



*Томский межвузовский центр
дистанционного образования*

С.И. Борисов

ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ

Учебное пособие

ТОМСК – 2002

Министерство образования Российской Федерации

**ТОМСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ СИСТЕМ
УПРАВЛЕНИЯ И РАДИОЭЛЕКТРОНИКИ (ТУСУР)**

**Кафедра компьютерных систем в управлении
и проектировании (КСУП)**

С.И. Борисов

ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ

Учебное пособие

2002

Борисов С.И.

Объектно-ориентированное программирование: Учебное пособие. – Томск: Томский межвузовский центр дистанционного образования, 2002. – 217 с.

© Борисов С.И., 2002

© Томский межвузовский центр
дистанционного образования, 2002

СОДЕРЖАНИЕ

Предисловие.....	9
Исторические замечания	10
Эффективность и структура.....	11
Правила правой руки	13
1. ГЛАВА ПЕРВАЯ	15
1.1. Начало.....	15
1.1.1. Вывод на экран.....	15
1.1.2. Компиляция.....	16
1.1.3. Ввод.....	16
1.2. Комментарии.....	17
1.3. Типы и описания.....	18
1.3.1. Основные типы	19
1.3.2. Производные типы	20
1.4. Выражения и операторы	21
1.4.1. Выражения.....	21
1.4.2. Операторы выражения	22
1.4.3. Пустой оператор	22
1.4.4. Блоки	22
1.4.5. Операторы if.....	23
1.4.6. Операторы switch.....	23
1.4.7. Оператор while	24
1.4.8. Оператор for	25
1.4.9. Описания.....	25
1.5. Функции	26
1.6. Структура программы	27
1.7. Классы	29
1.8. Перегрузка операций	31
1.9. Конструкторы	32
1.10. Векторы.....	33
1.11. Inline-подстановка.....	35
1.12. Производные классы	36
1.13. Еще об операциях	37
1.14. Друзья (friends).....	38
1.15. Обобщенные векторы.....	39
1.16. Полиморфные векторы.....	40
1.17. Виртуальные функции.....	41
2. ГЛАВА ВТОРАЯ	43
2.1. Описания	43
2.1.1. Область видимости.....	45

2.1.2. Объекты и адреса (Lvalue)	47
2.1.3. Время жизни	47
2.2. Имена	48
2.3. Типы	48
2.3.1. Основные типы	49
2.3.2. Неявное преобразование типа	51
2.3.3. Производные типы	51
2.3.4. Тип void	53
2.3.5. Указатели	53
2.3.6. Векторы	55
2.3.7. Указатели и векторы	57
2.3.8. Структуры	58
2.3.9. Эквивалентность типов	61
2.3.10. Ссылки	61
2.4. Константы	65
2.4.1. Целые константы	65
2.4.2. Константы с плавающей точкой	66
2.4.3. Символьные константы	67
2.4.4. Строки	68
2.4.5. Ноль	69
2.4.6. Const	69
2.4.7. Перечисления	71
2.5. Экономия пространства	72
2.5.1. Поля	73
2.5.2. Объединения	74
3. ГЛАВА ТРЕТЬЯ	77
3.1. Краткая сводка операций	77
3.1.1. Круглые скобки	79
3.1.2. Порядок вычисления	79
3.1.3. Увеличение и уменьшение	80
3.1.4. Побитовые логические операции	81
3.1.5. Преобразование типа	82
3.1.6. Свободная память	83
3.2. Сводка операторов	86
3.2.1. Проверки	87
3.2.2. Goto	90
3.3. Комментарии и выравнивание	90
4. ГЛАВА ЧЕТВЕРТАЯ	93
4.1. Введение	93
4.2. Компоновка	94
4.3. Заголовочные файлы	96

4.3.1. Один заголовочный файл.....	98
4.3.2. Множественные заголовочные файлы	98
4.3.3. Скрытие данных.....	99
4.4. Файлы как модули.....	100
4.5. Как создать библиотеку	101
4.6. Функции	102
4.6.1. Описания функций	103
4.6.2. Определения функций.....	103
4.6.3. Передача параметров.....	104
4.6.4. Возврат значения	105
4.6.5. Векторные параметры	106
4.6.6. Параметры по умолчанию	107
4.6.7. Перегрузка имен функций	108
4.6.8. Незаданное число параметров.....	110
4.6.9. Указатель на функцию	113
4.7. Макросы	116
5. ГЛАВА ПЯТАЯ. КЛАССЫ.....	119
5.1. Знакомство и краткий обзор.....	119
5.2. Классы и члены.....	120
5.2.1. Функции члены	120
5.2.2. Классы.....	122
5.2.3. Ссылки на себя.....	123
5.2.4. Инициализация	125
5.2.5. Очистка	127
5.2.6. Inline	128
5.3. Интерфейсы и реализации.....	129
5.3.1. Законченный класс	130
5.4. Друзья и объединения.....	132
5.4.1. Друзья	132
5.4.2. Уточнение имени члена	135
5.4.3. Вложенные классы	135
5.4.4. Статические члены	137
5.4.5. Структуры и объединения	137
5.5. Конструкторы и деструкторы	141
5.5.1. Предостережение	141
5.5.2. Статическая память	142
5.5.3. Свободная память	144
5.5.4. Объекты класса и члены	144
5.5.5. Векторы объектов класса.....	147
5.5.6. Небольшие объекты.....	148
5.5.7. Предостережение	150

5.5.8. Объекты переменного размера.....	151
6. ГЛАВА ШЕСТАЯ. ПЕРЕГРУЗКА ОПЕРАЦИЙ.....	153
6.1. Введение.....	153
6.2. Функции операции.....	154
6.2.1. Бинарные и унарные операции.....	155
6.2.2. Предопределенные значения операций.....	156
6.2.3. Операции и определяемые Пользователем типы.....	156
6.3. Определяемое преобразование типа.....	157
6.3.1. Конструкторы.....	158
6.3.2. Операции преобразования.....	159
6.3.3. Неоднозначности.....	161
6.4. Константы.....	163
6.5. Большие объекты.....	163
6.6. Присваивание и инициализация.....	164
6.7. Индексирование.....	167
6.8. Вызов функции.....	169
6.9. Класс «строка».....	171
6.10. Друзья и члены.....	174
6.11. Предостережение.....	176
7. ГЛАВА СЕДЬМАЯ.....	177
7.1. Введение.....	177
7.2. Производные классы.....	178
7.2.1. Построение производного класса.....	178
7.2.2. Функции члены.....	180
7.2.3. Видимость.....	182
7.2.4. Указатели.....	184
7.2.5. Иерархия типов.....	185
7.2.6. Конструкторы и деструкторы.....	186
7.2.7. Поля типа.....	187
7.2.8. Виртуальные функции.....	189
7.3. Альтернативные интерфейсы.....	191
7.3.1. Интерфейс.....	192
7.3.2. Реализация.....	193
7.3.3. Как этим пользоваться.....	195
7.3.4. Ограниченные интерфейсы.....	197
7.4. Добавление к классу.....	198
7.5. Неоднородные списки.....	200
8. ГЛАВА ВОСЬМАЯ. ПОТОКИ.....	201
8.1. Введение.....	201
8.2. Вывод.....	202

8.2.1. Вывод встроенных типов	202
8.2.2. Некоторые подробности разработки	203
8.2.3. Форматированный вывод	204
8.2.4. Виртуальная функция вывода	204
8.3. Файлы и потоки	205
8.3.1. Инициализация потоков вывода	205
8.3.2. Заккрытие потоков вывода	206
8.3.3. Открытие файлов	207
8.3.4. Копирование потоков	208
8.4. Ввод	208
8.4.1. Ввод встроенных типов	209
8.4.2. Состояния потока	211
8.4.3. Ввод типов, определяемых пользователем	212
8.4.4. Инициализация потоков ввода	213
8.5. Работа со строками	215
8.6. Буферизация	216
8.7. Эффективность	218

ПРЕДИСЛОВИЕ

C++ - универсальный язык программирования, являющийся в настоящее время де-факто стандартом в отрасли промышленного программирования. Этот язык задуман так, чтобы сделать программирование более приятным для серьезного программиста. За исключением второстепенных деталей, C++ является надмножеством языка программирования C. Это означает, что программа, написанная на языке C, будет корректно скомпилирована и абсолютно идентично воспринята компилятором языка C++. Помимо возможностей, которые дает C, C++ предоставляет гибкие и эффективные средства определения новых типов. Используя определения новых типов, точно отвечающих концепциям приложения, программист может делить разрабатываемую программу на легко контролируемые части. Такой метод построения программ часто называют абстракцией данных. Информация о типах содержится в некоторых объектах типов, определенных пользователем. Такие объекты просты и надежны в использовании в тех ситуациях, когда их тип нельзя установить на стадии компиляции. Программирование с применением таких объектов часто называют объектно-ориентированным. При правильном использовании этот метод дает более короткие, проще понимаемые и легче контролируемые программы.

Ключевым понятием C++ является класс. Класс - это тип, определяемый пользователем. Классы обеспечивают сокрытие данных, гарантированную инициализацию данных, неявное преобразование типов для типов, определенных пользователем, динамическое задание типа, контролируемое пользователем управление памятью и механизмы перегрузки операций. C++ предоставляет гораздо лучшие, чем в C, средства выражения модульности программы и проверки типов. В языке есть также усовершенствования, не связанные непосредственно с классами, включающие в себя символические константы, inline-подстановку функций, параметры функции по умолчанию, перегруженные имена функций, операции управления свободной памятью и ссылочный тип. В C++ сохранены возможности языка C по работе с основными объектами аппаратного обеспечения (биты, байты, слова, адреса и т.п.). Это позволяет весьма эффективно реализовывать типы, определяемые пользователем.

C++ и его стандартные библиотеки спроектированы так, чтобы обеспечивать переносимость. Имеющаяся на текущий момент реализация языка будет идти в большинстве систем, поддерживающих C. Из C++ программ можно использовать C-библиотеки, и с C++ можно ис-

пользовать большую часть инструментальных средств, поддерживающих программирование на С.

Исторические замечания

Язык С++ был разработан сотрудником лаборатории Белла (Bell Labs) при компании American Telegraph and Telephone Company (AT&T) Бьярном Страуструпом. Этот язык восходит главным образом к С, которой тоже, только несколько ранее, был разработан в этой лаборатории. Язык С сохранен как подмножество, поэтому сделанного в С акцента на средствах низкого уровня достаточно, чтобы справляться с самыми насущными задачами системного программирования. С, в свою очередь, многим обязано своему предшественнику BCPL; на самом деле, комментарии // (заново) введены в С++ из BCPL. Еще одним источником вдохновения послужил язык Simula67, из него была позаимствована концепция класса (вместе с производными классами и функциями членами). Это было сделано, чтобы способствовать модульности через использование виртуальных функций. Возможности С++ по перегрузке операций и свобода в расположении описаний везде, где может встречаться оператор, похожи на Алгол68 [14].

Название С++ придумано разработчиком (Бьярном Страуструпом) не сразу (в 1983 г). Более ранние версии языка использовались начиная с 1980 г., и были известны как «С с Классами». Первоначально язык был придуман потому, что автор хотел написать модели, управляемые прерываниями, для чего был бы идеален Simula67, если не принимать во внимание эффективность. «С с Классами» использовался для крупных проектов моделирования, в которых строго тестировались возможности написания программ, требующих минимального (только) пространства памяти и времени на выполнение. В «С с Классами» не хватало перегрузки операций, ссылок, виртуальных функций и многих деталей. С++ был впервые введен за пределами исследовательской группы автора в июле 1983 г., однако тогда многие особенности С++ были еще не придуманы.

Изначально С++ был разработан, чтобы автору и его друзьям не приходилось программировать на ассемблере, С или других современных языках высокого уровня. Основным его предназначением было сделать написание хороших программ более простым и приятным для отдельного программиста. Плана разработки С++ на бумаге никогда не было; проект, документация и реализация двигались одновременно. Разумеется, внешний интерфейс С++ был написан на С++. Никогда не существовало «Проекта С++» и «Комитета по разработке С++». Поэтому С++ развивался и продолжает развиваться во всех направлениях, чтобы справляться со

сложностями, с которыми сталкиваются пользователи, а также в процессе дискуссий автора с его друзьями и коллегами.

В качестве базового языка для C++ был выбран C, потому что он 1) многоцелевой, лаконичный и относительно низкого уровня; 2) отвечает большинству задач системного программирования; 3) идет везде и на всем; и 4) пригоден в среде программирования UNIX. В C есть свои сложности, но в наспех спроектированном языке тоже были бы свои, а сложности C нам известны. Самое главное, работа с C позволила «C с Классами» быть полезным (правда, неудобным) инструментом в ходе первых месяцев раздумий о добавлении к C Simula-образных классов.

C++ стал использоваться шире, и по мере того, как возможности, предоставляемые им помимо возможностей C, становились все более существенными, вновь и вновь поднимался вопрос о том, сохранять ли совместимость с C. Ясно, что отказавшись от определенной части наследия C, можно было бы избежать ряда проблем. Это не было сделано, потому что 1) есть миллионы строк на C, которые могли бы принести пользу в C++ при условии, что их не нужно было бы полностью переписывать с C на C++; 2) есть сотни тысяч строк библиотечных функций и сервисных программ, написанных на C, которые можно было бы использовать из или на C++ при условии, что C++ полностью совместим с C по загрузке и синтаксически очень похож на C; 3) есть десятки тысяч программистов, которые знают C и которым поэтому нужно только научиться использовать новые особенности C++, а не заново изучать его основы; и 4), поскольку C++ и C будут использоваться на одних и тех же системах одними и теми же людьми, отличия должны быть либо очень большими, либо очень маленькими, чтобы свести к минимуму ошибки и недоразумения. Позднее была проведена проверка определения C++, чтобы удостовериться в том, что любая конструкция, допустимая и в C и в C++, действительно означает в обоих языках одно и то же.

Эффективность и структура

C++ был развит из языка программирования C и за очень немногими исключениями сохраняет C как подмножество. Базовый язык, C надмножество C++, спроектирован так, что имеется очень близкое соответствие между его типами, операциями и операторами и компьютерными объектами, с которыми непосредственно приходится иметь дело: числами, символами и адресами. За исключением операций динамической работы с памятью new и delete, отдельные выражения и

операторы C++ обычно не нуждаются в скрытой поддержке во время выполнения или подпрограммах.

В C++ используются те же последовательности вызова и возврата из функций, что и в C. В тех случаях, когда даже этот довольно эффективный механизм является слишком дорогим, C++-функция может быть подставлена `inline`, удовлетворяя, таким образом, соглашению о записи функций без дополнительных расходов времени выполнения.

Одним из первоначальных предназначений C было применение его вместо программирования на ассемблере в самых насущных задачах системного программирования. Когда проектировался C++, были приняты меры, чтобы не ставить под угрозу успехи в этой области. Различие между C и C++ состоит, в первую очередь, в степени внимания, уделяемого типам и структурам. C выразителен и снисходителен. C++ еще более выразителен, но чтобы достичь этой выразительности, программист должен уделить больше внимания типам объектов. Когда известны типы объектов, компилятор может правильно обрабатывать выражения, тогда как в противном случае программисту пришлось бы задавать действия с мучительными подробностями. Знание типов объектов также позволяет компилятору обнаруживать ошибки, которые в противном случае остались бы до тестирования. Заметьте, что использование системы типов для того, чтобы получить проверку параметров функций, защитить данные от случайного искажения, задать новые операции и т.д., само по себе не увеличивает расходов по времени выполнения и памяти.

Особое внимание, уделенное при разработке C++-структуры, отразилось на возрастании масштаба программ, написанных со времени разработки C. Маленькую программу (меньше 1000 строк) вы можете заставить работать с помощью грубой силы, даже нарушая все правила хорошего стиля. Для программ больших размеров это не совсем так. Если программа в 10 000 строк имеет плохую структуру, то вы обнаружите, что новые ошибки появляются так же быстро, как удаляются старые. C++ был разработан так, чтобы дать возможность разумным образом структурировать большие программы так, чтобы для одного человека не было непомерным справляться с программами в 25 000 строк. Существуют программы гораздо больших размеров, однако те, которые работают, в целом, как оказывается, состоят из большого числа почти независимых частей, каждая из которых намного ниже указанных пределов. Естественно, сложность написания и поддержки программы зависит от сложности разработки, а не просто от числа строк текста программы, так что точные цифры, с помощью которых

были выражены предыдущие соображения, не следует воспринимать слишком серьезно.

Не каждая часть программы, однако, может быть хорошо структурирована, независима от аппаратного обеспечения, легко читаема и т.п. С++ обладает возможностями, предназначенными для того, чтобы непосредственно и эффективно работать с аппаратными средствами, не беспокоясь о безопасности или простоте понимания. Он также имеет возможности, позволяющие скрывать такие программы за элегантными и надежными интерфейсами.

В этой книге особый акцент делается на методах создания универсальных средств, полезных типов, библиотек и т.д. Эти средства пригодятся как тем программистам, которые пишут небольшие программы, так и тем, которые пишут большие. Кроме того, поскольку все нетривиальные программы состоят из большого числа полунезависимых частей, методы написания таких частей пригодятся и системным, и прикладным программистам.

У кого-то может появиться подозрение, что спецификация программы с помощью более подробной системы типов приведет к увеличению исходных текстов программы. В С++ это не так; С++-программа, описывающая типы параметров функций, использующая классы и т.д., обычно немного короче эквивалентной С-программы, в которой эти средства не используются.

Правила правой руки

Здесь приводится набор правил, которых вам хорошо бы придерживаться, изучая С++. Когда вы станете более опытными, вы можете превратить их в то, что будет подходить для вашего рода деятельности и вашего стиля программирования. Они умышленно сделаны очень простыми, поэтому подробности в них опущены. Не воспринимайте их чересчур буквально. Написание хороших программ требует ума, вкуса и терпения.

[1] Когда вы программируете, вы создаете конкретное представление идей вашего решения некоторой задачи. Пусть структура отражает эти идеи настолько явно, насколько это возможно:

[a] если вы считаете «это» отдельным понятием, сделайте его классом;

[b] если вы считаете «это» отдельным объектом, сделайте его объектом некоторого класса;

[c] если два класса имеют общим нечто существенное, сделайте его базовым классом. Почти все классы в вашей программе будут

иметь нечто общее; заведите (почти) универсальный базовый класс и разработайте его наиболее тщательно.

[2] Когда вы определяете класс, который не реализует некоторый математический объект, вроде матрицы или комплексного числа, или тип низкого уровня, вроде связанного списка, то:

[a] не используйте глобальные данные;

[b] не используйте глобальные функции (не члены);

[c] не используйте открытые данные-члены;

[d] не используйте друзей, если не нужно избежать [a], [b] или [c];

[e] не обращайтесь к данным-членам или другим объектам непосредственно;

[f] не помещайте в класс «поле типа»; используйте виртуальные функции;

[g] не используйте inline-функции иначе, чем как средство существенной оптимизации.

1. ГЛАВА ПЕРВАЯ

Эта глава представляет собой краткий обзор основных черт языка программирования C++. Сначала приводится программа на C++, затем показано, как ее откомпилировать и запустить и как такая программа может выводить выходные данные и считывать входные. Для иллюстрации приведен ряд программ и частей программ на C++. К концу у вас должно сложиться общее представление об основных особенностях C++ и будет достаточно информации, чтобы писать простые программы. Для точного и полного объяснения понятий, затронутых даже в самом маленьком законченном примере, потребовалось бы несколько страниц определений. Чтобы не превращать эту главу в описание или в обсуждение общих понятий, примеры снабжены только самыми короткими определениями используемых терминов. Термины рассматриваются позже, когда будет больше примеров, способствующих обсуждению.

1.1. Начало

1.1.1. Вывод на экран

Прежде всего, давайте напишем программу, выводящую строку приветствия:

```
#include <iostream.h>
main()
{
    cout << «Hello, world\n»;
}
```

Строка **#include** сообщает компилятору, чтобы он включил стандартные возможности потока ввода и вывода, находящиеся в файле **iostream.h**. Без этих описаний выражение **cout << «Hello, world\n»** не имело бы смысла. Операция **<<** («поместить в») пишет свой первый аргумент во второй (в данном случае, строку **«Hello, world\n»** в стандартный поток вывода **cout**). Строка - это последовательность символов, заключенная в двойные кавычки. В строке символ обратной косой \, за которым следует другой символ, обозначает один специальный символ; в данном случае \n является символом новой строки. Таким образом, выводимые символы состоят из **Hello, world** и перевода строки.

Остальная часть программы

```
main() { ... }
```

определяет функцию, названную **main**. Функций в программе может быть сколь угодно много, но каждая программа должна содержать функцию с именем **main**, и работа программы начинается с выполнения этой функции.

1.1.2. Компиляция

Для получения выполняемого кода написанная на C++ программа должна быть скомпилирована. По своей сути процесс компиляции такой же, как и для большинства других языков программирования. Производится чтение и анализ текста программы, и, если не обнаружены ошибки, то генерируется код. Затем программа проверяется на наличие имен и операций, которые использовались, но не были определены (в нашем случае это `cout` и `<<`). Если это возможно, то программа делается полной посредством дополнения недостающих определений из библиотеки (есть стандартные библиотеки, и пользователи могут создавать свои собственные). В нашем случае `cout` и `<<` были описаны в `stream.h`, то есть были указаны их типы, но не было дано никаких подробностей относительно их реализации. В стандартной библиотеке содержится спецификация пространства и инициализирующий код для `cout` и `<<`. На самом деле, в этой библиотеке содержится и много других вещей, часть из которых описана в `stream.h`, однако к скомпилированной версии добавляется только подмножество библиотеки, необходимое для того, чтобы сделать нашу программу полной.

Процесс компиляции программы сильно зависит от среды разработки программы. Некоторые рекомендации описаны в методическом пособии по лабораторным работам.

1.1.3. Ввод

Следующая (довольно многословная) программа предлагает вам ввести число дюймов. После того, как вы это сделаете, она напечатает соответствующее число сантиметров.

```
#include
main()
{
    int inch = 0;           // inch - дюйм
    cout << «inches»;
```



```

    cin >> inch;
    cout << inch;
    cout << « in = «;
    cout << inch*2.54;
    cout << « cm\n»;
}

```

Первая строка функции `main()` описывает целую переменную `inch`. Ее значение считывается с помощью операции `>>` («взять из») над стандартным потоком ввода `cin`. Описания `cin` и `>>`, конечно же, находятся в «`iostream.h`». После ее запуска ваш терминал может выглядеть примерно так:

```

C:\WORK>input.exe
inches=12
12 in = 30.48 cm
C:\WORK>

```

В этом примере на каждую команду вывода приходится один оператор; это слишком длинно. Операцию вывода `<<` можно применять к ее собственному результату, так что последние четыре команды вывода можно было записать одним оператором:

```

    cout << inch << « in = « << inch*2.54 << «
cm\n»;

```

В последующих разделах ввод и вывод будут описаны гораздо более подробно. Вся эта глава, фактически, может рассматриваться как объяснение того, как можно написать предыдущие программы на языке, который не обеспечивает операции ввода-вывода. На самом деле, приведенные выше программы написаны на C++, «расширенном» операциями ввода-вывода посредством использования библиотек и включения файлов с помощью `#include`. Другими словами, язык C++ в том виде, в котором он описан в справочном руководстве, не определяет средств ввода-вывода; вместо этого исключительно с помощью средств, доступных любому программисту, определены операции `<<` и `>>`.

1.2. Комментарии

Часто бывает полезно вставлять в программу текст, который предназначается в качестве комментария только для читающего программу

человека и игнорируется компилятором в программе. В C++ это можно сделать одним из двух способов.

Символы `/*` начинают комментарий, заканчивающийся символами `*/`. Вся эта последовательность символов эквивалентна символу пропуска (например, символу пробела). Это наиболее полезно для многострочных комментариев и изъятия частей программы при редактировании, однако следует помнить, что комментарии `/* */` не могут быть вложенными.

Символы `//` начинают комментарий, который заканчивается в конце строки, на которой они появились. Опять вся последовательность символов эквивалентна пропуску. Этот способ наиболее полезен для коротких комментариев. Символы `//` можно использовать для того, чтобы закомментировать символы `/*` или `*/`, а символами `/*` можно закомментировать `//`.

1.3. Типы и описания

Каждое имя и каждое выражение имеет тип, определяющий операции, которые могут над ними производиться. Например, описание

```
int inch;
```

определяет, что `inch` имеет тип `int`, то есть `inch` является целой переменной.

Описание - это оператор, который вводит имя в программе. Описание задает тип этого имени. Тип определяет правильное использование имени или выражения. Для целых определены такие операции, как `+`, `-`, `*` и `/`. После того как включен файл `stream.h`, объект типа `int` может также быть вторым операндом `<<`, когда первый операнд `ostream`.

Тип объекта определяет не только то, какие операции могут к нему применяться, но и смысл этих операций. Например, оператор

```
cout << inch << « in = » << inch*2.54 <<
«cm\n»;
```

правильно обрабатывает четыре входных значения различным образом. Строки печатаются буквально, тогда как целое `inch` и значение с плавающей точкой `inch*2.54` преобразуются из их внутреннего представления в подходящее для человеческого глаза символьное представление.

В C++ есть несколько основных типов и несколько способов создавать новые. Простейшие виды типов C++ описываются в следующих разделах, а более интересные оставлены на потом.

1.3.1. Основные типы

Основные типы, наиболее непосредственно отвечающие средствам аппаратного обеспечения, такие:

```
char short int long float double
```

Первые четыре типа используются для представления целых, последние два - для представления чисел с плавающей точкой. Переменная типа `char` имеет размер, естественный для хранения символа на данной машине (обычно, байт), а переменная типа `int` имеет размер, соответствующий целой арифметике на данной машине (обычно слово). Диапазон целых чисел, которые могут быть представлены типом, зависит от его размера. В C++ размеры измеряются в единицах размера данных типа `char`, поэтому `char` по определению имеет размер единица. Соотношение между основными типами можно записать так:

```
1 = sizeof(char) <= sizeof(short) <=
sizeof(int) <= sizeof(long)

sizeof(float) <= sizeof(double)
```

В целом, предполагать что-либо еще относительно основных типов неразумно. В частности, то, что целое достаточно для хранения указателя, верно не для всех машин.

К основному типу можно применять прилагательное `const`. Это дает тип, имеющий те же свойства, что и исходный тип, за исключением того, что значение переменных типа `const` не может изменяться после инициализации.

```
const float pi = 3.14;
```

```
const char plus = '+';
```

Символ, заключенный в одинарные кавычки, является символьной константой. Заметьте, что часто константа, определенная таким образом, не занимает память; просто там, где требуется, ее значение может использоваться непосредственно. Константа должна инициализироваться при описании. Для переменных инициализация необязательна, но настоятельно рекомендуется. Оснований для введения локальной переменной без ее инициализации очень немного.

К любой комбинации этих типов могут применяться арифметические операции:

```
+ (плюс, унарный и бинарный);
```

- (минус, унарный и бинарный);
- * (умножение);
- / (деление).

А также операции сравнения:

- == (равно);
- != (не равно);
- < (меньше);
- > (больше);
- <= (меньше или равно);
- >= (больше или равно).

Заметьте, что целое деление дает целый результат: $7/2$ есть 3. Над целыми может выполняться операция `%` – операция получения остатка: $7\%2$ равно 1.

При присваивании и арифметических операциях C++ выполняет все осмысленные преобразования между основными типами, чтобы их можно было сочетать без ограничений:

```
double d = 1;
int i = 1;
d = d + i;
i = d + i;
```

1.3.2. Производные типы

Вот операции, создающие из основных типов новые типы:

- * указатель на;
- ***const** константный указатель на;
- & ссылка на;
- [] вектор;
- () функция возвращающая.

Например: **char* p** // указатель на символ;
char *const q // константный указатель на символ;
char v[10] // вектор из 10 символов.

Все векторы в качестве нижней границы индекса имеют ноль, поэтому в **v** десять элементов: **v[0] ... v[9]**. Переменная указатель может содержать адрес объекта соответствующего типа:

```
char c;
// ...
p = &c; // p указывает на c
```

Унарное & является операцией взятия адреса.

1.4. Выражения и операторы

В С++ имеется богатый набор операций, с помощью которых в выражениях образуются новые значения и изменяются значения переменных. Поток управления в программе задается с помощью операторов, а описания используются для введения в программе имен переменных, констант и т.д. Заметьте, что описания являются операторами, поэтому они свободно могут сочетаться с другими операторами.

1.4.1. Выражения

В С++ имеется большое число операций, и они будут объясняться там, где (и если) это потребуется. Следует учесть, что операции:

~ (дополнение);

& (И);

^ (исключающее ИЛИ);

| (включающее ИЛИ);

<< (логический сдвиг влево);

>> (логический сдвиг вправо)

применяются к целым, и что нет отдельного типа данных для логических действий.

Смысл операции зависит от числа операндов: унарное & является операцией взятия адреса, а бинарное & - это операция логического И. Смысл операции зависит также от типа ее операндов: + в выражении $a+b$ означает сложение с плавающей точкой, если операнды имеют тип float, но целое сложение, если они типа int. Ниже объясняется, как можно определить операцию для типа, определяемого пользователем, без потери ее значения, предопределенного для основных и производных типов.

В С++ есть операция присваивания =, а не оператор присваивания, как в некоторых языках. Таким образом, присваивание может встречаться в неожиданном контексте; например, $x = \text{sqrt}(a = 3 * x)$. В приведенном примере производятся следующие действия: 1) в переменную a записывается результат вычисления выражения $3 * x$, 2) значение переменной a передается в функцию `sqrt` а результат, который вернула данная функция, записывается в переменную x . Это бывает весьма полезно. Например, $a=b=c$ означает присвоение c объекту b , а затем объекту a . Другим свойством операции присваивания является

то, что она может совмещаться с большинством бинарных операций. Например, $x[i+3]*=4$ означает $x[i+3]=x[i+3]*4$, за исключением того факта, что выражение $x[i+3]$ вычисляется только один раз. Это дает привлекательную степень эффективности без необходимости обращения к оптимизирующим компиляторам. К тому же это более кратко.

В большинстве программ на C++ широко применяются указатели. Унарная операция $*$ разыменовывает указатель, т.е. $*p$ есть объект, на который указывает p . Эта операция также называется косвенной адресацией. Например, если имеется $\text{char}^* p$, то $*p$ есть символ, на который указывает p . Часто при работе с указателями бывают полезны операция увеличения $++$ и операция уменьшения $--$. Предположим, p указывает на элемент вектора v , тогда $p++$ делает p указывающим на следующий элемент вектора.

1.4.2. Операторы выражения

Самый обычный вид оператора - оператор «выражение». Он состоит из выражения, за которым следует точка с запятой. Например:

```
a = b*3+c;
cout << «go go go»;
lseek(fd,0,2);
```

1.4.3. Пустой оператор

Простейшей формой оператора является пустой оператор:

```
;
```

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

1.4.4. Блоки

Блок - это возможно пустой список операторов, заключенный в фигурные скобки:

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

Блок позволяет рассматривать несколько операторов как один. Область видимости имени, описанного в блоке, простирается до конца блока. Имя можно сделать невидимым с помощью описаний такого же имени во внутренних блоках. По-другому блок еще называют составным оператором.

1.4.5. Операторы if

Программа в следующем примере осуществляет преобразование дюймов в сантиметры и сантиметров в дюймы; предполагается, что вы укажете единицы измерения вводимых данных, добавляя i для дюймов и c для сантиметров:

```
#include
main()
{
    const float fac = 2.54;
    float x, in, cm;
    char ch = 0;
    cout << «введите длину: <<
    cin >> x >> ch;
    if (ch == 'i') {        // inch - дюймы
        in = x;
        cm = x*fac;
    }
    else if (ch == 'c') // cm - сантиметры
        in = x/fac;
        cm = x;
    }
    else
        in = cm = 0;
    cout << in << « in = « << cm << « cm\n»;
```

Заметьте, что условие в операторе if должно быть заключено в круглые скобки. Секция else является необязательной и может быть пропущена.

1.4.6. Операторы switch

Оператор switch производит сопоставление значения с множеством констант. Проверки в предыдущем примере можно записать так:

```
switch (ch) {
    case 'i':
        in = x;
        cm = x*fac;
        break;
    case 'c':
        in = x/fac;
```

```

        cm = x;
        break;
default:
        in = cm = 0;
        break;
}

```

Операторы `break` применяются для выхода из оператора `switch`. Константы в вариантах `case` должны быть различными, и если проверяемое значение не совпадает ни с одной из констант, выбирается вариант `default`. Программисту не обязательно предусматривать `default`.

1.4.7. Оператор `while`

Рассмотрим копирование строки, когда заданы указатель `p` на ее первый символ и указатель `q` на целевую строку. По соглашению строка оканчивается символом с целым значением 0.

```

while (p != 0) {
    *q = *p;           // скопировать символ
    q = q+1;
    p = p+1;
}
*q = 0;               // завершающий символ 0 ско-
пирован не был

```

Следующее после `while` условие должно быть заключено в круглые скобки. Условие вычисляется, и если его значение не ноль, выполняется непосредственно следующий за ним оператор. Это повторяется до тех пор, пока вычисление условия не даст ноль.

Этот пример слишком пространен. Можно использовать операцию `++` для непосредственного указания увеличения, и проверка упростится:

```

while (*p)  *q++ = *p++;
*q = 0;

```

где конструкция `*p++` означает: «взять символ, на который указывает `p`, затем увеличить `p`».

Пример можно еще упростить, так как указатель `p` разыменовывается дважды за каждый цикл. Копирование символа можно делать тогда же, когда производится проверка условия:

```

while (*q++ = *p++) ;

```


Здесь берется символ, на который указывает *p*, *p* увеличивается, этот символ копируется туда, куда указывает *q*, и *q* увеличивается. Если символ ненулевой, цикл повторяется. Поскольку вся работа выполняется в условии, не требуется ни одного оператора. Чтобы указать на это, используется пустой оператор. C++ (как и C) одновременно любят и ненавидят за возможность такого чрезвычайно краткого, ориентированного на выразительность программирования.

1.4.8. Оператор for

Рассмотрим копирование десяти элементов одного вектора в другой:

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

Это эквивалентно

```
int i = 0;
while (i<10) {
    q[i] = p[i];
    i++;
}
```

но более удобочитаемо, поскольку вся информация, управляющая циклом, локализована. При применении операции ++ к целой переменной к ней просто добавляется единица. Первая часть оператора `for` не обязательно должна быть описанием, она может быть любым оператором. Например:

```
for (i=0; i<10; i++) q[i]=p[i];
```

тоже эквивалентно предыдущей записи при условии, что *i* соответствующим образом описано раньше.

1.4.9. Описания

Описание - это оператор, вводящий имя в программе. Оно может также инициализировать объект с этим именем. Выполнение описания означает, что, когда поток управления доходит до описания, вычисляется инициализирующее выражение (инициализатор) и производится инициализация. Например:

```
for (int i = 1; i...
```

1.5. Функции

Функция - это именованная часть программы, к которой можно обращаться из других частей программы столько раз, сколько требуется. Рассмотрим программу, печатающую степени числа 2:

```
extern float pow(float, int); //pow() определена в другом месте
main()
{
    for (int i=0; i<10; i++) cout << pow(2,i)
<< «\n»;
}
```

Первая строка функции - описание, указывающее, что pow - функция, получающая параметры типа float и int и возвращающая float. Описание функции используется для того, чтобы сделать определенными обращения к функции в других местах.

При вызове тип каждого параметра функции сопоставляется с ожидаемым типом точно так же, как если бы инициализировалась переменная описанного типа. Это гарантирует надлежащую проверку и преобразование типов. Например, обращение pow(12.3,»abcd») вызовет недовольство компилятора, поскольку «abcd» является строкой, а не int. При вызове pow(2,i) компилятор преобразует 2 к типу float, как того требует функция. Функция pow может быть определена, например, так:

```
float pow(float x, int n)
{
    if (n < 0) error(«извините, отрицательный показатель для pow()»);
    switch (n) {
        case 0: return 1;
        case 1: return x;
        default: return x*pow(x,n-1);
    }
}
```

Первая часть определения функции задает имя функции, тип возвращаемого ею значения (если таковое имеется) и типы и имена ее параметров (если они есть). Значение возвращается из функции с помощью оператора return.

Разные функции обычно имеют разные имена, но функциям, выполняющим сходные действия над объектами различных типов, ино-

гда лучше дать возможность иметь одинаковые имена. Если типы их параметров различны, то компилятор всегда может различить их и выбрать для вызова нужную функцию. Может, например, иметься одна функция возведения в степень для целых переменных и другая для переменных с плавающей точкой:

```
int pow(int, int);
double pow(double, double);
//...
x=pow(2,10);
y=pow(2.0,10.0);
Описание
overload pow;
```

сообщает компилятору, что использование имени `pow` более, чем для одной функции, является умышленным.

Если функция не возвращает значения, то ее следует описать как `void`:

```
void swap(int* p, int* q) // поменять местами
{
    int t = *p;
    *p = *q;
    *q = t;
}
```

1.6. Структура программы

Программа на C++ обычно состоит из большого числа исходных файлов, каждый из которых содержит описания типов, функций, переменных и констант. Чтобы имя можно было использовать в разных исходных файлах для ссылки на один и тот же объект, оно должно быть описано как внешнее. Например:

```
extern double sqrt(double);
extern istream cin;
```

Самый обычный способ обеспечить согласованность исходных файлов - это поместить такие описания в отдельные файлы, называемые заголовочными (или хидер) файлами, а затем включить, то есть скопировать эти заголовочные файлы во все файлы, где нужны эти описания. Например, если описание `sqrt` хранится в заголовочном файле для стандартных математических функций `math.h` и вы хотите извлечь квадратный корень из 4, можно написать:

```
#include
//...
x = sqrt(4);
```

Поскольку обычные заголовочные файлы включаются во многие исходные файлы, они не содержат описаний, которые не должны повторяться. Например, тела функций даются только для inline-подставляемых функций и инициализаторы даются только для констант. За исключением этих случаев, заголовочный файл является хранилищем информации о типах. Он обеспечивает интерфейс между отдельно компилируемыми частями программы.

В команде включения `include` имя файла, заключенное в угловые скобки, например, относится к файлу с этим именем в стандартном каталоге (например, это `c:\tc\include`); на файлы, находящиеся в каких-либо других местах ссылаются с помощью имен, заключенных в двойные кавычки. Например:

```
#include «math1.h»
#include «\usr\bs\math2.h»
```

включит `math1.h` из текущего пользовательского каталога, а `math2.h` из каталога `\usr\bs`.

