класса apple с нужным поведением.

```
class apple : public fruit, public tree {
public:   char *identify(){ return "apple"; }   };
ap->ldentify();
```

Все обычные связи между производным классом и каждым из его базовых при множественном наследовании сохраняются такими же, как и в случае единичного наследования. Например, apple::identify - виртуальная функция, переопределяющая и fruit::identify, и tree::identify. Кроме того, так как и fruit, и tree являются общедоступными базовыми классами apple, существуют предопределенные преобразования из apple и в fruit, и в tree:

```
apple a;
tree &t=a;          // да, apple это tree
fruit *f->&a;       // да, apple это fruit
*f=t;               // ошибка! tree это не fruit
a=t;                // ошибка! fruit это не apple
```

Вернемся к классу Monitor. Используя множественное наследование, скомбинируем две различные концепции Dial и Sampler для создания нового типа. Такое использование множественного наследования для комбинирования типов очень важно.

Теперь можно выводить специализированные типы из класса Monitor. Тип, выведенный из Monitor, должен представить инициализирующие значения, которые передаются через конструктор Monitor классам Dial и Sampler, и определить функцию, предоставляющую значения для мониторинга:

```
class Mem usage: public Monitor {
```

```
char *start,   *max;
public:    double get_value() {    return (current_top()-start)/(max-start); }
Mem_usage(): Monitor(" Memory Usage" ,0,100,0.1),
start(current_top()),max(get_limit()) {}     };
```

Объект Mem_usage отслеживает процент доступной памяти для процесса, в котором он появился. Виртуальная функция get_value возвращает отношение доступной памяти с момента создания объекта Mem_usage. Системная функция, названная здесь current_top, выдает верхний адрес памяти процесса для вычисления в get_value. Конструктор отмечает шкалу монитора, устанавливает интервал отображаемых значений от 0.0 до 100.0 и устанавливает дискретность, равную десять раз в секунду. Обратите внимание, что current_top снова используется для записи начального верхнего адреса процесса. Другая системная функция, названная get_limit, выдает максимальный адрес, до которого может увеличиваться память процесса.

В качестве другого примера предположим, что необходимо отслеживать скорость мыши, связанной с нашим терминалом. Пусть нам доступны обычные типы растровой графики: объекты типа Point (позиции на экране в декартовой системе координат) и Mouse (объект, представляющий мышь).

```
class Mouse_velocity : public Monitor {
Point prev;
public:   double get_value() {
        extern const int PIXELS_PER_INCH;
        double value=dist(Mouse.abs_pos,prev)
        / (freq * PXELS_PER_INCH);
        prev = Mouse.abs_pos;
        return value;    }
Mouse_velocity():Monltor("Veloclty",0.120,.01)
        { prev = Mouse.abs_pos;}     }
```

Объект Mouse_velocily отражает текущую скорость мыши в дюймах в секунду. Обратите внимание, что виртуальная функция get_value обращается к freq, элементу Sampler, который отличается на два уровня наследования.

2.4.8 Виртуальные базовые классы

Тип Monitor использует множественное наследование для комбинации двух полностью разных типов. Множественное наследование также можно использовать и для комбинации более близко связанных классов. Например, мы можем захотеть создать новый тип устройства, имеющий свойства двух существующих типов устройств:

```
class Monitored_devlce: public Device.public  Monitor { //... };
class Network_devlce: public Device,public Protocol { //... };
```

Monitored _device отслеживает темп передачи данных через устройство, а Network_device - интерфейс к драйверу платы сети, реализующий данный протокол сети. Создадим новый тип устройства, наблюдающий за темпом работы сети. Можно сконструировать новый тип устройства с самого начала:

```
class Monitored network device :
public Device.public Monitore.public Protocol { //... };
```

но тогда нельзя использовать новый тип как Network_device или как Monitored_device, так как между этими типами не будет наследственной связи. Альтернативный подход - выведение из двух существующих устройств - вызовет появление в иерархии двух базовых классов Device:

```
class Monitored_network_device;
public Monitored_devlce, public Network_device{ //... };
```

В некоторых случаях это именно то, что требовалось, но это не то, что мы хотим от нашего нового типа устройства. Попытка использования нового типа как устройства приведет к неопределенным ошибкам, и нужно будет каждый раз уточнять конкретный Device, на который мы ссылаемся.

```
Monitored_network_device iso;
iso.open("/dev/iso",O_RDWR.0);     // ошибка! двусмысленно
iso. Device::open("/dev/lso",O_RDWR,0); //ошибка! Все еще двусмысленно...
iso.Network_device::open("/dev/iso',O_RDWR,0);
    // OK, устройство типа Network_device
iso.Monitired_device::open("/dev/iso",O_RDWR,0);
    //OK. устройство типа Monltired_device
```

Более общая проблема состоит в том, что если даже мы работаем с одним физическим устройством, наш тип содержит два различных его представления. Это может вызвать трудности. Например, так как конструктор Device будет активизирован дважды для каждого объекта класса Monitored_network_device, ядро операционной системы, работающее с устройством только через интерфейс, предлагаемый типом Device, может заключить, что имеются два устройства там, где в действительности существует только одно.

Создадим класс, выведенный и из Monitored_device, и из Network_device, но представляющий единственный Device. Сделаем это с помощью виртуальных базовых классов:

```
    class Monitored_device:  public virtual Device, public Monitor { //... };
    class Network_device:  public virtual Device, public Protocol { //...};
    class Monitored_network_device: public Monitored_device.public
            Network_device{ //... };
```

Объявление базового класса виртуальным говорит о том, что его представление будет разделяться с любым другим виртуальным появлением этого базового класса в объекте. В классе Monitored_network_device Device появляется только один раз, так как все упоминания Device объявлены виртуальными. Так как существует единственный Device, неопределенные ссылки на его элементы не являются более двусмысленными:

```
    iso.open("/dev/iso",ORDWR,O);
```

Точно также, конструктор Device вызывается для объекта Monitored_network_device только один раз, так как необходимо инициализировать только один Device.

При использовании виртуальных базовых классов существуют некоторые ограничения. Во-первых, базовый класс, являющийся виртуальным, не может явно инициализироваться упоминанием в списке инициализации конструктора. Так как класс, имеющий конструктор, должен инициализироваться, то класс, используемый как виртуальный базовый, должен либо не иметь конструктора, либо иметь конструктор, который может вызываться без аргументов. Любая (неявная) инициализация виртуальных базовых классов выполняется раньше любого другого базового, включая те, которые явно инициализируются в списке инициализации элементов. Второе ограничение касается приведения типов: запрещается приводить указатель на виртуальный базовый класс к типу указателя на класс, прямо или косвенно выведенный из него.

2.5 ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ

Приемы процедурного программирования концентрируются на алгоритмах, используемых для решения задачи. При этом не обращается внимание на структуры данных, задействованные функциями, как на отдельные части организации программы. В противоположность этому, объектно-ориентированное программирование (ООП) концентрируется на сути задачи,
для которой пишется программа: элементы программы разрабатываются в соответствии с объектами в описании задачи. Общий подход ООП состоит в определении набора объектных типов. Объектные типы - модули, интегрирующие структуры данных, которые представляют

элементы задачи, с операциями, необходимыми для выработки решения. Как только объектные типы определены, создаются образцы объектов для конкретной задачи и вызываются операции для произведения обработки.

В С++ классы служат для представления объектных типов, а функциональные элементы предоставляют средства для встраивания операций в тип.

2.5.1 Проектирование в терминах объектов

Объектно-ориентированное проектирование программ - это расширение использования абстракции данных. Абстрактные типы данных, или объектные типы, не только скрывают структуру данных, но инкапсулируют и все функционирование как операции над объектами. Суть объектно-ориентированного проектирования состоит в нахождении наиболее подходящих объектных типов.

Перед обсуждением некоторых принципов проектирования объектных типов, рассмотрим две объектно-ориентированных программы, сортирующие список целых чисел. Чтобы сконцентрироваться на проектировании, мы не будем показывать деталей реализаций. Первая программа использует абстрактный тип данных intList, содержащий операторы для сортировки и выдачи списка. Целые числа считываются в список при создании объекта intList.:

```
class intList {   public:   intList(); sort(); write();   };
main()      {   intList *ilist = new intList();
                ilist->sort();    ilist0->write();   }
```

Другая программа использует абстрактный тип данных Sortedintlist, который читает целые числа в процессе создания отсортированного списка. Sortedintlist содержит оператор для выдачи списка:

```
class Sonedintlist{   public:   Sortedintlist();   write();   };
main(){   Sortedintlist *slist= new Sortedimlist();    slist->write();   }
```

Какой из этих типов лучше спроектирован, или они оба одинаково хороши?

Хороший способ проектирования объектных типов состоит в поиске существительных и глаголов в описании решаемой задачи. Существительные при проектировании программы становятся объектами, а глаголы становятся операциями. Используя это правило для задачи сортировка списка целых, разработка программы должна включать тип "список целых", имеющий операцию "сортировать". Программа intList ближе к "естественному" решению, на которое намекает постановка задачи, тогда как вторая программа вводит специализированный тип, не являющийся необходимым для решения.

Тип отсортированного списка может пригодиться для различных задач, особенно как специальный случай списка. Например, в программе, выполняющей множество разных обработок списков, тип отсортированного списка может пригодиться для оптимизации поиска или объединения списков. Тип сжатого списка, представляющий списки с неповторяющимися элементами, также может быть частью хорошей программы обработки списков. Это выявляет другой аспект объектно-ориентированного проектирования - использование наследования для создания специализированных версий базовых объектных типов. Вместо создания совершенно другого объектного типа для каждой программы можно строить новые типы из общих базовых типов. В нашей гипотетической программе обработки списков ,будем использовать List как базовый класс и получим Sortedlist и Compressedlist как производные от него. Операция sort может быть объявлена как виртуальная функция в базовом классе. Версия sort в Sortedlist не будет делать ничего. Для Compressedlist специальная версия sort не будет нужна. Способ определения ситуации, когда для специализации типов можно использовать наследование при объектно ориентированном проектировании состоит в поиске прилагательных в описании программируемой задачи.

Для задачи обработки списков "отсортированный список" и "сжатый список" определяют, где можно использовать производные типы.

Описание задачи не всегда подходящим образом определяет наилучшие объектные типы для реализации программы. Нахождение абстракции, общей для нескольких объектов, так что у них может быть одна реализация базового типа, не обязательно является прямой задачей. Проектирование программы - зачастую процесс проб и ошибок, требующий несколько попыток точного описания задачи, идентификации абстракций и предварительных реализаций. Это не ново и не является особенностью объектно-ориентированного проектирования.

Новое в ООП - способ рассмотрения программ как реализаций объектных типов, а не как реализаций алгоритмов. Этот концептуальный сдвиг иногда трудно освоить программисту. Абстрактные концепции и сущности, не являющиеся реальными вещами, может быть особенно сложно распознать как кандидатов в объектные типы.

Несложно осознать объектные типы, отвечающие знакомым нам физическим объектам реального мира. Например, устройства типа драйверов диска или ленты являются физическими объектами. Устройства - один из объектов в проблемной области ядра операционной системы. В ранее рассмотрен класс, служащий общим интерфейсом драйверов устройств.

```
class Device {    public;
        virtual int open(char *,int,int);
        virtual int close(int);
        virtual int read(int,char *,usigned);
        virtual int write(int.char * ,unsigned):
        int ioctl(int,int...);
        DevIce();    ~Device();    };
```

Этот класс также может быть рассмотрен как базовый объектный тип для всех разнообразных устройств при объектно ориентированном проектировании операционной системы. Используя наследование, можно создать вариации типов для представления дисковых и ленточных устройств.

```
class Disk :  public Device {
        public;    int open(char *,int,int);
        int close(int);
        int read(int,char *,usigned);
        int write(int,char *,unsigned);
        int ioctl(int,int...);
        Disk();    ~Disk();    };
class Tape: public Device {    public: // и т.п.    };
```

Хотя существует ряд различных типов устройств, ядро рассматривает их как несколько идентичных объектов устройств.

Многие программы никогда не работают напрямую с интуитивно понятными представлениями физических объектов. Необходимые в этих случаях объектные типы являются частью абстрактной проблемной области. В операционной системе процесс может быть объектом, даже если "process" понимается как деятельность. Типичное ядро операционной системы может представлять процесс разбросанными структурами данных и функциями, манипулирующими различными структурами. В объектно-ориентированном проектировании эти структуры данных и функции упаковываются в класс для создания объектного типа "процесс". Интерфейс объектного типа - это множество функций, с помощью которых ядро управляет процессами. В приведенном ниже примере class Proc предлагает функции для помещения процесса в состояние дремлющего ожидания конкретного события с конкретным приоритетом, пробуждения дремлющего процесса, сохранения и восстановления состояния работающего процесса и для посылки сообщения процессу в форме номера сообщения. Конструктор реализован так, что новый Proc может быть создан из другого, как при порождении параллельного процесса. Деструктор удаляет завершенный процесс из системы. Операции подкачки процессов вне и внутри системы не

поддерживаются интерфейсом примера, так как это предполагаемая часть управления памятью скрытых элементов объекта Proc:

```
class Proc { public:
        int sleep(Sleepq *,int);
        void wakeup();
        int save_runstate();
        void resume_runstate();
        int send_msg(int);
        Proc(Proc &);    ~Proc();    };
```

Процессы-объекты теперь можно добавить в область рассмотрения ядра.

Другой аспект объектно-ориентированного проектирования состоит в определении родового типа. Так, в программе обработки списков могут быть "списки целых", "списки строк" и "списки записей о работающих". Все эти списки с различными типами элементов могут быть построены как сущности родового типа список. Ключ к познанию родового типа состоит в том, что это вместилище других типов. Вероятно не стоит строить родовой тип, если только программист не использует более одного примера типа, или тип не будет повторно использоваться в других программах.

Для программиста, использовавшего для декомпозиции проектирование в терминах функций, необходима некоторая практика при переходе к проектированию в объектном стиле. Если программист вступил на путь объектно-ориентированного мышления, он должен быть внимателен, чтобы не перестараться. Как в предыдущем примере, создающем сортированный список вместо сортировки списка, обычно можно создать объектный тип для любой цели. Множество сфабрикованных объектных типов усложняет проектирование программы. Хорошее объектно ориентированное проектирование ясно соответствует задаче, которую должна решать программа, упрощая понимание, реализацию и сопровождение программы.

2.5.2 Объектные типы как модули

В ООП элементы задачи отвечают элементам проектирования, которые являются объектными типами, формирующими программные модули. Одно из преимуществ этого метода построения программ состоит в том, что есть концептуальное единство всех фаз создания программы, и получившиеся модули объектного типа легко повторно использовать.

Концептуальная структура программы сохраняется не только от описания задачи до реализации, но и при совершенствовании разработки. После идентификации объектных типов проектирование уточняется добавлением деталей этих типов, что контрастирует с пошаговым уточнением функции сверху вниз, при котором проект более высокого уровня преобразуется в другую структуру. Имея полную структуру проекта до того, как известны все детали, мы получаем возможность быстрого прототипирования. Прототипные реализации объектных типов могут быть получены помещением небольшого числа операций и скрытием незаконченных деталей, позволяя проверить осуществимость проекта на ранних стадиях. Возможность быстрого построения прототипов сокращает расходы на проектирование методом проб и ошибок.

Рассмотрим программу обработки слов, манипулирующую строками. Так как потребность в объектах типа строка при разработке определена, class String можно будет быстро реализовать с использованием простых структур данных и существующих библиотечных программ, как в примере с class String, приведенном ранее. Эта простая версия String может быть использована при прототипировании для точного определения, какие операции над строками требуются. Добавление операций в прототипный class String происходит параллельно с уточнением деталей в разработке программы. Пока доказывается осуществимость разработки, прототипную версию может заменить более сложная реализация String.

Поскольку объектно-ориентированное проектирование не является проектированием сверху вниз, полученные программные модули более независимы и, поэтому их легче повторно использовать. Модуль не включается в иерархию проекта конкретной программы, и поэтому не зависит от конкретной структуры программы. Модуль объектного типа является реализацией элемента конкретной проблемной области вместо того, чтобы служить подфункцией одного решения задачи. В этой связи модуль может использоваться в любой программе в той же или смежной проблемной области. Например, String может использоваться в любой программе, которой необходим тип для представления слов или текста.

Другой аспект повторного использования модуля состоит в легкости расширения объектных типов наследованием. Исходный модуль остается нетронутым, тогда как новое функционирование добавляется в производный тип. Модули, являющиеся общими объектными типами, могут быть легко повторно использованы для различных вариантов задачи. Специализированный тип Pathname, выведенный из String в - пример того, как общий тип может быть адаптирован для конкретных целей при помощи наследования.

2.5.3 Динамический объектно-ориентированный стиль

ООП связано со стилем программирования, который заключается в создании объектов во время выполнения программы и динамическом связывании операций над объектами. Этот стиль программирования восходит к интерпретирующим объектно-ориентированным языкам и системам типа Flavors и Smalltalk. Системы были разработаны для гибкости во время выполнения и показали себя как действенные инструменты прототипирования, непосредственного решения задач и имитационного моделирования.

Можно писать объектно-ориентированные программы на C++, состоящие полностью из создания образцов объектов во время выполнения и имеющие динамическое связывание операций над объектами с виртуальными функциями. Создание объектов и вызов объектных операций делается в том месте текста программы, которое компилируется. Хотя вызовы виртуальных функций динамически связаны, они проходят статическую проверку типов. То, чего не хватает C++ в смысле интерактивной гибкости, вызвано либо сильной проверкой типов на этапе компиляции, либо сделано для эффективного выполнения программ.

