МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ
РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ
УЧРЕЖДЕНИЕ ВЫСШЕГО ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ
«САМАРСКИЙ ГОСУДАРСТВЕННЫЙ АЭРОКОСМИЧЕСКИЙ
УНИВЕРСИТЕТ имени академика С.П.КОРОЛЕВА
(НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ)»

Е.В. Мясников

Язык программирования C++

Электронное учебное пособие

Самара
2011

Автор: МЯСНИКОВ Евгений Валерьевич

Учебное пособие посвящено одному из самых известных на сегодняшний день языков программирования – языку C++. В пособии отражены основные аспекты программирования на C++ как с использованием процедурного, так и объектно-ориентированного подходов. Особое внимание в пособии отводится перегрузке операций, созданию шаблонов функций и классов, а также работе с исключениями. Пособие снабжено примерами, в конце разделов приводятся вопросы для самоконтроля и задания на программирование.

Учебное пособие предназначено для студентов факультета информатики, направление 010400 – Прикладная математика и информатика, бакалавриат (010400.62)/магистратура (010400.68, магистерская программа – Технологии параллельного программирования и суперкомпьютинг).

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Пособие, которое Вы сейчас держите в руках посвящено одному из самых известных на сегодняшний день языков программирования – языку C++. На этом языке создано не только множество прикладных программ, но и практически все системное программное обеспечение, в том числе и многие операционные системы. И хотя в последние годы интерес к языку несколько снизился из-за выхода новой платформы и поддерживающего ее языка (имеется в виду платформа .NET и язык C#) и из-за стремительного роста числа internet – приложений, C++ продолжает прочно занимать нишу профессионального инструмента для создания эффективного программного обеспечения.

Возможности любого языка программирования определяются, прежде всего, синтаксисом языка, а уже затем набором стандартных библиотек. И здесь следует отметить, что C++ является, видимо, наиболее мощным из получивших широкое распространение на сегодняшний день языков. Будет нелишне также отметить, что все годы, начиная с рождения языка C, а затем и C++, в индустрии программного обеспечения накопилось огромное количество библиотек (в том числе и свободно распространяемых), нацеленных на решение самых разных прикладных задач.

Изучение C++ не будет лишним даже в том случае, если Вам не придется столкнуться с ним в своей профессиональной деятельности. Освоение таких распространенных языков программирования как C# и Java будет проходить гораздо проще, так как многие конструкции этих языков заимствованы из C++. Кроме того, полагаю, что резюме любого IT специалиста только выиграет, если в нем будет фигурировать знание этого замечательного языка.

Справедливости ради, стоит отметить, что для решения ряда задач на сегодняшний день существуют весьма эффективные средства, разработка на которых будет занимать меньше времени, чем с использованием классического C++. Этот факт, однако, ни сколь не умаляет достоинства C++, а говорит лишь о том, что IT-специалист в современном мире должен идти в ногу со временем и выбирать адекватные средства при решении встающих перед ним задач.

Настоящее пособие подготовлено автором на основе курса, читаемого им в СГАУ на протяжении ряда лет. Пособие условно можно разбить на две части, отражающие основы программирования на C++ с использованием процедурного (разделы 1-4) и объектно-ориентированного (разделы 5-9) подходов. К сожалению, ограниченный объем пособия не позволяет осветить ряд безусловно интересных нововведений языка, а представляет

сформировавшийся за время преподавания и профессиональной деятельности взгляд автора на «классический» язык C++.

Изучение настоящего пособия предполагает, что читатель уже знаком с основами алгоритмизации и имеет хотя бы минимальный опыт программирования на каком-либо другом языке. Для успешного освоения материала и выполнения приведенных в нем заданий, Вам потребуется какая-либо среда разработки, поддерживающая язык C++.

1 БАЗОВЫЕ ЭЛЕМЕНТЫ ЯЗЫКА, ТИПЫ И ПЕРЕМЕННЫЕ

1.1 Формальное описание

Задумывались ли Вы, чем определяется то, как выглядит программа на том или ином языке программирования? Конечно же, синтаксисом языка, так как именно синтаксис языка программирования определяет, является ли некоторая последовательность символов программой на этом языке. И если Вы уже знакомы, например, с языком Pascal, то при изучении C++ Вы, без сомнения, найдете немало отличий.

Видимо, трудно поспорить с тем, что любая программа на любом языке программирования представляет собой последовательность символов. Однако не все символы является допустимыми в том или ином языке программирования. Множество допустимых символов называется алфавитом языка и только из них может быть составлена программа.

В C++ *алфавит* языка включает заглавные и малые латинские буквы, арабские цифры и ряд специальных символов:

+, -, *, /, %, <, >, =, |, &, \, ', '', @, #, $, ^, ?, _, : , ; , . , (,), [,], {, }, !.

Не следует, однако, специально пытаться выучить их сейчас – по мере изучения материала вы их и так запомните. Однако сами по себе символы алфавита ничего не значат. Минимальными единицами, имеющими в программе какой-либо смысл, являются базовые элементы языка, называемые также *лексемами*. Базовых типов лексем несколько:

- идентификаторы (например, counter, myfunction),
- ключевые (зарезервированные) слова (например, for, while),
- константы (например, 123, 4.5, “abc”),
- операции (например, «+», «-», «&&»),
- разделители (например, «;», «,»).

1.2 Идентификаторы

Идентификаторы в C++ начинаются с большой или малой буквы латинского алфавита и состоят из букв, цифр, знака подчеркивания и используются при объявлении переменных, функций, пользовательских типов данных и т.д.

1.3 Ключевые слова

Ключевые слова языка состоят из тех же самых символов, что и идентификаторы, однако идентификаторы не могут совпадать с ключевыми словами. Другими словами, вы не можете

объявить идентификатор, совпадающий с ключевым словом, так как трактоваться он будет именно как ключевое слово. Приведем список ключевых слов языка C++ с краткими комментариями.

asm – вставка в программу кода на языке ассемблера;

auto – спецификатор, определяющий время жизни локальной переменной;

bool – булев тип данных;

break – прерывание выполнения цикла или оператора выбора;

case – обозначает начало ветви в операторе выбора;

catch – обозначает начало обработчика исключений;

char – символьный тип данных;

class – объявление класса;

const – модификатор, показывающий, что переменная (параметр) неизменяем или объявляемый метод является константным;

const_cast – оператор приведения типа;

continue – прерывание текущей итерации цикла и переход к следующей итерации;

default – обозначает ветвь по умолчанию в операторе выбора;

delete – освобождение динамической памяти;

do – оператор цикла с постусловием;

double – вещественный тип данных с двойной точностью;

dynamic_cast – оператор приведения типа;

else – альтернативная ветвь в условном операторе;

enum – объявление перечислимого типа;

explicit – спецификатор, запрещает вызов конструктора при неявных преобразованиях;

extern – спецификатор, показывающий что реализация функции или хранение переменной осуществляется в другом программном модуле;

false – встроенная константа булева типа;

float – вещественный тип данных;

for – оператор цикла;

friend – объявление функции или класса дружественным;

goto – оператор безусловного перехода;

if – условный оператор;

inline – объявление встраиваемой функции;

int – целочисленный тип данных;

long – целочисленный тип данных (модификатор);

mutable – спецификатор поля, которое может изменяться константными методами;

namespace – определение нового пространства имен;

new – выделение динамической памяти;

operator – перегрузка оператора;

private – объявление закрытых членов класса;

protected – объявление защищенных членов класса;

public – объявление открытых членов класса;

register – спецификатор, указывающий, что переменная должна храниться в регистре процессора;

reinterpret_cast - оператор приведения типа;

return – оператор выхода из функции, возможно, с возвратом значения;

short - целочисленный тип данных (модификатор);

signed – знаковый тип данных (модификатор);

sizeof – операция определения размера результата выражения или типа в байтах;

static – спецификатор статической переменной или статического члена класса;

static_cast – оператор приведения типа;

struct – объявление структуры (записи);

switch – оператор выбора;

template – объявление семейства параметризованных классов или функций (шаблонов);

this – указатель на объект класса, относительно которого выполняется функция;

throw – генерация исключения или описание исключений, выбрасываемых функцией;

true - встроенная константа булева типа;

try – начало блока операторов, в котором могут произойти исключения;

typedef – определение нового типа данных или синонима существующего типа;

typeid – операция определения типа во время выполнения;

typename – указывает, что идентификатор является типом (в шаблонах);

union – объявление объединения;

unsigned - беззнаковый тип данных (модификатор);

using – директива, позволяющая использовать или имена, определенные в каком-либо пространстве имен без явных префиксов (или отдельное имя);

virtual – объявление виртуального метода;

void – специальный пустой тип данных (при объявлении функций указывает на отсутствие параметров или возвращаемого значения, используется для объявления нетипизированного указателя);

volatile – показывает, что переменная или объект может изменяться каким-либо внешним процессом (прерыванием, другим потоком и т.п.);

wchar_t – расширенный символьный тип данных;

while – оператор цикла с предусловием.

Пугаться столь внушительного списка не стоит, ровно как не стоит сейчас пытаться полностью понять все комментарии к ключевым словам. По мере освоения материала Вы в деталях узнаете назначение многих из них. А с некоторыми (auto, explicit, mutable, register, volatile), вполне возможно, не встретитесь на практике никогда. В любом случае, Вы всегда сможете вернуться к приведенному списку, если почувствуете в этом необходимость.

Говоря о ключевых словах языка, нелишне будет сказать о том, что производители средств разработки дополняют стандартный набор ключевых слов своими словами. Часто ключевые слова из таких расширенных наборов начинаются со знаков подчеркивания. Информацию о них Вы, скорее всего, сможете найти в справочной системе вашей среды разработки.

1.4 Типы данных

Все *типы*, используемые в C++ можно разделить на две большие группы: *базовые* (или встроенные) и *производные* (или определяемые пользователем). Мы в настоящем разделе рассмотрим только базовые типы. Производные типы мы будем рассматривать по мере изучения материала.

Типы данных C++ можно представить в виде таблицы 1.1.

Базовые типы данных включают в себя особый тип void и *арифметические* типы данных, к которым относятся как интегральные (целочисленные), так и вещественные (с плавающей запятой) типы.

Тип *void* имеет пустое множество значений. Объявить в программе объект такого типа нельзя. Вместо этого этот тип используется при объявлении функций, чтобы указать, что возвращаемое значение отсутствует, а также при объявлении нетипизированных (свободных) указателей. Мы обязательно рассмотрим эти вопросы в рамках пособия.

К *интегральным* типам относят как целые типы данных, так и символьные типы, и логический тип. Тот факт, что логический и символьные типы относят к интегральным типам данных, делает возможным их использование в арифметических выражениях наравне с целыми типами.

Таблица 1.1 – Типы данных

<table>
<tr><th colspan="5">Тип</th><th>Размер</th></tr>
<tr><td rowspan="10">Базовые</td><td colspan="4">Пустой (void)</td><td>-</td></tr>
<tr><td rowspan="9">Скалярные</td><td rowspan="9">Арифметические</td><td rowspan="6">Интегральные</td><td>Логический (bool)</td><td>1</td></tr>
<tr><td>Символьный (char)</td><td>1</td></tr>
<tr><td>Целый (short)</td><td>2</td></tr>
<tr><td>Целый (int)</td><td>4</td></tr>
<tr><td>Целый (long)</td><td>4</td></tr>
<tr><td>Расширенный символьный (wchar_t)</td><td>2</td></tr>
<tr><td rowspan="3">Вещественные</td><td>float</td><td>4</td></tr>
<tr><td>double</td><td>8</td></tr>
<tr><td>long double</td><td>8</td></tr>
<tr><td rowspan="7">Производные</td><td rowspan="3">Скалярные</td><td colspan="3">Перечисления (enum)</td><td>4</td></tr>
<tr><td colspan="3">Указатели (тип *)</td><td>4</td></tr>
<tr><td colspan="3">Ссылки (тип &)</td><td>4</td></tr>
<tr><td rowspan="4">Составные</td><td colspan="3">Массивы</td><td rowspan="4"></td></tr>
<tr><td colspan="3">Структуры (struct)</td></tr>
<tr><td colspan="3">Объединения (union)</td></tr>
<tr><td colspan="3">Классы (class)</td></tr>
</table>

Целые типы данных в языке C++ представлены в трех вариантах: «короткие» целые (short int), целые (int) и «длинные» целые (long int). Каждый из типов определяет занимаемый числом объем памяти и, соответственно, диапазон возможных значений. Использовать int при объявлении short int и long int вовсе необязательно, то есть short int и short – синонимы. Каждый из трех целых типов может предваряться модификатором signed или unsigned, показывающим, будет ли трактоваться целое число, как знаковое или беззнаковое. Использование модификатора signed – дело вкуса, так как по умолчанию целые числа трактуются как числа со знаком. Кроме того, использование ключевого слова int при описании signed int или unsigned int также не является обязательным. Можно записать просто signed или unsigned. Заметим, что беззнаковые целые числа рекомендуется использовать с осторожностью там, где это действительно необходимо (например, при работе с отдельными битами). Использование беззнаковых чисел в арифметических выражениях, особенно совместно со знаковыми, может приводить к трудноуловимым ошибкам, связанным с неявным преобразованием чисел к знаковым.

Переменные *логического* типа bool могут принимать одно из двух значений: true (истина) или false (ложь). Значения типа bool допускают преобразование к целому типу. true преобразуется в 1, false – в 0. При обратном преобразовании 0 преобразуется в false, любое ненулевое значение – в true. Тип bool используется при вычислении выражений с логическими операциями, операциями сравнения.

Символьный тип char позволяет оперировать символами из используемого набора (например, ASCII). Так как каждому символу типа char соответствует целое число (код символа) и в C++ допускается неявное преобразование между целыми типами и char, то граница между char и целыми типами стирается. Этот факт позволяет рассматривать тип char как целый однобайтовый тип данных. Этому обстоятельству способствует также тот факт, что с типом char допускается использование спецификаторов signed и unsigned. Этот же факт, однако, вносит некоторую путаницу относительно того, какие же все-таки значения должны ставиться в соответствие набору символов – знаковые или беззнаковые.

Расширенный символьный тип данных оперирует символами из больших наборов (например, Unicode). Детальное рассмотрение особенностей, связанных с этим типом данных выходит за рамки настоящего пособия. Скажем лишь, что к этому типу неприменимы спецификаторы signed и unsigned, а значения этого типа могут использоваться в выражениях наравне со значениями других интегральных типов.

Вещественные числа представлены в C++ тремя типами: float, double и long double, называемыми соответственно числами с одинарной, двойной и расширенной точностью. Тип float используется большей частью тогда, когда стремятся минимизировать размер передаваемых (хранимых) вещественных данных или обеспечить совместимость со старым программным кодом. При создании новых приложений в том случае, если не предъявляются жесткие требования к размеру хранимой или передаваемой информации, лучше использовать тип double.

Отметим, что базовые типы языка C++ не включают привычный для некоторых других языков (например, Pascal) строковый тип данных. Размеры базовых типов данных и диапазоны возможных значений приведены в следующей таблице.

1.5 Константы

Константы в C++, как и в любом другом языке представляют собой заранее определенные значения, используемые в процессе выполнения программы. В C++ используются целочисленные, символьные, вещественные и строковые константы.

Целочисленные константы могут быть заданы в десятичной, восьмеричной и шестнадцатеричной системе счисления. При этом, если константа начинается с «0x» или «0X», то она считается заданной в шестнадцатеричной системе счисления (например, 0x123f, 0XA23FEA), если с лидирующего нуля, то – в восьмеричной (012, 0367), в остальных случаях – в десятичной системе счисления (12, 367). Будьте внимательны и не ставьте лидирующих нулей перед константами, записанными вами в десятичной системе счисления.

Как Вы уже знаете, в языке C++ определено несколько целочисленных типов. С использованием суффиксов при объявлении констант Вы можете контролировать, какого именно целочисленного типа будет ваша константа. Например, запись 0676456UL будет означать беззнаковую (суффикс U) целочисленную константу типа long (суффикс L), заданную в восьмеричной системе счисления. Мы не будем останавливаться на этом вопросе подробно, достаточно того, что Вы теперь знаете, что такая возможность есть.

Символьные константы записываются в одинарных кавычках, например: ‘a’, ‘A’, ‘@’. Так как некоторые символьные константы (например, перевод строки, возврат каретки) задать таким способом невозможно, то используются так называемые escape-последовательности. В них для обозначения символов используется обратная косая черта, за которой следуют определенные символы. Например, константа, обозначающая символ табуляции запишется так: ‘\t’. Возможные символы escape-последовательности приведены в таблице 1.2.

Вообще говоря, переменной символьного типа можно присваивать и целочисленные значения, которые в этом случае будут обозначать коды символов. Символьные константы для расширенного набора символов предваряет буква L, записываемая перед открывающей одинарной кавычкой.

Вещественные константы содержат мантиссу, а также могут содержать экспоненциальную часть. Мантисса представляет собой необязательный знак «-», целую часть числа, десятичную точку и дробную часть числа. При наличии экспоненциальной части десятичная точка может отсутствовать (в том случае, если число не содержит дробной части). Собственно экспоненциальная часть начинается с буквы «E» (или «e»), за которой следует необязательный знак минус и порядок. Порядок имеет смысл десятичного порядка, то есть мантисса умножается на 10 в степени указанного порядка. Приведем несколько примеров вещественных констант: 1.2, -0.09, 1.73E-10.

Строковые константы представляют собой последовательность символов, возможно содержащую escape-последовательность и заключенную в двойные кавычки. В том случае, если необходимо разорвать строковую константу в связи с форматированием текста программы или по каким-то другим причинам, можно воспользоваться двумя способами. Во-первых, единую

строковую константу (например, “abcdef”) можно разорвать, закрыв и заново открыв двойные кавычки (“abc” “def”). При этом части строки могут находиться на разных строках. Во-вторых, разорвать строку можно, поставив в конце строки обратную косую черту. В этом случае двойные кавычки ставятся только в самом начале и в самом конце константы:

```
“abc\
def”
```

Логические константы в C++ представлены двумя ключевыми словами: true (истина) и false (ложь).

Таблица 1.2 - Символы escape-последовательности

Последовательность	Код символа	Описание
\n	10	Перевод строки
\t	9	Горизонтальная табуляция
\v	11	Вертикальная табуляция
\b	8	Возврат на позицию с удалением символа
\r	13	Возврат каретки
\f	12	Перевод страницы
\a	7	Звуковой сигнал
\\	92	Обратная косая черта
\?	63	Знак вопроса
\'	39	Одинарная кавычка
\"	34	Двойная кавычка
\0	0	Символ с нулевым кодом
\ooo	Задание кода символа в восьмеричной системе счисления	
\xhhh	Задание кода символа в шестнадцатеричной системе счисления	

1.6 Операции

Операции в C++ обозначаются, большей частью, специальными символами. Однако некоторые из операций обозначаются ключевыми словами. Операции в C++ можно разбить на группы, представленные в таблице 1.3.

Таблица 1.3 - Операции в C++

Операция	Описание	Приоритет	Ассоциативность
Арифметические			
-	унарный минус	3	правая

+	унарный плюс	3	правая
*	умножение	5	левая
/	деление	5	левая
%	остаток от деления	5	левая
+	сложение	6	левая
-	вычитание	6	левая
++	префиксный инкремент	3	правая
--	префиксный декремент	3	правая
++	постфиксный инкремент	2	левая
--	постфиксный декремент	2	левая
Логические			
&&	логическое И	13	левая
||	логическое ИЛИ	14	левая
!	логическое отрицание	3	правая
Битовые			
&	побитовое И	10	левая
|	побитовое ИЛИ	12	левая
^	побитовое исключающее ИЛИ	11	левая
~	дополнение до единицы	3	правая
<<	сдвиг влево	7	левая
>>	сдвиг вправо	7	левая
Сравнение			
<	меньше	8	левая
<=	меньше или равно	8	левая
>	больше	8	левая
>=	больше или равно	8	левая
==	равно	9	левая
!=	не равно	9	левая
Присваивание			
=	простая операция присваивания	16	правая
Составные операции присваивания:			
+= – = *= /= %=	арифметические	16	правая

&= |= ^= <<= >>=	битовые	16	правая
Указатели и работа с памятью			
new	выделение памяти	3	правая
delete	освобождение памяти	3	правая
*	разыменование указателя	3	правая
&	взятие адреса	3	правая
sizeof	определение размера	3	правая
Работа с типами			
()	приведение типов	3	правая
typeid	динамическая идентификация типа	2	левая
static_cast	приведение типа с проверкой на этапе компиляции	2	левая
dynamic_cast	приведение типа с проверкой во время выполнения	2	левая
reinterpret_cast	приведение типа без проверки	2	левая
const_cast	снятие или добавление const / volatile	2	левая
Операции доступа			
::	расширение области видимости	1	-
[]	обращение по индексу	2	левая
.	доступ к члену класса или структуры	2	левая
->	доступ к члену класса или структуры по указателю	2	левая
.*	доступ к члену класса по указателю на член класса через объект	4	левая
->*	доступ к члену класса по указателю на член класса через указатель на объект	4	левая
Другие операции			
()	вызов функции	2	левая
?:	тернарная условная операция	15	правая
throw	генерация исключения	17	правая
,	запятая	18	левая

В приведенной таблице операции разбиты на группы, и для каждой операции указан ее приоритет и ассоциативность. *Приоритет* операции указывает порядок выполнения операций в выражении (в приведенной таблице, чем меньше значение приоритета, тем он выше). *Ассоциативность* операции показывает направление вычислений в выражении, когда операции имеют одинаковый приоритет. При этом для левоассоциативных операций вычисление выражения с участием операции происходит слева-направо, а для правоассоциативных - справа-налево. Очередность

В настоящем разделе мы не будем рассматривать все операции, а рассмотрим лишь часть из них. Начнем мы с *арифметических* операций. Бинарные арифметические операции умножения (*), деления (/), вычисления остатка от деления (%), сложения (+) и вычитания (-) работают вполне естественным образом. В том случае, если при выполнении этих операций хотя бы один из аргументов – вещественный, то операция выполняется над вещественными числами. Однако, в том случае, если оба аргумента целые - то выполняется операция над целыми числами. Это означает, в частности, что операция деления над целыми числами выполняется, как операция целочисленного деления, то есть 3/2 даст в результате 1, а не 1.5, даже если результат деления присваивается вещественной переменной. Если же Вы хотите получить в результате операции вещественное число, приведите хотя бы один из аргументов к вещественному типу.

Операции префиксного и постфиксного инкремента и декремента (++ и --) вызывают обычно повышенный интерес начинающих программировать на C++. Эти операции выполняют увеличение (++) или уменьшение (--) аргумента на единицу. Все эти операции возвращают значения, и, поэтому, могут участвовать в выражениях. Разница между постфиксными и префиксными операциями состоит в том, когда именно увеличивается или уменьшается значение аргумента операции при вычислении значения выражения. Для префиксных операций сначала выполняется инкремент или декремент, а уже затем новое значение участвует в вычислении выражения. Для постфиксных операций, наоборот, сначала значение аргумента используется при вычислении выражения, а уже затем выполняется инкремент или декремент. Рассмотрим два примера

```
x=2;                          x=2;
y=10 * ++x + 5;               y=10 * x++ + 5;
```

В первом случае сначала будет выполнена префиксная операция инкремента ++x и x примет значение 3, а уже затем будет рассчитано выражение и переменной y будет присвоено значение 35. Во втором случае сначала будет произведен расчет выражения и будет получено значение 25, и только затем будет выполнена постфиксная операция инкремента.

Перейдем к следующей группе – *логическим* операциям. Все логические операции (&&, ||, !) неявно приводят аргументы к булевому типу (при этом аргументы должны быть обязательно скалярного типа: целочисленного, указательного или булевого) и возвращают булев тип. Действие операторов можно показать в виде следующей таблицы.

Таблица 1.4 – Логические операции

аргументы	**логическое И** **x && y**	**логическое ИЛИ** **x || y**	**логическое отрицание** **! x**
x=false, y=false	false	false	true
x=false, y=true	false	true	true
x=true, y=false	false	true	false
x=true, y=true	true	true	false

Отметим, что аргументами для рассматриваемых операций могут быть, в том числе и подвыражения. И так как в некоторых случаях результат операции ясен уже после вычисления выражения для первого операнда, то вычисление второго операнда не производится. Такое возможно при вычислении логического «И», когда первый аргумент – ложь, или при вычислении логического «ИЛИ», когда первый аргумент - истина. Этот факт позволяет не только экономить время при вычислениях, но также используется для управления порядком вычисления подвыражений, что бывает очень полезно, если в условиях встречается разыменование указателей.

Битовые операции &, |, ^, ~ работают с целочисленными аргументами, побитово вычисляя результирующее значение следующим образом:

- побитовое «И»: если соответствующие биты операндов равны 1, то соответствующий бит результата устанавливается равным 1, в противном случае - 0.

- побитовое «ИЛИ»: если хотя бы один из соответствующих битов операндов равен 1, то соответствующий бит результата устанавливается равным 1, в противном случае - 0.

- побитовое «исключающее ИЛИ»: если соответствующие биты операндов равны друг другу, то соответствующий бит результата устанавливается равным 0, в противном случае - 0.

- дополнение до единицы: если бит операнда равен 1, то соответствующий бит результата устанавливается равным 0, в противном случае - 1.

Операции битового сдвига (<<, >>) осуществляют битовый сдвиг левого аргумента на число бит, указанное в правом аргументе. При сдвиге освобождающиеся биты заполняются нулями, то есть сдвиг – нециклический. При сдвиге вправо для знаковой величины должно осуществляться заполнение освобождающихся битов значением знакового разряда.

Операции битового сдвига часто используются для выполнения умножения и деления на степень двойки. Так, умножение x*16 можно выполнить с использованием битового сдвига: x<<4. Деление y/8 можно выполнить так: y>>3.

Пример использования битовых операций представлен в виде таблицы.

Таблица 1.5 – Битовые операции

	x & y	**x | y**	**x ^ y**	**~ x**	**x << 1**	**x >> 1**
x	11110101	11110101	11110101	11110101	11110101	11110101
y	10101100	10101100	10101100			
результат	10100100	11111101	01010110	00001010	11101010	01111010

Говоря о логических и битовых операциях нелишне будет сказать о том, что внешняя схожесть самих обозначений часто приводит к ошибкам в программном коде. Пусть, например, x=0xE0, y=0x0A, тогда операция x&&y даст в результате true (истину), а вычисление x&y даст 0x00, то есть ложь. Будьте внимательны и старайтесь не путать эти операции при программировании.

Операции *сравнения* (<, <=, >, >=, ==, !=) могут выполняться над скалярными величинами. Результатом операции является булево значение true или false. При выполнении операций над числовыми аргументами, они дают вполне ожидаемый результат. Случай, когда аргументами операции являются указатели, мы рассмотрим чуть позже, когда будем говорить об этом типе данных.