Здесь приводится очень маленький законченный пример программы, в котором строка определяется в одном файле, а ее печать производится в другом. Файл `header.h` определяет необходимые типы:

```
// header.h
extern char* prog_name;
extern void f();
```

В файле `main.c` находится главная программа:

```
// main.c
#include «header.h»
char* prog_name = «дурацкий, но полный»;
main()
{
    f();
}
```

а файл `f.c` печатает строку:

```
// f.c
#include
#include «header.h»
void f()
{
    cout << prog_name << «\n»;
}
```

1.7. Классы

Давайте посмотрим, как мы могли бы определить тип потока вывода `ostream`. Чтобы упростить задачу, предположим, что для буферизации определен тип `streambuf`. Тип `streambuf` на самом деле определен в файле `ostream.h`, где также находится и настоящее определение `ostream`. Пожалуйста, не испытывайте примеры, определяющие `ostream` в этом и последующих разделах: пока вы не сможете полностью избежать использования, компилятор будет возражать против переопределений.

Определение типа, определяемого пользователем (который в C++ называется `class`, т.е. класс), специфицирует данные, необходимые для представления объекта этого типа, и множество операций для работы с этими объектами. Определение имеет две части: закрытую (`private`) часть, содержащую информацию, которой может пользоваться только его разработчик, и открытую (`public`) часть, представляющую интерфейс типа с пользователем:

```
class ostream {
    streambuf* buf;
    int state;
public:
    void put(char*);
    void put(long);
    void put(double);
}
```

Описания после метки `public` задают интерфейс: пользователь может обращаться только к трем функциям `put()`. Описания перед меткой `public` задают представление объекта класса `ostream`; имена `buf` и `state` могут использоваться только функциями `put()`, описанными в открытой части.

class определяет тип, а не объект данных, поэтому чтобы использовать ostream, мы должны один такой объект описать (так же, как мы описываем переменные типа int):

```
ostream my_out;
```

Считая, что my_out был соответствующим образом проинициализирован, его можно использовать, например, так:

```
my_out.put («Hello, world\n»);
```

С помощью операции «точка» выбирается член класса для данного объекта этого класса. Здесь для объекта my_out вызывается член-функция put().

Функция может определяться так:

```
void ostream::put(char* p)
{
    while (*p) buf.sputc(*p++);
}
```

где sputc() - функция, которая помещает символ в streambuf. Префикс ostream необходим, чтобы отличить put() ostream'a от других функций с именем put().

Для обращения к функции члену должен быть указан объект класса. В функции члене можно сослаться на этот объект неявно, как это делалось выше в ostream::put(): в каждом вызове buf относится к члену buf-объекта, для которого функция вызвана.

Можно также сослаться на этот объект явно, посредством указателя с именем this. В функции члене класса X this неявно описан как X* (указатель на X) и инициализирован указателем на тот объект, для которого эта функция вызвана. Определение ostream::put() можно также записать в виде:

```
void ostream::put(char* p)
{
    while (*p) this->buf.sputc(*p++);
}
```

Операция -> применяется для выбора члена объекта, заданного указателем.

1.8. Перегрузка операций

Настоящий класс ostream определяет операцию <<, чтобы сделать удобным вывод нескольких объектов одним оператором. Давайте посмотрим, как это сделано.

Чтобы определить @, где @ - некоторая операция языка C++, для каждого определяемого пользователем типа вы определяете функцию с именем operator@, которая получает параметры соответствующего типа. Например:

```
class ostream {
    //...
    ostream operator<<(char*);
};

ostream ostream::operator<<(char* p)
{
    while (*p) buf.sputc(*p++);
    return *this;
}
```

определяет операцию << как член класса ostream; а если применить операцию взятия адреса, то вы получите адрес объекта, на который ссылается ссылка:

```
&s1 == &my_out
```

Первая очевидная польза от ссылок состоит в том, чтобы обеспечить передачу адреса объекта, а не самого объекта, в функцию вывода (в некоторых языках это называется передачей параметра по ссылке):

```
ostream& operator<<(ostream& s, complex z) {
    return s << «(« << z.real << «,» << z.imag << «)»»;
}
```

Достаточно интересно, что тело функции осталось без изменений, но если вы будете осуществлять присваивание s, то будете воздействовать на сам объект, а не на его копию. В данном случае то, что возвращается ссылка, также повышает эффективность, поскольку очевидный способ реализации ссылки - это указатель, а передача указателя гораздо дешевле, чем передача большой структуры данных.

Ссылки также существуют для определения потока ввода, поскольку операция ввода получает в качестве операнда переменную для считывания. Если бы ссылки не использовались, то пользователь должен был бы явно передавать указатели в функции ввода.

```

class istream {
    //...
    int state;
public:
    istream& operator>>(char&ap);
    istream& operator>>(char*);
    istream& operator>>(int&);
    istream& operator>>(long&);
    //...
};

```

Заметьте, что для чтения `long` и `int` используются разные функции, тогда как для их печати требовалась только одна. Это вполне обычно, и причина в том, что `int` может быть преобразовано в `long` по стандартным правилам неявного преобразования (§6.6), избавляя таким образом программиста от беспокойства по поводу написания обеих функций ввода.

1.9. Конструкторы

Определение `ostream` как класса сделало члены-данные закрытыми. Только функция-член имеет доступ к закрытым членам, поэтому надо предусмотреть функцию для инициализации. Такая функция называется конструктором и отличается тем, что имеет то же имя, что и ее класс:

```

class ostream {
    //...
    ostream(streambuf*);
    ostream(int size, char* s);
};

```

Здесь задано два конструктора. Один получает вышеупомянутый `streambuf` для реального вывода, другой получает размер и указатель на символ для форматирования строки. В описании необходимый для конструктора список параметров присоединяется к имени. Теперь вы можете, например, описать такие потоки:

```

ostream my_out(&some_stream_buffer);
char xx[256];
ostream xx_stream(256,xx);

```

Описание `my_out` не только задает соответствующий объем памяти где-то в другом месте, оно также вызывает конструктор

`ostream::ostream(streambuf*)`, чтобы инициализировать его параметром `&some_stream_buffer`, предположительно указателем на подходящий объект класса `streambuf`. Описание конструкторов для класса не только дает способ инициализации объектов, но также обеспечивает то, что все объекты этого класса будут проинициализированы. Если для класса были описаны конструкторы, то невозможно описать переменную этого класса так, чтобы конструктор не был вызван. Если класс имеет конструктор, не получающий параметров, то этот конструктор будет вызываться в том случае, если в описании нет ни одного параметра.

1.10. Векторы

Встроенное в C++ понятие вектора было разработано так, чтобы обеспечить максимальную эффективность выполнения при минимальном расходе памяти. Оно также (особенно когда используется совместно с указателями) является весьма универсальным инструментом для построения средств более высокого уровня. Вы могли бы, конечно, возразить, что размер вектора должен задаваться как константа, что нет проверки выхода за границы вектора и т.д. Ответ на подобные возражения таков: «Вы можете запрограммировать это сами». Давайте посмотрим, действительно ли оправдан такой ответ. Другими словами, проверим средства абстракции языка C++, попытавшись реализовать эти возможности для векторных типов, которые мы создадим сами, и посмотрим, какие с этим связаны трудности, каких это требует затрат и насколько получившиеся векторные типы удобны в обращении.

```
class vector {
    int* v;
    int sz;
public:
    vector(int);           // конструктор
    ~vector();             // деструктор
    int size() { return sz; }
    void set_size(int);
    int& operator[] (int);
    int& elem(int i) { return v[i]; }
};
```

Функция `size` возвращает число элементов вектора, таким образом индексы должны лежать в диапазоне `0 ... size()-1`. Функция `set_size` сделана для изменения этого размера, `elem` обеспечивает доступ к элементам без проверки индекса, а `operator[]` дает доступ с проверкой границ.

Идея состоит в том, чтобы класс сам был структурой фиксированного размера, управляющей доступом к фактической памяти вектора, которая выделяется конструктором вектора с помощью распределителя свободной памяти `new`:

```
vector::vector(int s)
{
    if (s<=0) error(«плохой размер вектора»);
    sz = s;
    v = new int[s];
}
```

Теперь вы можете описывать векторы типа `vector` почти столь же элегантно, как и векторы, встроенные в сам язык:

```
vector v1(100);
vector v2(nelem*2-4);
```

Операцию доступа можно определить как

```
int& vector::operator[](int i)
{
    if(i<0 || sz<=i) error(«индекс выходит за
границы»);
    return v[i];
}
```

Операция `||` (две вертикальные черты) - это логическая операция ИЛИ. Ее правый операнд вычисляется только тогда, когда это необходимо, то есть если вычисление левого операнда дало ноль. Возвращение ссылки обеспечивает то, что запись `[]` может использоваться с любой стороны операции присваивания:

```
v1[x] = v2[y];
```

Функция со странным именем `~vector` - это деструктор, то есть функция, описанная для того, чтобы она неявно вызывалась, когда объект класса выходит из области видимости. Деструктор класса `C` имеет имя `~C`. Если его определить как

```
vector::~vector()
{
    delete v;
}
```

то он будет, с помощью операции `delete`, освобождать пространство, выделенное конструктором, поэтому, когда `vector` выходит из области

видимости, все его пространство возвращается обратно в память для дальнейшего использования.

1.11. *Inline-подстановка*

Если часто повторяется обращение к очень маленькой функции, то вы можете начать беспокоиться о стоимости вызова функции. Обращение к функции члену не дороже обращения к функции не члену с тем же числом параметров (надо помнить, что функция член всегда имеет хотя бы один параметр), и вызовы в функций в C++ примерно столь же эффективны, сколь и в любом языке. Однако для слишком маленьких функций может встать вопрос о накладных расходах на обращение. В этом случае можно рассмотреть возможность спецификации функции как *inline*-подставляемой. Если вы поступите таким образом, то компилятор сгенерирует для функции соответствующий код в месте ее вызова. Семантика вызова не изменяется. Если, например, *size* и *elem* *inline*-подставляемые, то

```
vector s(100);
//...
i = s.size();
x = elem(i-1);
```

порождает код, эквивалентный

```
//...
i = 100;
x = s.v[i-1];
```

C++-компилятор обычно достаточно разумен, чтобы генерировать настолько хороший код, насколько вы можете получить в результате прямого макрорасширения. Разумеется, компилятор иногда вынужден использовать временные переменные и другие уловки, чтобы сохранить семантику.

Вы можете указать, что вы хотите, чтобы функция была *inline*-подставляемой, поставив ключевое слово *inline* или, для функции члена, просто включив определение функции в описание класса, как это сделано в предыдущем примере для *size()* и *elem()*.

При хорошем использовании *inline*-функции резко повышают скорость выполнения и уменьшают размер объектного кода. Однако *inline*-функции запутывают описания и могут замедлить компиляцию, поэтому, если они не необходимы, то их желательно избегать. Чтобы *inline*-функция давала существенный выигрыш по сравнению с обычной функцией, она должна быть очень маленькой.

1.12. Производные классы

Теперь давайте определим вектор, для которого пользователь может задавать границы изменения индекса.

```
class vec: public vector {
    int low, high;
public:
    vec(int, int);
    int& elem(int);
    int& operator[](int);
};
```

Определение `vec` как

```
:public vector
```

означает, в первую очередь, что `vec` это `vector`. То есть, тип `vec` имеет (наследует) все свойства типа `vector` дополнительно к тем, что описаны специально для него. Говорят, что класс `vector` является базовым классом для `vec`, а о `vec` говорится, что он производный от `vector`.

Класс `vec` модифицирует класс `vector` тем, что в нем задается другой конструктор, который требует от пользователя указывать две границы изменения индекса, а не длину, и имеются свои собственные функции доступа `elem(int)` и `operator[](int)`. Функция `elem()` класса `vec` легко выражается через `elem()` класса `vector`:

```
int& vec::elem(int i)
{
    return vector::elem(i-low);
}
```

Операция разрешения области видимости `::` используется для того, чтобы не было бесконечной рекурсии обращения к `vec::elem()` из нее самой. С помощью унарной операции `::` можно ссылаться на нелокальные имена. Было бы разумно описать `vec::elem()` как `inline`, поскольку, скорее всего, эффективность существенна, но необязательно, неразумно и невозможно написать ее так, чтобы она непосредственно использовала закрытый член `v` класса `vector`. Функции производного класса не имеют специального доступа к закрытым членам его базового класса.

Конструктор можно написать так:

```
vec::vec(int lb, int hb) : (hb-lb+1)
{
    if (hb-lb<0) hb = lb;
    low = lb;
    high = hb;
}
```

Запись : (hb-lb+1) используется для определения списка параметров конструктора базового класса `vector::vector()`. Этот конструктор вызывается перед телом `vec::vec()`. Вот небольшой пример, который можно запустить, если скомпилировать его вместе с остальными описаниями `vector`:

```
#include
void error(char* p)
{
    cerr << p << «n»; // cerr - выходной по-
ток сообщений об ошибках
    exit(1);
}

void vector::set_size(int) { /* пустышка */ }
int& vec::operator[](int i)
{
    if (i
```

1.13. Еще об операциях

Другое направление развития - снабдить векторы операциями:

```
class Vec : public vector {
public:
    Vec(int s) : (s) {}
    Vec(Vec&);
    ~Vec() {}
    void operator=(Vec&);
    void operator*=(Vec&);
    void operator*=(int);
    //...
};
```

Обратите внимание на способ определения конструктора производного класса, `Vec::Vec()`, когда он передает свой параметр конструктору базового класса `vector::vector()` и больше не делает ничего. Это полезная парадигма. Операция присваивания перегружена, ее можно определить так:

```
void Vec::operator=(Vec& a)
{
    int s = size();
    if (s!=a.size()) error(«плохой размер век-
тора для =»);
```

```

        for (int i = 0; i

void error(char* p) {
    cerr << p << «\n»;
    exit(1);
}

void vector::set_size(int) { /*...*/ }

int& vec::operator[](int i) { /*...*/ }

main()
{
    Vec a(10);
    Vec b(10);
    for (int i=0; i

```

1.14. Друзья (friends)

Функция `operator+()` не воздействует непосредственно на представление вектора. Действительно, она не может этого делать, поскольку не является членом. Однако иногда желательно дать функциям не членам возможность доступа к закрытой части класса. Например, если бы не было функции «доступа без проверки» `vector::elem()`, вам пришлось бы проверять индекс `i` на соответствие границам три раза за каждый проход цикла. Здесь мы избежали этой сложности, но она довольно типична, поэтому у класса есть механизм предоставления права доступа к своей закрытой части функциям не членам. Просто в описание класса помещается описание функции, перед которым стоит ключевое слово `friend`. Например, если имеется

```

class Vec; // Vec - имя класса
class vector {
    friend Vec operator+(Vec, Vec);
    //...
};

```

то вы можете написать

```

Vec operator+(Vec a, Vec b)
{
    int s = a.size();
    if (s != b.size()) error(«плохой размер
вектора для +»);

```

```

    Vec& sum = *new Vec(s);
    int* sp = sum.v;
    int* ap = a.v;
    int* bp = b.v;
    while (s--) *sp++ = *ap++ + *bp++;
    return sum;
}

```

Одним из особенно полезных аспектов механизма friend является то, что функция может быть другом двух и более классов. Чтобы увидеть это, рассмотрим определение `vector` и `matrix`, а затем определение функции умножения.

1.15. Обобщенные векторы

«Пока все хорошо, - можете сказать вы, - но я хочу, чтобы один из этих векторов был типа `matrix`, который я только что определил». К сожалению, в C++ не предусмотрены средства для определения класса векторов с типом элемента в качестве параметра. Один из способов - продублировать описание и класса, и его функций членов. Это не идеальный способ, но зачастую вполне приемлемый.

Вы можете воспользоваться препроцессором, чтобы механизировать работу. Например, класс `vector` - упрощенный вариант класса, который можно найти в стандартном заголовочном файле. Вы могли бы написать:

```

#include
declare(vector,int);
main()
{
    vector(int) vv(10);
    vv[2] = 3;
    vv[10] = 4; // ошибка: выход за границы
}

```

Файл `vector.h` таким образом определяет макросы, чтобы `declare(vector,int)` после расширения превращался в описание класса `vector`, очень похожий на тот, который был определен выше, а `implement(vector,int)` расширялся в определение функций этого класса. Поскольку `implement(vector,int)` в результате расширения превращается в определение функций, его можно использовать в программе только один раз, в то время как `declare(vector,int)` должно использоваться

по одному разу в каждом файле, работающем с этим типом целых векторов.

```
declare(vector, char);
//...
implement(vector, char);
```

даст вам отдельный тип «вектор-символов».

1.16. *Полиморфные Вектора*

У вас есть другая возможность - определить ваш векторный и другие вмещающие классы через указатели на объекты некоторого класса:

```
class common {
    //...
};
class vector {
    common** v;
    //...
public:
    cvector(int);
    common*& elem(int);
    common*& operator[] (int);
    //...
};
```

Заметьте, что поскольку в таких векторах хранятся указатели, а не сами объекты, объект может быть «в» нескольких таких векторах одновременно. Это очень полезное свойство подобных вмещающих классов, таких как векторы, связанные списки, множества и т.д. Кроме того, можно присваивать указатель на производный класс указателю на его базовый класс, поэтому можно использовать приведенный выше cvector для хранения указателей на объекты всех производных от common классов. Например:

```
class apple : public common { /*...*/ }
class orange : public common { /*...*/ }

class apple_vector : public cvector {
public:
    cvector fruitbowl(100);
    //...
    apple aa;
    orange oo;
    //...
```



```
fruitbowl[0] = &aa;
fruitbowl[1] = &oo;
}
```

Однако точный тип объекта, вошедшего в такой вмещающий класс, больше компилятору не известен. Например, в предыдущем примере вы знаете, что элемент вектора является `common`, но является он `apple` или `orange`? Обычно точный тип должен в последствии восстановлен, чтобы обеспечить правильное использование объекта. Для этого нужно или в какой-то форме хранить информацию о типе в самом объекте, или обеспечить, чтобы во вмещающий класс помещались только объекты данного типа. Последнее легко достигается с помощью производного класса. Вы можете, например, создать вектор указателей на `apple`:

```
class apple_vector : public cvector {
public:
    apple*& elem(int i)
        { return (apple*&) cvector::elem(i); }
    //...
};
```

используя запись приведения к типу (тип) выражение, чтобы преобразовать `common*&` (ссылку на указатель на `common`), которую возвращает `cvector::elem`, в `apple*&`. Такое применение производных классов создает альтернативу обобщенным классам. Писать его немного труднее (если не использовать макросы таким образом, чтобы производные классы фактически реализовывали бы обобщенные классы), но оно имеет то преимущество, что все производные классы совместно используют единственную копию функции базового класса. В случае обобщенных классов, таких как `vector(type)`, для каждого нового используемого типа должна создаваться (с помощью `implement()`) новая копия таких функций. Другой способ, хранение идентификации типа в каждом объекте, приводит нас к стилю программирования, который часто называют объекто-основанным или объектно-ориентированным.

1.17. Виртуальные функции

Предположим, что мы пишем программу для изображения фигур на экране. Общие атрибуты фигуры представлены классом `shape`, а специальные атрибуты - специальными классами:

```
class shape {
    point center;
    color col;
```

```

        //...
public:
    void move(point to) { center=to; draw(); }
    point where() { return center; }
    virtual void draw();
    virtual void rotate(int);
    //...
};

```

Функции, которые можно определить, не зная точно определенной фигуры (например, `move` и `where`, то есть, «передвинуть» и «где»), можно описать как обычно. Остальные функции описываются как `virtual`, то есть такие, которые должны определяться в производном классе. Например:

```

class circle: public shape {
    int radius;
public:
    void draw();
    void rotate(int i) {}
    //...
};

```

Теперь, если `shape_vec` - вектор фигур, то можно написать:

```

for (int i = 0; i

```

2. ГЛАВА ВТОРАЯ

В этой главе описаны основные типы (`char`, `int`, `float` и т.д.) и основные способы построения из них новых типов (функций, векторов, указателей и т.д.). Имя вводится в программе посредством описания, которое задает его тип и, возможно, начальное значение. Даны понятия описания, определения, области видимости имен, времени жизни объектов и типов. Описываются способы записи констант в C++, а также способы определения символических констант. Примеры просто демонстрируют характерные черты языка. Более развернутый и реалистичный пример приводится в следующей главе для знакомства с выражениями и операторами языка C++. Механизмы задания типов, определяемых пользователем, с присоединенными операциями представлены в главах 4, 5 и 6 и здесь не упоминаются.

2.1. Описания

Прежде чем имя (идентификатор) может быть использовано в C++-программе, оно должно быть описано. Это значит, что надо задать его тип, чтобы сообщить компилятору, к какого вида объектам относится имя. Вот несколько примеров, иллюстрирующих разнообразие описаний:

```
char ch;
int count = 1;
char* name = «Bjarne»;
struct complex { float re, im; };
complex cvar;
extern complex sqrt(complex);
extern int error_number;
typedef complex point;
float real(complex* p) { return p->re; };
const double pi = 3.1415926535897932385;
struct user;
```

Как можно видеть из этих примеров, описание может делать больше, чем просто ассоциировать тип с именем. Большинство описаний являются также определениями; то есть они также определяют для имени сущность, к которой оно относится. Для `ch`, `count` и `cvar` этой сущностью является соответствующий объем памяти, который должен использоваться как переменная - эта память будет выделена. Для `real` это заданная функция. Для `constant pi` это значение 3,1415926535897932385. Для `complex` этой сущностью является новый

тип. Для `point` это тип `complex`, поэтому `point` становится синонимом `complex`. Только описания

```
extern complex sqrt(complex);
extern int error_number;
struct user;
```

не являются одновременно определениями. Это означает, что объект, к которому они относятся, должен быть определен где-то еще. Код (тело) функции `sqrt` должен задаваться неким другим описанием, память для переменной `error_number` типа `int` должна выделяться неким другим описанием, и какое-то другое описание типа `user` должно определять, что он из себя представляет. В C++-программе всегда должно быть только одно определение каждого имени, но описаний может быть много, и все описания должны согласовываться с типом объекта, к которому они относятся, поэтому в этом фрагменте есть две ошибки:

```
int count;
int count;    // ошибка: переопределение
extern int error_number;
extern int error_number; // ошибка: несоответствие типов
```

а в этом - ни одной:

```
extern int error_number;
extern int error_number;
```

Некоторые описания задают «значение» для сущностей, которые они определяют:

```
struct complex { float re, im; };
typedef complex point;
float real(complex* p) { return p->re };
const double pi = 3.1415926535897932385;
```

Для типов, функций и констант «значение» неизменно; для неконстантных типов данных начальное значение может впоследствии изменяться:

```
int count = 1;
char* name = «Bjarne»;
//...
count = 2;
name = «Marian»;
```

Из всех определений только `char ch`; не задает значение. Всякое описание, задающее значение, является определением.

2.1.1. Область видимости

Описание вводит имя в области видимости, то есть имя может использоваться только в определенной части программы. Для имени, описанного в функции (такое имя часто называют локальным), эта область видимости простирается от точки описания до конца блока, в котором появилось описание; для имени не в функции и не в классе (называемого часто глобальным именем) область видимости простирается от точки описания до конца файла, в котором появилось описание. Описание имени в блоке может скрывать (прятать) описание во внутреннем блоке или глобальное имя. Это значит, что можно переопределять имя внутри блока для ссылки на другой объект. После выхода из блока имя вновь обретает свое прежнее значение. Например:

```

    int x;                // глобальное x
    f() {
        int x;           // локальное x прячет гло-
бальное x
        x = 1;            // присвоить локальному x
        {
            int x;        // прячет первое локальное x
            x = 2;        // присвоить второму локаль-
ному x
        }
        x = 3;            // присвоить первому локаль-
ному x
    }

    int* p = &x;          // взять адрес глобального x

```

Скрытие имен неизбежно при написании больших программ. Однако читающий человек легко может не заметить, что имя скрыто, и некоторые ошибки, возникающие вследствие этого, очень трудно обнаружить, главным образом, потому, что они редкие. Значит, скрытие имен следует минимизировать. Использование для глобальных переменных имен вроде `i` или `x` напрашивается на неприятности, так как велика вероятность использования этих имен в качестве локальных переменных.

С помощью применения операции разрешения области видимости `::` можно использовать скрытое глобальное имя. Например:

```

int x;
f()
{
    int x = 1;          // скрывает глобальное x
    ::x = 2;            // присваивает глобальному x
}

```

Но возможности использовать скрытое локальное имя нет.

Область видимости имени начинается в точке описания. Это означает, что имя можно использовать даже для задания его собственного значения. Например:

```

int x;
f()
{
    int x = x;          // не логично
}

```

Это не является недопустимым, хотя и бессмысленно, и компилятор предупредит, что `x` «used before set» («использовано до того, как задано»), если вы попытаетесь так сделать. Можно, напротив, не применяя операцию `::`, использовать одно имя для ссылки на два различных объекта в блоке. Например:

```

int x;

f()                      // не логично
{
    int y = x;           // глобальное x
    int x = 22;
    y = x;               // локальное x
}

```

Переменная `y` инициализируется значением глобального `x`, 11, а затем ему присваивается значение локальной переменной `x`, 22.

Имена параметров функции считаются описанными в самом внешнем блоке функции, поэтому

```

f(int x)
{
    int x;               // ошибка
}

```

содержит ошибку, так как `x` определено дважды в одной и той же области видимости.

2.1.2. Объекты и адреса (Lvalue)

Можно назначать и использовать переменные, не имеющие имен, и можно осуществлять присваивание выражениям странного вида (например, `*p[a+10]=7`). Следовательно, есть потребность в имени «нечто в памяти». Вот соответствующая цитата из справочного руководства по C++: «Объект есть область памяти; lvalue есть выражение, ссылающееся на объект». Слово «lvalue» первоначально было придумано для значения «нечто, что может стоять в левой части присваивания». Однако не всякий адрес можно использовать в левой части присваивания; бывают адреса, ссылающиеся на константу.

2.1.3. Время жизни

Если программист не указал иного, то объект создается, когда встречается его описание, и уничтожается, когда его имя выходит из области видимости. Объекты с глобальными именами создаются и инициализируются один раз (только) и «живут» до завершения программы. Объекты, определенные описанием с ключевым словом `static`, ведут себя так же. Например:

```
int a = 1;
void f()
{
    int b = 1;           // инициализируется
    при каждом вызове f()
    static int c = 1;    // инициализируется
    только один раз
    cout << « a = « << a++
         << « b = « << b++
         << « c = « << c++ << «\n»;
}

main()
{
    while (a < 4) f();
}
```

производит вывод

```
a = 1 b = 1 c = 1
a = 2 b = 1 c = 2
a = 3 b = 1 c = 3
```

Неинициализированная явно статическая (static) переменная неявно инициализируется нулем.

С помощью операций new и delete программист может также создавать объекты, время жизни которых управляется непосредственно.

2.2. Имена

Имя (идентификатор) состоит из последовательности букв и цифр. Первый символ должен быть буквой. Символ подчеркива _ считается буквой. C++ не налагает ограничений на число символов в имени, но некоторые части реализации находятся вне ведения автора компилятора (в частности, загрузчик), и они, к сожалению, такие ограничения налагают. Некоторые среды выполнения также делают необходимым расширить или ограничить набор символов, допустимых в идентификаторе; расширения (например, при допущении в именах символа \$) порождают непереносимые программы. В качестве имени не могут использоваться ключевые слова C++. Примеры имен:

```
hello    this_is_a_most_unusually_long_name
DEFINED  foO    bAr    u_name    HorseSense
var0     var1    CLASS   _class    ____
```

Примеры последовательностей символов, которые не могут использоваться как идентификаторы:

```
012      a fool    $sys    class    3var
pay.due  foo~bar   .name    if
```

Буквы в верхнем и нижнем регистрах считаются различными, поэтому Count и count - различные имена, но вводить имена, лишь незначительно отличающиеся друг от друга, нежелательно. Имена, начинающиеся с подчеркива, по традиции используются для специальных средств среды выполнения, поэтому использовать такие имена в прикладных программах нежелательно.

Во время чтения программы компилятор всегда ищет наиболее длинную строку, составляющую имя, поэтому var10 - это одно имя, а не имя var, за которым следует число 10; и elseif - одно имя, а не ключевое слово else, после которого стоит ключевое слово if.

2.3. Типы

Каждое имя (идентификатор) в C++-программе имеет ассоциированный с ним тип. Этот тип определяет, какие операции можно при-

менять к имени (то есть к объекту, на который оно ссылается) и как эти операции интерпретируются. Например:

```
int error number;
float real(complex* p);
```

Поскольку `error number` описано как `int`, его можно присваивать, использовать в арифметических выражениях и т.д. Тогда как функция `real` может вызываться с адресом `complex` в качестве параметра. Можно взять адрес любого из них. Некоторые имена, вроде `int` и `complex`, являются именами типов. Обычно имя типа используется в описании для спецификации другого имени. Единственные отличные от этого действия над именем типа - это `sizeof` (для определения количества памяти, которая требуется для хранения объекта типа) и `new` (для размещения объекта типа в свободной памяти). Например:

```
main()
{
    int* p = new int;
    cout << «sizeof(int) = « << sizeof(int) «\n»;
}
```

Имя типа можно также использовать для задания явного преобразования одного типа в другой, например:

```
float f;
char* p;
//...
long ll = long(p); // преобразует p в long
int i = int(f);    // преобразует f в int
```

2.3.1. Основные типы

В C++ есть набор основных типов, которые соответствуют наиболее общим основным единицам памяти компьютера и наиболее общим основным способам их использования:

```
char
short int
int
long int
```

для представления целых различных размеров,

```
float
double
```

для представления чисел с плавающей точкой,

```
unsigned char
```

```

unsigned short int
unsigned int
unsigned long int

```

для представления беззнаковых целых, логических значений, битовых массивов и т.п. Для большей компактности записи можно опускать `int` в комбинациях из нескольких слов, что не меняет смысла: так, `long` означает `long int`, и `unsigned` означает `unsigned int`. В общем, когда в описании опущен тип, он предполагается `int`. Например:

```

const a = 1;
static x;

```

все определяют объект типа `int`.

Целый тип `char` наиболее удобен для хранения и обработки символов на данном компьютере; обычно это 8-битовый байт. Размеры объектов C++ выражаются в единицах размера `char`, поэтому по определению `sizeof(char)=1`. В зависимости от аппаратного обеспечения `char` является знаковым или беззнаковым целым. Тип `unsigned char`, конечно, всегда беззнаковый, и при его использовании получают более переносимые программы, но из-за применения его вместо просто `char` могут возникать значительные потери в эффективности.

Причина того, что предоставляется более, чем один целый тип, более, чем один беззнаковый тип, и более, чем один тип с плавающей точкой, в том, чтобы дать возможность программисту воспользоваться характерными особенностями аппаратного обеспечения. На многих машинах между различными разновидностями основных типов существуют значительные различия в потребностях памяти, временах доступа к памяти и временах вычислений. Зная машину, обычно легко, например, выбрать подходящий тип для конкретной переменной. Написать действительно переносимую программу нижнего уровня сложнее. Вот все, что гарантируется относительно размеров основных типов:

```

1==sizeof(char)<=sizeof(short)<=
sizeof(int) <= sizeof(long)
sizeof(float)<=sizeof(double)

```

Однако обычно разумно предполагать, что в `char` могут храниться целые числа в диапазоне 0...127 (в нем всегда могут храниться символы машинного набора символов), что `short` и `int` имеют не менее 16 бит, что `int` имеет размер, соответствующий целой арифметике, и что `long` имеет по меньшей мере 24 бита. Предполагать что-либо помимо

этого рискованно, и даже эти эмпирические правила применимы не везде.

Беззнаковые (unsigned) целые типы идеально подходят для применений, в которых память рассматривается как массив битов. Использование unsigned вместо int с тем, чтобы получить еще один бит для представления положительных целых, почти никогда не оказывается хорошей идеей. Попытки гарантировать то, что некоторые значения положительны, посредством описания переменных как unsigned обычно срываются из-за правил неявного преобразования. Например:

```
unsigned surprise = -1;
```

допустимо (но компилятор обязательно сделает предупреждение).

2.3.2. Неявное преобразование типа

Основные типы можно свободно сочетать в присваиваниях и выражениях. Везде, где это возможно, значения преобразуются так, чтобы информация не терялась.

Существуют случаи, в которых информация может теряться или искажаться. Присваивание значения одного типа переменной другого типа, представление которого содержит меньшее число бит, неизбежно является источником неприятностей. Допустим, например, что следующая часть программы выполняется на машине с двоичным дополнением представления целых и 8-битовыми символами:

```
int i1 = 256+255;
char ch = i1           // ch == 255
int i2 = ch;           // i2 == ?
```

В присваивании `ch=i1` теряется один бит (самый значимый!), и `ch` будет содержать двоичный код «все-единицы» (т.е. 8 единиц); при присваивании `i2` это никак не может превратиться в 511! Но каким же может быть значение `i2`? На DEC VAX, где `char` знаковые, ответ будет -1; на AT&T 3B-20, где `char` беззнаковые, ответ будет 255. В C++ нет динамического (т.е. действующего во время исполнения) механизма для разрешения такого рода проблем, а выяснение на стадии компиляции вообще очень сложно, поэтому программист должен быть внимателен.

2.3.3. Производные типы

Другие типы модно выводить из основных типов (и типов, определенных пользователем) посредством операций описания:

* указатель,
 & ссылка,
 [] вектор,
 () функция

и механизма определения структур. Например:

```
int* a;
float v[10];
char* p[20];    // вектор из 20 указателей на
СИМВОЛ
void f(int);
struct str { short length; char* p; };
```

Правила построения типов с помощью этих операций подробно объясняются позже. Основная идея состоит в том, что описание производного типа отражает его использование. Например:

```
int v[10];    // описывает вектор
i = v[3];    // использует элемент вектора
int* p;      // описывает указатель
i = *p;      // использует указываемый объект
```

Вся сложность понимания записи производных типов проистекает из того, что операции * и & префиксные, а операции [] и () постфиксные, поэтому для формулировки типов в тех случаях, когда приоритеты операций создают затруднения, надо использовать скобки. Например, поскольку приоритет у [] выше, чем у *, то

```
int* v[10];    // вектор указателей
int (*p)[10];  // указатель на вектор
```

Большинство людей просто помнят, как выглядят наиболее обычные типы.

Описание каждого имени, вводимого в программе, может оказаться утомительным, особенно если их типы одинаковы. Но можно описывать в одном описании несколько имен. В этом случае описание содержит вместо одного имени список имен, разделенных запятыми. Например, два имени можно описать так:

```
int x, y;    // int x; int y;
```

При описании производных типов можно указать, что операции применяются только к отдельным именам (а не ко всем остальным именам в этом описании). Например:

```
int* p, y;    // int* p; int y; НЕ int* y;
int x, *p;    // int x; int* p;
```

```
int v[10], *p;    // int v[10]; int* p;
```

Мнение автора таково, что подобные конструкции делают программу менее удобочитаемой и их следует избегать.

2.3.4. Тип void

Тип void (пустой) синтаксически ведет себя как основной тип. Однако использовать его можно только как часть производного типа, объектов типа void не существует. Он используется для того, чтобы указать, что функция не возвращает значения, или как базовый тип для указателей на объекты неизвестного типа.

```
void f()          // f не возвращает значение
void* pv;         // указатель на объект неизвестно-
го типа
```

Переменной типа void* можно присваивать указатель любого типа. На первый взгляд это может показаться не особенно полезным, поскольку void* нельзя разыменовывать, но именно это ограничение и делает тип void* полезным. Главным образом, он применяется для передачи указателей функциям, которые не позволяют сделать предположение о типе объекта, и для возврата из функций нетипизированных объектов. Чтобы использовать такой объект, необходимо применить явное преобразование типа. Подобные функции обычно находятся на самом нижнем уровне системы, там, где осуществляется работа с основными аппаратными ресурсами. Например:

```
void* allocate(int size);    // выделить
void deallocate(void*);      // освободить

f() {
    int* pi = (int*)allocate(10*sizeof(int));
    char* pc = (char*)allocate(10);
    //...
    deallocate(pi);
    deallocate(pc);
}
```

2.3.5. Указатели

Для большинства типов T T* является типом указатель на T. То есть в переменной типа T* может храниться адрес объекта типа T. Для

указателей на векторы и указателей на функции вам, к сожалению, придется пользоваться более сложной записью:

```
int* pi;
char** cpp;           // указатель на указатель на char
int (*vp)[10];        // указатель на вектор из 10 int'ов
int (*fp)(char, char*); // указатель на функцию, получающую параметры (char, char*)
                        // и возвращающую int
```

Основная операция над указателем - разыменование, то есть ссылка на объект, на который указывает указатель. Эта операция также называется косвенным обращением. Операция разыменования - это унарное * (префиксное). Например:

```
char c1 = 'a';
char* p = &c1;    // в p хранится адрес c1
char c2 = *p;     // c2 = 'a'
```

Переменная, на которую указывает p, - это c1, а значение, которое хранится в c1, - это 'a', поэтому присваиваемое c2 значение *p есть 'a'.

Над указателями можно осуществлять некоторые арифметические действия. Вот, например, функция, подсчитывающая число символов в строке (не считая завершающего 0):

```
int strlen(char* p)
{
    int i = 0;
    while (*p++) i++;
    return i;
}
```

Другой способ найти длину состоит в том, чтобы сначала найти конец строки, а затем вычесть адрес начала строки из адреса ее конца:

```
int strlen(char* p)
{
    char* q = p;
    while (*q++) ;
    return q-p-1;
}
```

2.3.6. Векторы

Для типа `T T[size]` является типом «вектор из `size` элементов типа `T`». Элементы индексируются (нумеруются) от 0 до `size-1`. Например:

```
float v[3];    // вектор из трех float: v[0],
v[1], v[2]
int a[2][5];  // два вектора из пяти int
char* vpc;    // вектор из 32 указателей на
СИМВОЛ
```

Цикл для печати целых значений букв нижнего регистра можно было бы написать так:

```
extern int strlen(char*);

char alpha[] = «abcdefghijklmnopqrstuvwxyz»;

main()
{
    int sz = strlen(alpha);

    for (int i=0; i...
```

Функция `strlen()` использовалась для подсчета числа символов в `alpha`; вместо этого можно было использовать значение размера `alpha`. Если применяется набор символов ASCII, то выдача выглядит так:

```
'a' = 97 = 0141 = 0x61
'b' = 98 = 0142 = 0x62
'c' = 99 = 0143 = 0x63
...
```

Заметим, что задавать размер вектора `alpha` необязательно; компилятор считает число символов в символьной строке, указанной в качестве инициализатора. Использование строки как инициализатора для вектора символов - удобное, но, к сожалению, и единственное применение строк. Аналогичное этому присваивание строки вектору отсутствует. Например:

```
char v[9];
v = «строка»;           // ошибка
```

ошибочно, поскольку присваивание не определено для векторов.