Динамическое ООП часто используется для моделирования. Для моделирования различных сущностей в задаче реализуются объектные типы. Образцы объектов в этом случае задействуются в роли моделей. Объекты взаимодействуют, вызывая операции один над другим и создавая новые объекты, включающиеся в моделирование. Рассмотрим программу, моделирующую движение самолетов в аэропорту. В модели аэропорта используется библиотека задач C++ для того, чтобы разные действия работали как сопрограммы. Сопрограммы в моделировании - это конструкторы объектов класса task или производных от него классов. Библиотека задач реализует диспетчеризацию без вытеснении (nonpreemptive scheduling), поэтому каждая задача должна себя контролировать, чтобы дать возможность другим задачам работать. В этом моделировании выполнение задачи управляется функциональным элементом task, void task::delay(int), который приостанавливает выполнение задачи на данное число моделируемых единиц времени. Глобальная переменная библиотеки clock отслеживает время. Функция void task::cancel(int) заканчивает выполнение задачи.

Библиотека также предоставляет классы, реализующие очереди. Классовые типы qhead и qtail представлены в библиотеке отдельно, потому что они должны иметь возможность функционировать независимо для различных задач, ставящих и выбирающих объекты из очереди.

Функции для работы с очередью:

qtail *qhead::tail() - получение qtail, который сцепляется с конкретным qhead;

int qtail::put(object *) - добавляет объект в хвост очереди;

object *qhead::get() - удаляет объект с головы очереди.

ФУНКЦИИ ДЛЯ ПРОВЕРКИ, ЯВЛЯЕТСЯ ЛИ ОЧЕРЕДЬ ЗАПОЛНЕННОЙ ИЛИ ПУСТОЙ:

int qtail::rdspace() - возвращает объем, оставшийся для добавления объектов в очередь;

int qhead::rdcount() - дает число объектов, находящихся в очереди.

Классовый тип object - это базовый тип, из которого должны быть выведены все элементы очереди. Так как task и другие классы из библиотеки задач имеют корневой базовый тип object, объекты этих типов можно помещать в очереди.

Библиотека задач также содержит генераторы случайных чисел. Будем использовать class urand для получения случайных чисел, равномерно распределенных по данному интервалу, через функцию int urand::draw()

Функция main создает объект-задачу Airport, дает ей возможность поработать период моделируемого времени, затем приостанавливает моделирование. Как только Airport создан, main сама становится задачей, которая выполняется как сопрограмма с другими задачами. Системный указатель задачи thistask. необходим для выполнения main как сопрограммы:

```
#include "Airport.h"
main(){    Airport *ap = new Airport;
thistask->delay(500);
delete ap;    thistask->resultls(0);  }
```

Заголовочный файл аэропорта включает заголовок библиотеки задачи. Объектные типы для моделирования строятся из типов библиотеки задач.

```
/* Это файл Airport.h */
#include "task.h"
class Plane : object {
    static intfitcount;    long start;    int fitno;
    public:    Plane() { fitno =++fitcount; }
    long howlong(){ return clock-start; }
    void set() { start=clock; }
    int fit(){ return fitno; }    };
class Ptane() {
    qhead *head;    qtail *tail;
    public:  Plane() {    head = new qhead(ZMODE,50);
                          tail = head->tail();}
    void put(Plane *p){  p->set();  tall->put((object*)p);}
    Plane *get(){  return (Plane *)head->get();}
    int isroom(){  return tail->rdspace(); }
    int notempty(){ return head->rdcount(); }    };
class AirControl : public task {
    Plane() *landing, *inair;
    public:   AirControl(Plane() *,Plane() *);    };
class GroundControl : public task {
    Plane *takingoff, *onground;
    public:   GroundControt(PlaneQ*, PlaneQ*);   };
class Airport : public task {
    PlaneQ *takeingoff, *landing, *inair, *onground;
    AirControl *acontrol: GroundControl *gcontrol;
    public: Airport();    ~Airport();   };
```

Объектные типы в этом примере: Plane - для самолетов, PlaneQ - для очередей самолетов, ожидающих возможности использования аэродрома, AirControl - для управления самолетами в зоне аэропорта, GroundControl - для управления самолетами на земле и Airport,

состоящий из очередей самолетов и контроллеров для управления движением в воздухе и на земле. Airport сам управляет своим выполнением.

Plane - класс, производный от object - базового типа, с которым работает система задач. Реализация Plane следит за временем полета и задержкой, используя переменную clock из библиотеки задач, которая измеряет модельное время.

PlaneQ инкапсулирует qhead и qtail, реализующие очередь ожидающих самолетов. PlaneQ устроена так, что она содержит пятьдесят самолетов, и get от пустой очереди возвращает указатель NULL.

AirControl, GroundControl и Airport - это задачи (tasks). Когда создаются образцы этих типов, их конструкторы выполняются как сопрограммы. Конструктор Airport создает очереди самолетов, в которых они ожидают обслуживания, а затем - управляющие задачи, обслуживающие очереди. После этого происходит бесконечный цикл до тех пор, пока задача Airport не аннулируется. Аэропорт обслуживает один садящийся и один взлетающий самолет за каждые десять тактов модельного времени. Если нет самолетов, уже ожидающих, берутся самолеты из очередей. Сообщения о движении самолета делются с использованием printf. Израсходование самолетами топлива в ожидании посадки неслишком элегантно моделируется аварийным приземлением, удаляющим самолет из модели.

```
Airport::Airport(){
takingoff = new PlaneQ;
inair= new PtaneQ;
landing = new PlaneQ;
onground = new PlaneQ;
acontrol = new AirControl(landing>inair);
gcontrol = new GroundControl(takingoff,onground);
Plane *tp=0, *lp=0); // ожидающие самолеты
int maxwait=30;
for(;;){ delay(10);
    if (!lp) lp=landing->get();
    if (!tp) tp=takingoff->get();
    if (lp) {    if (onground->isroom())
      { printf("рейс %d совершает посадку \n", lp->flt());
        onground->put(p);    lp=0;          }
          else if (lp->howlong() >> maxwait)
          { printf("рейс %d разбился!\п", lp->flt());
          delete lp;   lp=0;    }
          else printf("рейс %d .посадка задерживается \n",lp->flt(); }
    if(tp){
    if (inair->isroom()) { printf("рейс %d взлетает off\n", tp->flt0);
          inair->put(tp); tp=0; }
    else printf("рейс %d .взлет откладывается \n", lp->flt0);   }   }   }
```

Деструктор Airport прекращает задачу AirControl тем, что в список ожидающих посадку не поступает больше самолетов. Это продолжается до тех пор, пока все ожидающие самолеты не приземлятся, затем прекращаются оставшиеся задачи, моделирующие аэропорт.

```
Alrport::~Airport() { acontrol->cancel(0);
    while (tanding->notempty())
    thistask->delay(10);
    gcontrol->cancel(0);
    printf("appoint закрыт \n");
    cancel(0);    }
```

AirControl обрабатывает запросы от самолетов на посадку с интервалом от одного до тридцати тактов. Случайный интервал между прибытиями реализуется с использованием urand - тип генератора случайных чисел, предлагаемый библиотекой задач. В нашем случае, когда моделируется единственный аэропорт, самолеты создаются для постановки в очередь

на приземление или пересоздаются из уже взлетевших из аэропорта.

```
AirControl::AlrControl(PlaneQ landing, PlaneQ *inair){
    urand n(1,30);
    for(;;){ delay(n,draw());
            Plane *p = inair->get();
            if (!p)  p=new Plane;
            if (landlng->lsroom()) {
                printf("рейс %d запрашивает посадку\п",p->fit());
                landing->put(p); }
            else { printf("рейс %d направлен на другой аэродром \п".
                  p->fit());
delete p;          }          }          }
```

GroundControl использует для обслуживания самолета на земле от десяти до тридцати тактов, а затем самолет может снова взлетать.

```
GroundControl::GroundControl(PlaneQ *takingoff, PlaneQ *onground){
    urand n(10,30);
    Plane *p=0;
    for(;;){ delay(n.draw0);
            if (!p) p=onground->get();
            if(p){ if (takingoff->sroom())
            { printf("peach %d отправляется \n", p->fit());
                    takingoff->put(p);   p=0;      }
    else printf(^"рейс %d задерживается \n",p->fit());   }      }  }
```

Вот результат работы программы, моделирующей аэропорт:

рейс 5 запрашивает посадку
рейс 82 разбился
рейс 84 запрашивает посадку
рейс 9 посадка задерживается
рейс 48 отправляется
рейс 9 совершает посадку
рейс 48 взлетает
рейс 48 запрашивает посадку
рейс 17 отправляется
рейс 83 совершает посадку
рейс 17 взлетает
рейс 17 запрашивает посадку
рейс 5 разбился!
рейс 84 разбился!
рейс 32 отправляется
рейс 48 совершает посадку
рейс 32 взлетает
рейс 17 разбился!
рейс 60 отправляется
Аэропорт закрыт

В приведенном примере возникло множество аварийных приземлений, что вызвано, по-видимому, не перегрузкой приземляющимися самолетами, а временем ожидания и перегрузкой наземных очередей. Можно испробовать различные значения для наибольшей задержки посадки, ожидания наземного обслуживания, интервала между прибытиями и размера очередей и рассмотреть их влияние на появление аварий, задержки и переадресованные полеты.

2.6 УПРАВЛЕНИЕ ПАМЯТЬЮ

В книгах по программированию тема управления памятью часто считается не слишком важной или же областью, которой занимается язык или операционная система. Соответствующий контроль над управлением памятью, тем не менее, существенен для корректности программы и может быть важен для ее эффективности.

Неэффективное выделение и освобождение памяти может сильно снизить эффективность программы, порождая слишком много системных вызовов для получения дополнительной памяти, фрагментируя доступную память и вызывая метания, при которых страницы

непрерывно подкачиваются и откачиваются из памяти, или (в определенных аппаратных средах) полностью выходя за пределы доступной памяти. Прямолинейные попытки избавиться от таких неэффективностей могут привести к некорректному управлению памятью, как, например, преждевременное освобождение или случайное переименование блока памяти.

Требования к управлению памятью определяются задачей. В некоторых случаях все потребности в памяти статически определены, в других - они почти полностью динамические. Часто имеется специфическая информация о задаче, которую можно учитывать при построении эффективной схемы управления памятью. Хотя C++ поддерживает общее управление памятью по умолчанию, он также дает возможность приспосабливать управление памятью для данной проблемной области, класса или приложения.

2.6.1 Управление памятью при помощи конструкторов и деструкторов

Мы рассматривали конструкторы как средство для размещения новых объектов, как инициализаторы уже существующих объектов и как род предопределенного пользователем преобразования. Это просто три различных взгляда на одну и ту же операцию, создающую объект данного типа из последовательности значений.

В отличие от функциональных элементов, конструкторы имеют скрытый механизм, который явно не приводится программистом. Мы уже писали о некоторых его свойствах, таких, как автоматическая инициализация базовых классов и элементов, имеющих конструкторы, которые можно вызывать без аргументов. Рассмотрим некоторые конструкторы детально и посмотрим, что происходит за кулисами:

```
complex::complex( double r=0.0, double i=0.0;)
    { re = r;        im - i;   };
```

Конструктор класса complex прямолинеен. Во-первых, он убеждается, что есть память для комплексного числа, а затем он выполняет тело конструктора (представленное программистом) для инициализации памяти. Каждый конструктор начинается с некоторого неявно вставленного кода, проверяющего значение this, чтобы определить, существует ли объект для работы. Если this - пустой, то объекта нет и конструктор вызывает operator new для получения памяти под объект.

Два случая, в которых неявный аргумент this не null - это для статических и автоматических объектов. Статический объект (со спецификаторами static либо extern) существует всегда. Автоматический объект получает память автоматически при входе в функцию, в которой он определен.

```
complex zero;                    // статическая
complex zilch() {  complex r(1,2);        //автоматическая
return r * zero;    }
```

Память для zero существует всегда, а память под r выделяется при входе в zilch. В обоих случаях operator new не вызывается в complex::complex. Но во фрагменте типа complex *cp = new complex; значение this при входе в complex::complex равно null и вызывается operator new.

Другой случай, когда this - не null при входе в конструктор, - это инициализация базового класса и элементов, так как память для этого будет уже выделена конструктором производного или содержащегося класса. Например, при размещении одного из производных типов вершин абстрактного синтаксического дерева, вызвался конструктор производного класса, который выделяет память (при необходимости) и вызывает конструктор базового класса. Фрагмент Node *np = new int(5); вызывает конструктор для класса Int с пустым (NULL) значением this.

```
int::int(int v){ value = v; }
```

Сначала конструктор выделяет память для объекта Int, вызывая operator new и присваивая полученный адрес this. Затем неявно вызывается конструктор базового класса

Node:: Node. Так как this - не null при вызове конструктора базового класса, то operator new в Node:: Node не вызывается.

Тот же результат имеет место и для многоуровневой иерархии классов. Если записать

```
extern Node *x, *y;
Node *np = new Plus(x,y);
```

то будет вызван Plus::Plus(Node *l, Node *r) : Binop(l.r){} выделяющий память для объекта Plus и явно вызывающий конструктор базового класса Binop::Binop с инициализированным указателем this и указателями на левое и правое поддеревья. Конструктор Binop неявно вызывает конструктор Node. Так как память под объект Plus уже была выделена (в том числе и для частей Plus - Node и Binop), конструкторы Node и Binop не вызывают operator new.

Как и конструкторы, деструкторы имеют скрытый механизм вызова деструкторов элемента и базового класса и operator delete для освобождения памяти. Точно так же, как конструкторы классов выделяют память для элементов и базовых классов, деструктор класса освобождает память своих элементов и базовых классов. Вернемся к типу вершины унарный минус из иерархии вершин:

```
class Uminus : public Node {   Node *operand;
    public:    Uminus(Node *o) { operand = o; }
    ~Uminus(){ delete operand;}    };
```

Конструктор Uminus сначала выделяет память, потом вызывает конструктор базового класса Node. После возврата из конструктора базового класса, он выполняет свое тело, инициализируя operand. Деструктор действует, по сути, в обратной последовательности. Сначала он выполняет свое тело, удаляя operand (явным вызовом деструктора), потом вызывает деструктор базового класса Node. После возврата из деструктора базового класса Uminus::~Uminus вызывает operator delete для освобождения памяти класса. Аналогично вызову operator new, operator delete не вызывается деструкторами базового класса или элемента, а также если уничтожаемый объект статический или автоматический. Последовательность вызовов деструкторов базового класса или элемента и вызов operator delete вставляется перед каждым return в теле деструктора, включая неявный return в конце блока функции деструктора.

2.6.2 Операторы new и delete

Операторы new и delete - предопределенные библиотечные функции, управляющие свободной памятью или "кучей".

Для увеличения контроля над выделением памяти внешний указатель _new_handler может быть установлен на функцию, которую надо вызывать, если operator new потерпел неудачу. До тех пор, пока operator new не сможет выделить требуемую память или _new_handler не null, operator new повторно вызывает функцию, на которую ссылается_new_handler.

```
extern void (*_new_handler) ();
void give_up()  { error(" исчерпали память!"); exit(); }
void increase_limit()   { // запрашивает память у ОС
  extern long ulimit(int,long);
   ulimit(3,0);   new hendler~give_up;   }
f() {_new_handler = increase_limit;    int*lp-new int[1000000]; }
```

Теперь, если operator new не сможет выделить требуемый объем памяти, он будет вызывать increase_limit (неявно, через _new_handler) для увеличения объема памяти, доступного программе. Если это не удается или последующие вызовы operator new безрезультатны, give_up печатает сообщение и прекращает выполнение программы. Это очень гибкий механизм, и стандартные библиотечные операторы new и delete подходят для

большинства задач. Можно также перегрузить стандартные версии, предлагая собственные operator new и operator delete и использовать их вместо стандартных. Например, operator new не гарантирует, что выделяемая память иниализирована нулями, поэтому можно поступить следующим образом:

```
extern void * operator new(long sz) { // calloc возвращая обнуленную память
        extern char *calloc(unsigned,unsigned);
        return calloc(1,sz); }
extern void operator delete(void *p)  {
        extern void free(char *);    free((char *)p); }
```

Запомните, что установка _new_handler не будет влиять на поведение operator new (если это не указано явно), _new_handler - это просто часть реализации стандартного библиотечного operator new.

Если программа производит много выделений небольших фрагментов памяти и редко освобождает память сразу после выделения, можно разработать эффективную схему управления памятью, учитывающую такую модель использования:

```
externvoid(*new handler)() =0;
extern void * operator new(long nbytes)
    {    extern char *calloc(unsigned,unsigned);
        static const long BSIZ=4* 1024;
        static char ibuft[BSIZ];
        static char *start = ibuf;
        static char *end = ibuf + BSIZ;
        static const long ALIGN = sizeof(double);
        if ( nbytes &(ALIGN-1))   nbytes += ALIGN - nbytes % ALIGN;
        if (end start < bytes)
         {    long bufsiz =BSIZ > nbytes ? BSIZ: nbytes;
             while (!(start - calloc(1, bufsiz)))
             if  (_new_handler)_new_handler();
             else   return 0;
             end = start + bufsiz;    }
        start += nbytes;
        return start = nbytes;           }
```

Версия operator new сначала округляет запрашиваемый объем памяти для обеспечения выравнивания (следующего требования), потом проверяет, достаточно ли имеется свободного места в текущем буфере для выполнения запроса. Для удобства пусть два указателя определяют первое свободное место в текущем буфере и конец буфера. Если в текущем буфере достаточно места для удовлетворения запроса, указатель, определяющий следующее свободное место, обновляется и возвращается адрес выделенной памяти.

Если в текущем буфере недостаточно места для удовлетворения запроса, тогда у операционной системы запрашивается новый буфер минимального размера, но достаточно большой для выполнения запроса. Если запрос нового буфера неудачен, применяется стандартный механизм_new_handler, дающий пользователю возможность сделать некоторое восстановление.