Таблица 1.6 – Операции сравнения

x<y	Операция возвращает true, если x меньше y. В противном случае - false.
x>y	Операция возвращает true, если x больше y. В противном случае - false.
x<=y	Операция возвращает true, если x меньше или равно y, иначе - false.
x>=y	Операция возвращает true, если x больше или равно y, иначе - false.
x==y	Операция возвращает true, если x равно y. В противном случае - false.
x!=y	Операция возвращает true, если x неравно y. В противном случае - false.

Операций *присваивания* в C++ несколько. Существует простая операция присваивания (=) и составные операции присваивания (+=, –=, *=, /=, %=, &=, |=, ^=, <<=, >>=). Простая операция присваивания сохраняет значение правого операнда в объект, обозначаемый левым операндом. Составные операции присваивания сперва выполняют над операндами соответствующую арифметическую или битовую операция, а уже затем присваивание. Составные операции присваивания имеют вид <op>=, где <op> – бинарная арифметическая или битовая операция.

Отличительной особенностью всех операций присваивания в C++ является то, что эти операции возвращают новое значение левого операнда. Это позволяет использовать операции присваивания в различных выражениях, получая весьма компактный программный код. Например, выражение a=b=c=0 позволяет обнулить три переменные a,b,c. Такая возможность, однако, нередко становится источником ошибок в программном коде. Например, выражение (a=x) будучи по ошибке написанным где-то в условии вместо (a==x) будет не только присваивать содержимое x переменной a, но и возвращать ненулевое (истинное) значение в том случае, если новое значение a будет ненулевым вместо того, чтобы сравнивать a и x.

Приведем несколько примеров:

```
x=a+=2*(b+=3)
```

Пусть a=2, b=3. Тогда сначала переменная b будет увеличена на 3 и примет значение 6, затем значение a будет увеличено на 12 и примет значение 14, а уже затем новое значение a будет помещено в x, и x примет значение 14.

```
x=(a+=2)*(b*=3)
```

Пусть по-прежнему a=2, b=3. Сначала переменная a будет увеличена на 2 и примет значение 4, затем b будет умножено на 3 и примет значение 9, затем новые значения a и b будет перемножены и результат помещен в x. x примет значение 36.

```
x=(a+=2)*=3
```

Здесь сначала переменная a будет увеличена на 2 и примет значение 4, затем новое значение будет умножено на 3, и a примет значение 12, которое и будет помещено в x.

Безусловно, поупражняться в составлении таких конструкций весьма интересно, но в практической работе следует помнить, что код должен быть не только компактен, но и понятен, поэтому не стоит злоупотреблять ими.

1.7 Описание переменных

Конечно, ни одна сколь-нибудь значимая программа не обходится без переменных. Объявление переменных базового типа выглядит следующим образом:

```
⟦<спецификатор>⟧ ⟦const⟧ <тип> <идентификатор> ⟦<инициализатор>⟧
                                  ⟦,<идентификатор> ⟦<инициализатор>⟧
                                   , … ⟧;
```

В этой записи <спецификатор> - необязательный спецификатор класса памяти (речь о таких спецификаторах пойдет чуть позже). const – необязательный модификатор, показывающий, что значение переменной нельзя будет изменить. Вообще говоря, модификатор const может быть расположен и после типа:

```
⟦<спецификатор>⟧ <тип> const идентификатор
```

Его действие в обоих случаях будет одинаково.

Необязательный <инициализатор> задает значение, которое получит переменная при инициализации. При этом начальное значение переменной указывается после знака равенства:

⟦<спецификатор>⟧ ⟦const⟧ <тип> <идентификатор> =<начальное значение>

Иногда можно встретить другой формат инициализации:

⟦<спецификатор>⟧ ⟦const⟧ <тип> <идентификатор>(<начальное значение>)

Это инициализатор в стиле функции, который с простыми типами данных используется достаточно редко.

Вы можете объявить и, если необходимо проинициализировать несколько переменных одного типа, перечисляя их через запятую. Приведем несколько примеров объявления переменных.

```
char c1, c2='A', c3('\t');
bool b, f=true;
float f=1.2;
double g, h=-0.09;
```

Переменные можно описывать практически где угодно. Глобальные переменные описываются в тексте программы вне каких-либо пользовательских типов и функций. Примером могут служить приведенные выше объявления. Локальные переменные описываются непосредственно внутри тела функции, например:

```
void func()
{
    int a, b;
    {
        int a;  // другая переменная a
        // некоторый код
    }
    {
        int a;  // третья переменная a
        // некоторый код
    }
    // некоторый код
}
```

Здесь же стоит сказать о том, что в приведенном выше примере все три переменные a – различные переменные, объявленные в разных областях видимости, ограниченных операторными скобками.

1.8 Классы памяти

При описании переменных Вы можете указать один из четырех спецификаторов класса памяти: auto, register, static и external.

Спецификатор ***auto*** указывает компилятору на то, что объявляется автоматическая (локальная) переменная. Такая переменная размещаются на стеке и ее время жизни ограничено временем исполнения того блока операторов, в котором она определена. Инициализация

автоматических переменных не выполняется, если инициализатор не указан явным образом. Если инициализатор указан, инициализация выполняется всякий раз, когда поток исполнения команд проходит через объявление переменной. Можно считать, что память, занимаемая под переменную, выделяется при входе в блок, в котором она объявлена, и освобождается при выходе из блока. Следует отметить, что в явном виде спецификатор используется крайне редко, так как все переменные, объявленные внутри функций и любых операторных блоков, по умолчанию являются автоматическими.

Спецификатор ***register*** указывает компилятору на то, что объявляемую переменную желательно разместить в регистре процессора. Этот спецификатор используется для целей оптимизации и может быть проигнорирован компилятором. В последнем случае, переменная будет автоматической.

Спецификатор ***static*** означает, что объявляется статическая переменная. Время жизни такой переменной определяется временем работы программы. Память под переменную выделяется при запуске программы и освобождается при ее завершении. Значения статических переменных объявленных внутри функции, сохраняется между вызовами функций. Инициализация таких переменных выполняется один раз в соответствии с инициализатором, если таковой указан. Если инициализатор не указан, то переменая инициализируется нулевыми значениями. Вообще говоря, любая глобальная переменная (описанная на уровне файла вне каких-либо функций), имеет время жизни статической переменной. Для таких переменных спецификатор static влияет только на область видимости переменной. Переменная, объявленная, как static, будет видна только в том файле, в котором она объявлена.

Спецификатор ***extern*** говорит о том, что объявленная переменная является внешней. Внешняя переменная может определена в другом месте того же файла или в другом файле. Внешняя переменная имеет статическое время жизни. По умолчанию все глобальные переменные (описанная на уровне файла вне каких-либо функций) являются внешними. По сути, данный модификатор показывает, что внешняя переменная будет использована, но память под нее будет выделена где-то в другом месте, в котором она будет определена.

```
int f()
{
    static int callCount;     // Объявляем статическую переменную
                              // По умолчанию инициализируется нулем
    callCount ++;
    return callCount;         // Возвращаем номер вызова 1,2,3,…
}
double weight(double m)
{
    double res;          // Объявляем автоматическую переменную
    extern double g;     // Будем использовать внешнюю переменную
```

```
    res= m*g;
    return res;
}
double g = 9.8;     // Переменная по умолчанию является внешней
                    // и может быть определена в другом модуле
```

1.9 Вопросы и задания

1. Назовите базовые элементы языка.

2. Какие цепочки символов являются корректными идентификаторами?

3. Назовите базовые и производные типы данных. Какие ключевые слова используются при описании базовых типов данных?

4. Дайте краткую характеристику для каждого из базовых типов данных.

5. Для приведенных ниже цепочек символов скажите, являются ли они корректными константами. Если да, то назовите тип константы.

1.75; 4.3.4; “false”, 0, ‘1E1’; -1E1.3; ‘\a’, ‘/t’, 5778, 0x1UL, “0123”, abcd, 0xabcd, 0xefgh, true.

6. Укажите порядок выполнения операций в следующих выражениях:

a) x+y*z/v%w

b) x&y&&w&v

c) a=--y*x++

d) a=b==c

e) x=i++,j++

f) x=y=a>=b==c<d

7. Какие спецификаторы классов памяти Вы знаете? Каким образом различается область видимости и время жизни переменных объявленных с этими спецификаторами?

2. УПРАВЛЯЮЩИЕ ИНСТРУКЦИИ ЯЗЫКА

Мы обсудили базовые элементы языка, настало время поговорить об операторах C++. Начнем мы, конечно же, с управляющих инструкций C++. Управляющие инструкции можно встретить в каждой программе в большом количестве. Именно они дают нам возможность управлять ходом вычислений в программе. В C++ имеется весь "джентельменский" набор таких инструкций:

- условный оператор (оператор ветвления if)
- тернарная условная операция (операция «?:»)
- оператор цикла с предусловием (while)
- оператор цикла с постусловием (do ... while)
- цикл for
- оператор множественного выбора (переключатель switch)
- оператор безусловного перехода (goto)
- оператор досрочного выхода из цикла или оператора множественного выбора (break)
- оператор пропуска итерации (continue)
- операция вызова функции (операция "()")
- оператор выхода из функции (return)

Будучи знакомыми с каким - либо другим языком программирования, Вы наверняка встретили среди приведенных операторов уже знакомые Вам. Мы, однако, не будем торопиться и последовательно все их рассмотрим.

2.1 Условный оператор

Начнем мы, конечно же, с уже известного Вам условного оператора. Этот оператор реализует выполнение команды (или группы команд) при условии, что некоторое условие принимает истинное (ненулевое) значение. Опционально может быть выполнена другая команда (или группа команд), если условие принимает ложное (нулевое) значение.

Синтаксис оператора выглядит следующим образом:

```
if (<условие>) <оператор>;
⟦else <оператор>;⟧
```

Следует отметить, что в случае, когда вместо одной команды записывается группа команд, необходимо использовать операторные скобки, роль которых в C++ играют фигурные скобки «{}».

Говоря о синтаксисе оператора if, следует обратить внимание на следующие особенности:

- условие всегда записывается в круглых скобках;

- исполняемый оператор следует непосредственно за условием (между ними отсутствуют какие-либо ключевые слова, например then);

- перед альтернативной ветвью else ставится ";" (исключение составляют случаи, когда перед else стоят операторные скобки).

На этом, однако, все отличия и заканчиваются. Мы часто будем пользоваться этим оператором, поэтому приведем здесь лишь пару небольших примеров:

```
if (a<b) a = b+1; else a = b-1;
if (a<b) c=a; else c=b;
```

2.2 Тернарная условная операция

Тернарная условная операция может заменить условный оператор в случае выполнения сравнительно простых вычислений. Результат этой операции может непосредственно использоваться в выражениях, что способствует созданию более компактного программного кода. Однако, в подавляющем большинстве случаев та же функциональность может быть достигнута и при использовании условного оператора if.

Синтаксис тернарной условной операции выглядит следующим образом:

```
<выражение 1> ? <выражение 2> : <выражение 3>
```

Синтаксис такой операции может показаться немного непривычным, однако, начав ее использовать, Вы вскоре поймете, что это удобно. Пусть например, для расчета некоторого значения требуется вычислить минимальное значение двух других величин (x=a+b*min{c,d}). С использованием тернарной условной операции такой расчет будет выглядеть следующим образом:

```
x = a +b * (c<d ? c: d);
```

В качестве упражнения попробуйте произвести те же вычисления с использованием оператора if.

2.3 Цикл с предусловием

Рассмотрим теперь одну из наиболее часто используемых управляющих конструкций языка - оператор цикла с предусловием. Этот оператор предназначен для организации многократного исполнения какого - либо оператора (или набора операторов) в зависимости от значения, которое принимает выражение, стоящее в условии цикла. При этом цикл выполняется, пока условие истинно.

Синтаксис оператора выглядит следующим образом:

```
while (<условие>) <оператор>;
```

Заметим, что здесь условие цикла обязательно заключается в круглые скобки, а тело цикла следует непосредственно за условием (без привычного для знакомых с языком pascal ключевого слова do).

Приведем несколько примеров использования оператора.

Подсчет суммы от 1 до 10:

```
i = 1; s = 0;
while (i<=10) s += i;
```

Подсчет суммы первых десяти элементов массива a:

```
i = 0; s = 0;
while (i<10) s += a[i];
```

Организация "бесконечного" цикла:

```
while (1) ;
```

Следующий цикл не выполнится ни разу

```
while (0) ;
```

2.4 Цикл с постусловием

Оператор цикла с постусловием имеет в C++ следующий синтаксис.

```
do <оператор> while (<условие>)
```

Основная разница с циклом while с предусловием состоит в том, что условие вычисляется после итерации цикла, а не перед ней. Из этого следует, что цикл выполнится хотя бы один раз, даже если условие ложно еще до выполнения цикла.

Отметим также, что несмотря на внешнюю схожесть с циклом repeat ... until ... языка pascal, цикл do ... while ... языка C++ будет выполняться пока условие цикла принимает истинное значение.

Организация "бесконечного" цикла:

```
do while (1) ;
```

Следующий цикл выполнится один раз:

```
do while (0) ;
```

2.5 Цикл for

Цикл for также является одной из самых часто используемых управляющих конструкций. В C++ цикл for имеет следующий синтаксис:.

```
for (<инициализация>; <условие>; <модификация>) <оператор>;
```

Здесь <инициализация>, <условие> и <модификация> представляют собой по сути необязательные выражения произвольного вида. При этом выражение <инициализация> вычисляется один раз до начала выполнения цикла. Выражение <условие> рассчитывается перед каждой итерацией цикла. Итерация цикла выполняется, если

значение истинно, если нет - цикл завершается. Выражение <модификация> рассчитывается после каждой итерации цикла.

Таким образом, в отличие от ряда языков программирования, в которых цикл for представляет собой цикл со счетчиком, в C++ этот цикл представляет собой фактически цикл с предусловием. Эквивалентный ему фрагмент цикла while выглядит следующим образом:

```
<инициализация>
while (<условие>) {
     <оператор>
     <модификация>)
}
```

В том случае, если какое-либо выражение из заголовка цикла for отсутствует, оно подразумевается равным единице. То есть следующие три записи эквивалентны и представляют собой "бесконечный" цикл:

```
for (  ;  ;  );
for (  ; 1;  );
for ( 1; 1; 1);
```

Приведем несколько примеров.

Подсчет суммы от 1 до n:

```
s = 0;
for ( i = 1; i<=n; i++ ) s += i;
```

Этот же фрагмент может быть записан следующим образом

```
for ( i = 1, s = 0; i<=n; s += i++ );
```

Следующий цикл не выполнится ни разу

```
for ( ; 0; );
```

Как видите, синтаксис оператора позволяет писать довольно изощренные конструкции. Иногда применение их бывает оправдано, но не следует забывать, что написанный Вами код должен быть понятен не только Вам, но и другим людям.

2.6 Операторы break и continue

В C++ имеются для управления вычислениями при использовании циклов используются еще два оператора: break и continue**.** Оператор break вызывает выход из цикла, выполняемого в настоящий момент. Этот оператор позволяет прервать выполнение цикла в том случае, когда условие выхода ещё не достигнуто.

Оператор continue вызывает пропуск оставшихся операторов и вызывает немедленный переход к следующей итерации. Этот оператор используют, когда необходимо пропустить исполнение всех команд до конца тела цикла.

Формат использования этих операторов предельно прост:

```
break;
continue;
```

Несмотря на то, что многие сторонники классического структурного подхода к программированию считают, что использование операторов break и continue является свидетельством плохой проработки алгоритмов, взгляд автора на использование этих операторов менее консервативен. Во многих случаях использование break или continue позволяет получить более компактный и понятный код. Приведем несколько примеров. Рассмотрим вычисление суммы положительных элементов массива.

```
for (int i=0; i<count; i++ )
  if (a[i]>0) s+=a[i];
```

```
for (int i=0; i<count; i++) {
  if (a[i] <= 0)  continue;
  s += a[i];
}
```

Конечно же, в этом случае следует предпочесть первое из двух приведенных решений. Однако, уже в следующем примере выбор уже не так очевиден:

```
for (int i=0; i<count; i++) {
  value1 = func1(i);
  if ( value1 >= 0 ) {
    value2 = func2(value1);
    if (value2 >= 0)  {
      value3=func3(value2)
      if (value3 >= 0)
        s+=value1/value2;
    }
  }
}
```

```
for (int i=0; i<count; i++) {
  value1 = func1(i);
  if ( value1 < 0 ) continue;
  value2 = func2(value1);
  if (value2 < 0) continue;
  value3 = func3(value2);
  if (value3 < 0) continue;
  s += value1/value2;
}
```

Предположим теперь, что продолжать суммирование при получении хотя бы одного отрицательного значения в приведенном выше примере бессмысленно:

```
for(int i=0,f=1;f&&i<count;i++)
{
  val1 = func1(i);
  if ( val1 >= 0 ) {
    val2 = func2(val1);
    if (val2 >= 0)  {
      val3 = func3(val2)
      if (val3 >= 0)
        s+= val1/val2+val3;
      else f = 0;
    }
    else f = 0;
  }
  else f=0;
}
```

```
for (int i = 0; i < count; i++)
{
  val1 = func1(i);
  if ( val1 < 0 ) break;
  val2 = func2(val1);
  if (val2 < 0) break;
  val3 = func3(val2);
  if (val3 < 0) break;
  s += val1/val2+val3;
}
```

Конечно же, приведенные выше примеры никоим образом не призваны убедить Вас в том, что рассматриваемые операторы - всегда лучшее решение проблемы. Здесь, как и во многих других случаях, предпочтение следует отдать более понятному и эффективному решению.

2.7 Оператор множественного выбора

Оператор множественного выбора, также называемый иногда переключателем, в C++ выполняет ту же роль, что и условный оператор, однако может иметь множество ветвей. Синтаксис оператора множественного выбора выглядит следующим образом

```
switch (<выражение>)
{
    case <значение 1>; [[<операторы 1>] [break];
  [ case <значение 2>; [[<операторы 2>] [break];
    … ]
  [ default: <операторы N>;]
}
```

При выполнении этого оператора вычисляется значение выражения, стоящего в заголовке оператора и после этого осуществляется исполнение операторов, стоящих в той ветви case, значение для которой совпадает с вычисленным значением выражения. В том случае, если значение выражения не совпадает ни с одним из значений, указанных в ветвях case, будут выполнены операторы, стоящие в необязательной ветви default, которая вовсе не обязательно указывается последней.

При использовании оператора switch следует помнить о нескольких важных моментах. Во-первых, значение выражения, стоящего в заголовке оператора, должно иметь тип, приводимый к целому типу. Во-вторых, в ветвях case указываются либо значения, либо константные выражения, приводимые к целому типу. В-третьих, в том случае, если управление передано в какую-либо из ветвей case, операторы будут выполняться до тех пор, пока не встретится оператор break. Другими словами, если Вы забудете написать break в какой либо ветви Вашего оператора, то при выполнении условия перехода на эту ветвь будут выполнены, как операторы, стоящие в этой ветви, так и операторы следующих за ней ветвей (в том числе и default). Выполнение будет продолжаться независимо от указанных в case значений, пока не встретится оператор break. Такое поведение оператора позволяет в некоторой степени оптимизировать программный код, хотя и является потенциальным источником ошибок в программах.

Приведем несколько примеров

```
switch (s[i])  {
    case ‘X’:
    case ‘x’: xcount++;
    default: i++;
}
switch (p)  {
    case 8: x=x*x;
    case 4: x=x*x;
    case 2: x=x*x;
```

```
    case 1: break;
    case 0: x=1;  break;
    default: x=0;
}
```

2.8 Оператор безусловного перехода

Оператор безусловного перехода имеет формат:

```
goto <метка>;
<метка>: …
```

Использование goto бывает оправдано, например, для решения проблемы освобождения ресурсов (при невозможности использования механизма обработки исключений), для выхода из вложенных циклов при создании высокопроизводительных систем. Тем не менее, в тех случаях, когда это не является действительно необходимым, лучше обходиться без использования этого оператора.

2.9 Операция вызова функции и оператор return

Наконец, мы подошли к рассмотрению такой важной управляющей инструкции, как вызов функции. За вызов функции в C++ отвечает операция вызова функции, обозначаемая круглыми скобками. Для вызова функции указывают имя функции, за которым следуют круглые скобки. В том случае, если необходимо, параметры вызова указываются при вызове в круглых скобках. В том случае, если функция не имеет параметров, скобки можно оставить пустыми или указать внутри скобок void.

```
<имя функции>(<параметры>);
<имя функции>(void);
<имя функции>();
```

Вообще говоря, для вызова функций их имена используются не всегда. Вместо них могут использоваться указатели на функции. Мы, однако, не будем сейчас останавливаться подробно на вопросах, связанных с функциями в языке C++, так как вернемся к ним позже. Вместо этого, мы рассмотрим управляющую инструкцию return, неразрывно связанную с функциями.

Команда return позволяет завершить выполнение функции и вернуть управление на оператор, следующий за ее вызовом. Формат оператора return следующий:

```
return [<выражение>];
```

Как Вы, наверное, уже догадались, оператор return позволяет также указывать возвращаемое из функции значение (если функция должна его возвращать). Дело в том, что в языке C++ процедуры не отделяются от функций, а являются их подмножеством. В этом случае при объявлении функции указывается тип возвращаемого значения – void, что говорит о том, что функция не возвращает никакого значения. При выходе из таких функций (процедур)

используется оператор return без параметра. В том случае, если тип возвращаемого значения отличен от void, используется оператор return с параметром.

Нелишним будет также отметить, что оператор return – единственный способ указать возвращаемое функцией значение. В C++ нельзя так, как, например, в Pascal-е присвоить значение некоторой встроенной переменной (Result) или имени функции.

2.10 Вопросы и задания

1. Назовите инструкции, с использованием которых производится управление потоком выполнения программы. Дайте краткую характеристику каждой из них.

2. Скажите, с использованием каких операторов можно организовать цикл? В чем заключаются отличия между этими операторами?

3. Назовите особенности конструкции цикла for.

4. Какие операторы позволяют изменить порядок выполнения цикла. Дайте их краткую характеристику.

5. Напишите фрагмент программы, отыскивающий минимум из трех чисел (x,y,z):

a) с использованием условного оператора

b) с использованием тернарной условной операции

6. Напишите фрагмент программы, вычисляющий значение факториала числа x:

a) с использованием цикла while

b) с использованием цикла for

7. Напишите фрагмент программы, вычисляющий значение степени k числа x:

a) с использованием цикла while

b) с использованием цикла for

8. С использованием оператора множественного выбора напишите фрагмент программы, который по символу арифметической операции (+, -, *, /), задаваемой переменной op, вычисляет результат соответствующей операции для переменных x и y.

3 ПРОИЗВОДНЫЕ ТИПЫ ДАННЫХ

3.1 Указатели

Как мы уже говорили ранее, все типы языка C++ разделяются на базовые и производные. Первый из производных типов данных, который мы рассмотрим – указательный. Оговоримся сразу, что указательный тип часто называют также ссылочным, употребляя термины указатель и ссылка, как синонимы. Применительно к языку C++ это неверно, так как в нем указатели и ссылки – разные типы.

Прежде, чем дать определение указателю, давайте еще раз подумаем над тем, что же представляет собой переменная какого-либо базового типа. Рассмотрим, например переменную целочисленную переменную i, определяемую следующим образом:

```
int i;
```

Мы уже знаем, что в памяти под эту переменную отводится 4 байта, то есть 32 разряда, которые трактуются, как целое число со знаком. В программе Вы можете обращаться к этой области памяти по имени. Причем существенным моментом здесь является то, что на протяжении всего времени жизни переменной ее имя остается неизменным, и, используя его, Вы всегда обращаетесь к одной и той же области памяти. Изменяя значение переменной, Вы изменяете только содержимое той области памяти, с которой связано имя переменной. Каким-либо образом изменить саму связь Вы не можете.

На самом деле, на практике существует множество задач, для решения которых возможность изменять связь имени переменной с данными во время работы программы была бы очень полезна. И такая возможность в C++ есть. Для ее реализации вводится указательный тип данных и само понятие указателя.

Указатель – это переменная, которая содержит в качестве значения адрес памяти. В том случае, когда в указателе содержится адрес некоторой другой переменной или объекта, говорят, что он указывает на эту переменную или объект. Как Вы знаете, многие типы данных занимают более одного байта. В этом случае указатель содержит адрес первого байта, связанного с таким типом.

Для того чтобы иметь возможность интерпретации той области памяти, с которой связан указатель, вводят понятие *типизированного* (ограниченного) указателя. Предполагается что типизированный указатель может содержать адрес объекта только того типа, с которым он связан.

Помимо типизированных бывают также и *нетипизированные* (свободные) указатели. Для таких указателей не делается никаких предположений относительно интерпретации той области памяти, с которой они связаны. Соответственно, такие указатели могут адресовать любой объект в оперативной памяти.

3.1.1 Объявление указателей

Объявление указателя в C++ выполняется следующим образом (конечно же, при объявлении указателя, как и при объявлении переменных базовых типов, могут быть использованы все те же спецификаторы и инициализаторы, которые здесь намеренно опущены):

```
<тип> *<имя указателя>;
```

В зависимости от того, какой тип используется в объявлении, мы получим типизированный или нетипизированный указатель. В том случае, если указан тип void, мы получим нетипизированный указатель, во всех остальных случаях – типизированный, связанный именно указанным типом. Приведем несколько примеров:

```
int *pi;    // типизированный указатель на целое число
void *p;    // нетипизированный указатель
double *pd; // типизированный указатель на вещественное число
void**ppv;  // типизированный указатель на void*
```

Заметим, что также как и для переменных базового типа, возможно объявить сразу несколько указателей. Однако в этом случае символ * должен ставиться перед именем каждой переменной:

```
int *pi1, *pi2, *pi3;
```

В том случае, если в приведенном выше объявлении один из символов * будет не указан, соответствующая переменная будет объявлена как целочисленная.

```
int *pi1, pi2, *pi3;  // p2-не указатель!
```

Этот особенность языка часто становится причиной затруднений у начинающих. Чтобы избежать этого, постарайтесь не смешивать в одной строке объявления указателей и базовых типов, а также приучите себя «прижимать» символ * к имени переменной, а не типа.

3.1.2 Операции с указателями

Для работы с указателями в C++ определены следующие операции:

- простое присваивание (=);
- определение адреса (&);
- разыменование (раскрытие) указателя (*);
- приведение типа («()»);
- сложение и вычитание (+, -);

- составное присваивание (+=, -=);
- инкремент и декремент (++, --);
- сравнение (<, >, <=, >=, ==, !=);
- выделение и освобождение динамической памяти (new и delete).

Операция присваивания (=) используется для задания значения указателю. В указатель может быть помещен либо уже известный адрес некоторого объекта (содержащийся в некотором другом указателе), либо предопределенная константа NULL, показывающая, что указатель не указывает ни на один из объектов в оперативной памяти.

```
int *pi;                    int *pi;
pi=p;                       pi=NULL;
```

Следует отметить, что совместимыми по присваиванию считаются либо два указателя одного типа, либо типизированный и свободный указатель. Типизированные указатели разных типов не являются совместимыми.