Конечно, для инициализации символьных массивов подходят не только строки. Для остальных типов нужно применять более сложную

запись. Эту запись можно использовать и для символьных векторов. Например:

```
int v1[] = { 1, 2, 3, 4 };
int v2[] = { 'a', 'b', 'c', 'd' };
char v3[] = { 1, 2, 3, 4 };
char v4[] = { 'a', 'b', 'c', 'd' };
```

Заметьте, что v4 - вектор из четырех (а не пяти) символов; он не оканчивается нулем, как того требуют соглашение и библиотечные подпрограммы. Обычно применение такой записи ограничивается статическими объектами.

Многомерные массивы представляются как векторы векторов, и применение записи через запятую, как это делается в некоторых других языках, дает ошибку при компиляции, так как запятая (,) является операцией последования. Попробуйте, например, сделать так:

```
int bad[5,2]; // ошибка
```

и так:

```
int v[5][2];
int bad = v[4,1]; // ошибка
int good = v[4][1]; // ошибка
```

Описание

```
char v[2][5];
```

описывает вектор из двух элементов, каждый из которых является вектором типа char[5]. В следующем примере первый из этих векторов инициализируется первыми пятью буквами, а второй - первыми пятью цифрами.

```
char v[2][5] = {
    'a', 'b', 'c', 'd', 'e',
    '0', '1', '2', '3', '4'
}

main() {
    for (int i = 0; i<2; i++) {
        for (int j = 0; j<5; j++)
            cout << «v[» << i << «][» << j
                << «]=» << chr(v[i][j]) << « »;
        cout << «\n»;
    }
}
```


это дает в результате

```
v[0][0]=a v[0][1]=b v[0][2]=c v[0][3]=d v[0][4]=e
v[1][0]=0 v[1][1]=1 v[1][2]=2 v[1][3]=3 v[1][4]=4
```

2.3.7. Указатели и векторы

Указатели и векторы в C++ связаны очень тесно. Имя вектора можно использовать как указатель на его первый элемент, поэтому пример с алфавитом можно было написать так:

```
char alpha[] = «abcdefghijklmnopqrstuvwxyz»;
char* p = alpha;
char ch;
```

```
while (ch = *p++)
    cout << chr(ch) << « = » << ch
        << « = 0» << oct(ch) << «\n»;
```

Описание `p` можно было также записать как

```
char* p = &alpha[0];
```

Эта эквивалентность широко используется в вызовах функций, в которых векторный параметр всегда передается как указатель на первый элемент вектора; так, в примере

```
extern int strlen(char*);
char v[] = «Annemarie»;
char* p = v;
strlen(p);
strlen(v);
```

функции `strlen` в обоих вызовах передается одно и то же значение. Вся штука в том, что этого невозможно избежать; то есть не существует способа описать функцию так, чтобы вектор `v` в вызове функции копировался.

Результат применения к указателям арифметических операций `+`, `-`, `++` или `--` зависит от типа объекта, на который они указывают. Когда к указателю `p` типа `T*` применяется арифметическая операция, предполагается, что `p` указывает на элемент вектора объектов типа `T`; `p+1` означает следующий элемент этого вектора, а `p-1` - предыдущий элемент. Отсюда следует, что значение `p+1` будет на `sizeof(T)` больше значения `p`. Например, выполнение

```
main()
{
    char cv[10];
```

```

int iv[10];

char* pc = cv;
int* pi = iv;

cout << «char* « << long(pc+1)-long(pc) << «\n»;
cout << «int* « << long(ic+1)-long(ic) << «\n»;
}

```

дает

```

char* 1
int* 4

```

поскольку на моей машине каждый символ занимает один байт, а каждое целое занимает четыре байта. Перед вычитанием значения указателей преобразовывались к типу `long` с помощью явного преобразования типа. Они преобразовывались к `long`, а не к «очевидному» `int`, поскольку есть машины, на которых указатель не влезет в `int` (то есть, `sizeof(int)`).

2.3.8. Структуры

Вектор есть совокупность элементов одного типа; `struct` является совокупностью элементов (практически) произвольных типов. Например:

```

struct address {           // почтовый адрес
char* name;               // имя «Jim Dandy»
long number;              // номер дома 61
char* street;             // улица «South Street»
char* town;               // город «New Providence»
char* state[2];           // штат 'N' 'J'
int zip;                  // индекс 7974
};

```

определяет новый тип, названный `address` (почтовый адрес), состоящий из пунктов, требующихся для того, чтобы послать кому-нибудь корреспонденцию (вообще говоря, `address` не является достаточным для работы с полным почтовым адресом, но в качестве примера достаточен). Обратите внимание на точку с запятой в конце; это одно из очень немногих мест в C++, где необходимо ставить точку с запятой после фигурной скобки, поэтому люди склонны забывать об этом.

Переменные типа `address` могут описываться точно так же, как другие переменные, а доступ к отдельным членам получается с помощью операции `.` (точка). Например:

```
address jd;
jd.name = «Jim Dandy»;
jd.number = 61;
```

Запись, которая использовалась для инициализации векторов, можно применять и к переменным структурных типов. Например:

```
address jd = {
    «Jim Dandy»,
    61, «South Street»,
    «New Providence», {'N', 'J'}, 7974
};
```

Однако обычно лучше использовать конструктор. Заметьте, что нельзя было бы инициализировать `jd.state` строкой «NJ». Строки оканчиваются символом `'\0'`, поэтому в «NJ» три символа, то есть на один больше, чем поместится в `jd.state`.

К структурным объектам часто обращаются посредством указателей, используя операцию `->`. Например:

```
void print_addr(address* p)
{
    cout << p->name << «\n»
         << p->number << « » << p->street << «\n»
         << p->town << «\n»
         << chr(p->state[0]) << chr(p->state[1])
         << « » << p->zip << «\n»;
}
```

Объекты типа структур можно присваивать, передавать как параметры функции и возвращать из функции в качестве результата. Например:

```
address current;

address set_current(address next)
{
    address prev = current;
    current = next;
    return prev;
}
```

Остальные осмысленные операции, такие как сравнение (`==` и `!=`), не определены. Однако пользователь может определить эти операции (см. Главу 6).

Размер объекта структурного типа нельзя вычислить просто как сумму его членов. Причина этого состоит в том, что многие машины требуют, чтобы объекты определенных типов выравнивались в памяти только по некоторым зависящим от архитектуры границам (типичный пример: целое должно быть выровнено по границе слова), или просто гораздо более эффективно обрабатывают такие объекты, если они выровнены в машине. Это приводит к «дырам» в структуре. Например, (на моей машине) `sizeof (address)` равен 24, а не 22, как можно было ожидать.

Заметьте, что имя типа становится доступным сразу после того, как оно встретилось, а не только после того, как полностью просмотрено все описание. Например:

```
struct link{
    link* previous;
    link* successor;
}
```

Новые объекты структурного типа не могут описываться, пока все описание не просмотрено, поэтому

```
struct no_good {
    no_good member;
};
```

является ошибочным (компилятор не может установить размер `no_good`). Чтобы дать возможность двум (или более) структурным типам ссылаться друг на друга, можно просто описать имя как имя структурного типа. Например:

```
struct list;          // должна быть определена
позднее
```

```
struct link {
    link* pre;
    link* suc;
    link* member_of;
};

struct list {
    link* head;
}
```

Без первого описания `list` описание `link` вызвало бы к синтаксическую ошибку.

2.3.9. Эквивалентность типов

Два структурных типа являются различными, даже когда они имеют одни и те же члены. Например:

```
struct s1 { int a; };
struct s2 { int a; };
```

есть два разных типа, поэтому

```
s1 x;
s2 y = x;    // ошибка: несоответствие типов
```

Структурные типы отличны также от основных типов, поэтому

```
s1 x;
int i = x;    // ошибка: несоответствие типов
```

Однако существует механизм для описания нового имени для типа без введения нового типа. Описание с префиксом `typedef` описывает не новую переменную данного типа, а новое имя этого типа. Например:

```
typedef char* Pchar;
Pchar p1, p2;
char* p3 = p1;
```

Это может служить удобной сокращенной записью.

2.3.10. Ссылки

Ссылка является другим именем объекта. Главное применение ссылок состоит в спецификации операций для типов, определяемых пользователем; они обсуждаются в Главе 6. Они могут также быть полезны в качестве параметров функции. Запись `x&` означает: ссылка на `x`. Например:

```
int i = 1;
int& r = i;    // r и i теперь ссылаются на один int
int x = r      // x = 1
r = 2;         // i = 2;
```

Ссылка должна быть инициализирована (должно быть что-то, для чего она является именем). Заметьте, что инициализация ссылки есть нечто совершенно отличное от присваивания ей.

Вопреки ожиданиям, ни одна операция на ссылку не действует. Например,

```
int ii = 0;
int& rr = ii;
rr++;           // ii увеличивается на 1
```

допустимо, но `rr++` не увеличивает ссылку; вместо этого `++` применяется к `int`, которым оказывается `ii`. Следовательно, после инициализации значение ссылки не может быть изменено; она всегда ссылается на объект, который ей было дано обозначать (денотировать) при инициализации. Чтобы получить указатель на объект, денотируемый ссылкой `rr`, можно написать `&rr`.

Очевидным способом реализации ссылки является константный указатель, который разыменовывается при каждом использовании. Это делает инициализацию ссылки тривиальной, когда инициализатор является `lvalue` (объектом, адрес которого вы можете взять). Однако инициализатор для `&T` не обязательно должен быть `lvalue` и даже не должен быть типа `T`. В таких случаях:

- [1] во-первых, если необходимо, применяются преобразования типа;
- [2] затем полученное значение помещается во временную переменную; и
- [3] наконец, ее адрес используется в качестве значения инициализатора.

Рассмотрим описание

```
double& dr = 1;
```

Это интерпретируется так:

```
double* drp;           // ссылка, представленная как
указатель
double temp;
temp = double(1);
drp = &temp;
```

Ссылку можно использовать для реализации функции, которая, как предполагается, изменяет значение своего параметра.

```
int x = 1;
void incr(int& aa) { aa++; }
incr(x)                // x = 2
```

По определению, семантика передачи параметра та же, что семантика инициализации, поэтому параметр `aa` функции `incr` становится другим именем для `x`. Однако, чтобы сделать программу читаемой, в большинстве случаев лучше всего избегать функций, которые изменяют значение своих параметров. Часто предпочтительно явно возвращать значение из функции или требовать в качестве параметра указатель:

```
int x = 1;
int next(int p) { return p+1; }
x = next(x);           // x = 2

void inc(int* p) { (*p)++; }
inc(&x);                // x = 3
```

Ссылки также можно применять для определения функций, которые могут использоваться и в левой, и в правой части присваивания. Опять, большая часть наиболее интересных случаев этого обнаруживается в проектировании нетривиальных определяемых пользователем типов. Для примера давайте определим простой ассоциативный массив. Вначале мы определим структуру пары следующим образом:

```
struct pair {
    char* name;
    int val;
};
```

Основная идея состоит в том, что строка имеет ассоциированное с ней целое значение. Легко определить функцию поиска `find()`, которая поддерживает структуру данных, состоящую из одного `pair` для каждой отличной от других строки, которая была ей представлена. Для краткости представления используется очень простая (и неэффективная) реализация:

```
const large = 1024;
static pair vec[large+1];

pair* find(char* p)
/*
    поддерживает множество пар «pair»:
    ищет p, если находит, возвращает его «pair»,
    иначе возвращает неиспользованную «pair»
*/
{
    for (int i=0; vec[i].name; i++)
```

```

        if (strcmp(p,vec[i].name)==0) return &vec[i];

        if (i == large) return &vec[large-1];

        return &vec[i];
    }

```

Эту функцию может использовать функция value(), реализующая массив целых, индексированный символьными строками (вместо обычного способа):

```

int& value(char* p)
{
    pair* res = find(p);
    if (res->name == 0) {           // до сих пор
не встречалось:
        res->name = new
char[strlen(p)+1]; // инициализировать
        strcpy(res->name,p);
        res->val = 0;              // начальное
значение 0
    }
    return res->val;
}

```

Для данной в качестве параметра строки value() находит целый объект (а не значение соответствующего целого); после чего она возвращает ссылку на него. Ее можно использовать, например, так:

```

const MAX = 256; // больше самого большого
слова

```

```

main()
// подсчитывает число вхождений каждого слова
во вводе
{
    char buf[MAX];

    while (cin>>buf) value(buf)++;

    for (int i=0; vec[i].name; i++)
        cout << vec[i].name << «: « << vec
[i].val << «\n»;
}

```


На каждом проходе цикл считывает одно слово из стандартной строки ввода `cin` в `buf` (см. Главу 8), а затем обновляет связанный с ней счетчик с помощью `find()`. И, наконец, печатается полученная таблица различных слов во введенном тексте, каждое с числом его встречаемости. Например, если вводится

```
aa bb bb aa aa bb aa aa
```

то программа выдаст:

```
aa: 5
```

```
bb: 3
```

Легко усовершенствовать это в плане собственного типа ассоциированного массива с помощью класса с перегруженной операцией выбора `[]`.

2.4. Константы

C++ дает возможность записи значений основных типов: символьных констант, целых констант и констант с плавающей точкой. Кроме того, ноль (0) может использоваться как константа любого указательного типа, и символьные строки являются константами типа `char[]`. Можно также задавать символические константы. Символическая константа - это имя, значение которого не может быть изменено в его области видимости. В C++ имеется три вида символических констант: 1) любому значению любого типа можно дать имя и использовать его как константу, добавив к его описанию ключевое слово `const`; 2) множество целых констант может быть определено как перечисление; и 3) любое имя вектора или функции является константой.

2.4.1. Целые константы

Целые константы предстают в четырех обличьях: десятичные, восьмеричные, шестнадцатеричные и символьные константы. Десятичные используются чаще всего и выглядят так, как можно было бы ожидать:

```
0      1234      976      12345678901234567890
```

Десятичная константа имеет тип `int`, при условии, что она влезает в `int`, в противном случае ее тип `long`. Компилятор должен предупреждать о константах, которые слишком длинны для представления в машине.

Константа, которая начинается нулем, за которым идет x ($0x$), является шестнадцатеричным числом (с основанием 16), а константа, которая начинается нулем за которым идет цифра, является восьмеричным числом (с основанием 8). Вот примеры восьмеричных констант:

0 02 077 0123

их десятичные эквиваленты - это 0, 2, 63, 83. В шестнадцатеричной записи эти константы выглядят так:

0x0 0x2 0x3f 0x53

Буквы a, b, c, d, e и f, или их эквиваленты в верхнем регистре, используются для представления чисел 10, 11, 12, 13, 14 и 15, соответственно. Восьмеричная и шестнадцатеричная записи наиболее полезны для записи набора битов; применение этих записей для выражения обычных чисел может привести к неожиданностям. Например, на машине, где `int` представляется как двоичное дополнительное шестнадцатеричное целое, `0xffff` является отрицательным десятичным числом -1; если бы для представления целого использовалось большее число битов, то оно было бы числом 65535.

2.4.2. Константы с плавающей точкой

Константы с плавающей точкой имеют тип `double`. Как и в предыдущем случае, компилятор должен предупреждать о константах с плавающей точкой, которые слишком велики, чтобы их можно было представить. Вот некоторые константы с плавающей точкой:

1.23 .23 0.23 1. 1.0 1.2e10 1.23e-15

Заметьте, что в середине константы с плавающей точкой не может встречаться пробел. Например, `65.43 e-21` является не константой с плавающей точкой, а четырьмя отдельными лексическими символами (лексемами):

65.43 e - 21

и вызовет синтаксическую ошибку.

Если вы хотите иметь константу с плавающей точкой типа `float`, вы можете определить ее так:

```
const float pi = 3.14159265;
```

2.4.3. Символьные константы

Хотя в C++ и нет отдельного символьного типа данных, точнее, символ может храниться в целом типе, в нем для символов имеется специальная и удобная запись. Символьная константа - это символ, заключенный в одинарные кавычки; например, 'a' или '0'. Такие символьные константы в действительности являются символическими константами для целого значения символов в наборе символов той машины, на которой будет выполняться программа (который не обязательно совпадает с набором символов, применяемом на том компьютере, где программа компилируется). Поэтому, если вы выполняете на машине, использующей набор символов ASCII, то значением '0' будет 48, но если ваша машина использует EBCDIC, то оно будет 240. Употребление символьных констант вместо десятичной записи делает программу более переносимой. Несколько символов также имеют стандартные имена, в которых обратная косая \ используется как escape-символ:

- 'b' возврат назад ;
- 'f' перевод формата;
- 'n' новая строка;
- 'r' возврат каретки;
- 't' горизонтальная табуляция;
- 'v' вертикальная табуляция;
- '\ ' обратная косая (обратный слэш);
- '" ' одинарная кавычка;
- '\>' двойная кавычка;
- '\0' null, пустой символ, целое значение 0.

Вопреки их внешнему виду, каждое является одним символом. Можно также представлять символ одно-, дву- или трехзначным восьмеричным числом (символ \, за которым идут восьмеричные цифры), или одно-, дву- или трехзначным шестнадцатеричным числом (x, за которым идут шестнадцатеричные цифры). Например:

'\6'	'\x6'	6	ASCII ack
'\60'	'\x30'	48	ASCII '0'
'\137'	'\x05f'	95	ASCII ' _ '

Это позволяет представлять каждый символ из машинного набора символов, и в частности вставлять такие символы в символьные строки (см. следующий раздел). Применение числовой записи для симво-

лов делает программу переносимой между машинами с различными наборами символов.

2.4.4. Строки

Строковая константа - это последовательность символов, заключенная в двойные кавычки:

«это строка»

Каждая строковая константа содержит на один символ больше, чем кажется; все они заканчиваются пустым символом '\0' со значением 0. Например:

```
sizeof («asdf») == 5;
```

Строка имеет тип «вектор из соответствующего числа символов», поэтому «asdf» имеет тип char[5]. Пустая строка записывается «» (и имеет тип char[1]). Заметьте, что для каждой строки s strlen(s)==sizeof(s)-1, поскольку strlen() не учитывает завершающий 0.

Соглашение о представлении неграфических символов с обратной косой можно использовать также и внутри строки. Это дает возможность представлять в строке двойные кавычки и escape-символ \. Самым обычным символом этого рода является, безусловно, символ новой строки '\n'. Например:

```
cout << «грудок в конце сообщения\007\n»
```

где 7 - значение ASCII символа bel (звонок).

В строке невозможно иметь «настоящую» новую строку:

```
«это не строка,  
а синтаксическая ошибка»
```

Однако в строке может стоять обратная косая, сразу после которой идет новая строка; и то, и другое будет проигнорировано. Например:

```
cout << «здесь все \  
ok»
```

напечатает

```
здесь все ok
```

Новая строка, перед которой идет escape (обратная косая), не приводит к появлению в строке новой строки, это просто договоренность о записи.

В строке можно иметь пустой символ, но большинство программ не будет предполагать, что есть символы после него. Например, строка

«asdf000hjk!» будет рассматриваться стандартными функциями, вроде `strcpy()` и `strlen()`, как «asdf».

Вставляя численную константу в строку с помощью восьмеричной или шестнадцатеричной записи благоразумно всегда использовать число из трех цифр. Читать запись достаточно трудно и без необходимости беспокоиться о том, является ли символ после константы цифрой или нет. Разберите эти примеры:

```
char v1[] = «a\x0fah\0129»;      // 'a' '\xfa'
'h' '\12' '9'
char v2[] = «a\xfah\129»;        // 'a' '\xfa'
'h' '\12' '9'
char v3[] = «a\xfad\127»;        // 'a' '\xfad'
'\127'
```

Имейте в виду, что двузначной шестнадцатеричной записи на машинах с 9-битовым байтом будет недостаточно.

2.4.5. Ноль

Ноль (0) можно употреблять как константу любого целого, плавающего или указательного типа. Никакой объект не размещается по адресу 0. Тип нуля определяется контекстом. Обычно (но не обязательно) он представляется набором битов все-нули соответствующей длины.

2.4.6. Const

Ключевое слово `const` может добавляться к описанию объекта, чтобы сделать этот объект константой, а не переменной. Например:

```
const int model = 145;
const int v[] = { 1, 2, 3, 4 };
```

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

```
model = 145;           // ошибка
model++;              // ошибка
```

Заметьте, что `const` изменяет тип, то есть ограничивает способ использования объекта, вместо того, чтобы задавать способ размещения константы. Поэтому, например, вполне разумно, а иногда и полезно, описывать функцию как возвращающую `const`:

```
const char* peek(int i)
{
    return private[i];
}
```

Функцию вроде этой можно было бы использовать для того, чтобы давать кому-нибудь читать строку, которая не может быть затерта или переписана (этим кем-то).

С другой стороны, компилятор может несколькими путями воспользоваться тем, что объект является константой (конечно, в зависимости от того, насколько он сообразителен). Самое очевидное - это то, что для константы не требуется выделять память, поскольку компилятор знает ее значение. Кроме того, инициализатор константы часто (но не всегда) является константным выражением, то есть он может быть вычислен на стадии компиляции. Однако для вектора констант обычно приходится выделять память, поскольку компилятор в общем случае не может вычислить, на какие элементы вектора сделаны ссылки в выражениях. Однако на многих машинах даже в этом случае может достигаться повышение эффективности путем размещения векторов констант в память, доступную только для чтения.

Использование указателя вовлекает два объекта: сам указатель и указываемый объект. Снабжение описания указателя «префиксом» `const` делает объект, но не сам указатель, константой. Например:

```
const char* pc = «asdf»;           // указатель на
константу
pc[3] = 'a';                       // ошибка
pc = «ghjk»;                       // ok
```

Чтобы описать сам указатель, а не указываемый объект, как константный, используется операция `const*`. Например:

```
char *const cp = «asdf»;           // константный
указатель
cp[3] = 'a';                       // ok
cp = «ghjk»;                       // ошибка
```

Чтобы сделать константами оба объекта, их оба нужно описать `const`. Например:

```
const char *const cpc = «asdf»;     // const
указатель на const
cpc[3] = 'a';                       // ошибка
cpc = «ghjk»;                       // ошибка
```

Объект, являющийся константой при доступе к нему через один указатель, может быть переменной, когда доступ осуществляется другими путями. Это, в частности, полезно для параметров функции. Посредством описания параметра указателя как `const` функции запрещается изменять объект, на который он указывает. Например:

```
char* strcpy(char* p, const char* q); // не
может изменить q
```

Указателю на константу можно присваивать адрес переменной, поскольку никакого вреда от этого быть не может. Однако нельзя присвоить адрес константы указателю, на который не было наложено ограничение, поскольку это позволило бы изменить значение объекта. Например:

```
int a = 1;
const c = 2;
const* p1 = &c;    // ok
const* p2 = &a;    // ok
int* p3 = &c;      // ошибка
*p3 = 7;           // меняет значение c
```

Как обычно, если тип в описании опущен, то он предполагается `int`.

2.4.7. Перечисления

Есть другой метод определения целых констант, который иногда более удобен, чем применение `const`. Например:

```
enum { ASM, AUTO, BREAK };
```

определяет три целых константы, называемые перечислителями, и присваивает им значения. Поскольку значения перечислителей по умолчанию присваиваются начиная с 0 в порядке возрастания, это эквивалентно записи:

```
const ASM = 0;
const AUTO = 1;
const BREAK = 2;
```

Перечисление может быть именованным. Например:

```
enum keyword { ASM, AUTO, BREAK };
```

Имя перечисления становится синонимом `int`, а не новым типом. Описание переменной `keyword`, а не просто `int`, может дать как про-

граммисту, так и компилятору подсказку о том, что использование преднамеренное. Например:

```
keyword key;

switch (key) {
case ASM:
    // что-то делает
    break;
case BREAK:
    // что-то делает
    break;
}
```

побуждает компилятор выдать предупреждение, поскольку только два значения keyword из трех используются.

Можно также задавать значения перечислителей явно. Например:

```
enum int16 {
    sign=0100000,           // знак
    most_significant=040000, // самый зна-
чимый
    least_significant=1      // наименее
значимый
};
```

Такие значения не обязательно должны быть различными, возрастающими или положительными.

2.5. Экономия пространства

В ходе программирования нетривиальных разработок неизбежно наступает время, когда хочется иметь больше пространства памяти, чем имеется или отпущено. Есть два способа выжать побольше пространства из того, что доступно:

- [1] помещение в байт более одного небольшого объекта; и
- [2] использование одного и того же пространства для хранения разных объектов в разное время.

Первого можно достичь с помощью использования полей, второго - через использование объединений. Эти конструкции описываются в следующих разделах. Поскольку обычное их применение состоит чисто в оптимизации программы и они в большинстве случаев не переносимы, программисту следует дважды подумать, прежде чем использовать их. Часто лучше изменить способ управления данными, например,

больше полагаться на динамически выделяемую память и меньше на заранее выделенную статическую память.

2.5.1. Поля

Использование `char` для представления двоичной переменной, например, переключателя включено/выключено, может показаться экстравагантным, но `char` является наименьшим объектом, который в C++ может выделяться независимо. Можно, однако, сгруппировать несколько таких крошечных переменных вместе в виде полей `struct`. Член определяется как поле путем указания после его имени числа битов, которые он занимает. Допустимы неименованные поля; они не влияют на смысл именованных полей, но неким машинно-зависимым образом могут улучшить размещение:

```
struct sreg {
    unsigned enable : 1;
    unsigned page : 3;
    unsigned : 1;      // неиспользуемое
    unsigned mode : 2;
    unsigned : 4;      // неиспользуемое
    unsigned access : 1;
    unsigned length : 1;
    unsigned non_resident : 1;
}
```

Получилось размещение регистра 0 состояния DEC PDP11/45 (в предположении, что поля в слове размещаются слева направо). Этот пример также иллюстрирует другое основное применение полей: именовать части внешне предписанного размещения. Поле должно быть целого типа и должно использоваться как другие целые, за исключением того, что невозможно взять адрес поля. В ядре операционной системы или в отладчике тип `sreg` можно было бы использовать так:

```
sreg* sr0 = (sreg*)0777572;
//...
if (sr->access) {          // нарушение доступа
    // чистит массив
    sr->access = 0;
}
```

Однако применение полей для упаковки нескольких переменных в один байт не обязательно экономит пространство. Оно экономит пространство, занимаемое данными, но объем кода, необходимого для

манипуляции этими переменными, на большинстве машин возрастает. Известны программы, которые значительно сжимались, когда двоичные переменные преобразовывались из полей бит в символы! Кроме того, доступ к `char` или `int` обычно намного быстрее, чем доступ к полю. Поля - это просто удобная и краткая запись для применения логических операций с целью извлечения информации из части слова или введения информации в нее.

2.5.2. Объединения

Рассмотрим проектирование символьной таблицы, в которой каждый элемент содержит имя и значение, и значение может быть либо строкой, либо целым:

```

    struct entry {
        char* name;
        char  type;
        char* string_value;           // используется
если type == 's'
        int   int_value;             // используется
если type == 'i'
    };

    void print_entry(entry* p)
    {
        switch p->type {
        case 's':
            cout << p->string_value;
            break;
        case 'i':
            cout << p->int_value;
            break;
        default:
            cerr << «испорчен type\n»;
            break;
        }
    }

```

Поскольку `string_value` и `int_value` никогда не могут использоваться одновременно, ясно, что пространство пропадает впустую. Это можно легко исправить, указав, что оба они должны быть членами `union` (объединения); например, так:

```

    struct entry {

```

```

        char* name;
        char type;
        union {
            char* string_value;    // используется
если type == 's'
            int int_value;        // используется
если type == 'i'
        };
    };

```

Это оставляет всю часть программы, использующую entry, без изменений, но обеспечивает, что при размещении entry string_value и int_value имеют один и тот же адрес. Отсюда следует, что все члены объединения вместе занимают лишь столько памяти, сколько занимает наибольший член.

Использование объединений таким образом, чтобы при чтении значения всегда применялся тот член, с применением которого оно записывалось, совершенно оптимально. Но в больших программах непросто гарантировать, что объединения используются только таким образом, и из-за неправильного использования могут появляться трудно уловимые ошибки. Можно капсулизовать объединение таким образом, чтобы соответствие между полем типа и типами членов было гарантированно правильным.

Объединения иногда используют для «преобразования типов» (это делают главным образом программисты, воспитанные на языках, не обладающих средствами преобразования типов, где жульничество является необходимым). Например, это «преобразует» на VAX'e int в int*, просто предполагая побитовую эквивалентность:

```

struct fudge {
    union {
        int i;
        int* p;
    };
};

fudge a;
a.i = 4096;
int* p = a.p;    // плохое использование

```

Но на самом деле это совсем не преобразование: на некоторых машинах int и int* занимают неодинаковое количество памяти, а на других никакое целое не может иметь нечетный адрес. Такое приме-

ние объединений непереносимо, а есть явный способ указать преобразование типа.

Изредка объединения умышленно применяют, чтобы избежать преобразования типов. Можно, например, использовать `fudge`, чтобы узнать представление указателя 0:

```
fudge.p = 0;  
int i = fudge.i; // i не обязательно должно быть 0
```

Можно также дать объединению имя, то есть сделать его полноправным типом. Например, `fudge` можно было бы описать так:

```
union fudge {  
    int i;  
    int* p;  
};
```

и использовать (неправильно) в точности, как раньше. Имеются также и оправданные применения именованных объединений.

3. ГЛАВА ТРЕТЬЯ

C++ имеет небольшой, но гибкий набор различных видов операторов для контроля потока управления в программе и богатый набор операций для манипуляции данными. В этой главе представлена краткая сводка операций, а в конце обсуждаются стиль выравнивания и комментарии.

3.1. Краткая сводка операций

Здесь же приводится краткая сводка и некоторые примеры. После каждой операции приведено одно или более ее общеупотребительных названий и пример ее использования. В этих примерах имя_класса - это имя класса, член - имя члена, объект - выражение, дающее в результате объект класса, указатель - выражение, дающее в результате указатель, выр - выражение, а lvalue - выражение, денотирующее неконстантный объект. Тип может быть совершенно произвольным именем типа (со *, () и т.п.) только когда он стоит в скобках, во всех остальных случаях существуют ограничения.

Унарные операции и операции присваивания правоассоциативны, все остальные левоассоциативны. Это значит, что $a=b=c$ означает $a=(b=c)$, $a+b+c$ означает $(a+b)+c$, и $*p++$ означает $*(p++)$, а не $(*p)++$.

В каждой отчерченной части находятся операции с одинаковым приоритетом. Операция имеет приоритет больше, чем операции из частей, расположенных ниже. Например: $a+b*c$ означает $a+(b*c)$, так как * имеет приоритет выше, чем +, а $a+b-c$ означает $(a+b)-c$, поскольку + и - имеют одинаковый приоритет (и поскольку + левоассоциативен).

Сводка операций

№	Операция	Название	Формат
1	:: ::	разрешение области видимости Глобальное	имя_класса :: член :: глобальное имя
2	-> [] () () sizeof sizeof	выбор члена индексация вызов функции построение значения размер объекта размер типа	указатель->член указатель [выр] выр (список_выр) тип (список_выр) sizeof выр sizeof (тип)

№	Операция	Название	Формат
3	++ ++ -- -- ~ ! - + & * new delete de lete[] ()	приращение после приращение до уменьшение после уменьшение до дополнение не унарный минус унарный плюс адрес объекта разыменование создание (размещение) уничтожение (освобождение) уничтожение (освобождение) век- тора приведение (преобразование типа)	lvalue++ ++lvalue lvalue— —lvalue ~ выр ! выр - выр + выр & lvalue выр new тип delete указатель delete[выр] ука- затель (тип) выр
4	* / %	Умножение деление взятие по модулю (остаток)	выр * выр выр / выр выр % выр
5	+ -	сложение (плюс) вычитание (минус)	выр + выр выр – выр
6	<< >>	сдвиг влево сдвиг вправо	lvalue << выр lvalue >> выр
7	< <= > >=	< меньше меньше или равно больше больше или равно	выр < выр выр <= выр выр > выр выр >= выр
8	== !=	равно не равно	выр == выр выр != выр
9	&	& побитовое И	выр & выр
10	^	побитовое исключающее ИЛИ	выр ^ выр
11		побитовое включающее ИЛИ	выр выр
12	&&	логическое И	выр && выр
13		логическое включающее ИЛИ	выр выр
14	? :	арифметический if	выр ? выр : выр
15	= *= /= %=	простое присваивание умножить и присвоить разделить и присвоить взять по модулю и присвоить	lvalue = выр lvalue = выр lvalue /= выр lvalue %= выр

№	Операция	Название	Формат
	+= -= <<= >>= &= = ^=	сложить и присвоить вычесть и присвоить сдвинуть влево и присвоить сдвинуть вправо и присвоить И и присвоить включающее ИЛИ и присвоить исключающее ИЛИ и присвоить	lvalue += вып lvalue -= вып lvalue <<= вып lvalue >>= вып lvalue &= вып lvalue = вып lvalue ^= вып
16	,	запятая (последование)	выр , выр

3.1.1. Круглые скобки

Скобками синтаксис C++ злоупотребляет; количество способов их использования приводит в замешательство: они применяются для заключения в них параметров в вызовах функций, в них заключается тип в преобразовании типа (приведении к типу), в именах типов для обозначения функций, а также для разрешения конфликтов приоритетов. К счастью, последнее требуется не слишком часто, потому что уровни приоритета и правила ассоциативности определены таким образом, чтобы выражения «работали ожидаемым образом» (то есть, отражали наиболее привычный способ употребления). Например, значение

```
if (i<=0 || max
```

3.1.2. Порядок вычисления

Порядок вычисления подвыражений в выражении не определен. Например

```
int i = 1;
v[i] = i++;
```

может вычисляться или как $v[1]=1$, или как $v[2]=1$. При отсутствии ограничений на порядок вычисления выражения может генерироваться более хороший код. Было бы замечательно, если бы компилятор предупреждал о подобных неоднозначностях, но большинство компиляторов этого не делают.

Относительно операций

```
,      &&      ||
```

гарантируется, что их левый операнд вычисляется раньше, чем правый. Например, $b=(a=2,a=1)$ присвоит b 3. Заметьте, что операция последования $,$ (запятая) логически отличается от запятой, которая используется для разделения параметров в вызове функции. Рассмотрим

```
f1(v[i], i++);           // два параметра
f2( (v[i], i++) )       // один параметр
```

В вызове `f1` два параметра, `v[i]` и `i++`, и порядок вычисления выражений-параметров не определен. Зависимость выражения-параметра от порядка вычисления - это очень плохой стиль, а также непереносимо. В вызове `f2` один параметр, выражение с запятой, которое эквивалентно `i++`.

С помощью скобок нельзя задать порядок вычисления. Например, `a*(b/c)` может вычисляться и как `(a*b)/c`, поскольку `*` и `/` имеют одинаковый приоритет. В тех случаях, когда важен порядок вычисления, можно вводить дополнительную переменную, например, `(t=b/c, a*t)`.

3.1.3. Увеличение и уменьшение

Операция `++` используется для явного выражения приращения вместо его неявного выражения с помощью комбинации сложения и присваивания. По определению `++lvalue` означает `lvalue+=1`, что в свою очередь означает `lvalue=lvalue+1` при условии, что `lvalue` не вызывает никаких побочных эффектов. Выражение, обозначающее (дентирующее) объект, который должен быть увеличен, вычисляется один раз (только). Аналогично, уменьшение выражается операцией `--`. Операции `++` и `--` могут применяться и как префиксные, и как постфиксные. Значением `++x` является новое (то есть увеличенное) значение `x`. Например, `y=++x` эквивалентно `y=(x+=1)`. Значение `x++`, напротив, есть старое значение `x`. Например, `y=x++` эквивалентно `y=(t=x, x+=1, t)`, где `t` - переменная того же типа, что и `x`.

Операции приращения особенно полезны для увеличения и уменьшения переменных в циклах. Например, оканчивающуюся нулем строку можно копировать так:

```
inline void cpy(char* p, const char* q)
{
    while (*p++ = *q++) ;
}
```

Напомню, что увеличение и уменьшение указателей, так же как сложение и вычитание указателей, осуществляется в терминах элементов вектора, на которые указывает указатель; `p++` приводит к тому, что `p` указывает на следующий элемент. Для указателя `p` типа `T*`, по определению, выполняется следующее:

```
long(p+1) == long(p)+sizeof(T);
```


3.1.4. Побитовые логические операции

Побитовые логические операции

& | ^ ~ >> <<

применяются к целым, то есть к объектам типа char, short, int, long и их unsigned аналогам, результаты тоже целые.

Одно из стандартных применений побитовых логических операций - реализация маленького множества (вектора битов). В этом случае каждый бит беззнакового целого представляет один член множества, а число членов ограничено числом битов. Бинарная операция & интерпретируется как пересечение, | – как объединение, а ^ – как разность. Для именования членов такого множества можно использовать перечисление. Вот маленький пример, заимствованный из реализации (непользовательского интерфейса):

```
enum state_value { _good=0, _eof=1, _fail=2, _bad=4};
                // хорошо, конец файла, ошибка, плохо
```

Определение _good не является необходимым. Автор хотел, чтобы состояние, когда все в порядке, имело подходящее имя. Состояние потока можно установить заново следующим образом:

```
cout.state = _good;
```

Например, так можно проверить, не был ли испорчен поток или допущена операционная ошибка:

```
if (cout.state & (_bad | _fail)) // не good
```

Еще одни скобки необходимы, поскольку & имеет более высокий приоритет, чем |.

Функция, достигающая конца ввода, может сообщать об этом так:

```
cin.state |= _eof;
```

Операция |= используется потому, что поток уже может быть испорчен (то есть, state == _bad), поэтому

```
cin.state = _eof;
```

очистило бы этот признак. Различие двух потоков можно находить так:

```
state_value diff = cin.state ^ cout.state;
```

В случае типа stream_state (состояние потока) такая разность не очень нужна, но для других похожих типов она оказывается самой полезной. Например, при сравнении вектора бит, представляющего мно-

жество прерываний, которые обрабатываются, с другим, представляющим прерывания, ждущие обработки.

