


СТЕНЛИ ЛИПМАН

Езикът C++ в примери

A yellow starburst graphic with multiple points, containing the text "Object-Oriented Programming".

Object-Oriented
Programming

Драги читатели,
пред вас е първата от поредицата книги, посветени на
съвременните средства за програмиране, които започва да
издава “Колхида Трейд” КООП. Книгите са предназначени за
хора с интереси в областта на новите среди за програмиране.

Надяваме се, че с книгите от тази поредица ще получите
знания, които ще използвате в практиката.

© Copyright 1989 by AT&T Bell Laboratories

© Reprinted with corrections March, 1990

© Татяна Станева Илиева, преводач, 1993

© Илия Димитров Илиев, преводач, 1993

© Илия Димитров Илиев, оформление на корицата, 1993

© Всички права на това издание са запазени за “Колхида Трейд” КООП с изричното
съгласие на преводача

Печат “Дунав-прес” ООД – Русе

СТЕНЛИ ЛИПМАН

Езикът C++ в примери

Превод от английски език

Татяна Илиева

Илия Илиев

“Колхида Трейд” КООП

София, 1993

Съдържание

Предговор	11
Структура на книгата	11
Бележки по реализацията.....	12
Благодарности	12
ГЛАВА 0.....	14
0.1 Решаване на задачи.....	14
0.2 Програма на C++	15
0.3 Запознаване с организацията на вход-изхода	18
0.4 Коментари	21
0.5 Директиви на предпроцесора	22
ГЛАВА 1	25
1.1 Константи	26
1.2 Променливи.....	27
Какво представляват променливите?	29
Име на променлива.....	30
Дефиниране на променливи	31
1.3. Указатели.....	33
Указател към низ	36
1.4 Тип reference.....	40
1.5 Константен тип	41
1.6 Изброим тип данни.....	44
1.7 Масиви	45
Многомерни масиви	48
Връзка между указатели и масиви	50
1.8 Класове	52
1.9 Описанието typedef.....	61
ГЛАВА 2.....	63
2.1 Какво е това израз?	63
2.2 Аритметични операции.....	63
2.3 Операции за сравнение и логически операции.....	65
2.4 Операция за присвояване.....	66
2.5 Операциите "++" и "--"	68
2.6 Операция sizeof.....	71
2.7 Операция за условен израз (аритметичен if).....	72
2.8 Побитови операции	73

2.9 Приоритет на операциите	76
2.10 Преобразуване на типове	78
Автоматично преобразуване на типове	80
Преобразуване по желание на потребителя	81
2.11 Оператори	82
Съставни оператори и блокове	83
2.12 Управляващи оператори	83
2.13 Оператор if	83
2.14 Оператор switch	88
Оператори за цикъл	90
2.15 Оператор while	91
2.16 Оператор for	92
2.17 Оператор do	93
Оператори за преход	94
2.18 Оператор break	94
2.19 Оператор continue	95
2.20 Оператор goto	95
ГЛАВА 3	97
3.1 Рекурсия	99
3.2 Inline-функции	100
3.3 Строг контрол на типовете	101
3.4 Резултат на функция	102
3.5 Списък от аргументи	104
Синтаксис на списъка от аргументи	105
Специалната сигнатура "..." (Ellipses)	105
Сигнатура с инициализация	106
3.6 Предаване на параметри	108
3.7 Аргументи от тип reference	110
3.8 Масив като аргумент на функция	112
3.9 Области на действие	115
Операция за глобална дефиниция — "::"	116
Външни (extern) променливи	117
Статични (static) глобални променливи	118
3.10 Локална област на действие	119
Статични (static) локални променливи	121
Регистрови (register) локални променливи	122
ГЛАВА 4	123

4.1 Динамично разпределение на паметта	123
4.2 Клас IntList (линеен списък). Примерна реализация	128
4.3 Функции с еднакви имена.....	136
Защо са необходими функции с еднакви имена?	136
Как се интерпретират функции с еднакви имена?	137
Кога не се създават функции с еднакви имена?	138
Разрешаване на двусмислието при функции с общо име	140
Особености при точно съответствие	141
Особености при съответствие чрез стандартно преобразуване.....	144
Съпоставяне на аргументи от тип reference	145
Съпоставяне на функции с няколко аргумента	145
Съпоставяне при сигнатура с инициализация	146
Функции с еднакви имена и области на действие.....	146
Предефиниране на операция new.....	147
4.4 Указател към функция	148
Тип на указател към функция.....	148
Инициализация и присвояване	149
Обръщение към функция чрез указател към нея.....	150
Масив от указатели към функции	151
Указателят към функция като аргумент и резултат от функция	152
_new_handler.....	153
4.5 Особенности при свързване на програмни модули.....	154
Несъвместими декларации	155
Отново за заглавните файлове.....	156
Обръщения към функции, написани на друг програмен език.....	157
ГЛАВА 5.....	159
5.1 Дефиниране на класове	159
Членове-данни	159
Функции-членове на клас	161
Скриване на информация.....	162
5.2 Обекти.....	163
5.3 Функции-членове на клас	165
Управляващи методи.....	165
Инструментални методи	166
Помощни методи	167
Методи за достъп.....	169
Константни методи.....	170

5.4 Указател this	172
Какво представлява указателят this	173
Използване на указател this	174
5.5 Приятелски декларации	177
5.6 Статични членове	181
5.7 Указатели към членове на клас	184
Тип на членовете на клас	185
Използване на указатели към членове на клас	188
Указатели към статични членове	190
5.8 Област на действие при класове	191
Определяне областта на действие.....	193
Локални класове	195
5.9 Обединения — класове, спестяващи памет	196
5.10 Битови полета — членове, спестяващи памет	200
5.11 Предаване на обекти при сигнатура "... "	201
ГЛАВА 6.....	202
6.1 Инициализиране на обекти.....	202
Дефиниране на конструктор.....	203
Конструктори и скриване на информация	206
Деструктори	207
Масиви от обекти.....	210
Членове-обекти на клас.....	211
6.2 Инициализация чрез копиране на членове.....	215
Дефиниране на явен конструктор X(const X&)	217
X(const X&) и членове-обекти.....	218
Резюме върху конструктори и деструктори.....	220
6.3 Дефиниране на операции	220
Начални сведения за предефинирането на операции.....	221
Стандартни операции и класове.....	222
Операция за индексване "[]".....	224
Операция за обръщение към функция "()"	225
Операция new и операция delete	227
X::Operator=(const X&).....	231
Операция за избор на член "—>"	234
Определяне на необходимите операции	235
6.4 Примерна реализация на клас BitVector.....	236
6.5 Операции за преобразуване	244

Конструктори като операции за преобразуване	247
Операции за преобразуване	248
Противоречивост	251
ГЛАВА 7.....	255
7.1 Обектно-ориентирано програмиране.....	256
7.2 Клас ISortArray.....	259
7.3 Описание на зоологическа градина	262
7.4 Производни класове	264
Дефиниране на производни класове.....	265
Достъп до наследени членове.....	267
Инициализиране на основни класове	268
7.5 Производни класове и скриване на информация	270
7.6 Смисъл на атрибута public/private за основен клас	272
7.7 Стандартни преобразувания при производни класове	274
7.8 Области на действие и производни класове	278
Производни класове и функции с еднакви имена	279
7.9 Инициализация и присвояване при производни класове	280
7.10 Ред на инициализиране при производни класове.....	283
ГЛАВА 8.....	284
8.1 Функции с еднакви имена с аргумент обект от клас.....	284
Точно съответствие	284
Стандартни преобразувания	285
Преобразувания, дефинирани от потребителя.....	286
8.2 Виртуални функции	289
Динамичното свързване — вид капсулиране	289
Дефиниране на виртуални функции	290
Виртуални деструктори	297
Извикване на виртуална функция	298
8.3 Виртуални основни класове	306
Дефиниране на виртуални основни класове	308
Достъп до членове на виртуален основен клас.....	310
Ред на извикване на конструкторите и деструкторите	312
Смесване на виртуални и "обикновени" класове	314
Резюме	314
Приложение А.....	315
Входно-изходна библиотека в C++.....	315
А.1 Извеждане на информация	316

A.2 Дефиниране на операцията "<<" за класове.....	322
A.3 Въвеждане на информация	325
A.4 Дефиниране на операцията ">>" за класове.....	331
A.5 Четене и запис във файл.....	333
A.6 Флагове на състоянието	341
A.7 Флагове за формат	342
A.8 Операции с масиви от символи	347
Строг контрол на типовете	351
A.9 Резюме	352
Приложение Б	353
Насоки в развитието на C++	353
Б.1 Параметризирани типове.....	353
Б.2 Параметризирани класове	355
Параметризирани класове и наследяване.....	356
Б.3 Параметризирани функции	357
Приложение В	358
Съвместимост между С и C++	358
В.1 Прототип на функция в C++	359
В.2 Разлики между С и C++.....	362
Разлики, породени от по-голямата функционалност на C++	362
Разлики, породени от строгия контрол на типовете в C++	363
В.3 Свързване на модули, написани на С и C++	364
В.4 Преминаване от С на C++	365
Приложение Г	366
Съвместимост със стандарт 1.2.....	366
Г.1 Характеристики, които не се поддържат в стандарт 1.2	366
Множествено наследяване.....	367
Свързване с модули, написани на други езици	368
Статични членове	370
Г.2 Характеристики, променени в стандарт 2.0.....	371
Инициализиране и присвояване чрез побитово копиране.....	371
Дефиниране на функции с едно и също име.....	372
Литерални символни константи.....	375
Инициализиране на глобални променливи	376
Ограничения, свързани с декларациите на функции	376
Стойности по подразбиране за аргументите на функции.....	377
Изброими типове, локални за даден клас.....	377

Анонимни обединения в глобалната област на действие.....	377
Правила в стандарт 2.0, по-строги от правилата в стандарт 1.2	378
Ниво на достъп за операции, които са членове на клас.....	378
Интерпретиране на константите	378
Константен аргумент от тип reference	379
Справочник	380
Бележки по отпечатването.....	381

Предговор

Езикът C++ е разработен от AT&T Bell Laboratories в началото на 80-те години от Bjarne Stroustrup. Той представлява разширение на езика C в три важни направления:

1. създаване и използване на абстрактни типове данни
2. обектно-ориентирано програмиране
3. подобрения на конструкции, които съществуват в езика C

C++ запазва простотата и скоростта на изпълнение, характерни за езика C. Той е вече широко достъпен език, който се използва за разработване на реални приложения и системи. През първите шест месеца от появата му в края на 1985г. се появили над 24 търговски реализации на езика, предназначени за различни компютри.

Бързото разпространение на езика предизвика недостиг от обучаващи и образователни материали. Тази книга е една голяма стъпка в запълването на тази празнина. Доскоро основно справочно пособие беше книгата на Stroustrup "The C++ Programming Language", в която се описва първата версия на езика. От тогава C++ беше развит. Бяха добавени нови характеристики. Тази книга предлага обширно въведение в езика C++, съответстващо на по-късна негова реализация стандарт 2.0. Познаването на езика C не е задължително, но е необходимо владение на поне един съвременен структурен език. Книгата не е замислена като първа книга по програмиране.

Мощността на C++ се основава на новия подход за програмиране и новия начин на мислене. Овладяването на C++ не означава просто запознаване с новия набор от синтактични и семантични правила. За да се улесни изучаването на езика, книгата е организирана на групи от обширни примери, които поясняват конструкциите на езика и мотивират тяхното използване. Изучаването на езиковите конструкции в контекста на примерите изяснява ползата от тях, като едновременно с това създава усет кога и как да се използват те в реални приложения. Първите примери в книгата запознават с основните концепции на езика C++. По-късно се дават други примери, които разясняват пълно споменатите вече характеристики. По този начин базовите знания на читателя се доизграждат.

Структура на книгата

Фундаментална концепция в езика C++ е създаването на нови типове данни, които се използват с гъвкавостта и простотата на вградените типове данни. Първа стъпка в овладяването на езика е запознаването с неговите основи. Главите от 0 до 4 разглеждат основите на езика, а главите от 5 до 8 създаването на нови типове и обектно-ориентирания стил на програмиране.

Глава 0 описва основните елементи в една програма на C++, нейното въвеждане и изпълнение. Типовете данни, изразите и операторите са обект на разглеждане в глави 1 и 2. Глава 1 включва пример за създаване и използване на клас. По-нататък в книгата идеите от този пример се обогатяват и допълват. Глава 3 разглежда дефинирането на функции Глава 4 обсъжда възможностите за дефиниране на функции с еднакви имена и операции за управление на динамичната памет. Професионалният програмист и, особено, програмистът на C може да реши да прескочи тези глави. Не го правете. В тях са включени нови концепции и дори опитният програмист на C може да има полза от тях. Глави 5 и 6 разглеждат дефинирането на класове, проектирането и реализацията на абстрактни типове данни. Създаването на нови типове освобождава програмиста от редица досадни аспекти на програмирането. Основните за дадено приложение типове може да се реализират веднъж и да се използват многократно. Това позволява на програмиста да се концентрира върху проблема, който трябва да реши, а не върху детайлите на неговата реализация. В добавка

капсулирането на данните опростява съществено поддържането и изменението на готови приложения.

Обектно-ориентираното програмиране се разглежда в седма и осма глава. Глава 7 описва механизма за наследяване. Класически пример за наследяване са геометричните фигури. Всяка фигура се характеризира с положение в равнината, възможност за завъртане, изчертаване и т.н. Наследяването позволява общите атрибути да се поделят между геометричните фигури и реализацията на всяка геометрична фигура да дефинира само различните и допълнителни нейни характеристики. За обектно-ориентираното програмиране е характерно, че при изпълнение на програмата може да се избере функцията, която съответства на типа на аргумента. Например, ако искаме да завъртим фигура, трябва да сме сигурни, че ще се изпълни подходящата функция за завъртане в зависимост от вида на фигурата. В C++ това се осъществява чрез виртуални функции. Те се обсъждат в глава 8. Глава 8 разглежда и наследствена йерархия с виртуални основни класове.

Както в C, така и в C++ входно-изходните операции се осъществяват от стандартна библиотека. В стандарт 2.0 тази библиотека е значително разширена. Приложение А разглежда нейните възможности. Освен това в стандарт 2.0 са включени редица нови характеристики:

- множествово наследяване и виртуални основни класове.
- инициализиране и присвояване чрез копиране на членове.
- безопасно свързване с други езици.
- нови дефиниции на операциите new, delete, >>, и , (comma).
- константни и статични функции-членове на клас.