В том случае, если адрес объекта заранее неизвестен, а сам объект уже существует, для получения адреса служит операция определения адреса (&). Это унарная операция, возвращающая адрес своего операнда.

```
char c = 'A';       // записываем в c 'A'
char* pc = &c;      // записываем в pc адрес c
char** ppc = &pc;   // записываем в ppc адрес pc
```

Операция разыменования (раскрытия) указателя (*) служит для получения самого объекта по его указателю. Это унарная операция, операндом которой является указатель, а результатом – объект на который этот указатель указывает.

```
c = 'B';        // записываем в c 'B'
*pc = 'C';      // записываем в c 'C'
**ppc = 'D';    // записываем в c 'D'
```

Неявное приведение типов возможно только к указателю void*. Для остальных случаев используется операция приведения типа «( )». С ее использованием можно выполнять приведение, как между указателями различных типов, так и между указателями и интегральными типами. Использование этой операции может выглядеть следующим образом:

```
void* p;
int* q;
p=q;
q=(int*) p;
```

Из бинарных арифметических операций к указателям применимы только сложение и вычитание. Вычитание указателей разрешено только для типизированных указателей одного типа, при этом результат определяется, как расстояние между двумя участками памяти на которые указывают указатели, деленное на размер того типа, с которым связаны указатели.

Из указателя можно вычитать целочисленное значение, что означает уменьшение фактического адреса, хранящегося в указателе на вычитаемое значение помноженное на размер того типа, с которым связан указатель.

Сложение указателей не допускается. При сложении указателя с целым числом, значение адреса, хранящееся в указателе увеличивается на добавляемое значение, помноженное на размер того типа, с которым связан указатель.

Операции инкремента и декремента (++, --) увеличивают или уменьшают фактический адрес, хранящийся в указателе на размер того типа, с которым связан указатель. С указателями можно использовать составные операции присваивания += и –=.

Чтобы проиллюстрировать работу арифметических операций с указателями приведем следующий пример. Пусть переменные int i1, i2 размещаются в смежных областях памяти.

```
int *pi1=&i1, *pi2=&i2;
char *pc1=(char*)pi1,
     *pc2=(char*)pi2;
int di=pi2-pi1; // 1
int dc=pc2-pc1; // 4
```

В C++ допускается сравнение указателей с использованием любой из операций сравнения. При этом сравниваются адреса областей памяти, хранящиеся в указателях. Указатели могут быть операндами логических операций и использоваться в условиях циклов и условных операторов. Истинными значениями считаются указатели, отличные от нуля, нулевые (NULL) указатели считаются ложными значениями

Помимо приведенных выше операций при работе с указателями используются также операции new и delete, предназначенные для выделения и освобождения динамической памяти соответственно. Мы, однако, изучим эти операции позже.

3.1.3 Использование модификатора const при объявлении переменных

Как мы говорили выше, при объявлении указателя, так же как и при объявлении переменной базового типа может использоваться модификатор const. Этот модификатор может указываться перед или после типа при объявлении переменной базового типа, что фактически означает объявление символической константы. В этом случае ее значение задается обязательно и впоследствии не может быть изменено.

```
int const c1 = 10;
const int c2 = 20;
```

С указателями ситуация несколько сложнее. Дело в том, что при объявлении указателя может потребоваться, чтобы константным был сам указатель, либо объект на который этот указатель указывает, либо и то и другое. Пусть, например, объявлен указатель на строку символов:

```
char *psz;
```

Если требуется, чтобы константным был сам указатель, const указывается перед названием переменной. В этом случае cpsz является символической константой, значение которой должно быть задано сразу при инициализации и изменено впоследствии быть не может.

```
char *const cpsz = cpsz;
```

В том случае, когда требуется, чтобы неизменяемым был сам объект, модификатор может указываться перед или после базового типа указателя. Приведенные ниже два объявления эквивалентны:

```
const char *pcsz= psz;
char const *pcsz2= psz;
```

В этом случае сам указатель не обязательно должен быть проинициализирован при объявлении и впоследствии может быть изменен. Такое объявление говорит о том, что изменить сам объект с использованием указателя будет нельзя (изменить объект с использованием других указателей, конечно, будет можно).

Наконец, при объявлении указателя Вы можете сделать константным как сам указатель, так и объект, на который он указывает. Для этого модификатор const указывается дважды: один раз до или после типа, связанного с указателем, а второй раз - перед именем указателя. Приведенные ниже два объявления эквивалентны:

```
const char* const cpcsz= psz;
char const* const cpcsz2= psz;
```

В заключение приведем несколько примеров корректного и некорректного использования объявленных выше указателей:

```
*psz   = 'A'; // допустимо        psz++;   // допустимо
*cpsz  = 'A'; // допустимо        cpsz++;  // ошибка
*pcsz  = 'A'; // ошибка           *pcsz++;  // допустимо
*cpcsz = 'A'; // ошибка           *cpcsz++; // ошибка
```

3.2 Ссылки

Ссылки в C++ выделены в отдельный от указателей тип и определяются как другое имя уже существующего объекта. Определение ссылки выполняется сходным с указателем способом, но вместо символа * используется символ &:

```
<тип> &<ссылка>
```

Несмотря на то, что также как и в случае с указателем, в ссылке фактически хранится адрес связанного с ней объекта, синтаксически ссылка ведет себя также как сам объект, становясь синонимом (псевдонимом) существующего объекта. Это означает, что при работе со ссылками, в отличие от указателей, не требуется использовать операцию разыменования.

Ссылки обязательно должны быть проинициализированы и впоследствии не могут быть изменены.

```
int a=5, b=3;
int &ra=a, &rb=b;   // Объявляем и инициализируем ссылки
ra = 25;            // С использованием ссылок присваиваем
rb = 9;             // значения 25 и 9 переменным a и b
```

В приведенном выше виде ссылки практически не используются. Напротив, очень часто ссылки используются при передаче параметров в функции. При изучении функций мы подробно рассмотрим этот вопрос.

3.3 Массивы

Если Вы уже знакомы с каким-либо языком программирования, то наверняка знакомы и с понятием массива. Тем не менее, дадим этому понятию определение.

Массив - это упорядоченное множество располагающихся в смежных областях памяти однотипных элементов, каждый из которых характеризуется индексом (или набором индексом). Использование массивов позволяет организовать хранение и обработку однотипных данных сравнительно большого объема в оперативной памяти.

Обычно изучение массивов начинают с одномерных массивов (векторов), а затем переходят к многомерным, в частности, двумерным массивам (матрицам).

3.3.1 Одномерные массивы

Объявление одномерного массива в языке C++ выполняется следующим образом (спецификаторы класса памяти и модификаторы const, volatile здесь опущены):

```
<тип> <имя> [<размер>] ⟦ = <инициализатор>⟧;
```

Здесь размер - значение (или константное выражение), задающее количество элементов в одномерном массиве, <инициализатор> - заключенный в фигурные скобки список значений элементов массива, перечисленных через запятую.

При определении массива обязательно должен быть задан либо размер массива, либо инициализатор. Не указать размер и список массива возможно только для внешних (extern) массивов. Если указано и то и другое, то количество элементов в списке инициализации не должно превышать объявленный размер. Оставшиеся элементы для базовых типов инициализируются нулевыми значениями, а для объектов - с использованием конструктора по умолчанию. В том случае, если инициализатор не указан вообще, инициализация массива будет выполнена, только если массив является статическим или внешним.

Если размер массива не указан, то он считается равным количеству элементов в списке инициализации. Ниже приводится несколько корректных объявлений одномерных массивов.

Объявление массива из 3 элементов с явной инициализацией всех элементов:

```
char ar1[3] = {‘A’, ‘B’, 0};
```

Объявление массива из 3 элементов с явной инициализацией первого элемента, остальные элементы инициализируются нулями:

```
char ar2[3] = {‘A’};
```

Объявление массива из 5 элементов без инициализации:

```
int ar3[5];
```

Объявление массива из 5 элементов в соответствии с инициализатором:

```
int ar4[] = {10, 200, 3, 404, 55};
```

Доступ к элементам массива осуществляется с использованием операции «[ ]». Внутри квадратных скобок указывается индекс элемента, к которому необходимо получить доступ. При этом следует учитывать, что элементы массивов в C++ всегда номеруются с нуля. Таким образом, если в массиве N элементов, элементы массива имеют индексы с нулевого по (N-1). Ниже приводится пример заполнения элементов целочисленного массива квадратами индексов элементов.

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

На практике при работе с массивами часто нужно знать, сколько элементов содержится в массиве. Конечно, при явном указании количества элементов это не является неразрешимой проблемой. Однако есть и другой достаточно элегантный способ. Этот способ основан на использовании операции sizeof, возвращающей размер памяти, отведенной под хранение ее аргумента в байтах. Очевидно, во всех приведенных выше примерах, для хранения массива резервируется непрерывный участок памяти, размером достаточным для хранения всех его элементов, и этот размер равен `sizeof(<имя массива>)`. С другой стороны, узнать размер, отводимый под хранение одного элемента можно, узнав размер первого элемента: `sizeof(<имя массива>[0])`. Разделив размер всего массива на размер первого элемента, мы получим количество элементов в массиве. Если Вам показалось, что метод будет приводить к ошибке в случае, когда в массиве нет элементов, не расстраивайтесь – определить такой массив не удастся. С использованием операции sizeof приведенный выше пример заполнения массива мог бы выглядеть так:

```
for (int i = 0; i<sizeof(ar)/sizeof(ar[0]); i++)
    ar[i] = i*i;
```

Давайте теперь рассмотрим, как же происходит обращение к конкретному элементу массива. Пусть объявлен некоторый массив: T a[N]. Так как элементы в массиве располагаются последовательно, начиная с нулевого элемента, то доступ к i-му элементу можно представить в виде *(p+i). Здесь p – типизированный указатель на первый элемент массива: T *p=&a[0].

Таким образом, приведенный выше код мог быть записан с использованием указателей в следующем виде:

```
int *p = &ar[0];
for (int i = 0; i<sizeof(ar)/sizeof(ar[0]); i++)
    *(p+i) = i*i;
```

Далее, возможно, это покажется Вам неожиданным, но имя массива в C++ может использоваться в качестве указателя на его первый элемент (ar ==&ar[0]). . Другими словами, везде, где Вам необходим указатель на первый элемент, можете использовать имя массива. С учетом сказанного рассмотренный выше фрагмент примет вид:

```
for (int i = 0; i<sizeof(ar)/sizeof(ar[0]); i++)
    *(ar+i) = i*i;
```

Однако из того, что имя массива можно использовать как указатель, вовсе не следует, что всегда можно делать наоборот. Так, описанная выше операция sizeof будет возвращать размер указателя (4 байта для 32 разрядной архитектуры) независимо от того, на что он указывает. Поэтому, например, определить размер массива с использованием указателей не удастся. В остальных же случаях указатель может использоваться как имя некоторого массива, состоящего из элементов того типа, с которым связан указатель. В частности, к указателям применима операция обращения по индексу «[]».

3.3.2 Строки

Как мы уже говорили, в C++ отсутствует базовый тип для работы со строками. Тем не менее, это вовсе не означает, что поддержка работы со строками в языке не предусмотрена. Итак, что же такое строка.

Строка - это последовательность символов, завершающаяся символом с нулевым кодом. В C++ существует два базовых типа символов: char и wchar_t, но мы будем рассматривать только первый из них. Тогда объявление строки может выглядеть как объявление массива символов:

```
char str[]={‘A’,’B’,’C’,0};
```

Здесь мы не только объявили массив, но и инициализировали его так, что он содержит строку из трех символов и завершающий ноль. Но, оказывается, инициализировать массив символов можно проще - с использованием строковой константы:

```
char str[] = “ABC”;
```

Несмотря на то, что в константе указано только три символа, завершающий ноль будет добавлен автоматически. Можно легко проверить это, вызвав операцию sizeof(“ABC”), которая вернет 4. Более того, строковую константу можно присвоить указателю:

```
char *psz = “ABC”;
```

Хотя последние два выражения на первый взгляд кажутся идентичными, нужно ясно представлять себе разницу между ними. В первом случае происходит объявление массива и копирование в него константы. Во втором - объявляется указатель, и в него записывается адрес константы. Существенным моментом здесь является то, что изменение символа строки в первом случае (str[0]='X') будет успешным, а во втором случае (*psz='X') может привести к ошибке, связанной с попыткой присваивания константе. Чтобы обезопасить себя от таких ошибок объявляйте указатели на строковые константы с модификатором const:

```
const char *psz = "ABC";
```

Следует отметить, что при работе со строками указатели используются повсеместно. Например, приведем фрагмент кода, вычисляющий длину строки:

```
const char *pcsz = "Example";
const char *p = pcsz;
while (*p) p++;
int len = p-pcsz;
```

Естественно, стандартная библиотека языка включает в себя целый ряд функций работы со строками (заголовочный файл <string.h>), и Вы, наверняка, найдете в ней все, что Вам нужно. Мы не будем здесь подробно изучать эти функции, а только перечислим некоторые из них.

Таблица 3.1 – Некоторые функции работы со строками

Функция	**Назначение**
size_t strlen(const char *str);	Вычисление длины строки
char *strcpy(char *strDestination, const char *strSource);	Копирование строк
int strcmp(const char *string1, const char *string2);	Сравнивание строк
char *strcat(char *strDestination, const char *strSource);	Конкатенация строк
char *strchr(char * str, int c)	Поиск символа в строке
char *strstr(const char *str, const char *strSearch);	Поиск подстроки в строке
char *strtok(char *strToken, const char *strDelimit);	Разбиение строки на лексемы
char *strdup(const char *s); (может не входить в стандартную библиотеку)	Создание копии строки в динамической памяти

3.3.3 Динамические массивы

Очень часто на практике возникают задачи, для которых использование массива фиксированной наперед заданной длины неприемлемо. Представьте себе, например, что Вы составляете программу для работы с таблицами вещественных данных. В этом случае Вам придется завести в программе массив, достаточный для хранения таблицы максимально допустимого размера. Далее, в том случае, если возникает необходимость произвести более-

менее сложную обработку, требующую создание временной таблицы для хранения промежуточных результатов, размер памяти, требуемый программе для работы, увеличивается во столько раз, сколько таких таблиц требуется хранить одновременно. Ситуация усугубляется в том случае, если от программы требуется возможность открывать несколько документов сразу. Если на практике работа с таблицами максимально допустимого размера встречается довольно редко, ровно как и открытие максимально допустимого количества документов, такая бессмысленная трата памяти - непозволительная роскошь.

Представим теперь себе, что мы могли бы выделить ровно столько памяти, сколько требуется для хранения той или иной таблицы, а по окончании работы с таблицей - освободить память. В этом случае, мы бы не только высвободили бессмысленно затрачиваемые ресурсы (например, для выполнения других задач), но и могли бы гибко работать с ограничениями на максимальный размер и количество открытых таблиц. В том случае, если памяти достаточно, мы могли бы открыть таблицу большего размера, или, наоборот, открыть большее количество небольших таблиц.

Таких примеров, как рассмотренный ваше, можно привести достаточно много. Для нас же сейчас важно то, что в языке C++ существует возможность работы с *динамической памятью* - памятью выделяемой и освобождаемой по явному запросу из программы. Эта возможность реализуется, как функциями стандартной библиотеки, так и операциями языка.

Библиотечные функции для работы с динамической памятью определены в стандартном заголовочном файле <*stdlib.h*>. Ниже приводятся заголовки функций с кратким описанием.

```
void *malloc( size_t size );
```

Функция выделяет область памяти размером size байт. В случае, если память была успешно выделена возвращается указатель на первый байт памяти, в противном случае - нулевой указатель. Содержимое выделенного блока памяти никак не инициализируется (не определено).

```
void *calloc( size_t num,  size_t size );
```

Функция выделяет память под массив из num элементов, каждый из которых имеет размер size и инициализирует массив нулевыми значениями. Функция возвращает указатель на первый элемент массива в случае, если функция отработала успешно. В противном случае возвращается нулевой указатель.

```
void *realloc(  void *memblock,  size_t size );
```

Функция меняет размер ранее выделенной области памяти, на которую указывает memblock на новый размер size. Функция возвращает указатель на возможно новую область памяти. Функция копирует содержимое в новую область памяти, при необходимости усекая его. Дополнительный объем памяти при этом никак не инициализируется. В случае когда

memblock равен NULL, функция выделяет память подобно malloc. В случае когда size равен 0, функция освобождает ранее выделенную память, подобно функции free.

```
void free( void *memblock );
```

Функция освобождает ранее выделенную с использованием malloc, calloc или realloc область памяти на которую указывает memblock.

Приведем небольшой пример выделения памяти под массив из 100 вещественных чисел двойной точности, заполнения массива индексами элементов и освобождения памяти.

```
double* buf = (double*)malloc(100*sizeof(double));
if (buf)  {
   for (int i = 0, double *p=buf; i < 100; i++, p++) *p=i;
   free(buf);
}
```

Другим способом работы с динамической памятью является использование операций new и delete. Эти операторы имеют различный синтаксис в случае выделения памяти под одиночные объекты и под массивы объектов. В случае, когда требуется выделить и освободить память под одиночный объект, операции имеют следующий вид:

```
<имя> = new <тип>⟦(< инициализатор>)⟧;
delete <имя>;
```

Здесь <имя> - название указателя на создаваемый объект, <тип> - тип элементов объекта в массиве. Отметим, что при использовании операции new имеется возможность указать инициализатор в стиле вызова функции (в круглых скобках). Операции new и delete используются при работе с объектами классов, и мы обязательно рассмотрим этот вопрос чуть позже, а пока приведем пример выделения и освобождения памяти под объекты базовых типов:

```
double* pdbl = new double;
char * pch = new char('A');
delete pdbl;
delete pch;
```

В случае, когда требуется выделить или освободить память под одномерный массив объектов операции имеют следующий вид:

```
<имя> = new <тип> [<выражение>];
delete [] < имя >;
```

Размер динамического массива, определяемый во время выполнения программы в соответствии с выражением <выражение>, показывает количество элементов создаваемого массива. Для оператора new размер обязательно указывается в квадратных скобках, для оператора delete скобки остаются пустыми. С использованием операций приведенный выше пример с массивом вещественных чисел мог бы выглядеть следующим образом:

```
int size = 100;
double* ar = new double[size];
for (int i = 0; i < 100; i++) ar[i] = i;
delete [] ar;
```

Как Вы, наверное, заметили, в последнем примере мы не проверяли результат операции выделения памяти new. Дело в том, что вызываемая таким образом операция new в случае ошибки (например, при нехватке памяти) вместо того, чтобы возвратить NULL сгенерирует исключение и следующие за new операторы не будут выполнены. Подробнее об исключениях мы поговорим позднее в соответствующем разделе.

В заключение отметим, что память, отведенную с использованием операции new, следует освобождать с использованием операции delete, и, наоборот, память, выделенную с использованием функций malloc, calloc, realloc следует освобождать с использованием функции free. Кроме того, если Вы выделяли массив с использованием операции new[], то освобождайте его с использованием соответствующей операции delete[] с указанием квадратных скобок. Освобождать такой массив через delete без указания квадратных скобок является ошибкой.

3.3.4 Многомерные массивы

Многомерные массивы на практике используются очень часто. Даже в приведенном в предыдущем разделе примере с таблицами идет речь о двумерном массиве. Рассмотрим, как объявляются многомерные массивы в C++. Многомерный массив фиксированного размера объявляется подобно одномерному с той лишь разницей, что после имени массива указывается не одни квадратные скобки с размером массива, а последовательность таких скобок:

```
<тип> <имя> [<размер>][[<размер>], …] [ = <инициализатор>];
```

Здесь каждая размерность массива по-прежнему задается константным выражением, а вот формат инициализатора меняется. В инициализаторе многомерного массива могут быть использованы вложенные фигурные скобки, например так, как это показано в следующих примерах:

```
int ar[3][4]={{11,12,13,14},{21,22,23,24},{31,32,33,34}};
int ar[2][2][3]={ { {111,112,113},{121,122,123} },
                  { {211,212,213},{221,222,223} } };
```

Однако, учитывая, что многомерные массивы хранятся как непрерывная последовательность элементов, не стоит удивляться, что инициализация приведенных выше массивов может быть выполнена и следующим образом:

```
int ar[3][4]={11,12,13,14,21,22,23,24,31,32,33,34};
int ar[2][2][3]=
    {111,112,113,121,122,123,211,212,213,221,222,223};
```

Вообще говоря, на практике многомерные массивы фиксированного размера применяются достаточно редко, поэтому мы не будем подробно останавливаться на них, а перейдем к рассмотрению *многомерных динамических массивов*.

Первый из способов, объявления динамического массива основан на создании динамического одномерного массива массивов фиксированного размера. Как создается подобный массив, рассмотрим на примере:

```
int (*ar) [3][4];          // объявляем указатель на массив 3x4
int size = 5;              // задаем размер во время выполнения
ar = new int [size][3][4];// создаем массив 5x3x4
ar[0][1][2] = 123;         // здесь работаем с массивом
…
delete [] ar;              // освобождаем память
```

В этом примере Вы, вероятно, встретили для себя сразу несколько новых моментов. Во-первых, объявление указателя на производный тип данных - двумерный массив. Во-вторых, вызов операции new для создания многомерного массива, и на этом моменте стоит остановиться подробнее. Дело в том, что синтаксис операции new допускает следующий вид:

```
<указатель> = new <тип>[<размер1>][[<размер i], … ]];
```

Особенностью здесь является то, что задаваться во время выполнения программы здесь может только <размер1>. То есть, только первая по счету размерность может быть задана неконстантным выражением. Все остальные размерности должны задаваться строго константными выражениями.

Обратим внимание на то, что приведенный выше код может быть записан с использованием оператора определения типа typedef несколько иначе:

```
typedef int ArType [3][4]; // определяем тип массив 3x4
int size = 5;              // задаем размер во время выполнения
ArType *ar = new ArType [size];// вектор из 5 массивов 3x4
ar[0][1][2] = 123;         // здесь работаем с массивом
…
delete [] ar;              // освобождаем память
```

Из приведенного примера становится совершенно очевидно, что представляет собой такого рода массив, и это не что иное, как массив массивов.

Вообще говоря, таким способом заданные динамические многомерные массивы практически не используются на практике. Причина этого очевидна – в таком многомерном массиве фиксированы все размерности, кроме одной.

Гораздо чаще на практике используется другой способ создания многомерных динамических массивов. Он основан на следующем принципе:

- создается одномерный массив указателей, по размеру соответствующий первой размерности массива;

- затем для каждого элемента созданного массива создается одномерный массив указателей, по размеру соответствующий второй размерности массива и т.д.

- В конце концов, для каждого элемента созданных массивов создаются одномерные массивы элементов нужного типа, по размеру соответствующие последней размерности многомерного массива.

Рассмотрим этот способ на примере создания двумерного массива:

```
const in N=2, M=3;      // размеры массива
int** ar = new int* [N];// создание вектора указателей на строки
for (int i=0; i<N; i++) {    // для каждой строки
     ar[i] = new int [M];    // создание массива-строки
     for (int j=0; j<M; j++) // для каждого столбца
          ar[i][j] = i+j;    // заполняем элемент
}
```

Удаление таким образом организованного многомерного массива также отличается от того, что мы делали ранее:

```
for (int i=0; i<N; i++) // для каждой строки
    delete [] ar[i];    // освобождаем память из под строки
delete [] ar;    // освобождаем память из под массива указателей
```

Несмотря на некоторую громоздкость, связанную с созданием множества одномерных массивов, этот способ обладает неоспоримыми преимуществами. Во-первых, все размерности массива могут быть заданы во время выполнения программы. Во-вторых, массив может размещаться в оперативной памяти не непрерывно, а сравнительно небольшими частями, что позволяет разместить массив даже в сильно фрагментированной памяти, когда размещение сплошного блока памяти такого размера было бы невозможно. Впрочем за приведенные преимущества мы платим бОльшим объемом исходного кода и накладными расходами, связанными с хранением массива(-ов) указателей.

В том случае, если предыдущий подход использовать нерационально, можно воспользоваться, наверное, наиболее простым способом - трактовать линейную последовательность элементов (одномерный массив), как многомерный массив. В этом случае, нам нужно лишь правильно рассчитать размер одномерного массива и определить порядок размещения элементов многомерного массива в одномерном массиве. Для двумерного случая реализация такого подхода может выглядеть следующим образом:

```
const in N=2, M=3;          // размеры массива
int* ar = new int [N*M];    // создаем массив NxM
for (int i=0; i<N; i++) {   // по всем «строкам»
     for (int j=0; j<M; j++)// по всем «столбцам»
          ar[i*M+j] = i+j;  // обращение к элементу i,j
}
…                           // работа с массивом
delete [] ar;    // освобождаем память из под массива
```

Неудобство, возникающее вследствие необходимости расчета индекса в одномерном массиве можно устранить, объявив и один раз проинициализировав вектор указателей на «строки». В этом случае обращаться к такому массиву можно будет как к двумерному.

Конечно же, последний рассмотренный подход является всего лишь имитацией многомерного массива. Тем не менее, он позволяет эффективно решать многие задачи, прост в реализации, и, поэтому, часто используется на практике.

3.5 Структуры

В предыдущем разделе мы рассмотрели массивы, элементами которых являлись однотипные данные. *Структуры* же представляют собой набор именованных атрибутов различных типов, называемых полями. Объявление структуры выглядит следующим образом:

```
struct ⟦<тип структуры>⟧
{
    <тип> <имя>⟦, <имя>, …⟧;
  [ <тип> <имя>⟦, <имя>, …⟧;
    … ]
} ⟦<список переменных>⟧;
```

Здесь <тип структуры> - определяемое имя типа структуры, <тип> - тип поля структуры, <имя> - имя поля структуры, <список переменных> - необязательный список переменных нового типа структуры, перечисленных через запятую.

Приведем несколько примеров определения структур:

```
struct Stud {
    char* surname;
    char* name;
    int group;
};
struct {
    int key;
    double val;
} s1, s2;
struct ListElm {
    Stud stud;
    ListElm *next;
} *pFirst;
```

Обратите внимание, что для второй структуры имя типа самой структуры не задано, но объявлены две переменных структуры такого типа. В третьем примере в качестве типа поля используется указатель на сам объявляемый тип. Это вполне допустимо, в то время как задавать в качестве поля структуры саму структуру нельзя.

В C++ возможно при объявлении структуры указывать в качестве полей указатели на еще не введенные типы. Для этого необходимо предописать вводимый позднее тип с использованием ключевого слова struct (или class). Например:

```
struct Group;
struct Stud {
    char* surname;
    char* name;
    Group* pGroup;
};
struct Group {
    int num;
    int studCount;
    Stud *studs;
};
```

Объявление переменных типа структура может быть сделано не только при определении самого типа, как было показано выше, но и в любом месте после определения структуры. Объявление выполняется следующим образом (спецификаторы класса памяти и модификаторы const, volatile здесь опущены):

```
⟦struct⟧ <тип структуры> <имя> ⟦ = <инициализатор>⟧;
```

В C++ ключевое слово struct при объявлении переменных чаще всего не используется, а осталось от более ранних версий C, где использование struct было обязательным.