Следует заметить, что использование полей в действительности является сокращенной записью сдвига и маскирования для извлечения полей бит из слова. Это, конечно, можно сделать и с помощью побитовых логических операций. Например, извлечь средние 16 бит из 32-битового `int` можно следующим образом:

```
unsigned short middle(int a) { return (a>>8)&0xffff; }
```

Не путайте побитовые логические операции с логическими операциями:

```
&&      ||      !
```

Последние возвращают 0 или 1, и они главным образом используются для записи проверки в операторах `if`, `while` или `for`. Например, `!0` (не ноль) есть значение 1, тогда как `~0` (дополнение нуля) есть набор битов все-единицы, который обычно является значением -1.

3.1.5. Преобразование типа

Бывает необходимо явно преобразовать значение одного типа в значение другого. Явное преобразование типа дает значение одного типа для данного значения другого типа. Например:

```
float r = float(1);
```

перед присваиванием преобразует целое значение 1 к значению с плавающей точкой 1.0. Результат преобразования типа не является lvalue, поэтому ему нельзя присваивать (если только тип не является ссылочным типом).

Есть два способа записи явного преобразования типа: традиционная в C запись приведения к типу `(double)a` и функциональная запись `double(a)`. Функциональная запись не может применяться для типов, которые не имеют простого имени. Например, чтобы преобразовать значение к указательному типу надо или использовать запись приведения

```
char* p = (char*)0777;
```

или определить новое имя типа:

```
typedef char* Pchar;
char* p = Pchar(0777);
```

Функциональная запись, как правило, в нетривиальных случаях предпочтительна. Рассмотрим два эквивалентных примера

```

Pname n2 = Pbase(n1->tp)->b_name;           //
функциональная запись
Pname n3 = ((Pbase)n2->tp)->b_name;         //
запись приведения к типу

```

Поскольку операция `->` имеет больший приоритет, чем приведение, последнее выражение интерпретируется как

```
((Pbase)(n2->tp))->b_name
```

С помощью явного преобразования типа к указательным типам можно симитировать, что объект имеет совершенно произвольный тип. Например:

```
any_type* p = (any_type*)&some_object;
```

позволит работать посредством `p` с некоторым объектом `some_object` как с любым типом `any_type`.

Когда преобразование типа не необходимо, его следует избегать. Программы, в которых используется много явных преобразований типов, труднее понимать, чем те, в которых это не делается. Однако такие программы легче понимать, чем программы, просто не использующие типы для представления понятий более высокого уровня (например, программу, которая оперирует регистром устройства с помощью сдвига и маскирования, вместо того, чтобы определить подходящую `struct` и оперировать ею). Кроме того, правильность явного преобразования типа часто критическим образом зависит от понимания программистом того, каким образом объекты различных типов обрабатываются в языке, и очень часто от подробностей реализации. Например:

```

int i = 1;
char* pc = «asdf»;
int* pi = &i;

i = (int)pc;
pc = (char*)i;           // остерегайтесь: значение
pc может измениться
                        // на некоторых машинах
                        // sizeof(int)

```

3.1.6. Свободная память

Именованный объект является либо статическим, либо автоматическим. Статический объект размещается во время запуска программы и существует в течение всего выполнения программы. Автоматиче-

ский объект размещается каждый раз при входе в его блок и существует только до тех пор, пока из этого блока не вышли. Однако часто бывает полезно создать новый объект, существующий до тех пор, пока он не станет ненужным. В частности, часто полезно создать объект, который можно использовать после возврата из функции, где он создается. Такие объекты создает операция `new`, а в последствие уничтожать их можно операцией `delete`. Про объекты, выделенные с помощью операции `new`, говорят, что они в свободной памяти. Такими объектами обычно являются вершины деревьев или элементы связанных списков, являющиеся частью большей структуры данных, размер которой не может быть известен на стадии компиляции.

Объект, созданный с помощью `new`, существует, пока он не будет явно уничтожен `delete`, после чего пространство, которое он занимал, опять может использоваться `new`. Никакого «сборщика мусора», который ищет объекты, на которые нет ссылок, и предоставляет их в распоряжение `new`, нет. Операция `delete` может применяться только к указателю, который был возвращен операцией `new`, или к нулю. Применение `delete` к нулю не вызывает никаких действий.

С помощью `new` можно также создавать векторы объектов. Например:

```
char* save_string(char* p)
{
    char* s = new char[strlen(p)+1];
    strcpy(s,p);
    return s;
}
```

Следует заметить, что, чтобы освободить пространство, выделенное `new`, `delete` должна иметь возможность определить размер выделенного объекта. Например:

```
int main(int argc, char* argv[])
{
    if (argc < 2) exit(1);
    char* p = save_string(argv[1]);
    delete p;
}
```

Это приводит к тому, что объект, выделенный стандартной реализацией `new`, будет занимать больше места, чем статический объект (обычно больше на одно слово).

Можно также явно указывать размер вектора в операции уничтожения `delete`. Например:

```
int main(int argc, char* argv[])
{
    if (argc < 2) exit(1);
    int size = strlen(argv[1])+1;
    char* p = save_string(argv[1]);
    delete[size] p;
}
```

Заданный пользователем размер вектора игнорируется, за исключением некоторых типов, определяемых пользователем.

Операции свободной памяти реализуются функциями:

```
void operator new(long);
void operator delete(void*);
```

Стандартная реализация new не инициализирует возвращаемый объект.

Что происходит, когда new не находит памяти для выделения? Поскольку даже виртуальная память конечна, это иногда должно происходить. Запрос вроде

```
char* p = new char[1000000000];
```

как правило, приводит к каким-то неприятностям. Когда у new ничего не получается, она вызывает функцию, указываемую указателем `_new_handler`. Вы можете задать указатель явно или использовать функцию `set_new_handler()`. Например:

```
#include
```

```
void out_of_store()
{
    cerr << «операция new не прошла: за преде-
лами памяти\n»;
    exit(1);
}
```

```
typedef void (*PF)();      // тип указатель на
функцию
```

```
extern PF set_new_handler(PF);
```

```
main()
{
    set_new_handler(out_of_store);
    char* p = new char[1000000000];
    cout << «сделано, p = « << long(p) << «\n»;
```

```
}
```

как правило, не будет писать «сделано», а будет вместо этого выдавать операция new не прошла: за пределами памяти

_new_handler может делать и кое-что поумнее, чем просто завершать выполнение программы. Если вы знаете, как работают new и delete, например, потому, что вы задали свои собственные operator new() и operator delete(), программа обработки может попытаться найти некоторое количество памяти, которое возвратит new. Другими словами, пользователь может сделать сборщик мусора, сделав, таким образом, использование delete необязательным. Но это, конечно, все-таки задача не для начинающего.

По историческим причинам new просто возвращает указатель 0, если она не может найти достаточное количество памяти и не был задан никакой _new_handler. Например

```
#include

main()
{
    char* p = new char[1000000000];
    cout << «сделано, p = « << long(p) << «\n»;
}
```

выдаст

```
сделано, p = 0
```

Вам сделали предупреждение! Заметьте, что тот, кто задает _new_handler, берет на себя заботу по проверке истощения памяти при каждом использовании new в программе (за исключением случая, когда пользователь задал отдельные подпрограммы для размещения объектов заданных типов, определяемых пользователем).

3.2. Сводка операторов

Операторы C++ систематически и полностью изложены в соответствующем разделе, прочитайте, пожалуйста, этот раздел. А здесь приводится краткая сводка и некоторые примеры.

Синтаксис оператора

```
оператор:
    описание
    {список_операторов opt}
    выражение opt
```

```

if ( выражение ) оператор
if ( выражение ) оператор else оператор
switch ( выражение ) оператор

while ( выражение ) оператор
do оператор while (выражение)
for ( оператор выражение opt ; выражение
opt ) оператор

case константное_выражение : оператор
default : оператор
break ;
continue ;

return выражение opt ;

goto идентификатор ;
идентификатор : оператор

список_операторов:
    оператор
    оператор список_операторов

```

Заметьте, что описание является оператором и что нет операторов присваивания и вызова процедуры. Присваивание и вызов функции обрабатываются как выражения.

3.2.1. Проверки

Проверка значения может осуществляться или оператором if, или оператором switch:

```

if ( выражение ) оператор
if ( выражение ) оператор else оператор
switch ( выражение ) оператор

```

В C++ нет отдельного булевского типа. Операции сравнения

```

==      !=      <      <=      >      >=

```

возвращают целое 1, если сравнение истинно, иначе возвращают 0. Не так уж непривычно видеть, что ИСТИНА определена как 1, а ЛОЖЬ определена как 0.

В операторе if первый (или единственный) оператор выполняется в том случае, если выражение ненулевое, иначе выполняется второй оператор (если он задан). Отсюда следует, что в качестве условия может использоваться любое целое выражение. В частности, если *a* целое, то

```
if (a) // ...
```

эквивалентно

```
if (a != 0) // ...
```

Логические операции

```
&&    ||    !
```

наиболее часто используются в условиях. Операции && и || не будут вычислять второй аргумент, если это не нужно. Например:

```
if (p && lcount) // ...
```

вначале проверяет, является ли *p* не нулем, и только если это так, то проверяет *lcount*.

Некоторые простые операторы if могут быть с удобством заменены выражениями арифметического if. Например:

```
if (a <= d)
    max = b;
else
    max = a;
```

лучше выражается так:

```
max = (a <= b) ? b : a;
```

Скобки вокруг условия необязательны, но автор пособия считает, что когда они используются, программу легче читать.

Некоторые простые операторы switch можно по-другому записать в виде набора операторов if. Например:

```
switch (val) {
case 1:
    f();
    break;
case 2:
    g();
    break;
default:
    h();
    break;
}
```

иначе можно было бы записать так:


```

if (val == 1)
    f();
else if (val == 2)
    g();
else
    h();

```

Смысл тот же, однако первый вариант (switch) предпочтительнее, поскольку в этом случае явно выражается сущность действия (сопоставление значения с рядом констант). Поэтому в нетривиальных случаях оператор switch читается легче.

Забойтесь о том, что switch должен как-то завершаться, если только вы не хотите, чтобы выполнялся следующий case. Например:

```

switch (val) {    // осторожно
case 1:
    cout << «case 1\n»;
case 2:
    cout << «case 2\n»;
default:
    cout << «default: case не найден\n»;
}

```

при val==1 напечатает

```

case 1
case 2
default: case не найден

```

к великому изумлению непосвященного. Самый обычный способ завершить случай - это break, иногда можно даже использовать goto. Например:

```

switch (val) {    // осторожно
case 0:
    cout << «case 0\n»;
case1:
case 1:
    cout << «case 1\n»;
    return;
case 2:
    cout << «case 2\n»;
    goto case1;
default:
    cout << «default: case не найден\n»;
}

```

```

        return;
    }
    При обращении к нему с val==2 выдаст
    case 2
    case 1

```

Заметьте, что метка case не подходит как метка для употребления в операторе goto:

```
goto case 1;           // синтаксическая ошибка
```

3.2.2. Goto

C++ снабжен имеющим дурную репутацию оператором goto.

```
goto идентификатор;
идентификатор : оператор
```

В общем, в программировании высокого уровня он имеет очень мало применений, но он может быть очень полезен, когда C++ программа генерируется программой, а не пишется непосредственно человеком. Например, операторы goto можно использовать в синтаксическом анализаторе, порождаемом генератором синтаксических анализаторов. Оператор goto может быть также важен в тех редких случаях, когда важна наилучшая эффективность, например, во внутреннем цикле какой-нибудь программы, работающей в реальном времени.

Одно из немногих разумных применений состоит в выходе из вложенного цикла или переключателя (break лишь прекращает выполнение самого внутреннего охватывающего его цикла или переключателя). Например:

```
for (int i = 0; i
```

3.3. Комментарии и выравнивание

Продуманное использование комментариев и согласованное использование отступов может сделать чтение и понимание программы намного более приятным. Существует несколько различных стилей согласованного использования отступов. Автор не видит никаких серьезных оснований предпочесть один другому (хотя, как и у большинства, у автора есть свои предпочтения). Сказанное относится также и к стилю комментариев.

Неправильное использование комментариев может серьезно повлиять на удобочитаемость программы, Компилятор не понимает со-

держание комментария, поэтому он никаким способом не может убедиться в том, что комментарий

- [1] осмыслен;
- [2] описывает программу и
- [3] не устарел.

Непонятные, двусмысленные и просто неправильные комментарии содержатся в большинстве программ. Плохой комментарий может быть хуже, чем никакой.

Если что-то можно сформулировать средствами самого языка, следует это сделать, а не просто отметить в комментарии. Данное замечание относится к комментариям вроде:

```
// переменная «v» должна быть инициализирована.

// переменная «v» должна использоваться только
функцией «f()».

// вызвать функцию init() перед вызовом

// любой другой функции в этом файле.

// вызовите функцию очистки «cleanup()» в конце
вашей программы.

// не используйте функцию «wierd()».

// функция «f()» получает два параметра.
```

При правильном использовании C++ подобные комментарии, как правило, становятся ненужными. Чтобы предыдущие комментарии стали излишними, можно, например, использовать правила компоновки и видимость, инициализацию и правила очистки для классов.

Если что-то было ясно сформулировано на языке, второй раз упоминать это в комментариях не следует. Например:

```
a = b+c;      // a становится b+c
count++;      // увеличить счетчик
```

Такие комментарии хуже, чем просто излишни, они увеличивают объем текста, который надо прочитать, они часто затуманивают структуру программы, и они могут быть неправильными.

Бьярн Страуструп рекомендует использовать:

[1] комментарий для каждого исходного файла, сообщающий, для чего, в целом, предназначены находящиеся в нем комментарии, даю-

ций ссылки на справочники и руководства, общие рекомендации по использованию и т.д.;

[2] комментариев для каждой нетривиальной функции, в котором сформулировано ее назначение, используемый алгоритм (если он неочевиден) и, быть может, что-то о принимаемых в ней предположениях относительно среды выполнения;

[3] небольшое число комментариев в тех местах, где программа неочевидна и/или непереносима; и

[4] очень мало что еще.

Например:

```
// tbl.c: Реализация таблицы имен
/*
    Гауссовское исключение с частичным
    См. Ralston: «A first course ...» стр. 411.
*/

// swap() предполагает размещение стека AT&T sB20.

/*****

    Copyright (c) 1984 AT&T, Inc.
    All rights reserved

    *****/
```

Удачно подобранные и хорошо написанные комментарии - существенная часть программы. Написание хороших комментариев может быть столь же сложным, сколь и написание самой программы.

Заметьте также, что если в функции используются исключительно комментарии `//`, то любую часть этой функции можно закомментировать с помощью комментариев `/* */`, и наоборот.

4. ГЛАВА ЧЕТВЕРТАЯ

Все нетривиальные программы собираются из нескольких раздельно компилируемых единиц (их принято называть просто файлами). В этой главе описано, как раздельно откомпилированные функции могут обращаться друг к другу, как такие функции могут совместно пользоваться данными (разделять данные) и как можно обеспечить согласованность типов, которые используются в разных файлах программы. Функции обсуждаются довольно подробно. Сюда входят передача параметров, параметры по умолчанию, перегрузка имен функций и, конечно же, описание и определение функций. В конце описываются макросы.

4.1. Введение

Иметь всю программу в одном файле обычно невозможно, поскольку коды стандартных библиотек и операционной системы находятся где-то в другом месте. Кроме того, хранить весь текст пользовательской программы в одном файле, как правило, непрактично и неудобно. Способ организации программы в файлы может помочь читающему охватить всю структуру программы, а также может дать возможность компилятору реализовать эту структуру. Поскольку единицей компиляции является файл, то во всех случаях, когда в файл вносится изменение (сколь бы мало оно ни было), весь файл нужно компилировать заново. Даже для программы умеренных размеров время, затрачиваемое на перекомпиляцию, можно значительно снизить с помощью разбиения программы на файлы подходящих размеров.

Программа, состоящая из нескольких раздельно компилируемых файлов, должна быть согласованной в смысле использования имен и типов, точно так же, как и программа, состоящая из одного исходного файла. В принципе, это может обеспечить и компоновщик. Компоновщик - это программа, стыкующая отдельно скомпилированные части вместе. Компоновщик часто (путая) называют загрузчиком. В UNIX'e компоновщик называется `ld`. Однако компоновщики, имеющиеся в большинстве систем, обеспечивают очень слабую поддержку проверки согласованности.

Программист может скомпенсировать недостаток поддержки со стороны компоновщика, предоставив дополнительную информацию о типах (описания). После этого согласованность программы обеспечивается проверкой согласованности описаний, которые находятся в раздельно компилируемых частях. Средства, которые это обеспечивают, в

вашей системе будут. C++ разработан так, чтобы способствовать такой явной компоновке.

4.2. Компоновка

Если не указано иное, то имя, не являющееся локальным для функции или класса, в каждой части программы, компилируемой отдельно, должно относиться к одному и тому же типу, значению, функции или объекту. То есть, в программе может быть только один нелокальный тип, значение, функция или объект с этим именем. Рассмотрим, например, два файла:

```
// file1.c:
    int a = 1;
    int f() { /* что-то делает */ }

// file2.c:
    extern int a;
    int f();
    void g() { a = f(); }
```

`a` и `f()`, используемые `g()` в файле `file2.c`, те же, что определены в файле `file1.c`. Ключевое слово `extern` (внешнее) указывает, что описание `a` в `file2.c` является (только) описанием, а не определением. Если бы `a` инициализировалось, `extern` было бы просто проигнорировано, поскольку описание с инициализацией всегда является определением. Объект в программе должен определяться только один раз. Описываться он может много раз, но типы должны точно согласовываться. Например:

```
// file1.c:
    int a = 1;
    int b = 1;
    extern int c;

// file2.c:
    int a;
    extern double b;
    extern int c;
```

Здесь три ошибки: `a` определено дважды (`int a`; является определением, которое означает `int a=0;`), `b` описано дважды с разными типами, `a` с описано дважды, но не определено. Эти виды ошибок (ошибки компоновки) не могут быть обнаружены компилятором, который за

один раз видит только один файл. Компоновщик, однако, их обнаруживает.

Следующая программа не является C++-программой (хотя C-программой является):

```
// file1.c:
    int a;
    int f() { return a; }

// file2.c:
    int a;
    int g() { return f(); }
```

Во-первых, file2.c не C++, потому что f() не была описана, и поэтому компилятор будет недоволен. Во-вторых, (когда file2.c фиксирован) программа не будет скомпонована, поскольку а определено дважды.

Имя можно сделать локальным в файле, описав его static. Например:

```
// file1.c:
    static int a = 6;
    static int f() { /* ... */ }

// file2.c:
    static int a = 7;
    static int f() { /* ... */ }
```

Поскольку каждое а и f описано как static, получающаяся в результате программа является правильной. В каждом файле своя а и своя f().

Когда переменные и функции явно описаны как static, часть программы легче понять (вам не надо никуда больше заглядывать). Использование static для функций может, помимо этого, выгодно влиять на расходы по вызову функции, поскольку дает оптимизирующему компилятору более простую работу.

Рассмотрим два файла:

```
// file1.c:
    const int a = 6;
    inline int f() { /* ... */ }
    struct s { int a,b; }

// file1.c:
    const int a = 7;
    inline int f() { /* ... */ }
    struct s { int a,b; }
```

Раз правило «ровно одно определение» применяется к константам, inline-функциям и определениям функций так же, как оно применяется к функциям и переменным, то file1.c и file2.c не могут быть частями одной C++-программы. Но если это так, то как же два файла могут использовать одни и те же типы и константы? Коротко, ответ таков: типы, константы и т.п. могут определяться столько раз, сколько нужно, при условии, что они определяются одинаково. Полный ответ несколько более сложен (это объясняется в следующем разделе).

4.3. Заголовочные файлы

Типы во всех описаниях одного и того же объекта должны быть согласованными. Один из способов это достичь мог бы состоять в обеспечении средств проверки типов в компоновщике, но большинство компоновщиков - образца 1950-х гг., и их нельзя изменить по практическим соображениям. Другой подход состоит в обеспечении того, что исходный текст, как он передается на рассмотрение компилятору, или согласован, или содержит информацию, которая позволяет компилятору обнаружить несогласованности. Один несовершенный, но простой способ достичь согласованности состоит во включении заголовочных файлов, содержащих интерфейсную информацию, в исходные файлы, в которых содержится исполняемый код и/или определения данных.

Механизм включения с помощью `#include` - это чрезвычайно простое средство обработки текста для сборки кусков исходной программы в одну единицу (файл) для ее компиляции. Директива

```
#include «to_be_included»
```

замещает строку, в которой встретилось `#include`, содержимым файла «to_be_included». Его содержимым должен быть исходный текст на C++, поскольку дальше его будет читать компилятор. Часто включение обрабатывается отдельной программой, называемой C-препроцессором, которую СС вызывает для преобразования исходного файла, который дал программист, в файл без директив включения перед тем, как начать собственно компиляцию. В другом варианте эти директивы обрабатывает интерфейсная система компилятора по мере того, как они встречаются в исходном тексте.

Для включения файлов из стандартной директории включения вместо кавычек используются угловые скобки `< и >`. Например:

```
#include          // из стандартной директории
включения
```



```
#define «myheader.h»      // из текущей дирек-
тории
```

Использование \diamond имеет то преимущество, что в программу фактическое имя директории включения не встраивается (как правило, сначала просматривается /usr/include/CC, а потом usr/include). К сожалению, пробелы в директиве include существенны:

```
#include < stream.h >      // не найдет
```

Может показаться, что перекомпилировать файл заново каждый раз, когда он куда-либо включается, расточительно, но время компиляции такого файла обычно слабо отличается от времени, которое необходимо для чтения его некоторой заранее откомпилированной формы. Причина в том, что текст программы является довольно компактным представлением программы, и в том, что включаемые файлы обычно содержат только описания и не содержат программ, требующих от компилятора значительного анализа.

Следующее эмпирическое правило относительно того, что следует, а что не следует помещать в заголовочные файлы, является не требованием языка, а просто предложением по разумному использованию аппарата #include.

В заголовочном файле могут содержаться:

Определения типов	struct point { int x, y; }
Описания функций	extern int strlen(const char*);
Определения inline-функций	inline char get() { return *p++; }
Описания данных	extern int a;
Определения констант	const float pi = 3.141593
Перечисления	enum bool { false, true };
Директивы include	#include
Определения макросов	#define Case break;case
Комментарии /* проверка на конец файла */	

но никогда

Определения обычных функций	char get() { return *p++; }
Определения данных	int a;
Определения сложных	
константных объектов	const tbl[] = { /* ... */ }

В системе UNIX принято, что заголовочные файлы имеют суффикс (расширение) *.h. Файлы, содержащие определение данных или функций, должны иметь суффикс *.c. Такие файлы часто называют, соот-

ветственно, «*.h файлы» и «*.c файлы». Ниже описываются макросы. Следует заметить, что в C++ макросы гораздо менее полезны, чем в C, поскольку C++ имеет такие языковые конструкции, как `const` для определения констант и `inline` для исключения расходов на вызов функции.

Причина того, почему в заголовочных файлах допускается определение простых констант, но не допускается определение сложных константных объектов, прагматическая. В принципе, сложность тут только в том, чтобы сделать допустимым дублирование определений переменных (даже определения функций можно было бы дублировать). Однако для компоновщиков старого образца слишком трудно проверять тождественность нетривиальных констант и убирать ненужные повторы. Кроме того, простые случаи гораздо более обиходны и потому более важны для генерации хорошего кода.

4.3.1. Один заголовочный файл

Проще всего решить проблему разбиения программы на несколько файлов, поместив функции и определения данных в подходящее число исходных файлов и описав типы, необходимые для их взаимодействия, в одном заголовочном файле, который включается во все остальные файлы

Важный случай, когда размер заголовочных файлов становится серьезной помехой. Набор заголовочных файлов и библиотеку можно использовать для расширения языка множеством обще- и специально-прикладных типов (см. главы 5-8). В таких случаях не принято осуществлять чтение тысяч строк заголовочных файлов в начале каждой компиляции. Содержание этих файлов обычно «заморожено» и изменяется очень нечасто. Наиболее полезным может оказаться метод затравки компилятора содержанием этих заголовочных файлов. По сути, создается язык специального назначения со своим собственным компилятором. Никакого стандартного метода создания такого компилятора с затравкой не принято. Тем не менее, почти все современные коммерческие компиляторы снабжены подобным механизмом, в Borland C/C++, например, это называется «precompiled headers».

4.3.2. Множественные заголовочные файлы

Стиль разбиения программы с одним заголовочным файлом наиболее пригоден в тех случаях, когда программа невелика и ее части не предполагается использовать отдельно. Поэтому то, что невозможно установить, какие описания зачем помещены в заголовочный файл,

несущественно. Помочь могут комментарии. Другой способ - сделать так, чтобы каждая часть программы имела свой заголовочный файл, в котором определяются предоставляемые этой частью средства. Тогда каждый *.c-файл имеет соответствующий *.h-файл, и каждый *.c-файл включает свой собственный (специфицирующий то, что в нем задается) *.h-файл и, возможно, некоторые другие *.h-файлы (специфицирующие то, что ему нужно).

Сколько заголовочных файлов использовать в программе, зависит от многих факторов. Многие из этих факторов сильнее связаны с тем, как ваша система работает с заголовочными файлами, нежели с C++. Например, если в вашем редакторе нет средств, позволяющих одновременно видеть несколько файлов, использование большого числа файлов становится менее привлекательным. Аналогично, если открытие и чтение 10 файлов по 50 строк в каждом требует заметно больше времени, чем чтение одного файла в 500 строк, вы можете дважды подумать, прежде чем использовать в небольшом проекте стиль множественных заголовочных файлов. Слово предостережения: набор из десяти заголовочных файлов плюс стандартные заголовочные файлы обычно легче поддаются управлению. С другой стороны, если вы разбили описания в большой программе на логически минимальные по размеру заголовочные файлы (помещая каждое описание структуры в свой отдельный файл и т.д.), у вас легко может получиться неразбериха из сотен файлов.

4.3.3. Скрытие данных

Используя заголовочные файлы, пользователь может определять явный интерфейс, чтобы обеспечить согласованное использование типов в программе. С другой стороны, пользователь может обойти интерфейс, задаваемый заголовочным файлом, вводя в *.c-файлы описания extern.

Заметьте, что такой стиль компоновки не рекомендуется:

```
// file1.c:           // «extern» не используется
int a = 7;
const c = 8;
void f(long) { /* ... */ }
```

```
// file2.c:           // «extern» в .c файле
extern int a;
extern const c;
extern f(int);
int g() { return f(a+c); }
```

Поскольку описания extern в file2.c не включаются вместе с определениями в файле file1.c, компилятор не может проверить согласованность этой программы. Следовательно, если только загрузчик не окажется гораздо сообразительнее среднего, две ошибки в этой программе останутся, и их придется искать программисту.

Пользователь может защитить файл от такой недисциплинированной компоновки, описав имена, которые не предназначены для общего пользования, как static, чтобы их областью видимости был файл, и они были скрыты от остальных частей программы. Например:

```
// table.c: определения таблицы имен

#include «error.h»
#include
#include «table.h»

const TBLSZ = 23;
static name* table[TBLSZ];

name* look(char* p; int ins) { /* ... */ }
```

Это гарантирует, что любой доступ к table действительно будет осуществляться именно через look(). «Прятать» константу TBLSZ не обязательно.

4.4. Файлы как Модули

В предыдущем разделе *.c- и *.h-файлы вместе определяли часть программы. Файл *.h является интерфейсом, который используют другие части программы; *.c-файл задает реализацию. Такой объект часто называют модулем. Доступными делаются только те имена, которые необходимо знать пользователю, остальные скрыты. Это качество часто называют скрытием данных, хотя данные - лишь часть того, что может быть скрыто. Модули такого вида обеспечивают большую гибкость. Например, реализация может состоять из одного или более *.c- файлов, и в виде *.h-файлов может быть предоставлено несколько интерфейсов. Информация, которую пользователю знать не обязательно, искусно скрыта в *.c-файлах. Если важно, что пользователь не должен точно знать, что содержится в *.c-файлах, не надо делать их доступными в исходном виде. Достаточно эквивалентных им выходных файлов компилятора (*.o-файлов).

Иногда возникает сложность, состоящая в том, что подобная гибкость достигается без формальной структуры. Сам язык не распознает

такой модуль как объект, и у компилятора нет возможности отличить *.h-файлы, определяющие имена, которые должны использовать другие модули (экспортируемые), от *.h-файлов, которые описывают имена из других модулей (импортируемые).

В других случаях может возникнуть такая проблема, когда модуль определяет множество объектов, а не новый тип. Например, модуль `table` определяет одну таблицу, и если вам нужно две таблицы, то нет простого способа задать вторую таблицу с помощью понятия модуля. Решение этой проблемы приводится в Главе 5.

Каждый статически размещенный объект по умолчанию инициализируется нулем, программист может задать другие (константные) значения. Это только самый примитивный вид инициализации. К счастью, с помощью классов можно задать код, который выполняется для инициализации перед тем, как модуль каким-либо образом используется, и/или код, который запускается для очистки после последнего использования модуля.

4.5. Как создать библиотеку

Фразы типа «помещен в библиотеку» и «ищется в какой-то библиотеке» используются часто (и в этой книге, и в других), но что это означает для C++-программы? К сожалению, ответ зависит от того, какая операционная система используется; в этом разделе объясняется, как создать библиотеку в 8-ой версии системы UNIX. Другие системы предоставляют аналогичные возможности.

Библиотека в своей основе является множеством *.o-файлов, полученных в результате компиляции соответствующего множества *.c-файлов. Обычно имеется один или более *.h-файлов, в которых содержатся описания для использования этих *.o-файлов. В качестве примера рассмотрим случай, когда нам надо задать (обычным способом) набор математических функций для некоторого неопределенного множества пользователей. Заголовочный файл мог бы выглядеть примерно так:

```
extern double sqrt(double);    // подмножество
extern double sin(double);
extern double cos(double);
extern double exp(double);
extern double log(double);
```

а определения этих функций хранились бы соответственно в файлах `sqrt.c`, `sin.c`, `cos.c`, `exp.c` и `log.c`.

Библиотеку с именем `math.h` можно создать, например, так:

```
$ CC -c sqrt.c sin.c cos.c exp.c log.c
$ ar cr math.a sqrt.o sin.o cos.o exp.o log.o
$ ranlib math.a
```

Вначале исходные файлы компилируются в эквивалентные им объектные файлы. Затем используется команда `ar`, чтобы создать архив с именем `math.a`. И, наконец, этот архив индексируется для ускорения доступа. Если в вашей системе нет команды `ranlib`, значит она вам, вероятно, не понадобится. Подробности посмотрите, пожалуйста, в вашем руководстве в разделе под заголовком `ar`. Использовать библиотеку можно, например, так:

```
$ CC myprog.c math.a
```

Теперь разберемся, в чем же преимущества использования `math.a` перед просто непосредственным использованием *.o-файлов? Например:

```
$ CC myprog.c sqrt.o sin.o cos.o exp.o log.o
```

Для большинства программ определить правильный набор *.o-файлов, несомненно, непросто. В приведенном выше примере они включались все, но если функции в `myprog.c` вызывают только функции `sqrt()` и `cos()`, то кажется, что будет достаточно

```
$ CC myprog.c sqrt.o cos.o
```

Но это не так, поскольку `cos.c` использует `sin.c`.

Компоновщик, вызываемый командой `CC` для обработки *.a-файла (в данном случае, файла `math.a`) знает, как из того множества, которое использовалось для создания *.a-файла, извлечь только необходимые *.o-файлы.

Другими словами, используя библиотеку, можно включать много определений с помощью одного имени (включения определений функций и переменных, используемых внутренними функциями, никогда не видны пользователю) и, кроме того, обеспечить, что в результате в программу будет включено минимальное количество определений.

4.6. Функции

Обычный способ сделать что-либо в C++-программе - это вызвать функцию, которая это делает. Определение функции является способом задать то, как должно делаться некоторое действие. Функция не может быть вызвана, пока она не описана.

4.6.1. Описания функций

Описание функции задает имя функции, тип возвращаемого функцией значения (если таковое есть) и число и типы параметров, которые должны быть в вызове функции. Например:

```
extern double sqrt(double);
extern elem* next_elem();
extern char* strcpy(char* to, const char* from);
extern void exit(int);
```

Семантика передачи параметров идентична семантике инициализации. Проверяются типы параметров, и, когда нужно, производится неявное преобразование типа. Например, если были заданы предыдущие определения, то

```
double sr2 = sqrt(2);
```

будет правильно обращаться к функции `sqrt()` со значением с плавающей точкой 2.0. Значение такой проверки типа и преобразования типа огромно.

Описание функции может содержать имена параметров. Это может помочь читателю, но компилятор эти имена просто игнорирует.

4.6.2. Определения функций

Каждая функция, вызываемая в программе, должна быть где-то определена (только один раз). Определение функции - это описание функции, в котором приводится тело функции. Например:

```
extern void swap(int*, int*);    // описание

void swap(int*, int*)           // определение
{
    int t = *p;
    *p = *q;
    *q = t;
}
```

Чтобы избежать расходов на вызов функции, функцию можно описать как `inline`, а чтобы обеспечить более быстрый доступ к параметрам, их можно описать как `register`. Оба средства могут использоваться неправильно, и их следует избегать везде где есть какие-либо сомнения в их полезности.

4.6.3. Передача Параметров

Когда вызывается функция, дополнительно выделяется память под ее формальные параметры и каждый формальный параметр инициализируется соответствующим ему фактическим параметром. Семантика передачи параметров идентична семантике инициализации. В частности, тип фактического параметра сопоставляется с типом формального параметра, и выполняются все стандартные и определенные пользователем преобразования типов. Есть особые правила для передачи векторов, средство передавать параметр без проверки и средство для задания параметров по умолчанию. Рассмотрим

```
void f(int val, int& ref)
{
    val++;
    ref++;
}
```

Когда вызывается `f()`, `val++` увеличивает локальную копию первого фактического параметра, тогда как `ref++` увеличивает второй фактический параметр. Например:

```
int i = 1;
int j = 1;
f(i, j);
```

увеличивает `j`, но не `i`. Первый параметр, `i`, передается по значению, второй параметр, `j`, передается по ссылке. Как уже отмечалось выше, использование функций, которые изменяют переданные по ссылке параметры, могут сделать программу трудночитаемой, и их следует избегать. Однако передача большого объекта по ссылке может быть гораздо эффективнее, чем передача его по значению. В этом случае параметр можно описать как `const`, чтобы указать, что ссылка применяется по соображениям эффективности, а также чтобы не позволить вызываемой функции изменять значение объекта:

```
void f(const large& arg)
{
    // значение «arg» не может быть изменено
}
```

Аналогично, описание параметра указателя как `const` сообщает читателю, что значение объекта, указываемого указателем, функцией не изменяется. Например:


```
extern int strlen(const char*);           // из
extern char* strcpy(char* to, const char* from);
extern int strcmp(const char*, const char*);
```

Важность такой практики растет с размером программы.

Заметьте, что семантика передачи параметров отлична от семантики присваивания. Это важно для `const` параметров, ссылочных параметров и параметров некоторых типов, определяемых пользователем.

4.6.4. Возврат значения

Из функции, которая не описана как `void`, можно (и должно) возвращать значение. Возвращаемое значение задается оператором `return`. Например:

```
int fac(int n) {return (n>1) ? n*fac(n-1) : 1; }
```

В функции может быть больше одного оператора `return`:

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

Как и семантика передачи параметров, семантика возврата функцией значения идентична семантике инициализации. Возвращаемое значение рассматривается как инициализатор переменной возвращаемого типа. Тип возвращаемого выражения проверяется на согласованность с возвращаемым типом, и выполняются все стандартные и определенные пользователем преобразования типов. Например:

```
double f()
{
    // ...
    return 1; // неявно преобразуется к double(1)
}
```

Каждый раз, когда вызывается функция, создается новая копия ее параметров и автоматических переменных. После возврата из функции память используется заново, поэтому возвращать указатель на локальную переменную неразумно. Содержание указываемого места изменится непредсказуемо:

```
int* f() {
    int local = 1;
    // ...
    return &local;           // так не делайте
}
```

Эта ошибка менее обычна, чем эквивалентная ошибка при использовании ссылок:

```
int& f() {
    int local = 1;
    // ...
    return local;           // так не делайте
}
```

К счастью, о таких возвращаемых значениях предупреждает компилятор. Вот другой пример:

```
int& f() { return 1;}       // так не делайте
```

4.6.5. Векторные параметры

Если в качестве параметра функции используется вектор, то передается указатель на его первый элемент. Например:

```
int strlen(const char*);

void f()
{
    char v[] = «a vector»
    strlen(v);
    strlen(«Nicholas»);
};
```

Иначе говоря, при передаче переменной как параметр типа `T[]` преобразуется к `T*`. Следовательно, присваивание элементу векторного параметра изменяет значение элемента вектора, который является параметром. Другими словами, вектор отличается от всех остальных типов тем, что вектор не передается (и не может передаваться) по значению.

Размер вектора недоступен вызываемой функции. Это может быть неудобно, но эту сложность можно обойти несколькими способами. Строки оканчиваются нулем, поэтому их размер можно легко вычислить. Для других векторов можно передавать второй параметр, который задает размер, или определить тип, содержащий указатель и индикатор длины, и передавать его вместо просто вектора. Например:

```

    void computel(int*  vec_ptr,  int  vec_size);
// один способ

    struct vec {
        int* ptr;
        int size;
    };

    void compute2(vec v);

```

С многомерными массивами все хитрее, но часто можно вместо них использовать векторы указателей, которые не требуют специального рассмотрения. Например:

```

char* day[] = {
    «mon», «tue», «wed», «thu», «fri», «sat», «sun»
};

```

С другой стороны, рассмотрим определение функции, которая работает с двумерными матрицами. Если размерность известна на стадии компиляции, то никаких проблем нет:

```

void print_m34(int m[3][4])
{
    for (int i = 0; i<3; i++) {
        for (int j = 0; j<4; j++)
            cout << « » << m[i][j];
        cout << «\n»;
    }
}

```

Матрица, конечно, все равно передается как указатель, а размерности используются просто для удобства записи.