Приложение Г разглежда как тези нови възможности на езика въздействуват на програми, създадени с по-ранни версии на C++. В приложение Б са описани възможните бъдещи разширения на езика.

C++ притежава характеристики, които не са представени в C. Има други характеристики, които са общи за двата езика, но с малки разлики в реализацията. Те са разгледани в Приложение В.

Бележки по реализацията

Програмите в книгата са взети директно от изходните текстове, компилирани на DEC VAX 8550 като е използвана ОС UNIX V версия 2. Всички програми и програмни фрагменти са компилирани за стандарт 2.0 на езика C++, разработен от Bell Laboratories.

Част от книгата е използвана за обучение. Тя е представена от автора на професионална конференция. Чернова на ръкописа се използва като обучаващ и справочен материал за вътрешно ползуване в AT&T Bell Laboratories. Много от програмите и програмните фрагменти в книгата са използвани като тестове при разработката на стандарт 2.0.

Описанието на езика C++ в тази книга се основава на неговата разработка от май 1989г., направена от Stroustrup.

Благодарности

Отправлям най-сърдечна благодарност към Barbara Moo. Нейното насърчаване, съвети и внимателен прочит на последователните чернови на ръкописа бяха неоценими. Специални благодарности отправям към Bjarne Stroustrup за помощта му, както и за чудесния език, който ни е предоставил. Благодарност изразявам и на Stephen Dewhurst, който ми оказва помощ при първоначалното запознаване с езика.

Martin Carroll, William Hopkins, Brian Kernighan, Andrew Koenig, Alexis Layton, Phil Brown, James Coplein, Elizabeth Flanagan, David Jordan, Don Kretsch, Craig Rubin и Judy Warad прегледаха различни чернови на ръкописа и направиха полезни забележки. David Jordan изясни множество въпроси, свързани със стандарта ANSI C. Jerry Schwarz, който е написал пакета за вход/изход, предложи оригиналната документация, върху която е основано приложение А.

Благодаря на Brian Kernighan и Andrew Koenig за набора от средства за печат, които ми предоставиха.

ГЛАВА 0

При начално запознаване с езика C++ възникват два естествени въпроса:

1. Какво представлява програмата, написана на C++ и как се пише тя?
2. Как се създава готова за изпълнение програма?

Този глава предлага основни знания за езика C++ и създаването на изпълними програми.

0.1 Решаване на задачи

Много често програмите се създават в отговор на някакъв практически проблем или задача. Да разгледаме един пример. В една книжарница въвеждат във файл заглавието и името на издателя на всяка продадена книга. Информацията се въвежда по реда на продажба на книгите. На всеки две седмици собственикът изчислява ръчно броя на продадените книги от всяко заглавие и всеки издател. Списъкът се подрежда в азбучен ред по името на издателя и се използва за следващи заявки. Нашата задача е да създадем програма, която върши това. Сложните задачи се решават като се разделят на по-малки и по-лесни за решаване задачи. В случая можем да формулираме четири подзадачи:

1. Прочитане на файла с информация за продадените книги.
2. Преброяване на продажбите по заглавие и издател.
3. Сортиране на заглавията по име на издател.
4. Отпечатване на резултатите.

Първа, трета и четвърта точка представляват задачи, които знаем как да решим. Те не се нуждаят от по-нататъшно разглеждане. Към точка две ще приложим същия метод на разделяне на подзадачи:

- 2a. Сортиране на продажбите по издател.
- 2b. За всеки един издател сортиране на продажбите по заглавие.
- 2c. Откриване на еднаквите заглавия в издателските групи и за всяка такава двойка увеличаване броя на продажбите в първата група и премахване на заглавието от втората.

Точки 2a, 2b и 2c са задачи които знаем как да решим. Това означава, че вече можем да решим целия проблем. Нещо повече: оказва се, че първоначалният ред на подзадачите е неправилен. По-подходяща е следната последователност от действия:

1. Прочитане на файла с информация за продадените книги.
2. Сортиране на файла първо по издател, а след това за всеки издател, сортиране по заглавие.
3. Премахване на еднаквите заглавия за различните издателски групи.
4. Запис на резултатите в нов файл.

Получената последователност от действия се нарича алгоритъм. Следващата стъпка е да преведем получения алгоритъм на някакъв език за програмиране, в случая C++.

0.2 Програма на C++

В C++ действията се представят чрез изрази. Всеки израз, разделен с ";", се нарича оператор. Операторът е най-малката независима единица в програма на C++. В естествените езици аналог на оператора е изречението.. Да разгледаме няколко примера:

```
int value;  
value = 7 + 1;  
cout << value;
```

Първият оператор се нарича оператор за деклариране. Той дефинира област от паметта на компютъра, свързана с името `value`, която ще съдържа целочислени стойности. Вторият оператор се нарича оператор за присвояване. Той поставя в клетка, свързана с името `value`, резултата от събирането на 7 и 1. Третият оператор е оператор за изход. С `cout` се означава стандартното изходно устройство. Операцията "<<" е операция за изход. Третият оператор записва на стандартното изходно устройство стойността на `value`.

Група оператори, отделени в наименована част представляват функция. Например, всички оператори, които се използват при четене от файла с продажби, се организират във функция с име `readIn()`. Аналогично се създават функции с имена `sort()`, `compact()` и `print()`.

Всяка програма в C++ трябва да съдържа функция с име `main()`. Ето как може да изглежда функцията `main()` за горния пример:

```
int main()  
{  
    readIn();  
    sort();  
    compact();  
    print();  
    return 0;  
}
```

Изпълнението на програма, написана на C++, започва с първия оператор на `main()`. В случая изпълнението започва с обръщение към функцията `readIn()` и продължава с изпълнение на следващите оператори на `main()`. Програмата завършва нормално след изпълнението на последния оператор на `main()`.

Всяка функция се състои от четири части: тип на резултата, име на функцията, списък от аргументи и тяло на функцията. Първите три части взети заедно, се наричат прототип на функция (`function prototype`). Списъкът от аргументи се състои от нула или повече аргументи, отделени със запетая и заградени в скоби. Тялото на функцията се огражда във фигурни скоби "{}". То представлява поредица от оператори.

В дадения пример в тялото на `main()` се извикват и изпълняват последователно `readIn()`, `sort()`, `compact()` и `print()`. Накрая се изпълнява операторът `return 0`. Той осигурява завършване на функцията. Стойността след `return` се връща от функцията в точката на нейното извикване. В нашия случай се връща 0, което означава успешно завършване на `main()`.

Как да подготвим програмата за изпълнение? Първо трябва да дефинираме `readIn()`, `sort()`, `compact()` и `print()`. Засега са достатъчни следните дефиниции:

```
void readIn()    { cout << "readIn()\n";    }  
void sort()      { cout << "sort()\n";      }  
void compact()   { cout << "compact()\n";   }
```

```
void print()      { cout << "print()\n";      }
```

void е служебна дума и означава, че функцията не връща стойност. Така дефинирани функциите ще изпишат името си на екрана. По-нататък тези дефиниции може да се заменят с действителни. Създаване на програми чрез изграждане на техния скелет, последвано от запълване със съдържание на всяка функция, осигурява контрол на грешките, които неизбежно правим.

Текстът на програмата се записва във файл. Неговото име се състои от две части: име и разширение. Разширението подсказва предназначението на файла. То е различно за отделните реализации на C++. Например, в UNIX за разширение на файл с програма на C++ служи ".c" или ".C". Ако искате да наблегнете на факта, че даден файл съдържа програма на C++, използвайте ".c". В MS-DOS се използват следните разширения: sxx (x е "+", завъртян на 45 градуса) или cpp (p от "plus").

Въведете следващата програма в текстов файл с име prog1.c.

```
#include <stream.h>

void readln()      { cout << "readln()\n"; }
void sort()        { cout << "sort()\n";   }
void compact()     { cout << "compact()\n"; }
void print()       { cout << "print()\n";  }

int main()
{
    readln();
    sort();
    compact();
    print();
    return 0;
}
```

stream.h е заглавен файл (header file). Той съдържа необходимата информация за cout. #include е директива на предпроцесора. Тя предизвиква включване на stream.h в текстовия файл. Раздел 0.5 разглежда подробно директивите на предпроцесора.

След създаване на текстов файл с програма, тя трябва да се компилира. Затова се стартира компилаторът на езика:

```
$CC prog1.C
```

В AT&T C++ неговото име е CC. В други системи името може да е различно.

Част от работата на компилатора се състои в проверка "коректността" на програмата. Компилаторът не може да вникне в смисъла на програмата и "да разбере" дали тя върши това, за което е предназначена. Той открива два вида грешки, свързани с нейната "външна" (видима) страна.

1. Синтактични грешки (Syntax errors). Това са "граматическите грешки" на програмиста при програмиране. Например:

```
int main ( {          //грешка: пропусната е ')'
    readln():         //грешка: неправилен символ ':'
    sort();
```



```
compact();
print();
return 0      //грешка: пропуснат символ ';'
}
```

2. Грешки, свързани с типа на данните (Type errors). Данните в C++ имат определен тип. Например 10 се възприема като цяло число. Думата "hello", заградена в кавички, представлява низ, т.е. тя е от тип string. Ако функция с аргумент цяло число се извика с низа "hello", компилаторът ще открие грешка, свързана с типа на данните.

Съобщението за грешка съдържа номера на реда, където е открита грешка и кратко описание на предполагаемата причина за нея. Добре е грешките да се коригират в последователността, в която са обявени, защото една грешка може да доведе до т.нар. "каскаден ефект", при който компилаторът открива повече грешки, отколкото реално съществуват. Коригираният текст на програмата трябва да се компилира отново.

Втората част от работата на компилатора е транслиране на формално правилния текст на програмата. Транслация се нарича генерирането на код (code generation), т.е. създаване на разбираем за компютъра обектен код.

След успешна компилация се генерира изпълним файл. По подразбиране се създава изпълним файл с име a.out. Той се стартира така:

```
$a.out
```

При изпълнението му се извежда следния текст:

```
readln()
sort()
compact()
print()
```

Чрез опцията -o<name> програмистът може да посочи име на изпълним файл, различно от a.out. Например,

```
$CC -oprog1 prog1.C
```

създава изпълним файл с име prog1. Той се стартира така:

```
$prog1
```

В допълнение различните версии на C++ притежават множество стандартни библиотеки, които представляват сбор от компилирани функции. Например, функциите за вход и изход се намират в стандартна библиотека. Програмистът може да използва библиотечните функции в програмите, както използва собствените си функции. В следващия раздел се разглежда библиотеката за вход/изход в C++.

0.3 Запознаване с организацията на вход-изхода

Входно-изходните операции не са част от езика C++. Те се поддържат от стандартна библиотека, известна като `iostream library`. Приложение А разглежда тази библиотека. Читателите, които използват предишни версии на езика си служат с библиотеката `stream`. Примерите в тази книга се основават на общо подмножество от функции за двете версии. Ако работите под управление на MS-DOS, може да проверите в документацията за името на библиотеката.

Входът от потребителския терминал се нарича стандартен вход. Той се означава с `cin`. Насоченият към потребителския терминал изход се нарича стандартен изход. Той се означава с `cout`.

Операцията за изход "<<" извежда стойност на стандартния изход (екрана). Например:

```
cout << "The sum of 7 + 3 = ";  
cout << 7 + 3;  
cout << "\n";
```

Последователността от символи `\n` е символ за нов ред. Той предизвиква преминаване на нов ред. Няколко последователни оператори за изход могат да се обединят в един:

```
cout << "The sum of 7 + 3 = " << 7 + 3 << "\n";
```

За по-добра читаемост конкатенираните оператори за изход могат да се разположат на няколко реда. Например:

```
cout << "The sum of"  
    << v1  
    << " + "  
    << v2  
    << " = "  
    << v1 + v2  
    << "\n";
```

Аналогично, операцията за вход ">>" прочита стойност от стандартното входно устройство. Следващата програма прочита две числа, определя по-голямото от тях и го отпечатва.

```
#include<stream.h>  
  
void read2( int&, int& );  
int max( int, int );  
void writeMax( int );  
  
main() {  
    int val1, val2;  
    read2( val1, val2 );  
    int maxVal = max( val1, val2 );  
    writeMax( maxVal );  
    return 0;  
}  
  
void read2( int& v1, int& v2);
```

```

{
    cout << "\nВъведете две числа: ";
    cin >> v1 >> v2;
}

int max( int v1, int v2);
{
    if ( v1 > v2 )
        return v1;
    else
        return v2;
}

void writeMax( int val )
{
    cout << val << " е по-голямото число.\n";
}

```

Трябва да направим някои бележки. Първо, имената на трите функции са записани преди дефиницията на `main()`. Това са техните декларации. Те показват, че функциите са дефинирани в отделен файл или по-нататък в същия файл.

Преди да се извика в програмата, всяка функция трябва да се декларира. `val1` и `val2` се наричат променливи. Те се дефинират чрез оператора

```
int val1, val2;
```

За променливите важи същото правило: те трябва да са известни за програмата преди да се използват. На тях е посветена глава 1.

v1 и **v2** се наричат формални параметри. Те образуват списъка от аргументи на `read2()` и `max()`. `val` е единственият формален параметър на `writeMax()`. Смесът на формалните параметри се обсъжда в раздел 3.6.

Може би сте забелязали, че този път `main()` е обявена по-различно — не е определен тип на резултата. В C++ това е разрешено. Функция, без тип на резултата, връща стойност от тип `int`. След компилиране и изпълнение на програмата получаваме следния резултат:

```

Въведете две числа: 17 124
124 е по-голямото число.

```

Въведените от потребителя числа са разделени с интервал. Новите редове, интервалите и табулациите в C++ се наричат `white space` (празни позиции). Операторът

```
cin >> v1 >> v2;
```

прочита правилно двете стойности, защото операцията за вход `">>"` премахва всички празни позиции, които среща.

Предварителната декларация (forward declaration) на `cin` и `cout` се намира в `stream.h`. Ако програмистът не е включил този файл, при всяко обръщение към `cin` или `cout` компилаторът открива грешка, свързана с типа (type error). Предварителните декларации се съхраняват в т.нар. заглавни файлове (header files). Те са разгледани подробно в раздел 4.5.