Инициализация структур, так же как и для массивов выполняется перечислением значений полей в фигурных скобках. Значения полей при этом перечисляются в порядке их объявления:

```
Stud st1 = {“Иванов”, “Петр”, NULL};
```

Обращение к отдельным полям структуры осуществляется с использованием операций доступа «.» или «->». Первая из операций применяется к самим объектам структур, вторая – к указателям на такие объекты:

```
Stud st;
Stud* pst = &st;
st.surname = “Петров”;
pst->name = “Иван”;
```

Следует отметить, что последнюю запись с использованием операции «->» можно заменить на эквивалентную, но не настолько удобную запись с использованием операции разыменования:

```
(*pst).name = “Иван”;
```

Использование круглых скобок в последнем случае обязательно, так как операция доступа имеет более высокий приоритет, чем разыменование.

Следует отметить, что в структурах, объединениях и классах возможно использование так называемых ***битовых полей***, размер которых указывается в битах. Использование битовых полей связано с существенными ограничениями: нельзя получить адрес битового поля, одна и та же структура с битовыми полями может быть по-разному представлена в памяти в

зависимости от архитектуры, операции с битовыми полями требуют большего времени исполнения. По этой причине мы здесь не будем рассматривать битовые поля.

3.6 Объединения

При решении практических задач иногда возникает необходимость размещать в одной и той же переменной разнотипные данные. Вернемся к тому примеру, который был описан в разделе про динамические массивы. Представим себе, что пользователь захотел модифицировать программу таким образом, чтобы в рамках одной таблицы могли обрабатываться как вещественные, так и целочисленные, символьные, а, возможно, и строковые данные. В этом случае нам пришлось бы вводить некоторый универсальный тип, способный содержать в себе данные разных типов. Для этой цели мы могли бы воспользоваться структурой, определенной, например, таким образом:

```
struct Cell {
    int iVal;
    double dblVal;
    char cval;
    char *pszVal;
    unsigned char type;
};
```

Нижнее из полей структуры предназначено для определения того, значение какого типа хранится в структуре в тот или иной момент времени. Конечно, при таком подходе значительный объем памяти тратится впустую, так как в один момент времени используется только одно из всех полей структуры (не считая поля type). Выходом из этой ситуации является использование объединений.

Объединения могут включать данные разных типов, при этом все атрибуты имеют один и тот же начальный адрес. Другими словами все атрибуты объединения разделяют одну область памяти. При этом размер всего объединения равен максимальному из размеров своих атрибутов. Таким образом, объединения по-разному интерпретируют одну и ту же область памяти. Объявление объединения выглядит следующим образом:

```
union ⟦<тип объединения >⟧
{
    <тип> <имя>⟦, <имя>, …⟧;
  ⟦ <тип> <имя>⟦, <имя>, …⟧;
    … ⟧
} ⟦<список переменных>⟧;
```

Приведем пример объявления объединения

```
union CellData {
    int iVal;
    double dblVal;
    char cval;
```

```
    char *pszVal;
};
```

Мы не включили в объединение поле type, так оно должно храниться снаружи объединения, так чтобы изменение полей объединения не могло повредить поле type.

Объявление объединений может быть сделано в любом месте после определения объединения. Объявление выполняется следующим образом (спецификаторы класса памяти и модификаторы const, volatile здесь опущены):

```
[union] <тип объединения > <имя> [ = <инициализатор>];
```

В C++ ключевое слово union при объявлении переменных может не использоваться (в отличие от более ранних версий C). Инициализация объединения, в отличие от структур, выполняется заданием значения первого поля объединения в фигурных скобках:

```
union CellData cd = {123};
```

Обращение к отдельным полям объединения осуществляется так же, как к полям структуры.

В заключение отметим, что определение приведенного выше типа Cell может быть выполнено так:

```
struct Cell {
    CellData ct;
    unsigned char type;
};
```

То же самое может быть сделано с использованием анонимного объединения в составе структуры:

```
struct Cell {
    union {
        int iVal;
        double dblVal;
        char cval;
        char *pszVal;
    };
    unsigned char type;
};
```

Возможно, приведенный код выглядит немного неожиданно. Здесь union {…} выступает как определение типа анонимного поля в составе структуры. Обращаться к элементам объединения в этом случае можно так:

```
Cell c;
c.iVal=1234;
c.type=0;
```

3.7 Перечисления

Помимо ключевого слова const в C++ существует еще один способ задания символических констант. Этот способ связан с использованием *перечислимого* типа enum. Задание типа выглядит следующим образом:

```
enum ⟦<имя типа>⟧
    {   <имя> ⟦=<значение>⟧;
      ⟦ <имя> ⟦=<значение>⟧;
        … ⟧
    } ⟦<список переменных>⟧;
```

Здесь <имя типа> - необязательное имя перечислимого типа, <имя> - имя константы перечислимого типа, <значение> - необязательное значение соответствующей константы. По умолчанию, если первое значение не задано явно, оно принимается равным нулю, а каждое последующее значение на один больше предыдущего. В том случае, если какое-то из значений задано явным образом, последующие значения отсчитываются от предыдущего. Значения не обязательно должны быть уникальными и могут повторяться.

Объявление переменной перечислимого типа выполняться следующим образом:

```
⟦enum⟧ <имя типа> <имя перем> ⟦=<инициал>⟧;
```

Пример:

```
// объявление перечисление
enum EMonth {Jan, Feb, Mar, Apr, May, Jun,
             Jul, Aug, Sep, Oct, Nov, Dec };
// Объявленные константы будут иметь значения
// Jan=0, Feb=1, Mar=3, … Dec=11
// объявление переменной
month cur_mon = Sep;

// объявление перечисления
enum En {En1=100, En2, En3, En4=0, En5, En6=50}
//Объявленные константы будут иметь значения
// En1=100, En2=101, En3=102, En4=0, En5=1, En6=50
```

Следует отметить, что значение типа enum может быть неявно приведено к целому типу, обратное преобразование выполняется явным образом.

Задание констант с использованием макроопределений

Помимо двух уже рассмотренных способов задания констант в языке C++ есть еще один часто используемый способ, связанный с макроопределениями. Мы здесь не будем рассматривать, как задаются макроопределения в общем виде. Скажем лишь, что для задания константы поместить запись следующего вида:

```
#define <имя> <подставляемое значение>
```

Здесь #define – директива препроцессора, <имя> - имя макроопределения, <подставляемое значение> - текст, который будет подставлен в Вашу программу взамен имени макроопределения. Указанная запись может быть помещена в любом месте исходного текста программы, важно лишь, чтобы помимо нее на данной строке ничего не было (за исключением, возможно, комментариев). Приведем несколько примеров задания макроопределений.

```
#define MAX 100
#define STUDENT “Петров”
```

3.8 Вопросы и задания

1. Что такое указатель? Какие виды указателей Вы знаете? Как объявить указатель?

2. Какие операции допустимы над указателями? Дайте краткую характеристику таких операций.

3. Какие ограничения накладываются на указатель и сам объект при использовании модификатора const?

4. В чем отличие ссылочного и указательного типов в C++?

5. Каким образом можно объявить и инициализировать одномерный массив?

6. Как связано имя массива с указателем на первый элемент. Какие между ними различия?

7. Что представляет собой строка?

8. Какие способы выделения и освобождения динамической памяти Вы знаете?

9. Что представляют собой многомерные массивы в C++? Как объявить и проинициализировать такой массив? Приведите пример для массива 2x3.

10. Какими способами можно выделить многомерный динамический массив? В чем отличие этих способов?

11. Что представляют собой структуры и объединения в C++. В чем различия между ними?

12. Дайте характеристику перечислимому типу enum. Какие еще способы задания констант Вы знаете?

13. Напишите фрагмент кода, в котором выделяемый массив фиксированного размера из N элементов (N-символическая константа) заполняется членами арифметической прогрессии с заданными параметрами.

14. Напишите фрагмент кода, вычисляющий длину строки.

15. Напишите фрагмент кода, устанавливающий указатель на заданный символ строки. В случае если заданный символ отсутствует в строке указателю должно присваиваться нулевое значение.

16. Напишите фрагмент кода, в котором выделяемый динамический массив из n элементов заполняется членами геометрической прогрессии с указанными параметрами, после чего освобождается.

17. Напишите фрагмент кода, в котором двумерный динамический массив m x n элементов заполняется случайными числами, после чего по заполненному массиву рассчитывается среднее арифметическое его элементов, и массив освобождается. Для получения случайного значения в диапазоне [0;1] используйте функцию rand (заголовочный файл <stdlib.h>), генерирующую случайное целочисленное значения значение в диапазоне [0; RAND_MAX]:

```
int rand(void).
```

18. Объявите перечислимый тип, соответствующий дням недели. Напишите фрагмент кода, вычисляющий, является ли день недели выходным днем.

4 ФУНКЦИИ

Функции представляют собой синтаксически выделенные части программы, которые могут быть многократно вызваны из других частей программы. Использование функций в программе позволяет не только избежать дублирования кода в программе, но и структурировать программу. Это облегчает, как разработку программы, так и ее сопровождение. Один раз написанная и отлаженная функция, может быть повторно использована в других проектах.

4.1 Определение функции

В языке C++ функции определяются следующим образом:

```
⟦<спецификаторы>⟧ <тип> <название>( ⟦<список параметров>⟧ )
{
    //тело функции
}
```

Здесь <спецификаторы> включают в себя как уже знакомые нам спецификаторы static и extern, так и новый спецификатор inline, <тип> - тип возвращаемого функцией значения, <название> - название функции, <список параметров> - список формальных параметров функции.

Отметим, что в языке C++ процедуры синтаксически не отделяются от функций, а являются подмножеством функций. В том случае, когда необходимо определить процедуру, следует указать void в качестве типа возвращаемого значения. Это будет указывать на тот факт, что функция не возвращает никаких значений и фактически является процедурой. Во всех остальных случаях следует указать отличный от void тип. Тип возвращаемого значения может быть базовым или производным. Следует иметь в виду, однако, что функция не может возвращать массивы или функции.

Появление спецификаторов класса памяти при описании функции может показаться несколько неожиданным. Тем не менее, спецификаторы static и extern могут использоваться при объявлении функций и определяют видимость объявляемой функции. В случае если функция задается как *static*, она может быть использована только в том модуле, где она описана. В том случае если функция объявлена со спецификатором *extern*, она может быть использована также из других модулей. По умолчанию считается, что функция объявлена как extern.

Спецификатор *inline* служит для объявления так называемых *встраиваемых* (подставляемых) функций. Особенность обработки таких функций заключается в том, что вместо передачи управления единственному экземпляру функции код функции подставляется в точку вызова. Это приводит к разрастанию кода программы, но увеличивает скорость её

выполнения. Объявлять со спецификатором inline имеет смысл небольшие, часто вызываемые в программе функции. Следует отметить, что спецификатор inline лишь указывает компилятору на Ваше желание генерировать встраиваемую функцию, которое компилятор может проигнорировать из-за ограничений, накладываемых на такие функции.

В случае, когда функция не имеет параметров список параметров остается пустым или на его месте указывается void. В остальных случаях формальные параметры перечисляются через запятую, перед каждым параметром указывается его тип. Имена неиспользуемых параметров могут быть опущены. Кроме того, имена могут быть опущены при объявлении функции (в отличие от определения, см. ниже).

Следует отметить, что в C++ могут быть объявлены функции с переменным числом параметров, функции с параметрами, имеющими значения по умолчанию. О таких функциях мы поговорим чуть позднее.

Тело функции указывается тогда, когда происходит *определение* функции (в отличие от объявления, при котором тело функции отсутствует). При этом последовательность операторов, реализующих функцию, записывается в фигурных скобках непосредственно за заголовком функции. *Объявления* функции, в отличие от определения делаются в тех местах кода, где определение используемой функции недоступно, например, в других программных модулях или в том же модуле, но до определения функции. Объявления должны с точностью до имен параметров совпадать с определением функции.

Приведем несколько примеров объявления и определения функций.

```
double func1(void);     // объявление функции без параметров
static int func2(void)  // определение функции без параметров
{ return 0; };
inline double mean(double a, double b) // определение
{   return 0.5*(a+b);   }              // встраиваемой функции
void proc(int, double); // объявление процедуры без указания
                        // имен параметров
```

Каждая исполняемая программа должна иметь *точку входа* в программу, определяющую, откуда начнется исполнение программы. Для программ, написанных на C++, такой точкой входа является главная функция программы, как правило, имеющая имя main:

```
int main(int argc, char* argv[])
```

Параметрами этой функции являются количество аргументов и сами аргументы командной строки, указанной при запуске программы. Для программ, написанных под операционную систему Windows, функция имеет имя WinMain и список формальных параметров отличается от приведенного выше. Главная функция программы вызывается при запуске программы автоматически. Все остальные функции, определенные в программе могут быть вызваны из нее напрямую, либо косвенно.

4.2 Вызов функции

Вызов функций производится с использованием оператора «( )». В скобках через запятую перечисляются выражения, значения которых должны по типу соответствовать формальным параметрам вызываемой функции. Оператор может следовать за названием функции или указателем на функцию.

При вызове функции на стеке отводится память под формальные параметры функции, результат и адрес возврата. Там же отводится память под автоматические переменные, объявленные в вызываемой функции. По завершении работы функции значения результата функции и изменяемых параметров восстанавливаются из стека, и осуществляется переход по адресу возврата к оператору, следующему за операцией вызова функции. Описанный порядок вызова функции позволяет совершать в программе любые вызовы функций, в том числе и рекурсивные вызовы.

Рекурсивным называется вызов, при котором некоторая функция в процессе выполнения обращается сама к себе. В случае если такое обращение происходит непосредственно, говорят о прямой рекурсии. В случае, когда такое обращение производится в результате вызовов с участием других функций, говорят о косвенной рекурсии. Ниже приводится пример рекурсивной функции, вычисляющей факториал.

```
int fact( int n )
{
    if (n >1 )  return n*fact(n-1);
    else  return 1;
}
```

Для приведенного выше примера, конечно же, можно написать нерекурсивную функцию с использованием *итеративного* алгоритма (цикла). Следует, однако, помнить, что рекурсивный алгоритм далеко не всегда может быть заменен итеративным. А вот для итеративного алгоритма всегда может быть найден рекурсивный эквивалент.

Часто в программах требуется, чтобы функция возвращала более одного значения или изменяла свои параметры. Однако в том случае, когда параметры передаются по значению, в стеке содержатся лишь копии исходных параметров, и их изменение никак не отразится на тех переменных, которые передавались в функцию при вызове.

Чтобы изменить значение некоторой переменной, передаваемой в функцию можно, конечно, передавать указатель на переменную вместо значения переменной. В этом случае из функции к такому параметру придется обращаться, используя операцию разыменования. Например, рассмотрим функцию поиска минимального и максимального элемента в массиве:

```
bool FindMinMax( int *Ar, int Cnt, int *Min, int *Max)
{
```

```
    if ( !(Cnt>0 && Ar && Min && Max))  return false;
    *Min = *Max = Ar[0];
    for (int i = 1; i < Cnt; i++)  {
        if (Ar[i]<*Min)  *Min = Ar[i];
        else if (Ar[i]>*Max)  *Max = Ar[i];
    }
    return true;
}
```

Все параметры, которые мы передавали в этой функции, передавались *по значению*. Для них при вызове функции на стеке создавались копии. В случае с указателями создавались копии указателей. Однако в C++ возможно передавать параметры не только по значению, но и *по ссылке*. В этом случае на стеке сохраняется не значение передаваемого параметра, а его адрес. Передаваемый по ссылке параметр объявляется с использованием символа «&». Внутри процедуры работа с параметром выполняется так же, как и с параметром, передаваемым по значению (не требуется разыменований). Однако все действия будут производиться с исходными переменными, переданными в функцию, а не копиями их значений. В качестве примера рассмотрим ту же функцию поиска минимального и максимального элемента:

```
bool FindMinMax( int *Ar, int Cnt, int &Min, int &Max)
{
    if ( Cnt<=0 || !Ar ))  return false;
    Min = Max = Ar[0];
    for (int i = 1; i < Cnt; i++)  {
        if (Ar[i]<Min)  Min = Ar[i];
        else if (Ar[i]>Max)  Max = Ar[i];
    }
    return true;
}
```

Передача параметра по ссылке позволяет не только выполнять модификацию передаваемых параметров, но и экономить время и память при передаче параметров. Как мы говорили выше, при передаче параметров по значению осуществляется их копирование. Для параметров значительного размера и при частом обращении к функции на копирование может тратиться значительный объем памяти и времени, чего можно избежать, передавая параметр по ссылке или указателю. В том случае, если параметр не изменяется внутри функции, лучше передавать его по константной ссылке или указателю:

```
struct Person{
    char Name[20];
    char Surname[20];
    char PassportNo[11];
    char Addr[99];
};
void DoSomething( const Person &person );
void DoSomething( const Person *person );
```

Ссылка может использоваться не только в качестве параметра функции, но и в качестве возвращаемого значения. Рассмотрим, например, функцию, возвращающую минимальный элемент в массиве по ссылке:

```
int &MinElm(int Count, int *Ar)
{
    int ind=0;
    for (int i=0; i<Count; i++)
        if (Ar[i] < Ar[ind])  ind=i;
    return Ar[ind];
}
```

Обращение к определенной таким образом функции может использоваться как при вычислении выражений, где требуется целочисленное значение, так и находиться слева от операции присваивания, что может показаться несколько необычным. Например, будет корректен следующий код:

```
int ar[] = {10,5,7,30,55,1,20};
int cnt = sizeof(ar)/sizeof(ar[0]);
MinElm(cnt, ar) = 0;
```

или, иначе:

```
int &min = MinElm(cnt, ar);
min = 0;
```

В обоих случаях элемент ar[5] станет равным нулю.

4.3 Задание параметров по умолчанию

Еще одним важным моментом при описании параметров функций является возможность задания параметрам функции значений по умолчанию. Задавать значения по умолчанию можно как при объявлении, так и при определении функции:

```
int print( double dvalue, int prec = 2 ); // объявление
int print( double dvalue )                // определение
{...}
```

или

```
int print( double dvalue );               // объявление
int print( double dvalue, int prec = 2 )  // определение
{...}
```

Повторное объявление параметра по умолчанию в объявлении или определении функции не допускается. Параметры по умолчанию всегда должны следовать в конце. Ошибкой будет, например, такое объявление:

```
int print(int prec = 2, double dvalue);
```

Значение, задаваемое по умолчанию не обязательно должно быть константным выражением. В выражении могут участвовать глобальные переменные, функции, находящиеся в области видимости и т.п. Главное, что нужно запомнить - проверка выражения выполняется во время компиляции, а вычисление – во время вызова функции. Следует также отметить, что

если функция имеет несколько параметров по умолчанию, то незаданными считаются всегда последние параметры.

4.4 Функции с неопределенным числом параметров

Наверное, самой интересной возможностью языка, касающейся функций, является возможность определения функций с неопределенным числом параметров. Хотя на практике необходимость в определении подобных функций возникает не очень часто, порой такая возможность бывает очень полезна. В качестве примера функции с неопределенным числом параметров можно привести функцию printf стандартной библиотеки. Синтаксически список параметров для таких функций выглядит, как последовательность фиксированного числа параметров, записанных обычным образом, за которыми ставится троеточие.

<список фиксированных параметров>, …

Таким образом, в функциях с неопределенным числом параметров тип и количество параметров во время компиляции определить не возможно. Известными они становятся только при вызове функции. При этом за определение количества фактически переданных параметров и их типов отвечает программист. Как правило, для определения фактического количества параметров и их типов используют один или несколько фиксированных параметров. Другим способом может быть добавление в конец списка параметров некоего параметра-индикатора с уникальным значением, обозначающим конец списка параметров.

Для определения фактических значений требуемых параметров вспомним, что параметры при вызове функции размещаются на стеке последовательно. При этом, принятое в языке С соглашение говорит о том, что параметры размещаются в порядке их следования в списке параметров. Это означает, что установив указатель непосредственно за последним фиксированным параметром, мы получим доступ к следующему параметру. Далее, зная типы передаваемых параметров и последовательно смещая указатель, мы можем получить доступ к остальным параметрам функции.

Ниже приводится пример функции, вычисляющей среднее арифметическое переданных ей вещественных чисел. В этом примере количество переданных параметров задается в единственном фиксированном параметре.

```
double average(int n, ...)          // функция с неопределенным
{                                   // числом параметров
    double *p = (double*)(&n + 1); // устанавливаем указатель на
                                    // первый нефиксир. параметр
    double s = 0.0;                     // результат суммирования
    for (int i=0; i<n; i++)  s += *p++;// суммируем n параметров
    return s/n;                         // возвращаем результат
}
```

Использовать функцию можно, например, следующим образом:

```
double s2 = average(3, 1.1, 2.2, 3.3);
```

Здесь сначала указывается, сколько будет передано чисел, а потом указываются сами эти числа. В приведенном примере все нефиксированные параметры имеют один и тот же тип, который и определяет тип используемого внутри функции указателя. За соответствие фактически переданных функции параметров ожидаемым типам отвечает программист, реализующий код вызова функции. При реализации функции, однако, нужно знать, что при передаче параметров типа char, short или перечисления (enum) они автоматически преобразуются в int, а float - в double.

В том случае, когда ожидаются параметры различных типов, при реализации функции нужно быть внимательным, и использовать при чтении параметров указатели соответствующих типов. Значительно упростить работу можно с использованием библиотеки <STDARG.H>. В ней для работы с функциями с неопределенным числом параметров определены макросы va_start, va_arg, va_end и тип va_list.

Использование этих макросов покажем на примере

```
double average( int n, ... )
{
   va_list vl;        // объявляем переменную специального типа
   va_start( vl, n ); // начинаем работу со списком нефиксир.
                      // параметров, n-последний фиксир.параметр
   double s = 0.0;
   for( i = 0; i < n; i++ )
      s += va_arg( vl, double );// доступ к очередному параметру
   va_end( vl );               // заканчиваем работу со списком
   return s/n;
}
```

4.5 Перегрузка функций

Перегруженными называют функции, объявленные в одной области видимости и имеющие одинаковые имена. Перегружаемые функции должны различаться типом, количеством входных параметров или наличием неопределенного числа аргументов. Как правило, перегруженные функции выполняют аналогичные действия для аргументов различных типов. Возможность перегрузки функций весьма удобна – программисту не нужно помнить названия различных функций, совершающих похожие действия, достаточно запомнить одно название. Ниже приведены примеры объявления перегруженных функций:

```
int print( int  ival );            // Печать целого
int print( double dval );          // Печать вещественного
int print( double dval, int prec );// … с указанной точностью
int print( const char *sval );     // Печать строки
```

При обращении к функции компилятор осуществляет поиск наиболее подходящей для вызова функции из числа перегруженных. При этом возможно как неполное совпадение типов, разрешаемое с использованием неявных преобразований типов, так и возникновение неоднозначности, какую из функций вызвать. В последнем случае компиляция заканчивается с ошибкой, устранить которую можно либо выполнив явные преобразования типов при передаче параметров, либо определив еще одну перегруженную функцию, более точно соответствующую по типам параметров, чем уже имеющиеся.

4.6 Указатели на функции

При создании современных программных систем их работа часто организуется на основе двустороннего взаимодействия входящих в их состав компонентов. В качестве примера такого взаимодействия можно привести управление элементами пользовательского интерфейса (вывод главного меню программы) и обработку событий (выбор пользователем пункта меню). Другим примером может быть создание программируемого таймера и реакция при наступлении заданного момента времени. Двусторонние взаимодействия возникают при отслеживании операций с файловой системой (создание или удаление файла), работе с сетью (отправка и поступление новой порции данных или сообщения) и т.п. Как правило, при проектировании программный код, реализующий логику реакции на некоторые события, отделяется от программного кода, отвечающего за генерацию таких событий. Более того, в качестве источников событий зачастую используются готовые программные компоненты, в частности, входящие в состав среды разработки. В этом случае такое разделение диктуется используемым набором готовых библиотек. Однако, даже в том случае, когда готовые библиотеки не используются, использование механизма событий дает возможность разделить компоненты, входящие в состав системы и повысить тем самым степень повторного использования программного кода.

Одним из основных способов организации двустороннего взаимодействия на процедурном уровне является использование *указателей на функции*. Синтаксически указатель на функцию объявляется следующим образом:

```
<тип> (*<название>) (<список параметров>);
```

Например, указатель на функцию

```
bool Equal( int x, int y);
```

объявляется следующим образом:

```
bool (*pf)( int x, int y);
```

В том случае, если указатель на функции предполагается использовать часто, лучше ввести соответствующий тип указателя

```
typedef bool (*PFunType)(int a, int b);
```
или самой функции
```
typedef bool FunType(int a, int b);
```
С использованием введенных типов объявление указателя на функцию будет выглядеть так:
```
PFunType pf;
```
или
```
FunType *pf;
```
Получить адрес функции можно с использованием уже знакомой нам операции «&».
```
pf = &Equal;
```
С использованием операций разыменования и вызова функции можно обратиться к функции через указатель:
```
bool res = *pf(3,4);
```
Вероятно, неожиданной особенностью будет то, что указатели на функции обрабатываются специальным образом и использовать операции получения адреса и разыменования необязательно. То есть корректным будет следующий код:
```
pf = Equal;
bool res = pf(3,4);
```
Для иллюстрации приведем пример универсальной функции поиска элемента в массиве, в которую передается функция сравнения элементов.
```
//вводим новый тип - указатель на функцию сравнения
typedef bool (*PCmpFn)(void*, void*);
//определяем универсальную функцию поиска элемента в массиве
void* Find(void* Ar, int Cnt, int ElmSz, void* Elm, PCmpFn Cmp)
{
    int i; char* p;
    for (i = 0, p=(char*)Ar; i < Cnt; i++, p+=ElmSz)
        if (Cmp(p, Elm))  return p;
    return NULL;
}
```
Для использования функции поиска с конкретным типом данных нужно лишь определить функцию сравнения элементов. Пусть определены следующие функции сравнения:
```
bool CmpDbl(void* pa, void* pb)
{  return *(double*)pa==*(double*)pb;  }
bool CmpPerson(void* pa, void* pb)
{   const char *ida=((Person*)pa)->PassportNo,
               *idb=((Person*)pb)->PassportNo;
    return (0==strcmp(ida, idb));
}
```
Тогда функция поиска может быть использована так:
```
double arDbl[4]={3.3,9.8,7.9,6.5};
Person arPers[2]={{"Иван","Иванов","3600123456","Ленина,1-1"},
                  {"Петр","Петров","3600987654","Авроры,1-1"}};
double *p, dblVal=9.8;
```

```
Person *q, person={"Петр","Иванов","3600111111","Кирова,1-1"};
p=(double*)Find(arDbl, 4, sizeof(double), &dblVal, &CmpDbl);
q=(Person*)Find(arPers, 2, sizeof(Person), &person, &CmpPerson);
```

Ключевое слово typedef

Ключевое слово typedef позволяет дать новое имя уже существующим или вновь вводимым типам. Синтаксически использование typedef выглядит следующим образом:

```
typedef <описание типа> <имя>;
```

После задания новое имя типа может использоваться всюду, где необходимо. Приведем несколько примеров.

```
typedef unsigned long ulong;
ulong ul = 9;

typedef char * PCHAR;
PCHAR p1, p2;


typedef int (*PF)(int a, int b);
int sum(int a, int b) {return a+b;}
PF pf = &sum;


typedef int FUN(int a, int b);
FUN *pf = &sum;


typedef double Matrix[4][3];
Matrix* arr = new Matrix[5];
arr[3][2][1]=321;
delete [] arr;
```

При одновременном описании исходного и производного типов можно использовать запятую:

```
typedef struct tagList {int info; List* link; } Item, *PItem;
```

4.7 Вопросы и задания

1. В чем отличие объявления и определения функции?