Запрашивая память у операционной системы нечасто, большими блоками, версия operator new будет работать лучше, чем стандартная версия, для программ, выдающих частые запросы на небольшие объемы памяти. Так как начальный буфер размещен статически, запросов к операционной системе может вообще не поступить.

Большинство программы редко освобождают память, поэтому operator delete можно эффективно реализовать подставляемой "пустой" функцией.

```
inline void operator delete(void *) {}
```

Важным свойством, присущим представленным версиям operator new и operator delete

является то, что они имеют тот же интерфейс пользователя, что и стандартные new и delete. Частично это обязательно: operator new должен быть типа void* (long), а operator delete должен быть типа void (void*). Новая версия operator new сохраняет также действия по ошибке, то есть возврат NULL при неудаче. Эти программы можно инкапсулировать в файл, предохраняя пользователя от использования специфических особенностей реализации. Поддержка стандартного интерфейса и скрытие деталей реализации дает те же возможности, что и абстрактный тип данных (фактически, это и есть абстрактный тип данных), и можно менять реализацию управления памятью, не влияя на программы пользователя.

2.6.3 Управление памятью для массивов

Стандартная библиотека C++ определяет две функции, управляющие памятью для массивов классовых объектов:

```
void *_vec_new(void *, int, int, void *);
void _vec_delete(void *,int, int, void *, int);
```

Указанные функции используются не для всех размещений и удалений массивов, а только для тех, которые требуют инициализацию конструктором, или уничтожения. Так, int *ip = new int[3][4][5]; вызывает в конечном итоге operator new, запрашивая память для 60-ти элементов int , а struct Pair{ void *first, *second; }; Pair *list=new Pair[12]; требует пространства, достаточного для 12-ти Pair. Если все элементы массива требуют инициализации конструктором, как в

```
struct Pair2{         void *first, *second;
              Pair2() { first = second = 0; }      };
Pair2 *list2 = new Pair2[12];
```

то сначала вызывается _vec_new для выделения пространства под массив, а потом конструктор для каждого элемента.

Аналогично, _vec_delete вызывается, если тип элементов удаляемого массива имеет деструктор. В этом случае необходимо явно передавать размер массива:

```
struct Strlist    {  Strlist *nexf;    String str;
                  Strlist();         ~Strlist();     };
Strlist *slist = new Strlist[100];  //вызывает _vec_new
delete [100] silst;                  //вызывает _vec_delete
```

Если информация о размере пропущена, компилятор сгенерирует вызов operator delete для удаления одного Strlist. В случае удалений, не приводящих к вызову _vec_delete, информация о размере игнорируется:

```
int *ip=new int[1000000];
delete [10] ip;             // информация о размере игнорируется
delete [12] Ust2;           // информация о размере игнорируется
```

Как для operator new, так и для operator delete, можно предложить собственные версии _vec_new и _vec_delete. Аргументами _vec_new являются адрес "размещаемого" массива, число элементов в массиве, размер (в байтах) элемента массива и адрес инициализирующего конструктора. Первый аргумент используется аналогично this в конструкторе, так как _vec_new и _vec_delete используются и для инициализации и уничтожения статических и автоматических массивов.

```
static Strlist hashtab[5][2];
```

Первый аргумент _vec_new - адрес массива для статических и автоматических массивов и нуль в противном случае. Аргументы _vec_delete те же, что и для _vec_new, кроме того, что есть добавочный аргумент, сигнализирующий о необходимости освобождения памяти. Для статических и автоматических массивов флаг равен нулю и память не освобождается; для других - флаг равен единице и память освобождается.

Интерфейс _vec_new определяет его строго ограниченное использование. Так как

единственная информация, представляемая _vec_new - это адрес конструктора, то невозможно вызвать _vec_new с конструктором, имеющим аргументы (даже предопределенные). В существующих реализациях С++ нельзя вызывать _vec_new с конструктором для класса, имеющего виртуальные базовые классы.

```
complex cary[10];          //ошибка !
Monltored_network_device boards[3];   //ошибка !
```

В первом объявлении неявная инициализация конструктором использует предопределенные аргументы. Во втором объявлении некоторые из базовых классов виртуальны. Так как область допустимых инициализаторов для массивов классовых объектов этим ограничивается, имеется хороший, в общем-то, совет - избегать массивов из классовых объектов за исключением наиболее простых классовых типов.

2.6.4 Специфические new и delete для классов

Рассмотренные реализации new и delete были общего назначения. Как и стандартные библиотечные версии, их предполагалось использовать для всех типов во всех случаях. Как уже говорилось ранее, для значительного ускорения управления памятью может быть задействована специфическая информация о типе или реализации. Для использования специализированных сведений С++ предлагается возможность определения специфически-классовых версий operator new и operator delete. Например, класс complex имеет очень маленькую и простую реализацию, и, кроме того, не требует обнуления памяти, поэтому можно предложить для него специфически-классовые операторы new и delete, учитывающие конкретные детали реализации.

```
class complex { double re, im;
public:   void *operator new(long);
void operator delete(void *);   };
```

операторы вызываются при любом размещении или удалении объекта типа complex, независимо оттого, выполняется размещение и удаление в контексте класса complex или нет.

```
#include <complex.h>
main()    {complex *cp = new complex(12);
           int *ip = new int;
           *ip =12;
           complex cgross = (*cp) * (*ip);    }
```

Обратите внимание, что ср ссылается на память, выделенную complex::operator new, а ip ссылается на память, выделенную operator new. Кроме того, cgross - автоматический и не требует вызова для инициализации какого-либо operator new.

Управление памятью для complex реализовано путем размещения элементов статического массива. Освобожденные элементы содержатся в списке свободных. Реализация complex::operator new сначала просматривает список свободных в поисках предварительно размещенного и освобожденного элемента. Если такого нет, но есть элементы массива, которые еще не размещались, размещается один из них. При нехватке памяти operator new возвращает NULL:

```
static const int MAX = 100;
static class rep    {          static rep *free;
      static int nur used;
      union { double store[2];
               rep *next;   };
       friend complex;     }    mem[MAX];
void * complex::operator new(long)
     {      if (rep::free)     { rep *tmp = rep::free;
               rep::free = rep::free->next;
```

```
                return tmp->store;   }
         else if(rep::num used < MAX) return mem[rep::num_used++].store;
                else  return 0;    }
```

Реализация complex::operator delete как раз и добавляет освобожденный элемент массива в голову списка свободных элементов:

```
void  complex::operator delete(void *p)
{ ((rep *)p)->next = rep::free;      rep::free = (rep *)p; }
```

Помните, что complex::operator new не только проверяет значение аргумента, но всегда дает память достаточного размера для объекта типа complex. Неявно предполагается, что не будет классов, производных от класса complex, в противном случае может возникнуть неприятная ситуация.

```
class polnt3 : public complex    {   // точка в трехмерном пространстве
              double z;
              public:            };
point3 *p = new point3;
```

Так как point3 наследует new и delete вместе с другими элементами класса complex, вызов конструктора для point3 вызывает complex::operator new и возвращает память, достаточную для complex, но недостаточно большую для point3.

Можно продвинуться в разработке специфически классовых new и delete еще дальше и разработать схему распределения памяти, зависящую от свойств программы в целом.

Рассмотрим программу-калькулятор, разработанную ранее. Пусть эта программа строит только одно абстрактное синтаксическое дерево, вычисляет его и затем удаляет все дерево. Учитывая это, можно разработать более эффективное управление памятью:

```
classNode
 { protected: Node () {}
   void *operator new(long);
   void operator delete(void *);    };
```

Node::operator new и Node::.operator delete унаследованы классами, выведенными из Node, поэтому требуется или предоставить отдельный operator new для каждого производного класса, или сделать Node::operator new достаточно общим для работы со всеми типами вершин. Здесь реализовано последнее.

```
static const int SZ = 1000:
static struct buf   { buf *next;
                  char mem[SZ];     }
b.*bp = &b;
static char *memp = h.mem;
void * Node::operator new(long sz)
                 { if (memp+sz > &bp->mem[SZ])
                     if (bp->next) { bp = bp->next;
                        memp = bp->mem;   }
                    else  if (bp = bp->next = new buf)  memp = bp->mem;
                     else  return 0;
   char *r- memp;
   memp += sz;
   return r;         }
```

Вначале размещается статический буфер, размер которого предполагается достаточным для большинства запусков программы и по необходимости будут выделяться дополнительные буферы из свободной памяти. Помните, что использование new для выделения памяти под дополнительные буферы имеет в виду общий operator new, так же, как и для буфера, у которого тип не Node и не производный от него; operator delete определяет тот факт, что любое использование delete предназначено для удаления целого абстрактного синтаксического дерева. Здесь игнорируется аргумент и переустанавливается указатель управления памятью в его начальное значения. Необходимо помнить обо всех добавочных

буферах, которые, возможно, были размещены для предотвращения повторного размещения. Избавимся от деструкторов в Node и в производных от него классах, так как они излишни:

```
void Node::operator delete(void *)
{   bp=&b;     memp = b.mem;    }
```

Эта версия new и delete работает гораздо быстрее стандартной, и она тесно связана со способом использования абстрактных синтаксических деревьев в конкретной программе.

Как и для функциональных элементов, возможно присваивание this в конструкторе. Делая так, программист снимает задачу выделения памяти классовому объекту. Если есть явное присваивание this в теле конструктора, то неявное обращение к operator new не производится, и все явные и неявные инициализации базовых классов и элементов после каждого такого присваивания повторяются.

Присваивание this в конструкторе опасно. Рассмотрим выделение памяти в конструкторе элемента списка, рассмотренного в предыдущих главах. Для иллюстрации вынесем инициализации элементов из тела конструктора в список инициализации:

```
list::list(ETYPE e,list *lst) : el(e),link(lst)
{if  (!this) this = my_malloc(slzeof(list)); }
```

Фрагмент не будет работать для статических и автоматических объектов. Инициализации в списке инициализации элементов выполняются только после присваиваний this, но для статических и автоматических объектов присваивание не выполняется, так как this при входе в конструктор имеет непустое значение.

Даже если ожидается, что статические и автоматические объекты типа list создаваться не будут, проблемы остаются, так как list может служить базовым классом или элементом другого класса,

```
list2::list2(ETYPE e,list2 *ft,ist2 *bl) : list(e,fl),link(bl){}
```

В этом фрагменте есть две проблемы. Первая состоит в том, что так как значение this непустое для инициализации базового класса, list::el и list::link не будут инициализированы. Вторая - память для объекта list2, включая часть list, представляется operator new, а не my_alloc. Будет ли это проблемой или нет, зависит от программы, но предварительно имеются основания для реализации альтернативной процедуры выделения памяти для list.

Первую проблему можно уладить, вводя кажущиеся ненужным присваивания this в list::list, чтобы значение this на входе не влияло на инициализацию.

```
if (this)  this = this;
else  this = my_altoc(sizeof(list));
```

Это не только неестественный и располагающий к ошибкам путь; он также дважды вызывает вставку неявного инициализирующего кода в тело конструктора.

Аналогично можно получить контроль над освобождением памяти в деструкторах присваиванием значения this. Если значение this перед return пустое, то деструкторы элементов и базовых классов не вызываются и возвращение к operator delete не производится.

В общем случае предпочтительнее использовать специфически-классовые версии operator new и operator delete, чем присваивание this.

2.6.5 Оператор ->

Управление памятью связано не только с выделением и освобождением памяти, но и с доступом к ней. Программы часто должны содержать фрагменты, которые выполняют некоторые действия либо до, либо после доступа к памяти. Так как эти фрагменты не вставляются автоматически, их можно пропустить неумышленно или для "оптимизации".

```
class info     { void f();    //...     };
info *p; // ВСЕГДА проверяйте, что p - не NULL !
void process_info()    {  //p обычно не null..
```

```
P->f();    //... }
```

Разумная оптимизация в process_info ведет к катастрофе. Чтобы помочь в преодолении трудностей такого типа, C++ позволяет перегружать оператор ->, давая возможность создавать "умные" указатели, инкапсулирующие семантики проверки и доступа. Как operator () и operator [], operator -> должен быть функциональным элементом. Он должен быть объявлен без аргументов и возвращать либо классовый тип, для которого определен operator ->, либо указатель на классовый тип.

Имейте в виду, что хотя предопределенный оператор **->** бинарный, определяемый пользователем operator **->** объявляется как унарный. Идея состоит в том, что определенный пользователем operator -> вырабатывает, прямо или косвенно, указатель на объект классового типа. Этот указатель затем используется с предопределенным оператором ->для доступа к элементу класса.

```
class infoP
   { static info null info;    info *p;
     public:    info *operator >0 {return p ? p : &null_info; }
     infoP (info *ip) { p=ip;}
     infoP &operator=(infoP ip) { return *this = ip; }    };
```

это одно из решений предыдущей проблемы; infoP - это "умный" тип указателя, никогда не возвращающий пустое значение для использования в ссылке. Кроме того, он защищает любые операции над указателями, типа ++, + и т.п., которые могут привести к ошибочному адресу.

```
infoP p;
void process_info()   { p_.f(); // безопасно   //...    }
```

Выражение p->f сначала вызывает info P::operator -> для p, вырабатывающий значение типа info*, которое затем используется для доступа к элементу info::f: operator ->   можно определить так, что он будет возвращать классовый тип, определяющий operator ->.

```
class infowarn    {  infoP &p;
  public:
  infowarn (infoP &ip): p(ip){}
  infoP &operator ->()
     {warn("получаем информацию!"); return p; }       };
```

Объект infowarn при использовании ведет себя как объект infoP*,* за исключением того, что он печатает сообщение перед выполнением надежного доступа: infowarn wp = p;   //... wp->f();

Обращение wp->f сначала вызывает infowarn::oferator->, печатающий сообщение и возвращающий infoP. Затем вызывается infoP::opera1or ->, возвращающий info*, используемый для доступа к info::f.

2.6.6 Конструктор со ссылкой на собственный класс

Конструктор, который можно вызвать с аргументом его собственного классового типа, определяет, как инициализировать объект этого класса. Так как такой конструктор обычно пишется как воспринимающий единственный аргумент типа ссылки на класс, элементом которого он является, то эта концепция записывается как X(X&) (произносится "X от ссылки X"). Эта возможность уже использовалась. Тип String использует конструктор String(String&) для проверки, что когда бы ни копировалась строка (при инициализации и присваивании, соответственно), поддерживаются корректные счетчики обращений в соответствующих объектах String_rep. Определение конструктора X(X&) для класса констатирует, что операция копирования объектов этого классового типа должна внимательно контролироваться.

Как уже было сказано, инициализации возникают не только как часть объявления или в

списке инициализации элементов конструктора, но и в двух других контекстах: инициализация формальных аргументов функции фактических и инициализация возвращаемых функцией значений (в возвращаемых выражениях).

Рассмотрим инициализацию формального аргумента фактическим. В некоторых случаях для преобразования фактического аргумента в значение допустимого для формального аргумента типа вызывается конструктор или оператор преобразования. В других случаях, вызывающему необходимо создать временный объект, в котором создается значение перед копированием его в формальный аргумент:

```
void f(complex);
 f(12);
```

При преобразовании целого 12 в значение complex в вызывающем контексте создается временный объект complex, который инициализируется конструктором класса complex. Значение временного объекта копируется в формальный аргумент. Поэтому формальный аргумент f инициализируется в контексте, вызывающем f, а не в самом f. Вот другой пример, использующий типы вершин абстрактного синтаксического дерева.

```
int incr(int)  { return i.eval()+1;  }          //опасность!
```

Фрагмент проблематичен по нескольким причинам. Во-первых, так как тип возвращаемого выражения (int) не соответствует возвращаемому типу функции int, компилятор создает временный объект int, инициализирует сто конструктором int, и возвращает значение временного объекта. Класс int имеет также деструктор (унаследованный от класса Node), поэтому временный объект int уничтожается до возвращения incr. Аналогично, формальный аргумент i уничтожается до возвращения. Если вызов incr генерирует временный объект для инициализация формального аргумента, то и этот временный объект будет уничтожен в вызывающем контексте:

```
extern int val;
int x = incr(val); // создает временный.
```

Поэтому при работе с формальным аргументом происходят два вызова деструктора и только одно обращение к конструктору: вызов конструктора и деструктора для временного объекта в вызывающем контексте и (только) деструктора для формального аргумента в incr.

Фактически, это не создает проблемы для пользователей иерархии абстрактного синтаксического дерева, так как программы, на ней базирующиеся, не используют типы вершин для аргументов и возвращаемых значений, а используют только указатели и ссылки на типы вершин:

```
int* incr(int &i) { return new int(i.eval()+1);
```

В данной версии incr в теле функции не активизируют деструктор, это может быть серьезной проблемой для пользователей класса String, так как несбалансированная генерация обращений к конструктору и деструктору повредит счетчику обращений в объектах String_rep. По этой причине классовые типы, определяющие конструктор X(X&), ведут себя по-другому, нежели прочие типы при использовании в качестве аргументов и возвращаемых значений. Если класс X определяет X(X&), то формальные аргументы типа X используются в целях инициализации и деструкции так, как если бы они были типа X&, и как X во всех других контекстах. Поэтому для аргументов X, как и для X&, память для формального аргумента фактически выделяется в вызывающем контексте, а не в вызываемой функции.

Аналогично, функция, объявленная как возвращающая значение X, фактически инициализирует память для объекта X в вызывающем контексте вместо возврата объекта X. При переносе памяти для формальных аргументов в возвращаемых значений в вызывающий контекст деструкторы н вызываемой функции для этих объектов не активизируются и поддерживается однозначное соответствие между созданием и уничтожением:

```
String incr(String s)
{  return s +"1"; }
```

Для формального параметра s деструктор не активизируется и оператор возврата

инициализирует память в вызвавшем объекте. Объект инициализирует и уничтожает формальный аргумент и уничтожает возвращаемое значение, инициализированное return.