С **cerr** се означава стандартното устройство за грешки. **cerr** се използва за предупреждаване на потребителя в изключителни ситуации при изпълнение на програмата. Например, следващата част от програма предотвратява делението на нула.

```
if ( v2 == 0 ) {  
    cerr << "\nгрешка: опит за деление на нула";  
    return;  
}  
v3 = v1/v2;
```

Ще дадем пример за програма, която чете от стандартния вход символ по символ до срещане на end-of-file (маркер за край на файл) и брой символите и редовете. Техният брой се отпечатва във вида:

брой_редове брой_символи

Ето и самата програма:

```
#include <stream.h>  
  
main () {  
    char ch;  
    int lineCnt=0, charCnt=0;  
    while ( cin.get(ch) ) {  
        switch ( ch ) {  
            case '\t' :  
            case ' ' : break;  
            case '\n' : ++lineCnt; break;  
            default  : ++charCnt; break;  
        }  
    }  
    cout << lineCnt << " " << charCnt << "\n";  
    return 0;  
}
```

`get()` е функция за вход. Тя прочита символ и го присвоява на своя аргумент — в случая `ch`. `\t` представлява символа `tab`.

Оператор `switch` сравнява стойността на една променлива със стойностите след етикетите `case`. При съвпадение се изпълняват операторите след съответния `case`. В противен случай се изпълняват операторите след `default`. Всеки прочетен символ за нов ред увеличава с 1 стойността на `lineCnt`. `charCnt` се увеличава с 1, когато прочетеният символ не е интервал, символ за нов ред или табулация. Оператор `while` е оператор за цикъл. Той изпълнява множество оператори, докато условието в цикъла е удовлетворено. В примера `switch` се изпълнява, докато `get()` прочете символ за край на файл. `switch` и `while` се обсъждат във втора глава.

Упражнение 0-1. Запишете програмата във файл `prog2.<разширение>` и го компилирайте. Нека изпълнимият файл има име `prog2`.

Упражнение 0-2. Изпълнете `prog2` над текстовия файл с програмата от предишния пример. Изпълнете я над текстов файл, съдържащ ваша собствена програма. Изпълнете я над файл, съдържащ само интервали, табулации и нови редове. Изпълнете я над файл, съдържащ само маркер за край на файл.

Упражнение 0-3. Променете програмата, така че тя да брои отделно броя на табулациите tabCnt и броя на интервалите blankCnt. На екрана да се отпечата следното:

Общ брой символи: xx
Редове: x
Символи: x
Табулации: x
Интервали: x

0.4 Коментари

Коментарите са част от добрия стил на програмиране. Те служат за лесно разчитане на програмите. В тях може да се включи описание на алгоритъма и променливите. Може да се обясни част от текста на програмата. Те не увеличават размера на изпълнимата програма, защото преди генерирането на кода компилаторът ги премахва. В C++ се използват два ограничителя за коментар. Символите "/*" и "*/" се използват както в езика C. Всичко между тях се счита за коментар. Допуска се използването на интервали, табулации и нови редове. Това означава, че коментарите могат да заемат повече от един ред. Например:

```
/*
 * Това е запознаване с класовете в C++.
 * Класовете се използват в обектно-ориентираното
 * програмиране. В глава 5 е разгледан клас Screen
 */

class Screen { /* тяло на класа */
public:
    void home(); /* Придвижва курсора до точката (0,0) */
    void refresh(); /* Прерисува екрана */
private:
    /* Класовете позволяват да се скрива информация. */
    /* По този начин се ограничава достъпа на програмата */
    /* до вътрешното представяне на класа, т.е. до неговите данни. */
    /* Скриването на информация се осъществява чрез използване на етикета "private:". */
    char *cursor; /* Текуща позиция на екрана */
};
```

Смесването на коментари и програмен текст затруднява четенето на програмата. Затова се препоръчва блокът с коментари да се разполага над текста, който обяснява. Коментарите не бива да се враждат един в друг. Следващата програма ще предизвика много грешки при компилация. Как бихте ги отстранили?

```
#include <stream.h>
/*
 * Двойката ограничители на коментари /* */ не бива да се
 * вражда. "не бива да се вражда." се смята за текст на
 * програмата, както и следващите два реда.
 */

main() {
    cout << "hello, world\n";
}
```

Втората двойка ограничители на коментар е `"/"`. За коментар се счита всичко, което се намира на текущия ред, надясно от `"/"`. Например:

```
#include <stream.h>
#include "myIO.h"

int isOdd( int );

main() {
    int v1, v2;           // Съдържат необходими на потребителя стойности
    read2( v1, v2 ); // Декларирана в myIO.h
    if ( isOdd( v1 ) == 1 )
        cout << v1 << "е нечетно\n";
    if ( isOdd( v2 ) == 1 )
        cout << v2 << "е нечетно\n";
    return 0;
}

isOdd( int val )
{ /* Връща 1 ако val е нечетно, иначе връща 0
   * % е операция за остатък от деление: 3 % 2 = 1
   */
    return(val % 2 != 0);
}
```

Програмата определя дали две произволно въведени числа са четни или нечетни. Тя използва функцията `read2()`, дефинирана в предишния раздел. Прототипът на `read2()` се намира в заглавния файл `myIO.h`. След компилиране и изпълнение на програмата се получава:

```
Въведете две числа: 497 -25
497 е нечетно
-25 е нечетно
```

Символите за коментар не бива да се разделят с интервал, табулация или нов ред. Следващите два реда не са коментар, а програмен текст.

```
/ * Не е коментар: празни позиции не са разрешени */
/ / Също не е коментар: трябва да се запише //
```

Обикновено в програмите се използват двата начина за коментар. За по-дълги обяснения е удобно да се използва двойката `"/"` и `"/"`. Кратките бележки се маркират с `"/"`.

0.5 Директиви на предпроцесора

Наред със стандартните библиотеки съществува множество от стандартни заглавни файлове, които съдържат необходимата на потребителя информация за използване на библиотеките. За да имаме достъп до функциите от стандартните библиотеки, трябва да включим съответния заглавен файл в своята програма.

Заглавните файлове стават част от програмата чрез директивата `include`. Директивите се означават със знака `"#"`, който се записва в първата колона на реда. Директивите се обработват преди обръщение към компилатора. Програмата, която ги обработва се нарича предпроцесор.

Директивата `#include` прочита съдържанието на указания в нея файл. Тя има една от следните две форми:

```
#include <stream.h>
#include "myIO.h"
```

Когато името на файла е заградено в ъглови скоби "<>", заглавният файл е стандартен. Той се търси в директория със специално име. Можем да посочим друга директория с опцията **-I**. Например:

```
$CC -lincl -I/usr/local/include prog1.C
```

Сега предпроцесорът ще търси първо в директорията `incl`, после в `/usr/local/include` и едва тогава, в директорията със специално име. При откриване на файл с указаното име търсенето завършва.

Когато името на файла е заградено в кавички се смята, че заглавният файл е създаден от потребителя. Търсенето на такъв файл започва от текущата директория. Ако там го няма, търсенето ще продължи в директорията със специално име. Опцията **-I** работи по същия начин и със заглавни файлове, създадени от потребителя. Указаният в директива `#include` файл може да съдържа също директива `#include`. Затова в една програма даден заглавен файл може да се включи многократно. За да избегнем този ефект, трябва да използваме условни директиви. Например:

```
#ifndef STRING_H
#define STRING_H
    /* Включване на String.h */
#endif
```

Условната директива `#ifndef` получава стойност истина, ако името след нея не е дефинирано. Тогава ще се включат всички редове между `#ifndef` и `#endif`. В противен случай редовете до `#endif` ще се игнорират. `#define` дефинира името, което я следва. В нашия случай се дефинира `STRING_H`. Ако заглавният файл `string.h` е включен вече, директивата `#ifndef` ще приеме стойност лъжа и съдържанието на `string.h` няма да се включи отново. Директивата `#ifdef` получава стойност истина, ако името след нея е дефинирано. Например:

```
#ifdef u3b2
    /* Системен код за компютри AT&T 3B */
#endif

#ifdef sun
    /* Системен код за компютри Sun */
#endif
```

C++ дефинира предварително името `__cplusplus`. Потребител, който иска да смеси програми на C++ и C може да напише следното:

```
#ifdef __cplusplus
    extern min( int, int );
    int *pi = new int;
```

```
#else
    extern min();
    int *pi;
#endif
```

Операторите между `#else` и `#endif` ще бъдат включени, ако `#ifdef` или `#ifndef` получат стойност лъжа.

Предпроцесорът е тясно свързан с езика C (често го наричат `cpp` или C preprocessor). Много версии използват точно `cpp` и следователно не признават `/*` като символ за коментар. Ако искате да включите коментар, използвайте двойката символи `/*` и `*/`. Например:

```
#ifdef u3b2
#define SYSV      /* Система UNIX V */
#endif
```


ГЛАВА 1

В C++ е дефинирано множество от стандартни типове данни, операции с тях и оператори за управление на изчислителния процес. Тези елементи формират азбуката на езика, чрез която могат да се създават големи, сложни и реални системи. На това базово ниво C++ е прост език. Неговата изразителна сила нараства от възможността да се дефинират нови типове данни.

Първа стъпка в овладяването на C++ е запознаването с основите на езика. Това е темата на тази и следващата глава. Тук ще разгледаме как се дефинират типовете данни и механизмите за конструиране на нови типове. Глава 2 разглежда операциите за тях. Текстът на програмата и нейните данни се съхраняват като последователност от битове в паметта на компютъра. Бит е проста клетка със стойност 0 или 1. Във физически термини тази стойност означава отсъствие или наличие на електрически заряд. Обикновено сегмент от паметта на компютъра изглежда така:

...00011011011100010110010000111011...

Тази редица от битове няма структура, т.е. не може да се говори за нейния смисъл, защото такъв няма.

За да се структурира тя, се въвеждат понятията байт и дума. Един байт се състои от 8 бита. Обикновено една дума се състои от 16 или 32 бита. Байтът и думата имат различен размер при различните компютри, т.е. размерът им е машинно зависим. Фиг. 1.1 показва редица от битове, организирана в 4 адресируеми байта.

1024	0	0	0	1	1	0	1	1
1032	0	1	1	1	0	0	0	1
1040	0	1	1	0	0	1	0	0
1048	0	0	1	1	1	0	1	1

Фигура 1.1. Адресируема машинна памет

Организацията на паметта позволява да се обръщаме към адресируема съвкупност от битове. Може да се говори за думата на адрес 1024 или за байта на адрес 1040. Има смисъл да се каже, че байтът на адрес 1032 не е равен на байта на адрес 1048. Но не е възможно да се говори смислено за съдържанието на тези байтове. Защо? Защото не знаем как да интерпретираме последователността от битове. За да говорим за значението на един байт, трябва да знаем типа на представената стойност.

Абстракцията "тип данни" позволява смислена интерпретация на последователност от битове с фиксирана дължина, разположена на определен адрес. Символите, целите числа и числата с плаваща точка са примери за типове данни. Друг тип представляват адреси от паметта и се наричат указатели.

В C++ има множество стандартни типове. Такива са целите числа, числата с плаваща точка и символите.

- Тип `char` се използва за представяне на отделни символи или малки цели числа. `char` се представя в паметта с един байт.

- Тип `int` е използва за представяне на цели числа. В паметта `int` се представя с една дума.

C++ поддържа и типовете `short` и `long`. Те се използват съответно за малки и големи цели числа. Действителният размер на тези типове е машинно зависим. Взеи заедно `char`, `short`, `int` и `long` се наричат целочислени типове. Целочислените данни може да са със или без знак. Разликата е в интерпретацията на най-левия бит в представянето на числото. В типа със знак този бит е знаков, а останалите представят стойността. В типа без знак всички битове представят стойността на числото. Ако знаковият бит има стойност 1, числото се счита за отрицателно, иначе е положително. Тип `signed char` представя числата от -128 до 127, а тип `unsigned char` от 0 до 255.

Тип `float` и тип `double` представят числа с плаваща точка, съответно с единична и двойна точност. Обикновено тип `float` се представя с една дума, а тип `double` с две. Действителните размери са машинно зависими.

Изборът на тип данни зависи от интервала, в който се очаква да попаднат стойностите. Например, ако стойностите не надхвърлят 255 и не са по-малки от 0, тип `unsigned char` е подходящ. В противен случай е добре да се използва `int` или `long`.

1.1 Константи

Всяко число (например 1) в програмата се нарича литерална константа. Литерална, защото можем да говорим за него като цитираме стойността му и константа, защото тази стойност не може да се променя.

Всяка литерална константа е свързана с някакъв тип. Например, 1 е от тип `int`, 3.14159 е литерална константа от тип `double`. Литералните константи са неадресируеми за нас. Макар че техните стойности се съхраняват в паметта, ние нямаме достъп до адресите им.

Целите литерални константи могат да се записват в десетична, осмична или шестнайсетична система. Това не променя тяхното представяне в паметта. Например, числото 20 може да се запише по един от следните три начина:

```
20      // десетично число
024     // осмично число
0x14    // шестнайсетично число
```

Символът "0" (нула) пред цяла литерална константа показва, че числото е осмично. Аналогично символите "0x" или "0X" се използват за означаване на шестнайсетично число. В приложение А се обсъжда отпечатването на осмични и шестнайсетични числа.

Литералната константа е от тип `long`, ако след нея стои символът "L" или "l". Обикновено употребата на "l" се избягва, защото се грешки с 1. По подобен начин символите "U" и "u" означават, че константата е без знак. При комбинация от тези символи се обявява тип `unsigned long`. Например:

```
128u      1024UL      1L      8Lu
```

Литерална константа от тип `float` може да се запише със или без експонента. Експонентата се означава с "E" или "e". Константата е от тип `float`, ако след нея стои символът "F" или "f". Например:

```
3.15159F  0.1f      0.0
```

3e1 1.0E-3 2.