2. В чем заключается разница между функцией и процедурой?

3. Что означают спецификаторы static, extern и inline?

4. В чем отличие передачи параметра по ссылке и по значению?

5. Как задать значение параметра по умолчанию?

6. Как определить функцию с неопределенным числом параметров?

7. Что такое перегрузка функций?

8. Как определить указатель на функцию? Приведите пример без использования и с использованием typedef.

9. Напишите функцию, создающую копию входной строки в динамической памяти (аналог strdup).

10. Напишите функцию поиска подстроки в строке (аналог strstr).

11. Напишите функцию создания списка, содержащего буквы латинского алфавита в обратном порядке.

12. Напишите рекурсивную функцию создания списка, содержащего буквы латинского алфавита в прямом порядке.

13. Напишите функцию обхода бинарного дерева:

a) с подсчетом среднего арифметического всех узлов дерева

b) с подсчетом суммы элементов концевых узлов дерева

c) с определением высоты дерева

14. Напишите функцию с неопределенным числом аргументов, выполняющую создание и заполнение одномерного динамического массива значениями переданных в функцию параметров.

5 КЛАССЫ

Как было сказано во введении, данное учебное пособие условно состоит из двух частей, первая из которых была посвящена процедурному походу к программированию. Начиная с настоящего раздела, мы начнем осваивать объектно-ориентированный подход к программированию с использованием C++.

Центральными понятиями в объектно-ориентированном программировании (ООП) являются понятия объекта и ***класса***. Для иллюстрации этих понятий приведем следующие примеры:

класс - «студент», объект – студент Петр Иванович Сидоров из группы 619,

класс – «здание», объект – здание по адресу ул. Ленина, дом 1,

класс – «автомобиль», объект – автомобиль с идентификационным номером (VIN) 1M8GDM9AXKP042788.

Список примеров можно продолжать до бесконечности. Главное сейчас усвоить, что ***объект*** является экземпляром, представителем класса.

В зависимости от решаемой задачи информационное наполнение объектов может быть разным. Так, для системы автоматической регистрации транспортных средств важны госномер, марка и модель автомобиля, время регистрации, фотография конкретного автомобиля. Для автосалона – марка и модель, цвет, комплектация, цена, дата производства и поступления в продажу. Таким образом, можно сказать, что в зависимости от решаемой задачи объект в программе должен отражать лишь некоторую часть характеристик объекта реального мира, важную для рассматриваемой задачи.

Конечно, объекты в программе не обязаны быть связаны с объектами реального мира. Объектом, например, может быть элемент пользовательского интерфейса, соединение между клиентским и серверным программным обеспечением, диспетчер каких-либо ресурсов, используемых в самой программе (памяти, потоков управления) и т.д.

Следует отметить, что одно и то же понятие в тех или иных случаях может быть выделено или не выделено в отдельный класс. Например, для программы учета транспортных средств может не иметь большого значения, является ли автомобиль легковым или грузовым. Учет и тех и других выполняется одинаковым образом. Для транспортной компании, напротив, разница существенна, так как первые используются для пассажирских перевозок, а вторые – для перевозки грузов.

Последний пример также подводит нас к мысли о том, что полезно иметь возможность организовывать объекты в иерархии. Вы, конечно же, сталкивались с классификациями (таксономиями) при изучении зоологии (классификация животного мира), ботаники (классификация растительного мира) и т.д. В обычной жизни к понятию класса мы чаще всего приходим, изучая совокупность объектов со сходными характеристиками, а к понятию суперкласса (надкласса) – изучая свойства классов. Более того, в обычной жизни классы и суперклассы являются только понятиями и без составляющих их объектов вообще не могут быть определены. В программировании ситуация несколько иная. Здесь первичным понятием является именно класс, а не объект, так как создать объект мы можем лишь как экземпляр класса. Более того, иерархия классов может быть определена только сверху вниз. Мы можем определить подклассы на основе некоторого уже имеющегося класса, как наследники этого класса, а не наоборот. В ООП возможность создания иерархии классов поддерживается концепцией ***наследования***.

Конечно, также как и в обычной жизни, в программе объекты могут состоять из других объектов или быть связанными с ними. Так, автомобиль может состоять из отдельных частей (кузова, двигателя, ходовой и т.д.) и быть связанным со своим владельцем, маршрутом следования и т.д. В программировании такой объект, состоящий из других объектов, называется ***агрегатом***.

Безусловно, объекты имеют возможность взаимодействовать между собой, при этом воздействовать на объект можно только с использованием предоставляемого им ***интерфейса***. В реальной жизни это может быть мышь и клавиатура для компьютера, руль, педали и рычаг переключения передач для автомобиля, кнопки в лифте и т.д. В программе интерфейс взаимодействия определяется теми данными и функциями, которые доступны другим объектам.

Далеко не все данные и функции работы с ними могут быть доступны другим объектам. Несанкционированный доступ к внутренним данным объекта может нарушить логику его функционирования, что приведет к нежелательным последствиям. Чтобы исключить несанкционированное вмешательство, заводы изготовители ставят пломбы на технику. В ООП для этих целей служит концепция ***инкапсуляции***, предполагающая скрытие (помещение в капсулу) данных и функций объекта, не предназначенных для работы с объектом извне.

Другими словами, предполагается, что объект имеет некоторый интерфейс для взаимодействия с внешним миром, а все детали его внутренней организации скрыты. Это подобно тому, как пользуясь автомобилем, вы взаимодействуете с ним посредством руля, педалей, органов управления на панели приборов. Кроме того, возможно, вы доливаете жидкость в бачок омывателя и проверяете давление в шинах. Все, что происходит внутри, как

готовится и подается топливно-воздушная смесь в камеру сгорания, отводятся отработанные газы и т.д. остается от Вас скрыто.

Тем не менее, для прохождения регламентных работ Вы, вероятно, регулярно обращаетесь на станцию технического обслуживания. Там, без всякого сомнения, высококвалифицированные работники знают все о внутреннем устройстве и особенностях функционирования Вашего автомобиля. Собственно, именно благодаря такому квалифицированному вмешательству автомобиль и может верой и правдой отслужить положенный срок. Подобная схема работы с объектом в некоторой степени противоречит принципу инкапсуляции, и, безусловно, для многих объектов вмешательство извне действительно трудно веско обосновать. Однако, встречаются и такие задачи, где объекты должны быть тесно связаны, и в C++ есть возможность реализовать именно такую схему взаимодействия объектов. Для ее реализации используется механизм ***дружественных*** классов, методов и функций.

Вообще говоря, маловероятно, что Вы всегда ездили и будете ездить только на одном автомобиле, и, пересев в другой автомобиль, Вы совсем не удивитесь, обнаружив там руль и педали. Более того, Вы, без сомнения, найдете их и в седане, и в универсале, и даже в автобусе. Конечно, это кажется Вам само собой разумеющимся, но представьте себе на минуту, например, что в разных моделях автомобилей педали будут переставлены местами. Можно привести множество других примеров, где последствия изменения интерфейса взаимодействия не так трагичны, но, согласитесь, что в тех случаях, когда мы хотим достигнуть одного и того же результата или выполнить одни и те же действия, иметь один и тот же интерфейс взаимодействия удобно и естественно. В ООП идея такая идея организации единого интерфейса составляет основу концепции ***полиморфизма***.

Конечно, приведенные примеры позволяют сформировать лишь самое общее представление об объектно-ориентированном подходе к разработке программного обеспечения. Вполне возможно, Вам не до конца ясен смысл, вкладываемый в основные концепции ООП, которыми являются абстракция, инкапсуляция, наследование и полиморфизм. В настоящей главе мы рассмотрим, как эти концепции отражены в C++. Надеемся по окончании изучения пособия у Вас сформируется достаточно полное представление и об объектно-ориентированном подходе и об особенностях его применения с использованием C++.

5.1 Описание класса

Начинать изучение ООП на языке C++ следует конечно же с понятия класса. Класс в C++ представляет собой производный структурированный тип, содержащий совокупность данных и функций их обработки. Синтаксис описания класса выглядит следующим образом*:*

```
class [[<имя класса> [: <список базовых классов>]]
{
    <список членов класса>
} [[<список переменных>]];
```

Здесь <имя класса>- корректный идентификатор, <список базовых классов> - один или несколько базовых классов, от которых наследуется определяемый класс (более подробно об этом будет рассказано позднее), <список членов класса> - описывает поля и методы класса. Здесь же описываются дружественные классы, методы и функции, а также вложенные типы. Необязательный <список переменных> позволяет описать набор переменных описываемого класса.

В настоящем разделе нас более всего будут интересовать поля и методы класса, так как именно они определяют данные, связанные с классом, а также поведение объектов и действия, которые над ними определены. Следует отметить, что в C++ поля и методы классов часто называют «членами» классов: «*данными-членами*» и «*функциями-членами*» классов. В литературе также встречаются термины «*компонентные данные*» и «*компонентные функции*» классов.

Синтаксис при объявлении полей класса ничем не отличается от синтаксиса объявления полей структур. Объявление методов класса выполняется почти так же, как и объявление обычных функций. Разница заключается в используемых спецификаторах (explicit, virtual, inline, const, спецификатор чистоты «=0», исключений throw), о которых будет рассказано по мере изучения материала. Для начала рассмотрим пример очень простого класса вещественного числа:

```
class CDbl{
    double v;
    void Add( double x )  {  v += x;  }
};
```

Как видно из примера методы класса могут быть определены сразу же внутри класса. Однако, довольно часто при описании класса метод лишь объявляют, вынося его реализацию за пределы класса. При этом объявление и реализация могут содержаться в одном или разных файлах (например, объявление класса содержится в заголовочном *.h файле, а реализация в *.cpp файле). При реализации метода вне класса используется следующий синтаксис:

```
<тип> <имя класса>::<имя функции> (<список параметров>)
```

С его использованием приведенный выше пример может быть записан следующим образом:

```
class CDbl{
    double v;
    void Add( double x );
};
void CDbl::Add( double x )  {  v += x;  }
```

Методы класса так же, как и обычные функции могут быть перегруженными, иметь параметры по умолчанию или неопределенное число параметров.

Заметим, что определенный указанным выше образом класс бесполезен, и вовсе не потому, что не имеет никаких преимуществ перед базовым типом, а по той причине, что все члены этого класса недоступны для использования извне самого класса.

Дело в том, что доступ к членам класса регулируется спецификаторами доступа, которых в C++ три: private, protected и public. Спецификаторы доступа указываются в списке членов класса следующим образом:

<спецификатор доступа>:

Действие спецификатора распространяется до следующего спецификатора или до конца описания класса. Собственно, значение спецификаторов следующее:

private – закрытый член класса, доступ к такому члену класса имеют только члены-функции того же класса, а также дружественные функции, методы и классы;

protected – защищенный член класса, доступ к такому члену класса имеют не только члены-функции того же класса и дружественные функции, методы и классы, но и члены-функции классов наследников;

public – открытый член класса, доступ к такому члену класса имеется всюду, где имеется доступ к самому классу.

По умолчанию, члены класса считаются private членами, поэтому в приведенном выше примере все члены класса закрыты.

Рассмотрим теперь несколько более сложный пример многомерного вектора вещественных чисел.

```
class CVector{
  private:
    double *m_pData;
    int     m_iSize;
  public:
    void Init( int size )  {
        m_pData = new double[size];
        m_iSize = size;
    }
    void Uninit()  {
```

```
        delete [] m_pData;
    }
    double& Elm( int ind )  {  return m_pData[ind];  }
    int GetSize()  {  return m_iSize;  }
    void Add( const CVector &v )  {
        for ( int i = 0; i < m_iSize; i++ )
            m_pData[i] += v.m_pData[i];
    }
    void Mul( double m )  {
        for ( int i = 0; i < m_iSize; i++ )
            m_pData[i] *= m;
    }
};
```

В приведенном примере для хранения элементов вектора мы используем динамический массив m_pData, а количество элементов храним в поле m_iSize. Воспользовавшись спецификатором доступа private, мы скрыли от доступа извне оба этих члена класса, несколько защитив целостность хранимых в векторе данных. Действительно, если бы мы объявили эти данные как public, невнимательный программист мог бы модифицировать размер массива (поле m_iSize) без соответствующего перераспределения памяти или модифицировать сам указатель m_pData. Весьма вероятно, это привело бы к разрушительным последствиям не только для самого объекта, но и для программы в целом.

Все методы работы с вектором объявлены открытыми (public) и определяют интерфейс взаимодействия с нашим объектом. Функция Init предназначена для инициализации объекта и его приведения в согласованное состояние (когда размер выделенной памяти соответствует размеру вектора). Функция Uninit освобождает выделенную память. Функция Elm предназначена для обращения к элементу вектора, Add – для сложения векторов, Mul – для умножения числа на вектор.

Объявление объектов класса производится так же, как и объявление других переменных. Обращение к членам класса производится с использованием операций доступа «.» или «->». Первая из операций применяется к объектам, а вторая – к указателям на объекты:

```
CVector x, y;
x.Init(2);
y.Init(2);
x.Elm(0) = 1.1;     x.Elm(1) = 1.2;
y.Elm(0) = 2.1;     y.Elm(1) = 2.2;
x.Add(y);
CVector *px=&x;
px->Mul(3.3);
printf( “x=(%f; %f)”, x.Elm(0), px->Elm(1) );
x.Uninit();
y.Uninit();
```

Из методов самого класса обращаться к членам класса можно напрямую. Так, из метода Add мы обращались к членам m_iSize и m_pData не используя никаких других идентификаторов. При этом в качестве значений соответствующх полей берутся значения именно того объекта, для которого вызывается метод. Так, при вызове функции-члена Add класса CVector для объекта x, в качестве m_iSize и m_pData брались именно данные объекта x. Для доступа к данным объекта y, переданного в качестве параметра мы пользовались квалифицированным идентификатором v.m_pData[i]. Таким образом, в случае, если не указано иное, при обращении из метода класса к какому-либо члену класса происходит обращение к данным того объекта, относительно которого выполняется вызванный метод.

При выполнении той или иной функции-члена класса в ряде случаев требуется знать, для какого именно объекта выполняется метод. Для таких целей служит *указатель this*, доступный для всех нестатических методов класса. Этот указатель имеет тип:

```
<имя_класса>* const this;
```

и указывает на объект, для которого происходит выполнение метода. Можно считать, что указатель this неявно передается при вызове каждого метода класса в качестве первого скрытого параметра. С использованием этого указателя метод Add может быть реализован следующим образом:

```
void CVector::Add(/*CVector *const this,*/ const CVector &v)
{
    for ( int i = 0; i < this->m_iSize; i++ )
        this->m_pData[i] += v.m_pData[i];
}
```

Примечание: скорее всего, покажется удивительным тот факт, что структуры struct и объединения union также являются в C++ классами. Разница заключается, в том, что по умолчанию поля структур и объединений считаются public полями, а поля классов – private полями. Тем не менее, из общих соображений можно рекомендовать использовать ключевое слово class тогда, когда у объектов присутствуют такие признаки, как сокрытие членов класса, полиморфное поведение и т.д.

5.2 Конструкторы и деструкторы

Внимательный анализ класса CVector говорит о том, что, несмотря на использование спецификаторов доступа, класс остается недостаточно защищенным, и мы слишком полагаемся на корректные действия программиста-пользователя класса. Так, ничто не мешает обратиться к Init несколько раз подряд, тем самым вызывая утечку памяти, или, наоборот, обратиться к другим функциям объекта, не вызвав Init вовсе. Невнимательный программист также может сложить вектора разных длин или обратиться к несуществующему элементу с использованием

Elm. И если в последнем случае можно избежать фатальных последствий, добавив проверку индексов и длин векторов, то инициализация и деинициализация объектов – более серьезная проблема, которая с использованием конструкторов и деструкторов может быть решена достаточно элегантным образом.

5.2.1 Определение конструкторов и деструкторов

Конструкторы и деструкторы - это функции класса специального вида. *Конструкторы* служат для инициализации данных объекта и вызываются при создании объекта автоматически. Конструкторы имеют то же имя, что и сам класс, не имеют никакого выходного значения.

Объявление конструктора выглядит следующим образом:

```
<имя класса>(<параметры конструктора>);
```

Для класса вполне допустимо определить набор конструкторов с различными параметрами. В этом случае говорят о перегрузке конструкторов.

Деструктор служит для освобождения ресурсов, выделенных при создании и накопленных за время жизни объекта. Деструкторы вызываются автоматически при уничтожении объектов. Дестукторы не возвращают значений, не имеют параметров и не могут быть перегружены. Объявление деструктора выглядит следующим образом:

```
~<имя класса>();
```

С использованием конструктора и деструктора приведенный выше пример с многомерным вектором действительных чисел будет выглядеть следующим образом:

```
class CVector{
  private:
    double *m_pData;
    int     m_iSize;
    void Init( int size )  {
        m_pData = new double[size];
        m_iSize = size;
    }
  public:
    CVector( int size )  {
        Init( size );
    }
    ~CVector()  {
        delete [] m_pData;
    }
    double& Elm( int ind );
    int GetSize();
    void Add( const CVector &v );
    void Mul( double m );
};
```

Здесь мы оставили функцию Init, переместив ее в закрытую секцию, так как будем пользоваться ею в дальнейшем. Ничто не мешает, однако, удалить ее, перенеся реализующий ее код в конструктор.

В том случае, если в классе не определен ни один конструктор, то компилятором будет автоматически сгенерирован конструктор без параметров. Аналогично, если отсутствует определение деструктора, будет автоматически сгенерирован деструктор. Это делает возможным создание и удаление объектов для которых конструкторы и деструкторы не были определены. Не следует думать, однако, что автоматически созданные конструкторы и деструкторы выполняют сколь-нибудь полезные действия за исключением вызова конструкторов и деструкторов базовых классов.

На практике часто возникает потребность инициализировать объекты разными способами. В этом случае возможно определить несколько конструкторов. Так, для приведенного выше примера могут быть полезны следующие конструкторы.

```
CVector( int size, double v )  {
    Init( size );
    for (int i = 0; i < m_iSize; i++)  m_pData[i] = v;
}
CVector(double *data, int size )  {
    Init( size );
    for (int i = 0; i < m_iSize; i++)  m_pData[i] = data[i];
}
```

Первый из них позволят заполнить элементы вектора некоторым значением, второй – заполнить вектор элементами вещественного массива.

Для указания параметров конструктора при создании объекта используется инициализация объекта в стиле функции. Так, для класса CVector возможны:

```
CVector x1(2);
CVector x2(2, 1.0);
double m[] ={1.1,2.2,3.3};
CVector x3(m, sizeof(m)/sizeof(m[0]));
```

Допустим также вариант с инициализацией следующего вида:

```
CVector y1 = CVector(2);
CVector y2 = CVector(2, 1.0);
CVector y3 = CVector(m, sizeof(m)/sizeof(m[0]));
```

В том случае, если объекты создаются в динамической памяти, то:

```
CVector *p1 = new CVector(2);
CVector *p2 = new CVector(2, 1.0);
CVector *p3 = new CVector(m, sizeof(m)/sizeof(m[0]));
```

Отметим, что создать вектор с использованием одного из следующих способов невозмножно:

```
CVector x;
CVector *p = new CVector;
```

Дело в том, что в классе нет конструктора без параметров, но определены перегруженные конструкторы, что исключает возможность создания такого конструктора компилятором автоматически. Конструкторы без параметров называются также *конструкторами по умолчанию*. Отсутствие такого конструктора делает невозможным не только содание одиночных объектов приведенными выше способами, но и создание массивов объектов, например, одним из следующих способов :

```
CVector ar[10];
CVector *ar = new CVector[10];
```

В нашем случае не обязательно определять еще один конструктор, достаточно указать значение по умолчанию для уже имеющегося:

```
CVector( int size = 2 )  {
    Init(size );
}
```

Дело в том, что конструктор, при объявлении которого заданы значения всех параметров по умолчанию также является конструктором по умолчанию. В нашем случае по умолчанию будет создаваться двумерный вектор.

Интересным является тот факт, что конструктор с одним параметром может рассматриваться как оператор приведения типа. Действительно, если переменная v имеет тип CVector, то корректными будут, например, следующие выражения:

```
v=CVector(2);
v=(CVector)2;
v=2;
```

Во всех приведенных случаях перед выполнением операции присваивания с испоьзованием конструктора с одним параметром конструировались временные объекты типа CVector.

Перед тем как перейти к следующему разделу напомним, что *время жизни* объектов определяется тем, в какой памяти объект размещается. Объекты в автоматической памяти создаются в том месте, где они объявляются, а уничтожаются при выходе из того блока, в котором они объявлены. Объекты, размещаемые в статической памяти создаются при запуске программы. а уничтожаются при ее завершении. Объекты, размешаемые в динамической памяти создаются при обращении к операции new, а уничтожаются при обращении к операции delete. Во всех случаях вызов конструкторов и деструкторов происхоит автоматически соответственно при создании и уничтожении объектов.

Следует отметить, что функции стандартной библиотеки для работы с динамической памтью (malloc, free и др.) не могут быть использованы для создания и уничтожения объектов класса, так как при их использовании конструкторы и деструкторы не вызываются.

```
CA* pA = new CA;    // выделение памяти и вызов конструктора
```

```
CA* pB = (CA*)malloc(sizeof(CA)); // только выделение памяти
…
delete pA;     // освобождение памяти и вызов деструктора
free(pB);      // только освобождение памяти
```

5.2.2 Конструктор копирования

На самом деле роль конструкторов и деструкторов более важна, чем может показаться на первый взгляд. Рассмотрим, например, следующую функцию:

```
CVector Norm( CVector v )
{
    CVector r = v;
    double d = 0.0;
    for (int i = 0; i < r.GetSize(); i++)
        d+= r.Elm(i)* r.Elm(i);
    r.Mul(1.0 / sqrt(d));
    return r;
}
```

Как Вы думаете, сколько раз будут вызваны конструкторы и деструкторы при выполнении следующей строки:

```
y = Norm( x );
```

Конструктор и деструктор будут вызваны по три раза. Первый раз конструктор будет вызван при копировании объекта x в автоматическую память в качестве параметра функции при вызове Norm(x). Второй раз – при выполнении оператора CVector r = v. Здесь создается копия объекта v. Третий раз конструктор будет вызван при выполнении return r. Здесь будет создана временная копия объекта r для возврата значения из функции.

Во всех этих случаях будет вызван так называемый *конструктор копирования*, который нами еще определен не был. Но в том случае, если конструктор копирования не объявлен явным образом, компилятор автоматически создает конструктор побитового копирования.

Вообще говоря, автоматически созданный конструктор, выполняющий побитовое копирование, в подавляющем большинстве случаев приносит больше вреда, нежели пользы. В частности, в рассмотренном примере конструктор копирования копирует адрес динамического массива и его размер. Такое поведение конструктора приводит к тому, что создаваемые экземпляры содержат m_pData, указывающий на ту же область памяти, что и m_pData исходного объекта. При удалении исходного объекта или любой из его копий память, отведенная под m_pData, будет освобождена деструктором, а оставшиеся объекты продолжат работать с этой памятью, что чревато серьезными ошибками при работе программы. Чтобы избежать неприятных последствий конструктор копирования следует определить явно. Отметим, что при выполнении операции присваивания (y = Norm(x)), возникают похожие проблемы, перегрузку этой операции мы рассмотрим в соответствующем разделе.

Объявляется конструктор копирования следующим образом:

```
<имя класса>(const <имя класса> &<имя параметра>)
```

Определение конструктора копирования для нашего примера выглядит следующим образом:

```
CVector(const CVector& src)
{
    Init( src.m_iSize );
    for (int i = 0; i < m_iSize; i++)
        m_pData[i] = src.m_pData[i];
}
```

В том случае, когда использование конструктора копирования не предполагается, следует хотя бы скрыть конструктор копирования, применив спецификатор доступа private. В этом случае компилятор не допустит копирования объектов вне класса, что поможет избежать непредвиденных ошибок.

В конце раздела напомним еще раз, в каких случаях вызывается конструктор копирования:

1. При инициализации объекта существующим объектом того же типа:

```
CA a;           // a – объект некоторого класса CA
…
CA b(a);        // вызов конструктора копирования
CA c = a;       // вызов конструктора копирования
```

2. При передаче параметра в функцию по значению, например:

```
void f( CA param );      // функция принимает объект класса CA
…
f( a );                  // вызов конструктора копирования
```

(после завершения объект-копия будет удален, т.е. вызван деструктор)

3. При возврате значения из функции для хранения возвращаемого значения неявно создается временный безымянный объект, который будет удален, как только значение будет возвращено.

```
CA f()              // функция возвращает объект класса CA
{
    CA a;
    …
    return a;       // вызов конструктора копирования
}
```

5.3 Статические члены класса

Во всех примерах, рассмотренных до сих пор данные и методы связывались с отдельными объектами, однако при решении некоторых задач бывает нужно, чтобы некоторые данные и функции были связаны с классом в целом. Для этой цели в C++ предусмотрены статические

члены класса. Они могут быть использованы для работы с некоей метаинформацией о классе, для концентрации совместно используемых объектами класса ресурсов и т.д.

Например, если Вы создаете класс каких-либо элементов пользовательского интерфейса (кнопок, меню и т.д.), возможно, Вам покажется излишней возможность отрисовки элементов разными цветами. В этом случае, скажем, цвета активных и неактивных пунктов меню будут задаваться статическими переменными класса для всех меню Вашей программы. В том случае, если Вы реализуете класс, связанный с математическими или физическими вычислениями, вероятно, будет удобно объявить соответствующие константы статическими членами класса. Если у Вас в программе устанавливаются соединения по сети, будет полезно иметь общую статистику объема переданных (полученных) данных, среднюю скорость передачи и т.д.