2.6.7 Семантика неявной копии

Мы уже видели, что для управления семантикой копирования для объектов класса X необходимо определять и элемент operator=(X&). Можно также использовать operator (X&,X&), не являющийся элементом, но этот метод неудовлетворителен по причинам, изложенным в примере со wiring. Объекты класса X могут служить, однако, элементами и базовыми классами других классовых типов и та же семантика копирования, что и для объектов X, не внедренных в другие объекты, должна применяться для элементов и базовых классов типа X:

```
class Text    { Text *next;      String phrase;
   public:       Text(Text &);
    Text &operator =(Text &);     };
```

Класс Text определяет семантику копирования с помощью конструктора X(X&) и элемента-оператора присваивания. Они реализованы так, что сохраняют семантику копирования элемента String:

```
Text ::Text(Text &t) : phrase(t.phrase)   {    //...    }
Text & Text::operator =(Text &t)    { phrase =t.phrase;     //...    }
```

В обоих случаях явно реализована корректная инициализация или присваивание элемента String.

Даже если классовый тип не определяет явно свою семантику копирования, непредопределенная семантика требуется, если семантика копирования определена для элемента или базового класса:

```
class Empl     { long ssn;      String name;      //...     };
```

Чтобы во всех случаях соблюдалась корректная семантика копирования, C++ при необходимости определяет ее неявно. Если какой-то элемент или базовый класс данного класса определяет X(X&) или элемент operator =(X&) и для этого класса семантика копирования не определена явно, то она определяется неявно компилятором. Для класса X определяется (неявно)

```
X::X(const X &);
const X &X::operator=(const X &);
```

Семантика этого фрагмента состоит в вызове соответствующего X(X&) или элемента operator=(X&) для каждого элемента или базового класса, который этого требует, и простом копировании оставшихся элементов класса. Поэтому определение класса Empl включает неявные определения

```
Empl::Empl(const Empi &);  и  const Empl &Empl::operator=(const Empi &);
```

Это гарантирует, что для элемента String из Empl реализуется надлежащая семантика копирования и фрагмент, вызывающий копирование объектов Empl, не влияет на корректность элемента name из String:

```
extern Empl a;
Empl b=a;// Empl::Empl
a=b;        // Empl::operator =
```

3 ПОДГОТОВКА И ВЫПОЛНЕНИЕ ПРОГРАММ НА КОМПЬЮТЕРЕ

Операционная система - совокупность программных средств, обеспечивающих управление аппаратной частью компьютера и прикладными программами, а также их

взаимодействие между собой и пользователем. С точки зрения пользователя, операционная система - это посредник между пользователем и компьютером, который позволяет пользователю, не зная подробностей архитектуры компьютера, системы команд процессора и других микросхем, требовать от компьютера выполнения желаемых действий.

Операционная система образует некоторую среду, обеспечивающую возможность выполнения прикладных программ, причем как правило, программа, разработанная для выполнения в одной операционной системе, не может выполняться в среде другой операционной системы.

К настоящему времени для персональных компьютеров фирмы IBM разработано множество различных операционных систем, которые можно разделить на три группы: операционные системы семейства DOS, семейство операционных систем Windows, и операционные системы семейства UNIX.

Разработка операционных систем семейства DOS вызвана появлением персональных компьютеров IBM XT и затем IBM 286. Более поздние версии этих операционных систем позволяли эксплуатировать их на компьютерах IBM 386 и IBM 486. Полностью реализовать все возможности этих компьютеров операционная система DOS не могла, и ей на смену была разработана операционная система Windows 3.0, предназначенная для эксплуатации на компьютерах IBM 386 и выше. Отличительными чертами этой системы являются наличие развитого графического интерфейса, широкое использование мыши и многозадачность. Современные компьюторы Pentium, Pentium II, и Pentium III, для реализации всех своих возможностей требуют использования операционных систем Windows 98, Windows NT, Windows 2000. Эти системы имеют удобный дружественный интерфейс пользователя, но несмотря на наличие средств поддержки сетевых технологий по сути остаются однопользовательскими.

Операционная система UNIX была создана в 1969-1970 гг. и с тех пор продолжает удерживать достаточно прочное место среди пользователей компьютеров. Вместе в появлением все новых, более современных компьютеров, появляются и все новые версии операционных систем этого семейства, предназначенные для работы на этих компьютерах. К преимуществам UNIX, по сравнению с другими операционными системами, следует отнести ее мобильность, т.е. переносимость на различные модели компьютеров (т.е. не только IBM, но например и Apple), а также то, что эта система изначально разрабатывалась как многопользовательская и многозадачная, т.е. она позволяет работать на одном компьютере одновременно нескольким пользователям (при наличии аппаратуры для нескольких рабочих мест), а одному пользователю можно одновременно работать с несколькими задачами.

С точки зрения программиста операционная система UNIX является наиболее приспособленной для программирования на языке СИ, поскольку язык СИ был разработан как средство для создания самой этой операционной системы. Большинство систем семейства UNIX понимает программу на СИ совершенно одинаково, чего нельзя сказать об операционных системах DOS и Windows. Именно поэтому в данной книге вопросы, связанные с процессами подготовки и выполнения программ на компьютере, рассматриваются на примере операционной системы UNIX.

Общая схема подготовки программы, существующая во многих операционных системах, включает этапы подготовки текста программы с помощью какого-либо текстового редактора, трансляцию подготовленного исходного модуля в объектный модуль, сборку загрузочного модуля из подготовленного объектного модуля и библиотечных объектных модулей и выполнение получаемого загрузочного модуля. Выполнении этих действий осуществляется различными командами операционной системы и сервисными программами, поэтому рассмотрим сначала некоторые основные команды операционной системы UNIX.

Поскольку существует множество различных операционных систем семейства UNIX, рассмотрим только основные команды, которые имеются в любой операционной системе семейства UNIX и которых вполне достаточно, чтобы подготовить и выполнить программу на СИ. Команды вводятся с клавиатуры, после набора команды следует нажать клавишу Enter.

3.1 НАЧАЛО И КОНЕЦ СЕАНСА

Время, в течение которого пользователь работает с системой UNIX, называется сеансом. Для начала сеанса необходимо, чтобы система была готова начать сеанс. Признаком готовности системы является выдача приглашения login:

Для входа в систему пользователь должен ввести свое имя, а затем пароль. Например:

login: serg

password: TGTU&ipu при наборе пароля он не отображается на экране дисплея.

После этого система выдает приглашение (prompter), сигнализирующее о готовности к вводу команд. Обычно в качестве приглашения используется символ %, но это необязательно, поскольку вид приглашения можно изменить при помощи специальной команды.

Попробуем ввести, например, команду date. В ответ система сообщит текущую дату и время и снова выдаст приглашение:

% date

Mon Sep 28 15:47:36 EDT 1998

%

Команда cal выдает календарь на текущий месяц.

Любая команда состоит из обязательного имени, вслед за которым могут следовать ключи и параметры. Ключи изменяют действия, выполняемые командой, а параметры служат для указания объектов, над которыми выполняются действия команды. Например, команда cal может вывести календарь на любой год, если этот год задать в качестве параметра, например, %cal 1998

Команда date может не только сообщать дату и время, но и устанавливать их. Для установки даты 1 октября 1998 г и времени 9 часов 15 минут параметром будет 9810010915: % date 9810010915

Заметим, что устанавливать дату и время, а также регистрировать новых пользователей и лишать их права работать в системе UNIX, может только администратор системы (суперпользователь), ему доступны все средства и возможности системы. Обычному пользователю доступны только команды общего назначения:

Команда с именем echo выводит список своих параметры на экран

%echo hello unix

hello unix

При завершении работы с системой следует ввести команду % exit.

ПРИ ЭТОМ СИСТЕМА ЗАВЕРШАЕТ СЕАНС И ВНОВЬ ВЫДАЕТ ПРИГЛАШЕНИЕ LOGIN ДЛЯ НАЧАЛА СЕАНСА.

Если пользователь желает лишь временно уступить рабочее место другому пользователю, он может позволить ему набрать команду su с его именем пользователя в качестве параметра:

% su anna

password: TGTU&ahp

После ввода команды su и пароля другой пользователь может работать с системой. После завершения его работы командой exit первый пользователь может продолжить свою

работу, причем состояние системы для первого пользователя будет таким же, каким оно было до выполнения команды su.

Для того, чтобы узнать, кто в данный момент работает с ситемой следует использовать команду % who . Эта команда выводит имена пользователей работающих в данный момент на каждом из рабочих мест компьютера.

3.2 ФАЙЛОВАЯ СИСТЕМА

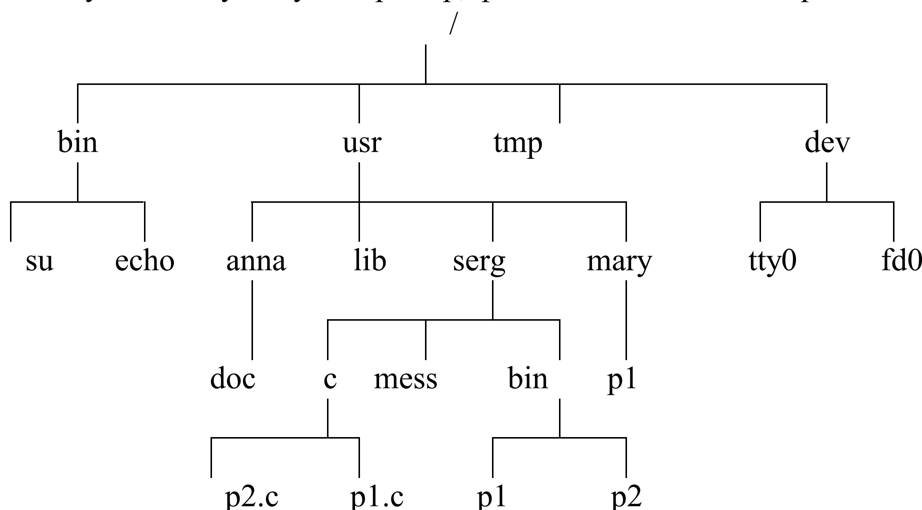
Вся информация, находящаяся на различных носителях (дисках, лентах и т.п.) хранится в виде некоторых именованных порций различного размера, называемых файлами. Операционная система UNIX не накладывает никаких ограничений ни на информацию, хранящуюся в файле, ни на ее внутреннюю структуру, ни на способы ее обработки. Она только обеспечивает учет имеющихся файлов и доступ к ним.

Каждый файл имеет свое собственное имя, под которым он известен системе. Для формирования имен файлов допускается использовать любые символы, но некоторые программы по именам файлов распознают их типы. При программировании на СИ принято хранить исходные тексты программ в файлах с окончанием имени .c , объектные модули в файлах с окончанием имени .o , имена с окончанием .h предназначены для заголовочных файлов, а с окончанием .a для программ на ассемблере.

Для удобства работы с файлами их объединяют в именованные группы, называемые каталогами или директориями. В свою очередь, каталоги тоже можно объединить в каталоги, а все полученные таким образом каталоги объединить в один каталог, который будет главным или корневым.

Для операционной системы каталог - это файл, содержащий информацию о месте нахождения других файлов и каталогов, находящихся в данном каталоге, причем способ хранения и обработки этой информации (в отличие от других файлов) известен операционной системе и используется ею для доступа к файлам и каталогам, записанным в данном каталоге.

Структуру файловой системы можно представить в виде дерева, начальным узлом которой является корневой каталог. Этот каталог является вместилищем всего, и поэтому имя у него отсутствует. Пример, файловой системы изображен на рисунке.



Как видно из рисунка, некоторые узлы этой системы (bin, dev и др.) не являются конечными, а содержат ветви идущие к другим узлам. Такие узлы соответствуют каталогам.

Концевые узлы могут быть файлами (su,echo,p1.c) или каталогами (anna, tmp), причем если это каталог, то обязательно пустой, т.е. не содержащий других каталогов и файлов.

Для того, чтобы найти какой-либо файл, необходимо указать путь по дереву до этого файла. Этот путь называется полным именем файла. Путь содержит имена каталогов, начиная с корневого, содержащих данный файл и имя файла. Элементы полного имени файла разделяются знаком /, при этом, поскольку корневой каталог имени не имеет, полное имя файла начинается со знака /. Например, полным именем файла p2 будет /usr/serg/bin/p2, а полным именем файла su будет /bin/su. Отдельные элементы полного имени одного файла или каталога могут совпадать с элементами полного имени другого файла или каталога, но полное имя каждого файла или каталога должно быть уникальным. Например, каталог с именем bin имеется в корневом каталоге и в каталоге serg, но их полные имена /bin и /usr/serg/bin различны. То же самое относится и к файлам /usr/mary/p1 и /usr/serg/bin/p1.

При входе в систему UNIX пользователь получает доступ к файлам и каталогам своего основного каталога (обычно каталоги пользователей хранятся в каталоге /usr), например пользователь Анна получит доступ к каталогу anna, пользователь Мария к каталогу mery, пользователь Сергей к каталогу serg. Этот каталог становится для пользователя текущим и для указания пути к нужному файлу за начало точки отсчета можно принять текущий каталог. В этом случае полное имя файла будет начинаться с символа отличного от /. Например, если текущим является каталог serg, то для обращения к файлу mess достаточно просто указать его имя, а для файла p1 можно использовать имя /bin/p1. Этот способ позволяет обращаться к файлам и каталогам, находящимся внутри текущего каталога, для обращения к файлам из соседних ветвей нужно указывать специальное имя .. являющееся обозначением имени родительского каталога по отношению к текущему. Например, родительским каталогом для каталога serg является каталог usr, и если serg является текущим, то к файлу p1 из каталога mery можно обратиться по имени ../mery/p1. Специальное имя .. можно использовать неоднократно, например, если текущий каталог serg, то к файлу su можно обратиться по имени ../../bin/su.

Каждый каталог (даже пустой) кроме каталога с именем .. содержит также каталог с именем . , этот каталог хранит информацию о самом себе и используется для обращения к файлам текущего каталога.

3.2.1 Права доступа к файлам и каталогам

Пользователь имеет право в любой момент сделать текущим любой из доступных ему каталогов.

В отношении каждого каталога и файла существуют специальные права доступа, определяющие, кто и что может делать с файлом или каталогом. Для каждого файла могут быть разрешены или запрещены следующие операции: чтение (т.е. исследование содержимого), запись (т.е. изменение содержимого) и выполнение (т.е. запуск как программы). Естественно, что кроме наличия прав на выполнение файл действительно должен быть выполняемой программой. Права доступа к каждому файлу формируются для трех категорий пользователей: владелец файла, член группы владельца, остальные пользователи. Например, владелец файла может разрешить членам группы, с которыми он работает совместно чтение своих файлов, но запретить внесение в них изменений, а другим пользователям можно запретить чтение и запись, но разрешить им выполнение некоторых своих программ.

Для выполнения какой либо программы имя соответствующего файла нужно набрать как команду. Например, для выполнения программы p2 из каталога /usr/serg/bin нужно набрать %/usr/serg/bin/p2 и если требуется указать параметры.

По существу команды операционной системы это программы созданные разработчиками операционной системы и хранящиеся в каталоге /bin. Естественно, что право на чтение и изменение этих программ есть только у суперпользователя, все остальные пользователи имеют право только на выполнение этих программ.

Права доступа к каталогам такие же как и к файлам, но имеют несколько другой смысл. Право на чтение обеспечивает возможность чтения каталога, т.е. можно узнать какие файлы и каталоги хранятся в данном каталоге. Право на запись дает возможность записи файлов в каталог и удаление их из каталога, причем удалять можно даже файлы защищенные от записи, так как при этом изменяется не файл, а информация о нем в каталоге. Право на выполнение каталога дает возможность обращения к файлам, содержащимся в каталоге. Заметим, что право на чтение каталога дает возможность только узнать о наличии файлов в каталоге, но не право обращения к ним, а право на выполнение каталога дает возможность обращения к файлам каталога, даже при отсутствии права на чтение каталога.

Такая организация прав доступа обеспечивает каждому пользователю доступ к своей части файловой системы, другие части системы для него остаются невидимыми. Причем только владелец файла имеет право решать какие файлы и каталоги сделать общедоступными, а какие недоступными для других пользователей.

Суперпользователь имеет особые права, ему доступна вся файловая система и он может читать изменять и выполнять любые файлы и каталоги.

3.2.2 Команды работы с файловой системой

Команды работы с файловой системой обеспечивают выполнение некоторых действий по обслуживанию файловой системы: копирование, удаление, переименование файлов и другие действия, не связанные с изменением их содержимого. Во многих этих командах в качестве параметров используются имена файлов и каталогов. Как уже говорилось, если имя файла начинается с символа /, то поиск файла осуществляется от корневого каталога, если имя начинается с любого другого символа, то поиск осуществляется от текущего каталога. Если для выполнения команды требуется некоторый каталог, а он не задан, то, как правило, подразумевается текущий каталог.

Для того, чтобы сделать текущим некоторый каталог используется команда % cd [имя каталога]. Она делает указанный каталог текущим, а если имя нового каталога не задано (квадратные скобки показывают, что наличие параметров необязательно), то текущим становится основной каталог пользователя.

Узнать имя текущего каталога можно с помощью команды %pwd. Команда не имеет параметров и выводит имя текущего каталога. Ее полезно использовать, когда Вы "заблудились" в лабиринтах файловой системы.

Команда ls позволяет выполнить просмотр содержимого некоторого каталога и имеет формат %ls [ключи] [список имен].

Список имен может состоять из имен каталогов или файлов. В первом случае выводится содержимое указанных каталогов, во втором выводится информация только об указанных файлах. Если имена не заданы, то выводится информация о файлах текущего каталога.