Печатимите литерални константи от тип `char` се ограждат в апострофи. Например:

`'a' '2' ' ' ' (интервал)`

Непечатимите символи, апострофа, кавичките и символа `"\"` (backslash) могат да се представят така:

нов ред	<code>\n</code>
хоризонтална табулация	<code>\t</code>
вертикална табулация	<code>\v</code>
<code>backspace</code>	<code>\b</code>
върни каретката	<code>\r</code>
нова страница	<code>\f</code>
звънец	<code>\a</code>
<code>backslash</code>	<code>\\</code>
символ <code>"?"</code>	<code>\?</code>
апостроф	<code>\'</code>
кавички	<code>\"</code>

Може да се използва и означение `\ooo`, където `ooo` представлява последователност от три осмични цифри. Тяхната стойност е ASCII кода на съответния символ. Например:

<code>\007</code> (звънец)	<code>\010</code> (нов ред)
<code>\062</code> ('2')	<code>\0</code> (край на низ)

Литералните константи от тип `string` се състоят от нула или повече символи, заградени в кавички. Ако има непечатими символи, те се представят както по-горе. Константа от тип `string` може да обхваща повече от един ред. Когато символът `"\"` е последен в реда, низът продължава и на следващия ред. Например:

```
"" ( празен низ )
"a"
"\nCC\toptions\file:[cC]\n"
"Това е низ, разположен на три реда. \
Символът backslash в края на реда означава, \
че низът продължава и на следващия ред."
```

Тип `string` представлява масив от символи. Той включва самия низ и символ за край (`null`), добавен от компилатора. Например, докато `'a'` представлява само простия символ `a`, то `"a"` е символът `a`, следван от символа `null`.

1.2 Променливи

Нека да съставим програма за пресмятане на 2^{10} . Ето един начален вариант:

```
#include <stream.h>

main() {
```

```

// първо решение
cout << " 2 на степен 10 е ";
cout << 2 * 2 * 2 * 2 * 2 * 2 * 2 * 2 * 2 * 2;
cout << "\n";
return 0;
}

```

Програмата изчислява правилно 1024, но ако искаме да изчислим 2^{17} или 2^{23} , трябва да я променим. Ако след това искаме да изведем всички степени на 2 от 2^0 до 2^{31} и използваме константи от тип string, ще са необходими 64 реда като тези:

```

cout << "2 на степен X\t";
cout << 2 * ... * 2;

```

където X ще се увеличава с 1 във всяка такава двойка редове. Естествено съществува по-добър начин за реализация. Той използва две възможности, които още не сме представили.

1. Използване на променливи, позволяващи запазване и възстановяване на стойности.
2. Оператор for, позволяващ многократно изпълнение на част от програмата.

Ето как изглежда един втори вариант:

```

#include <stream.h>

main() {
    //второ по-общо решение
    int value = 2;
    int pow = 10;
    cout << value << " повдигнато на степен " << pow << ": \t";
    for ( int i = 1, res = 1; i <= pow; ++i ) {
        res = res * value;
    }
    cout << res << "\n";
    return 0;
}

```

for се нарича оператор за цикъл: докато i е по-малко или равно на pow се изпълнява тялото на цикъла, т.е. заградените в скоби оператори. value, pow, res и i са променливи. Те позволяват запазване, изменение и възстановяване на стойности. Променливите се азглеждат в следващите подраздели. Нека сега отделим във функция тази част от програмата, която изчислява експоненциалната стойност.

```

unsigned int pow( int val, int exp )
{
    // изчислява val на степен exp
    for ( unsigned int res = 1; exp > 0; --exp )
        res = res * val;
    return res;
}

```

Сега всяка програма, която изисква изчисляване на експоненциална стойност трябва да извика само pow() със съответните аргументи. Например, таблица със степените на 2 може да се генерира така:

```
#include <stream.h>

extern unsigned int pow( int, int );

main() {
    int val = 2;
    int exp = 16;
    cout << "\nСтепени на 2\n";
    for ( int i = 0; i <= exp; ++i )
        cout << i << ":" << pow( val, i ) << "\n";
    return 0;
}
```

Резултатът от изпълнението на последната програма е даден в табл. 1.1.

Степени на 2	
0:1	
1:2	
2:4	
3:8	
4:16	
5:32	
6:64	
7:128	
8:256	
9:512	
10:1024	
11:2048	
12:4096	
13:8192	
14:16384	
15:32768	
16:65536	

Табл. 1.1

Реализацията на pow() не разглежда редица специални случаи — изчисляване на отрицателна експонента или получаване като резултат на много голямо число.

Упражнение 1-1. Какво ще се получи, ако pow() се изпълни с отрицателен втори аргумент? Как би могла да се промени pow() за да обработва и този случай?

Упражнение 1-2. Всеки тип данни има горна и долна граница, която зависи от броя на битовете в представянето на типа. Как би трябвало да се отрази това на реализацията на pow()? Модифицирайте pow(), така че pow(2,48) да връща верен резултат.

Какво представляват променливите?

Всяка променлива се идентифицира с дадено и от потребителя име. Тя задължително е от някакъв тип. Например

```
char ch;
```

декларира променлива с име `ch` от тип `char`. `char` се нарича спецификатор на тип (`type specifier`). `short`, `int`, `long`, `float` и `double` също са спецификатори на тип. Всяка декларация на променлива започва със спецификатор на тип. Той определя броя байтове, в които ще се съхранява тя и множеството от операции, в които може да участва. Както променливите, така и константите имат тип и се съхраняват в паметта. Разликата е в това, че променливите са адресируеми, т.е. с една променлива се свързват две стойности:

1. Нейната собствена стойност, запазена в някаква позиция на паметта. Тя се нарича `rvalue`.
2. Стойността на адреса от паметта, в който се записва `rvalue`. Тази стойност се нарича `lvalue`.

Да разгледаме израза:

```
ch = ch - '0';
```

В него променливата `ch` се среща както в лявата, така и в дясната страна на оператора за присвояване. Тази, която стои отляво, се прочита, т.е. нейната собствена стойност се извлича от съответната позиция в паметта. След това символната константа `'0'` се изважда от прочетената вече стойност. Полученият резултат се присвоява на `rvalue` на израза. Стойността `rvalue` може да бъде прочетена, но не може да бъде променена. Това е стойност само за четене. Резултатът се записва в променливата `ch`, отляво на оператора за присвояване. При това предишната и стойност се губи. `lvalue` на променливата в лявата част на израза представлява `lvalue` на израза.

Дефинирането на променлива предизвиква заделянето на място в паметта. Дефиницията определя типа и името на тази променлива. По избор тя може да получи и начална стойност. В програмата трябва да съществува точно една дефиниция на дадена променлива.

Декларирането на променлива показва, че в програмата променливата е дефинирана. Декларацията включва името на променливата и нейния тип, предшествани от ключовата дума `extern`. (За подробности виж раздел 3.9.) Декларацията не заделя място за променливата. С нея само се потвърждава, че някъде в програмата съществува дефиниция на тази променлива. Една променлива може да се декларира няколко пъти в една програма.

В C++ всяка променлива трябва да се бъде дефинирана или декларирана преди да се използва.

Име на променлива

Името на всяка променлива се нарича **идентификатор**. Идентификаторът е последователност от букви, цифри или знак за подчертаване (`underscore`). Той задължително започва с буква или знак за подчертаване. Големите и малките букви се различават. В езика не са предвидени ограничения за дължината на идентификаторите, но такива могат да съществуват в някои конкретни реализации.

В C++ има множество запазени думи, наречени ключови. Например, спецификаторите за тип на данните са такива запазени думи. Те не могат да се използват извън стандартното предназначение. Табл. 1.2 съдържа запазените думи в C++. Съществуват общоприети правила за именуване на променливите, свързани с по-добра читаемост на програмите:

asm	delete	if	register	template ¹
auto	do	inline	return	try
break	double	int	short	typedef
case	else	long	signed	union
catch	enum	new	sizeof	unsigned
char	extern	operator	static	virtual
class	float	overload	struct	void
const	for	private	switch	volatile
continue	friend	protected	this	while
default	goto	public		

Табл. 1.2. Ключови думи в C++

Обикновено идентификаторите се записват с малки букви. Името трябва да е свързано с предназначението — т.нар. мнемонично име. Ако името включва няколко отделни думи, тогава се използва знак за подчертаване между тях или главна първа буква за всяка отделна дума.

Дефиниране на променливи

Простата дефиниция на променлива включва спецификатор на тип и име. Тя завършва с точка и запетая ";". Например:

```
double salary;
double wage;
int month;
int day;
int year;
unsigned long distance;
```

Когато трябва да се дефинират повече от една променливи от един и същи тип, след спецификатора на тип, може да се използва списък на променливите. Този списък може да обхваща няколко реда. Дефиницията завършва с ";". Например, горните дефиниции могат да се запишат и така:

```
double salary, wage;
int month, day, year;
unsigned long distance;
```

Простата дефиниция определя типа и името на променливата. Тя не осигурява начална стойност. Променлива без начална стойност се нарича неинициализирана. Такава променлива притежава стойност, защото в паметта, заделена за нея има нещо, което е останало от предишното използване на тази памет. Нещо повече: тази стойност може да се променя при всяко ново изпълнение на програмата. Следващият пример илюстрира произволното поведение на неинициализирана променлива:

¹ **template** е запазена за бъдещи разширения на езика за т.нар. параметризирани типове

```

#include <stream.h>
const iterations = 2;

void func() {
    // показва опасностите при използване на неинициализирана променлива
    int value1, value2; // неинициализирана променлива
    static int depth = 0;
    if ( depth < iterations ) {
        ++depth;
        func();
    }
    else depth = 0;
    cout << "\nvalue1:\t" << value1;
    cout << "\nvalue2:\t" << value2;
    cout << "\tsum:\t" << value1 + value2;
}

main() {
    for ( int i = 0; i < iterations; ++i )
        func();
}

```

След компилиране и изпълнение на програмата се получава изненадващ резултат. Нещо повече: резултатът е различен при всяко ново компилиране и изпълнение на програмата:

value1: 0	value2: 74924	sum: 74924
value1: 0	value2: 68748	sum: 68748
value1: 0	value2: 68756	sum: 68756
value1: 148620	value2: 2350	sum: 150970
value1: 2147479844	value2: 671088640	sum: -1476398812
value1: 0	value2: 68756	sum: 68756

В тази програма `iterations` е обявена като константа. Затова служи ключовата дума `const`. Обявяването на константи се обсъжда в раздел 1.5. `depth` е дефинирана като локална статична променлива (local static variable). Значението на ключовата дума `static` се обяснява в раздел 3.10. `func()` е рекурсивна функция. Раздел 3.1 е посветен на рекурсивните функции.

Начална стойност на една променлива може да се зададе при дефинирането и. Тогава променливата се нарича инициализирана. Например:

```

#include <math.h>

double price = 109.99, discount = 0.16,
salePrice = price * discount;
int val = getValue();
unsigned absVal = abs( val );

```

`abs()` е стандартна функция и се намира в библиотеката `math`. Тя връща абсолютната стойност на своя аргумент. `getValue()` е функция, дефинирана от потребителя. Тя връща случайно цяло число. Една променлива може да се инициализира с произволно сложен израз.

1.3. Указатели

Променлива, която е указател, съдържа адреса на някакъв обект. Чрез указателя можем да се обърнем директно към обекта. Типичен пример за използване на указатели е създаването на свързани списъци и управлението на динамични обекти по време на изпълнение на програмата.

Всеки указател е свързан с някакъв тип. Този тип определя типа на обектите, които указателят адресира. Например, указател от тип `int` адресира обекти от тип `int`. За да адресираме обект от тип `double`, трябва да дефинираме указател от тип `double`.

Мястото в паметта, отделено за един указател е с размер, достатъчен да събере адрес от паметта. Това означава, че указател от тип `int` и указател от тип `double` обикновено имат еднакъв размер. Типът на указателя определя размера на данните, които се интерпретират, и начина на интерпретацията им. Следват няколко дефиниции на указатели:

```
int *ip1, *ip2;
unsigned char *ucp;
double *dp;
```

В дефиницията на указател идентификаторът се предшества от символа `"*"`. Да разгледаме дефиницията:

```
long *lp, lp2;
```

Тук `lp` е указател към обект от тип `long`, докато `lp2` е обект от тип `long` и не е указател. В следващия пример `fp` се дефинира като обект от тип `float`, а `fp2` като указател от същия тип.

```
float fp, *fp2;
```

За яснота е по-добре да се записва

```
char *cp;
```

отколкото

```
char* cp;
```

Често се допуска грешка, ако по-късно се наложи да се дефинира втори указател от тип `char`. Програмистът може неправилно да модифицира горната дефиниция така

```
char* cp, cp2;
```

Указател може да се инициализира с адреса на обект от същия тип. За да получим стойността на обекта, трябва да приложим специалната операция `"&"` (амперсанд). Например:

```
int i = 1024;
```

```
int *ip = &i; // на ip се присвоява адреса на i
```

Указател може да се инициализира с друг указател от същия тип. В този случай употребата на операцията "&" е неправилна:

```
int *ip2 = ip;
```

Инициализирането на указател със стойността на някакъв обект е грешка. Затова компилаторът ще обяви следващите дефиниции за неправилни:

```
int i = 1024;  
int *ip = i; // грешка
```

Грешка е също да се инициализира указател с адреса на обект от друг тип. Следващите дефиниции на `uip` и `uip2` са неправилни:

```
int i = 1024; int *ip = &i; // ok  
unsigned int *uip = &i, // грешка  
             *uip2 = ip; // грешка
```

В C++ всяко присвояване и инициализация се проверяват от компилатора за съответствие на типовете. Ако има несъответствие, но съществува правило за преобразуване на типовете, компилаторът ще го приложи. Раздел 2.10 съдържа правилата **за преобразуване на типове**. Ако не съществува правило, компилаторът ще сигнализира за грешка. Ще се появи предупреждение, защото има вероятност това присвояване или инициализация да предизвика грешка при изпълнение на програмата.

Очевидно е, че е опасно на указател да се присвои стойността на обект. Указателят ще приеме тази стойност за адрес от паметта. Всеки опит за четене или запис на този мним адрес може да предизвика големи поражения.

Когато указател получи адреса на обект от друг тип, последиците от това присвояване са по-неуловими. Те са свързани с начина на интерпретиране на данните на съответния адрес. Например, указател от тип `int` и указател от тип `double` могат да съдържат един и същ адрес от паметта, в която ще се чете или записва чрез двата указателя, но тя ще е с различен размер. Освен това организацията и значението на поредицата битове може да се различава за двата типа данни.