Как было сказано выше, статическими могут быть как поля, так и функции класса. Объявляются *статические члены класса* с модификатором *static*. Как любые статические переменные, статические переменные-члены класса создаются в начале работы программы и существуют до закрытия программы. Их время жизни никак не связано с временем жизни отдельных объектов класса.

Статические функции класса также не связываются с конкретными объектами класса. Статические функции класса могут обращаться к другим статическим функциям и данным класса, однако, для обращения к данным или методам конкретного объекта класса такой объект должен быть передан в статическую функцию явным образом. Обращение же к статическим членам класса возможно как из функций класса (статических или нет), так и извне класса с использованием названия класса и оператора «::» или любого объекта класса:

```
<имя класса>::<статический член класса>
<объект класса>.<статический член класса>
```

Так как статические переменные не связаны с объектами класса, то их инициализацию не следует выполнять в конструкторе. Вместо этого инициализация статических переменных класса производится в основной программе так же, как инициализация глобальной переменной (здесь же фактически происходит размещение статической переменной класса):

```
<тип> <имя класса>::<статическая переменная класса>=
<инициализатор>;
```

В качестве примера покажем, как можно реализовать подсчет количества существующих объектов класса CVector:

```
class CVector{
  private:
    double *m_pData;
    int     m_iSize;
    static int m_iCount;
    void Init( int size )  {
        m_pData = new double[size];
```

```
        m_iSize = size;
        m_iCount++;
    }
  public:
    ~CVector()  {
        delete [] m_pData;
        m_iCount--;
    }
    static int GetCount() {
        return m_iCount;
    }
    …
};
```

Теперь для того, чтобы узнать, сколько в настоящий момент времени существует объектов рассматриваемого класса, достаточно обратиться к статической функции класса:

```
int iCurrentCount = CVector::GetCount(0);
```

До настоящего момента создание нового объекта мы неразрывно связывали с обращением к конструктору. Однако иногда более удобно было бы обратиться к некоей статической функции, возвращающей вновь созданный объект, чем к конструктору. Например, в классе CVector мы могли бы ввести следующие функции, возвращающие новые нулевой и единичный вектора:

```
class CVector{
    …
  public:
    static CVector Zero( int size = 2)  {
        return CVector(size, 0.0);
    }
    static CVector Unit(int unit_ind, int size = 2 )  {
        CVector res(size, 0.0);
        res.m_pData[unit_ind]=1.0;
        return res;
    }
    …
};
```

Создание нулевого или единичного вектора теперь выглядит более естественно

```
CVector::Zero()
```

Такие статические функции могут рассматриваться как некая имитация конструктора, имеющего имя. По это причине, статические функции, возвращающие вновь созданный объект, иногда называют *именованными конструкторами*.

5.4 Встраиваемые методы класса

Так же как и обычные функции, функции-члены класса могут быть встраиваемыми. Для этого их объявляют с модификатором *inline* либо определяют непосредственно в описании класса (в этом случае реализация функции следует сразу же за ее прототипом в описании

класса). Как и в случае с обычными функциями, inline функция-член класса в зависимости от ее реализации может быть воспринята компилятором, как обычная функция-член класса.

5.5 Константные методы класса

Мы уже обсуждали ранее значение модификатора const при объявлении переменных и говорили о том, что такое объявление гарантирует, что объект не будет изменен. Для объектов классов дело обстоит немного сложнее в связи с тем, что компилятору необходимо запрещать не только изменение данных, но и вызовы методов, в которых данные объекта могут быть изменены. Для того чтобы отличать те методы, которые не меняют данные объекта от остальных методов, используют все тот же модификатор *const*. Этот модификатор записывается после списка параметров и должен присутствовать как в объявлении, так и в определении функции-члена класса.

На реализацию константных методов класса накладывается ряд очевидных ограничений: такие методы не могут изменять значения полей объекта и вызывать неконстантные методы класса. При вызове неконстантного метода для константного объекта компилятором должна быть сгенерирована ошибка (или, по крайней мере, предупреждение).

Для класса CVector очевидным кандидатом для того, чтобы стать константным, является метод GetSize(), а вот метод Elm(int ind) константным быть не может, так как возвращает неконстантную ссылку на элемент вектора. По этой причине определение, например, функции разности векторов следующего вида некорректно:

```
CVector Sub( const CVector &v1, const CVector &v2 )
{
    CVector r( v1.GetSize() );
    for ( int i = 0; i < v1.GetSize(); i++ )
        r.Elm(i)= v1.Elm(i)- v2.Elm(i);
    return r;
}
```

Чтобы такая функция могла быть определена, изменим функцию GetSize() и определим новую константную функцию GetElem(int ind).

```
class CVector{
    …
    double GetElm( int ind ) const {  return m_pData[ind];  }
    int GetSize()  const {  return m_iSize;  }
    …
};
```

Конечно же, теперь функция Sub должна быть немного изменена

```
CVector Sub( const CVector &v1, const CVector &v2 )
{
    CVector r( v1.GetSize() );
    for ( int i = 0; i < v1.GetSize(); i++ )
```

```
        r.Elm(i)= v1.GetElm(i)- v2.GetElm(i);
    return r;
}
```

В качестве еще одного примера константной функции можно привести функцию вычисления длины вектора.

```
class CVector{
    …
    double Length()  const {
        double r = 0.0;
        for (int i = 0; i < m_iSize; i++ )
            r += m_arData[i]* m_arData[i];
        return sqrt( r );
    }
    …
};
```

На первый взгляд может показаться, что само использование константных объектов и методов лишь усложняет разработку. На самом же деле потраченные на грамотную проработку интерфейсов классов и функций усилия окупятся многократно, так как многие ошибки будут обнаруживаться еще на этапе компиляции, что облегчит все последующие этапы создания продукта.

С константными функциями связано использование еще одного спецификатора – *mutable*. Этот спецификатор указывается для некоторого поля класса в том случае, если изменения этого поля в константном методе избежать не удается. Спецификатор mutable используется довольно редко. В частности, он может быть использован для полей, в которых кэшируются результаты ресурсоемких вычислений, в то время как сами вычисления не изменяют данные объекта.

5.6 Дружественные функции, классы и методы

При разработке системы связанных друг с другом классов может возникнуть необходимость обращения к закрытым или защищенным членам классов из внешних функций или классов. В этом случае такие функции, методы или классы можно объявить *дружественными*. Конечно же, прежде чем принять такое решение, следует взвесить все «за» и «против».

Представим, например, что кроме класса CVector у нас имеется класс CMatrix

```
class CMatrix  {
  private:
    int m_iH, m_iW;
    double *m_arData;
  public:
    CMatrix( int H, int W );
    ~CMatrix( int H, int W );
    int GetW() const;
```

```
    int GetH() const;
    double &Elm(int m, int n);
    double GetElm(int m, int n);
};
```

В этом случае было бы полезно иметь функцию умножения матрицы на вектор.

```
CVector Mul(const CMatrix &, const CVector &);
```

Эту функцию можно реализовать вне обоих классов, используя функции доступа к отдельным элементам. Конечно же, это будет не самым эффективным решением. Однако сделать эту функцию членом сразу обоих классов, так, чтобы она имела доступ к закрытым данным этих классов, мы не можем. Но мы можем объявить эту функцию дружественной этим классам.

Объявляются дружественные функции, методы или классы внутри самого класса с использованием модификатора *friend*. Такое объявление может быть сделано в любом месте объявления класса.

```
class CVector  {
    ...
    friend CVector Mul(const CMatrix &, const CVector &);
};
class CMatrix  {
    ...
    friend CVector Mul(const CMatrix &, const CVector &);
};
```

Реализация этой функции может быть следующей (проверка соответствия размеров массива и вектора опущена):

```
CVector Mul(const CMatrix &M, const CVector &V)
{
    CVector r(M.m_iH);
    double *p=M.m_arData, *q, *t=r.m_arData,
           *qf = V.m_arData+V.m_iSize, *tf = r.m_arData+r.m_iSize;
    while (t != tf)  {
        q = V.m_arData;
        *t = 0.0;
        while (q != qf)  *t+=(*p++)*(*q++);
        t++;
    }
    return r;
}
```

В том случае, если в классе CMatrix нам необходимо реализовать доступ к строке или столбцу, как вектору, мы могли бы объявить такие функции следующим образом:

```
CVector GetRow( int ind );
CVector GetCol( int ind );
```

И в том случае, если для их реализации целесообразно иметь доступ к скрытым данным класса CVector, мы можем объявить методы дружественными:

```
class CVector  {
```

```
    ...
    friend CVector CMatrix ::GetRow( int );
    friend CVector CMatrix ::GetCol( int );
};
```

Если дружественных методов некоторого класса много, то вместо того, чтобы перечислять их, можно объявить этот класс дружественным целиком.

```
class CVector  {
    ...
    friend CMatrix;
};
```

Следует отметить, что объявление метода дружественным не может быть выполнено до того, как описан класс, в котором сам этот метод объявлен. Поэтому класс, в котором объявлен дружественный метод, должен быть описан раньше класса, в котором метод объявлен дружественным. Для приведенного выше примера это означает, что класс CMatrix должен быть описан раньше класса CVector. Реализация самого метода может быть выполнена как вне классов, так и внутри одного из классов. Важно лишь, чтобы все используемые в реализации поля и методы классов были уже определены.

5.7 Вопросы и задания

1. Дате определения понятиям класс и объект. Приведите примеры.

2. Как описывается класс на языке C++? Приведите пример.

3. Как Вы понимаете концепцию инкапсуляции? Какие средства языка ее поддерживают?

4. Каково предназначение конструкторов и деструкторов? Как они определяются в C++?

5. Что такое конструктор по умолчанию?

6. В каких случаях производится вызов конструктора копирования?

7. В чем отличие статических членов класса от нестатических?

8. Какие особенности реализации имеют константные методы класса?

9. Для чего предназначен механизм дружественных функций, методов и классов? Приведите примеры, где его использование может быть целесообразно.

10. Опишите класс для работы со списком пар «целочисленный ключ - вещественное значение». Реализуйте конструктор по умолчанию, конструктор копирования, деструктор, функции добавления новой пары, поиска значения по ключу и удаления пары с заданным ключом. Реализуйте функцию добавления таким образом, чтобы список был упорядочен по возрастанию ключей.

11. Опишите класс для работы с дихотомическим деревом (деревом поиска). Каждый узел дерева должен содержать пару «целочисленный ключ – строковое значение». Реализуйте

конструктор по умолчанию, конструктор копирования, деструктор, функции добавления новой пары, поиска значения по ключу и удаления пары с заданным ключом.

12. Для класса из упражнения 10 опишите дружественный класс, позволяющий осуществлять однонаправленный проход по списку (итератор). Реализуйте конструктор с параметром (параметр - объект класса из упражнения 10), функции перехода к первому элементу списка, следующему элементу, последнему элементу, функцию проверки конца списка, а также функции доступа к ключу и значению.

6 ПЕРЕГРУЗКА ОПЕРАЦИЙ

Конечно же, использовать определенные выше функции работы с векторами не слишком удобно. Более естественным и компактным было бы использовать привычные для нас операции сложения, вычитания, умножения, обращаться к элементу вектора с использованием квадратных скобок, а сравнивать вектора с использованием операций сравнения на равенство и неравенство. К счастью, в C++ все это осуществимо путем перегрузки стандартных операций. Перегрузка операций позволяет распространить действие стандартных операций языка на классы пользователя.

6.1 Общие сведения

Для того чтобы перегрузить операцию следует объявить функцию или функцию-член класса следующего вида:

```
<тип> operator <символ> (⟦<список параметров>⟧);
```

Здесь <тип> - тип результата операции, <символ> - символьное обозначение операции, <список параметров> - параметры функции перегрузки операции. Количество параметров в списке зависит от того является ли операция ***унарной*** или ***бинарной***, а также от способа перегрузки операции. В случае, когда перегрузка выполняется функцией вне пользовательского класса, количество параметров функции совпадает с количеством операндов перегружаемой операции. Так, для унарной операции <символ> <аргумент>, перегружаемой вне класса функция имеет единственный параметр

```
<тип> operator <символ> (<тип1> <парам1>);
```

Для бинарной операции

```
<аргумент1> <символ> <аргумент2>,
```

перегружаемой вне класса, функция имеет следующий вид:

```
<тип> operator <символ> (<тип1> <парам1>, <тип2> <парам2>);
```

Например, для операции сложения векторов:

```
CVector operator +(const CVector &v1, const CVector &v2);
```

В случае, когда перегрузка выполняется функцией-членом класса количество параметров функции на единицу меньше числа операндов перегружаемой операции. В этом случае первый из операндов операции передается в функцию-член класса в качестве указателя this и отсутствует в списке параметров функции. Для унарной операции, перегружаемой внутри класса, функция имеет следующий вид:

```
<тип> operator <символ>();
```

Для бинарной операции, перегружаемой внутри класса, функция имеет один параметр:

```
<тип> operator <символ>( <тип2> <парам2>);
```
Например:
```
CVector {
…
    CVector operator +( const CVector &v2);
};
```
Из сказанного следует, что операция, в которой первый аргумент не является объектом пользовательского класса, не может быть перегружена функцией-членом класса, а только внешней по отношению к классу функцией. К сожалению, ввести новое символьное обозначение для операции нельзя. Также нельзя перегрузить оператор для базовых типов данных.

Приведенные выше правила справедливы для большинства стандартных операций языка, однако некоторые из операций имеют особенности. Так не допускается перегрузка следующих операций:

- ::
- ?:
- .*
- .
- typeid
- sizeof

Существуют операции, которые могут быть перегружены только с использованием нестатических функций-членов классов. Это операции =, (), [], ->.

Операция *присваивания*. Как мы уже сказали, операция присваивания может быть перегружена только функцией-членом класса. Операция присваивания – единственная из всех перегружаемых операций не наследуется классами-потомками. Однако, в том случае, если операция не определена для класса явным образом, компилятором будет создана реализация операции присваивания по умолчанию, которая будет выполнять почленное копирование объектов класса. Как и в случае с конструктором копирования, автоматическая реализация операции присваивания может принести больше вреда. Поэтому, операцию присваивания для классов, содержащих данные в динамической памяти, лучше корректно перегрузить, или хотя бы закрыть. При реализации операции присваивания следует иметь в виду, что возможен случай самоприсваивания объектов, когда оба аргумента операции являются на самом деле одним и тем же объектом.

Операция *обращения по индексу* [ ] является бинарной и может быть перегружена только как функция-член класса. В качестве второго операнда допускается объект любого типа, а не обязательно целое число. Так, для класса, содержащего набор пар «строковый (const char *)

ключ – вещественное (double) значение», операция доступа по индексу может быть определена как

```
double& operator[](const char*);
```

Операция *вызова функции* «( )». Эта операция рассматривается как бинарная операция, второй аргумент которой представляет собой список параметров вызова (содержащий ноль или более параметров). Перегружается операция также только как функция-член класса. При перегрузке следует иметь в виду, что в зависимости от решаемой задачи функция-член класса может иметь как набор фиксированных параметров, так и иметь неопределенное число параметров. Конечно же, перегруженная операция применяется только к объектам класса, а не к функциям, как ее базовая реализация. В качестве примера можно привести перегрузку этой операции для обращения к элементу матрицы:

```
class CMatrix { …
    double operator () (int m, int n);
};
```

Операции *преобразования типа* (функции преобразования). Операции преобразования типа позволяют выполнять явное преобразование объекта класса к любому другому типу. Для перегрузки операции преобразования типа объявляется особая нестатическая функция-член класса вида:

```
operator <тип> ();
```

Здесь функция-член класса не имеет никаких параметров, тип возвращаемого значения для таких функций также не указывается, а определяется частью объявления <тип>. В части <тип> помимо собственно типа могут фигурировать спецификаторы типа и обозначения указателя (например, const char*). Перегруженные операции приведения типа наследуются и могут быть переопределены в классах потомках. При этом будут переопределены только операции с полностью совпадающей сигнатурой. Например, если в предке были определены функции operator float () и operator double (), то определенная в потомке operator float() переопределит только первую из функций, а вторая функция будет унаследована потомком.

Операции *обращения к члену класса* (->) и *разыменования* указателя (*) перегружаются сравнительно редко. Перегрузка этих операции используется для создания класса «*умных указателей*» (smart pointers). Кроме того, с использованием операции обращения к члену класса может быть реализована концепция делегирования. Мы не будем здесь подробно останавливаться на перегрузке этих операций.

Операции *инкремента и декремента* (++ и --). Особенность перегрузки этих операций связана с тем, что обе операции могут быть как префиксными, так и постфиксными. Для того чтобы отличить префиксные и постфиксные операции друг от друга, для постфиксных операций вводится дополнительный целочисленный параметр, который, как правило, не

используется. Поэтому при объявлении в классе рассматриваемые префиксные операции имеют вид:

```
<тип> operator ++();
<тип> operator --();
```

Соответствующие постфиксные операции:

```
<тип> operator ++ (int);
<тип> operator ++ (int);
```

При объявлении вне класса в функции первым параметром добавляется объект пользовательского класса, относительно которого выполняется операция.

При вычислении выражений дополнительный целочисленный параметр принимает значение 0. Передать иное значение можно только при явном вызове операции

```
<объект>.operator++(<значение>);
```

Такой способ вызова, однако, практически не используется.

При перегрузке любых операций часто возникает вопрос о том, как должен передаваться в функцию тот или иной объект – по ссылке или по значению. Аналогичные вопросы возникают и в отношении возвращаемого значения. Здесь можно дать следующие общие рекомендации.

Если предполагается, что операнд при выполнении операции не изменяется (например, операнды в операциях +, -, &&, ||), то в функцию его лучше передавать по значению или константной ссылке. Причем, если операндом является объект, занимающий большой объем памяти, то его лучше передавать по константной ссылке. Для базовых типов данных вполне подойдет передача по значению.

Если операнд изменяется при выполнении операции (например, первый операнд в операциях +=, &=, ++), то такой операнд должен передаваться по неконстантной ссылке.

Когда мы говорим о возвращаемом значении, то здесь можно придерживаться такого правила. Если возвращается объект, который не существовал до выполнения операции (например, результат операции +,*,|), то такой объект лучше возвращать по значению. В том случае, если объект уже существовал до выполнения операции (например, ++,+=,[]), то такой объект нужно возвращать по ссылке. Конечно, выше приведены лишь общие рекомендации и в некоторых случаях, например, в целях оптимизации, можно откланяться от них. Для того чтобы лучше разобраться в особенностях перегрузки операций, рассмотрите приведенный ниже пример.

6.2 Примеры перегрузки операций

Здесь мы приводим пример пользовательского класса для работы со строками. Для класса определяются все необходимые конструкторы и деструктор, а также следующие операции, которые могут быть полезны при работе с цепочками символов:

= (бинарная) - присваивание строк;

==, != (бинарная) - сравнение строк на равенство и неравенство, строки считаются равными, если они содержат одинаковые символы, следующие в том же порядке;

[] (бинарная) – обращение к символу строки по индексу;

+ (бинарная) - конкатенация (сложение) двух строк, новая строка содержит все символы первой строки, за которыми следуют символы второй строки;

+= (бинарная) – составная операция, дописывает символы второй строки к первой;

^ (бинарная) – итерация строки, новая строка содержит определенное число раз записанные друг за другом копии исходной строки;

! (унарная) – обращение строки, новая строка получается из исходной переписыванием символов исходной строки в обратном порядке;

операция приведения к типу const char*;

() – получение из исходной строки новой строки, состоящей из символов с указанными индексами. .

```
class CMyString
{
    private:
        char* m_data;
        // конструктор с параметром
        CMyString( int SymbCnt )
        { m_data = (char*)malloc(Cnt+1); };
    public:
        // конструктор по-умолчанию
        CMyString() : m_data(NULL) {};
        // конструктор копирования
        CMyString( const CMyString &Src )
            { m_data = strdup(Src.m_data); };
        // конструктор с параметром
        CMyString( const char* Src ) { m_data = strdup(Src); };
        // деструктор
        ~CMyString() {  if ( m_data )  free(m_data); }
        // функция определения длины строки
        int Len() const {
            if ( m_data ) return strlen( m_data );
            else return 0;
        }
        // перегрузка операции присваивания
        CMyString& operator = ( const CMyString & Src )  {
            if ( this == (&Src) )  return *this;
            if ( m_data )  free(m_data);
            if ( Src.m_data )  m_data = strdup( Src.m_data );
            else  m_data = NULL;
            return *this;
        }
```

```
// перегрузка операции сравнения
int operator == ( const CMyString & Src )  {
    if ( m_data && Src.m_data )
        return ( 0 == strcmpi(m_data, Src.m_data) );
    else if ( m_data == Src.m_data )  return 1;
    else return 0;
}
// перегрузка операции обращения по индексу
char& operator[](int ind) {
    return m_data[ind];
}
// перегрузка операции сложения (конкатенация строк)
CMyString operator + (CMyString& s)  {
    if ( ! Len() ) return s;
    if ( ! s.Len() )  return *this;
    CMyString res;
    res.m_data = (char*)malloc( Len() + s.Len() + 1 );
    strcpy( res.m_data, m_data );
    strcat( res.m_data, s.m_data );
    return res;
}
// перегрузка операции исключающее ИЛИ (итерация)
friend CMyString operator^( const CMyString& Src,
                            int Pow) {
    int iSz = Pow * Src.Len() + 1;
    CMyString s;
    if ( (Pow <= 0) || (0 == Src.Len()) ) return s;
    char* p = s.m_data = (char*)malloc(iSz);
    *p = 0;
    for (int i=0; i < Pow; i++)  strcat(p, Src.m_data);
    return s;
}
// перегрузка унарного оператора NOT (обращение)
CMyString operator ! ()  {
    CMyString res;
    if ( ! Len() )  return res;
    res.m_data = (char*)malloc( Len() + 1 );
    char *p = res.m_data + Len(),
         *q = m_data;
    *p-- = 0;
    while ( *q != 0 ) *p-- = *q++;
    return res;
}
// Операция приведения к типу
operator const char*() {
    return m_data;
}
/* операция вызова функции (получение из исходной строки
новой строки из n символов с указанными индексами.
Например, для строки s, содержащей "a b c d 1 2 3 "
результат s(3,1,3,6) будет содержать "bd3"*/
CMyString operator() (int n, ...)  {
```

```
            CMyString r(n);
            int* p = &n + 1;
            for (int i = 0; i < n; i++ )  {
                r.m_data[i] = m_data[*p++];
            }
            r.m_data[n] = 0;
            return r;
        }
};
// Операция сравнения
int operator != (CMyString& s1, CMyString& s2)
{
    return !( s1 == s2 );
}
// составная операция сложения
CMyString& operator += ( CMyString &s1, const char* s2 )
{
    CMyString tmp(s2);
    s1 = s1 + tmp;
    return s1;
}
```

6.3 Перегрузка операций ввода/вывода

Для ввода-вывода в языке С использовались такие функции стандартной библиотеки, как printf, scanf и др. В С++ эти функции сохранены, однако, появилась новая по сравнению с языком С возможность организации ввода-вывода с использованием операций. Для этих целей в библиотеке <iostream.h> определены следующие объекты:

cin (тип istream) – для ввода данных;

cout (тип ostream) – для вывода данных;

cerr (тип ostream) – для вывода сообщений об ошибках (не буферизованный);

clog (тип ostream) - для вывода сообщений об ошибках;

Причем объект cin связывается со стандартным потоком ввода stdin. Объект cout связывается со стандартным потоком вывода stdout. А cerr и clog связываются со стандартным потоком вывода ошибок stderr.

Сами классы istream и ostream определены в стандартной библиотеке, перегрузка операций в них имеет следующий вид:

```
class istream: public virtual ios   //класс входного потока
     …
     istream &operator >> (int &);
     istream &operator >> (bool &);
     istream &operator >> (double &);
     …
};
class ostream: public virtual ios { //класс выходного потока
```

```
    …
    ostream &operator << (int);
    ostream &operator << (bool);
    ostream &operator << (double);
    …
};
```

С использованием объектов cin и cout ввод-вывод данных выглядит достаточно естественным образом. Например:

```
cout<<”Введите переменную i:”;
cin>>i;
cout<<“\nПеременная i равна ”<<i;
```

Указанные для istream операции используются для ввода данных и называются ***извлечением*** или ***чтением данных из потока***. Соответственно указанные для ostream операции называются ***постановкой*** или ***записью данных в поток***.

Обратим внимание, что определенные в рассматриваемых классах операции ввода-вывода возвращают ссылки на объекты соответствующих классов. Это позволяет применять операцию непосредственно к результату другой такой же операции. Поэтому допустимо выполнять запись операций ввода-вывода в виде:

```
cin>>a>>b>>c;
cout<<a<<b<<c;
```

Особенно интересен тот факт, что операции ввода-вывода могут быть переопределены для пользовательских типов данных. После переопределения Вы сможете записывать объекты своего класса в поток и читать из потока так, как Вы это делаете с базовыми типами данных.

Так как первым аргументом операций ввода-вывода являются всегда объекты типов, определенных в стандартной библиотеке, то для пользовательских типов операции ввода/вывода могут быть переопределены только как глобальные функции, возможно, дружественные пользовательскому классу (T):

```
〖friend〗 istream &operator >> (istream &is, T& obj);
〖friend〗 ostream &operator << (ostream &os, const T& obj);
```

В качестве примера рассмотрим перегрузку операций ввода-вывода для типа CVector.

```
#include<iostream>
class CVector {
    …
    friend ostream& operator<<( ostream& s, const CVector& o)
    {
        s<<"(";
        for ( int i = 0; i < o.m_iSize; i++ )  {
            s<<o.m_arData[i];
            if (i!= o.m_iSize-1) s<<’;’;
        }
        s<<’)’;
        return s;
```

```
    }
    friend istream& operator>>( istream& s, CVector& o)
    {
        char c;
        s>>c; //вводится (
        for ( int i = 0; i < o.m_iSize; i++ )  {
            s>>o.m_arData[i];
            if (i!= o.m_iSize-1) s>>c; // вводится ;
        }
        s>>c; // вводится )
        return s;
    }
};
```

В приведенном примере операция вывода определена таким образом, чтобы допускать строки того же вида, что получаются в результате операции вывода. Приведенную функцию можно снабдить проверками ввода открывающей и закрывающей скобок, проверять, что элементы вектора разделены точкой с запятой, а не каким-либо другим символом.

Примечание. Рассмотренные объекты и классы стандартной библиотеки определены в пространстве имен std. В связи с этим имена объектов и классов должны предваряться префиксом std (std::cin; std::ostream и т.д.). Чтобы не делать этого каждый раз можно воспользоваться следующей директивой

```
using namespace std;
```

6.4 Вопросы и задания

1. Назовите основные правила перегрузки унарных и бинарных операций.

2. Какие особенности имеет перегрузка операций присваивания, обращения по индексу, вызова функции, преобразования тип

3. Каким образом осуществляется перегрузка префиксных и постфиксных операций инкремента (декремента).