Первая размерность массива не имеет отношения к задаче отыскания положения элемента. Поэтому ее можно передавать как параметр:

```

void print_mi4(int m[][4], int dim1)
{
    for (int i = 0; i

```

4.6.6. Параметры по умолчанию

Часто в самом общем случае функции требуется больше параметров, чем в самом простом и более употребительном случае. Например, в библиотеке потоков есть функция `hex()`, порождающая строку с ше-

стнадцатеричным представлением целого. Второй параметр используется для задания числа символов для представления первого параметра. Если число символов слишком мало для представления целого, происходит усечение, если оно слишком велико, то строка дополняется пробелами. Часто программист не заботится о числе символов, необходимых для представления целого, поскольку символов достаточно. Поэтому для нуля в качестве второго параметра определено значение «использовать столько символов, сколько нужно». Чтобы избежать засорения программы вызовами вроде `hex(i,0)`, функция описывается так:

```
extern char* hex(long, int =0);
```

Инициализатор второго параметра является параметром по умолчанию. То есть, если в вызове дан только один параметр, в качестве второго используется параметр по умолчанию. Например:

```
cout << «**» << hex(31) << hex(32,3) << «**»;
```

интерпретируется как

```
cout << «**» << hex(31,0) << hex(32,3) << «**»;
```

и напечатает:

```
** 1f 20**
```

Параметр по умолчанию проходит проверку типа во время описания функции и вычисляется во время ее вызова. Задавать параметр по умолчанию возможно только для последних параметров, поэтому

```
int f(int, int =0, char* =0);    // ok
int g(int =0, int =0, char*);    // ошибка
int f(int =0, int, char* =0);    // ошибка
```

Заметьте, что в этом контексте пробел между `*` и `=` является существенным (`*` является операцией присваивания):

```
int nasty(char*=0);    // синтаксическая ошибка
```

4.6.7. Перегрузка имен функций

Как правило, давать разным функциям разные имена - мысль хорошая, но когда некоторые функции выполняют одинаковую работу над объектами разных типов, может быть более удобно дать им одно и то же имя. Использование одного имени для различных действий над различными типами называется перегрузкой (overloading). Метод уже используется для основных операций C++: у сложения существует

только одно имя, `+`, но его можно применять для сложения значений целых, плавающих и указательных типов. Эта идея легко расширяется на обработку операций, определенных пользователем, то есть, функций. Чтобы уберечь программиста от случайного повторного использования имени, имя может использоваться более, чем для одной функции, только если оно сперва описано как перегруженное. Например:

```
void print(int);
void print(char*);
```

Что касается компилятора, единственное общее, что имеют функции с одинаковым именем, это имя. Предположительно, они в каком-то смысле похожи, но в этом язык ни стесняет программиста, ни помогает ему. Таким образом, перегруженные имена функций - это главным образом удобство записи. Это удобство значительно в случае функций с общепринятыми именами вроде `sqrt`, `print` и `open`. Когда имя семантически значимо, как это имеет место для операций вроде `+`, `*` и `<<` и в случае конструкторов, это удобство становится существенным. Когда вызывается перегруженная `f()`, компилятор должен понять, к какой из функций с именем `f` следует обратиться. Это делается путем сравнения типов фактических параметров с типами формальных параметров всех функций с именем `f`. Поиск функции, которую надо вызвать, осуществляется за три отдельных шага:

[1] искать функцию, соответствующую точно, и использовать ее, если она найдена;

[2] искать соответствующую, функцию используя встроенные преобразования, и использовать любую найденную функцию; и

[3] искать соответствующую функцию, используя преобразования, определенные пользователем, и, если множество преобразований единственно, использовать найденную функцию.

Например:

```
overload print(double), print(int);
```

```
void f();
{
    print(1);
    print(1.0);
}
```

Правило точного соответствия гарантирует, что `f` напечатает `1` как целое и `1.0` как число с плавающей точкой. Ноль, `char` или `short` точно

соответствуют параметру `int`. Аналогично, `float` точно соответствует `double`.

К параметрам функций с перегруженными именами стандартные C++-правила преобразования применяются не полностью. Преобразования, могущие уничтожить информацию, не выполняются. Остаются `int` в `long`, `int` в `double`, ноль в `long`, ноль в `double` и преобразования указателей: ноль в указатель, ноль в `void*`, и указатель на производный класс в указатель на базовый класс.

Вот пример, в котором преобразование необходимо:

```
overload print(double), print(long);
```

```
void f(int a);
{
    print(a);
}
```

Здесь `a` может быть напечатано или как `double`, или как `long`. Неоднозначность разрешается явным преобразованием типа (или `print(long(a))` или `print(double(a))`).

При этих правилах можно гарантировать, что когда эффективность или точность вычислений для используемых типов существенно различаются, будет использоваться простейший алгоритм (функция). Например:

```
overload pow;
int pow(int, int);
double pow(double, double);           // из
complex pow(double, complex);         // из
complex pow(complex, int);
complex pow(complex, double);
complex pow(complex, complex);
```

Процесс поиска подходящей функции игнорирует `unsigned` и `const`.

4.6.8. Незаданное число параметров

Для некоторых функций невозможно задать число и тип всех параметров, которые можно ожидать в вызове. Такую функцию описывают, завершая список описаний параметров многоточием (...), что означает «и, может быть, еще какие-то параметры». Например:

```
int printf(char* fmt, ...);
```

Это задает, что в вызове `printf` должен быть по меньшей мере один параметр, `char*`, а остальные могут быть, а могут и не быть. Например:

```
printf(«Hello, world\n»);
printf(«Мое имя %s %s\n», first_name, second_name);
printf(«%d + %d = %d\n», 2, 3, 5);
```

Такая функция полагается на информацию, которая недоступна компилятору при интерпретации ее списка параметров. В случае `printf()` первым параметром является строка формата, содержащая специальные последовательности символов, позволяющие `printf()` правильно обрабатывать остальные параметры. `%s` означает «жди параметра `char*`», а `%d` означает «жди параметра `int`». Однако компилятор этого не знает, поэтому он не может убедиться в том, что ожидаемые параметры имеют соответствующий тип. Например:

```
printf(«Мое имя %s %s\n», 2);
```

откомпилируется и в лучшем случае приведет к какой-нибудь странного вида выдаче.

Очевидно, если параметр не был описан, то у компилятора нет информации, необходимой для выполнения над ним проверки типа и преобразования типа. В этом случае `char` или `short` передаются как `int`, а `float` передается как `double`. Это не обязательно то, чего ждет пользователь.

Чрезмерное использование многоточий, вроде `wild(...)`, полностью исключает проверку типов параметров, оставляя программиста открытым перед множеством неприятностей, которые хорошо знакомы программистам на C. В хорошо продуманной программе требуется самое большее несколько функций, для которых типы параметров не определены полностью. Для того, чтобы позаботиться о проверке типов, можно использовать перегруженные функции и функции с параметрами по умолчанию в большинстве тех случаев, когда иначе пришлось бы оставить типы параметров незадаанными. Многоточие необходимо, только если изменяются и число параметров, и тип параметров. Наиболее обычное применение многоточия в задании интерфейса с функциями C-библиотек, которые были определены в то время, когда альтернативы не было:

```
extern int fprintf(FILE*, char* ...); // из
extern int execl(char* ...); // из
extern int abort(...); // из
```

Разберем случай написания функции ошибок, которая получает один целый параметр, указывающий серьезность ошибки, после которого идет произвольное число строк. Идея состоит в том, чтобы составлять сообщение об ошибке с помощью передачи каждого слова как отдельного строкового параметра:

```

void error(int ...);

main(int argc, char* argv[])
{
    switch(argc) {
        case 1:
            error(0,argv[0],0);
            break;
        case 2:
            error(0,argv[0],argv[1],0);
        default:
            error(1,argv[0],»с»,dec(argc-
1),»параметрами»,0);
    }
}

```

Функцию ошибок можно определить так:

```

#include

void error(int n ...)
/*
    «n» с последующим списком char*, оканчи-
    вающихся нулем
*/
{
    va_list ap;
    va_start(ap,n);                // раскрутка arg

    for (;;) {
        char* p = va_arg(ap,char*);
        if(p == 0) break;
        cerr << p << « «;
    }

    va_end(ap);                    // очистка arg

    cerr << «\n»;
    if (n) exit(n);
}

```

Первый из `va_list` определяется и инициализируется вызовом `va_start()`. Макрос `va_start` получает имя `va_list`'а и имя последнего

формального параметра как параметры. Макрос `va_arg` используется для выбора неименованных параметров по порядку. При каждом обращении программист должен задать тип; `va_arg()` предполагает, что был передан фактический параметр, но обычно способа убедиться в этом нет. Перед возвратом из функции, в которой был использован `va_start()`, должен быть вызван `va_end()`. Причина в том, что `va_start()` может изменить стек так, что нельзя будет успешно осуществить возврат; `va_end()` аннулирует все эти изменения.

4.6.9. Указатель на функцию

С функцией можно делать только две вещи: вызывать ее и брать ее адрес. Указатель, полученный взятием адреса функции, можно затем использовать для вызова этой функции. Например:

```
void error(char* p) { /* ... */ }

void (*efct)(char*); // указатель на функцию

void f()
{
    efct = &error; // efct указывает на error
    (*efct) («error»); // вызов error через efct
}
```

Чтобы вызвать функцию через указатель, например, `efct`, надо сначала этот указатель разыменовать, `*efct`. Поскольку операция вызова функции `()` имеет более высокий приоритет, чем операция разыменования `*`, то нельзя писать просто `*efct («error»)`. Это означает `*efct («error»)`, а это ошибка в типе. То же относится и к синтаксису описаний.

Заметьте, что у указателей на функции типы параметров описываются точно так же, как и в самих функциях. В присваиваниях указателя должно соблюдаться точное соответствие полного типа функции. Например:

```
void (*pf)(char*); // указатель на void(char*)
void f1(char*);    // void(char*)
int f2(char*);     // int(char*)
void f3(int*);     // void(int*)

void f()
{
    pf = &f1;      // ok
}
```

```

        pf = &f2;          // ошибка: не подходит воз-
        вращаемый тип
        pf = &f3;          // ошибка: не подходит тип
        параметра

        (*pf) («asdf»);    // ok
        (*pf) (1);         // ошибка: не подходит
        тип параметра

        int i = (*pf) («qwer»); // ошибка: void
        присваивается int'y
    }

```

Правила передачи параметров для непосредственных вызовов функции и для вызовов функции через указатель одни и те же.

Часто, чтобы избежать использования какого-либо неочевидного синтаксиса, бывает удобно определить имя типа указатель-на-функцию. Например:

```

typedef int (*SIG_TYP) ();          // из
typedef void (*SIG_ARG_TYP);
SIG_TYP signal(int, SIG_ARG_TYP);

```

Бывает часто полезен вектор указателей на функцию. Например, система меню для редактора с мышью, которым пользуется автор, реализована с помощью векторов указателей на функции для представления действий. Подробно эту систему здесь описать не получится, но вот общая идея:

```

typedef void (*PF) ();

PF edit_ops[] = { // операции редактирования
    cut, paste, snarf, search
};

PF file_ops[] = { // управление файлом
    open, reshape, close, write
};

```

Затем определяем и инициализируем указатели, определяющие действия, выбранные в меню, которое связано с кнопками (button) мыши:

```

PF* button2 = edit_ops;
PF* button3 = file_ops;

```

В полной реализации для определения каждого пункта меню требуется больше информации. Например, где-то должна храниться строка, задающая текст, который высвечивается. При использовании системы значение кнопок мыши часто меняется в зависимости от ситуации. Эти изменения осуществляются (частично) посредством смены значений указателей кнопок. Когда пользователь выбирает пункт меню, например пункт 3 для кнопки 2, выполняется связанное с ним действие:

```
(button2[3])();
```

Один из способов оценить огромную мощь указателей на функции - это попробовать написать такую систему, не используя их. Меню можно менять в ходе использования программы, внося новые функции в таблицу действий. Во время выполнения можно также легко сконструировать новое меню.

Указатели на функции можно использовать для задания полиморфных подпрограмм, то есть подпрограмм, которые могут применяться к объектам многих различных типов:

```
typedef int (*CFT)(char*,char*);
```

```
int sort(char* base, unsigned n, int sz, CFT cmp)
/*
```

```
Сортирует «n» элементов вектора «base»
в возрастающем порядке
с помощью функции сравнения, указываемой «cmp».
Размер элементов «sz».
```

Очень неэффективный алгоритм: пузырьковая
сортировка

```
*/
{
    for (int i=0; i<n; i++)
        for (int j=i+1; j<n; j++)
            if (cmp(base[j], base[i]) < 0)
                swap(base[i], base[j]);
}
```

```
int cmp2(char*p, char* q) // Сравнивает числа dept
{
    return Puser(p)->dept-Puser(q)->dept;
}
```

Эта программа сортирует и печатает:

```

main ()
{
    sort((char*)heads, 6, sizeof(user), cmp1);
    print_id(heads, 6); // в алфавитном порядке
    cout << «\n»;
    sort((char*)heads, 6, sizeof(user), cmp2);
    print_id(heads, 6); // по порядку подраз-
делений
}

```

Можно взять адрес inline-функции, как, впрочем, и адрес перегруженной функции.

4.7. Макросы

Макросы в С очень важны, но в С++ применяются гораздо меньше. Первое правило относительно них такое: не используйте их, если вы не обязаны это делать. Как было замечено, почти каждый макрос проявляет свой изъян или в языке, или в программе. Если вы хотите использовать макросы, прочитайте, пожалуйста, вначале очень внимательно руководство по вашей реализации С-препроцессора.

Простой макрос определяется так:

```
#define name rest of line
```

Когда name встречается как лексема, оно заменяется на rest of line. Например:

```
named = name
```

после расширения даст:

```
named = rest of line
```

Можно также определить макрос с параметрами. Например:

```
#define mac(a,b) argument1: a argument2: b
```

При использовании mac должно даваться две строки параметра. После расширения mac() они заменяют а и b. Например:

```
expanded = mac(foo bar, yuk yuk)
```

после расширения даст

```
expanded = argument1: foo bar argument2: yuk yuk
```

Макросы обрабатывают строки и о синтаксисе C++ знают очень мало, а о типах C++ или областях видимости - ничего. Компилятор видит только расширенную форму макроса, поэтому ошибка в макросе диагностируется, когда макрос расширен, а не когда он определен. В результате этого возникают непонятные сообщения об ошибках.

Вот такими макросы могут быть вполне:

```
#define Case break; case
#define nl <<«\n»
#define forever for(;;)
#define MIN(a,b) ((a)<(b))?(a):(b))
```

Вот совершенно ненужные макросы:

```
#define PI 3.141593
#define BEGIN {
#define END }
```

А вот примеры опасных макросов:

```
#define SQUARE(a) a*a
#define INCR_xx (xx)++
#define DISP = 4
```

Чтобы увидеть, чем они опасны, попробуйте провести расширения в следующем примере:

```
int xx = 0;                // глобальный счетчик

void f() {
    int xx = 0;            // локальная переменная
    xx = SQUARE(xx+2);     // xx = xx+2*xx+2
    INCR_xx;               // увеличивает локальный xx
    if (a-DISP==b) {       // a-= 4==b
        // ...
    }
}
```

Если вы вынуждены использовать макрос, при ссылке на глобальные имена, используйте операцию разрешения области видимости :: и заключайте вхождения имени параметра макроса в скобки везде, где это возможно (см. MIN выше).

Обратите внимание на различие результатов расширения этих двух макросов:

```
#define m1(a) something(a)    // глубокомысленный комментарий
```

```
#define m2(a) something(a)      /* глубокомыс-
ленный комментарий */
```

например,

```
int a = m1(1)+2;
int b = m2(1)+2;
```

расширяется в

```
int a = something(1)           // глубокомысленный
комментарий+2;
int b = something(1)           /* глубокомысленный
комментарий */+2;
```

С помощью макросов вы можете разработать свой собственный язык. Скорее всего, для всех остальных он будет непостижим. Кроме того, С-препроцессор - очень простой макропроцессор. Когда вы попытаетесь сделать что-либо нетривиальное, вы, вероятно, обнаружите, что сделать это либо невозможно, либо чрезвычайно трудно.

5. ГЛАВА ПЯТАЯ. КЛАССЫ

В этой главе описываются возможности определения новых типов в C++, для которых доступ к данным ограничен заданным множеством функций доступа. Объясняются способы защиты структуры данных, ее инициализации, доступа к ней и, наконец, ее уничтожения. Примеры содержат простые классы для работы с таблицей имен, манипуляции стеком, работу с множеством и реализацию дискриминирующего (то есть «надежного») объединения. Две следующие главы дополняют описание возможностей определения новых типов в C++ и познакомят читателя еще с некоторыми интересными примерами.

5.1. Знакомство и краткий обзор

Предназначение понятия класса, которому посвящены эта и две последующие главы, состоит в том, чтобы предоставить программисту инструмент для создания новых типов, столь же удобных в обращении, сколь и встроенные типы. В идеале тип, определяемый пользователем, способом использования не должен отличаться от встроенных типов, только способом создания.

Тип есть конкретное представление некоторой концепции (понятия). Например, имеющийся в C++ тип `float` с его операциями `+`, `-`, `*` и т.д. обеспечивает ограниченную, но конкретную версию математического понятия действительного числа. Новый тип создается для того, чтобы дать специальное и конкретное определение понятия, которому ничто прямо и очевидно среди встроенных типов не отвечает. Например, в программе, которая работает с телефоном, можно было бы создать тип `trunk_module` (элемент линии), а в программе обработки текстов - тип `list_of_paragraphs` (список параграфов). Как правило, программу, в которой создаются типы, хорошо отвечающие понятиям приложения, понять легче, чем программу, в которой это не делается. Хорошо выбранные типы, определяемые пользователем, делают программу более четкой и короткой. Это также позволяет компилятору обнаруживать недопустимые использования объектов, которые в противном случае останутся необнаруженными до тестирования программы.

В определении нового типа основная идея - отделить несущественные подробности реализации (например, формат данных, которые используются для хранения объекта типа) от тех качеств, которые существенны для его правильного использования (например, полный список функций, которые имеют доступ к данным). Такое разделение

можно описать так, что работа со структурой данных и внутренними административными подпрограммами осуществляется через специальный интерфейс (канализуется).

Эта глава состоит из четырех практически отдельных частей:

Классы и члены. Этот раздел знакомит с основным понятием типа, определяемого пользователем, который называется класс (class). Доступ к объектам класса может ограничиваться набором функций, которые описаны как часть этого класса. Такие функции называются функциями членами. Объекты класса создаются и инициализируются функциями членами, специально для этой цели описанными. Эти функции называются конструкторами. Функция член может быть специальным образом описана для «очистки» каждого классового объекта при его уничтожении. Такая функция называется деструктором.

Интерфейсы и реализации. В этом разделе приводится два примера того, как класс проектируется, реализуется и используется.

Друзья и объединения. В этом разделе приводится много дополнительных подробностей, касающихся классов. В нем показано, как предоставить доступ к закрытой части класса функции, которая не является членом этого класса. Такая функция называется друг (friend). В этом разделе показано также, как определить дискриминирующее объединение.

Конструкторы и деструкторы. Объект может создаваться как автоматический, статический или как объект в свободной памяти. Объект может также быть членом некоторой совокупности (типа вектора или класса), которая в свою очередь может размещаться одним из этих трех способов. Довольно подробно объясняется использование конструкторов и деструкторов.

5.2. Классы и члены

Класс - это определяемый пользователем тип. Этот раздел знакомит с основными средствами определения класса, создания объекта класса, работы с такими объектами и, наконец, уничтожения таких объектов после использования.

5.2.1. Функции члены

Рассмотрим реализацию понятия даты с использованием struct для того, чтобы определить представление даты date и множества функций для работы с переменными этого типа:

```
struct date { int month, day, year; };
```



```

        // дата:      месяц, день, год  }
date today;
void set_date(date*, int, int, int);
void next_date(date*);
void print_date(date*);
// ...

```

Никакой явной связи между функциями и типом данных нет. Такую связь можно установить, описав функции как члены:

```

struct date {
    int month, day, year;

    void set(int, int, int);
    void get(int*, int*, int*);
    void next();
    void print();
};

```

Функции, описанные таким образом, называются функциями членами и могут вызываться только для специальной переменной соответствующего типа с использованием стандартного синтаксиса для доступа к членам структуры. Например:

```

date today;           // сегодня
date my_burthday;     // мой день рождения

void f()
{
    my_burthday.set(30,12,1950);
    today.set(18,1,1985);

    my_burthday.print();
    today.next();
}

```

Поскольку разные структуры могут иметь функции члены с одинаковыми именами, при определении функции члена необходимо указывать имя структуры:

```

void date::next()
{
    if ( ++day > 28 ) {
        // делает сложную часть работы
    }
}

```

В функции члене имена членов могут использоваться без явной ссылки на объект. В этом случае имя относится к члену того объекта, для которого функция была вызвана.

5.2.2. Классы

Описание `date` в предыдущем подразделе дает множество функций для работы с `date`, но не указывает, что эти функции должны быть единственными для доступа к объектам типа `date`. Это ограничение можно наложить используя вместо `struct` `class`:

```
class date {
    int month, day, year;
public:
    void set(int, int, int);
    void get(int*, int*, int*);
    void next();
    void print();
};
```

Метка `public` делит тело класса на две части. Имена в первой, закрытой части, могут использоваться только функциями членами. Вторая, открытая часть, составляет интерфейс к объекту класса. `Struct` - это просто `class`, у которого все члены общие, поэтому функции члены определяются и используются точно так же, как в предыдущем случае. Например:

```
void date::ptinr()           // печатает в записи, принятой в США
{
    cout << month << «/» << day << «/» year;
}
```

Однако функции не члены отгорожены от использования закрытых членов класса `date`. Например:

```
void backdate()
{
    today.day--;             // ошибка
}
```

В том, что доступ к структуре данных ограничен явно описанным списком функций, есть несколько преимуществ. Любая ошибка, которая приводит к тому, что дата принимает недопустимое значение (например, Декабрь 36, 1985) должна быть вызвана кодом функции члена,

поэтому первая стадия отладки, локализация, выполняется еще до того, как программа будет запущена. Это частный случай общего утверждения, что любое изменение в поведении типа `date` может и должно вызываться изменениями в его членах. Другое преимущество - это то, что потенциальному пользователю такого типа нужно будет только узнать определение функций членов, чтобы научиться им пользоваться.

Защита закрытых данных связана с ограничением использования имен членов класса. Это можно обойти с помощью манипуляции адресами, но это уже, конечно, жульничество.

5.2.3. Ссылки на себя

В функции члене на члены объекта, для которого она была вызвана, можно ссылаться непосредственно. Например:

```
class x {
    int m;
public:
    int readm() { return m; }
};

x aa;
x bb;

void f()
{
    int a = aa.readm();
    int b = bb.readm();
    // ...
}
```

В первом вызове члена `member()` `m` относится к `aa.m`, а во втором - к `bb.m`.

Указатель на объект, для которого вызвана функция член, является скрытым параметром функции. На этот неявный параметр можно ссылаться явно как на `this`. В каждой функции класса `x` указатель `this` неявно описан как

```
x* this;
```

и инициализирован так, что он указывает на объект, для которого была вызвана функция член. `this` не может быть описан явно, так как это ключевое слово. Класс `x` можно эквивалентным образом описать так:

```
class x {
    int m;
```

```
public:
    int readm() { return this->m; }
};
```

При ссылке на члены использование `this` излишне. Главным образом `this` используется при написании функций членов, которые манипулируют непосредственно указателями. Типичный пример этого - функция, вставляющая звено в дважды связанный список:

```
class dlink {
    dlink* pre;    // предшествующий
    dlink* suc;    // следующий
public:
    void append(dlink*);
    // ...
};

void dlink::append(dlink* p)
{
    p->suc = suc;    // то есть, p->suc = this->suc
    p->pre = this;   // явное использование this
    suc->pre = p;    // то есть, this->suc->pre = p
    suc = p;        // то есть, this->suc = p
}

dlink* list_head;

void f(dlink*a, dlink *b)
{
    // ...
    list_head->append(a);
    list_head->append(b);
}
```

Цепочки такой общей природы являются основой для списковых классов, которые описываются в Главе 7. Чтобы присоединить звено к списку, необходимо обновить объекты, на которые указывают указатели `this`, `pre` и `suc` (текущий, предыдущий и последующий). Все они типа `dlink`, поэтому функция член `dlink::append()` имеет к ним доступ. Единицей защиты в C++ является `class`, а не отдельный объект класса.

5.2.4. Инициализация

Использование для обеспечения инициализации объекта класса функций вроде `set_date()` (установить дату) неэлегантно и чревато ошибками. Поскольку нигде не утверждается, что объект должен быть инициализирован, то программист может забыть это сделать, или (что приводит, как правило, к столь же разрушительным последствиям) сделать это дважды. Есть более хороший подход: дать возможность программисту описать функцию, явно предназначенную для инициализации объектов. Поскольку такая функция конструирует значения данного типа, она называется конструктором. Конструктор распознается по тому, что имеет то же имя, что и сам класс. Например:

```
class date {
    // ...
    date(int, int, int);
};
```

Когда класс имеет конструктор, все объекты этого класса будут инициализироваться. Если для конструктора нужны параметры, они должны даваться:

```
date today = date(23, 6, 1983);
date xmas(25, 12, 0);           // сокращенная форма
                                // (xmas - рождество)
date my_burthday;               // недопустимо, опущена
на инициализация
```

Часто бывает хорошо обеспечить несколько способов инициализации объекта класса. Это можно сделать, задав несколько конструкторов. Например:

```
class date {
    int month, day, year;
public:
    // ...
    date(int, int, int);        // день месяц год
    date(char*);                // дата в строковом
    // представлении
    date(int);                  // день, месяц и
    // год сегодняшние
    date();                     // дата по
    // умолчанию: сегодня
};
```

Конструкторы подчиняются тем же правилам относительно типов параметров, что и перегруженные функции. Если конструкторы существенно различаются по типам своих параметров, то компилятор при каждом использовании может выбрать правильный:

```
date today(4);
date july4 («Июль 4, 1983»);
date guy («5 Ноя»);
date now;                                     // инициализиру-
ется по умолчанию
```

Заметьте, что функции члены могут быть перегружены без явного использования ключевого слова `overload`. Поскольку полный список функций членов находится в описании класса и, как правило, короткий, то нет никакой серьезной причины требовать использования слова `overload` для предотвращения случайного повторного использования имени.

Размножение конструкторов в примере с `date` типично. При разработке класса всегда есть соблазн обеспечить «все», поскольку кажется проще обеспечить какое-нибудь средство просто на случай, что оно кому-то понадобится или потому, что оно изящно выглядит, чем решить, что же нужно на самом деле. Последнее требует больших размышлений, но обычно приводит к программам, которые меньше по размеру и более понятны. Один из способов сократить число родственных функций - использовать параметры по умолчанию. В случае `date` для каждого параметра можно задать значение по умолчанию, интерпретируемое как «по умолчанию принимать: `today`» (сегодня).

```
class date {
    int month, day, year;
public:
    // ...
    date(int d = 0, int m = 0, int y = 0);
    date(char*);           // дата в строко-
вом представлении
};

date::date(int d, int m, int y)
{
    day = d ? d : today.day;
    month = m ? m : today.month;
    year = y ? y : today.year;
    // проверка, что дата допустимая
    // ...
}
```

Когда используется значение параметра, указывающее «брать по умолчанию», выбранное значение должно лежать вне множества возможных значений параметра. Для дня `day` и месяца `month` ясно, что это так, но для года `year` выбор нуля неочевиден. К счастью, в европейском календаре нет нулевого года. Сразу после 1 г. до н.э. (`year==0`) идет 1 г. н.э. (`year==1`), но для реальной программы это может оказаться слишком тонко.

Объект класса без конструкторов можно инициализировать путем присваивания ему другого объекта этого класса. Это можно делать и тогда, когда конструкторы описаны. Например:

```
date d = today;    // инициализация посредством присваивания
```

По существу, имеется конструктор по умолчанию, определенный как побитовая копия объекта того же класса. Если для класса `X` такой конструктор по умолчанию нежелателен, его можно переопределить конструктором с именем `X(X&)`.

5.2.5. Очистка

Определяемый пользователем тип чаще имеет, чем не имеет, конструктор, который обеспечивает надлежащую инициализацию. Для многих типов также требуется обратное действие, деструктор, чтобы обеспечить соответствующую очистку объектов этого типа. Имя деструктора для класса `X` есть `~X()` («дополнение конструктора»). В частности, многие типы используют некоторый объем памяти из свободной памяти, который выделяется конструктором и освобождается деструктором. Вот, например, традиционный стековый тип, из которого для краткости полностью выброшена обработка ошибок:

```
class char_stack {
    int size;
    char* top;
    char* s;
public:
    char_stack(int sz) { top=s=new
char[size=sz]; }
    ~char_stack() { delete s; } // деструктор
    void push(char c) { *top++ = c; }
    char pop() { return *--top; }
}
```

Когда `char_stack` выходит из области видимости, вызывается деструктор:

```
void f()
{
    char_stack s1(100);
    char_stack s2(200);
    s1.push('a');
    s2.push(s1.pop());
    char ch = s2.pop();
    cout << chr(ch) << «\n»;
}
```

Когда вызывается `f()`, конструктор `char_stack` вызывается для `s1`, чтобы выделить вектор из 100 символов, и для `s2`, чтобы выделить вектор из 200 символов. При возврате из `f()` эти два вектора будут освобождены.

5.2.6. Inline

При программировании с использованием классов очень часто используется много маленьких функций. По сути, везде, где в программе традиционной структуры стояло бы просто какое-нибудь обычное использование структуры данных, дается функция. То, что было соглашением, стало стандартом, который распознает компилятор. Это может страшно понизить эффективность, потому что стоимость вызова функции (хотя и вовсе не высокая по сравнению с другими языками) все равно намного выше, чем пара ссылок по памяти, необходимая для тела функции.

Чтобы справиться с этой проблемой, был разработан аппарат `inline`-функций. Функция член, определенная (а не просто описанная) в описании класса, считается `inline`. Это значит, например, что в функциях, которые используют приведенные выше `char_stack`, нет никаких вызовов функций, кроме тех, которые используются для реализации операций вывода! Другими словами, нет никаких затрат времени выполнения, которые стоит принимать во внимание при разработке класса. Любое, даже самое маленькое действие, можно задать эффективно. Это утверждение снимает аргумент, который чаще всего приводят в пользу открытых членов данных.

Функцию член можно также описать как `inline` вне описания класса. Например:


```

char char_stack {
    int size;
    char* top;
    char* s;
public:
    char pop();
    // ...
};

inline char char_stack::pop()
{
    return *--top;
}

```

5.3. Интерфейсы и реализации

Что представляет собой хороший класс? Нечто, имеющее небольшое и хорошо определенное множество действий. Нечто, что можно рассматривать как «черный ящик», которым манипулируют только посредством этого множества действий. Нечто, чье фактическое представление можно любым мыслимым способом изменить, не повлияв на способ использования множества действий. Нечто, чего можно хотеть иметь больше одного.

Для всех видов контейнеров существуют очевидные примеры: таблицы, множества, списки, векторы, словари и т.д. Такой класс имеет операцию «вставить», обычно он также имеет операции для проверки того, был ли вставлен данный элемент. В нем могут быть действия для осуществления проверки всех элементов в определенном порядке, и, кроме всего прочего, в нем может иметься операция для удаления элемента. Обычно контейнерные (то есть, вмещающие) классы имеют конструкторы и деструкторы.

Скрытие данных и продуманный интерфейс может дать концепция модуля (см. например: «файлы как модули»). Класс, однако, является типом. Чтобы использовать его, необходимо создать объекты этого класса, и таких объектов можно создавать столько, сколько нужно. Модуль же сам является объектом. Чтобы использовать его, его надо только инициализировать, и таких объектов ровно один.

5.3.1. Законченный класс

Программирование без скрытия данных (с применением структур) требует меньшей продуманности, чем программирование со скрытием данных (с использованием классов). Структуру можно определить не слишком задумываясь о том, как ее предполагается использовать. А когда определяется класс, все внимание сосредотачивается на обеспечении нового типа полным множеством операций; это важное смещение акцента. Время, потраченное на разработку нового типа, обычно многократно окупается при разработке и тестировании программы.

Вот пример законченного типа `intset`, который реализует понятие «множество целых»:

```
class intset {
    int cursize, maxsize;
    int *x;
public:
    intset(int m, int n);    // самое большее,
m int'ов в 1..n
    ~intset();

    int member(int t);      // является ли t
элементом?
    void insert(int t);     // добавить «t» в
множество

    void iterate(int& i)    { i = 0; }
    int ok(int& i)         { return i

void error(char* s)
{
    cerr << «set: « << s << «\n»;
    exit(1);
}
```

Класс `intset` используется в `main()`, которая предполагает два целых параметра. Первый параметр задает число случайных чисел, которые нужно сгенерировать. Второй параметр указывает диапазон, в котором должны лежать случайные целые:

```
main(int argc, char* argv[])
{
    if (argc != 3) error(«ожидается два пара-
метра»);
```

```

        int count = 0;
        int m = atoi(argv[1]);           // число
элементов множества
        int n = atoi(argv[2]);           // в диапа-
зоне 1..n
        intset s(m,n);

        while (count maxsize) error(«слишком много
элементов»);
        int i = cursize-1;
        x[i] = t;

        while (i>0 && x[i-1]>x[i]) {
            int t = x[i];                 // пере-
ставить x[i] и [i-1]
            x[i] = x[i-1];
            x[i-1] = t;
            i--;
        }
    }

```

Для нахождения членов используется просто двоичный поиск:

```

int intset::member(int t)    // двоичный поиск
{
    int l = 0;
    int u = cursize-1;

    while (l <= u) {
        int m = (l+u)/2;
        if (t < x[m])
            u = m-1;
        else if (t > x[m])
            l = m+1;
        else
            return 1;                // найдено
    }
    return 0;                        // не найдено
}

```

И наконец, нам нужно обеспечить множество операций, чтобы пользователь мог осуществлять цикл по множеству в некотором порядке, поскольку представление intset от пользователя скрыто. Множе-

ство внутренней упорядоченности не имеет, поэтому мы не можем просто дать возможность обращаться к вектору (завтра я, наверное, реализую `intset` по-другому, в виде связанного списка).

Дается три функции: `iterate()` для инициализации итерации, `ok()` для проверки, есть ли следующий элемент, и `next()` для того, чтобы взять следующий элемент:

```
class intset {
    // ...
    void iterate(int& i)    { i = 0; }
    int ok(int& i)         { return iiterate(var);
    while (set->ok(var)) cout << set->next(var) << «\n»;
}
```

5.4. Друзья и объединения

В этом разделе описываются еще некоторые особенности, касающиеся классов. Показано, как предоставить функции не члену доступ к закрытым членам. Описывается, как разрешать конфликты имен членов, как можно делать вложенные описания классов и как избежать нежелательной вложенности. Обсуждается также, как объекты класса могут совместно использовать члены данные и как использовать указатели на члены. Наконец, приводится пример, показывающий, как построить дискриминирующее (экономное) объединение.

5.4.1. Друзья

Предположим, вы определили два класса, `vector` и `matrix` (вектор и матрица). Каждый скрывает свое представление и предоставляет полный набор действий для манипуляции объектами его типа. Теперь определим функцию, умножающую матрицу на вектор. Для простоты допустим, что в векторе четыре элемента, которые индексируются 0...3, и что матрица состоит из четырех векторов, индексированных 0...3. Допустим также, что доступ к элементам вектора осуществляется через функцию `elem()`, которая осуществляет проверку индекса, и что в `matrix` имеется аналогичная функция. Один подход состоит в определении глобальной функции `multiply()` (перемножить) примерно следующим образом:

```
vector multiply(matrix& m, vector& v);
{
    vector r;
    for (int i = 0; i<3; i++) {    // r[i] = m[i] * v;
```

```

        r.elem(i) = 0;
        for (int j = 0; j<3; j++)
            r.elem(i) += m.elem(i,j) * v.elem(j);
    }
    return r;
}

```

Это своего рода «естественный» способ, но он очень неэффективен. При каждом обращении к `multiply()` `elem()` будет вызываться $4 \cdot (1 + 4 \cdot 3)$ раза.

Теперь, если мы сделаем `multiply()` членом класса `vector`, мы сможем обойтись без проверки индексов при обращении к элементу вектора, а если мы сделаем `multiply()` членом класса `matrix`, то мы сможем обойтись без проверки индексов при обращении к элементу матрицы. Однако членом двух классов функция быть не может. Нам нужно средство языка, предоставляющее функции право доступа к закрытой части класса. Функция не член, получившая право доступа к закрытой части класса, называется другом класса (`friend`). Функция становится другом класса после описания как `friend`. Например:

```

class matrix;

class vector {
    float v[4];
    // ...
    friend vector multiply(matrix&, vector&);
};

class matrix {
    vector v[4];
    // ...
    friend vector multiply(matrix&, vector&);
};

```

Функция друг не имеет никаких особенностей, помимо права доступа к закрытой части класса. В частности, `friend`-функция не имеет указателя `this` (если только она не является полноправным членом-функцией). Описание `friend` - настоящее описание. Оно вводит имя функции в самой внешней области видимости программы и сопоставляется с другими описаниями этого имени. Описание друга может располагаться или в закрытой, или в открытой части описания класса; где именно, значения не имеет.

Теперь можно написать функцию умножения, которая использует элементы векторов и матрицы непосредственно:

```
vector multiply(matrix& m, vector& v);
{
    vector r;
    for (int i = 0; i<3; i++) {    // r[i] = m[i] * v;
        r.v[i] = 0;
        for (int j = 0; j<3; j++)
            r.v[i] += m.v[i][j] * v.v[j];
    }
    return r;
}
```

Есть способы преодолеть эту конкретную проблему эффективно, не используя аппарат friend (можно было бы определить операцию векторного умножения и определить multiply() с ее помощью). Однако существует много задач, которые проще всего решаются, если есть возможность предоставить доступ к закрытой части класса функции, которая не является членом этого класса. В Главе 6 есть много примеров применения friend. Достоинства функций друзей и членов будут обсуждаться позже.

Функция член одного класса может быть другом другого. Например:

```
class x {
    // ...
    void f();
};

class y {
    // ...
    friend void x::f();
};
```

Нет ничего необычного в том, что все функции члены одного класса являются друзьями другого. Для этого есть даже более краткая запись:

```
class x {
    friend class y;
    // ...
};
```

Такое описание friend делает все функции члены класса у друзьями x.