Не е забранено да се заменя указател от един тип с указател от друг тип. Тъй като това е потенциално опасно, трябва да се прави много внимателно. Указател от произволен тип може да получи стойност 0, което означава, че той не адресира нищо. Тази стойност на указател се означава още като `NULL`. Съществува и специален тип указател `void*`, който може да адресира обект от произволен тип. Раздел 2.10 разглежда такъв тип указатели.

За да получим достъп до обект, адресиран от указател, трябва да приложим операцията `"*"`. Например:

```
int i = 1024;  
int *ip = &i; // сега ip сочи i  
int k = *ip; // k съдържа 1024
```

Без тази операция `k` се инициализира с адреса на `i`, а не с неговата стойност. Това ще предизвика грешка при компиляция:

```
int *ip = &i; // сега ip сочи i
int k = ip;   // грешка
```

За да присвоим стойност на обект, адресиран от указател, трябва да приложим операцията `**` към указателя. Например:

```
int *ip = &i; // ip сочи i

*ip = k;      // i = k
*ip = abs(*ip); // i = abs(i)
*ip = *ip + 1; // i = i + 1
```

Следващите два оператора дават различни резултати, макар че и двата са допустими. Първият увеличава адреса, който се съдържа в указателя `ip`. Вторият увеличава стойността на обекта, адресиран от `ip`.

```
int i, j, k;
int *ip = &i;
ip = ip + 2; // увеличаване на адреса, съдържащ се в ip
*ip = *ip + 2; // i = i + 2
```

Адресът, който се съдържа в указател, може да се извади или събере с някаква стойност. Такива действия се наричат адресна аритметика. Трябва да се отбележи, че събирането с 2 увеличава адреса не с числото 2, а с броя байтове, отговарящи на 2 обекта от съответния тип. Тъй като тип `char` заема 1 байт, тип `int` — 4 байта (за IBM PC 2 байта), тип `double` — 8, добавянето на 2 към указател ще увеличи адреса с 2, 8 (4) или 16 байта в зависимост от неговия тип.

Упражнение 1-3. Дадени са дефинициите:

```
int ival = 1024;
int *iptr;
double *dptr;
```

Кои от следните присвоявания са неправилни? Обяснете защо.

- | | |
|-------------------------------------|------------------------------------|
| (a) <code>ival = *iptr;</code> | (b) <code>ival = ipptr;</code> |
| (c) <code>*iptr = ival;</code> | (d) <code>iptr = ival;</code> |
| (e) <code>*iptr = &ival;</code> | (f) <code>iptr = &ival;</code> |
| (g) <code>dptr = iptr;</code> | (h) <code>dptr = *iptr;</code> |

Упражнение 1-4. Променлива може да приема само една от следните три стойности: 0, 128, 255. Обсъдете предимствата и недостатъците от декларирането на тази променлива като:

- | | |
|-------------------------|--------------------------------|
| (a) <code>double</code> | (c) <code>unsigned char</code> |
| (b) <code>int</code> | (d) <code>char</code> |

Указател към низ

Най-често се дефинира указател от тип `char*`, защото всички операции с низове в C++ се извършват чрез указатели от този тип. Настоящият подраздел разглежда подробно употребата на `char*`. В глава 6 ще дефинираме клас `String` (низ). Низовете се задават чрез указател към първия им символ. Това означава, че всеки низ е от тип `char*`. Можем да дефинираме променлива от тип `char*` и да я инициализираме с някакъв низ:

```
char *st = "The expense of spirit\n";
```

Следващата програма изчислява дължината на низ. Тя използва адресна аритметика за предвижване по дължината на низа. Програмата съдържа цикъл, който завършва при достигане на символа `null`. Символът `null` се поставя от компилатора в края на всеки низ. В първия вариант на програмата има грешки. Можете ли да ги откриете?

```
#include <stream.h>

char *st = "The expense of spirit\n";

main() {
    int len = 0;
    while ( st++ != '\0' ) ++len;
    cout << len << ": " << st;
    return 0;
}
```

Програмата няма да работи, защото `st` трябва да се предшества от `"*"`. Така

```
st++ != '\0'
```

проверява дали адресът в указателя `st` не е равен на символа `null`, а не дали символът, адресиран от `st` е различен от `null`. Условието в `while` ще бъде вярно винаги и цикълът ще се изпълнява безкрайно. Втората версия на програмата коригира откритата грешка. Сега програмата завършва, но отново връща грешен резултат — не отпечатва сочения от `st` низ. Каква е причината за тази грешка?

```
#include <stream.h>

char *st = "The expense of spirit\n";

main() {
    int len = 0;
    while ( *st++ != '\0' ) ++len;
    cout << len << ": " << st;
    return 0;
}
```

Причината сега е, че в края на програмата `st` не адресира низа, а сочи символа за край на низ `null`. Това е и символът, който се извежда на екрана. Ето как може да се реши възникналият проблем:

```
st -= len;  
cout << len << ": " << st;
```

След компилация и изпълнение ще получим следния печат:

```
22: he expense of spirit
```

Полученият резултат все още не е очакваният. Можете ли да се справите с допуснатата грешка?

Следващият ред поправя грешката. `st` трябва да се премести назад с `len + 1` позиции, заради символа за край.

```
st -= len + 1;
```

Сега след компилиране и изпълнение на програмата ще се отпечата:

```
22: The expense of spirit
```

Получената програма е вярна, но по отношение на стила е далеч от желаното съвършенство. Изразът

```
st -= len + 1;
```

беше добавен за да коригира грешката, получена от директното увеличаване на `st`. Направеното връщане към първоначалната стойност на `st` не е алгоритмически издържано, но въпреки това програмата е лесна за разбиране. В малка програма като тази, един по-труден за разбиране оператор не създава проблеми. Но ако предположим, че тези оператори обхващат 20% от програмния текст при една нетривиална задача, става ясно, колко ще се затрудни четенето и проверката на програмата. Направените по-горе корекции, често се наричат "закърпване на дупките". Ние трябваше да "закърпим" програмата за да компенсираме грешките при проектирането и. По-добре да създадем нов проект. Една възможност да направим това е идеята да дефинираме втори указател и да го инициализираме с `st`:

```
char *p = st;
```

Сега можем да използваме `p` за изчисляване на дължината на низа, докато `st` остава непроменен и сочи началото на низа:

```
while (*p++ != '\0');
```

Нека направим подобрене в програмата, което ще позволи създаденото от нас да се използва и от други. Ще отделим частта, която изчислява дължината на низа в отделна функция, достъпна за всяка програма. Функцията се нарича `stringLength()`:

```
#include <stream.h>

void stringLength( char *st )
{ // пресмята дължината на st
  int len = 0;
  char *p = st;
  while ( *p++ ) ++len;
  cout << len << ": " << st;
}
```

Дефиницията

```
char *p = st;
```

отстранява недостатъците на първоначалния вариант. Изразът

```
while ( *p++ )
```

е кратък запис на

```
while ( *p++ != '\0' )
```

Сега ще променим `main()`, за да използваме дефиницията на `stringLength()`.

```
extern void stringLength( char* );
char *st = "The expense of spirit\n";
main() {
  stringLength( st );
  return 0;
}
```

`stringLength()` е записана във файл `string.C`. Да компилираме и изпълним получената програма:

```
$CC main.C string.C
$a.out
22: The expense of spirit
$
```

`stringLength()` отразява точно първоначалните изисквания. Тя не е достатъчно обща за да обслужва множество други програми. Представете си, че трябва да напишем функция, която определя дали два низа са равни. Може да използваме следния алгоритъм:

- Проверяваме дали двата указателя сочат към един и същ низ. Ако е така, двата низа са равни.

- В противен случай проверяваме дължините им. Ако те не са равни, низовете са различни.
- Ако дължините са равни, проверяваме всяка двойка съответни символи. Ако в поне една такава двойка има несъответствие, низовете са различни, иначе те съвпадат.

Дефиницията на `stringLength()`, не позволява използването и за тази задача. Затова трябва да я модифицираме. Тя трябва да връща дължината на низа. Отпечатването на низа може да се остави за извикващата функция. По-долу е даден новият вариант на `stringLength()`.

```
int stringLength( char *st )
{ // връща дължината на st
  int len = 0;
  while ( *st++ ) ++len;
  return len;
}
```

Читателят може да се изненада, че и тук `st` се увеличава директно. Сега няма да има проблеми, защото:

- За разлика от по-рано, в този вариант на `stringLength()` `st` не се използва за достъп до низа след промяната му.
- Всички промени на стойността на `st` в `stringLength()` изчезват след излизане от функцията. Казва се, че `st` е предадена по стойност (passed-by-value) във функцията `stringLength()`. Това означава, че `stringLength()` работи с копие на `st`. Раздел 3.6 разглежда предаването на параметри по стойност.

`stringLength()` може да се извика от всяка програма, която иска да изчисли дължината на низ. Ако желаем програмата да отпечата дължината на низа и самия низ, ще променим `main()`:

```
main() {
  int len = stringLength( st );
  cout << len << ": " << st;
  return 0;
}
```

`stringLength()` има същото действие като стандартната функция `strlen()`. Чрез включване на стандартния заглавен файл `string.h`, програмистът може да използва редица полезни функции за работа с низове, включително и следните:

```
// копира src в dst
char *strcpy( char *dst, char *src );

// сравнява два низа, връща 0, ако са равни
int strcmp( char *s1, char *s2 );

// връща дължината на st
int strlen( char *st );
```

За по-пълна информация и списък на всички стандартни функции за работа с низове, разгледайте документацията за съответната версия на C++.

Упражнение 1-5. Обяснете разликата между `0`, `'0'`, `"0"` и `"0"`.

Упражнение 1-6. Дадени са следните дефиниции на променливи:

```
int *ip1, ip2;  
char ch, *cp;
```

В кои присвоявания има грешки, свързани с типа? Обяснете защо.

- | | |
|---|----------------|
| (a) ip1 = "All happy families are alike"; | (b) cp = 0; |
| (c) cp = '\0'; | (d) ip1 = 0; |
| (e) ip1 = '\0'; | (f) cp = &'a'; |
| (g) cp = &ch; | (h) ip1 = ip2; |
| (i) *ip1 = ip2; | |

1.4 Тип reference

При дефиниране на обекти от тип reference, след спецификатора на тип трябва да стои символът "&". Дефиницията изисква задължително начална инициализация. Например:

```
int val = 10;  
int &refval = val;    // ok  
int &refval; // грешка: неинициализирано
```

Типът служи като алтернативно име на използвания при инициализацията обект. Операциите с този тип въздействат върху използвания за инициализация обект. Например:

```
refval += 2;
```

добавя 2 към val. Стойността на val става 12.

```
int ii = refval;
```

присвоява на ii текущата стойност на val, докато

```
int *pi = &refval;
```

инициализира pi с адреса на val. Тип reference може да се счита за специален указател. Изразът:

```
( *pi == refval && pi == &refval )
```

е верен винаги, когато pi и refval адресират един и същ обект. За разлика от обикновения указател, този трябва да се инициализира при дефиниране и повторната му инициализация е забранена.

Ако в дефиницията искаме да включим няколко обекта от тип reference, трябва да използваме списък от имена, като всяко име се предшества от "&". Например:


```
int i;  
int &r1 = i, r2 = i;    // един обект от тип за връзка, r1 и един обект от тип int, r2  
int r1, &r2 = i;       // един обект от тип за връзка, r2 и един обект от тип int, r1  
int &r1 = i, &r2 = i;    // два обекта от тип за връзка, r1 и r2
```

За инициализация може да се използва не само променлива, но и конкретно число. Тогава се създава временна променлива, която се инициализира с това число. Обектът от тип `reference` се инициализира с временната променлива. Например:

```
int &ir = 1024;
```

се преобразува в

```
int T1 = 1024;  
int &ir = T1;
```

Ако за инициализация се използва променлива от друг тип, отново се създава временна променлива. Например:

```
unsigned int ui = 20;  
int &ir = ui;
```

се преобразува в

```
int T2 = int(ui);  
int &ir = T2;
```

Тип `reference` се използва главно за предаване на параметрите на функция по адрес. На тази тема е посветен раздел 3.7.

1.5 Константен тип

В следващия цикъл `for` съществуват два проблема, свързани с използването на 512 като горна граница.

```
for (int i = 0; i < 512; ++i);
```

Първият проблем е свързан с читаемостта. Какво означава да проверим дали `i` е 512? Какво представлява стойността 512? От контекста на програмния фрагмент не става ясно какъв е нейният смисъл. Тя сякаш е "взета от въздуха". Вторият проблем е свързан с настройката на програмата. Представете си, че програмата се състои от 10 000 реда. Нека цикъл `for` с такава заглавна част се появява в 4% от кода. Ако се наложи 512 да се удвои, т.е. да се замени с 1024, трябва да се открият всичките 400 срещания на 512. Един единствен пропуск ще доведе до нежелани резултати.

Решение на двата проблема можем да получим като дефинираме променлива и я инициализираме с 512. Използвайки мнемонично име, например bufSize, създаваме лесно читаема програма. Сега проверката е спрямо този идентификатор:

```
i < bufSize;
```

При промяна на стойността на bufSize, 400-те срещания на горния ред вече не се нуждаят от откриване и корекция. Трябва да се промени само един ред от текста на програмата — реда с инициализацията на bufSize. Това не само е далеч по-лесна работа, но вероятността за грешка е минимална. Цената на това решение е една допълнителна променлива.

```
int bufSize = 512; // начална стойност на bufSize
//...
for (int i = 0; i < bufSize; ++i );
//...
```

Но при такова решение са възможни случайни промени на bufSize. Тук е показана една обща грешка, допускана от програмисти, свикнали с Паскал и подобни на него езици.

```
// случайни промени на стойността на bufSize
if ( bufSize = 1 )
//...
```

В C++, "=" е операция за присвояване, а "===" операция за проверка на равенство. Паскал и подобни на него езици, използват "==" за сравнение. Затова програмистът може да допусне грешка и да промени без да иска стойността на bufSize на 1. Такава грешка е трудна за откриване, защото програмистът не счита изрази, като горния, за неправилни. Това е причина много компилатори да издават предупреждение при откриване на подобни изрази.