4. Как перегрузить операции потокового ввода-вывода для пользовательских типов данных?

5. Для класса вектор действительных чисел выполните перегрузку следующих операций:

a) присваивания,

b) обращения к элементу вектора по индексу,

c) сложения векторов,

d) умножения числа на вектор,

e) составных операций сложения векторов и умножения вектора на число,

f) сравнения векторов на равенство и неравенство.

6. Опишите класс для работы со списком пар «строковый ключ - вещественное значение». Реализуйте необходимые конструкторы и деструктор, а также следующие операции:

a) сложения списка с новой парой значений,

b) сложения двух списков,

c) обращения к элементу списка по индексу,

d) обращения к элементу списка по строковому ключу (через перегрузку операции [] со строковым аргументом).

Выполните перегрузку операции вывода для описанного класса.

7. Для предыдущего упражнения опишите дружественный класс, позволяющий осуществлять однонаправленный проход по списку (итератор). Реализуйте необходимые конструкторы, функцию перехода к первому элементу списка, функцию проверки конца списка, функции доступа к ключу и значению. Реализуйте также следующие операции:

a) присваивание,

b) инкремент (переход к следующему элементу),

c) вычитание (возвращает расстояние между элементами),

d) сравнение на равенство и неравенство,

e) сравнение на больше и меньше.

7 НАСЛЕДОВАНИЕ

Наследование является одним из основных принципов объектно-ориентированного программирования. Наследование позволяет создать новый класс на основе уже существующего класса. При этом вновь созданный класс может расширять функциональность базового класса, некоторым образом модифицировать или уточнить ее. Например, «автомобиль с автоматической коробкой переключения передач» расширяет функциональность базового класса «автомобиль», а класс «трехмерный вектор» уточняет поведение класса «вектор вещественных чисел». Базовых классов может быть несколько, и в C++ реализована именно концепция множественного наследования. Такая возможность встречается в языках программирования достаточно редко, хотя в ряде случаев именно концепция множественного наследования является более естественной. Вообще говоря, множественное наследование полезно всюду, где объект проявляет себя с разных позиций или выступает в различных ролях. Например, седан с автоматической коробкой передач может быть наследован от двух классов: легковой автомобиль седан и автомобиль с автоматической коробкой переключения передач. Конечно, Вы можете определить только класс легковой автомобиль, а тип корпуса и коробки передач сделать полями Вашего класса. Однако представьте себе, что Вы пишете программу с базовыми классами не сейчас, а на заре становления автомобильной промышленности и еще понятия не имеете о возможных типах корпусов и, тем более, об автоматических коробках передач. Другими словами, наследование представляет собой не столько инструмент, предоставляющий возможность более адекватного отображения в Вашей программе моделируемых объектов, сколько средство повторного использования программного кода с возможностью его последовательного развития естественным эволюционным путем.

Отметим, что в литературе *базовый класс* называют также *предком*, *родительским классом* или *суперклассом*, а *производный* от него класс называют *потомком*, *дочерним классом*, или *подклассом*.

7.1 Одиночное наследование

Изучение особенностей реализации концепции наследования в C++ мы начнем с одиночного наследования. Синтаксически определение нового класса на базе уже существующего выглядит следующим образом:

```
class <имя класса>:⟦<атрибут наследования>⟧ <базовый класс>
{…};
```

Здесь атрибут наследования позволяет ограничить область видимости полей и методов класса предка в классе наследнике. Хотя атрибут наследования и является необязательным, в большинстве случаев он указывается, так как по умолчанию наследование производится с атрибутом private. Влияние атрибута наследования на видимость полей и методов базового класса можно понять из приведенной ниже таблицы.

Таблица 7.1 – Атрибуты наследования

Видимость в базовом классе	**Атрибут наследования**		
	private	**protected**	**public**
	Видимость в производном классе		
Private	Недоступен	Недоступен	Недоступен
Protected	Private	Protected	Protected
Public	Private	Protected	Public

Самым распространенным случаем является использование атрибута наследования public, при котором модификаторы protected и public базового класса в наследуемом классе остаются без изменений. Данные и члены базового класса, объявленные как private недоступны для производных классов. Поэтому, если Вы желаете, чтобы потомки вашего класса могли каким-то образом использовать данные или методы базового класса, объявите их, как protected. При наследовании производный класс имеет возможность сделать все доступные методы базового класса защищенными (используя protected) или скрыть все защищенные и открытые методы (используя private).

Пусть, например, имеется класс, реализующий стек свободных указателей на основе динамического массива указанного размера:

```
class CStack {
    private:
      void **m_dat;
      int m_cnt, m_sz;
    public:
      CStack(int sz = 100) {
          m_dat = new void*[sz];
          m_sz = sz;
          m_cnt = 0;
      }
      ~CStack(){
          delete [] m_dat;
      }
      // добавление нового указателя в вершину стека
      bool Push( void *data ) {
          if (m_cnt >= m_sz)  return false;
          m_dat[m_cnt++]=data;
          return true;
```

```
    }
    // извлечение из вершины стека
    bool Pop() {  return m_cnt ? (m_cnt--): 0;  };
    // доступ к вершине стека
    void* Get()  {  return m_cnt ? m_dat[m_cnt-1] : 0;  }
};
```

Такой стек универсален, в том смысле, что в нем можно хранить любые объекты, однако делать это совсем не удобно. Предположим, что нам нужно хранить в стеке строки. В этом случае, например, добавление новой строки (str) в стек (s) может выглядеть так:

```
char* tmp = strdup(str);
s.Add(tmp);
```

извлечение элемента – следующим образом:

```
char* tmp = s.Get();
s.Pop(i);
free(tmp);
```

Писать такой код каждый раз неудобно. Устранить недостатки такого решения можно, создав класс-наследник:

```
class CStrStack : protected CStack
{
     public:
        ~CStrStack()  {
            while (Pop());
        };
        void Push( const char *data ) {
           CStack::Push( strdup(data) );
        };
        bool Pop() {
            if (CStack::Get())  free( CStack::Get() );
            return CStack::Pop();
        };
        const char* Get()  {
            return (const char*)CStack::Get();
        }
};
```

Здесь мы воспользовались спецификатором protected, чтобы исключить в новом классе возможность доступа к открытым членам класса CStack. Реализация функций-членов нового класса выполнена полностью на основе функций-членов класса-предка. Обратим внимание, что для доступа к одноименным функциям класса-предка используется операция расширения видимости «::».

В приведенном примере для класса-наследника мы не определяли конструктор, полагаясь на конструктор по умолчанию, созданный компилятором автоматически. Действительно, созданный компилятором конструктор будет вызывать конструктор класса-предка по умолчанию (будет вызываться конструктор CStack(int sz = 100) со значением параметра sz по умолчанию). В том случае, когда класс-предок не имеет конструктора по умолчанию,

конструктор для производного класса уже не может быть создан автоматически. В этом случае конструктор для производного класса должен быть реализован явно, а вызов конструктора класса-предка должен выполняться из списка инициализации конструктора. Список инициализации представляет собой список следующего вида:

<имя>(⟦<список параметров>⟧) ⟦, <имя>(⟦<список параметров>⟧), …⟧

Здесь <имя> – имя переменной-члена класса или класса-предка, <список параметров> - список параметров инициализатора соответствующего члена класса или конструктора. Таким образом, в списке инициализации можно обратиться не только к конструктору базового класса по умолчанию, но и конструктору с параметрами. С использованием списка инициализации конструкторы приведенных выше классов могли бы выглядеть следующим образом:

```
CStack::CStack(int sz = 100)
  : m_sz(sz), m_cnt(0)
{  m_dat = new void*[sz];  }
CStrStack::CStrStack(int sz = 20)
  :  CStack(sz)
{}
```

При создании объекта сначала вызывается конструктор базового класса, а уже затем - конструктор производного. Деструкторы вызываются в порядке обратном вызовам конструкторов.

Следует отметить, что список инициализации конструктора является единственным местом, где могут быть проинициализированы такие члены класса, как ссылки и константные объекты. Отметим также, что инициализация переменных-членов выполняется в порядке их объявления в классе, независимо от порядка их следования в списке инициализации.

7.2 Виртуальные функции и абстрактные классы

7.2.1 Виртуальные функции

Рассмотрим следующий небольшой фрагмент кода:

```
class CA {
  public:
     const  char* f()  { return "CA::f()"; }
};
class CB: public CA {
  public:
     const char* f()  { return "CB::f()"; }
};
void test1()
{
    CA a;
    CB b;
```

```
    puts( a.f() );      // 1.a
    puts( b.f() );      // 1.b
    CA *pa = &a;
    CB *pb = &b;
    CA *pbase = pb;
    puts( pa->f() );    // 2.a
    puts( pb->f() );    // 2.b
    puts( pbase->f() ); // 2.c
}
```

Попытаемся теперь ответить на вопрос, что же будет выведено на экран. Так как объекты a и b объявлены как объекты классов CA и CB соответственно, то логично предположить, что в случае 1.a и 1.b будут вызваны функции f соответствующих классов и на экране появятся строки:

```
CA::f()
CB::f()
```

Далее, указатели pa и pb объявлены как указатели на объекты классов CA и CB соответственно. Проинициализированы эти указатели адресами объектов соответствующих классов. Поэтому, после выполнения строк 2.a и 2.b, на экране появятся:

```
CA::f()
CB::f()
```

В последнем случае не все так очевидно. Указатель pbase объявлен как указатель на объект базового класса CA, но проинициализирован указателем на объект производного класса CB. Так какая же функция будет выведена в результате выполнения 2.c? Верным ответом будет:

```
CA::f()
```

Другими словами, при вызове обычной функции-члена класса выбор зависит не от фактического класса объекта, на котором установлен указатель, а от типа самого указателя. Поэтому, в 2.c независимо от того, что указатель указывает на объект класса-потомка, все равно будет вызвана функция класса-предка. Такое поведение, конечно, не всегда является приемлемым.

Представим себе, например, иерархию классов, представляющих собой геометрические фигуры на плоскости. В этом случае в базовом классе (CShape) мы могли бы определить такие функции управления фигурами, как параллельный перенос, вращение, отображение. Далее, в производных классах (CLine, CRectangle, CCircle) мы могли бы переопределить эти функции так, чтобы они изменяли состояния соответствующих объектов корректным образом или отображали их на экране. Далее, представляя некоторую группу объектов в виде списка указателей, мы могли бы одним проходом по циклу выполнить параллельный перенос, вращение или отрисовку группы объектов. Чтобы описанная схема работала верно, нам пришлось бы использовать разнородный список, чтобы хранить в нем разнотипные указатели, либо приводить каждый указатель к указателю на объект производного типа (для этого каким-

то образом хранить фактический тип объекта). Однако существует очень элегантное решение, основанное на использовании виртуальных функций.

Виртуальные функции-члены класса объявляются с использованием спецификатора *virtual*, причем если в базовом классе функция объявлена как виртуальная, то в производном классе функция автоматически становится виртуальной, независимо от того, используется ли virtual в производном классе. Виртуальные функции определяются в базовых классах и, как правило, переопределяются в производных. Механизм виртуальных функций является средством реализации концепции динамического полиморфизма (полиморфизм во время выполнения программы) и позволяет однообразно работать с объектами различных типов.

Чтобы разобраться, в чем отличие виртуальных методов от невиртуальных, рассмотрим следующий фрагмент:

```
class CA {
  public:
      virtual const  char* vf()  { return "CA::vf()"; }
};
class CB: public CA {
  public:
      virtual const  char* vf()  { return "CB::vf()"; }
};
void test2()
{
    CA a;
    CB b;
    CA *pa = &a;
    CB *pb = &b;
    CA *pbase = pb;
    puts( pa->vf() );    // 3.a
    puts( pb->vf() );    // 3.b
    puts( pbase->vf() ); // 3.c
}
```

После выполнения функции test2() на экран будет выведено

```
CA::vf()
CB::vf()
CB::vf()
```

Такой результат обусловлен тем фактом, при обращении к виртуальной функции выбор вызываемой функции зависит не от типа указателя, а от фактического класса объекта, на котором этот указатель установлен. Поэтому, в случае 3.c независимо от того, что указатель имеет тип CA*, будет вызвана функция класса-потомка CB.

Такая разница в поведении программы при обращении к виртуальным и невиртуальным функиям-членам класса обусловлена типом связывания. При обращении к невиртуальным функциям-членам класса, также как и при обращении к статическим функциям класса или обычным функциям, используется так называемое *раннее связывание*, когда адрес вызываемой

функции определяется на этапе компиляции программы. Такое определение выполняется по типу объекта или указателя, имени функции, количеству и типам параметров. При обращении к виртуальным функциям используется *позднее (динамическое) связывание*, когда адрес вызываемой функции определяется непосредственно при обращении к такой функции во время выполнения программы. В этом случае при определении адреса учитывается фактический тип объекта, для которого вызывается виртуальная функция.

В качестве иллюстрации приведем простой пример иерархии классов.

```
class CShape {
  public:
      virtual void Shift( int x, int y) {}
};
class CPoint: public CShape {
    int m_x, m_y;
  public:
      CPoint(int x, int y): m_x(x), m_y(y){}
      virtual void Shift( int x, int y)
      {  m_x+=x;  m_y+=y;  }
};
class CLine: public CShape {
    CPoint m_pt1, m_pt2;
  public:
      CLine (int x1, int y1, int x2, int y2)
        : m_pt1(x1,y1), m_pt2(x2, y2){}
      virtual void Shift( int x, int y)
      {  m_pt1.Shift(x,y);  m_pt2.Shift(x,y);  }
};
```

С использованием приведенной иерархии можно определить функцию сдвига набора объектов, представленного массивом указателей:

```
void Shift( int x, int y, CShape **Shapes, int Count)
{
    while (Count)  Shapes[--Count]->Shift(x, y);
}
```

Иногда на практике возникает необходимость обратиться к реализации виртуальной функции в каком-либо конкретном классе. Для этого используют операцию расширения видимости, добавляя перед названием метода название того класса, к функции которого хотим обратиться. Так, после выполнения следующей функции

```
void test3()
    CB b;
    CB *pb = &b;
    CA *pbase = pb;
    puts( pb->CA::vf() );    // 3.b
    puts( pbase->CA::vf() ); // 3.c
}
```

на печать будет выведено

```
CA::vf()
CA::vf()
```

Следует отметить, что статические методы класса не могут быть объявлены виртуальными.

7.2.2 Виртуальные деструкторы

Еще один важный аспект работы с виртуальными функциями проявляется при использовании деструкторов. Изменим приведенный выше тестовый класс:

```
class CA {
  public:
      ~CA()  { puts( "CA::~CA()" ); }
};
class CB: public CA {
  public:
      ~CB()  { puts( "CB::~CB()" ); }
};
void test()
{
    CB *pb = new CB();
    CA *pbase = new CB();
    puts( “delete pb:” );
    delete pb;
    puts( “delete pbase:” );
    delete pbase;
}
```

После выполнения функции test() на экран будет выведено

```
delete pb:
CB::~CB()
CA::~CA()
delete pbase:
CA::~CA()
```

Здесь при удалении объекта через указатель на объект класса-наследника корректно отработали оба деструктора, сначала деструктор производного класса, а уже затем – базового. При удалении объекта через указатель на объект базового класса отработал только деструктор базового класса. Другими словами, деструкторы при использовании указателей на объекты ведут себя точно также как и обычные невиртуальные функции-члены классов.

При этом если деструктор производного класса выполняет сколь-нибудь значимые действия, использование невиртуальных деструкторов приведет к определенным проблемам в программе. Предположим, что в определенной выше системе классов геометрических фигур точки должны иметь некоторые названия. В этом случае класс CPoint может быть изменен следующим образом.

```
class CPoint: public CShape {
    int m_x, m_y;
```

```
    char* m_name;
  public:
      CPoint(int x, int y, const char* name = “”)
        : m_x(x), m_y(y)  {  m_name = strdup(name);  }
      ~CPoint()  {  free m_name;  }
      virtual void Shift( int x, int y)
      {  m_x+=x;  m_y+=y;  }
};
```

Теперь, удаление группы объектов с использованием, например, приведенной ниже процедуры, приведет к весьма нежелательному эффекту.

```
void DeleteObjects(CShape **Shapes, int Count)
{
    for ( int i = 0; i<Count; i++)  {
        delete Shapes[i];
        Shapes[i]=NULL;
    }
}
```

Дело в том, что для объектов класса CPoint, на которые ссылаются элементы массива Shapes, фактически будут вызваны только деструкторы базового класса CShape, и копии строк, хранящиеся в поле m_name, останутся неосвобожденными. Последствия приведенного примера ведут лишь к утечке памяти, но можно привести примеры и с более серьезными последствиями.

Чтобы избежать указанных выше проблем, нужно всего лишь объявить деструктор в базовом классе виртуальным:

```
class CShape {
  public:
      virtual ~CShape() {}
      virtual void Shift( int x, int y) {}
};
```

7.2.3 Чисто виртуальные функции и абстрактные классы

Очень часто в базовом классе определяется лишь интерфейс взаимодействия с объектами, а реализация методов остается пустой. При этом предполагается, что такие методы будут переопределены в классах-потомках. В C++ в таком случае используются так называемые чисто виртуальные функции. Синтаксически чисто виртуальные функции обозначаются как виртуальные функции с указанием «=0» после списка параметров метода.

Например, базовый класс геометрических фигур может быть описан следующим образом:

```
class CShape {
 public:
    virtual void Shift( int x, int y) = 0;
    virtual void Rotate( double a) = 0;
    virtual void Show() = 0;
```

```
};
```

Чисто виртуальные функции обязательно должны быть переопределены в производных классах и часто вообще не имеют реализации. При необходимости, чисто виртуальный метод может иметь реализацию, например:

```
class CShape {
    bool m_visible;
  public:
    CShape()  m_visible(false)  {}
    virtual void Show() = 0
    {  m_visible = true;  }
};
```

Реализация чисто виртуального метода может быть выполнена и вне класса:

```
class CShape {
    bool m_visible;
  public:
    CShape()  m_visible(false)  {}
    virtual void Show() = 0;
};
void CShape::Show()
{  m_visible = true;  }
```

В том случае, если класс содержит хотя бы одну чисто виртуальную функцию, такой класс называется ***абстрактным***. Экземпляр такого класса не может быть создан, независимо от того, выполнена ли реализация чисто виртуальной функции или нет. Это, в частности, означает, что объекты абстрактных классов не могут быть передаваться в функции и возвращаться из функций по значению. В цепочке наследования производные классы становятся абстрактными до тех пор, пока все чисто виртуальные функции не будут переопределены.

Не следует недооценивать практическую значимость чисто виртуальных функций. При разработке программных систем часто применяются классы, содержащие только чисто виртуальные функции. Такие классы называют ***интерфейсными*** или ***классами протокола***. Такие классы используют для определения строгого протокола взаимодействия между компонентами программной системы. Это позволяет устранить взаимные зависимости от конкретных реализаций тех или иных классов, что упрощает как разработку, так и

последующее сопровождение системы, делает возможной как замену отдельных классов, так и целых систем классов.

7.3 Множественное наследование

В C++ возможно наследование более чем от одного класса предка – ***множественное наследование***. Множественное наследование применяется для того, чтобы наделить производный класс свойствами и поведением более чем одного базового класса. Во многих случаях множественное наследование является более естественным, чем одиночное. Несмотря на это, поддержка множественного наследования встречается далеко не во всех языках программирования.

Представим себе, что существует класс протокола, поддерживающий сохранение и восстановление объекта из файла.

```
class CPersistent
{
    public:
      virtual void Load(const char* Filename) = 0;
      virtual void Save(const char* Filename) = 0;
};
```

Пусть требуется создать новый класс вектор действительных чисел (CPersistentVector), способный сохранять и восстанавливать свои данные. Конечно, мы могли бы просто добавить функции сохранения и восстановления в класс CVector, однако это означало бы, что результирующий класс будет несовместим по типу с CPersistent, то есть мы не сможем передать объект класса CPersistentVector туда, где будет требоваться объект CPersistent.

Конечно, мы могли бы выйти из положения, например, таким способом:

```
class CPersistentVector;
class CPV : public CPersistent
{
      CPersistentVector *m_obj;
    public:
      CPV(CPersistentVector *obj) : m_obj(obj) {};
      virtual void Load(const char* Filename);
      virtual void Save(const char* Filename);
};
class CPersistentVector : public CVector
{
    protected:
      CPV m_persistent;
    public:
      CPersistentVector( int size = 2 )
        : CVector(size), m_persistent(this)  {}
      virtual ~CPersistentVector() {}
      operator CPersistent*()  {  return &m_persistent;  }
```

```
    friend class CPV;
};
```

Здесь мы сделали предварительное объявление класса CPersistentVector, затем описали вспомогательный класс CPV, сделав его наследником CPersistent, а уже затем объявили требуемый CPersistentVector, отнаследовав его от CVector. При этом мы реализовали операцию приведения к типу CPersistent*, что позволяет передавать объект класса туда, где требуется указатель на CPersistent. Полученное решение не отличается компактностью и не решает всех проблем. Так для обращения к функциям Load и Save нам потребуется приводить тип к CPersistent*.

Более правильным решением будет воспользоваться механизмом множественного наследования.

7.3.1 Описание класса при множественном наследовании

Синтаксически тот факт, что класс является наследником нескольких классов, указывается при описании класса следующим образом:

```
class <имя класса>: [[<атрибут наследования>]] <базовый класс>
                    [[ , [[<атрибут наследования>]] <базовый класс>
                      ,… ]]
{  …  };
```

Для приведенного выше примера в случае множественного наследования класс CPersistentVector примет вид:

```
class CPersistentVector : public CVector, public CPersistent
{
    public:
      CPersistentVector( int size = 2 )
        : CVector(size), CPersistent()  {}
      virtual ~CPersistentVector() {}
      virtual void Load(const char* Filename);
      virtual void Save(const char* Filename);
};
```

Атрибуты наследования при множественном наследовании имеют тот же смысл, что и при одиночном наследовании. В случае использования для классов-предков конструкторов с параметрами, вызов таких конструкторов следует производить из списка инициализации конструктора (в приведенном примере мы специально явно указали оба конструктора в списке инициализации). При этом порядок вызова конструкторов базовых классов определяется порядком следования базовых классов при наследовании, а не порядком их указания в списке инициализации.

7.3.2 Виртуальные базовые классы и using-объявления

При множественном наследовании довольно часто встречается ситуация, когда один и тот же базовый класс оказывается унаследованным дважды. Например, имея класс CPersistentVector и класс CPrintableVector, поддерживающий вывод вектора на печать, мы могли бы создать производный класс, поддерживающий обе указанные возможности.

```
class CPersistentPrintableVector:  public CPersistentVector,
                                   public CPrintableVector
{…};
```

Базовый класс CVector при этом оказался бы унаследованным дважды – через класс CPersistentVector и через CPrintableVector. Это означало бы, с одной стороны, дублирование всех членов класса CVector в производном классе CPersistentPrintableVector, а с другой стороны – неудобство в работе с объектом производного класса. Дело в том, что обращение к членам базового класса CVector (например, к функции Length()) для объекта v класса CPersistentPrintableVector из-за возникающей неоднозначности пришлось бы выполнять с указанием класса, через который будет такая неоднозначность будет разрешаться:

```
v.CPersistentVector::Length()
```

или

```
v.CPrintableVector::Length()
```

Указанный способ разрешения неоднозначности применяется и в тех случаях, когда при множественном наследовании имеет место совпадение имен членов базовых классов.

В C++ существует, однако, способ кардинально решить возникшую проблему. Этот способ состоит в использовании *виртуальных базовых классов*. Синтаксически указание того факта, что класс наследуется, как виртуальный обозначается ключевым словом virtual в списке базовых классов.

```
class CPersistentVector : virtual public CVector,
                          public CPersistent
{
    public:
      CPersistentVector( int size = 2 )
        : CVector(size), CPersistent()  {}
      …
};
class CPrintableVector: virtual public CVector,
                        public CPrintable
{
    public:
      CPrintableVector( int size = 2 )
        : CVector(size), CPrintable()  {}
      …
};
class CPersistentPrintableVector: public CPersistentVector,
```

```
                                    public CPrintableVector
{
    public:
      CPersistentPrintableVector( int size = 2 )
        : CPersistentVector(size), CPrintableVector(size),
          CVector(size)
      {}
      …
};
```

При таком определении классов иерархии каждый объект класса CPersistentPrintableVector будет содержать лишь по одному экземпляру данных и функций из класса CVector, что избавит от неоднозначности. Другими словами, каждый объект CPersistentPrintableVector будет содержать лишь один вложенный объект CVector, используемый вложенными объектами CPersistentVector и CPrintableVector совместно.

Следует отметить, что даже в тех случаях множественного наследования, когда в качестве базовых классов используются классы, не имеющие общих предков, существует вероятность совпадения имен членов базовых классов. В этом случае решением проблемы неоднозначности является явное указание базового класса, через который осуществляется доступ к члену класса или использование ***using-объявлений***. В последнем случае в производном классе указывается, какие члены базовых классов должны быть введены в область видимости. При этом если одноименные функции базовых классов отличаются списками формальных параметров, можно ввести в область видимости все одноименные функции, получив в производном классе набор перегруженных функций.

7.4 Вопросы и задания

1. Какие типы наследования Вы знаете?

2. Каково назначение атрибутов наследования?

3. Что такое список инициализации конструктора? Что может быть проинициализировано с его использованием?

4. Каков порядок вызова деструкторов при наследовании?

5. Каково предназначение виртуальных функций? Чем они отличаются от обычных функци-членов?

6. Могут ли конструкторы и деструкторы быть виртуальными? В чем преимущества последних?

7. Что такое чисто виртуальная функция? Абстрактный класс? Интерфейсный класс (класс протокола)?

8. Каким образом указывается, что у производного класса несколько классов-предков? Как производится передача параметров конструкторам классов – предков?

9. Как обратиться к членам классов – предков из производного класса и как разрешается проблема конфликтующих по именам членов классов-предков.

10. В чем смысл использования виртуальных базовых классов? Каким образом сделать базовые классы виртуальными?

11. Предложите иерархии классов в следующих областях:

a) транспортные средства,

b) бытовая техника,

c) печатная продукция.

12. На основе интерфейсных классов CPersistent и CPrintable реализуйте классы-наследники, предназначенные для работы с действительными числами:

a) список,

b) матрица MxN,

c) бинарное дерево.

Для организации файлового ввода-вывода используйте стандартные классы файловых потоков ifstream, ofstream, iofstream (заголовочный файл <fstream>).