Ключи влияют на объем и порядок выводимой информации:

-l - выводить полную информацию о файлах;

-a - включать в список файлы с именами . и .. ;

-i - выводить номера дескрипторов файлов;

-r - вывод списка в обратном порядке;

-t - вывод списка, упорядоченного по времени создания;

-u - вывод списка, упорядоченного по времени последнего использования.

Ключи можно комбинировать, например, команда % ls -lat/usr/serg выдаст список имен файлов каталога /user/serg упорядоченный по времени создания, примерно, в следующем виде:

total 6
drwxr-xr-x 2 serg 512 NOV 2 14:17 .
drwxr-xr-x 2 root 512 NOV 2 14:17 ..
drwxr-xr-x 2 serg 512 NOV 2 14:27 bin
drwxr-xr-x 2 serg 512 NOV 3 09:31 c
-rw-r--r-- 2 serg 47 NOV 4 11:45 mess

В первой строке указано число блоков занятых файлами и каталогами данного каталога. В следующих строках представлена информация о содержании каталога. Каждая строка содержит среди прочего информацию о правах доступа (запись вида drwxr-xr-x), о владельце файла (serg), о дате создания файла. Первый символ строки говорит о том является ли данная запись записью о каталоге (d) или файле (-). Далее следуют права доступа состоящие из трех групп по три символа. Первая группа информирует о правах владельца файла, вторая - о правах члена группы владельца, третья - о правах других пользователей. Символы в группе могут следовать в порядке rwx, причем если соответствующее право (чтение - r, запись - w, выполнение - x) запрещено, то вместо буквы указано тире.

Для изменения прав доступа служит команда chmod. Она имеет два параметра %chmod права_доступа имя_файла.

Конструкция права_доступа может состоять из трех цифр, каждая из которых получена сложением чисел 4 для чтения, 2 для записи и 1 для выполнения, если соответствующее право разрешается и 0 если запрещается. Эти три цифры формируют права доступа для владельца, члена группы и всех остальных. Например, чтобы разрешить всем выполнение файла /usr/serg/bin/p1 , но запретить всем, кроме владельца, чтение и изменение этой программы, нужно выполнить команду %chmod 711 /usr/serg/bin/p1.

Если пользователь желает еще и защитить эту программу от стирания другими пользователями, он должен запретить всем, кроме себя, запись в каталог /usr/serg/bin командой %chmod 755 /usr/serg/bin.

Изменять режим доступа могут только владелец файла и суперпользователь. Владельцем файла считается пользователь создавший файл.

Для удобной работы с файлами следует размещать их в разных каталогах. Чтобы создать каталог, нужно использовать команду %mkdir имя_каталога. Эта команда имеет один обязательный параметр имя_каталога и создает каталог с указанным именем.

Для удаления каталога используется команда %rmdir имя_каталога. Она удаляет указаный каталог, но только в том случае, если в нем нет никаких файлов и каталогов, т.е. если удаляемый каталог пуст.

При работе с файлами часто бывает необходимо переписывать файлы из одного каталога в другой или создавать копии файлов. Для копирования файла используется команда %cp имя_оригинала имя_копии. Она позволяет создать копию файла в том же самом каталоге или любом другом. Например, команда %cp /usr/serg/c/p1.c /usr/serg/c/p17.c создаст копию файла p1.c с именем p17.c в каталоге /usr/serg/c, а команда %cp /usr/serg/c/p1.c /usr/anna/p1.c поместит копию файла p1.c в каталог /usr/anna.

Для переименования файла или каталога служит команда %mv старое_имя новое_имя. Она позволяет переименовать файл или каталог, который останется там же где и был, но получит другое имя. Такая операция и есть переименование. Если же файл или каталог переносится из одного каталога в другой, с изменением его имени или нет, то эта операция называется перенос. При переименовании изменяется только имя файла, а путь не

изменяется, при переносе обязательно изменяется путь, меняется ли при этом имя файла неважно. Например, команда %mv /usr/serg/c/p2.c /usr/anna/s.c осуществит перенос файла p2.c из каталога /usr/serg/c в каталог /usr/anna и при этом переименует его в s.c, т.е. эта операция переноса, а команда %mv /usr/serg/c/p2.c /usr/serg/c/p5.c осуществит переименование файл p2.c в файл p5.c, не перемещая его в другой каталог. Удаление файлов производится командой %rm имя_файла. Удалить файл может любой пользователь, если у него есть право на запись в каталог, в котором находится удаляемый файл. При этом неважно, есть ли у него права на запись этого файла или нет.

3.2.3 Команды работы с файлами

Команды работы с файлами позволяют выполнять вывод файлов на экран и на печать и некоторые другие действия. Большинство этих команд правильно работают только с текстовыми файлами, т.е. предполагается, что файл состоит из строк, каждая из которых заканчивается символом переброса каретки, а строки состоят из букв и некоторых других символов. Этим условиям удовлетворяют файлы, в которых хранятся тексты на человеческих языках или тексты программ на языках программирования. Выполняемые программы обычно содержат различные неотображаемые символы и понятие строка к таким файлам неприменимо.

Для вывода файлов на экран используется команда cat, она выводит содержимое всех указанных файлов. Например, команда %cat /usr/serg/c/p1.c /usr/serg/c/p2.c выведет на экран содержимое файлов p1.c и p2.c, при этом какие либо промежутки между файлами будут отсутствовать. Для вывода содержимого одного файла команда должна иметь один параметр. Например, %cat /usr/serg/c/p1.c .

Команда pr выполняет туже самую операцию, но содержимое файлов выводится отдельными страницами по 66 строк считая заголовок, где указан номер страницы и имя файла. Кроме того, команда pr может печатать в несколько колонок, для этого нужно использовать ключ указывающий число колонок. Например, команда %pr -2 /usr/serg/p1.c выполнит вывод файла p1.c на экран в две колонки.

Для печати файлов используется команда lpr. Она работает так же как и pr, но осуществляет вывод на принтер. Поскольку размер экрана ограничен, при выводе больших файлов на экране будут видны только несколько последних строк и нет смысла выводить такие файлы целиком. Для вывода частей файла служит команда tail. Если в команде указано только имя файла, то будет выведено 10 последних строк этого файла. Если задать ключ -n (здесь n - это произвольное число), то будет выведено n последних строк, при использовании ключа +n файл будет выведен начиная со строки n., т.е., при выполнении команды %tail -5 /usr/serg/c/p1.c будут выведены 5 последних строк файла p1.c, а команда %tail +5 /usr/serg/c/p1.c выведет этот файл, начиная с пятой строки.

Команда wc позволяет получить некоторую статистическую информацию о файлах. Она позволяет узнать число строк, слов и символов в указанных файлах, в каждом отдельно и во всех вместе. Задавая ключи -l, -w, -c (отдельно или в любой комбинации) можно узнать информацию только о числе строк, слов или символов. Например, команда %wc /usr/serg/c/p1.c выведет примерно следующую информацию 21 145 386 p1.c.

Это значит, что в файле p1.c 21 строка, 145 слов и 386 символов. То же самое будет если использовать ключ -lwc, если же использовать ключ –w: %wc -w /usr/serg/c/p1.c, то будет выводиться только число слов.

Для сравнения содержимого двух разных файлов служат команды cmp и diff. Они имеют по два обязательных параметра - имена сравниваемых файлов. Команда cmp это быстродействующая команда, которая сравнивает любые (не только текстовые) файлы и

находит первое место, где файлы отличаются. Например, команда %cmp /usr/serg/c/p1.c usr/serg/c/p2.c сообщит p1.c p2.c differ: char 68, line 4, т.е. различие в файлах обнаружено в 68 символе, 4 строке.

Команда diff сравнивает только текстовые файлы и сообщает обо всех строках, которые изменены, добавлены или удалены. Ее рекомендуется использовать для файлов, которые незначительно отличаются друг от друга.

Для поиска в файлах определенной последовательности символов служит команда grep. Первый параметр этой команды является образцом для поиска, остальные параметры содержат имена файлов, в которых осуществляется поиск. Например, команда %grep int /usr/serg/c/p1.c /usr/serg/c/p2.c выведет все строки файлов p1.c и p2.c, в которых встречается последовательность символов int как отдельное слово или часть некоторого другого слова. Ключ -v позволит вывести только те строки, в которых указанная последовательность символов не встретилась.

Команда sort выводит строки файла в отсортированном порядке. Она имеет разные ключи, влияющие на порядок сортировки, при отсутствии ключей используется алфавитный порядок. Наиболее часто используются ключи:

-f - не учитывать различие прописных и строчных букв;
-n - числовой порядок;
-r - обратный порядок;
+n - (n - число) сортировать с поля n+1;
-u - удалять повторяющиеся строки;
-d - игнорировать любые символы кроме букв, цифр и пробела.

Все эти ключи можно комбинировать, например, команда %sort -fr /usr/serg/c/p1.c выведет строки файла p1.c в обратном алфавитном порядке без различия прописных и строчных букв. Кроме того, сортировку можно применить к результату сортировки. Для этого нужно использовать несколько комбинаций ключей. Например, команда %sort -f +0 -u /usr/serg/c/p1.c сначала сортирует строки не различая прописных и строчных букв (ключ -f), далее одинаковые строки после первой сортировки сортируются обычным порядком (ключ +0), а затем (ключ -u) удаляются все кроме одной из повторяющихся строк.

3.2.4 Шаблоны имен файлов

Обычно имена файлов, содержащих информацию одинакового типа, имеют похожие имена и часто эти файлы обрабатываются одной и той же командой. Например, для вывода на печать всех программ на СИ имена соответствующих файлов нужно перечислить в команде lpr. Другим способом выполнения этой операции является задание специального шаблона имени файлов, которому соответствовали бы имена всех файлов, содержащих программы на СИ. По общепринятому соглашению эти имена оканчиваются символами .c .

Для формирования шаблонов можно использовать любые символы и символы имеющие специальный смысл. Символ * в имени файла обозначает любое число любых символов, а символ ? обозначает один любой символ. Например, шаблон *.c обозначает имена всех файлов оканчивающихся .c, шаблон part.* обозначает имена всех файлов начинающихся с part. , а шаблон p?.c описывает имена файлов начинающихся с символа p и заканчивающихся символами .c, между которыми имеется один какой либо символ. Поэтому для печати СИ программ можно использовать команду %lpr *.c.

Шаблон [...] задает любой символ из перечисленных в квадратных скобках, т.е. %lpr part[1247] обозначает имена файлов part1, part2, part4, part7. С помощью тире можно указать начало и конец некоторого упорядоченного списка символов, т.е. команда %lpr part[1-7a-f] осуществит печать файлов имена которых оканчиваются цифрой с 1 до 7 или буквой с a до f.

Отметим, что шаблоны можно применять к именам существующих файлов и нельзя создать файл с помощью шаблона. Поэтому, например, команда создания копий файлов с похожими именами %cp *.c *.cpp не работает, поскольку второй шаблон не задает имен существующих файлов.

3.2.5 Переключение ввода-вывода

Большинство рассмотренных выше команд осуществляют вывод результатов своей работы на экран, а входные данные выбираются из указанных в качестве параметров файлов. При отсутствии соответствующего параметра входные данные должны поступать с клавиатуры. Например, команда sort без параметров будет сортировать строки, вводимые с клавиатуры, для завершения ввода данных с клавиатуры необходимо одновременно нажать клавиши Ctrl и D .

%sort

zx □
abc □ входные данные
qwerty □

Ctrl+D символ конца файла

abc □
qwerty □результаты работы
zx □

%

Таким образом, почти каждая из команд организована так, что имеет некоторый входной поток информации и выходной поток. Операционная система позволяет переопределить входной и выходной поток любой команды, направив его в некоторый файл. Для переключения вы водного потока используется символ > или >> вслед, за которым следует имя файла, в который будут помещены результаты работы команды. Причем если используется символ >, то создается новый файл, а при использовании >> выходной поток добавляется к содержимому указанного файла. Например, команда %ls -la >listdir создаст файл listdir и поместит в него оглавление текущего каталога. Команда %cat p1.c p2.c > all.c создаст файл all.c в который последовательно поместит тексты файлов p1.c и p2.c, а команда %cat p1.c >>p2.c добавит к содержимому файла p2.c содержимое файла p1.c.

Аналогично символ < позволяет переключить входной поток, так, что он будет браться из указанного файла. Например, команда %sort < f1 будет сортировать файл f1 и выводить его на экран, точно также как и команда %sort f1.

Просто в этих командах разным способом задан входной поток. В первом случае входной поток считается стандартным, но переключенным в файл f1, а во втором случае входной поток задается с помощью параметра команды.

Переключение ввода-вывода позволяет получать эффекты, недостижимые другим способом. Например, можно вычислить число файлов в некотором каталоге

%ls >dir

%wc -l <dir

ХОТЯ В ЭТО ЧИСЛО ВОЙДЕТ И ПРОМЕЖУТОЧНЫЙ ФАЙЛ DIR. ЧТОБЫ ЭТОТ ФАЙЛ НЕ ВХОДИЛ В ЧИСЛО ФАЙЛОВ КАТАЛОГА, ЕГО МОЖНО РАЗМЕСТИТЬ В КАКОМ ЛИБО ДРУГОМ КАТАЛОГЕ. ДЛЯ ВЫВОДА СПИСКА ФАЙЛОВ ИЗМЕНЯВШИХСЯ В АВГУСТЕ СЛЕДУЕТ ВЫПОЛНИТЬ ПОСЛЕДОВАТЕЛЬНОСТЬ КОМАНД

%ls -l > temp
%grep AUG <temp >temp2
%wc -l <temp2

В данном примере в команде grep переключены и входной и выходной потоки, что вполне допустимо.

3.3 ПРОГРАММНЫЕ КАНАЛЫ И КОНВЕЕРЫ

Последние примеры, приведенные в предыдущем разделе, основаны на том приеме, что выходной поток одной команды считается входным для следующей команды, для чего используются временные файлы. Сам временный файл после его использования не нужен и его требуется удалять. Для того, чтобы не следить за созданием и удалением временных файлов, можно использовать программные каналы - средство связи команд, при котором выходной поток первой команды считается входным потоком второй команды, без создания временных файлов.

Соединение двух и более команд программным каналом называется конвеером. Для связи команд в конвеер используется символ | помещаемый между командами. Так для подсчета числа файлов в каталоге можно использовать конвеер %ls -l | wc -l при этом "лишний" временный файл не будет включен в число существующих файлов.

Для подсчета числа файлов измененных в августе можно использовать конвеер %ls -l | grep AUG | wc -l, а конвеер %ls -l *.c | grep AUG | wc -l будет вычислять число программ на СИ, измененных в августе.

До сих пор были использованы только команды операционной системы. Но по сути эти команды есть не что иное как обыкновенные программы и все ранее сказанное можно отнести к любой программе. Например, для вызова программы p1 из каталога /usr/serg/bin нужно указать ее имя в командной строке, а для того, чтобы направить выходной поток этой программы в файл outp1, нужно использовать символ > и имя файла:

%/usr/serg/bin/p1 > /usr/serg/bin/outp1

Можно организовать конвеер из программ p1 и p2:

%/usr/serg/bin/p1 | /usr/serg/bin/p2

3.3.1 Последовательное и одновременное выполнение нескольких команд

Система UNIX позволяет запустить на выполнение сразу несколько команд или программ, которые будут выполняться последовательно или одновременно. Для запуска некоторой последовательности команд их следует набрать в нужной последовательности, разделяя точкой с запятой. Например, если ввести строку %ls ; /usr/serg/bin/p1 ; ls, то будут последовательно выполнены три программы, и только после их выполнения система опять выдаст приглашение.

Программу или последовательность программ можно запустить таким образом, что система не будет дожидаться окончания выполнения этой программы, а продолжит диалог с пользователем. Для этого в конце командной строки следует набрать символ &. Например, для подсчета числа слов в файлах каталога /usr/serg/doc можно набрать команду

% wc /usr/serg/doc/txt* > /usr/serg/doc/stat &
1234 номер процесса ,

А ПОКА ОНА ВЫПОЛНЯЕТСЯ, МОЖНО, НАПРИМЕР, КОПИРОВАТЬ ЭТИ ФАЙЛЫ В КАТАЛОГ ANNA ИЛИ ВЫПОЛНЯТЬ ЛЮБУЮ ДРУГУЮ РАБОТУ. ПЕРЕНАПРАВЛЕНИЕ

ВЫВОДА ПРЕДОТВРАТИТ СМЕШИВАНИЕ ИНФОРМАЦИИ РАЗНЫХ ПРОГРАММ. ДЛЯ ВЫПОЛНЕНИЯ ЭТОЙ КОМАНДЫ СИСТЕМА СОЗДАСТ ОТДЕЛЬНЫЙ ПРОЦЕСС, ПРИСВОИВ ЕМУ НЕКОТОРЫЙ НОМЕР, И СООБЩИТ ЭТОТ НОМЕР. ЕСЛИ И СЛЕДУЮЩАЯ ПРОГРАММА БУДЕТ ЗАПУЩЕНА С СИМВОЛОМ &, ТО СОЗДАСТСЯ ЕЩЕ ОДИН НОВЫЙ ПРОЦЕСС, КОТОРЫЙ БУДЕТ ВЫПОЛНЯТЬСЯ ОДНОВРЕМЕННО С ПЕРВЫМ. ЕСЛИ ЗАПУСТИТЬ КАК ОТДЕЛЬНЫЙ ПРОЦЕСС КОНВЕЕР, ТО ВСЕ ПРОГРАММЫ ЭТОГО КОНВЕЕРА БУДУТ ВЫПОЛНЯТЬСЯ ОДНОВРЕМЕННО, НО НОМЕР ПРОЦЕССА БУДЕТ ОТНОСИТСЯ К ПОСЛЕДНЕЙ ПРОГРАММЕ.