За да нямаме такъв проблем ще използваме тип const. Той превръща една променлива в константа. Например,

```
const int bufSize = 512; // начална стойност на bufSize
```

дефинира bufSize като константа, инициализирана със стойността 512. Всеки опит за промяна на стойността на bufSize ще предизвика грешка при компилация. Обект от тип const може да се разглежда като променлива, чиято стойност може да се чете, но не може да се променя. Тъй като веднъж дефиниран такъв обект не може да се променя, той трябва да се инициализира още в дефиницията. Дефиниция без инициализация ще предизвика грешка при компилация. Например:

```
const double PI; // грешка: неинициализирана константа
```

Грешка би било също, ако указател получи адреса на обект от тип const. Ако това беше възможно, стойността на константа би могла да се промени индиректно. Например:

```
const double minWage = 3.60;  
double *p = &minWage; // грешка  
*p += 1.40;
```

Но програмистът може да дефинира указател, който адресира константа:

```
const double *pc;
```

В примера `pc` е указател към константа от тип `double`, но самият указател не е константа. Това означава следното:

1. Указателят `pc` може да се променя, за да адресира друга променлива от тип `double` по всяко време в програмата.
2. Стойността на обекта, който `pc` адресира, не може да се променя чрез `pc`. Например:

```
pc = &minWage;      //ok  
double d;  
pc = &d;            //ok  
  
d = 3.14159;        //ok  
*pc = 3.14159;      // грешка
```

Адресът на константа може да се присвои само на указател, който адресира константи, такъв като `pc`. Но на указател към константа, може да се присвои адреса на произволна променлива:

```
pc = &d;
```

Макар че `d` не е константа, нейната стойност не може да се променя чрез `pc`. Указатели към константи се дефинират често като формални параметри на функция. Раздел 3.6 разглежда подробно това тяхно приложение.

Разрешава се и дефинирането на константен указател. Например:

```
int errNumb;  
int *const curErr = &errNumb; // константен указател
```

`curErr` е константен указател към обект от тип `int`. Програмистът може да променя стойността на обекта, който `curErr` адресира. Например:

```
if ( *curErr ) {  
    errorHandler();  
    *curErr = 0;  
}
```

Той, обаче, не може да променя съдържащия се в `curErr` адрес. Например:

```
curErr = &myErrNumb;    // грешка
```

Позволено е и дефинирането на указател от тип `const` към обект от тип `const`:

```
const int pass = 1;  
const int *const true = &pass;
```

В този случай нито стойността на обекта, който `true` адресира, нито самият адрес може да се променят.

Упражнение 1-7. Обяснете значението на следващите дефиниции. Посочете неправилните.

- | | |
|----------------------------------|---|
| (a) <code>int i;</code> | (d) <code>int *const cpi;</code> |
| (b) <code>const int ic;</code> | (e) <code>const int *const cpic;</code> |
| (c) <code>const int *pic;</code> | |

Упражнение 1-8. Кои от следните инициализации са допустими? Обяснете защо.

- | | |
|---|--|
| (a) <code>int i = 'a';</code> | (b) <code>const int ic = i;</code> |
| (c) <code>const int *pic = &ic;</code> | (d) <code>int *const cpi = &ic;</code> |
| (e) <code>const int *const cpic = &ic;</code> | |

Упражнение 1-9. Използвайте дефинициите от горния пример за да определите кои от следните присвоявания са допустими. Обяснете защо.

- | | |
|---------------------------------|----------------------------------|
| (a) <code>i = ic;</code> | (d) <code>pic = cpic;</code> |
| (b) <code>pic = &ic;</code> | (e) <code>cpic = &ic;</code> |
| (c) <code>cpi = pic;</code> | (f) <code>ic = *cpic;</code> |

1.6 Изброим тип данни

Изброимият тип е крайно множество от константи. Елементите на типа се различават от еквивалентните `const` дефиниции по това, че с тях не се свързва адрес от паметта. Прилагането на операцията "&" към елемент на типа ще предизвика грешка. Изброим тип се дефинира чрез ключовата дума `enum`, следвана от списък от идентификатори, заградени във фигурни скоби. По подразбиране първият идентификатор получава стойност 0. Всеки следващ идентификатор получава стойност с единица по-голяма от стойността на предходния. Например следващата дефиниция присвоява 0 на `FALSE` и 1 на `TRUE`.

```
enum { FALSE, TRUE }; // FALSE == 0, TRUE == 1
```

Елемент на изброим тип може да получи явно някаква стойност. Отново при липса на явно присвояване, стойността на поредния елемент е с единица по-голяма от стойността на предходния. В следващия пример `FALSE` и `FAIL` имат стойност 0, а `PASS` и `TRUE` стойност 1:

```
enum { FALSE, FAIL = 0, PASS, TRUE = 1 };
```

Съществува възможност изброим тип да получи конкретно име. Изброим тип с име дефинира уникален тип. Той може да се използва като спецификатор на тип в други дефиниции. Например:

```
enum TestStatus { NOT_RUN = -1, FAIL, PASS };
enum Boolean { FALSE, TRUE };

main() {
    const testSize = 100;
    TestStatus testSuite[ testSize ];
    Boolean found = FALSE;
    for ( int i = 0; i < testSize; ++i ) testSuite[ i ] = NOT_RUN;
}
```

В по-ранни реализации на C++ изброим тип с име не е отделен тип. За съвместимост с по-старите версии, несъответствие на типа при инициализиране и присвояване не се счита за грешка в настоящата AT&T версия. Генерират се предупреждения (warnings). Например:

```
main() {
    TestStatus test = NOT_RUN;
    Boolean found = FALSE;
    test = -1;      // грешка: TestStatus = int
    test = 10;     // грешка: TestStatus = int
    test = found;   // грешка: TestStatus = Boolean
    test = FALSE;   // грешка: TestStatus = const Boolean
    int st = test;  // ok: автоматично преобразуване на типовете
}
```

Когато програмистът дефинира променливата test от тип TestStatus, компилаторът ще следи test да приема една от трите валидни стойности. В допълнение използването на изброими типове подобрява нагледността на програмата.

1.7 Масиви

Масивът е от еднотипни обекти. Отделните обекти нямат име; по-скоро всеки един от тях е свързан с някаква позиция в масива. Тази форма на достъп се нарича индексирание. Например,

```
int i;
```

дефинира отделен обект от тип int, докато

```
int ia[10];
```

дефинира масив от десет цели числа. Всеки отделен обект се нарича елемент на масива. Така чрез оператора

```
i = ia[2];
```

i получава стойността на втория елемент на ia. Аналогично, операторът

```
ia[7] = i;
```

присвоява на седмия елемент на ia стойността на i.

Дефиницията на масив съдържа спецификатор на тип, идентификатор и индекс. Индексът определя броя на елементите на масива и се загражда в квадратни скоби "[]". Стойността му трябва да бъде поне единица и да е константен израз, т.е. израз, чиято стойност може да се изчисли по време на компилация. Това означава, че в дефиницията на масив променлива не може да се използва като индекс. Следват няколко правилни дефиниции на масиви:

```
const  bufSize = 512,
       stackSize = 25,
       maxFiles = 20,
       staffSize = 27;
char inputBuffer[ bufSize ];
int tokenStack[ stackSize ];
char *filetable[ maxFiles - 3 ];
double salaries[ staffSize ];
```

Номерирането на елементите на масив започва от 0. За масив от 10 елемента правилните стойности на индекса са от 0 до 9 включително, а не от 1 до 10. Тази особеност е причина за програмни грешки. Следващата програма използва цикъл for, за да обходи елементите на масива, като инициализира съответния елемент със стойността на неговия индекс.

```
const SIZE = 10;
int ia[ SIZE ];

main() {
    for ( int i = 0; i < SIZE; ++i ) ia[i] = i;
}
```

Масив може да се инициализира още при дефинирането чрез списък от стойности, отделени със запетаи и заградени във фигурни скоби. Тази възможност се отнася само за глобални масиви, т.е. масиви, дефинирани извън функции. Разликата между глобални и локални дефиниции се разглежда в глава 3. Например:

```
const SZ =3;
int ia[ SZ ] = { 0, 1, 2 };
```

Масив, който се инициализира при дефиниране, не е необходимо да задава стойност на индекса. Компиляторът определя броя на неговите елементи по броя на заградените в скоби числа. Например:

```
// масив с 3 елемента
int ia[] = { 0, 1, 2 };
```

Ако се посочи стойност на индекса, броят на заградените в скоби числа не бива да надминава тази стойност. В противен случай компилаторът регистрира грешка. Когато стойността на индекса е по-голяма от броя на числата, останалите елементи на масива се инициализират с 0. Например:

```
// ia —> { 0, 1, 2, 0, 0 }  
const SZ = 5;  
int ia[ SZ ] = { 0, 1, 2 };
```

Масив от символи може да се инициализира чрез списък от символни константи, отделени със запетаи и заградени във фигурни скоби или чрез низ. Забележете, че двата начина не са еквивалентни, защото всеки низ съдържа един символ повече — символа за край null. Например:

```
char ca1[] = { 'C', '+', '+' };  
char ca2[] = "C++";
```

ca1 се състои от 3 елемента, а ca2 от 4. Следващата дефиниция е неправилна:

```
// грешка: "Pascal" се състои от 7 елемента  
char ch3[6] = "Pascal";
```

Инициализирането на един масив с друг и присвояването на един масив на друг не е позволено:

```
const int SZ = 3;  
int ia[SZ] = { 0, 1, 2 };  
int ia2[] = ia;      // грешка  
int ia3[SZ];  
ia3 = ia;            // грешка
```

За да копираме един масив във друг, трябва да копираме всеки елемент отделно. Ще създадем функция с такова предназначение с име `copyArray()`. `copyArray()` изисква два масива. Да ги наречем `target` и `source`. Те ще бъдат аргументи на функцията. В общия случай трябва да създадем различни функции за всеки възможен тип масиви, като кодът на функциите ще бъде един и същ. За нашите цели ще дефинираме `copyArray()` за масиви, чиито елементи са цели числа. Функцията има следния прототип:

```
void copyArray( int target[], int source[] );
```

Когато масив е аргумент на функция, функцията получава указател към първия елемент на масива. Информация за размера на масива липсва. Функцията може да узнае дължината на масива, като я получи чрез втори параметър. Ето самата функция:

```
void copyArray( int target[], int source[], int targetSize, int sourceSize )  
{ /* Функцията копира source в target, като нулира всеки от  
   * елементите на target, чиито индекс е по-голям от sourceSize */
```

```

int upperBound = targetSize;
if (targetSize > sourceSize) upperBound = sourceSize;
for (int i = 0; i < upperBound; i++) target[ i ] = source[ i ];
while (i < targetSize) target[ i++ ] = 0;
}

```

Всеки израз, който връща определена стойност, може да се използва за индексване на масив. В езика не е предвидена проверка за излизане извън границите на масива, нито при компилация, нито при изпълнение. Контролът е оставен на вниманието на програмиста.

Упражнение 1-10. Кои от дефинициите са допустими? Обяснете защо.

```

int getSize();
int bufSize = 1024;

```

- | | |
|--------------------------|---------------------------|
| (a) int ia[bufSize]; | (c) int ia[4 * 7 - 14]; |
| (b) int ia[getSize()]; | (d) int ia[2 * 7 - 14]; |

Упражнение 1-11. Защо следващата инициализация е грешна?

```

char st[11] = "fundamental";

```

Упражнение 1-12. В следващия програмен фрагмент има две грешки, свързани с индексването на масива ia. Кои са те?

```

main()
{
    const max = 10;
    int ia[ max ];
    for (int index = 1; index <= max; ++index) ia[ index ] = index;
    // ...
}

```

Многомерни масиви

В C++ се разрешава дефиниране на многомерни масиви. Всеки индекс на такъв масив се загражда в отделни квадратни скоби. Например,

```

int ia[4] [3];

```

дефинира двумерен масив. Първият индекс определя реда, а вторият колоната, в която се намира елемента. В този случай ia е двумерен масив с 4 реда, с по 3 елемента на ред. Двумерните масиви се наричат още матрици.

Многомерните масиви могат да се инициализират още при дефиниране:

```

int ia[4] [3] = {
    { 0, 1, 2 },
    { 3, 4, 5 },
    { 6, 7, 8 },
    { 9, 10, 11 }
};

```



```
{ 9, 10, 11 }  
};
```

Вложените скоби отделят елементите на различните редове. Те не са задължителни. Следващото инициализиране е еквивалентно на горното, но е по-неясно:

```
int ia[4][3] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11 };
```

Следващата дефиниция инициализира първия елемент на всеки ред. Останалите елементи се инициализират с 0.

```
int ia[4][3] = { {0}, {3}, {6}, {9} };
```

Ако махнем вложените скоби, резултатът е много различен. Ще се инициализират трите елемента на първия ред и първия елемент на втория ред. Останалите елементи ще се инициализират с 0.

```
int ia[4][3] = { 0, 3, 6, 9 };
```

Индексирането в многомерен масив изисква по една двойка скоби за всеки индекс. Следващият пример инициализира двумерен масив чрез два вложени цикъла `for`.

```
main()  
{  
    const rowSize = 4;  
    const colSize = 3;  
    int ia[ rowSize ][ colSize ];  
  
    for ( int i = 0; i < rowSize; ++i )  
        for ( int j = 0; j < colSize; ++j )  
            ia[ i ][ j ] = i + j;  
}
```

В Паскал и Ада индексите на многомерните масиви са заградени само в една двойка квадратни скоби. Макар че изразът

```
ia[ 1, 2 ]
```

е валиден и в C++, и в Ада, той има различен смисъл в двата езика. В Ада по този начин се обръщаме към втория елемент от първия ред на масива `ia`. В C++ стойността на израза е указател от тип `int*`, адресиращ първия елемент от трети ред. (Припомняме, че в C++ индексиранието започва от 0). Всъщност изразът е еквивалентен на израза `ia[2][0]`, като индексът 0 се подразбира. В C++ изразът

```
[ 1, 2 ]
```

се третира като comma expression, който връща една стойност — в случая [2].

Един израз се нарича comma expression, ако представлява последователност от изрази, отделени със запетаи. Такъв израз се изчислява отляво надясно чрез последователно изчисление на отделните изрази. Стойността на comma expression е стойността на най-десния израз. Например, резултатът от следващия comma expression е 3.

```
7, 6+4, ia[0][0] = 0, 4-1;    // comma expression
```

Връзка между указатели и масиви

Дефиницията на всеки масив включва четири различни неща: спецификатор на тип, идентификатор, операция за индексване "[]" и индекс, заключен между "[]". Например,

```
char buf[ 8 ];
```

дефинира buf като масив с 8 елемента от тип char.