8 ИСКЛЮЧИТЕЛЬНЫЕ СИТУАЦИИ

Настоящий раздел посвящен обработке ошибок в программе с использованием исключений. Однако прежде, чем перейти непосредственно к изучению механизма работы с исключениями, рассмотрим такой пример. Пусть имеется класс, представляющий список вещественных чисел. В этом случае мы могли бы реализовать функцию чтения элемента по индексу, например, так, как показано ниже.

```
class CDblList
{
    struct list{
        double data;
        list* link;
    } *m_head;
  public:
    double Get(int index) {
          list *p = m_head;
          while (index)  {  p = p->link;  index--;  }
          return p->data;
    }
    …
};
```

Полагаем, даже начинающий программист заметит, что приведенное решение далеко не самое лучшее, так как задание неверного индекса приведет к ошибке в программе из-за обращения по нулевому указателю. Чтобы исключить такую возможность мы могли бы вставить дополнительные проверки:

```
    double Get(int index) {
          list *p = m_head;
          while (p && index)  {  p = p->link;  index--;  }
          if (p)  return p->data;
    }
```

Однако в этом случае при ошибочном индексе значение, возвращаемое функцией, не задается. Таким образом, мы оказываемся в ситуации, когда хорошо было бы вернуть некоторое значение и сообщить вызывающей функции об ошибочном индексе. Конечно, можно поступить и кардинальным образом:

```
    double Get(int index) {
          list *p = m_head;
          while (p && index)  {  p = p->link;  index--;  }
          if (p)  return p->data;
          else abort();
    }
```

Однако пользователя вряд ли устроит, если на каждое неверное действие программа будет аварийно завершаться. Рассмотрим другой вариант - проигнорировать ошибку и вернуть некоторое допустимое значение:

```
double Get(int index) {
      list *p = m_head;
      while (p && index)  {  p = p->link;  index--;  }
      if (p)  return p->data;
      else return 0;
}
```

В этом случае функция будет работать всегда, и если предопределенное значение (в нашем случае – 0) не может содержаться в списке, то это не такой уж и плохой способ сообщить об ошибке. В частности, для указателей таким специальным значением является NULL, для индексов в массиве можно зарезервировать -1 и т.д. Однако если такого значения или диапазона значений зарезервировать не удается, то вызывающая функция никогда не узнает о том, произошла ли ошибка или возвращено требуемое значение.

Один из традиционных способов сообщить об ошибке – возвращать признак успешности выполнения функции (или код ошибки):

```
bool Get(int index, double &result) {
      list *p = m_head;
      while (p && index)  {  p = p->link;  index--;  }
      if (p)  result = p->data;
      return (p != NULL);
}
```

или

```
double Get(int index, bool &result) {
      list *p = m_head;
      while (p && index)  {  p = p->link;  index--;  }
      result = (p != NULL);
      return p ? p->data : 0;
}
```

Этот способ используется достаточно часто, однако здесь мы вынуждены изменить сигнатуру функции. Кроме того, для проверки успешности операции в вызывающей функции нужно вводить дополнительную переменную, а автор вызывающей функции может просто забыть или полениться проверить успешность выполнения функции.

Еще один способ – использовать специальную переменную (глобальную или переменную-член класса) для сообщения об ошибке.

```
class CDblList
{
    struct list{
        double data;
        list* link;
    } *m_head;
```

```
  public:
    bool m_succeeded;
    double Get(int index) {
          list *p = m_head;
          while (p && index)  {  p = p->link;  index--;  }
          m_succeeded = (p != NULL);
          return p ? p->data : 0;
    }
    …
};
```

Здесь для вызывающей функции нет необходимости заводить дополнительные переменные, но забыть проверить успешность операции еще проще, чем в предыдущем варианте. Кроме того, существуют определенные проблемы при использовании такого способа в многопоточной среде.

Альтернативой описанным выше способам является использование механизма обработки исключений.

8.1 Механизм обработки исключений

При работе с исключениями используются три ключевых слова try, catch и throw. При этом первые два обозначают так называемый блок try-catch и связаны с обработкой возникших исключений, а throw – с выбрасыванием исключений. Синтаксически блок try-catch выглядит следующим образом:

```
try  {
    <операторы>
}
catch(<описание исключения>) {
    <операторы обработки исключения>
}
⟦catch(<описание исключения>) {
    <операторы обработки исключения>
} … ⟧
```

Здесь <операторы> в блоке try представляют собой команды, во время выполнения которых может произойти исключительная ситуация. В том случае если некоторый набор операторов заключен в блок try, говорят, что операторы находятся в защищенном блоке. <описание исключения> показывает тип исключения, которое будет обрабатываться в блоке catch. Здесь может быть представлен любой тип данных, как базовым, так и производный. Кроме того здесь может быть описана переменная указанного типа, которую в дальнейшем можно будет использовать при обработке исключения. Помимо описания типа или переменной здесь может стоять троеточие, которое показывает, что данный блок catch обрабатывает исключения любого типа. <Операторы обработки исключения> в блоке catch предназначены для обслуживания возникшей исключительной ситуации определенного типа.

Однако в том случае, если при обработке исключения обнаруживается ошибка, или полностью нейтрализовать возникшую ситуацию не представляется возможным, блок catch сам может быть источником исключений, которые должны отлавливаться и обрабатываться в каком-либо внешнем блоке try-catch.

Для генерации исключения внутри блока try должен быть выполнен оператор throw, параметром которого является объект-исключение желаемого типа.

```
throw [<выражение>]
```

Если при выполнении программы достигнут блок try-catch, то происходит выполнение операторов, находящихся внутри блока try. Если при выполнении этих операторов не возникает исключительной ситуации (не выполняется оператор throw), то выполнение продолжается с оператора, следующего за последним из блоков try, то есть никакие команды из блоков try не выполняются.

Если при выполнении операторов в try производится генерация исключения с использованием ключевого слова throw, то на основе объекта, переданного throw, создается объект - исключение и производится поиск соответствующего блока обработки catch. При поиске обработчики просматриваются в порядке следования блоков catch в программном коде. При этом, если соответствующий по типу обработчик не найден, то выполняется поиск во внешнем блоке try-catch (если таковой имеется), по отношению к рассматриваемому. В том случае, если обработчик найти не удается, то производится аварийное завершение программы.

Если соответствующий обработчик найден и его формальный параметр указан, как параметр, передаваемый по значению (`catch (<тип> <параметр>)`), то параметр инициализируется копией объекта-исключения. В том случае, если параметр указывается по ссылке (`catch (<тип> &<параметр>)`), ссылка инициализируется адресом объекта-исключения.

После инициализации параметра запускается процесс так называемой *раскрутки стека*. В этом процессе выполняется уничтожение (вызываются деструкторы) всех объектов, созданных внутри блока try и находящихся в автоматической памяти. При этом уничтожение объектов производится в порядке обратном их созданию.

После раскрутки стека выполняются <операторы обработки исключения>, располагающиеся в соответствующем блоке catch. В том случае, если при их выполнении никаких исключений не происходит, выполнение продолжается с оператора, следующего за последним из блоков try.

В качестве примера рассмотрим функцию вычисления скорости

```
double velocity( double distance, double time )
{
```

```
    if ( distance < 0 ) throw “ошибка в расстоянии”;
    if ( time <= 0 ) throw 1;
    return distance / time;
}
```

Здесь функция проверяет значения переданных ей аргументов и выбрасывает исключения разных типов (char* и int), если аргументы некорректны. Фрагмент кода, вызывающего такую функцию может выглядеть следующим образом.

```
void test()
{
    try  {
        double d, t, v;
        cout<<”Введите расстояние:”;
        cin>>d;
        cout<<”Введите время:”;
        cin>>t;
        v = velocity(d, t);
        cout<<”скорость: “ << v;
    }
    catch (char* msg)  {
        cout>>msg;
    }
    catch (int)  {
        cout>>”ошибка в параметре ‘время’”;
    }
}
```

В представленном выше примере мы ожидаем исключения двух базовых типов, при этом для типа int мы не стали вводить какой-либо параметр, так как никакой полезной информации в этом случае мы не ожидаем. На практике чаще всего для работы с исключениями используются классы, при этом класс соответствует типу ошибки, а переменные-члены класса определяют ту полезную информацию, которую можно и целесообразно извлечь из создавшейся ситуации. Например, для приведенного выше примера мы могли бы ввести следующий тип исключений.

```
class EInvalidArg {
  protected:
    char m_name[20];
  public:
    EInvalidArg(const char *Name) {
        strncpy(m_name, Name, 20);
        m_name[19] = 0;
    }
    void  Print()    {  printf(“Параметр  <%s>  имеет  недопустимое
значение”, m_name ); };
};
```

Однако, подхода, при котором классы исключений являются независимыми друг от друга, следует избегать. Дело в том, что на программиста, использующего такой код, ложится обязанность отлавливать всюду и обрабатывать по отдельности все введенные типы

исключений. Чтобы облегчить работу с исключениями и сделать ее более элегантной классы исключений должны образовывать некоторую иерархию. Например:

```
class EMyException {
  protected:
    char* m_msg;
  public:
    EMyException( const char *Msg )
    {  m_msg = strdup(Msg);  }
    EMyException( const EMyException &src )
    {  m_msg = strdup(src.m_msg);  }
    ~EMyException()
    {  free(m_msg);  }
    const char* Message() { return m_msg; };
    virtual void Print()  {  puts( m_msg );  }
};
class EInvalidArg:  public EMyException {
  protected:
    char *m_name;
  public:
    EInvalidArg( const char *Name ):
        EMyException("Недопустимый параметр")
    {  m_name = strdup(Name);  }
    EInvalidArg( const EInvalidArg &src ):
        EMyException(src)
    {  m_name = strdup(src.m_name);  }
    ~EInvalidArg ()
    {  free(m_name);  }
    virtual void Print()
    {  printf( "Недопустимый параметр <%s>", m_name );  }
};
class EInvalidIndex:  public EMyException {
  protected:
    int m_min, m_max, m_ind;
  public:
    EInvalidIndex ( int Min, int Max, int Ind ):
        EMyException("Неверный индекс")
    {  m_min=Min; m_max=Max; m_ind=Ind;  }
    EInvalidIndex ( const EInvalidIndex &src ):
        EMyException(src)
    {  m_min=src.m_min; m_max=src.m_max; m_ind=src.m_ind;  }
    virtual void Print()
    {  printf( "Значение индекса (%d) выходит за диапазон"
               " [%d;%d]", m_ind, m_min, m_max );  }
};
```

Напомним, что при работе с исключениями создаются копии объектов-исключений. Поэтому для тех классов исключений, для которых автоматически создаваемый компилятором конструктор копирования некорректен, конструктор копирования должен быть реализован явным образом.

С использованием введенных здесь типов исключений операции с объектами класса вектор могут быть реализованы, например, следующим образом:

```
class CVector{
  private:
    double *m_pData;
    int     m_iSize;
  public:
    …
    CVector operator +( const CVector &v2)  {
        if (m_iSize != v2.m_iSize)  throw EInvalidArg(“v2”);
        CVector res(m_iSize);
        for ( int i = 0; i < m_iSize; i++)
            res.m_pData[i] = m_pData[i]+v2.m_pData[i];
        return res;
    }
    double& operator []( int ind )  {
        if (ind < 0 || ind >= m_iSize)
            throw EInvalidIndex(0,m_iSize-1,ind);
        return m_pData[ind];
    }
};
```

Фрагмент кода работы с реализованными выше операциями приведен ниже.

```
try {
    CVector v1(3,3.3), v2(2,2.2), v3;
    if (exc_no==1)  v1[3] = 1.1;
    else if (exc_no==2)  v3=v1+v2;
    else puts(”Исключений не будет”);
}
catch (const EMyException &e)
{ e.Print(); }
catch ( … )
{ puts( "Неизвестное исключение" ); }
```

Здесь в зависимости от значения целочисленной переменной exc_no будет создана исключительная ситуация, соответствующая одному из введенных выше типов. Обратите внимание, что обработчик исключений здесь принимает параметр не по значению, а по ссылке, что гарантирует появление правильного сообщения при вызове виртуального метода Print (будут выдаваться сообщения «Значение индекса (3) выходит за диапазон [0;2]» и «Недопустимый параметр <v2>»). Если бы параметр обработчика был указан по значению (catch (EMyException e)), то вызывался бы конструктор копирования для класса EMyException, который создавал бы объект базового класса, и мы бы увидели сообщения без важных деталей («Недопустимый параметр» и «Неверный индекс»).

Следует отметить, что, так как при поиске обработчиков они просматриваются в том порядке, в котором записаны, и обработчик считается найденным, если объект-исключение является производным от типа, указанного в обработчике, то типы в обработчике исключений

следует располагать в порядке расширения типа. При этом обработчик catch(…), который будет ловить все исключения, следует располагать последним.

В ряде случаев обработчик не в состоянии до конца обработать возникшую ошибку. В этом случае он может выбросить новое исключение или повторно сгенерировать то же самое исключение путем вызова throw без параметров, передавая, таким образом, исключение во внешний блок try-catch.

8.2 Вопросы и задания

1. Перечислите, какими способами можно сообщить об ошибке в функции.

2. Каким образом можно сгенерировать исключение? Какие типы могут использоваться при этом?

3. Как поймать и обработать исключение? Опишите механизм обработки исключений.

4. В чем отличие передачи объекта-исключения в обработчик по ссылке и по значению?

5. Зачем создавать иерархии классов для описания исключений? Каким должен быть порядок записи обработчиков исключений?

6. Для класса, разработанного в упражнении 6 раздела 6.4 введите следующие типы исключений:

a) пара с таким ключом уже есть в списке (генерируется при выполнении операций сложения списка с парой и сложения двух списков),

b) индекс выходит за допустимый диапазон (для операции обращения по целочисленному индексу),

c) ключ не найден (для перегруженной операции обращения по индексу со строковым аргументом).

Указанные классы исключений наследуйте от базового класса EMyException. Организуйте обработку исключений с выводом соответствующих сообщений.

7. Для класса, разработанного в упражнении 7 раздела 6.4 введите следующие типы исключений:

a) попытка перехода к несуществующему элементу списка (генерируется при выполнении операции инкремент),

b) операция не применима для разных списков (для операций вычитания и сравнения),

c) попытка доступа к несуществующему элементу (для функций доступа к ключу и значению).

Указанные классы исключений наследуйте от базового класса EMyException. Организуйте обработку исключений с выводом соответствующих сообщений.

9 ШАБЛОНЫ

Довольно часто причиной дублирования кода является необходимость написания одного и того же кода для различных типов данных. Одним из способов решения этой проблемы является использование параметризованных макроопределений. Например, для вычисления минимума двух значений можно воспользоваться следующим макроопределением

```
# define min (a,b) ((a<b)? a: b)
```

Такой способ избежать дублирования кода, однако, весьма далек от совершенства. Во-первых, использование макроопределений приводит к разрастанию кода программы, так как код, реализующий макроопределение просто подставляется в место обращения. Во-вторых, отсутствие какого-либо контроля в макроопределениях часто приводит к ошибкам в программах. В-третьих, отладка макроопределений связана со значительными трудностями.

Механизм *шаблонов* представляет собой более изящное решение проблемы дублирования кода и позволяет отделить алгоритм от его реализации, выполняемой для конкретных типов данных. В C++ используются два вида шаблонов: шаблоны функций и шаблоны классов.

9.1 Шаблоны функций

Объявление шаблона функции начинается с ключевого слова template, за которым следует список параметров шаблона, заключенный в угловые скобки. За списком параметров шаблона идет собственно определение функции.

```
template <class <имя>[, class <имя>, …] >
<определение функции>
```

Здесь <имя> - название параметра шаблона, вместо ключевого слова `class` может использоваться `typename`. В блоке <определение функции> введенные в заголовке шаблона названия параметров шаблона могут использоваться как уже определенные типы.

Шаблон приведенной выше функции, возвращающей минимальное из двух значений можно определить следующим образом:

```
template <class T>
T min (T a, T b)
{  return (a<b) ? a: b;};
```

Использование определенной выше шаблонной функции выглядит следующим образом:

```
int i=10, j=20, k;
k=min(i,j);
double u=2.2, v=3.3, w;
w=min(u, v);
```

При компиляции приведенного фрагмента программы будет автоматически порождено две перегруженные версии шаблонной функции:

`int min (int, int)` и `double min (double, double).`

Следует отметить, что параметр шаблона в приведенном выше примере определялся автоматически по типам фактических параметров (int и double), переданных шаблонной функции min. Естественно, что на практике могут возникнуть случаи, когда определение параметра автоматически будет невозможным, как, например, в следующем примере:

```
w = min(u, j);
```

Здесь в качестве фактических параметров функции переданы значения различных типов, что приводит к неоднозначности при определении параметра шаблона. Решить проблему такой неоднозначности можно разными способами. Очевидно, можно привести параметры к одному типу явным образом, например: w = min(u, (double)j);

Менее очевидный способ – изменить сам шаблон функции таким образом, чтобы было возможным использовать в шаблоне разные типы. Конечно, реализация такого способа будет зависеть от конкретного шаблона. Для нашего случая шаблон может быть изменен следующим образом:

```
template <class T1,class T2>
T1 min (T1 a, T2 b)  {  return (a<b) ? a: b;  };
```

Наконец, Вы можете не полагаться на автоматическое определение параметров шаблона и указать их явным образом. В случае явного указания параметров шаблона они помещаются в угловых скобках сразу после имени шаблонной функции. Например:

```
min<double>(u, j);
```

К сожалению, для некоторых типов данных реализация той или иной шаблонной функции может оказаться неэффективной или вообще некорректной. В таких случаях требуется изменить поведение шаблона для некоторых типов данных. Сделать это можно, определив перегруженную функцию, или, используя *специализацию шаблона*.

В качестве примера рассмотрим определение перегруженной функции для получения минимальной из двух строк:

```
char *min(char *a, char *b)
{  return (strcmp(a,b)<0) ? a : b;  }
```

Специализация шаблона для строк будет выглядеть иначе:

```
template <> char* min<char*>( char* a, char* b)
{  return (strcmp(a,b)<0) ? a : b;  }
```

Следует отметить, что в некоторых средах при специализации допускается опускать параметр шаблона:

```
template <> char* min( char* a, char* b)
{  return (strcmp(a,b)<0) ? a : b;  }
```

9.2 Шаблоны классов

Аналогично тому, как шаблон функций представляет собой параметризованную функцию, шаблон класса представляет собой параметризованный класс. Шаблон класса позволяет создать семейство родственных классов. Создаваемые классы имеют одинаковое поведение и состав свойств, однако отличаются используемыми типами данных, задаваемыми в качестве параметров шаблона.

9.2.1 Определение шаблона класса

Синтаксически шаблон класса определяется следующим образом:

```
template < <список параметров шаблона> >
<определение класса>
```

<список параметров шаблона> представляет собой последовательность параметров, разделенных запятыми

```
<описание параметра> [,<описание параметра> , …]
```

В качестве параметров шаблона (<описание параметра>) могут использоваться:

- типы данных

```
class <имя> [=<тип по умолчанию>]
typename <имя> [=<тип по умолчанию>]
```

- константы

```
<тип_константы> <идентификатор>[=<значение по умолчанию>]
```

- другие шаблоны

```
template < <список параметров шаблона> > class <имя шаблона класса> [=<тип по умолчанию>]
```

Любые параметры шаблона могут быть заданы значениями по умолчанию.

Шаблоны целесообразно использовать в тех случаях, когда типы используемых в классе данных не оказывают существенного влияния на реализацию класса. В частности, шаблоны целесообразно применять при создании контейнерных классов.

Рассмотрим, например, как может быть реализован шаблон класса стек.

```
template < class T, int Size = 256 >
class TFixedStack{
    private:
      T m_dat[Size];
      int m_cnt;
    public:
      TFixedStack()  : m_cnt( 0 ) {}
      // добавление в вершину стека
      bool Push( const T &data ) {
          if (m_cnt >= Size)  return false;
          m_dat[m_cnt++] = data;
          return true;
      }
```

```
      // извлечение из вершины стека
      bool Pop(T &data) {
          if (! m_cnt)  return false;
          data = m_dat[m_cnt--];
          return true;
      };
      // доступ к вершине стека
      bool Get(T &data) {
          if (! m_cnt)  return false;
          data = m_dat[m_cnt];
          return true;
      };
};
```

В отличие от шаблонов функций при создании объекта шаблона класса параметры шаблона всегда требуется указывать явным образом, например:

```
TFixedStack <double, 10> s;
```

После того, как объект создан, работа с ним ничем не отличается от работы с объектами обычных классов:

```
s.Push(105.01);
double val;
s.Get(val);
s.Pop(val);
```

В случае если часть параметров шаблона имеет значения по умолчанию, при создании объекта они могут не указываться, например:

```
TFixedStack <double> s;
```

Шаблоны классов могут участвовать в наследовании, как в качестве базовых, так и в качестве производных классов:

```
template < class T >
class TAbstractStack {
    public:
      // виртуальный деструктор
      virtual ~TAbstractStack(){}
      // добавление в вершину стека
      virtual bool Push( const T &data ) = 0;
      // извлечение из вершины стека
      virtual bool Pop(T &data) = 0;
      // доступ к вершине стека
      virtual bool Get(T &data) = 0;
};
template < class T, int Size = 256 >
class TFixedStack : public TAbstractStack<T> {
    private:
      T m_dat[Size];
      int m_cnt;
    public:
      TFixedStack()  : m_cnt( 0 ) {}
      bool Push( const T &data );
      bool Pop(T &data);
```

```
      bool Get(T &data);
};
```

Здесь сначала объявлен класс протокола стека, затем от него наследуется класс стека на основе массива фиксированного размера (реализация ничем не отличается от приведенной выше, поэтому не приводится). Ниже описан еще один производный класс стек, работа которого основана на односвязной списковой структуре данных.

```
template <class T>
class TDynamicStack: public TAbstractStack<T>
{
    struct list{
        T data;
        list* link;
    } *m_head;
  public:
    TDynamicStack() {  m_head = NULL;  }
    ~TDynamicStack() {
         list *p;
         while (p = m_head) { m_head = p->link; delete p; }
    }
    bool Push( const T &data ) {
         list* p = new list;
         p->data = data;
         p->link = m_head;
         m_head = p;
         return true;
    };
    bool Pop( T &data );
    bool Get(T &data) {
          if (! m_head)  return false;
          data = m_head->data;
          return true;
      };

};
```

На практике часто возникает необходимость выноса реализации какого-либо метода за границу класса. Вынесение реализации за границы описания шаблона класса выглядит следующим образом:

```
template <class T>
bool TDynamicStack<T>::Pop( T &data)
{
    if (! m_head)  return false;
    list* p = m_head;
    m_head = p->link;
    data = p->data;
    delete p;
    return true;
}
```

Следует отметить, что в случае указания всех параметров шаблона при наследовании от него может быть унаследован обычный класс.

9.2.2 Специализация

Как и в случае с шаблонами функций, если реализация шаблона для каких-то типов данных оказывается неэффективной или некорректной, выполняется специализация (специальная реализация). Она может быть выполнена для шаблона класса целиком или для отдельного метода класса.

При специализации всего класса, после описания базового варианта класса размещается полное описание специализированного класса. Например.

```
template <>
class TDynamicStack<char*>: public TAbstractStack<char*>
{
    struct list{
        char* data;
        list* link;
    } *m_head;
  public:
    TDynamicStack() {  m_head = NULL;  }
    ~TDynamicStack() {
         list *p;
         while (p = m_head) { m_head = p->link; delete p; }
    }
    bool Push( char* const &data ) {
         list* p = new list;
         p->data = strdup(data);
         p->link = m_head;
         m_head = p;
         return true;
    };
    bool Pop( char* &data ) {
        if (! m_head)  return false;
        list* p = m_head;
        m_head = p->link;
        data = p->data;
        delete p;
        return true;
    }
    bool Get(char* &data) {
          if (! m_head)  return false;
          data = strdup(m_head->data);
          return true;
      };
};
```

При специализации отдельного метода размещают специальную реализацию метода. Например.

```
template <>
bool TDynamicStack<char*>::Push( char* const &data ) {
        list* p = new list;
        p->data = strdup(data);
        p->link = m_head;
        m_head = p;
        return true;
    };
```

9.3 Вопросы и задания

1. Объясните назначение шаблонов в C++.

2. В чем заключаются отличия шаблонов функций и параметризованных макроопределений

3. При каких обстоятельствах автоматическое определение параметров шаблона функции может быть неудачным? Какие способы разрешения возникшей проблемы Вы знаете?

4. Каким образом и зачем выполняется специализация шаблона функции?

5. Каким образом определяется шаблон класса? Каким образом создать экземпляр шаблона класса?

6. Как описать функцию-член шаблона класса вне объявления шаблона класса.

7. Каким образом выполняется специализация шаблона класса и функции-члена шаблона класса?

8. Определите шаблон класса «вектор» с необходимыми конструкторами и деструктором, а также с операциями присваивания, обращения к элементу вектора по индексу, сложения векторов, сравнения векторов на равенство и неравенство.

9. Выполните специализацию шаблона из задания 8 для вектора строк.

10. Определите шаблон класса для работы со списком пар «строковый ключ – значение произвольного типа». Реализуйте необходимые конструкторы и деструктор, а также операции сложения списка с новой парой значений, сложения двух списков, обращения к элементу списка по индексу, обращения к элементу списка по строковому ключу.

11. Выполните специализацию необходимых методов шаблона класса из задания 10 для пар вида «строковый ключ – строковое значение».

ЗАКЛЮЧЕНИЕ

В заключение отметим, что настоящее пособие, содержит, конечно же, достаточно сжатое описание основ языка, и ограниченный объем как самого пособия, так и курса, не позволяет рассмотреть все особенности. Автор, тем не менее, надеется, что освещенные в пособии основы были успешно освоены, а само пособие заинтересовало читателя в дальнейшем изучении этого замечательного языка. Если это действительно так, то ниже предлагается список тем, которые можно предложить читателю для самостоятельной проработки:

- директивы компилятора,
- организация пространств имен,
- указатели на члены классов,
- информация о типе во время выполнения и операции приведения типа,
- управления ресурсами при использовании исключений,
- спецификация исключений,
- стандартные библиотеки языка C/C++.

Этот список, конечно же, не является исчерпывающим. Информацию по этим и другим темам можно найти во многих книгах, в названии которых фигурирует название языка “C++”. Эту информацию можно почерпнуть, конечно же, и из фундаментальной работы Бьерна Страуструпа «Язык программирования C++» [1].

ЛИТЕРАТУРА

1. Страуструп Б. Язык программирования C++. Специальное издание. М.: Радио и связь, 1991. - 349с.

2. Вирт Н. Алгоритмы и структуры данных: Пер. с англ. - М.: Мир, 1989.

3. Савитч У. Язык C++. Курс объектно-ориентированного программирования. – М.: Вильямс, 2001. – 696с.

4. Вайнер Р., Пинсон Л. C++ изнутри.- Киев:НПИФ «ДиаСофт», 1993. -301с.

5. Язык программирования Си / Б. Керниган, Д. Ритчи. - 3-е изд., испр. - СПб. : Невский Диалект, 2004. - 351 с.

6. Программирование на C++ / С. Дьюхарст, К. Старк. - Киев : НИПФ "ДиаСофт", 1993. - 271 с.

7. Практикум по программированию на языке СИ / В. В. Подбельский. - М. : Финансы и статистика, 2004