5.4.2. Уточнение имени члена

Иногда полезно делать явное различие между именами членов класса и прочими именами. Для этого используется операция `::` разрешения области видимости:

```
class x {
    int m;
public:
    int readm()          { return x::m; }
    void setm(int m) { x::m = m; }
};
```

В `x::setm()` имя параметра `m` прячет член `m`, поэтому единственный способ сослаться на член - это использовать его уточненное имя `x::m`. Операнд в левой части `::` должен быть именем класса.

Имя с префиксом `::` (просто) должно быть глобальным именем. Это особенно полезно для того, чтобы можно было использовать часто-употребимые имена вроде `read`, `put` и `open` как имена функций членов, не теряя при этом возможности обращаться к той версии функции, которая не является членом. Например:

```
class my_file {
    // ...
public:
    int open(char*, char*);

    int my_file::open(char* name, char* spec)
    {
        // ...
        if (::open(name, flag)) { // использовать
open() из UNIX(2)
            // ...
        }
        // ...
    }
}
```

5.4.3. Вложенные классы

Описание класса может быть вложенным. Например:

```
class set {
    struct setmem {
        int mem;
```

```

        setmem* next;
        setmem(int m, setmem* n) { mem=m;
next=n; }
    };
    setmem* first;
public:
    set() { first=0; }
    insert(int m) { first = new set-
mem(m,first); }
    // ...
};

```

Если только вложенный класс не является очень простым, в таком описании трудно разобраться. Кроме того, вложение классов - это не более, чем соглашение о записи, поскольку вложенный класс не является скрытым в области видимости лексически охватывающего класса:

```

class set {
    struct setmem {
        int mem;
        setmem* next;
        setmem(int m, setmem* n)
    };
    // ...
};

setmem::setmem(int m, setmem* n) { mem=m, next=n}
setmem m1(1,0);

```

Такая запись, как `set::setmem::setmem()`, не является ни необходимой, ни допустимой. Единственный способ скрыть имя класса - это сделать это с помощью метода файлы-как-модули. Большую часть нетривиальных классов лучше описывать отдельно:

```

class setmem {
    friend class set;           // доступ только с
помощью членов set
    int mem;
    setmem* next;
    setmem(int m, setmem* n) { mem=m; next=n;
}

};

class set {

```



```

        setmem* first;
public:
    set() { first=0; }
    insert(int m) { first = new setmem(m,first);}
    // ...
};

```

5.4.4. Статические члены

Класс - это тип, а не объект данных, и в каждом объекте класса имеется своя собственная копия данных, членов этого класса. Однако некоторые типы наиболее элегантно реализуются, если все объекты этого типа могут совместно использовать (разделять) некоторые данные. Предпочтительно, чтобы такие разделяемые данные были описаны как часть класса. Например, для управления задачами в операционной системе или в ее модели часто бывает полезен список всех задач:

```

class task {
    // ...
    task* next;
    static task* task_chain;
    void shedule(int);
    void wait(event);
    // ...
};

```

Описание члена `task_chain` (цепочка задач) как `static` обеспечивает, что он будет всего лишь один, а не по одной копии на каждый объект `task`. Он все равно остается в области видимости класса `task`, и «извне» доступ к нему можно получить, только если он был описан как `public`. В этом случае его имя должно уточняться именем его класса:

```

task::task_chain

```

В функции члене на него можно ссылаться просто `task_chain`. Использование статических членов класса может заметно снизить потребность в глобальных переменных.

5.4.5. Структуры и объединения

По определению `struct` - это просто класс, все члены которого общие, то есть

```

struct s { ...

```

есть просто сокращенная запись

```
class s { public: ...
```

Структуры используются в тех случаях, когда скрытие данных неуместно.

Именованное объединение определяется как `struct`, в которой все члены имеют один и тот же адрес. Если известно, что в каждый момент времени нужно только одно значение из структуры, то объединение может сэкономить пространство. Например, можно определить объединение для хранения лексических символов С-компилятора:

```
union tok_val {
    char* p;           // строка
    char v[8];         // идентификатор (максимум 8 char)
    long i;             // целые значения
    double d;           // значения с плавающей
                        // точкой
};
```

Сложность состоит в том, что компилятор, вообще говоря, не знает, какой член используется в каждый данный момент, поэтому надлежащая проверка типа невозможна. Например:

```
void strange(int i)
{
    tok_val x;
    if (i)
        x.p = «2»;
    else
        x.d = 2;
    sqrt(x.d);           // ошибка если i != 0
}
```

Кроме того, объединение, определенное так, как это, нельзя инициализировать. Например:

```
tok_val curr_val = 12; // ошибка: int присваивается
                        tok_val'y
```

является недопустимым. Для того чтобы это преодолеть, можно воспользоваться конструкторами:

```
union tok_val {
    char* p;           // строка
```

```

        char v[8];           // идентификатор (максим-
мум 8 char)
        long i;              // целые значения
        double d;            // значения с плавающей
точкой

        tok_val(char*);      // должна выбрать меж-
ду p и v
        tok_val(int ii)      { i = ii; }
        tok_val()             { d = dd; }
    };

```

Это позволяет справляться с теми ситуациями, когда типы членов могут быть разрешены по правилам для перегрузки имени функции. Например:

```

void f()
{
    tok_val a = 10;          // a.i = 10
    tok_val b = 10.0;        // b.d = 10.0
}

```

Когда это невозможно (для таких типов, как `char*` и `char[8]`, `int` и `char` и т.п.), нужный член может быть найден только посредством анализа инициализатора в ходе выполнения или с помощью задания дополнительного параметра. Например:

```

tok_val::tok_val(char* pp)
{
    if (strlen(pp) <= 8)
        strncpy(v, pp, 8); // короткая строка
    else
        p = pp;            // длинная строка
}

```

Таких ситуаций лучше избегать.

Использование конструкторов не предохраняет от такого случайного неправильного употребления `tok_val`, когда сначала присваивается значение одного типа, а потом рассматривается как другой тип. Эта проблема решается встраиванием объединения в класс, который отслеживает, какого типа значение помещается:

```

class tok_val {
    char tag;
    union {

```

```

char* p;
char v[8];
long i;
double d;
};
int check(char t, char* s)
    { if (tag!=t) { error(s); return 0; }
return 1; }
public:
    tok_val(char* pp);
    tok_val(long ii)  { i=ii; tag='I'; }
    tok_val(double dd) { d=dd; tag='D'; }

    long& ival()      { check('I',»ival»);
return i; }
    double& fval()    { check('D',»fval»);
return d; }
    char*& sval()     { check('S',»sval»);
return p; }
    char*  id()       { check('N',»id»);
return v; }
};

```

Конструктор, получающий строковый параметр, использует для копирования коротких строк `strncpy()`. `strncpy()` похожа на `strcpy()`, но получает третий параметр, который указывает, сколько символов должно копироваться:

```

tok_val::tok_val(char* pp)
{
    if (strlen(pp) <= 8) { // короткая строка
        tag = 'N'
        strncpy(v,pp,8); // скопировать 8 символов
    }
    else { // длинная строка
        tag = 'S'
        p = pp; // просто сохранить указатель
    }
}

```

Тип `tok_val` можно использовать так:

```

void f()
{

```

```

        tok_val t1(«short»); // короткая, присвоить v
        tok_val t2(«long string»); // длинная
строка, присвоить p
        char s[8];
        strncpy(s,t1.id(),8);           // ok
        strncpy(s,t2.id(),8);           // проверка
check() не пройдет
    }

```

5.5. Конструкторы и деструкторы

Если у класса есть конструктор, то он вызывается всегда, когда создается объект класса. Если у класса есть деструктор, то он вызывается всегда, когда объект класса уничтожается. Объекты могут создаваться как:

[1] автоматический объект: создается каждый раз, когда его описание встречается при выполнении программы, и уничтожается каждый раз при выходе из блока, в котором оно появилось;

[2] статический объект: создается один раз, при запуске программы, и уничтожается один раз, при ее завершении;

[3] объект в свободной памяти: создается с помощью операции new и уничтожается с помощью операции delete;

[4] объект член: как объект другого класса или как элемент вектора.

Объект также может быть сконструирован с помощью явного применения конструктора в выражении, в этом случае он является автоматическим объектом. В следующих подразделах предполагается, что объекты принадлежат классу, имеющему конструктор и деструктор. Примером может служить класс table, приведенный выше.

5.5.1. Предостережение

Если x и y - объекты класса cl , то $x=y$ в стандартном случае означает побитовое копирование y в x . Такая интерпретация присваивания может привести к изумляющему (и обычно нежелательному) результату, если оно применяется к объектам класса, для которого определены конструктор и деструктор. Например:

```

class char_stack {
    int size;
    char* top;
    char* s;
public:

```

```

        char_stack(int      sz)      {      top=s=new
char[size=sz]; }
        ~char_stack()      { delete s; }      // дест-
руктор
        void push(char c)  { *top++ = c; }
        char pop()         { return *--top; }

};

void h()
{
    char_stack s1(100);
    char_stack s2 = s1;  // неприятность
    char_stack s3(99);
    s3 = s2;             // неприятность
}

```

Здесь конструктор `char_stack::char_stack()` вызывается дважды: для `s1` и для `s3`. Для `s2` он не вызывается, поскольку эта переменная инициализируется присваиванием. Однако деструктор `char_stack::~~char_stack()` вызывается трижды: для `s1`, `s2` и `s3`! Кроме того, по умолчанию действует интерпретация присваивания как побитовое копирование, поэтому в конце `h()` каждый из `s1`, `s2` и `s3` будет содержать указатель на вектор символов, размещенный в свободной памяти при создании `s1`. Не останется никакого указателя на вектор символов, выделенный при создании `s3`. Таких отклонений можно избежать (см. Главу 6).

5.5.2. Статическая память

Рассмотрим следующее:

```

table tbl1(100);

void f() {
    static table tbl2(200);
}

main()
{
    f();
}

```

Здесь конструктор `table::table()`, будет вызываться дважды: один раз для `tbl1` и один раз для `tbl2`. Деструктор `table::~~table()` также будет вызван дважды: для уничтожения `tbl1` и `tbl2` после выхода из `main()`. Конструкторы для глобальных статических объектов в файле выполняются в том порядке, в котором встречаются описания; деструкторы вызываются в обратном порядке. Не определено, вызывается ли конструктор для локального статического объекта, если функция, в которой этот объект описан, не вызывается. Если конструктор для локального статического объекта вызывается, то он вызывается после того, как вызваны конструкторы для лексически предшествующих ему глобальных статических объектов.

Параметры конструкторов для статических объектов должны быть константными выражениями:

```
void g(int a)
{
    static table t(a);  // ошибка
}
```

Традиционно выполнение `main()` считалось выполнением программы. Так никогда не было, даже в С, но только размещение статических объектов класса с конструктором и/или деструктором дают программисту простой и очевидный способ задания того, что будет выполняться до и/или после вызова `main()`.

Вызов конструкторов и деструкторов для статических объектов играет в С++ чрезвычайно важную роль. Это способ обеспечить надлежащую инициализацию и очистку структур данных в библиотеках. Рассмотрим, откуда берутся `cin`, `cout` и `cerr`. Где они получают инициализацию? И, что самое главное, поскольку потоки вывода имеют внутренние буферы символов, как же эти буферы становятся заполненными? Простой и очевидный ответ, что эта работа осуществляется соответствующими конструкторами и деструкторами до и после выполнения `main()`. Для инициализации и очистки библиотечных средств есть возможности, альтернативные использованию конструкторов и деструкторов. Все они или очень специальные, или очень уродливые.

Если программа завершается с помощью функции `exit()`, то деструкторы для статических объектов будут вызваны, а если она завершается с помощью `abort()`, то не будут. Заметьте, что это подразумевает, что `exit()` не завершает программу мгновенно. Вызов `exit()` в деструкторе может привести к бесконечной рекурсии.

Иногда, когда вы разрабатываете библиотеку, необходимо или просто удобно создать тип с конструктором и деструктором, предна-

значенными только для одного: инициализировать и очистить. Такой тип обычно используется только с одной целью, для размещения статического объекта так, чтобы вызывались конструктор и деструктор.

5.5.3. Свободная память

Рассмотрим:

```
main() {
    table* p = new table(100);
    table* q = new table(200);
    delete p;
    delete p;                      // возможно, ошибка
}
```

Конструктор `table::table()` будет вызван дважды, как и деструктор `table::~~table()`. То, что C++ не дает никаких гарантий, что для объекта, созданного с помощью `new`, когда-либо будет вызван деструктор, ничего не значит. В предыдущей программе `q` не уничтожается, а `p` уничтожается дважды! Программист может счесть это ошибкой, а может и не счесть, в зависимости от типа `p` и `q`. Обычно то, что объект не уничтожается, является не ошибкой, а просто лишней тратой памяти. Уничтожение `p` дважды будет, как правило, серьезной ошибкой. Обычно результатом применения `delete` дважды к одному указателю приводит к бесконечному циклу в подпрограмме управления свободной памятью, но определение языка не задает поведение в таком случае, и оно зависит от реализации.

Пользователь может определить новую реализацию операций `new` и `delete`. Можно также определить способ взаимодействия конструктора или деструктора с операциями `new` и `delete`.

5.5.4. Объекты класса и члены

Рассмотрим

```
class classdef {
    table members;
    int no_of_members;
    // ...
    classdef(int size);
    ~classdef();
};
```


Очевидное намерение состоит в том, что `classdef` должен содержать таблицу длиной `size` из членов `member`, а сложность в том, как сделать так, чтобы конструктор `table::table()` вызывался с параметром `size`. Это делается примерно так:

```
classdef::classdef(int size)
: members(size)
{
    no_of_members = size;
    // ...
}
```

Параметры для конструктора члена `member` (здесь это `table::table()`) помещаются в определение (не в описание) конструктора класса, вмещающего его (здесь это `classdef::classdef()`). После этого конструктор члена вызывается перед телом конструктора, задающего его список параметров.

Если есть еще члены, которым нужны списки параметров для конструкторов, их можно задать аналогично. Например:

```
class classdef {
    table members;
    table friends;
    int no_of_members;
    // ...
    classdef(int size);
    ~classdef();
};
```

Список параметров для членов разделяется запятыми (а не двоеточиями), и список инициализаторов для членов может представляться в произвольном порядке:

```
classdef::classdef(int size)
: friends(size), members(size)
{
    no_of_members = size;
    // ...
}
```

Порядок, в котором вызываются конструкторы, не определен, поэтому не рекомендуется делать списки параметров с побочными эффектами:

```
classdef::classdef(int size)
```

```

        : friends(size=size/2), members(size); // дурной
СТИЛЬ
    {
        no_of_members = size;
        // ...
    }

```

Если конструктору для члена не нужно ни одного параметра, то никакого списка параметров задавать не надо. Например, поскольку `table::table` был определен с параметром по умолчанию 15, следующая запись является правильной:

```

classdef::classdef(int size)
: members(size)
{
    no_of_members = size;
    // ...
}

```

и размер `size` таблицы `friend`ов будет равен 15.

Когда объект класса, содержащий объект класса, (например, `classdef`) уничтожается, первым выполняется тело собственного деструктора объекта, а затем выполняются деструкторы членов.

Рассмотрим традиционную альтернативу тому, чтобы иметь объекты класса как члены, иметь члены указатели и инициализировать их в конструкторе:

```

class classdef {
    table* members;
    table* friends;
    int no_of_members;
    // ...
    classdef(int size);
    ~classdef();
};

classdef::classdef(int size)
{
    members = new table(size);
    friends = new table; // размер
таблицы по умолчанию
    no_of_members = size;
    // ...
}

```

Так как таблицы создавались с помощью `new`, они должны уничтожаться с помощью `delete`:

```
classdef::~~classdef()
{
    // ...
    delete members;
    delete friends;
}
```

Раздельно создаваемые объекты вроде этих могут оказаться полезными, но учтите, что `members` и `friends` указывают на отдельные объекты, что требует для каждого из них действие по выделению памяти и ее освобождению. Кроме того, указатель плюс объект в свободной памяти занимают больше места, чем объект член.

5.5.5. Векторы объектов класса

Чтобы описать вектор объектов класса, имеющего конструктор, этот класс должен иметь конструктор, который может вызываться без списка параметров. Нельзя использовать даже параметры по умолчанию. Например:

```
table tblvec[10];
```

будет ошибкой, так как для `table::table()` требуется целый параметр. Нет способа задать параметры конструктора в описании вектора. Чтобы можно было описывать вектор таблиц `table`, можно модифицировать описание `table`, например, так:

```
class table {
    // ...
    void init(int sz); // как старый конструктор
public:
    table(int sz)      // как раньше, но
    без «по умолчанию»
    { init(sz); }
    table()            // по умолчанию
    { init(15); }
}
```

Когда вектор уничтожается, деструктор должен вызываться для каждого элемента этого вектора. Для векторов, которые не были размещены с помощью `new`, это делается неявно. Однако для векторов в свободной памяти это не может быть сделано неявно, поскольку ком-

пилятор не может отличить указатель на один объект от указателя на первый элемент вектора объектов. Например:

```
void f()
{
    table* t1 = new table;
    table* t2 = new table[10];
    delete t1; // одна таблица
    delete t2; // неприятность: 10 таблиц
}
```

В этом случае длину вектора должен задавать программист:

```
void g(int sz)
{
    table* t1 = new table;
    table* t2 = new table[sz];
    delete t1;
    delete[] t2;
}
```

Но почему же компилятор не может найти число элементов вектора из объема выделенной памяти? Потому, что распределитель свободной памяти не является частью языка и может быть задан программистом.

5.5.6. Небольшие объекты

Когда вы используете много небольших объектов, размещаемых в свободной памяти, то вы можете обнаружить, что ваша программа тратит много времени, выделяя и освобождая память под эти объекты. Первое решение - это обеспечить более хороший распределитель памяти общего назначения, второе для разработчика классов состоит в том, чтобы взять под контроль управление свободной памятью для объектов некоторого класса с помощью подходящих конструкторов и деструкторов.

Рассмотрим класс `name`, который использовался в примерах `table`. Его можно было бы определить так:

```
struct name {
    char* string;
    name* next;
    double value;
    name(char*, double, name*);
    ~name();
};
```

Программист может воспользоваться тем, что размещение и освобождение объектов заранее известного размера может обрабатываться гораздо эффективнее (и по памяти, и по времени), чем с помощью общей реализации `new` и `delete`. Общая идея состоит в том, чтобы предварительно разместить «куски» из объектов `name`, а затем сцеплять их, чтобы свести выделение и освобождение к простым операциям над связанным списком. Переменная `nfree` является вершиной списка неиспользованных `name`:

```
const NALL = 128;
name* nfree;
```

Распределитель, используемый операцией `new`, хранит размер объекта вместе с объектом, чтобы обеспечить правильную работу операции `delete`. С помощью распределителя, специализированного для типа, можно избежать этих накладных расходов. Например, на машине автора данного пособия следующий распределитель использует для хранения `name` 16 байт, тогда как для стандартного распределителя свободной памяти нужно 20 байт. Вот как это можно сделать:

```
name::name(char* s, double v, name* n)
{
    register name* p = nfree;          // сначала
выделить

    if (p)
        nfree = p->next;
    else {                               // выде-
лить и сцепить
        name* q = (name*)new char[
NALL*sizeof(name) ];
        for (p=nfree=&q[NALL-1]; qnext = p-1;
            (p+1)->next = 0;
        )

        this = p;                       // затем
инициализировать
        string = s;
        value = v;
        next = n;
    }
```

Присвоение указателю `this` информирует компилятор о том, что программист взял себе управление, и что не надо использовать стандартный механизм распределения памяти. Конструктор `name::name()` обрабатывает только тот случай, когда `name` размещается посредством `new`, но для большей части типов это всегда так. Ниже объясняется, как написать конструктор для обработки как размещения в свободной памяти, так и других видов размещения.

Заметьте, что просто как

```
name* q = new name[NULL];
```

память выделять нельзя, поскольку это приведет к бесконечной рекурсии, когда `new` вызовет `name::name()`.

Освобождение памяти обычно тривиально:

```
name::~~name()
{
    next = nfree;
    nfree = this;
    this = 0;
}
```

Присваивание указателю `this` 0 в деструкторе обеспечивает, что стандартный распределитель памяти не используется.

5.5.7. Предостережение

Когда в конструкторе производится присваивание указателю `this`, значение `this` до этого присваивания не определено. Таким образом, ссылка на член до этого присваивания не определена и скорее всего приведет к катастрофе. Имеющийся компилятор не пытается убедиться в том, что присваивание указателю `this` происходит на всех траекториях выполнения:

```
mytype::mytype(int i)
{
    if (i) this = mytype_alloc();
    // присваивание членам
};
```

откомпилируется, и при `i==0` никакой объект размещен не будет.

Конструктор может определить, был ли он вызван операцией `new`, или нет. Если он вызван `new`, то указатель `this` на входе имеет нулевое значение, в противном случае `this` указывает на пространство, уже выделенное для объекта (например, на стек). Поэтому можно просто на-

писать конструктор, который выделяет память, если (и только если) он был вызван через `new`. Например:

```
mytype::mytype(int i)
{
    if (this == 0) this = mytype_alloc();
    // присваивание членам
};
```

Эквивалентного средства, которое позволяет деструктору решить вопрос, был ли его объект создан с помощью `new`, не имеется, как нет и средства, позволяющего ему узнать, вызвала ли его `delete`, или он вызван объектом, выходящим из области видимости. Если для пользователя это существенно, то он может сохранить где-то соответствующую информацию для деструктора. Другой способ пользователь обеспечивает, что объекты этого класса размещаются только соответствующим образом. Если удастся справиться с первой проблемой, то второй способ интереса не представляет.

Если тот, кто реализует класс, является одновременно и его единственным пользователем, то имеет смысл упростить, исходя из предположений о его использовании. Когда класс разрабатывается для более широкого использования, таких допущений, как правило, лучше избегать.

5.5.8. Объекты переменного размера

Когда пользователь берет управление распределением и освобождением памяти, он может конструировать объекты, размер которых во время компиляции недетерминирован. В предыдущих примерах вмещающие (или контейнерные - перев.) классы `vector`, `stack`, `intset` и `table` реализовывались как структуры доступа фиксированного размера, содержание указатели на реальную память. Это подразумевает, что для создания таких объектов в свободной памяти необходимо две операции по выделению памяти и что любое обращение к хранимой информации будет содержать дополнительную косвенную адресацию. Например:

```
class char_stack {
    int size;
    char* top;
    char* s;
public:
    char_stack(int      sz)      {      top=s=new
char[size=sz]; }
```

```

        ~char_stack()          { delete s; }          //
деструктор
        void push(char c)      { *top++ = c; }
        char pop()             { return *--top; }
    };

```

Если каждый объект класса размещается в свободной памяти, это делать не нужно. Вот другой вариант:

```

class char_stack {
    int size;
    char* top;
    char s[1];
public:
    char_stack(int sz);
    void push(char c) { *top++ = c; }
    char pop()        { return *--top; }
};

char_stack::char_stack(int sz)
{
    if (this) error(«стек не в свободной памяти»);
    if (sz < 1) error(«размер стека < 1»);
    this      = (char_stack*) new
char[sizeof(char_stack)+sz-1];
    size = sz;
    top = s;
}

```

Заметьте, что деструктор больше не нужен, поскольку память, которую использует char_stack, может освободить delete без всякого содействия со стороны программиста.

6. ГЛАВА ШЕСТАЯ. ПЕРЕГРУЗКА ОПЕРАЦИЙ

В этой главе описывается аппарат, предоставляемый в C++ для перегрузки операций. Программист может определять смысл операций при их применении к объектам определенного класса. Кроме арифметических, можно определять еще и логические операции, операции сравнения, вызова () и индексирования [], а также можно переопределять присваивание и инициализацию. Можно определить явное и неявное преобразование между определяемыми пользователем и основными типами. Показано, как определить класс, объект которого не может быть никак иначе скопирован или уничтожен, кроме как специальными определенными пользователем функциями.

6.1. Введение

Часто программы работают с объектами, которые являются конкретными представлениями абстрактных понятий. Например, тип данных `int` в C++ вместе с операциями `+`, `-`, `*`, `/` и т.д. предоставляет реализацию (ограниченную) математического понятия целых чисел. Такие понятия обычно включают в себя множество операций, которые кратко, удобно и привычно представляют основные действия над объектами. К сожалению, язык программирования может непосредственно поддерживать лишь очень малое число таких понятий. Например, такие понятия, как комплексная арифметика, матричная алгебра, логические сигналы и строки не получили прямой поддержки в C++. Классы дают средство спецификации в C++ представления неэлементарных объектов вместе с множеством действий, которые могут над этими объектами выполняться. Иногда определение того, как действуют операции на объекты классов, позволяет программисту обеспечить более общепринятую и удобную запись для манипуляции объектами классов, чем та, которую можно достичь, используя лишь основную функциональную запись. Например:

```
class complex {
    double re, im;
public:
    complex(double r, double i) { re=r; im=i; }
    friend complex operator+(complex, complex);
    friend complex operator*(complex, complex);
};
```

определяет простую реализацию понятия комплексного числа, в которой число представляется парой чисел с плавающей точкой двойной точности, работа с которыми осуществляется посредством операций $+$ и $*$ (и только). Программист задает смысл операций $+$ и $*$ с помощью определения функций с именами `operator+` и `operator*`. Если, например, даны `b` и `c` типа `complex`, то `b+c` означает (по определению) `operator+(b,c)`. Теперь есть возможность приблизить общепринятую интерпретацию комплексных выражений. Например:

```
void f()
{
    complex a = complex(1, 3.1);
    complex b = complex(1.2, 2);
    complex c = b;

    a = b+c;
    b = b+c*a;
    c = a*b+complex(1,2);
}
```

Выполняются обычные правила приоритетов, поэтому второй оператор означает `b=b+(c*a)`, а не `b=(b+c)*a`.

6.2. Функции операции

Можно описывать функции, определяющие значения следующих операций:

<code>+</code>	<code>-</code>	<code>*</code>	<code>/</code>	<code>%</code>	<code>^</code>	<code>&</code>	<code> </code>	<code>~</code>	<code>!</code>
<code>=</code>	<code><</code>	<code>></code>	<code>+=</code>	<code>-=</code>	<code>*=</code>	<code>/=</code>	<code>%=</code>	<code>^=</code>	<code>&=</code>
<code> =</code>	<code><<</code>	<code>>></code>	<code>>>=</code>	<code><<=</code>	<code>==</code>	<code>!=</code>	<code><=</code>	<code>>=</code>	<code>&&</code>
<code> </code>	<code>++</code>	<code>--</code>	<code>[]</code>	<code>()</code>	<code>new</code>	<code>delete</code>			

Последние четыре - это индексирование, вызов функции, выделение свободной памяти и освобождение свободной памяти. Изменить приоритеты перечисленных операций невозможно, как невозможно изменить и синтаксис выражений. Нельзя, например, определить унарную операцию `%` или бинарную `!`. Невозможно определить новые лексические символы операций, но в тех случаях, когда множество операций недостаточно, вы можете использовать запись вызова функции. Используйте, например, не `**`, а `pow()`. Эти ограничения могут показаться слишком строгими, но более гибкие правила могут очень легко привести к неоднозначностям. Например, на первый взгляд определе-

ние операции `**`, означающей возведение в степень, может показаться очевидной и простой задачей, но подумайте еще раз. Должна ли `**` связываться влево (как в Фортране) или вправо (как в Алголе)? Выражение `a**p` должно интерпретироваться как `a*(p)` или как `(a)**(p)`?

Имя функции операции есть ключевое слово `operator` (то есть, операция), за которым следует сама операция, например, `operator<<`. Функция операция описывается и может вызываться так же, как любая другая функция. Использование операции - это лишь сокращенная запись явного вызова функции операции. Например:

```
void f(complex a, complex b)
{
    complex c = a + b;           // сокращенная
запись                          complex d = operator+(a,b); // явный вызов
    }
}
```

При наличии предыдущего описания `complex` оба инициализатора являются синонимами.

6.2.1. Бинарные и унарные операции

Бинарная операция может быть определена или как функция член, получающая один параметр, или как функция друг, получающая два параметра. Таким образом, для любой бинарной операции `@ aa@bb` может интерпретироваться или как `aa.operator@(bb)`, или как `operator@(aa,bb)`. Если определены обе, то `aa@bb` является ошибкой. Унарная операция, префиксная или постфиксная, может быть определена или как функция член, не получающая параметров, или как функция друг, получающая один параметр. Таким образом, для любой унарной операции `@ aa@` или `@aa` может интерпретироваться или как `aa.operator@()`, или как `operator@(aa)`. Если определено и то, и другое, то и `aa@` и `@aa` являются ошибками. Рассмотрим следующие примеры:

```
class X {
    // друзья

    friend X operator-(X);      // унарный минус
    friend X operator-(X,X);   // бинарный минус
    friend X operator-();      // ошибка: нет
операндов                     friend X operator-(X,X,X); // ошибка: тер-
нарное
```

```
// члены (с неявным первым параметром: this)

X* operator&(); // унарное & (взятие адреса)
X operator&(X); // бинарное & (операция И)
X operator&(X,X); // ошибка: тернарное

};
```

Когда операции ++ и -- перегружены, префиксное использование и постфиксное различить невозможно.

6.2.2. Предопределенные значения операций

Относительно смысла операций, определяемых пользователем, не делается никаких предположений. В частности, поскольку не предполагается, что перегруженное = реализует присваивание ее первому операнду, не делается никакой проверки, чтобы удостовериться, является ли этот операнд lvalue.

Значения некоторых встроенных операций определены как равносильные определенным комбинациям других операций над теми же аргументами. Например, если `a` является `int`, то `++a` означает `a+=1`, что в свою очередь означает `a=a+1`. Такие соотношения для определенных пользователем операций не выполняются, если только не случилось так, что пользователь сам определил их таким образом. Например, определение `operator+=()` для типа `complex` не может быть выведено из определений `complex::operator+()` и `complex::operator=()`.

По историческому совпадению операции `=` и `&` имеют предопределенный смысл для объектов классов. Никакого элегантного способа «не определить» эти две операции не существует. Их можно, однако, сделать недееспособными для класса `X`. Можно, например, описать `X::operator&()`, не задав ее определения. Если где-либо будет браться адрес объекта класса `X`, то компоновщик обнаружит отсутствие определения `*1`. Или, другой способ, можно определить `X::operator&()` так, чтобы вызывалась ошибка во время выполнения.

6.2.3. Операции и определяемые пользователем типы

Функция операция должна или быть членом, или получать в качестве параметра по меньшей мере один объект класса (функциям, которые переопределяют операции `new` и `delete`, это делать не обязательно). Это правило гарантирует, что пользователь не может изменить

смысл никакого выражения, не включающего в себя определенного пользователем типа. В частности, невозможно определить функцию, которая действует исключительно на указатели.

Функция операция, первым параметром которой предполагается основной тип, не может быть функцией членом. Рассмотрим, например, сложение комплексной переменной `aa` с целым `2`: `aa+2`, при подходящим образом описанной функции члене, может быть проинтерпретировано как `aa.operator+(2)`, но с `2+aa` это не может быть сделано, потому что нет такого класса `int`, для которого можно было бы определить `+` так, чтобы это означало `2.operator+(aa)`. Даже если бы такой тип был, то для того, чтобы обработать и `2+aa` и `aa+2`, понадобилось бы две различных функции члена. Так как компилятор не знает смысла `+`, определенного пользователем, то не может предполагать, что он коммутативен, и интерпретировать `2+aa` как `aa+2`. С этим примером могут легко справиться функции друзья.

Все функции операции по определению перегружены. Функция операция задает новый смысл операции в дополнение к встроенному определению, и может существовать несколько функций операций с одним и тем же именем, если в типах их параметров имеются отличия, различимые для компилятора, чтобы он мог различать их при обращении.

6.3. Определяемое преобразование типа

Приведенная во введении реализация комплексных чисел слишком ограничена, чтобы она могла устроить кого-либо, поэтому ее нужно расширить. Это будет в основном повторением описанных выше методов. Например:

```
class complex {
    double re, im;
public:
    complex(double r, double i) { re=r; im=i; }

    friend complex operator+(complex, complex);
    friend complex operator+(complex, double);
    friend complex operator+(double, complex);

    friend complex operator-(complex, complex);
    friend complex operator-(complex, double);
    friend complex operator-(double, complex);
```

```

        complex operator-()      // унарный -

        friend complex operator*(complex, com-
plex);
        friend complex operator*(complex, double);
        friend complex operator*(double, complex);

        // ...
    };

```

Теперь, имея описание `complex`, мы можем написать:

```

void f()
{
    complex a(1,1), b(2,2), c(3,3), d(4,4), e(5,5);
    a = -b-c;
    b = c*2.0*c;
    c = (d+e)*a;
}

```

Но писать функцию для каждого сочетания `complex` и `double`, как это делалось выше для `operator+()`, невыносимо нудно. Кроме того, близкие к реальности средства комплексной арифметики должны предоставлять по меньшей мере дюжину таких функций; посмотрите, например, на тип `complex`.

6.3.1. Конструкторы

Альтернативу использованию нескольких функций (перегруженных) составляет описание конструктора, который по заданному `double` создает `complex`. Например:

```

class complex {
    // ...
    complex(double r) { re=r; im=0; }
};

```

Конструктор, требующий только один параметр, не обязательно вызывать явно:

```

complex z1 = complex(23);
complex z2 = 23;

```

И `z1`, и `z2` будут инициализированы вызовом `complex(23)`.

Конструктор - это предписание, как создавать значение данного типа. Когда требуется значение типа и когда такое значение может быть создано конструктором, тогда, если такое значение дается для присваивания, вызывается конструктор. Например, класс `complex` можно было бы описать так:

```
class complex {
    double re, im;
public:
    complex(double r, double i = 0) { re=r; im=i; }

    friend complex operator+(complex, complex);
    friend complex operator*(complex, complex);
};
```

и действия, в которые будут входить переменные `complex` и целые константы, стали бы допустимы. Целая константа будет интерпретироваться как `complex` с нулевой мнимой частью. Например, `a=b*2` означает:

```
a=operator*( b, complex( double(2), double(0)
) )
```

Определенное пользователем преобразование типа применяется неявно только тогда, когда оно является единственным.

Объект, сконструированный с помощью явного или неявного вызова конструктора, является автоматическим и будет уничтожен при первой возможности, обычно сразу же после оператора, в котором он был создан.

6.3.2. Операции преобразования

Использование конструктора для задания преобразования типа является удобным, но имеет следствия, которые могут оказаться нежелательными:

[1] не может быть неявного преобразования из определенного пользователем типа в основной тип (поскольку основные типы не являются классами);

[2] невозможно задать преобразование из нового типа в старый, не изменяя описание старого; и

[3] невозможно иметь конструктор с одним параметром, не имея при этом преобразования.

Последнее не является серьезной проблемой, а с первыми двумя можно справиться, определив для исходного типа операцию преобра-

зования. Функция член `X::operator T()`, где `T` - имя типа, определяет преобразование из `X` в `T`. Например, можно определить тип `tiny` (крошечный), который может иметь значение только в диапазоне 0...63, но все равно может свободно сочетаться с целыми в арифметических операциях:

```
class tiny {
    char v;
    int assign(int i)
    { return v = (i&~63) ? (error(«ошибка диа-
пазона»),0) : i; }
public:
    tiny(int i)                { assign(i); }
    tiny(tiny& i)              { v = t.v; }
    int operator=(tiny& i) { return v = t.v; }
    int operator=(int i)      { return assign(i); }
    operator int()             { return v; }
}
```

Диапазон значения проверяется всегда, когда `tiny` инициализируется `int`, и всегда, когда ему присваивается `int`. Одно `tiny` может присваиваться другому без проверки диапазона. Чтобы разрешить выполнять над переменными `tiny` обычные целые операции, определяется `tiny::operator int()`, неявное преобразование из `int` в `tiny`. Всегда, когда в том месте, где требуется `int`, появляется `tiny`, используется соответствующее ему `int`. Например:

```
void main()
{
    tiny c1 = 2;
    tiny c2 = 62;
    tiny c3 = c2 - c1;    // c3 = 60
    tiny c4 = c3;         // нет проверки диапа-
зона (необязательна)
    int i = c1 + c2;      // i = 64
    c1 = c2 + 2 * c1;     // ошибка диапазона:
c1 = 0 (а не 66)
    c2 = c1 - i;          // ошибка диапазона:
c2 = 0
    c3 = c2;              // нет проверки диапа-
зона (необязательна)
}
```


Тип вектор из `tiny` может оказаться более полезным, поскольку он экономит пространство. Чтобы сделать этот тип более удобным в обращении, можно использовать операцию индексирования.

Другое применение определяемых операций преобразования - это типы, которые предоставляют нестандартные представления чисел (арифметика по основанию 100, арифметика с фиксированной точкой, двоично-десятичное представление и т.п.). При этом обычно переопределяются такие операции, как `+` и `*`.

Функции преобразования оказываются особенно полезными для работы со структурами данных, когда чтение (реализованное посредством операции преобразования) тривиально, в то время как присваивание и инициализация заметно более сложны.

Типы `istream` и `ostream` опираются на функцию преобразования, чтобы сделать возможными такие операторы, как `while (cin>>x) cout<<x` выше возвращает `istream&`. Это значение неявно преобразуется к значению, которое указывает состояние `cin`, а уже это значение может проверяться оператором `while`. Однако определять преобразование из одного типа в другой так, что при этом теряется информация, обычно не стоит.

6.3.3. Неоднозначности

Присваивание объекту (или инициализация объекта) класса `X` является допустимым, если или присваиваемое значение является `X`, или существует единственное преобразование присваиваемого значения в тип `X`.