За индексване на масив към идентификатора му се прилага операцията за индексване.

```
buf[ 0 ];
```

връща стойността на първия елемент на buf.

Какво ще се получи, ако операцията за индексване се пропусне? Какво представлява стойността на самия идентификатор?

```
buf;
```

Идентификаторът на масива връща адреса на първия елемент на buf. Той е еквивалентен на:

```
&buf[ 0 ];
```

Припомняме, че прилагането на операцията "&" към обект от някакъв тип, връща указател от същия тип. В случая обектът е от тип char. Следователно, buf ще върне стойност от тип char*. Тогава указател от тип char* може да се инициализира с идентификатора на масива. Например:

```
char *pbuf = buf;    // ok
```

pbuf и buf съдържат адреса на първия елемент на масива. За да достигнем следващия елемент, трябва да използваме един от двата еквивалентни изрази:

```
// еквивалентни методи на адресиране  
pbuf + 1;  
&buf[ 1 ];
```

Или по-общо:

```
for ( int i = 0; i < arraySize; ++i )
{
    pbuf + i;
    &buf[ i ];
}
```

От тук следва, че

```
*pbuf;
buf[ 0 ];
```

са еквивалентни и връщат стойността на първия елемент на масива. За да получим стойността на следващия елемент, можем да използваме един от двата изрази:

```
*(pbuf + 1);
buf[ 1 ];
```

или още по-общо:

```
*(pbuf + index);
buf[ index ];
```

Двата метода за адресиране на елементите на масив са взаимозаменяеми. Начинаещите програмисти използват операцията за индексване, вместо указатели. По-долу е дефинирана функция, която слепва два низа.

```
#include <stream.h>

char *catString( char *st1, char *st2 )
{ // съединява низа st2 с низа st1, ако са два различни низа
    // ако st1 не адресира низ — за разлика от st2 — функцията връща st2
    if ( st1 == 0 && st2 )
        return st2;

    // ако st1 и st2 адресират един и същ низ, функцията връща st1
    if ( st2 == 0 || st1 == st2 )
        return st1;

    for ( int i = 0; st1[ i ] != '\0'; ++i ) ; // преминава през низа st1

    for ( int j = 0; st2[ j ] != '\0'; ++i, ++j ) st1[ i ] = st2[ j ];

    st1[ i ] = '\0';
    return st1;
}
```

Двата метода за адресиране на блок от паметта изглеждат еквивалентни, но в действителност не е така. Дефиницията на масив заделя блок от паметта. Идентификаторът на масива се инициализира с адреса на този блок и не може да се променя в програмата. Следователно, идентификаторът на масив може да се разглежда като константен указател. Увеличаването на `buf` в следващата програма е недопустимо:

```
char buf[] = "rampion";

main() {
    int cnt = 0;
    while ( *buf ) {
        ++cnt;
        ++buf;          // грешка: не е разрешено увеличаване на buf
    }
}
```

Проблемът може да се преодолее чрез дефиниране на указател. За да се използва безопасно, указателят трябва да се инициализира с адреса на дефиниран обект или блок от паметта. Често за целта се използва операцията `new`. Виж раздел 4.1.

Упражнение 1-13. Следващата програма се компилира без съобщения за грешки или предупреждения, но грешка има. В какво се състои проблемът? Как може да се открие?

```
char buf[] = "fiddleferns";

main() {
    char *ptr = 0;
    for ( int i = 0; buf[ i ] != '\0'; ++i ) ptr[ i ] = buf[ i ];
}
```

1.8 Класове

В C++ дефинираните от потребителя типове се наричат класове. Класът е съвкупност от данни, най-често с различен тип, и операции за обработката им. Добре дефинираният клас може да се използва така лесно, както стандартните типове данни. Класовете се разглеждат подробно в глави 5 - 8. Този раздел ще ни запознае с тях чрез един пример. Ще създадем клас за масивите, чиито елементи са цели числа.

Четири най-досадни черти на масивите в C++ са:

1. Размерът на масива трябва да е константен израз, но програмистът не винаги знае колко голям масив му е необходим по време на компилация. Ще създадем масив, чийто размер може да се определя и променя по време на изпълнение на програмата.
2. Няма контрол за напускане на границите на масива. Например, следващият програмен фрагмент използва като горна граница грешна константа. Резултатът от изпълнението вероятно ще покаже, че в програмата има нещо некоректно. Тя неправилно променя съдържанието на паметта над горната граница на масива.

```
const int SIZE = 25;
const int SZ = 10;
int ia[ SZ ];
// ...
```

```

main() {
    // не улавя излизането извън границите на масива
    for ( int i = 0; i < SIZE; ++i )
        ia[ i ] = i;
    // ...
}

```

3. Когато масив е аргумент на функция, функцията трябва да има и аргумент за дължината на масива.
4. Масивите не могат да се копират само с една операция. Например:

```

int ia[ SZ ];
int ia2[ SZ ] = ia;      // не е реализирано

```

В този раздел се дефинира клас, чиято реализация преодолява четирите неудобства при работа с масиви. Използването на обектите от класа е просто. Конструирването на класове се описва подробно в глава 6. Тук ще дефинираме само клас `IntArray` и ще обсъдим неговата реализация.

```

const int ArraySize = 24; // стойност по подразбиране

class IntArray {
public:
    // операции с масиви
    IntArray( int sz = ArraySize );
    IntArray( const IntArray& );
    ~IntArray() { delete ia; }
    IntArray& operator=( const IntArray& );
    int& operator[] ( int );
    int getSize() { return size; }
protected:
    // вътрешно представяне на абстрактния тип данни
    int size;
    int *ia;
};

```

Дефиницията на клас съдържа две части: заглавна част и тяло на класа. Заглавната част включва ключовата дума `class` и името на класа. Тялото на класа се загражда във фигурни скоби и завършва с `;`.

Името на класа служи като спецификатор на тип в дефиницията на обекти. Следват няколко дефиниции на обекти от клас `IntArray`:

```

const int SZ = 10;
int mySize;
int ia[SZ]; // дефиниция на "стандартен" масив
IntArray myArray( mySize ), iA( SZ );
IntArray *pA = &myArray;
IntArray ia2; // масива ia2 е с 24 елемента

```

Тялото на класа съдържа дефиниции на неговите членове. Членове на класа са операциите над обектите на класа и данните, чрез които се представя той. Клас `IntArray` се представя чрез два члена:

1. променливата `size`, чиято стойност съдържа дължината на масива.
2. указателят `ia`, адресиращ паметта, в която се съхраняват елементите на масива.

Ключовите думи `protected` и `public` контролират достъпа до членовете на класа. Членовете в секция `public` са достъпни във всяка част на програмата. Членовете в секция `protected` са достъпни само във функциите-членове на класа. Това ограничаване на достъпа се нарича скриване на информация.

В общия случай само функциите-членове имат достъп до данните, чрез които се представя класа. Така се осигуряват две предимства:

1. Ако трябва да се промени представянето на класа, достатъчно е да се коригират неговите членове-функции. Програмите, които използват този клас, остават непроменени.
2. При неправилно използване на членовете, чрез които се представя класа, трябва да се тестват и коригират само функциите-членове на класа. Останалата част от програмата остава същата.

Три от функциите-членове на клас `IntArray` използват името на класа за свое име. Това са инициализиращите функции и функцията за освобождаване на заетата от масива памет. Функция, чието име се предшества от знака `"~"` се нарича деструктор. Другите две функции се наричат конструктори. Конструкторът

```
IntArray( int sz = ArraySize );
```

осигурява произволни декларации на масив. `sz` е размерът на масива. Ако потребителят не зададе дължина на масива, по подразбиране се взема стойността на `ArraySize`. Така се формира размерът на масива `ia2` в дефиницията:

```
IntArray ia2;
```

Сега ще дефинираме конструктора `IntArray(int)`. Той използва операцията `new`. `new` заделя динамична памет. Раздел 4.1 разглежда тази операция.

```
IntArray::IntArray( int sz )
{
    size = sz;
    // заделя памет за масива
    ia = new int[ size ];      // ia се инициализира с адреса на заделената памет
    // инициализация на масива
    for ( int i = 0; i < sz; ++i ) ia[ i ] = 0;
}
```

Операцията `"::"` указва принадлежност на функция към даден клас. В случая тя указва, че функцията `IntArray(int)` е член на клас `IntArray`. Функция, която е член на клас, има директен достъп до членовете на този клас. В дефиницията на `IntArray(int)` използвахме израза:

```
size = sz;
```

size представлява дължината на всеки масив от тип IntArray. Вторият конструктор на класа инициализира един масив от тип IntArray с друг масив от същия тип. Той се извиква автоматично при всяка дефиниция от вида:

```
IntArray iA3 = myArray;
```

Дефиницията на втория конструктор е подобна на дефиницията на първия, но сега елементите и дължината на масива се копират.

```
IntArray::IntArray( const IntArray &iA )
{
    size = iA.size;
    ia = new int[ size ];
    for ( int i = 0; i < size; ++i ) ia[ i ] = iA.ia[ i ];
}
```

Операциите "." и "—>" осъществяват достъп до членовете на даден клас.

1. Операцията "." осъществява достъп до членовете чрез обект от класа.
2. Операцията "—>" осъществява достъп до членовете чрез указател към обект от класа.

Например:

```
iA.size;
```

реализира достъп до члена size на обекта iA. Операцията

```
IntArray& operator=( IntArray& )
```

присвоява на масив друг масив. Името на функцията изглежда странно, но така стандартната операция "=" разширява действието си за обекти от клас IntArray. Раздел 6.3 обсъжда подробно дефинирането на операции за класове. Сега да дефинираме тази функция. Забележете, че тя изравнява размерите на двата масива.

```
IntArray& IntArray::operator=( const IntArray &iA )
{
    delete ia;                // освобождава заетата от ia памет
    size = iA.size;           // изравнява размерите
    ia = new int[ size ];      // заделя ново парче памет
    for ( int i = 0; i < size; ++i )
        ia[ i ] = iA.ia[ i ]; // копира
    return *this;
}
```

Виж раздел 5.4 за смисъла на израза: return *this;

Дефинираната по-горе операция за присвояване се вика автоматично, когато един масив от тип IntArray се присвоява на друг масив от същия тип. Например:

```
ia2 = myArray;
```

Конструируваният клас няма да е полезен, ако не се реализира операцията индексирание. Трябва да може да се използват конструкции от вида:

```
for ( int i = 0; i < upperBound; ++i )  
myArray[ i ] = myArray[ i ] + i;
```

където `upperBound` е стойността на `size` за обекта `myArray`. Горният оператор е допустим благодарение на функциите-членове `getSize()` и `operator[]`. `getSize()` е съвсем малка функция, затова нейната дефиниция е включена в дефиницията на класа. `upperBound` може да получи стойността, която връща функцията `getSize()`:

```
upperBound = myArray.getSize();
```

Възможно е `getSize()` да замени `upperBound`:

```
for ( int i = 0; i < myArray.getSize(); ++i )
```

Описанието на `operator[]` не е по-дълго от това на `getSize()`, но е необходима някаква хитрина за да се осигури възможност за четене и запис на стойност. Например:

```
int i = myArray[ someValue ];  
myArray[ i ] = someValue;
```

За да може `myArray[i]` да участва отляво в оператора за присвояване, `operator[]` трябва да връща стойност от тип `reference`. Виж раздел 3.7, който разглежда функциите с тип на резултата `reference`. Следва дефиницията на `operator[]`:

```
int& IntArray::operator[](int index)  
{  
    return ia[index];  
}
```

Обикновено дефиницията на класа и свързаните с него константи се записват в заглавен файл. Заглавният файл се именува с името на класа. В случая заглавният файл се нарича `IntArray.h`. Всички програми, които използват клас `IntArray`, трябва да включат този заглавен файл. Аналогично, функциите-членове на класа се отделят в текстов файл с името на класа — в случая `IntArray.C`. Потребителските програми могат да използват тези функции, ако се свържат с тях в изпълнима програма. При това функциите трябва да се компилират с всяка програма, която ги използва. По-добре е да ги компилираме веднъж и да ги запазим в библиотека. Това се прави така:

```
$CC -c IntArray.C  
$ar cr IntArray.a IntArray.o
```


ar е команда за създаване на архивна библиотека в системата UNIX. Символите `sg`, които следват, са опции за тази команда. `IntArray.o` е файлът с обектния код на програмния текст. Той се генерира при извикване на компилатора с опцията `-c`. `IntArray.a` е името на библиотеката за клас `IntArray`. За да се използва библиотеката, името и трябва да се укаже явно в командния ред при компилация:

```
$CC main.c IntArray.a
```

Така функциите-членове на клас `IntArray` се включат в изпълнимата програма.

Упражнение 1-14. Създаденият клас илюстрира минимален брой възможности. Опишете други функции, които считате за необходими.

Упражнение 1-15. Едно полезно качество на `IntArray` е възможността обект от класа да се инициализира с целочислен масив. Опишете алгоритъм за реализиране на конструктора:

```
IntArray::IntArray( int *ia, int size );
```

Чрез него могат да се дефинират следните масиви:

```
int ia[ 4 ] = { 0, 1, 2, 3 };  
IntArray myIA( ia, 4 );
```

`IntArray` се нарича абстрактен тип данни. Потребителят може да използва клас `IntArray`, както използва стандартните типове данни в C++. Този аспект на класовете се разглежда подробно в глави 5 и 6.

Втора важна характеристика на класовете е механизмът на наследяване. Например, може да се дефинира клас `IntArrayRC`, подклас на `IntArray`, който проверява коректността на индекса. Следва дефиницията на `IntArrayRC`:

```
#include "IntArray.h"  
  
class IntArrayRC : public IntArray {  
public:  
    // конструкторите не се наследяват  
    IntArrayRC( int = ArraySize );  
    int& operator[] ( int );  
protected:  
    void rangeCheck( int );  
};
```

`IntArrayRC` трябва да дефинира само различните и допълнителните функции:

1. Трябва да се даде нова дефиниция на операцията за индексване, която следи коректността на индекса.
2. Трябва да се дефинира функция, която проверява коректността на индекса.
3. Трябва да се дефинират конструкторите на клас `IntArrayRC`.