Используя номер процесса, можно повлиять на его выполнение. Команда %ps -ag сообщит о всех выполняющихся процессах. Команда %wait ожидает завершения всех запущенных процессов. Если она не возвращается, значит еще не все процессы завершены, и если одна из программ содержит ошибки и никогда не завершится, то дальнейший диалог с системой будет невозможен.

Для остановки некоторого процесса можно использовать команду kill с номером снимаемого процесса, например, %kill 1234. Если в качестве номера указать 0, то будут уничтожены все запущенные процессы.

Следует отметить, что для диалога пользователя с системой тоже существует отдельный процесс, но повлиять на него команда kill не может. Для запуска этого процесса используются команды входа в систему login или us, а для снятия процесса команда выхода из системы exit. Причем команда exit автоматически уничтожает все созданные процессы созданные пользователем и еще не завершенные.

Специальная команда -% nohup программа &- позволяет запустить на выполнение некоторую программу в виде от дельного процесса, который будет продолжаться даже если пользователь завершит сеанс и выйдет из системы. Другая специальная команда -%at время любые команды и программы Ctrl+d- позволяет запустить некоторый процесс в назначенное время, даже если в это время пользователь не будет работать на компьютере. Эта команда имеет один параметр - время, которое может быть задано исходя из 24-х часового или 12-ти часового циклов. Например, время 21час 36 минут можно задать в виде 2136 или 936pm, а время 2 часа 18 минут в виде 218 или 218am.

3.3.2 Файлы устройств

К особенностям операционной системы UNIX относится своеобразная форма работы с внешними устройствами: дисками, принтерами, модемами и др. Все устройства ввода-вывода информации имеющиеся в компьютере и даже некоторые области оперативной памяти считаются частью файловой системы и как специальные файлы устройств хранятся в каталоге dev. При обращении к файлам устройств операционная система использует специальные программы, обеспечивающие работу с соответствующими устройствами, и поэтому пользователю нет необходимости знать, как работает конкретное устройство, он должен только соблюдать ограничения накладываемые возможностями устройства. Например, не имеют смысла операции чтения из принтера или запись в мышь.

Если использовать команду ls для файлов каталога /dev, то она выведет, примерно, следующее:

```
%ls -l /dev
crw--w--w- 1 root 0, 0 Sep 27 23:09 console
crw-r--r-- 1 root 3, 0 Aug 17 1998 mem
brw-rw-rw- 1 root 1,64 May 1 1997 rp00
crw-rw-rw- 1 root 3, 2 Nov 2 11:12 null
c-w--w--w- 1 root 1, 1 Nov 2 11:12 gp
```

crw-rw-rw- 1 root 2, 1 Nov 1 11:12 tty
c-w--w--w- 1 root 1, 1 Nov 1 11:12 print
brw-rw-rw- 1 root 1, 2 Nov 1 11:12 fd0
brw-rw-rw- 1 root 1, 2 Nov 1 11:12 wd01

Важным отличием списка файлов устройств от списка файлов является то, что первый символ в каждой строке характеризует тип устройства. Буква c говорит о том, что устройство символьное, а буква b - о том, что устройство блочное. Символьные устройства принимают и передают информацию как некоторую последовательность символов, а блочные устройства производят обмен информации порциями (блоками) определенного размера. К блочным устройствам относятся магнитные диски и ленты, остальные устройства: принтеры, линии сетевой связи, терминалы, графопостроители и т.п. - относятся к символьным.

Файлы устройств можно использовать как обыкновенные файлы. Например, команда %cat /user/serg/doc/*.* > /dev/gp выведет содержимое файлов каталога user/serg/doc на графопостроитель, а команда %cat /user/serg/doc/*.* > /dev/print выведет эти файлы на принтер.

Большинство файлов хранятся на жестких или гибких магнитных дисках для работы, с которыми предназначены накопители на жестких и гибких дисках и, таким образом, обращение к любому файлу каталога /user на самом деле есть обращение к накопителю на жестком магнитном диске или к файлу устойсва /dev/wd01. Это вызвано тем, что файл устройства /dev/wd01 содержит информацию представляющую собой некоторую файловую систему из различных каталогов и файлов. Принципиально имеется возможность получить доступ к некоторому файлу на диске, используя файл устройства /dev/wd01, но для этого нужно знать правила организации файлов на диске и использовать команды управления дисковым накопителем. Гораздо проще использовать возможности операционной системы, которая позволяет подключить к некоторому каталогу корневой файловой системы некоторое дисковое устройство с расположенной на нем отдельной файловой системой, которая при этом становится частью общей файловой системы. В частности, файловая система пользователей, расположенная на дисковом устройстве /dev/wd01, подключается к каталогу /user во время выполнения входа в систему.

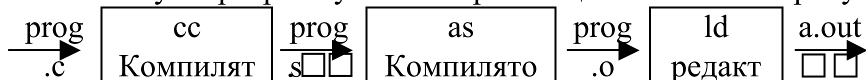
Пользователь по своему усмотрению может подключать к файловой системе и отключать от нее файловые системы, расположенные на различных дисковых устройствах. Для подключения файловой системы служит команда mount, параметрами которой являются имя файла устройства и имя каталога. Например, для подключения файловой системы на дискете как каталога flop следует выполнить команду %mount /dev/fd0 /flop после чего можно работать с файлами находящимися на дискете используя каталог /flop. После завершения работы с дискетой нужно отключить ее от каталога /flop командой %umount /flop.

Система UNIX позволяет работать с дисками и дискетами, содержащими файлы записанные по правилам операционной системы DOS, для этого следует подключать файловую систему командой mount_msdos. Например, %mount_msdos /dev/fd01 /usr/dos.

Отключается такая система обычным образом командой umount.

3.4 ПОДГОТОВКА И ВЫПОЛНЕНИЕ ПРОГРАММ В СРЕДЕ UNIX

Особенностью системы UNIX является то, что при трансляции программ они вначале транслируются на язык ассемблер, а затем преобразуются в объектный модуль и далее в исполняемую программу. Схема трансляции показана на рисунке.



Имена с расширением .c используются для файлов, содержащих программы на СИ, имена с расширением .s - для программ на ассемблере, с расширением .o - для объектных модулей, a - исполняемая программа, если не задано другого, имеет стандартное имя a.out.

Компилятор СИ за один вызов может выполнить всю цепочку преобразований программы на СИ в исполняемый модуль, автоматически вызывая компилятор ассемблера и редактор связей, либо некоторую часть этой цепочки. Компилятор СИ вызывается командой cc, которая может иметь следующие параметры:

%cc [ключи] [исходные_модули] [ключи_компоновщика]

[объектные_модули] [библиотеки]

Каждый из параметров не является обязательным, но хотя бы один из параметров исходные_модули или объектные модули должен присутствовать.

Ключи начинаются со знака - и задают режимы работы компилятора, после некоторых ключей могут задаваться дополнительные параметры. Наиболее употребительны следующие ключи:

-p - выполнить только препроцессорную обработку;

-S - транслировать в программу на ассемблере;

-c - транслировать в объектный модуль;

-o имя_исполняемого_модуля - задает произвольное (отличное от a.out) имя исполняемого модуля.

Ключи компоновщика задают режими работы компоновщика, наиболее часто используется ключ -l задающий имена системных библиотек объектных модулей. Системные библиотеки хранятся в каталоге /lib или /usr/lib а их имена начинаются с lib и заканчиваются .a. Например, стандартная библиотека СИ называется libc.a (эта библиотека используется всегда и ее указывать не требуется), библиотека математических функций хранится в файле libm.a. При использовании ключа -l нужно указать оригинальную часть имени системной библиотеки, т.е. для использования libm.a нужно указать ключ -lm.

Исходные_модули это имена одного или нескольких файлов с окончанием .c содержащих тексты программ на СИ.

Объектные_модули - это имена одного или нескольких файлов с расширением .o содержащих объектные модули, которые будут использованы для получения исполняемой программы.

Библиотеки - это имена одной или нескольких личных библиотек с расширением .a, в которых хранятся объектные модули нужные для получения исполняемой программы.

Если программа состоит только из одного исходного модуля (например, prog.c), то для ее компиляции и получения исполняемого модуля при вызове компилятора достаточно указать только имя программы %cc prog.c.

При этом имя исполняемого модуля программы будет a.out. Если в исходной программе используются математические функции, то необходимо задавать ключ компоновщика %cc prog.c -lm.

Для того, чтобы указать нужное имя исполняемого модуля, необходимо использовать ключ -o, после которого через пробел записывается имя исполняемого модуля %cc -o prog prog.c -lm.

Обычно большие программы состоят из нескольких отдельных модулей, и при отладке таких программ не всегда требуется заново компилировать все ее части.

Пусть, например, программа состоит из трех исходных модулей prog1.c, prog2.c, prog3.c, для получения исполняемого модуля их все нужно компилировать. Это можно сделать, указав их всех при вызове компилятора: %cc prog1.c prog.c prog3.c.

При внесении изменений в один из этих модулей необходимо вновь выполнить эту команду. При большом числе модулей такой подход неоправдан, удобнее было бы получить из исходных модулей объектные, а затем собирать из них исполняемую программу.

Для получения объектных модулей нужно их всех компилировать с ключом -c

%cc -c prog1.c

%cc -c prog2.c

%cc -c prog3.c

или с использованием маски %cc -c prog?.c, а потом собрать их в одну программу %cc prog?.o

Далее при внесении исправлений в один из исходных модулей, например prog2.c нужно транслировать заново только этот модуль, а другие подключать в виде объектных %cc prog2.c prog1.o prog3.o.

Для более удобной работы с большими программами следует использовать личные библиотеки объектных модулей, в которые надо помещать последние версии всех объектных модулей, а при вызове компилятора указывать только имя главного модуля (т.е. модуля содержащего функцию main) и имя библиотеки.

Для создания и обслуживания библиотек служит программа ar архиватор. Эта программа имеет следующие параметры:

%ar ключи имя_архива имена_файлов

Параметр имя_архива задает имя файла библиотеки, а параметр имена_файлов задает имена файлов содержащих объектные модули, которые будут помещены или удалены из библиотеки.

Ключи определяют режимы работы архиватора:

-d - исключить указанные файлы из архива;

-r - заменить указанные файлы в архиве, а если в архиве их нет, то добавить;

-u - этот ключ используется только с ключом r и вызывает замену только тех файлов, которые имеют более поздние даты модификаций, чем файлы в архиве;

-a и -b - эти ключи требуют указания вслед за ключами дополнительного параметра и определяют место указанного файла в архиве относительно файла, заданного дополнительным параметром. Ключ -b показывает, что новый файл нужно поместить до существующего файла, а ключ -a после него. Если ни один из ключей не задан, то новые файлы помещаются в конец архива. Порядок файлов в архиве важен при формировании исполняемого модуля, если библиотечный модуль содержит ссылки на другие модули этой же библиотеки, то он должен располагаться до вызываемого, так как компоновщик просматривает библиотечный файл только один раз;

-x - извлечь указанные файлы из архива (если имена файлов не заданы, то извлекаются все файлы), архив при этом не изменяется;

-t - вывести оглавление архива, если указаны имена файлов, то выводится информация только об этих файлах;

-v - вывести пояснительные сообщения.

Для создания библиотеки private.a и помещения в нее объектных модулей prog1.o prog2.o prog3.o следует выполнить программу

%ar -rv private.a prog?.o

Чтобы не заботиться о порядке размещения файлов в библиотеке, можно использовать программу ranlib, которая создаст внутри библиотеки оглавление, что позволяет компоновщику обращаться к файлам библиотеки в произвольном порядке %ranlib private.a.

Для получения исполняемого модуля следует вызвать компилятор %cc prog1.o private.a .

Далее, при внесении исправлений в один из модулей, например, prog2.c, его нужно откомпилировать и поместить в библиотеку

%cc -c prog2.c

%ar -r private.a prog2.o ,

а затем создать новую версию исполняемого модуля %cc prog1.o private.a.

4 МЕТОДЫ ЛИНЕЙНОГО ПРОГРАММИРОВАНИЯ

4.1 ПОСТАНОВКА ЗАДАЧИ ЛИНЕЙНОГО ПРОГРАММИРОВАНИЯ

Задача линейного программирования часто возникает на практике, особенно в области управления и планирования, когда ресурсы (все или некоторые) ограничены. Задача формулируется следующим образом:

Требуется найти экстремум линейной формы $Z = \sum_{j=1}^{n} c_j x_j$ *при условиях* $AX = b$, $b \geq 0$, $X \geq 0$, $x_j \leq \alpha_j$, $j \in J$ *и указать вектор X, при котором этот экстремум реализуется.*

Здесь A - матрица из m строк и n столбцов; $b(b_1, b_2, ..., b_m)$ - вектор свободных членов; c_j - коэффициенты линейной формы ($j = 1, 2, ..., n$); J - некоторое непустое подмножество множества $\{1, 2, ..., n\}$; α_j - верхние границы переменных x_j ($j \in J$).

Эту задачу можно решить обычным симплекс-методом, введя n дополнительных переменных $y_1, y_2, ..., y_n$, причем $y_j = \alpha_j - x_j$, и добавив условие $Y(y_1, y_2, ..., y_n) \geq 0$. Но тогда мы должны будем иметь дело с $2n$ переменными и $n + m$ условиями - равенствами, а следовательно, и с векторами условий размерностью $m + n$. Это существенно увеличивает трудоемкость каждой итерации и невыгодно для большинства практических задач, в которых n и m достаточно велики. В частности, может получиться задача, размеры которой будут превышать возможности машины.

Гораздо удобнее провести обобщение симплексного метода для задач с верхними ограничениями на переменные.

Здесь предлагаются три алгоритма и к ним следующие рекомендации:

Первый алгоритм - простой симплекс-метод, его удобно применять в том случае, если матрица A имеет более половины ненулевых элементов.

Второй алгоритм использует модифицированный симплекс-метод и компактное хранение информации. Поэтому при решении задач с большими m и n и матрицей, заполненной менее чем на 50 %, он дает существенную экономию машинного времени и памяти.

Третий алгоритм предназначен для тех задач, в которых требуется повышенная точность решения. В этом случае на каждой итерации используются только исходные

данные и ортогональные преобразования. Информация хранится в компактном виде. Недостатком третьего алгоритма является его непригодность для задач с плохо обусловленной матрицей А. Кроме того, он считает дольше первого или второго алгоритмов.

4.2 ПРОСТОЙ СИМПЛЕКС-МЕТОД
(*первый алгоритм*)

Здесь матрица условий преобразуется на каждой итерации с помощью метода исключения Гаусса.

Исходный опорный план ($n + m$-мерный):

$$X^0 := (0, 0, ..., 0, b_1, b_2, ..., b_m).$$

Исходный вектор номеров базисных переменных

$$N^0 = (j_1, j_2, ..., j_m) := (n + 1, n + 2, ..., n + m).$$

Рассмотрим r-ую итерацию.

Этап 1 - нахождение базисного допустимого решения.

1 Выбираем номер k среди номеров небазисных переменных из условия

$$a^r_{m+2,k} = \min a^r_{m+2,i}, \quad i \in J_r,$$

где $\bar{a}^r_{m+2,i} = \begin{cases} a^r_{m+2,i} & \text{при} \quad x_i^{r-1} = 0; \\ -a^r_{m+2,i} & \text{при} \quad x_i^{r-1} = \alpha_i, \end{cases}$ $\quad J_r = \{i/i \notin N^{r-1}, i \le n\}.$

2 Если $a^r_{m+2,k} \ge 0$, то переходим к п. 12, иначе выполняем п. 3.

3 Если $\alpha_k < 10^{15}$, то переходим к п. 6, иначе выполняем п. 4.

4 Если для всех i ($i = 1, 2, ..., m$) $a^r_{ik} < 0$, то выполняем п. 5, иначе переходим к п. 6.

5 Если $\alpha_{Jr} < 10^{15}$ хотя бы для одного i $\left(i = \overline{1,m}\right)$, то выполняем п. 6, иначе конец (линейная форма неограничена).

6 Вычисляем è

при $x_k^{r-1} = 0$: $\quad \text{è}^r = \min \begin{cases} \text{á}_k; \\ x_{Ji}^{r-1}/a^r_{ik}, & \left(a^r_{ik} > 0\right); \\ \left(x_{Ji}^{r-1} - \text{á}_{Ji}\right)/a^r_{ik}, & \left(a^r_{ik} < 0\right), \end{cases} \quad i = 1, 2, ..., m.$

при $x_k^{r-1} = \alpha_k$: $-\text{è}^r = \min \begin{cases} \text{á}_k; \\ x_{Ji}^{r-1}/\left(-a^r_{ik}\right), & \left(a^r_{ik} > 0\right); \\ \left(x_{Ji}^{r-1} - \text{á}_{Ji}\right)/\left(-a^r_{ik}\right), & \left(a^r_{ik} < 0\right). \end{cases}$

7 Если $\text{è}^r = \pm\,\alpha_k$, то $l^r := 0$, иначе l^r равно тому i, при котором реализуется значение è^r.

8 Пересчитываем план и значения линейных форм:

$$\left.\begin{array}{l} x^r_{Ji} := x^{r-1}_{Ji} - a^r_{1k}\text{è}^r \\ \dots\dots\dots\dots\dots\dots \\ x^r_{Jm} := x^{r-1}_{Jm} - a^r_{mk}\text{è}^r \end{array}\right\} \quad \text{- базисные компоненты}$$

$$x^r_k := \begin{cases} è^r & \text{при } è^r > 0; \\ á_k + è^r & \text{при } è^r < 0, \end{cases}$$

$x_s^r = x_s^{r-1}$ - для небазисных переменных ($S \neq k$).

$$X^r_{n+m+1} := x^{r-1}_{n+m+1} + a^r_{m+1,k} è^r,$$

$$X^r_{n+m+2} := x^{r-1}_{n+m+2} + a^r_{m+2,k} è^r.$$

9 Если $l^r \neq 0$, то l^r-ой компоненте вектора N^{r-1} присваиваем значение k, иначе переходим к п. 11.