В некоторых случаях значение нужного типа может сконструироваться с помощью нескольких применений конструкторов или операций преобразования. Это должно делаться явно; допустим только один уровень неявных преобразований, определенных пользователем. Иногда значение нужного типа может быть сконструировано более, чем одним способом. Такие случаи являются недопустимыми. Например:

```
class x { /* ... */ x(int); x(char*); };
class y { /* ... */ y(int); };
class z { /* ... */ z(x); };

overload f;
x f(x);
y f(y);

z g(z);
```

```

    f(1);                // недопустимо: неоднозначность
f(x(1)) или f(y(1))
    f(x(1));
    f(y(1));
    g(«asdf»);          // недопустимо: g(z(x(«asdf»)))
не пробуются
    g(z(«asdf»));

```

Определенные пользователем преобразования рассматриваются только в том случае, если без них вызов разрешить нельзя. Например:

```

class x { /* ... */ x(int); }
overload h(double), h(x);
h(1);

```

Вызов мог бы быть проинтерпретирован или как `h(double(1))`, или как `h(x(1))` и был бы недопустим по правилу единственности. Но первая интерпретация использует только стандартное преобразование и она будет выбрана по правилам, приведенным в разделе 3.1. Правила преобразования не являются ни самыми простыми для реализации и документации, ни наиболее общими из тех, которые можно было бы разработать. Возьмем требование единственности преобразования. Более общий подход разрешил бы компилятору применять любое преобразование, которое он сможет найти; таким образом, не нужно было бы рассматривать все возможные преобразования перед тем, как объявить выражение допустимым. К сожалению, это означало бы, что смысл программы зависит от того, какое преобразование было найдено. В результате смысл программы неким образом зависел бы от порядка описания преобразования. Поскольку они часто находятся в разных исходных файлах (написанных разными людьми), смысл программы будет зависеть от порядка компоновки этих частей вместе. Есть другой вариант - запретить все неявные преобразования. Нет ничего проще, но такое правило приведет либо к неэлегантным пользовательским интерфейсам, либо к бурному росту перегруженных функций, как это было в предыдущем разделе с `complex`.

Самый общий подход учитывал бы всю имеющуюся информацию о типах и рассматривал бы все возможные преобразования. Например, если использовать предыдущее описание, то можно было бы обработать `aa=f(1)`, так как тип `aa` определяет единственность толкования. Если `aa` является `x`, то единственное, дающее в результате `x`, который требуется присваиванием, - это `f(x(1))`, а если `aa` - это `y`, то вместо этого будет использоваться `f(y(1))`. Самый общий подход справился бы и с `g(«asdf»)`, поскольку единственной интерпретацией этого может быть

`g(z(x(«asdf»)))`). Сложность этого подхода в том, что он требует расширенного анализа всего выражения для того, чтобы определить интерпретацию каждой операции и вызова функции. Это приведет к замедлению компиляции, а также к вызывающим удивление интерпретациям и сообщениям об ошибках, если компилятор рассмотрит преобразования, определенные в библиотеках и т.п. При таком подходе компилятор будет принимать во внимание больше, чем, как можно ожидать, знает пишущий программу программист!

6.4. Константы

Константы классового типа определить невозможно в том смысле, в каком `1.2` и `12e3` являются константой типа `double`. Вместо них, однако, часто можно использовать константы основных типов, если их реализация обеспечивается с помощью функций членов. Общий аппарат для этого дают конструкторы, получающие один параметр. Когда конструкторы просты и подставляются `inline`, имеет смысл рассмотреть в качестве константы вызов конструктора. Если, например, в программе есть описание класса `complex`, то выражение `zz1*3+zz2*complex(1,2)` даст два вызова функций, а не пять. К двум вызовам функций приведут две операции `*`, а операция `+` и конструктор, к которому обращаются для создания `complex(3)` и `complex(1,2)`, будут расширены `inline`.

6.5. Большие объекты

При каждом применении для `complex` бинарных операций, описанных выше, в функцию, которая реализует операцию, как параметр передается копия каждого операнда. Расходы на копирование каждого `double` заметны, но с ними вполне можно примириться. К сожалению, не все классы имеют небольшое и удобное представление. Чтобы избежать ненужного копирования, можно описать функции таким образом, чтобы они получали ссылочные параметры. Например:

```
class matrix {
    double m[4][4];
public:
    matrix();
    friend matrix operator+(matrix&, matrix&);
    friend matrix operator*(matrix&, matrix&);
};
```

Ссылки позволяют использовать выражения, содержащие обычные арифметические операции над большими объектами, без ненужного копирования. Указатели применять нельзя, потому что для применения к указателю смысл операции переопределить невозможно. Операцию плюс можно определить так:

```
matrix operator+(matrix&, matrix&);
{
    matrix sum;
    for (int i=0; i<4; i++)
        for (int j=0; j<4; j++)
            sum.m[i][j] = arg1.m[i][j] + arg2.m[i][j];
    return sum;
}
```

Этот `operator+` обращается к операндам `+` через ссылки, но возвращает значение объекта. Возврат ссылки может оказаться более эффективным:

```
class matrix {
    // ...
    friend matrix& operator+(matrix&, matrix&);
    friend matrix& operator*(matrix&, matrix&);
};
```

Это является допустимым, но приводит к сложности с выделением памяти. Поскольку ссылка на результат будет передаваться из функции как ссылка на возвращаемое значение, оно не может быть автоматической переменной. Поскольку часто операция используется в выражении больше одного раза, результат не может быть и статической переменной. Как правило, его размещают в свободной памяти. Часто копирование возвращаемого значения оказывается дешевле (по времени выполнения, объему кода и объему данных) и проще программируется.

6.6. Присваивание и инициализация

Рассмотрим очень простой класс строк `string`:

```
struct string {
    char* p;
    int size;    // размер вектора, на который
                // указывает p

    string(int sz) { p = new char[size=sz]; }
    ~string() { delete p; }
};
```

Строка - это структура данных, состоящая из вектора символов и длины этого вектора. Вектор создается конструктором и уничтожается деструктором. Однако, как показано выше, это может привести к неприятностям. Например:

```
void f()
{
    string s1(10);
    string s2(20);
    s1 = s2;
}
```

будет размещать два вектора символов, а присваивание `s1=s2` будет портить указатель на один из них и дублировать другой. На выходе из `f()` для `s1` и `s2` будет вызываться деструктор и уничтожать один и тот же вектор с непредсказуемо разрушительными последствиями. Решение этой проблемы состоит в том, чтобы соответствующим образом определить присваивание объектов типа `string`:

```
struct string {
    char* p;
    int size;    // размер вектора, на который
                // указывает p

    string(int sz) { p = new char[size=sz]; }
    ~string() { delete p; }
    void operator=(string&)

};

void string::operator=(string& a)
{
    if (this == &a) return;    // остере-
    гаться s=s;
    delete p;
    p=new char[size=a.size];
    strcpy(p,a.p);
}
```

Это определение `string` гарантирует, и что предыдущий пример будет работать, как предполагалось. Однако небольшое изменение `f()` приведет к появлению той же проблемы в новом облике:

```

void f()
{
    string s1(10);
    s2 = s1;
}

```

Теперь создается только одна строка, а уничтожается две. К неинициализированному объекту определенная пользователем операция присваивания не применяется. Беглый взгляд на `string::operator=()` объясняет, почему было неразумно так делать: указатель `p` будет содержать неопределенное и совершенно случайное значение. Часто операция присваивания полагается на то, что ее аргументы инициализированы. Для такой инициализации, как здесь, это не так по определению. Следовательно, нужно определить похожую, но другую, функцию, чтобы обрабатывать инициализацию:

```

struct string {
    char* p;
    int size;    // размер вектора, на который
указывает p

    string(int sz) { p = new char[size=sz]; }
    ~string() { delete p; }
    void operator=(string&)
    string(string&);
};

void string::string(string& a)
{
    p=new char[size=a.size];
    strcpy(p,a.p);
}

```

Для типа `X` инициализацию тем же типом `X` обрабатывает конструктор `X(X&)`. Нельзя не подчеркнуть еще раз, что присваивание и инициализация - разные действия. Это особенно существенно при описании деструктора. Если класс `X` имеет конструктор, выполняющий нетривиальную работу вроде освобождения памяти, то скорее всего потребуется полный комплект функций, чтобы полностью избежать побитового копирования объектов:

```

class X {
    // ...
    X(something);    // конструктор: создает
объект
    X(&X);           // конструктор: копирует в
инициализации
    operator=(X&);  // присваивание: чистит и
копирует
    ~X();           // деструктор: чистит
};

```

Есть еще два случая, когда объект копируется: как параметр функции и как возвращаемое значение. Когда передается параметр, инициализируется неинициализированная до этого переменная - формальный параметр. Семантика идентична семантике инициализации. То же самое происходит при возврате из функции, хотя это менее очевидно. В обоих случаях будет применен `X(X&)`, если он определен:

```

string g(string arg)
{
    return arg;
}

main()
{
    string s = «asdf»;
    s = g(s);
}

```

Ясно, что после вызова `g()` значение `s` обязано быть «asdf». Копирование значения `s` в параметр `arg` сложности не представляет: для этого надо вызывать `string(string&)`. Для взятия копии этого значения из `g()` требуется еще один вызов `string(string&)`; на этот раз инициализируемой является временная переменная, которая затем присваивается `s`. Такие переменные, естественно, уничтожаются как положено с помощью `string::~~string()` при первой возможности.

6.7. Индексирование

Чтобы задать смысл индексов для объектов класса используется функция `operator[]`. Второй параметр (индекс) функции `operator[]` может быть любого типа. Это позволяет определять ассоциативные массивы и т.п. В качестве примера давайте перепишем пример из 2.3.10,

где при написании небольшой программы для подсчета числа вхождений слов в файле применялся ассоциативный массив. Там использовалась функция. Здесь определяется надлежащий тип ассоциативного массива:

```
struct pair {
    char* name;
    int val;
};

class assoc {
    pair* vec;
    int max;
    int free;
public:
    assoc(int);
    int& operator[] (char*);
    void print_all();
};
```

В `assoc` хранится вектор пар `pair` длины `max`. Индекс первого неиспользованного элемента вектора находится в `free`. Конструктор выглядит так:

```
assoc::assoc(int s)
{
    max = (s<16) ? s : 16;
    free = 0;
    vec = new pair[max];
}
```

При реализации применяется все тот же простой и неэффективный метод поиска, что использовался в 2.3.10. Однако при переполнении `assoc` увеличивается:

```
#include

int assoc::operator[] (char* p)
/*
    работа с множеством пар «pair»:
    поиск p,
    возврат ссылки на целую часть его «pair»
    делает новую «pair», если p не встречалось
*/
{
    register pair* pp;
```



```

        for (pp=&vec[free-1]; vec<=pp; pp--)
            if (strcmp(p,pp->name)==0) return pp->val;

        if (free==max) {           // переполнение: век-
тор увеличивается
            pair* nvec = new pair[max*2];
            for (int i=0; i<max*2; i++)
                nvec[i].name = new
char[strlen(p)+1];
            strcpy(pp->name,p);
            pp->val = 0;           // начальное значение: 0
            return pp->val;
        }

```

Поскольку представление `assoc` скрыто, нам нужен способ его печати. В следующем разделе будет показано, как определить подходящий итератор, а здесь мы используем простую функцию печати:

```

void assoc::print_all()
{
    for (int i = 0; i<buf; i++) vec[i].print_all();
}

```

6.8. Вызов функции

Вызов функции, то есть запись выражение(список_выражений), можно проинтерпретировать как бинарную операцию, и операцию вызова можно перегружать так же, как и другие операции. Список параметров функции `operator()` вычисляется и проверяется в соответствии с обычными правилами передачи параметров. Перегружающая функция может оказаться полезной главным образом для определения типов с единственной операцией и для типов, у которых одна операция настолько преобладает, что другие в большинстве ситуаций можно не принимать во внимание.

Для типа ассоциативного массива `assoc` мы не определили итератор. Это можно сделать, определив класс `assoc_iterator`, работа которого состоит в том, чтобы в определенном порядке поставлять элементы из `assoc`. Итератору нужен доступ к данным, которые хранятся в `assoc`, поэтому он сделан другом:

```

class assoc {
    friend class assoc_iterator;
    pair* vec;
}

```

```

        int max;
        int free;
    public:
        assoc(int);
        int& operator[] (char*);
};

```

Итератор определяется как

```

class assoc_iterator{
    assoc* cs; // текущий массив assoc
    int i;     // текущий индекс
    public:
        assoc_iterator(assoc& s) { cs = &s; i = 0; }
        pair* operator() ()
            { return (ifree)? &cs->vec[i++] : 0; }
};

```

Нужно инициализировать `assoc_iterator` для массива `assoc`, после чего он будет возвращать указатель на новую `pair` из этого массива всякий раз, когда его будут активизировать операцией `()`. По достижении конца массива он возвращает 0:

```

main() // считает вхождения каждого слова
    во вводе
    {
        const MAX = 256; // больше самого большо-
го слова
        char buf[MAX];
        assoc vec(512);
        while (cin>>buf) vec[buf]++;
        assoc_iterator next(vec);
        pair* p;
        while ( p = next() )
            cout << p->name << «: « << p->val << «\n»;
    }

```

Итераторный тип вроде этого имеет преимущество перед набором функций, которые выполняют ту же работу: у него есть собственные закрытые данные для хранения хода итерации. К тому же, обычно существенно, чтобы одновременно могли работать много итераторов этого типа.

Конечно, такое применение объектов для представления итераторов никак особенно с перегрузкой операций не связано. Многие любят

использовать итераторы с такими операциями, как `first()`, `next()` и `last()` (первый, следующий и последний).

6.9. Класс строка

Вот довольно реалистичный пример класса `string`. В нем производится учет ссылок на строку с целью минимизировать копирование и в качестве констант применяются стандартные символьные строки C++.

```
#include
#include

class string {
    struct srep {
        char* s;          // указатель на данные
        int    n;          // счетчик ссылок
    };
    srep *p;

public:
    string(char *);        // string x = «abc»
    string();              // string x;
    string(string &);      // string x = string ...
    string& operator=(char *);
    string& operator=(string &);
    ~string();
    char& operator[](int i);

    friend ostream& operator<<(ostream&,
string&);
    friend istream& operator>>(istream&,
string&);

    friend int operator==(string& x, char* s)
        {return strcmp(x.p->s, s) == 0; }

    friend int operator==(string& x, string& y)
        {return strcmp(x.p->s, y.p->s) == 0; }

    friend int operator!=(string& x, char* s)
        {return strcmp(x.p->s, s) != 0; }

    friend int operator!=(string& x, string& y)
```

```

        {return strcmp(x.p->s, y.p->s) != 0; }

};

```

Конструкторы и деструкторы просты (как обычно):

```

string::string()
{
    p = new srep;
    p->s = 0;
    p->n = 1;
}

string::string(char* s)
{
    p = new srep;
    p->s = new char[ strlen(s)+1 ];
    strcpy(p->s, s);
    p->n = 1;
}

string::string(string& x)
{
    x.p->n++;
    p = x.p;
}

string::~~string()
{
    if (--p->n == 0) {
        delete p->s;
        delete p;
    }
}

```

Как обычно, операции присваивания очень похожи на конструкторы. Они должны обрабатывать очистку своего первого (левого) операнда:

```

string& string::operator=(char* s)
{
    if (p->n > 1) {        // разъединить себя
        p->n--;
        p = new srep;
    }
}

```

```

    }
    else if (p->n == 1)
        delete p->s;

    p->s = new char[ strlen(s)+1 ];
    strcpy(p->s, s);
    p->n = 1;
    return *this;
}

```

Благоразумно обеспечить, чтобы присваивание объекта самому себе работало правильно:

```

string& string::operator=(string& x)
{
    x.p->n++;
    if (--p->n == 0) {
        delete p->s;
        delete p;
    }
    p = x.p;
    return *this;
}

```

Операция вывода задумана так, чтобы продемонстрировать применение учета ссылок. Она повторяет каждую вводимую строку (с помощью операции <<, которая определяется позднее):

```

ostream& operator<<(ostream& s, string& x)
{
    return s << x.p->s << « [« << x.p->n << «]\n»;
}

```

Операция ввода использует стандартную функцию ввода символьной строки.

```

istream& operator>>(istream& s, string& x)
{
    char buf[256];
    s >> buf;
    x = buf;
    cout << «echo: « << x << «\n»;
    return s;
}

```

Для доступа к отдельным символам предоставлена операция индексирования. Осуществляется проверка индекса:

```
void error(char* p)
{
    cerr << p << «\n»;
    exit(1);
}

char& string::operator[](int i)
{
    if (i<0 || strlen(p->s)s[i];
}
```

Главная программа просто немного опробует действия над строками. Она читает слова со ввода в строки, а потом эти строки печатает. Она продолжает это делать до тех пор, пока не распознает строку done, которая завершает сохранение слов в строках, или не встретит конец файла. После этого она печатает строки в обратном порядке и завершается.

```
main()
{
    string x[100];
    int n;

    cout << «отсюда начнем\n»;
    for (n = 0; cin>>x[n]; n++) {
        string y;
        if (n==100)    error(«слишком много
строк»);
        cout << (y = x[n]);
        if (y==«done») break;
    }
    cout << «отсюда мы пройдем обратно\n»;
    for (int i=n-1; 0<=i; i--) cout << x[i];
}
```

6.10. Друзья и члены

Теперь, наконец, можно обсудить, в каких случаях для доступа к закрытой части определяемого пользователем типа использовать члены, а в каких - друзей. Некоторые операции должны быть членами:

конструкторы, деструкторы и виртуальные функции (см. следующую главу), но обычно это зависит от выбора.

Рассмотрим простой класс X:

```
class X {
    // ...
    X(int);
    int m();
    friend int f(X&);
};
```

Внешне не видно никаких причин делать `f(X&)` другом дополнительно к члену `X::m()` (или наоборот), чтобы реализовать действия над классом X. Однако член `X::m()` можно вызывать только для «настоящего объекта», в то время как друг `f()` может вызываться для объекта, созданного с помощью неявного преобразования типа. Например:

```
void g()
{
    l.m();           // ошибка
    f(1);            // f(x(1));
}
```

Поэтому операция, изменяющая состояние объекта, должна быть членом, а не другом. Для определяемых пользователем типов операции, требующие в случае фундаментальных типов операнд `lvalue` (`=`, `*=`, `++` и т.д.), наиболее естественно определяются как члены.

И наоборот, если нужно иметь неявное преобразование для всех операндов операции, то реализующая ее функция должна быть другом, а не членом. Это часто имеет место для функций, которые реализуют операции, не требующие при применении к фундаментальным типам `lvalue` в качестве операндов (`+`, `-`, `||` и т.д.).

Если никакие преобразования типа не определены, то оказывается, что нет никаких существенных оснований в пользу члена, если есть друг, который получает ссылочный параметр, и наоборот. В некоторых случаях программист может предпочитать один синтаксис вызова другому. Например, оказывается, что большинство предпочитает для обращения матрицы `m` запись `m.inv()`. Конечно, если `inv()` действительно обращает матрицу `m`, а не просто возвращает новую матрицу, обратную `m`, ей следует быть другом.

При прочих равных условиях выбирайте, чтобы функция была членом: никто не знает, вдруг когда-нибудь кто-то определит операцию преобразования. Невозможно предсказать, потребуют ли будущие изменения изменить статус объекта. Синтаксис вызова функции члена

ясно указывает пользователю, что объект можно изменить; ссылочный параметр является далеко не столь очевидным. Кроме того, выражения в члене могут быть заметно короче выражений в друге. В функции друге надо использовать явный параметр, тогда как в члене можно использовать неявный `this`. Если только не применяется перегрузка, имена членов обычно короче имен друзей.

6.11. Предостережение

Как и большая часть возможностей в языках программирования, перегрузка операций может применяться как правильно, так и неправильно. В частности, можно так использовать возможность определять новые значения старых операций, что они станут почти совсем недоступными. Представьте, например, с какими сложностями столкнется человек, читающий программу, в которой операция `+` была переопределена для обозначения вычитания.

Данный аппарат должен уберечь программиста/читателя от худших крайностей применения перегрузки, потому что программист предохранил от изменения значения операций для основных типов данных вроде `int`, а также потому, что синтаксис выражений и приоритеты операций сохраняются.

Может быть, разумно применять перегрузку операций главным образом так, чтобы подражать общепринятому применению операций. В тех случаях, когда нет общепринятой операции или имеющееся в C++ множество операций не подходит для имитации общепринятого применения, можно использовать запись вызова функции.

7. ГЛАВА СЕДЬМАЯ

В этой главе описывается понятие производного класса в C++. Производные классы дают простой, гибкий и эффективный аппарат задания для класса альтернативного интерфейса и определения класса посредством добавления возможностей к уже имеющемуся классу без перепрограммирования или перекомпиляции. С помощью производных классов можно также обеспечить общий интерфейс для нескольких различных классов так, чтобы другие части программы могли работать с объектами этих классов одинаковым образом. При этом обычно в каждый объект помещается информация о типе, чтобы эти объекты могли обрабатываться соответствующим образом в ситуациях, когда их тип нельзя узнать во время компиляции. Для элегантной и надежной обработки таких динамических зависимостей типов имеется понятие виртуальной функции. По своей сути производные классы существуют для того, чтобы облегчить программисту формулировку общности.

7.1. Введение

Представим себе процесс написания некоторого средства общего назначения (например, тип связанный список, таблица имен или планировщик для системы моделирования), которое предназначается для использования многими разными людьми в различных обстоятельствах. Очевидно, что в кандидатах на роль таких средств недостатка нет и выгоды от их стандартизации огромны. Кажется, любой опытный программист написал (и отладил) дюжину вариантов типов множества, таблицы имен, сортирующей функции и т.п., но оказывается, что таблиц имен каждый программист и каждая программа используют свою версию этих понятий, из-за чего программы слишком трудно читать, тяжело отлаживать и сложно модифицировать. Более того, в большой программе вполне может быть несколько копий идентичных (почти) частей кода для работы с такими фундаментальными понятиями.

Причина этого хаоса частично состоит в том, что представить такие общие понятия в языке программирования сложно с концептуальной точки зрения, а частично в том, что средства, обладающие достаточной общностью, налагают дополнительные расходы по памяти и/или по времени, что делает их неудобными для самых простых и наиболее напряженно используемых средств (связанные списки, векторы и т.п.), где они были бы наиболее полезны. Понятие производного класса в C++ не обеспечивает общего решения всех этих проблем,

но оно дает способ справляться с довольно небольшим числом важных случаев. Будет, например, показано, как определить эффективный класс общего связанного списка таким образом, чтобы все его версии использовали код совместно.

Написание общецелевых средств - задача непростая, и часто основной акцент в их разработке другой, чем при разработке программ специального назначения. Конечно, нет четкой границы между средствами общего и специального назначения, и к методам и языковым средствам, которые описываются в этой главе, можно относиться так, что они становятся все более полезны с ростом объема и сложности создаваемых программ.

7.2. Производные классы

Чтобы разделить задачи понимания аппарата языка и методов его применения, знакомство с понятием производных классов делается в три этапа. Вначале с помощью небольших примеров, которые не надо воспринимать как реалистичные, будут описаны сами средства языка (запись и семантика). После этого демонстрируются некоторые неочевидные применения производных классов, и, наконец, приводится законченная программа.

7.2.1. Построение производного класса

Рассмотрим построение программы, которая имеет дело с людьми, служащими в некоторой фирме. Структура данных в этой программе может быть, например, такой:

```
struct employee {           // служащий
    char*   name;           // имя
    short   age;            // возраст
    short   department;    // подразделение
    int     salary;         //
    employee* next;
    // ...
};
```

Список аналогичных служащих будет связываться через поле `next`. Теперь давайте определим менеджера:

```
struct manager {           // менеджер
    employee emp;          // запись о менедже-
    // жере как о служащем
    employee* group;       // подчиненные люди
```

```

        // ...
    };

```

Менеджер также является служащим; относящиеся к служащему `employee` данные хранятся в члене `emp` объекта `manager`. Для читающего человека это, может быть, очевидно, но нет ничего выделяющего член `emp` для компилятора. Указатель на менеджера (`manager*`) не является указателем на служащего (`employee*`), поэтому просто использовать один там, где требуется другой, нельзя. В частности, нельзя поместить менеджера в список служащих, не написав для этого специальную программу. Можно либо применить к `manager*` явное преобразование типа, либо поместить в список служащих адрес члена `emp`, но и то и другое мало элегантно и довольно неясно. Корректный подход состоит в том, чтобы установить, что менеджер является служащим с некоторой добавочной информацией:

```

    struct manager : employee {
        employee* group;
        // ...
    };

```

`manager` является производным от `employee` и, обратно, `employee` есть базовый класс для `manager`. Класс `manager` дополнительно к члену `group` имеет члены класса `employee` (`name`, `age` и т.д.).

Имея определения `employee` и `manager`, мы можем теперь создать список служащих, некоторые из которых являются менеджерами. Например:

```

void f()
{
    manager m1, m2;
    employee e1, e2;
    employee* elist;
    elist = &m1;      // поместить m1, e1, m2 и
e2 в elist
    m1.next = &e1;
    e1.next = &m2;
    m2.next = &e2;
    e2.next = 0;
}

```

Поскольку менеджер является служащим, `manager*` может использоваться как `employee*`. Однако служащий необязательно является менеджером, поэтому использовать `employee*` как `manager*` нельзя.

7.2.2. Функции члены

Просто структуры данных вроде `employee` и `manager` на самом деле не столь интересны и часто не особенно полезны, поэтому рассмотрим, как добавить к ним функции. Например:

```
class employee {
    char* name;
    // ...
public:
    employee* next;
    void print();
    // ...
};

class manager : public employee {
    // ...
public:
    void print();
    // ...
};
```

Нужно ответить на некоторые вопросы. Как может функция-член производного класса `manager` использовать члены его базового класса `employee`? Как члены базового класса `employee` могут использовать функции-члены производного класса `manager`? Какие члены базового класса `employee` может использовать функция-член на объекте типа `manager`? Каким образом программист может повлиять на ответы на эти вопросы, чтобы удовлетворить требованиям приложения?

Рассмотрим:

```
void manager::print()
{
    cout << « имя » << name << «\n»;
    // ...
}
```

Член производного класса может использовать открытое имя из своего базового класса так же, как это могут делать другие члены последнего, то есть без указания объекта. Предполагается, что на объект указывает `this`, поэтому (корректной) ссылкой на имя `name` является `this->name`. Однако функция `manager::print` компилироваться не будет, член производного класса не имеет никакого особого права доступа к

закрытым членам его базового класса, поэтому для нее name недоступно.

Это многим покажется удивительным, но представьте себе другой вариант, функция член могла бы обращаться к закрытым членам своего базового класса. Возможность, позволяющая программисту получать доступ к закрытой части класса просто с помощью вывода из него другого класса, лишила бы понятие закрытого члена всякого смысла. Более того, нельзя было бы узнать все использования закрытого имени посмотрев на функции, описанные как члены и друзья этого класса. Пришлось бы проверять каждый исходный файл во всей программе на наличие в нем производных классов, потом исследовать каждую функцию этих классов, потом искать все классы, производные от этих классов, и т.д. Это по меньшей мере утомительно и скорее всего нереально.

С другой стороны, можно ведь использовать механизм friend, чтобы предоставить такой доступ или отдельным функциям, или всем функциям отдельного класса. Например:

```
class employee {
    friend void manager::print();
    // ...
};
```

решило бы проблему с manager::print(), и

```
class employee {
    friend class manager;
    // ...
};
```

сделало бы доступным каждый член employee для всех функций класса manager. В частности, это сделает name доступным для manager::print().

Другое, иногда более прозрачное решение для производного класса - использовать только открытые члены его базового класса. Например:

```
void manager::print()
{
    employee::print(); // печатает информацию о служащем
    // ...             // печатает информацию о менеджере
}
```

Заметьте, что надо использовать ::, потому что print() была переопределена в manager. Такое повторное использование имен типично. Неосторожный мог бы написать так:

```
void manager::print()
{
    print();                // печатает информа-
    цию о служащем          // печатает информа-
    // ...                  цию о менеджере
}
```

и обнаружить, что программа после вызова manager::print() неожиданно попадает в последовательность рекурсивных вызовов.

7.2.3. Видимость

Класс employee стал открытым (public) базовым классом класса manager в результате описания:

```
class manager : public employee {
    // ...
};
```

Это означает, что открытый член класса employee является также и открытым членом класса manager. Например:

```
void clear(manager* p)
{
    p->next = 0;
}
```

будет компилироваться, так как next - открытый член и employee и manager'a. Альтернатива - можно определить закрытый (private) класс, просто опустив в описании класса слово public:

```
class manager : employee {
    // ...
};
```

Это означает, что открытый член класса employee является закрытым членом класса manager. То есть функции члены класса manager могут, как и раньше, использовать открытые члены класса employee, но для пользователей класса manager эти члены недоступны. В частности, при таком описании класса manager функция clear() компилироваться не будет. Друзья производного класса имеют к членам базового класса такой же доступ, как и функции члены.

Поскольку, как оказывается, описание открытых базовых классов встречается чаще описания закрытых, жалко, что описание открытого базового класса длиннее описания закрытого. Это, кроме того, служит источником запутывающих ошибок у начинающих.

Когда описывается производная struct, ее базовый класс по умолчанию является public базовым классом. То есть,

```
struct D : B { ...
```

означает

```
class D : public B { public: ...
```

Отсюда следует, что если вы не сочли полезным то скрытие данных, которое дают class, public и friend, вы можете просто не использовать эти ключевые слова и придерживаться struct. Такие средства языка, как функции члены, конструкторы и перегрузка операций, не зависят от механизма скрытия данных.

Можно также объявить некоторые, но не все, открытые \$ члены базового класса открытыми членами производного класса. Например:

```
class manager : employee {
    // ...
public:
    // ...
    employee::name;
    employee::department;
};
```

Запись

```
имя_класса :: имя_члена ;
```

не вводит новый член, а просто делает открытый член базового класса открытым для производного класса. Теперь name и department могут использоваться для manager'a, а salary и age - нет. Естественно, сделать сделать закрытый член базового класса открытым членом производного класса невозможно. Невозможно с помощью этой записи также сделать открытыми перегруженные имена.

Подытоживая, можно сказать, что вместе с предоставлением средств дополнительно к имеющимся в базовом классе, производный класс можно использовать для того, чтобы сделать средства (имена) недоступными для пользователя. Другими словами, с помощью производного класса можно обеспечивать прозрачный, полупрозрачный и непрозрачный доступ к его базовому классу.

7.2.4. Указатели

Если производный класс `derived` имеет открытый базовый класс `base`, то указатель на `derived` можно присваивать переменной типа указатель на `base` не используя явное преобразование типа. Обратное преобразование, указателя на `base` в указатель на `derived`, должно быть явным. Например:

```
class base { /* ... */ };
class derived : public base { /* ... */ };

derived m;
base* pb = &m;      // неявное преобразование
derived* pd = pb;   // ошибка: base* не является
derived*            derived*
pd = (derived*)pb;  // явное преобразование
```

Иначе говоря, объект производного класса при работе с ним через указатель и можно рассматривать как объект его базового класса. Обратное неверно.

Будь `base` закрытым базовым классом класса `derived`, неявное преобразование `derived*` в `base*` не делалось бы. Неявное преобразование не может в этом случае быть выполнено, потому что к открытому члену класса `base` можно обращаться через указатель на `base`, но нельзя через указатель на `derived`:

```
class base {
    int m1;
public:
    int m2;      // m2 - открытый член base
};

class derived : base {
    // m2 НЕ открытый член derived
};

derived d;
d.m2 = 2;        // ошибка: m2 из закрытой части
ти класса
base* pb = &d;   // ошибка: (закрытый base)
pb->m2 = 2;      // ok
pb = (base*)&d;  // ok: явное преобразование
pb->m2 = 2;      // ok
```


Помимо всего прочего, этот пример показывает, что, используя явное приведение к типу, можно сломать правила защиты. Ясно, делать это не рекомендуется, и это приносит программисту заслуженную «награду». К несчастью, недисциплинированное использование явного преобразования может создать адские условия для невинных жертв, которые эксплуатируют программу, где это делается. Но, к счастью, нет способа воспользоваться приведением для получения доступа к закрытому имени `m1`. Закрытый член класса может использоваться только членами и друзьями этого класса.

7.2.5. Иерархия типов

Производный класс сам может быть базовым классом. Например:

```
class employee { ... };
class secretary : employee { ... };
class manager : employee { ... };
class temporary : employee { ... };
class consultant : temporary { ... };
class director : manager { ... };
class vice_president : manager { ... };
class president : vice_president { ... };
```

Такое множество родственных классов принято называть иерархией классов. Поскольку можно выводиться класс только из одного базового класса, такая иерархия является деревом и не может быть графом более общей структуры. Например:

```
class temporary { ... };
class employee { ... };
class secretary : employee { ... };

// не C++:
class temporary_secretary : temporary : secretary { ... };
class consultant : temporary : employee { ... };
```

И этот факт вызывает сожаление, потому что направленный ациклический граф производных классов был бы очень полезен. Такие структуры описать нельзя, но можно смоделировать с помощью членов соответствующих типов. Например:

```
class temporary { ... };
class employee { ... };
class secretary : employee { ... };
```

```
// Альтернатива:
class temporary_secretary : secretary
{ temporary temp; ... };
class consultant : employee
{ temporary temp; ... };
```

Это выглядит неэлегантно и страдает как раз от тех проблем, для преодоления которых были изобретены производные классы. Например, поскольку consultant не является производным от temporary, consultant'a нельзя помещать с список временных служащих (temporary employee), не написав специальной программы. Однако во многих полезных программах этот метод успешно используется.

7.2.6. Конструкторы и деструкторы

Для некоторых производных классов нужны конструкторы. Если у базового класса есть конструктор, он должен вызываться, и если для этого конструктора нужны параметры, их надо предоставить. Например:

```
class base {
    // ...
public:
    base(char* n, short t);
    ~base();
};

class derived : public base {
    base m;
public:
    derived(char* n);
    ~derived();
};
```

Параметры конструктора базового класса специфицируются в определении конструктора производного класса. В этом смысле базовый класс работает точно так же, как неименованный член производного класса. Например:

```
derived::derived(char* n) : (n,10),
m («member»,123)
{
    // ...
}
```

Объекты класса конструируются снизу вверх: сначала базовый, потом члены, а потом сам производный класс. Уничтожаются они в обратном порядке: сначала сам производный класс, потом члены а потом базовый.

7.2.7. Поля типа

Чтобы использовать производные классы не просто как удобную сокращенную запись в описаниях, надо разрешить следующую проблему. Если задан указатель типа `base*`, какому производному типу в действительности принадлежит указываемый объект? Есть три основных способа решения этой проблемы:

[1] обеспечить, чтобы всегда указывались только объекты одного типа;

[2] поместить в базовый класс поле типа, которое смогут просматривать функции; и

[3] использовать виртуальные функции.

Обыкновенно указатели на базовые классы используются при разработке контейнерных (или вмещающих) классов: множество, вектор, список и т.п. В этом случае решение 1 дает однородные списки, то есть списки объектов одного типа. Решения 2 и 3 можно использовать для построения неоднородных списков, то есть списков объектов (указателей на объекты) нескольких различных типов. Решение 3 - это специальный вариант решения 2, безопасный относительно типа.

Давайте сначала исследуем простое решение с помощью поля типа, то есть решение 2. Пример со служащими и менеджерами можно было бы переопределить так:

```
enum empl_type { M, E };

struct employee {
    empl_type type;
    employee* next;
    char*      name;
    short      department;
    // ...
};

struct manager : employee {
    employee* group;
    short      level;           // уровень
};
```

Имея это, мы можем теперь написать функцию, которая печатает информацию о каждом служащем:

```
void print_employee(employee* e)
{
    switch (e->type) {
        case E:
            cout << e->name << «\t» << e-
>department << «\n»;
            // ...
            break;
        case M:
            cout << e->name << «\t» << e-
>department << «\n»;
            // ...
            manager* p = (manager*)e;
            cout << « уровень » << p->level << «\n»;
            // ...
            break;
    }
}
```

и воспользоваться ею для того, чтобы напечатать список служащих:

```
void f()
{
    for      (;      ll;      ll=ll->next)
print_employee(ll);
}
```

Это прекрасно работает, особенно в небольшой программе, написанной одним человеком, но имеет тот коренной недостаток, что неконтролируемым компилятором образом зависит от того, как программист работает с типами. В больших программах это обычно приводит к ошибкам двух видов. Первый - это невыполнение проверки поля типа, второй - когда не все случаи case помещаются в переключатель switch, как в предыдущем примере. Оба избежать достаточно легко, когда программу сначала пишут на бумаге \$, но при модификации нетривиальной программы, особенно написанной другим человеком, очень трудно избежать и того, и другого. Часто от этих сложностей становится труднее уберечься из-за того, что функции вроде print()

часто бывают организованы так, чтобы использовать общность классов, с которыми они работают. Например:

```
void print_employee(employee* e)
{
    cout << e->name << «\t» << e->department
<< «\n»;
    // ...
    if (e->type == M) {
        manager* p = (manager*)e;
        cout << « уровень » << p->level << «\n»;
        // ...
    }
}
```

Отыскание всех таких операторов if, скрытых внутри большой функции, которая работает с большим числом производных классов, может оказаться сложной задачей, и даже когда все они найдены, бывает нелегко понять, что же в них делается.

7.2.8. Виртуальные функции

Виртуальные функции преодолевают сложности решения с помощью полей типа, позволяя программисту описывать в базовом классе функции, которые можно переопределять в любом производном классе. Компилятор и загрузчик обеспечивают правильное соответствие между объектами и применяемыми к ним функциями. Например:

```
struct employee {
    employee* next;
    char*      name;
    short      department;
    // ...
    virtual void print();
};
```

Ключевое слово `virtual` указывает, что могут быть различные варианты функции `print()` для разных производных классов и что поиск среди них подходящей для каждого вызова `print()` является задачей компилятора. Тип функции описывается в базовом классе и не может переписываться в производном классе. Виртуальная функция должна быть определена для класса, в котором она описана впервые. Например:

```
void employee::print()
```

```

    {
        cout << e->name << «\t» << e->department
<< «\n»;
        // ...
    }

```

Виртуальная функция может, таким образом, использоваться даже в том случае, когда нет производных классов от ее класса, и в производном классе, в котором не нужен специальный вариант виртуальной функции, ее задавать не обязательно. Просто при выводе класса соответствующая функция задается в том случае, если она нужна. Например:

```

    struct manager : employee {
        employee* group;
        short      level;
        // ...
        void print();
    };

    void manager::print()
    {
        employee::print();
        cout << «\tуровень» << level << «\n»;
        // ...
    }

```

Функция `print_employee()` теперь не нужна, поскольку ее место заняли функции члены `print()`, и теперь со списком служащих можно работать так:

```

    void f(employee* ll)
    {
        for (; ll; ll=ll->next) ll->print();
    }

```

Каждый служащий будет печататься в соответствии с его типом. Например:

```

    main()
    {
        employee e;
        e.name = «Дж.Браун»;
        e.department = 1234;
        e.next = 0;
    }

```

```

manager m;
    m.name = «Дж.Смит»;
    e.department = 1234;
    m.level = 2;
    m.next = &e;
    f(&m);
}

```

выдаст

```

Дж.Смит 1234
        уровень 2
Дж.Браун 1234

```

Заметьте, что это будет работать даже в том случае, если `f()` была написана и откомпилирована еще до того, как производный класс `manager` был задуман! Очевидно, при реализации этого в каждом объекте класса `employee` сохраняется некоторая информация о типе. Занимаемого для этого пространства (в текущей реализации) как раз хватает для хранения указателя. Это пространство занимает только в объектах классов с виртуальными функциями, а не во всех объектах классов и даже не во всех объектах производных классов. Вы платите эту пошлину только за те классы, для которых описали виртуальные функции.