10 Преобразуем матрицу:

$$a_{ij}^{r+1} := a^r_{ij} - \frac{a^r_{lj}}{a^r_{lk}} a^r_{ik}, \qquad i \neq l^r;$$

$$a_{lj}^{r+1} := \frac{a^r_{lj}}{a^r_{lk}}, \quad i = 1, 2, ..., m+2; \qquad j = 1, ..., n.$$

11 На 1 этапе переходим к п. 1.
На 2 этапе переходим к п. 13.
На 3 этапе переходим к п. 16.

12 Если в базисе содержатся искусственные переменные, то переходим к п. 15, иначе переходим ко 2 этапу.

Этап 2 - нахождение оптимального решения.

13 Выбираем номер k среди номеров небазисных компонент из условия

$$a^r_{m+1,k} = \min a^r_{m+1,i}, \quad i \in J_r,$$

где $$a^r_{m+1,i} = \begin{cases} a^r_{m+1,i} & \text{при } x_i^{r-1} = 0; \\ -a^r_{m+1,i} & \text{при } x_i^{r-1} = á_i, \end{cases} \qquad J_r = \{i / i \in N^{r-1}, i \leq n\}.$$

14 Если $a^r_{m+1,k} \geq 0$, то конец (найдено оптимальное решение), иначе переходим к п. 3.

15 Если $x^{r-1}_{n+m+2} > 0$, то конец (задача несовместна), иначе переходим к 3 этапу.

Этап 3 - нахождение оптимального решения в вырожденном случае.

Вырожденный случай может возникнуть в конце первого этапа, если невязка равна нулю, но не все искусственные переменные выведены из базиса.

16 Выбираем номер k среди номеров небазисных переменных из условия

$$a^r_{m+1,k} = \min a^r_{m+1,i}, \quad i \in J_r,$$

где $$a^r_{m+1,i} = \begin{cases} a^r_{m+1,i} & \text{при } x_i^{r-1} = 0; \\ -a^r_{m+1,i} & \text{при } x_i^{r-1} = á_i, \end{cases} \qquad J_r = \{i / i \notin N^{r-1}, i \leq n, a^r_{m+2,i} = 0\}.$$

17 Переходим к п. 14.

4.3 МОДИФИЦИРОВАННЫЙ СИМПЛЕКС-МЕТОД
(*второй алгоритм*)

Этот алгоритм использует тот факт, что данные, необходимые для перехода от одного опорного плана к другому, могут быть получены, если известны для каждого базиса обратная матрица U и условия задачи.

Основная разница между обычным симплексным методом и модифицированным заключается в том, что в первом мы преобразуем все элементы матрицы задачи по формулам исключения, тогда как во втором случае нам необходимо преобразовать по тем же формулам лишь элементы матрицы U.

Исходный опорный план ($n + m$-мерный)

$$X^0 := (0, 0, ..., 0, b_1, b_2, ..., b_m).$$

Исходный вектор номеров базисных компонент

$$N^0 = (j_1, j_2, ..., j_m) := (n + 1, n + 2, ..., n + m).$$

Матрица $U^0 = E_{m+2}$. Рассмотрим r-ую итерацию.

Этап 1 - нахождение допустимого решения.

1 Вычисляем оценки

$$\delta_j^r = u^r_{m+2,1} a_{1,j} + u_{m+2,2} a_{2j} + ... + u_{m+2,m+2} a_{m+2,j}, \quad j = 1, 2, ..., n.$$

2 Выбираем номер k среди номеров небазисных переменных из условия

$$\bar\delta_k^r = \min \bar\delta_i^r, \quad i \in J_r,$$

где $\bar\delta_i^r = \begin{cases} \delta_i^r & \text{при} \quad x_i^{r-1} = 0; \\ -\delta_i^r & \text{при} \quad x_i^{r-1} = \alpha_i, \end{cases}$ $\qquad J_r = \{i/i \notin N^{r-1}, i \le n\}.$

3 Если $\bar\delta_k^r \ge 0$, то переходим к п. 14, иначе выполняем п. 4.

4 Вычисляем коэффициенты разложения k-го столбца по векторам базиса

$$x^r_{ik} = u^r_{i1} a_{1k} + u^r_{i2} a_{2k} + ... + u^r_{i,m+2} a_{m+2,k}, \quad i = 1, 2, ..., m + 2.$$

5 Если $\alpha_k < 10^{15}$, то переходим к п. 18, иначе выполняем п. 6.

6 Если для всех i ($i = 1, 2, ..., m$) $x^r_{ik} < 0$, то выполняем п. 7, иначе переходим к п. 8.

7 Если $\alpha_{ji} < 10^{15}$ хотя бы для одного i ($i = 1, 2, ..., m$), то выполняем п. 8, иначе конец (линейная форма неограниченна).

8 Вычисляем θ^r

при $x_k^{r-1} = 0$: $\quad \theta^r = \min \begin{cases} \alpha_k; \\ x_{Ji}^{r-1} / x_{ik}^r, (x_{ik}^r > 0); \\ (x_{Ji}^{r-1} - \alpha_{Ji}) / x_{ik}^r, (x_{ik}^r < 0), \end{cases}$

при $x_k^{r-1} = \alpha_k$ $\quad -\theta^r = \min \begin{cases} \alpha_k; \\ x_{Ji}^{r-1} / (-x_{ik}^r), (x_{ik}^r < 0); \\ (x_{Ji}^{r-1} - \alpha_{Ji}) / (-x_{ik}^r), (x_{ik}^r > 0), \end{cases}$ $\qquad i = 1, 2, ..., m.$

9 Если $\theta^r = \pm \alpha_k$, то $l^r := 0$, иначе l^r равно тому i, при котором реализуется значение θ^r.

10 Пересчитываем план и значения линейных форм:

$$\left.\begin{array}{l} x^r_{J1} := x^{r-1}_{J1} - x^r_{1k}è^{\,r} \\ \text{.........................} \\ x^r_{Jm} := x^{r-1}_{Jm} - x^r_{mk}è^{\,r} \end{array}\right\}$$ - базисные компоненты,

$$x^r{}_k = \begin{cases} è^{\,r} & \text{при} \quad è^{\,r} > 0; \\ á_k + è^{\,r} & \text{при} \quad è^{\,r} < 0, \end{cases}$$

$x_s{}^r = x_s{}^{r-1}$ - для небазисных переменных ($S \neq k$);

$$x^r{}_{n+m+1} := x^{r-1}{}_{n+m+1} + x^r{}_{m+1,k}è^{\,r};$$

$$x^r{}_{n+m+2} := x^{r-1}{}_{n+m+2} + x^r{}_{m+2,k}è^{\,r}.$$

11 Если $l^r \neq 0$, то l^r-ой компоненте вектора N^{r-1} присваиваем значение k, иначе переходим к п. 13.

12 Преобразуем матрицу U^r:

$$u_{ij}{}^{r+1} := u^r{}_{ij} - \frac{u^r_{lj}}{x^r_{llk}} x^r{}_{ik}, \quad i \neq l^r;$$

$$u_{lj}{}^{r+1} := \frac{u^r_{lj}}{x^r_{llk}};$$

$$i = 1, 2, ..., m + 2;$$

$$j = 1, 2, ..., m.$$

13 На 1 этапе переходим к п. 1.
На 2 этапе переходим к п. 15.
На 3 этапе переходим к п. 19.

14 Если в базисе содержатся искусственные переменные, то переходим к п. 18, иначе переходим к 2 этапу.

Этап 2 - нахождение оптимального решения.

15 Вычисляем оценки

$ã^{\,r}_j = u^r{}_{m-1,1}a_{1,j} + u_{m-1,2}a_{2j} + ... + u_{m-1,m-1}a_{m-1,j}, \qquad j = 1, 2, ..., n.$

16 Выбираем номер k среди номеров небазисных компонент из условия

$$\bar{\gamma}^r_k = \min \bar{\gamma}^r_i, \quad i \in J_r,$$

где $\bar{\gamma}^r_i = \begin{cases} ã^{\,r}_i & \text{при} \quad x^{r-1}_i = 0; \\ -ã^{\,r}_i & \text{при} \quad x^{r-1}_i = á_i, \end{cases}$ $\qquad J_r = \{i / i \notin N^{r-1}, \ i \leq n\}.$

17 Если $\bar{\gamma}^r_k \geq 0$, то конец (найдено оптимальное решение), иначе переходим к п. 4.

18 Если $x^{r-1}{}_{n+m+2} > 0$, то конец (задача несовместна), иначе переходим к 3 этапу.

Этап 3 - нахождение оптимального решения в вырожденном случае.

19 Считаем оценки

$$\delta^r_j = u^r{}_{m+2,1}a_{1,j} + u^r{}_{m+2,2}A_{2j} + ... + u_{m+2,m+2}a_{m+2,j};$$

$$\gamma_j^r = u^r_{m+1,1}a_{1,j} + u_{m+1,2}a_{2j} + ... + u_{m+1,m+1}a_{m+1,j}; \qquad j = 1, 2, ..., n.$$

20 Выбираем k среди номеров небазисных компонент из условия

$$\bar{\gamma}_k^r = \min \bar{\gamma}_i^r, \qquad i \in J_r,$$

где $\bar{\gamma}_i^r = \begin{cases} \gamma_i^r & \text{при} \quad x_i^{r-1} = 0; \\ -\gamma_i^r & \text{при} \quad x_i^{r-1} = \alpha_i, \end{cases} \qquad J^r = \{i/i \notin \mathrm{N}^{r-1}, i \le n\}.$

4.4 СИМПЛЕКС-МЕТОД С ИСПОЛЬЗОВАНИЕМ МЕТОДА ОТРАЖЕНИЙ

(*третий алгоритм*)

Здесь сохраняется вся схема простого симплекс-метода (перебор планов в поисках оптимального), но все необходимые для такого перебора системы уравнений решаются с помощью метода отражений. Напомним, что в первых двух алгоритмах они решались методом исключения Гаусса.

Основная идея метода отражений состоит в разложении матрицы системы (B) в произведение правой треугольной (L) и ортогональной матрицы отражения (U). Метод описан в [3]. Исходный опорный план ($n + m$-мерный)

$$\mathrm{X}^0 := (0, 0, ..., 0, b_1, b_2, ..., b_m).$$

Исходный вектор свободных членов

$$b^0 := (b_1, b_2, ..., b_m).$$

Исходный вектор номеров базисных компонент

$$\mathrm{N}^0 = (j_1, j_2, ..., j_m) := (n + 1, n + 2, ..., n + m).$$

Исходная базисная матрица

$$\mathrm{B}^0 = \mathrm{E}_m \text{ (квадратная матрица).}$$

Рассмотрим r-ую итерацию.

Этап 1 - нахождение базисного допустимого решения.

1 Вычисляем вектор $\mathrm{Y}^r_m(y^r_{m,1}, y^r_{m,2}, y^r_{m,3}, y^r_{m,4}, ..., y^r_{m,m})$:

$$y^r_{m,1} = \frac{á_{m+2,ji}}{B_{11}^r},$$

$$y^r_{m,i} = \frac{á_{m+2,ji} - \sum_{q=1}^{i-1} \mathrm{B}_{q,i}^r y_{m,q}^r}{\mathrm{B}_{ii}^r}, \qquad i = 2, 3, ..., m.$$

2 Вычисляем вектор $\mathrm{Y}^r{}_1(y^r_{1,1}, y^r_{1,2}, y^r_{1,3}, y^r_{1,4}, ..., y^r_{1,m})$:

$$\mathrm{Y}_\nu^r = \mathrm{Y}_{\nu+1}^r - 2\left(\mathrm{Y}_{\nu+1}^r,\ \mathrm{W}_\nu^r\right)\mathrm{W}_\nu^r, \qquad \nu = m - 1, m - 2, ..., 1.$$

3 Считаем оценки $ä_j^r = a_{m+2,j} - (\mathrm{A}_j, \mathrm{Y}_1{}^r), \quad j = 1, 2, ..., n.$

4 Выбираем номер k среди номеров небазисных переменных из условия

$$\bar{\delta}_k^r = \min \bar{\delta}_i^r, \quad i \in J^r,$$

где
$$\bar{\delta}_i^r = \begin{cases} \delta_i^r & \text{при} \quad x_i^{r-1} = 0; \\ -\delta_i^r & \text{при} \quad x_i^{r-1} = \alpha_i, \end{cases} \qquad J_r = \{i/i \notin \mathrm{N}^{r-1}, i \le n\}.$$

5 Если $\bar{\delta}_k^r \le 0$, то переходим к п. 24, иначе выполняем п. 6.

6 Положив Z^r_m $(z^r_{m,1}, z^r_{m,2}, ..., z^r_{m,m}) = \mathrm{A}_k$, вычисляем

$$\mathrm{Z}^r_1\ (z^r_{1,1}, z^r_{1,2}, ..., z^r_{1,m}):$$

$$\mathrm{Z}_\nu^r = \mathrm{Z}_{\nu+1}^r - 2\left(\mathrm{Z}_{\nu+1}^r,\ \mathrm{W}_\nu^r\right)\mathrm{W}_\nu^r, \qquad \nu = m - 1, m - 2, ..., 1.$$

7 Вычисляем коэффициенты разложения вектора A_k по векторам базиса:

$$x^r_{mk} = \frac{z_{1,m}^r}{B_{m,m}^r},$$

$$x^r_{ik} = \frac{z_{1,i}^r - \sum\limits_{q=i+1}^{m} B_{i,q}^r x_{qk}^r}{B_{ii}^r}, \qquad i = m - 1, ..., 1.$$

8 Если $\alpha_k < 10^{15}$, то переходим к п. 11, иначе выполняем п. 9.

9 Если для всех i ($i = 1, 2, ..., m$) $x^r_{ik} < 0$, то выполняем п. 10, иначе переходим к п. 11.

10 Если $\alpha_{ji} < 10^{15}$ хотя бы для одного i ($i = 1, 2, ..., m$), то выполняем п. 11, иначе конец (линейная форма неограниченна).

11 Вычисляем è ^{r}

при $x_k^{r-1} = 0$: $\text{è}^r = \min \begin{cases} \alpha_k; \\ x_{Ji}^{r-1} / x_{ik}^r, (x_{ik}^r > 0); \\ (x_{Ji}^{r-1} - \alpha_{Ji}) / x_{ik}^r, (x_{ik}^r < 0), \end{cases}$

при $x_k^{r-1} = \alpha_k$: $-\text{è}^r = \min \begin{cases} \alpha_k; \\ x_{Ji}^{r-1} / (-x_{ik}^r), (x_{ik}^r < 0); \\ (x_{Ji}^{r-1} - \alpha_{Ji}) / (-x_{ik}^r), (x_{ik}^r > 0), \end{cases} \qquad i = 1, 2, ..., m.$

12 Если $\text{è}^r = \pm\alpha_k$, то полагаем 1^r: = 0, иначе 1^r равно тому i, при котором реализуется значение è^r.

13 Если $l^r \ne 0$, то переходим к п. 18.

14 Если $\text{è}^r > 0$, то x_k^r: = α_k, E^k: = 1; иначе x_k^r: = 0, E^k: = -1.

15 Преобразуем вектор свободных членов: b^r: = b^{r-1} - $\mathrm{A}_k \mathrm{E}^r \alpha_k$.

16 Пересчитываем план:

$$\left.\begin{array}{l} x_{J1}^r := x_{J1}^{r-1} - x_{1k}^r \theta^r \\ \dots\dots\dots\dots\dots \\ x_{Jm}^r := x_{Jm}^{r-1} - x_{mk}^r \theta^r \end{array}\right\} \text{ - базисные компоненты}$$

$$x^r_k = \begin{cases} \text{è}^r & \text{при} \quad \theta^r > 0; \\ \alpha_k + \theta^r & \text{при} \quad \theta^r < 0, \end{cases}$$

$x_s^r = x_s^{r-1}$ для небазисных переменных ($S \ne k$).

17 Переходим к п. 21.

18 l^r-ой компоненте вектора N^{r-1} присваиваем значение k. Получаем N^r.

19 Формализуем базисную матрицу B^{r+1} из столбцов матрицы A, соответствующих базисным переменным.

20 Если выводимая из базиса компонента $x_p^{\ r}$ вышла на верхнюю границу, то b^r: = b^{r-1}- $A_p \alpha_p$ и $x_p^{\ r}$: = α_p, иначе $x_p^{\ r}$: = 0.

21 Преобразуем B^{r+1}.

Считаем для $\nu = 1, 2, ..., m - 1$:

$$u^\nu = (0, 0, ..., 0, B_{\nu,\nu}^{(\nu-1)}, B_{\nu+1,\nu}^{(\nu-1)}, ..., B_{m,\nu}^{(\nu-1)}),$$

$$T^\nu = \sqrt{(u^\nu, u^\nu)} \times \mathrm{sign}\,(B_{\nu,\nu}^{(\nu-1)}),$$

$$V^\nu = \sqrt{2|T^\nu|^2 + 2|T^\nu||u_\nu^\nu|}.$$

Если V^ν, то конец (матрица плохо обусловлена), иначе считаем

$$W^\nu := \frac{1}{V^\nu}(u_1^{\ \nu}, u_2^{\ \nu}, ..., u_\nu^\nu - T^\nu, u_{\nu+1}^\nu, ..., u_m^\nu),$$

$$(b^r)^\nu: = (b^r)^{\nu-1} - 2((b^r)^{\nu-1}, W^\nu)W^\nu;$$

$$B_j^\nu := B_j^{\nu-1} - 2\left(B_j^{\nu-1}, W^\nu\right)W^\nu, \quad j = 1, 2, ..., m.$$

22 Находим план X^r:

$$x^r_{Jm} := \frac{b_m^r}{B_{mm}^{r+1}},$$

$$x^r_{Ji} := \frac{b_i^r - \sum_{q=i+1}^{m} B_{i,Jq}^{r+1} x_q^r}{B_{i,i}^{r-1}}, \quad i = m - 1, m - 2, ..., 1.$$

23 На 1 этапе переходим к п. 1.

На 2 этапе переходим к п. 13.

На 3 этапе переходим к п. 16.

24 Если в базисе остались искусственные переменные, то переходим к п. 30, иначе переходим ко 2 этапу.

Этап 2 - нахождение оптимального решения.

25 Вычисляем вектор Y^r_m ($y^r_{m,1}, y^r_{m,2}, ..., y^r_{m,m}$):

$$y^r_{m,1} = \frac{a_{m+2,ji}}{B_{11}^r},$$

$$y^r_{m,i} = \frac{a_{m+2,ji} - \sum_{q=1}^{i-1} B_{q,i}^r y_{m,q}^r}{B_{ii}^r}, \quad i = 2, 3, ..., m.$$

26 Вычисляем вектор $Y^r_1(y^r_{1,1}, y^r_{1,2}, y^r_{1,3}, y^r_{1,4}, ..., y^r_{1,m})$:

$$Y_\nu^r = Y_{\nu+1}^r - 2\left(Y_{\nu+1}^r, W_\nu^r\right)W_\nu^r, \qquad \nu = m - 1, m - 2, ..., 1.$$

27 Считаем оценки

$$\gamma^r{}_j = a_{m+1,j} - (A_j,\, Y_1{}^r), \quad j = 1, 2, ..., n.$$

28 Выбираем номер k среди номеров небазисных переменных из условия

$$\bar{\gamma}^r_k = \min \bar{\gamma}^r_i, \quad i \in J_r,$$

где $\bar{\gamma}^r_i = \begin{cases} \gamma^r_i & \text{при} \quad x_i^{r-1} = 0; \\ -\gamma^r_i & \text{при} \quad x_i^{r-1} = \alpha_i, \end{cases}$ $\quad J_r = \{i/i \notin N^{r-1},\, i \le n\}.$

29 Если $\bar{\gamma}^r_k \ge 0$, то конец (найдено оптимальное решение), иначе переходим к п. 6.

30 Считаем невязку $x_{n+m+2} = \sum\limits_{j=1}^{n} a_{m+2,j} x_j^{r-1} - a_{m+2,0}$. Если $x_{n+m+2} > 0$, то конец (задача несовместна), иначе переходим к 3 этапу.

Этап 3 - нахождение оптимального решения в вырожденном случае.

31 Вычисляем векторы $Y^r{}_m\,(y^r{}_{m,1}, y^r{}_{m,2}, ..., y^r{}_{m,m})$ и $C^r{}_m\,(c^r{}_{m,1}, c^r{}_{m,2}, ..., c^r{}_{m,m})$:

$$y^r{}_{m,1} = \frac{a_{m+2,j1}}{B^r_{11}}, \quad c^r{}_{m,1} = \frac{a_{m+1,j1}}{B^r_{11}};$$

$$y^r{}_{m,i} = \frac{a_{m+2,ji} - \sum\limits_{q=1}^{i-1} B^r_{q,i} y^r_{m,q}}{B^r_{ii}};$$

$$c^r{}_{m,i} = \frac{a_{m+1,ji} - \sum\limits_{q=1}^{i-1} B^r_{q,i} c^r_{m,q}}{B^r_{ii}}, \quad i = 2, 3, ..., m.$$

32 Вычисляем векторы $Y^r{}_1(y^r{}_{1,1}, y^r{}_{1,2}, y^r{}_{1,3}, y^r{}_{1,4}, ..., y^r{}_{1,m})$ и $C^r{}_1(c^r{}_{1,1}, c^r{}_{1,2}, c^r{}_{1,3}, c^r{}_{1,4}, ..., c^r{}_{1,m})$:

$$Y^r_\nu = Y^r_{\nu+1} - 2\left(Y^r_{\nu+1},\; W^r_\nu\right) W^r_\nu;$$

$$C^r_\nu = C^r_{\nu+1} - 2\left(C^r_{\nu+1},\; W^r_\nu\right) W^r_\nu, \quad \nu = m - 1, m - 2, ..., 1.$$

33 Считаем оценки

$$\nu^r{}_j = a_{m+2,j} - (A_j,\, Y_1{}^r);$$

$$\nu^r{}_j = a_{m+1,j} - (A_j,\, C_1{}^r), \quad j = 1, 2, ..., n.$$

34 Выбираем номер k среди номеров небазисных компонент из условия

$$\bar{\gamma}^r_k = \min \bar{\gamma}^r_i, \quad i \in J_r$$

где $\bar{\gamma}^r_i = \begin{cases} \gamma^r_i & \text{при} \quad x_i^{r-1} = 0; \\ -\gamma^r_i & \text{при} \quad x_i^{r-1} = \alpha_i, \end{cases}$ $\quad J^r = \{i/i \notin N^{r-1},\; i \le n,\, \nu^r{}_i = 0\}.$

Переходим к п. 29.

4.5 ПРОГРАММНАЯ РЕАЛИЗАЦИЯ МЕТОДОВ

Для решения задачи линейного программирования используются функции boundsimplex1(), boundsimplex2(), boundsimplex3(). Они вычисляют оптимальное решение задачи и вектор номеров базисных компонент оптимального решения по алгоритмам, описанным в разделах 4.2, 4.3, 4.4. Прототипы функций одинаковы и выглядят следующим образом:

```
int boundsimplex1 ( int mb, double **a, double *b, double *c, double *x,
double *alfa, double epsilon, double *zmin,
int m , int n, int z )
```

Параметры функций имеют следующий смысл:

mb - полагается равным "1", если задача поставлена на максимум, и "-1" в противном случае;

a - матрица коэффициентов A[m][n];

b - вектор ограничений b[m];

c - вектор коэффициентов целевой функции c[n];

x - вектор результатов x[n];

alfa - вектор ограничений [1 ... 2z], его нечетные компоненты содержат номера ограниченных сверху переменных, а четные компоненты - соответствующие значения верхних границ;

epsilon - точность вычисления невязки;

zmin - полученное значение целевой функции;

m - число строк матрицы A;

n - число столбцов матрицы A;

z - количество ограниченных сверху переменных ($z \neq 0$);

Все функции возвращают значение 1, 2 или 3. Если возвращаемое значение 2, то найдено оптимальное решение, при этом массив x содержит n компонент оптимального решения x_1, x_2, ..., x_n, если 1, то линейная форма не ограничена, если 3, то задача несовместна.

Пусть, например, надо минимизировать $Z = x_2 - 3x_3 + 2x_5$ при условиях

$$x_1 + 3x_2 - x_3 + 2x_5 = 7;$$

$$-2x_2 + 4x_3 + x_4 = 12;$$

$$-4x_2 + 3x_3 + 8x_5 + x_6 = 10;$$
$$x_j \geq 0, \quad j = 1, 2, ..., 6, \quad x_1 \leq 1, \quad x_3 \leq 5{,}5.$$

Тогда следует объявить переменные и присвоить им начальные значения

```
int m = 3, n = 6, mb = -1, z = 2, t;
    double epsilon = 1e-4, x[6],
        a[3][6] = { {1,3,-1,0,2,0}, {0,-2,4,1,0,0}, {0,-4,3,0,8,1} },
       c[6] = {0,1,-3,0,2,0},
       b[3] = {7,12,10};
       alfa[4] = {1,1,3,5.5},
       *zmin = new double;
```

После выполнения функции

```
t = boundsimplex1(mb, a, b, c, x, alfa, epsilon, zmin, m, n, z)
```

получим искомые значения переменных в массиве x и значение целевой функции в переменной zmin, значение t будет равным 2.

СПИСОК ЛИТЕРАТУРЫ

1 *Гольдштейн Е. С.*, *Юдин Д. Б.* Новые направления в линейном программировании. М.: Сов. радио, 1967. 523 с.

2 *Юдин Д. Б.*, *Гольштейн Е. С.* Линейное программирование теория и методы и приложения. М.: Наука, 1969. 424 с.

3 *Еремин И. И.*, *Астафьев Н. Н.* Введение в теорию линейного и выпуклого программирования. М.: Наука, 1970. 191 с.

СОДЕРЖАНИЕ

ОПИСАНИЕ ЯЗЫКА СИ …………………………….…… 3

1.1 БАЗОВЫЕ ПОНЯТИЯ ЯЗЫКА СИ ……………………… 3

1.1.1 Используемые символы (алфавит) …………………….. 3

1.1.2 Ключевые слова …………………………………..……… 5

1.1.3 Идентификаторы ………………………………... …... 5

1.1.4 Использование комментариев в тексте программы ….. 6

1.2 БАЗОВЫЕ ТИПЫ ДАННЫХ И ОБЪЯВЛЕНИЕ ПЕРЕМЕННЫХ …………………………………………………… 6

1.2.1 Целые типы данных …………………………………….. 7

1.2.2 Данные плавающего типа …………………………..… 10

1.2.3 Переменные перечислимого типа ……………………… 10

1.2.4 Указатели ………………………………………………... 12

1.2.5 Массивы …………………………………………………. 13

1.2.6 Структуры ……………………………………………… 15

1.2.7 Объединения (смеси) …………………………………... 16
1.2.8 Поля битов ………………………………………………. 17
1.2.9 Конструирование переменных с изменяемой структурой 17
1.2.10 Инициализация данных ……………………………….. 19
1.2.11 Конструирование переменных сложных типов …….. 21
1.2.12 Объявление типов и абстрактные типы ……………… 22
1.3 ВЫРАЖЕНИЯ И ПРИСВАИВАНИЯ ………………….... 23
1.3.1 Операнды и операции …………………………………... 23
1.3.2 Преобразования при вычислении арифметических выражений ………………………………………………………… . 30
1.3.3 Операции отрицания и дополнения ………………….... 30
1.3.4 Операции разадресации и адреса ……………………..… 31
1.3.5 Операция sizeof ………………………………………… 32
1.3.6 Мультипликативные операции ………………………… 32
1.3.7 Аддитивные операции ………………………………….. 33
1.3.8 Операции сдвига ……………………………………….. 34
1.3.9 Поразрядные операции ………………………………... 4
1.3.10 Логические операции …………………………………. 35
1.3.11 Операция последовательного вычисления …………… 36
1.3.12 Условная операция ……………………………………. 36

1.3.13 Простое присваивание ………………………………… 36
1.3.14 Операции увеличения и уменьшения ………………… 37
1.3.15 Составное присваивание …………………………….... 38
1.3.16 Побочные эффекты …………………………………..... 38
1.3.18 Преобразование типов ……………………………….... 39
1.4 ОПЕРАТОРЫ ……………………………………...… 41
1.4.1 Оператор выражение …………………………….…. 41
1.4.2 Пустой оператор ……………………………………... 41
1.4.3 Составной оператор ………………………………….. 42
1.4.4 Оператор if …………………………………..……….. 42
1.4.5 Оператор switch ……………………………………… 44
1.4.6 Оператор break …………………………………..…… 46
1.4.7 Оператор for …………………………………….….. 46
1.4.8 Оператор while ……………………………………….. 48
1.4.9 Оператор do while …………………………………… 48
1.4.10 Оператор continue ……………………………………… 49
1.4.11 Оператор return ……………………………….…… 49
1.4.12 Оператор goto ……………………………………….. 50
1.5 ОПРЕДЕЛЕНИЕ И ВЫЗОВ ФУНКЦИЙ ……………..… 50
1.6 СТРУКТУРА ПРОГРАММЫ И КЛАССЫ ПАМЯТИ …. 58

1.6.1 Исходные файлы и объявление переменных ………… 58
1.6.2 Объявления функций …………………………………… 62
1.6.3 Время жизни и область видимости программных объектов 63
1.6.4 Инициализация глобальных и локальных переменных 63
1.7 УКАЗАТЕЛИ И АДРЕСНАЯ АРИФМЕТИКА ………… 64
1.7.1 Методы доступа к элементам массивов ………………. 64
1.7.2 Указатели на многомерные массивы ………………….. 5
1.7.3 Операции с указателями ……………………………….. 67
1.7.4 Массивы указателей ……………………………………. 68
1.7.5 Динамическое размещение массивов …………………. 70
1.8 ДИРЕКТИВЫ ПРЕПРОЦЕССОРА ……………………… 73
1.8.1 Директива #include ……………………………………… 74
1.8.2 Директива #define ………………………………………. 74
1.8.3 Директива #undef ……………………………………….. 75
2 ОПИСАНИЕ ЯЗЫКА C++ ………………………………….….. 75
2.1 НОВЫЕ ВОЗМОЖНОСТИ C++, НЕ СВЯЗАННЫЕ С ООП 76
2.1.1 Комментарии ……………………………………………. 76
2.1.2 Объявление данных внутри блока …………………….. 76
2.1.3 Оператор области видимости ………………………….. 77
2.1.4 Операторы new и delete ………………………………… 77

2.1.5 Прототипы функций и аргументы по умолчанию ……. 77
2.1.6 Преобразование типов …………………………………. 78
2.1.7 Ссылки ………………………………………………….. 78
2.1.8 Подставляемые функции ………………………………. 80
2.1.9 Перегружаемые функции ………………………………. 80
2.1.10 Перегрузка операторов ………………………………. 81
2.2 ИСПОЛЬЗОВАНИЕ КЛАССОВ …………………………. 82
2.2.1 Классовые типы …………………………………………. 82
2.2.2 Компоненты данных ……………………………………. 84
2.2.3 Функциональные компоненты ………………………… 87
2.2.4 Операторные функции ………………………………….. 89
2.2.5 Защита доступа и дружественные функции ………….. 91
2.2.6 Инициализация и преобразования …………………….. 92
2.2.7 Указатели на компоненты класса ……………………… 95
2.3 АБСТРАКЦИЯ ДАННЫХ ……………………………… 97
2.3.1 Комплексные числа …………………………………….. 97
2.3.2 Строки ………………………………………………… 101
2.3.3 Упорядоченные последовательности ………………… 04
2.3.4 Общность …………………………………………….. 107
2.3.5 Абстракция управления ……………………………….. 109

2.4 НАСЛЕДОВАНИE ……………………………………….. 113
2.4.1 Базовые и производные классы ………………………... 113
2.4.2 Иерархии классов ………………………………………. 118
2.4.3 Виртуальные функции …………………………………. 120
2.4.4 Защищенные элементы …………………………………. 123
2.4.5 Наследование как инструмент проектирования ……… 124
2.4.6 Наследование для расширения интерфейса …………... 125
2.4.7 Множественное наследование ………………………… 127
2.4.8 Виртуальные базовые классы …………………………. 130
2.5 ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ ……………………………………………………………….. 131
2.5.1 Проектирование в терминах объектов ………………… 132
2.5.2 Объектные типы как модули …………………………… 135
2.5.3 Динамический объектно-ориентированный стиль …... 136
2.6 УПРАВЛЕНИЕ ПАМЯТЬЮ …………………………….. 141
2.6.1 Управление памятью при помощи конструкторов и деструкторов …………………………………………………… 141
2.6.2 Операторы new и delete ………………………………… 143
2.6.3 Управление памятью для массивов …………………… 145
2.6.4 Специфические new и delete для классов ……………... 146
2.6.5 Оператор -> ……………………………………………… 150

2.6.6 Конструктор со ссылкой на собственный класс ……… 151
2.6.7 Семантика неявной копии ……………………………… 153

3 ПОДГОТОВКА И ВЫПОЛНЕНИЕ ПРОГРАММ НА КОМПЬЮТЕРЕ …………………………………….………… 154

3.1 НАЧАЛО И КОНЕЦ СЕАНСА ………………………….. 156
3.2 ФАЙЛОВАЯ СИСТЕМА ………………………………… 157
3.2.1 Права доступа к файлам и каталогам …………………. 159
3.2.2 Команды работы с файловой системой ……………….. 60
3.2.3 Команды работы с файлами ……………….……… 163
3.2.4 Шаблоны имен файлов …….……………………… 65
3.2.5 Переключение ввода-вывода …………………………... 165
3.3 ПРОГРАММНЫЕ КАНАЛЫ И КОНВЕЕРЫ …………... 167
3.3.1 Последовательное и одновременное выполнение нескольких команд ………………………………………………... 167
3.3.2 Файлы устройств …………………………………...…… 169
3.4 ПОДГОТОВКА И ВЫПОЛНЕНИЕ ПРОГРАММ В СРЕДЕ UNIX ………………………………………………… 170

4 МЕТОДЫ ЛИНЕЙНОГО ПРОГРАММИРОВАНИЯ ……. 174

4.1 ПОСТАНОВКА ЗАДАЧИ ЛИНЕЙНОГО ПРОГРАММИРОВАНИЯ ………………………………………………. 174
4.2 ПРОСТОЙ СИМПЛЕКС-МЕТОД (*первый алгоритм*) 175
4.3 МОДИФИЦИРОВАННЫЙ СИМПЛЕКС-МЕТОД (*второй алгоритм*) ………………………………………………. 177

4.4 СИМПЛЕКС-МЕТОД С ИСПОЛЬЗОВАНИЕМ МЕТОДА ОТРАЖЕНИЙ (*третий алгоритм*) ……………………… 180
4.5 ПРОГРАММНАЯ РЕАЛИЗАЦИЯ МЕТОДОВ ………… 184
СПИСОК ЛИТЕРАТУРЫ ……………………………………….... 186