Вызов функции с помощью операции разрешения области видимости `::`, как это делается в `manager::print()`, гарантирует, что механизм виртуальных функций применяться не будет. Иначе `manager::print()` подвергалось бы бесконечной рекурсии. Применение уточненного имени имеет еще один эффект, который может оказаться полезным: если описанная как `virtual` функция описана еще и как `inline` (в чем ничего необычного нет), то там, где в вызове применяется `::`, может применяться `inline`-подстановка. Это дает программисту эффективный способ справляться с теми важными специальными случаями, когда одна виртуальная функция вызывает другую для того же объекта. Поскольку тип объекта был определен при вызове первой виртуальной функции, обычно его не надо снова динамически определять при другом вызове для того же объекта.

7.3. Альтернативные интерфейсы

После того как описаны средства языка, которые относятся к производным классам, обсуждение снова может вернуться к стоящим задачам. В классах, которые описываются в этом разделе, основопола-

гающая идея состоит в том, что они однажды написаны, а потом их используют программисты, которые не могут изменить их определение. Физически классы состоят из одного или более заголовочных файлов, определяющих интерфейс, и одного или более файлов, определяющих реализацию. Заголовочные файлы будут помещены туда, откуда пользователь может взять их копии с помощью директивы `#include`. Файлы, определяющие реализацию, обычно компилируют и помещают в библиотеку.

7.3.1. Интерфейс

Рассмотрим такое написание класса `slist` для однократно связанного списка, с помощью которого можно создавать как однородные, так и неоднородные списки объектов тех типов, которые еще должны быть определены. Сначала мы определим тип `ent`:

```
typedef void* ent;
```

Точная сущность типа `ent` несущественна, но нужно, чтобы в нем мог храниться указатель. Тогда мы определим тип `slink`:

```
class slink {
friend class slist;
friend class slist_iterator;
    slink* next;
    ent e;
    slink(ent a, slink* p) { e=a; next=p;}
};
```

В одном звене может храниться один `ent`, и с помощью него реализуется класс `slist`:

```
class slist {
friend class slist_iterator;
    slink* last;           // last->next - голова
списка

public:
    int insert(ent a);     // добавить в голову
списка
    int append(ent a);     // добавить в хвост
списка
    ent get();             // вернуться и убрать
голову списка
    void clear();          // убрать все звенья
```



```

        slist()          { last=0; }
        slist(ent a) { last=new slink(a,0); last-
>next=last; }
        ~slist()        { clear(); }
    };

```

Хотя список очевидным образом реализуется как связанный список, реализацию можно изменить так, чтобы использовался вектор из ent'ов, не повлияв при этом на пользователей. То есть, применение slink'ов никак не видно в описаниях открытых функций slist'ов, а видно только в закрытой части и определениях функций.

7.3.2. Реализация

Реализующие slist функции в основном просты. Единственная настоящая сложность - что делать в случае ошибки, если, например, пользователь попытается получить (метод get()) что-нибудь из пустого списка. Здесь приводятся определения членов slist. Обратите внимание, как хранение указателя на последний элемент кругового списка дает возможность просто реализовать оба действия append() и insert():

```

int slist::insert(ent a)
{
    if (last)
        last->next = new slink(a,last->next);
    else {
        last = new slink(a,0);
        last->next = last;
    }
    return 0;
}

int slist::append(ent a)
{
    if (last)
        last = last->next = new slink(a,last-
>next);
    else {
        last = new slink(a,0);
        last->next = last;
    }
    return 0;
}

```

```

    }

    ent slist::get()
    {
        if (last == 0)    slist_handler(«get
fromempty list»);
                                                                    //    взять    из
пустого списка
        slink* f = last->next;
        ent r  f->e;
        if (f == last)
            last = 0;
        else
            last->next = f->next;
        delete f;
        return f;
    }

```

Обратите внимание, как вызывается `slist_handler`. Этот указатель на имя функции используется точно так же, как если бы он был именем функции. Это является краткой формой более явной записи вызова:

```
(*slist_handler)(«get fromempty list»);
```

И `slist::clear()`, наконец, удаляет из списка все элементы:

```

void slist::clear()
{
    slink* l = last;
    if (l == 0) return;
    do {
        slink* ll = l;
        l = l->next;
        delete ll;
    } while (l!=last);
}

```

Класс `slist` не обеспечивает способа заглянуть в список, но только средства для вставки и удаления элементов. Однако оба класса, и `slist`, и `slink`, описывают класс `slist_iterator` как друга, поэтому мы можем описать подходящий итератор. Вот один:

```

class slist_iterator {
    slink* ce;

```

```

        slist* cs;
    public:
        slist_iterator(slist& s) { cs = &s; ce =
cs->last; }

        ent operator() () {
            // для индикации конца итерации воз-
вращает 0
            // для всех типов не идеален, хорош
для указателей
            ent ret = ce ? (ce=ce->next)->e : 0;
            if (ce == cs->last) ce= 0;
            return ret;
        }
};

```

7.3.3. Как этим пользоваться

Фактически класс slist в написанном виде бесполезен. В конечном счете, зачем можно использовать список указателей void*? Штука в том, чтобы вывести класс из slist и получить список тех объектов, которые представляют интерес в конкретной программе. Представим компилятор языка вроде C++. В нем широко будут использоваться списки имен; имя - это нечто вроде

```

struct name {
    char* string;
    // ...
};

```

В список будут помещаться указатели на имена, а не сами объекты имена. Это позволяет использовать небольшое информационное поле с slist'a, и дает возможность имени находиться одновременно более, чем в одном списке. Вот определение класса nlist, который очень просто выводится из класса slist:

```

#include «slist.h»
#include «name.h»

struct nlist : slist {
    void insert(name* a) { slist::insert(a); }
    void append(name* a) { slist::append(a); }
    name* get()          {}
    nlist(name* a) : (a) {}
};

```

Функции нового класса или наследуются от `slist` непосредственно, или ничего не делают, кроме преобразования типа. Класс `nlist` - это не что иное, как альтернативный интерфейс класса `slist`. Так как на самом деле тип `ent` есть `void*`, нет необходимости явно преобразовывать указатели `name*`, которые используются в качестве фактических параметров.

Списки имен можно использовать в классе, который представляет определение класса:

```
struct classdef {
    nlist friends;
    nlist constructors;
    nlist destructors;
    nlist members;
    nlist operators;
    nlist virtuals;
    // ...
    void add_name(name*);
    classdef();
    ~classdef();
};
```

и имена могут добавляться к этим спискам приблизительно так:

```
void classdef::add_name(name* n)
{
    if (n->is_friend()) {
        if (find(&friends,n))
            error(«friend redeclared»);
        else if (find(&members,n))

            error(«friend redeclared as mem-
ber»);
        else
            friends.append(n);
    }
    if (n->is_operator()) operators.append(n);
    // ...
}
```

где `is_iterator()` и `is_friend()` являются функциями членами класса `name`. Функцию `find()` можно написать так:

```
int find(nlist* ll, name* n)
{
```

```

    slist_iterator ff(*(slist*)ll);
    ent p;
    while ( p=ff() ) if (p==n) return 1;
    return 0;
}

```

Здесь применяется явное преобразование типа, чтобы применить `slist_iterator` к `nlist`. Более хорошее решение - сделать итератор для `nlist`'ов. Печатать `nlist` может, например, такая функция:

```

void print_list(nlist* ll, char* list_name)
{
    slist_iterator count(*(slist*)ll);
    name* p;
    int n = 0;
    while ( count() ) n++;
    cout << list_name << «\n» << n << «members\n»;
    slist_iterator print(*(slist*)ll);
    while ( p=(name*)print() ) cout << p->string << «\n»;
}

```

7.3.4. Ограниченные интерфейсы

Класс `slist` довольно общего характера. Иногда подобная общность не требуется или даже нежелательна. Ограниченные виды списков, такие как стеки и очереди, даже более обычны, чем сам обобщенный список. Такие структуры данных можно задать, не описав базовый класс как открытый. Например, очередь целых можно определить так:

```

#include «slist.h»

class iqueue : slist {
    //предполагается
    sizeof(int)<=sizeof(void*)
public:
    void put(int a) { slist::append((void*)a); }
    int det() { return
int(slist::get()); }
    iqueue() {}
};

```

При таком выводе осуществляются два логически разделенных действия: понятие списка ограничивается понятием очереди (сводится к нему) и задается тип `int`, чтобы свести понятие очереди к типу данных очередь целых, `iqueue`. Эти два действия можно выполнять и раздельно. Здесь первая часть - это список, ограниченный так, что он может использоваться только как стек:

```
#include «slist.h»

class stack : slist {
public:
    slist::insert;
    slist::get;
    stack() {}
    stack(ent a) : (a) {}
};
```

который потом используется для создания типа «стек указателей на символы»:

```
#include «stack.h»

class cp : stack {
public:
    void push(char* a) { slist::insert(a); }
    char* pop() { return (char*)slist::get(); }
    nlist() {}
};
```

7.4. Добавление к классу

В предыдущих примерах производный класс ничего не добавлял к базовому классу. Для производного класса функции определялись, только чтобы обеспечить преобразование типа. Каждый производный класс просто задавал альтернативный интерфейс к общему множеству программ. Этот специальный случай важен, но наиболее обычная причина определения новых классов как производных классов в том, что кто-то хочет иметь то, что предоставляет базовый класс, плюс еще чуть-чуть.

Для производного класса можно определить данные и функции дополнительно к тем, которые наследуются из его базового класса. Это дает альтернативную стратегию обеспечить средства связанного списка. Заметьте, когда в тот `slist`, который определялся выше, помещается элемент, то создается `slink`, содержащий два указателя. На их создание

тратится время, а ведь без одного из указателей можно обойтись, при условии, что нужно только чтобы объект мог находиться в одном списке. Так что указатель `next` на следующий можно поместить в сам объект, вместо того, чтобы помещать его в отдельный объект `slink`. Идея состоит в том, чтобы создать класс `olink` с единственным полем `next` и класс `olist`, который может обрабатывать указателями на такие звенья `olink`. Тогда `olist` сможет манипулировать объектами любого класса, производного от `olink`. Буква «о» в названиях стоит для того, чтобы напоминать вам, что объект может находиться одновременно только в одном списке `olist`:

```
struct olink {
    olink* next;
};
```

Класс `olist` очень напоминает класс `slist`. Отличие состоит в том, что пользователь класса `olist` манипулирует объектами класса `olink` непосредственно:

```
class olist {
    olink* last;
public:
    void insert(olink* p);
    void append(olink* p);
    olink* get();
    // ...
};
```

Мы можем вывести из класса `olink` класс `name`:

```
class name : public olink {
    // ...
};
```

Теперь легко сделать список, который можно использовать без накладных расходов времени на размещение или памяти.

Объекты, помещаемы в `olist`, теряют свой тип. Это означает, что компилятор знает только то, что они `olink`'и. Правильный тип можно восстановить с помощью явного преобразования типа объектов, вынутых из `olist`. Например:

```
void f()
{
    olist ll;
    name nn;
```

```

        ll.insert(&nn);                                // тип &nn
потерян
        name* pn = (name*)ll.get(); // и восста-
новлен
    }

```

Другой способ: тип можно восстановить, выведя еще один класс из olist для обработки преобразования типа:

```

class olist : public olist {
    // ...
    name* get() { return (name*)olist::get(); }
};

```

Имя name может одновременно находиться только в одном olist. Для имен это может быть и не подходит, но в классах, для которых это подойдет полностью, недостатка нет. Например, класс фигур shape использует для поддержки списка всех фигур именно этот метод. Обратите внимание, что можно было бы определить slist как производный от olist, объединяя таким образом оба понятия. Однако использование базовых и производных классов на таком микроскопическом уровне может очень сильно исказить код.

7.5. Неоднородные списки

Предыдущие списки были однородными. То есть в список помещались только объекты одного типа. Это обеспечивалось аппаратом производных классов. Списки не обязательно должны быть однородными. Список, заданный в виде указателей на класс, может содержать объекты любого класса, производного от этого класса. То есть список может быть неоднородным. Вероятно, это единственный наиболее важный и полезный аспект производных классов, и он весьма существенно используется в стиле программирования, который демонстрируется приведенным выше примером. Этот стиль программирования часто называют объектно-основанным или объектно-ориентированным. Он опирается на то, что действия над объектами неоднородных списков выполняются одинаковым образом. Смысл этих действий зависит от фактического типа объектов, находящихся в списке (что становится известно только на стадии выполнения), а не просто от типа элементов списка (который компилятору известен).

8. ГЛАВА ВОСЬМАЯ. ПОТОКИ

Язык C++ не обеспечивает средств для ввода/вывода. Ему это и не нужно; такие средства легко и элегантно можно создать с помощью самого языка. Описанная здесь стандартная библиотека потокового ввода/вывода обеспечивает гибкий и эффективный с гарантией типа метод обработки символьного ввода целых чисел, чисел с плавающей точкой и символьных строк, а также простую модель ее расширения для обработки типов, определяемых пользователем. Ее пользовательский интерфейс находится в 1.1. В этой главе описывается сама библиотека, некоторые способы ее применения и методы, которые использовались при ее реализации.

8.1. Введение

Разработка и реализация стандартных средств ввода/вывода для языка программирования зарекомендовала себя как заведомо трудная работа. Традиционно средства ввода/вывода разрабатывались исключительно для небольшого числа встроенных типов данных. Однако в C++-программах обычно используется много типов, определенных пользователем, и нужно обрабатывать ввод и вывод также и значений этих типов. Очевидно, средство ввода/вывода должно быть простым, удобным, надежным в употреблении, эффективным и гибким, и ко всему прочему полным. Ничье решение еще не смогло угодить всем, поэтому у пользователя должна быть возможность задавать альтернативные средства ввода/вывода и расширять стандартные средства ввода/вывода применительно к требованиям приложения.

C++ разработан так, чтобы у пользователя была возможность определять новые типы столь же эффективные и удобные, сколь и встроенные типы. Поэтому обоснованным является требование того, что средства ввода/вывода для C++ должны обеспечиваться в C++ с применением только тех средств, которые доступны каждому программисту. Описываемые здесь средства ввода/вывода представляют собой попытку ответить на этот вызов.

Средства ввода/вывода связаны исключительно с обработкой преобразования типизированных объектов в последовательности символов и обратно. Есть и другие схемы ввода/вывода, но эта является основополагающей в системе UNIX, и большая часть видов бинарного ввода/вывода обрабатывается через рассмотрение символа просто как набор бит, при этом его общепринятая связь с алфавитом игнорируется. Тогда для программиста ключевая проблема заключается в задании

соответствия между типизированным объектом и принципиально не типизированной строкой.

Обработка и встроенных и определенных пользователем типов однородным образом и с гарантией типа достигается с помощью одного перегруженного имени функции для набора функций вывода. Например:

```
put(cerr, »x = «); // cerr - поток вывода ошибок
put(cerr, x);
put(cerr, »\n»);
```

Тип параметра определяет то, какая из функций put будет вызываться для каждого параметра. Это решение применялось в нескольких языках. Однако ему недостает лаконичности. Перегрузка операции << значением «поместить в» дает более хорошую запись и позволяет программисту выводить ряд объектов одним оператором. Например:

```
cerr << «x = « << x << »\n»;
```

где cerr - стандартный поток вывода ошибок. Поэтому, если x является int со значением 123, то этот оператор напечатает в стандартный поток вывода ошибок

```
x = 123
```

и символ новой строки. Аналогично, если X принадлежит определенному пользователем типу complex и имеет значение (1,2.4), то приведенный выше оператор напечатает в cerr

```
x = 1,2.4)
```

Этот метод можно применять всегда, когда для x определена операция << и пользователь может определять операцию << для нового типа.

8.2. Вывод

В этом разделе сначала обсуждаются средства форматного и бесформатного вывода встроенных типов, потом приводится стандартный способ спецификации действий вывода для определяемых пользователем типов.

8.2.1. Вывод встроенных типов

Класс ostream определяется вместе с операцией << («поместить в») для обработки вывода встроенных типов:

```

class ostream {
    // ...
public:
    ostream& operator<<(char*);
    ostream& operator<<(int i) { return *this<

```

8.2.2. Некоторые подробности разработки

Операция вывода используется, чтобы избежать той многословности, которую дало бы использование функции вывода. Но почему `<<`? Возможности изобрести новый лексический символ нет. Операция присваивания была кандидатом одновременно и на ввод, и на вывод, но оказывается, большинство людей предпочитают, чтобы операция ввода отличалась от операции вывода. Кроме того, `=` не в ту сторону связывается (ассоциируется), то есть `cout=a=b` означает `cout=(a=b)`.

Делались попытки использовать операции `<` и `>`, но значения «меньше» и «больше» настолько прочно вросли в сознание людей, что новые операции ввода/вывода во всех реальных случаях оказались нечитаемыми. Помимо этого, «<<» находится на большинстве клавиатур как раз на «,>», и у людей получаются операторы вроде такого:

```
cout < x , y , z;
```

Для таких операторов непросто выдать хорошие сообщения об ошибках.

Операции `<<` и `>>` к такого рода проблемам не приводят, они асимметричны в том смысле, что их можно проассоциировать с «в» и «из», а приоритет `<<` достаточно низок, чтобы можно было не использовать скобки для арифметических выражений в роли операндов. Например:

```
cout << «a*b+c=» << a*b+c << «\n»;
```

Естественно, при написании выражений, которые содержат операции с более низкими приоритетами, скобки использовать надо. Например:

```
cout << «a^b|c=» << (a^b|c) << «\n»;
```

Операцию левого сдвига тоже можно применять в операторе вывода:

```
cout << «a»;
```

8.2.3. Форматированный вывод

Пока << применялась только для неформатированного вывода, и на самом деле в реальных программах она именно для этого главным образом и применяется. Помимо этого, существует также несколько форматирующих функций, создающих представление своего параметра в виде строки, которая используется для вывода. Их второй (необязательный) параметр указывает, сколько символьных позиций должно использоваться.

```
char* oct(long, int =0);          // восьмеричное
представление
char* dec(long, int =0);          // десятичное
представление
char* hex(long, int =0);          // шестнадцатерич-
ное представление
char* chr(int, int =0);           // символ
char* str(char*, int =0);         // строка
```

Если не задано поле нулевой длины, то будет производиться усе-
чение или дополнение; иначе будет использоваться столько символов
(ровно), сколько нужно. Например:

```
cout << «dec(« << x
      << «) = oct(« << oct(x,6)
      << «) = hex(« << hex(x,4)
      << «)»;
```

Если x==15, то в результате получится:

```
dec(15) = oct(  17) = hex(  f);
```

Можно также использовать строку в общем формате:

```
char* form(char* format ...);
cout<
```

8.2.4. Виртуальная функция вывода

Иногда функция вывода должна быть virtual. Рассмотрим пример класса shape, который дает понятие фигуры:

```
class shape {
    // ...
public:
    // ...
```

```

        virtual void draw(ostream& s); // рисует
«this» на «s»
    };

    class circle : public shape {
        int radius;
    public:
        // ...
        void draw(ostream&);
    };

```

То есть круг имеет все признаки фигуры и может обрабатываться как фигура, но имеет также и некоторые специальные свойства, которые должны учитываться при его обработке.

Чтобы поддерживать для таких классов стандартную парадигму вывода, операция << определяется так:

```

ostream& operator<<(ostream& s, shape* p)
{
    p->draw(s);
    return s;
}

```

Если next - итератор типа, то список фигур распечатывается, например, так:

```
while ( p = next() ) cout << p;
```

8.3. Файлы и потоки

Потоки обычно связаны с файлами. Библиотека потоков создает стандартный поток ввода cin, стандартный поток вывода cout и стандартный поток ошибок cerr. Программист может открывать другие файлы и создавать для них потоки.

8.3.1. Инициализация потоков вывода

ostream имеет конструкторы:

```

class ostream {
    // ...
    ostream(streambuf* s);           // связывает с
буфером потока
    ostream(int fd);                 // связывание
для файла

```

```
ostream(int size, char* p); // связывает с
вектором
};
```

Главная работа этих конструкторов - связывать с потоком буфер. `streambuf` - класс, управляющий буферами, как и класс `filebuf`, управляющий `streambuf` для файла. Класс `filebuf` является производным от класса `streambuf`.

Описание стандартных потоков вывода `cout` и `cerr`, которое находится в исходных кодах библиотеки потоков ввода/вывода, выглядит так:

```
// описать подходящее пространство буфера
char cout_buf[BUFSIZE]

// сделать «filebuf» для управления этим
пространством
// связать его с UNIX'овским потоком выво-
да 1 (уже открытым)
filebuf cout_file(1,cout_buf,BUFSIZE);

// сделать ostream, обеспечивая пользова-
ТЕЛЬСКИЙ интерфейс
ostream cout(&cout_file);

char cerr_buf[1];

// длина 0, то есть, небуферизованный
// UNIX'овский поток вывода 2 (уже откры-
тый)
filebuf cerr_file()2,cerr_buf,0;

ostream cerr(&cerr_file);
```

8.3.2. Закрытие потоков вывода

Деструктор для `ostream` сбрасывает буфер с помощью открытого члена функции `ostream::flush()`:

```
ostream::~ostream()
{
    flush(); // сброс
}
```

Сбросить буфер можно также и явно. Например:

```
cout.flush();
```

8.3.3. Открытие файлов

Точные детали того, как открываются и закрываются файлы, различаются в разных операционных системах и здесь подробно не описываются. Поскольку после включения становятся доступны `cin`, `cout` и `cerr`, во многих (если не во всех) программах не нужно держать код для открытия файлов. Вот, однако, программа, которая открывает два файла, заданные как параметры командной строки, и копирует первый во второй:

```
#include

void error(char* s, char* s2)
{
    cerr << s << « « << s2 << «\n»;
    exit(1);
}

main(int argc, char* argv[])
{
    if (argc != 3) error(«неверное число параметров»,»«);

    filebuf f1;
    if (f1.open(argv[1],input) == 0)
        error(«не могу открыть входной
файл»,argv[1]);
    istream from(&f1);

    filebuf f2;
    if (f2.open(argv[2],output) == 0)
        error(«не могу создать выходной
файл»,argv[2]);
    ostream to(&f2);

    char ch;
    while (from.get(ch)) to.put(ch);

    if (!from.eof() !! to.bad())
        error(«случилось нечто странное»,»«);
}
```

Последовательность действий при создании ostream для именованного файла та же, что используется для стандартных потоков: 1) сначала создается буфер (здесь это делается посредством описания filebuf); 2) затем к нему подключается файл (здесь это делается посредством открытия файла с помощью функции filebuf::open()); и, наконец, 3) создается сам ostream с filebuf в качестве параметра. Потоки ввода обрабатываются аналогично.

Файл может открываться в одной из двух мод:

```
enum open_mode { input, output };
```

Действие filebuf::open() возвращает 0, если не может открыть файл в соответствии с требованием. Если пользователь пытается открыть файл, которого не существует для output, он будет создан.

Перед завершением программа проверяет, находятся ли потоки в приемлемом состоянии. При завершении программы открытые файлы неявно закрываются.

Файл можно также открыть одновременно для чтения и записи, но в тех случаях, когда это оказывается необходимо, парадигма потоков редко оказывается идеальной. Часто лучше рассматривать такой файл как вектор (гигантских размеров). Можно определить тип, который позволяет программе обрабатывать файл как вектор (см. Упражнения 8-10).

8.3.4. Копирование потоков

Есть возможность копировать потоки. Например:

```
cout = cerr;
```

В результате этого получают две переменные, ссылающиеся на один и тот же поток. Главным образом это бывает полезно для того, чтобы сделать стандартное имя вроде cin ссылающимся на что-то другое.

8.4. Ввод

Ввод аналогичен выводу. Имеется класс istream, который предоставляет операцию >> («взять из») для небольшого множества стандартных типов. Функция operator>> может определяться для типа, определяемого пользователем.

8.4.1. Ввод встроенных типов

Класс `istream` определяется так:

```
class istream {
    // ...
public:
    istream& operator>>(char*);           // строка
    istream& operator>>(char&);          // символ
    istream& operator>>(short&);
    istream& operator>>(int&);
    istream& operator>>(long&);
    istream& operator>>(float&);
    istream& operator>>(double&);
    // ...
};
```

Функции ввода определяются в таком духе:

```
istream& istream::operator>>(char& c);
{
    // пропускает пропуски
    int a;
    // неким образом читает символ в «a»
    c = a;
}
```

Пропуск определяется как стандартный пропуск в C, через вызов `isspace()` в том виде, как она определена в стандартной библиотеке (пробел, табуляция, символ новой строки, перевод формата и возврат каретки).

В качестве альтернативы можно использовать функции `get()`:

```
class istream {
    // ...
    istream& get(char& c);
// char
    istream& get(char* p, int n, int = '\n');
// строка
};
```

Они обрабатывают символы пропуска так же, как остальные символы. Функция `istream::get(char)` читает один и тот же символ в свой параметр; другая `istream::get` читает не более `n` символов в вектор символов, начинающийся в `p`. Необязательный третий параметр использу-

ется для задания символа остановки (иначе, терминатора или ограничителя), то есть этот символ читаться не будет. Если будет встречен символ ограничитель, он останется как первый символ потока. По умолчанию вторая функция `get` будет читать самое большее `n` символов, но не больше, чем одну строку, `'\n'` является ограничителем по умолчанию. Необязательный третий параметр задает символ, который читаться не будет. Например:

```
cin.get(buf, 256, '\t');
```

будет читать в `buf` не более 256 символов, а если встретится табуляция (`'\t'`), то это приведет к возврату из `get`. В этом случае следующим символом, который будет считан из `cin`, будет `'\t'`.

Стандартный заголовочный файл определяет несколько функций, которые могут оказаться полезными при осуществлении ввода:

```
int isalpha(char)    // 'a'..'z' 'A'..'Z'
int isupper(char)    // 'A'..'Z'
int islower(char)    // 'a'..'z'
int isdigit(char)     // '0'..'9'
int isxdigit(char)    // '0'..'9' 'a'..'f' 'A'..'F'
int isspace(char)     // ' ' '\t' возврат новая
строка
int iscntrl(char)     // перевод формата
// управляющий символ
// (ASCII 0..31 и 127)
int ispunct(char)     // пунктуация: ни один из
вышеперечисленных
int isalnum(char)     // isalpha() | isdigit()
int isprint(char)     // печатаемый: ascii '
'..'_'
int isgraph(char)     // isalpha() | isdigit()
| ispunct()
int isascii(char c) { return 0<=c && c<=127; }
```

Все, кроме `isascii()`, реализуются внешне одинаково, с применением символа в качестве индекса в таблице атрибутов символов. Поэтому такие выражения, как

```
(( 'a' <= c && c <= 'z' ) || ( 'A' <= c && c <= 'Z' )) //
алфавитный
```

не только утомительно пишутся и чреваты ошибками (на машине с набором символов EBCDIC оно будет принимать неалфавитные символы), они также и менее эффективны, чем применение стандартной функции:

```
isalpha(c)
```

8.4.2. Состояния потока

Каждый поток (istream или ostream) имеет ассоциированное с ним состояние, и обработка ошибок и нестандартных условий осуществляется с помощью соответствующей установки и проверки этого состояния.

Поток может находиться в одном из следующих состояний:

```
enum stream_state { _good, _eof, _fail, _bad };
```

Если состояние `_good` или `_eof`, значит последняя операция ввода прошла успешно. Если состояние `_good`, то следующая операция ввода может пройти успешно, в противном случае она закончится неудачей. Другими словами, применение операции ввода к потоку, который не находится в состоянии `_good`, является пустой операцией. Если делается попытка читать в переменную `v` и операция оканчивается неудачей, значение `v` должно остаться неизменным (оно будет неизменным, если `v` имеет один из тех типов, которые обрабатываются функциями членами istream или ostream). Отличия между состояниями `_fail` и `_bad` очень незначительно и представляет интерес только для разработчиков операций ввода. В состоянии `_fail` предполагается, что поток не испорчен и никакие символы не потеряны. В состоянии `_bad` может быть все, что угодно.

Состояние потока можно проверять, например, так:

```
switch (cin.rdstate()) {
    case _good:
        // последняя операция над cin прошла успешно
        break;
    case _eof:
        // конец файла
        break;
    case _fail:
        // некоего рода ошибка форматирования
        // возможно, не слишком плохая
        break;
    case _bad:
        // возможно, символы cin потеряны
        break;
}
```

Для любой переменной `z`-типа, для которого определены операции `<<` и `>>`, копирующий цикл можно написать так:

```
while (cin>>z) cout << z << «\n»;
```

Например, если `z` - вектор символов, этот цикл будет брать стандартный ввод и помещать его в стандартный вывод по одному слову (то есть последовательности символов без пробела) на строку.

Когда в качестве условия используется поток, происходит проверка состояния потока и эта проверка проходит успешно (то есть, значение условия не ноль), только если состояние `_good`. В частности, в предыдущем цикле проверялось состояние `istream`, которое возвращает `cin >> z`. Чтобы обнаружить, почему цикл или проверка закончились неудачно, можно исследовать состояние. Такая проверка потока реализуется операцией преобразования.

Делать проверку на наличие ошибок каждого ввода или вывода действительно не очень удобно, и обычно источником ошибок служит программист, не сделавший этого в том месте, где это существенно. Например, операции вывода обычно не проверяются, но они могут случайно не сработать. Парадигма потока ввода/вывода построена так, чтобы когда в C++ появится (если это произойдет) механизм обработки исключительных ситуаций (как средство языка или как стандартная библиотека), его будет легко применить для упрощения и стандартизации обработки ошибок в потоках ввода/вывода.

8.4.3. Ввод типов, определяемых пользователем

Ввод для пользовательского типа может определяться точно так же, как вывод, за тем исключением, что для операции ввода важно, чтобы второй параметр был ссылочного типа. Например:

```
istream& operator>>(istream& s, complex& a)
/*
    форматы ввода для complex; «f» обозначает float:
    f
    ( f )
    ( f , f )
*/
{
    double re = 0, im = 0;
    char c = 0;

    s >> c;
    if (c == '(') {
        s >> re >> c;
        if (c == ',') s >> im >> c;
        if (c != ')') s.clear(_bad);    // установить state
```

```

    }
    else {
        s.putback(c);
        s >> re;
    }

    if (s) a = complex(re,im);
    return s;
}

```

Несмотря на то, что не хватает кода обработки ошибок, большую часть видов ошибок это на самом деле обрабатывать будет. Локальная переменная `s` инициализируется, чтобы ее значение не оказалось случайно `'(` после того, как операция окончится неудачно. Завершающая проверка состояния потока гарантирует, что значение параметра `a` будет изменяться только в том случае, если все идет хорошо.

Операция установки состояния названа `clear()` (очистить), потому что она чаще всего используется для установки состояния потока заново как `_good`. `_good` является значением параметра по умолчанию и для `istream::clear()`, и для `ostream::clear()`.

Над операциями ввода надо поработать еще. Было бы, в частности, замечательно, если бы можно было задавать ввод в терминах шаблона (как в языках Снобол и Икон), а потом проверять, прошла ли успешно вся операция ввода. Такие операции должны были бы, конечно, обеспечивать некоторую дополнительную буферизацию, чтобы они могли восстанавливать поток ввода в его исходное состояние после неудачной попытки распознавания.

8.4.4. Инициализация потоков ввода

Естественно, тип `istream`, так же как и `ostream`, снабжен конструктором:

```

class istream {
    // ...
    istream(streambuf* s, int sk =1, ostream*
t =0);
    istream(int size, char* p, int sk =1);
    istream(int fd, int sk =1, ostream* t =0);
};

```

Параметр `sk` задает, должны пропускаться пропуски или нет. Параметр `t` (необязательный) задает указатель на `ostream`, к которому

прикреплен istream. Например, cin прикреплен к cout; это значит, что перед тем, как попытаться читать символы из своего файла, cin выполняет

```
cout.flush(); // пишет буфер вывода
```

С помощью функции istream::tie() можно прикрепить (или открепить, с помощью tie(0)) любой ostream к любому istream. Например:

```
int y_or_n(ostream& to, istream& from)
/*
    «to», получает отклик из «from»
*/
{
    ostream* old = from.tie(&to);
    for (;;) {
        cout << «наберите Y или N: «;
        char ch = 0;
        if (!cin.get(ch)) return 0;

        if (ch != '\n') { // пропускает остаток строки
            char ch2 = 0;
            while (cin.get(ch2) && ch2 != '\n') ;
        }
        switch (ch) {
            case 'Y':
            case 'y':
            case '\n':
                from.tie(old); // восстанавливает старый tie
                return 1;
            case 'N':
            case 'n':
                from.tie(old); // восстанавливает старый tie
                return 0;
            default:
                cout << «извините, попробуйте еще раз: «;
        }
    }
}
```

Когда используется буферизованный ввод (как это происходит по умолчанию), пользователь не может, набрав только одну букву, ожидать отклика. Система ждет появления символа новой строки. `y_or_n()` смотрит на первый символ строки, а остальные игнорирует.

Символ можно вернуть в поток с помощью функции `istream::putback(char)`. Это позволяет программе «заглядывать вперед» в поток ввода.

8.5. Работа со строками

Можно осуществлять действия, подобные вводу/выводу, над символьным вектором, прикрепляя к нему `istream` или `ostream`. Например, если вектор содержит обычную строку, завершающуюся нулем, для печати слов из этого вектора можно использовать приведенный выше копирующий цикл:

```
void word_per_line(char v[], int sz)
/*
    печатет «v» размера «sz» по одному слову
на строке
*/
{
    istream ist(sz,v); // сделать istream для v
    char b2[MAX];      // больше наибольшего
слова
    while (ist>>b2) cout << b2 << «\n»;
}
```

Завершающий нулевой символ в этом случае интерпретируется как символ конца файла.

В помощь `ostream` можно отформатировать сообщения, которые не нужно печатать тотчас же:

```
char* p = new char[message_size];
ostream ost(message_size,p);
do_something(arguments,ost);
display(p);
```

Такая операция, как `do_something`, может писать в поток `ost`, передавать `ost` своим подоперациям и т.д. с помощью стандартных операций вывода. Нет необходимости делать проверку на переполнение, поскольку `ost` знает свою длину и, когда он будет переполняться, он будет переходить в состояние `_fail`. И, наконец, `display` может писать сообщения в «настоящий» поток вывода. Этот метод может оказаться

наиболее полезным, чтобы справляться с ситуациями, в которых окончательное отображение данных включает в себя нечто более сложное, чем работу с традиционным построчным устройством вывода. Например, текст из `ost` мог бы помещаться в располагающуюся где-то на экране область фиксированного размера.

8.6. Буферизация

При задании операций ввода/вывода мы никак не касались типов файлов, но ведь не все устройства можно рассматривать одинаково с точки зрения стратегии буферизации. Например, для `ostream`, подключенного к символьной строке, требуется буферизация другого вида, нежели для `ostream`, подключенного к файлу. С этими проблемами можно справиться, задавая различные буферные типы для разных потоков в момент инициализации (обратите внимание на три конструктора класса `ostream`). Есть только один набор операций над этими буферными типами, поэтому в функциях обрабатывают переполнение сверху и снизу, виртуальные. Этого достаточно, чтобы справляться с необходимой в данное время стратегией буферизации. Это также служит хорошим примером применения виртуальных функций для того, чтобы сделать возможной однородную обработку логически эквивалентных средств с различной реализацией. Описание буфера потока в выглядит так:

```
struct streambuf {           // управление буфером
    потока

    char* base;              // начало буфера
    char* pptr;              // следующий свободный char
    char* qptr;              // следующий заполнен-
    ный char
    char* eptr;              // один из концов бу-
    фера
    char  alloc;             // буфер, выделенный с
    помощью new

    // Опустошает буфер:
    // Возвращает EOF при ошибке и 0 в
    случае успеха
    virtual int overflow(int c =EOF);

    // Заполняет буфер
```



```

        // Возвращает EOF при ошибке или конце
ввода,
        // иначе следующий char
virtual int underflow();

int snextc()          // берет следующий
char
{
    return (++qptr==pptr) ? underflow() :
*qpтр&0377;
}

// ...

int allocate()        // выделяет некоторое
пространство буфера

streambuf() { /* ... */}
streambuf(char* p, int l) { /* ... */}
~streambuf() { /* ... */}
};

```

Обратите внимание, что здесь определяются указатели, необходимые для работы с буфером, поэтому обычные посимвольные действия можно определить (только один раз) в виде максимально эффективных inline-функций. Для каждой конкретной стратегии буферизации необходимо определять только функции переполнения `overflow()` и `underflow()`. Например:

```

struct filebuf : public streambuf {

    int fd;                // дескриптор файла
    char opened;           // файл открыт

    int overflow(int c =EOF);
    int underflow();

    // ...

    // Открывает файл:
    // если не срабатывает, то возвращает 0,
    // в случае успеха возвращает «this»
    filebuf* open(char *name, open_mode om);

```

```

int close() { /* ... */ }

filebuf() { opened = 0; }
filebuf(int nfd) { /* ... */ }
filebuf(int nfd, char* p, int l) : (p,l) {
/* ... */ }
~filebuf() { close(); }
};

int filebuf::underflow()           // заполняет
буфер из fd
{
    if (!opened || allocate()==EOF) return
EOF;

    int count = read(fd, base, eptr-base);
    if (count < 1) return EOF;

    qptr = base;
    pptr = base + count;
    return *qptr & 0377;
}

```

8.7. Эффективность

Можно было бы ожидать, что раз ввод/вывод определен с помощью общедоступных средств языка, он будет менее эффективен, чем встроенное средство. На самом деле это не так. Для действий вроде «поместить символ в поток» используются inline-функции, единственные необходимые на этом уровне вызовы функций возникают из-за переполнения сверху и снизу. Для простых объектов (целое, строка и т.п.) требуется по одному вызову на каждый. Как выясняется, это не отличается от прочих средств ввода/вывода, работающих с объектами на этом уровне.