Всички членове на `IntArray` са членове и на `IntArrayRC`, ако не са предефинирани в `IntArrayRC`. Такъв е смисълът на израза:

```
class IntArrayRC : public IntArray
```

Символът ":" показва, че `IntArrayRC` е производен клас на `IntArray`, т.е. негов подклас. `IntArrayRC` наследява членовете на `IntArray`. `IntArrayRC` може да се счита за разширение на `IntArray`, защото осигурява допълнителни възможности. Ще дефинираме новата операция за индексирание:

```
int& IntArrayRC::operator[]( int index ) {  
    rangeCheck( index );  
    return ia[ index ];  
}
```

`rangeCheck()` проверява стойността на индекса. Ако индексът е невалиден, извежда съобщение за грешка и прекратява изпълнението на програмата. `exit()` предизвиква спиране на програмата. Нейният аргумент е стойността, която програмата връща. Прототипът на `exit()` се намира в заглавния файл `stdlib.h`. Следва описанието на `rangeCheck()`:

```
#include <stdlib.h>  
#include <stream.h>  
  
enum { ERR_RANGE = 17 };  
void IntArrayRC::rangeCheck( int index ) {  
    if ( index < 0 || index >= size ) {  
        cerr << " Индексът е извън границите на масива:"  
            << "\n\tРазмер: " << size  
            << "\tИндекс: " << index << "\n";  
        exit( ERR_RANGE );  
    }  
}
```

Тъй като конструкторите не се наследяват, трябва да дефинираме конструктор на класа `IntArrayRC`:

```
// IntArrayRC трябва само да предаде своя аргумент на конструктора на IntArray  
IntArrayRC::IntArrayRC( int sz )  
    : IntArray( sz )  
    {} // Тялото на функцията е празно
```

Частта от конструктора, отделена със символа ":", се нарича списък за инициализация на членове (member initialization list). Тялото на конструктора е празно, защото той предава своя аргумент на конструктора на `IntArray`. Да разгледаме един пример за използване на клас `IntArrayRC`:

```
#include "IntArrayRC.h"  
const size = 12;  
  
main()  
{
```

```

IntArrayRC ia( size );
// грешка при индексирание
for ( int i = 1; i <= size; ++i ) ia[ i ] = i;
}

```

Програмата неправилно индексира `ia` от 1 до `size`, вместо от 0 до `size-1`. След компилиране и изпълнение получаваме:

Индексът е извън границите на масива:
 Размер: 12 Индекс: 12

Проверката за правилно индексирание осигурява контрол при употребата на масиви, но този контрол увеличава времето за изпълнение. Програмистът може да комбинира използването на `IntArrayRC` и `IntArray` в различни части на програмата. Наследяването обезпечава това по два начина:

1. Клас, от който произлизат други класове, се нарича основен клас. На обект от този клас може да се присвои обект от негов подклас. Например:

```

#include "IntArray.h"

void swap( IntArray &ia, int i, int j )
{
    // разменя i-ти и j-ти елемент на ia
    int tmp = ia[ i ];
    ia[ i ] = ia[ j ];
    ia[ j ] =tmp;
}

```

`swap()` може да се извика както с аргумент от тип `IntArray`, така и с аргумент от неговия подклас. Да дефинираме следните два обекта:

```

IntArray ia1;
IntArrayRC ia2;

```

Следващите две обръщения към `swap()` са правилни:

```

swap( ia1, 4, 7 );
swap( ia2, 4, 7 );

```

Съществува един проблем. Искаме операцията за индексирание да действа различно за класовете `IntArray` и `IntArrayRC`. Когато извикаме

```

swap( ia1, 4, 7 );

```

трябва да се изпълни операцията за индексирание на клас `IntArray`, докато при обръщението

```

swap( ia2, 4, 7 );

```

трябва да се изпълни операцията за индексване на клас IntArrayRC. Това означава, че операцията за индексване в swap() трябва да се променя в зависимост от действителния тип на аргумента. В C++ това се реализира автоматично чрез механизма на виртуалните функции (class virtual functions).

2. Виртуалните функции са членове, които се наследяват, като тяхната реализация зависи от действителния тип на класа. Такава е операцията за индексване в нашия пример.

За да направим операцията за индексване виртуална, трябва да модифицираме нейната декларация в тялото на клас IntArray:

```
class IntArray {
public:
    virtual int& operator[] ( int );
};
```

Сега при всяко извикване на swap() в зависимост от действителния тип на аргумента, ще се изпълни съответната операция за индексване. Следва един пример:

```
#include <stream.h>
#include "IntArray.h"
#include "IntArrayRC.h"

void swap( IntArray&, int, int );

main() {
    const size = 10;
    IntArray ia1[ size ];
    IntArrayRC ia2[ size ];

    // грешка: трябва да бъде size-1
    cout << "swap(), извикана за IntArray ia1\n";
    swap( ia1, 1, size );
    cout << "swap(), извикана за IntArrayRC ia2\n";
    swap( ia2, 1, size );
}
```

След компилация и изпълнение на тази програмата се отпечатва следния резултат:

```
swap(), извикана за IntArray ia1
swap(), извикана за IntArrayRC ia2
Индексът е извън границите на масива:
Размер: 10      Индекс: 10
```

Механизмът на наследяване и виртуалните функции са двата главни компонента на обектно-ориентираното програмиране. Те са разгледани в глави 7 и 8.

Упражнение 1-16. Въведете допълнителни възможности за клас IntArray. Какви допълнителни операции и данни изискват те? Необходимо ли е да се заменят вече съществуващи операции. Кои са те?

1.9 Описанието typedef

Масивите и указателите могат да се разглеждат като производни типове. Те са създадени на базата на други типове чрез операциите за индексване и адресиране. Тези операции могат да се разглеждат като вид конструктори. От своя страна производните типове могат да се използват за създаване на нови производни типове. Такъв тип е типът масив от указатели.

```
char *winter[ 3 ];  
char *spring[] = { "Март", "Април", "Май" };
```

winter и spring са масиви. Всеки от тях се състои от три елемента от тип char*, като spring е инициализиран. Изразът

```
char *cruellestMonth = spring[ 1 ];
```

инициализира cruellestMonth с адреса на низа "Април", адресиран до момента от втория елемент на spring.

Следващите два оператора за печат са еквивалентни:

```
main() {  
    cout << "Люлякът цъфти през"  
          << spring[ 1 ];  
    cout << "Люлякът цъфти през"  
          << cruellestMonth;  
}
```

Описанието typedef позволява да се въведат мнемонични синонимни имена за съществуващи стандартни, производни или дефинирани от потребителя типове данни. Например:

```
class IntArray;  
  
typedef double wages;  
typedef IntArray testScores;  
typedef unsigned int bitVector;  
typedef char *string;  
typedef string monthTable[3];
```

Имената, дефинирани чрез typedef, могат да се използват като спецификатори на тип в дефинициите на програма:

```
const classSize = 93;  
  
string myName = "stan";  
wages hourly, weekly;  
testScores finalExam( classSize );  
monthTable summer, fall = { "September", "Oktober", "November" };
```

Описанието `typedef` започва с ключовата дума `typedef`, следвана от тип на данните и идентификатор. Идентификаторът не представлява нов тип. Той е един синоним на името на съществуващ тип. Синонимът може да се използва навсякъде, където се използва оригиналното име.

Описанието `typedef` може да намали сложността на декларации. С тази цел се използва при документиране на програмите. То подобрява читаемостта на дефинициите на указатели към функции и указатели към функции-членове на клас. (Тези типове указатели се разглеждат в глави 4 и 5.)

Описанието `typedef` се използва и за намаляване на машинно зависимите параметри на програмата. Например, за някои компютри тип `int` може да е достатъчен за представяне на някакво множество, но при други може да се наложи да се използва тип `long`. Заменянето на стандартния тип с въведен от `typedef` тип, спестява време и усилия за коригиране на програмата при преминаване от един компютър на друг.

ГЛАВА 2

В глава 1 разгледахме стандартните типове данни и конструирането на нови типове в C++. В тази глава ще се спрем на основните операции и управляващи конструкции. Глава 3 е посветена на дефинирането на операции от потребителя.

2.1 Какво е това израз?

Изразът е съвкупност от една или повече операции. Обектите над които се извършват операциите се наричат операнди. Операциите с един операнд са унарни, а с два операнда бинарни. Операндите на бинарна операция се означават като ляв и десен операнд. Някои операции могат да бъдат както бинарни, така и унарни. Например,

```
*ptr
```

представява унарна операция за извличане на стойност чрез указател. Тя връща стойността на обекта, адресиран от ptr. В израза

```
var1 * var2
```

операцията "*" е бинарна. Тя намира произведението на var1 и var2.

Изразите се изчисляват чрез последователно изпълнение на включените в тях операции, като при това се спазва техният приоритет. Ако не е уговорено друго, резултатът от изчислението е rvalue. Типът на резултата зависи от типа на участващите в израза операнди. Когато участват операнди от различен тип, се прилагат правилата за преобразуване на типове. Раздел 2.10 разглежда тези правила подробно.

Израз, в който участват две или повече операции, се нарича съставен. Редът на изпълнение на операциите зависи от приоритета им. Приоритетът на операциите ще разгледаме след като се запознаем със основните операции в C++.

Най-простите изрази се състоят от една литерална константа или променлива. В тези изрази има един операнд без операция. Стойността им е стойността на самия операнд. По-долу са дадени няколко примера:

```
3.14159  
"melancholia"  
upperBound
```

Резултатът от 3.14159 е 3.14159. Неговия тип е double. Резултатът от melancholia е адрес от паметта, където се намира първият елемент на низа. Неговия тип е char*. Резултатът от upperBound е rvalue на тази променлива. Типът на резултата зависи от дефиницията на upperBound. Следващите раздели са посветени на основните операции в C++.

2.2 Аритметични операции

В табл. 2.1 са дадени всички аритметични операции в C++ с тяхното предназначение и употреба.

Оператор	Предназначение	Употреба
*	умножение	expr * expr
/	деление	expr / expr
%	остатък при деление	expr % expr
+	събиране	expr + expr
–	изваждане	expr - expr

Табл. 2.1. Аритметични операции

В C++ резултатът от делението на цели числа е цяло число. Ако точният резултат е дробно число, дробната част се отрязва. Например

```
21 / 6;
21 / 7;
```

връщат един и същ резултат 3.

Операцията "%" изчислява остатъка от делението на две цели числа. Нейните операнди задължително са от тип `int`. Левият операнд е делимо, а десният — делител. Да разгледаме няколко примера:

```
3.14 % 3 // грешка: левият операнд е от тип float
21 % 6   // ок: резултатът е 3
21 % 7   // ок: резултатът е 0

int i;
double f;

i % 2     // ок: ненулев резултат показва, че i е нечетно
i % f     // грешка: десният операнд е от тип float
```

При пресмятане на аритметични изрази може да се получи неправилна или неопределена стойност — т.нар. критични грешки (arithmetic exceptions). Те произтичат от математически правила (например деление на нула) или от представянето на числата в компютъра (например препълване отгоре, т.е. overflow). Знаем, че за променлива от тип `unsigned char` се заделят 8 бита. В тях може да се запишат стойностите от 0 до 255. В следващия пример на променлива от тип `unsigned char` се присвоява стойност 256.

```
unsigned char uc = 32;
int i = 8;
uc = i * uc; // препълване
```

За представяне на числото 256 са необходими 9 бита. Затова, когато на `uc` се присвои 256, ще се получи препълване. Действителната стойност на `uc` ще е различна за различните компютри, в зависимост от това колко байта са отделени за тип `unsigned char`.

2.3 Операции за сравнение и логически операции

Операциите за сравнение и логическите операции дават като резултат стойност истина (1) или лъжа (0). Те са изброени в табл. 2.2.

Оператор	Предназначение	Употреба
!	логическо отрицание	! expr
<	по-малко	expr < expr
<=	по-малко или равно	expr <= expr
>	по-голямо	expr > expr
>=	по-голямо или равно	expr >= expr
=	равенство	expr == expr
!=	неравенство	expr != expr
&&	логическо И	expr && expr
	логическо ИЛИ	expr expr

Табл. 2.2 Операции за сравнение и логически операции

Логическата операция AND "&&" дава истина, ако двата и операнда имат стойност истина. Логическата операция OR "||" дава истина, ако поне един от двата операнда има стойност истина. Операндите се изчисляват отляво надясно. Изчисляването може да спре преди края на израза, ако е ясно каква ще бъде неговата стойност. Да разгледаме изразите:

```
expr1 && expr2  
expr1 || expr2
```

expr2 не се изчислява, в следните случаи:

- Ако в израза с AND, expr1 има стойност лъжа.
- Ако в израза с OR, expr1 има стойност истина.

Употребата на AND е полезна, ако expr1 не е удовлетворен и изчислението на expr2 е опасно. Например:

```
while ( ptr != 0 &&  
        ptr->value < upperBound &&  
        notFound( ia[ ptr->value ] ) )
```

Указател със стойност нула не адресира обект. Избор на член от структура чрез указател, чиято стойност е нула, ще предизвика големи неприятности. Първият операнд на AND предотвратява тази възможност. Неприятности ще имаме и ако излезем извън границите на масива. Вторият операнд се грижи това да не се случи. Изчисляването на третия операнд е безопасно, само ако първите два операнда връщат стойност истина.

Логическата операция NOT "!" дава стойност, ако нейният операнд има стойност нула. В противен случай нейната стойност е лъжа. Например:

```

int found = 0;
while ( !found ) {
    found = lookup( *ptr++ );
    if ( ptr == endPtr )      // ptr сочи края
        return 0;
}
return ptr;

```

Изразът

```
( !found )
```

върща стойност истина, когато found е равно на 0.

Използването на логическата операция NOT касае въпроса за стила на програмиране. Например значението на изрази

```
( !found )
```

е ясно — не е открит (not found). Но какво означава следващото условие?

```
!strcmp( string1, string2 )
```

strcmp() е стандартна функция, която проверява дали два низа са равни. Ако функцията върне резултат нула, низовете са равни.

```
!strcmp( string1, string2 )
```

означава, че string1 е равен на string2. Използването на операцията NOT в този случай затруднява четимостта на програмата.

2.4 Операция за присвояване

Левият операнд на операцията за присвояване трябва да е lvalue. Ефектът от присвояването е запазване на новата стойност на адреса на левия операнд. Нека имаме следните три дефиниции:

```

int i;
int *ip;
int ia[ 4 ];

```

Тогава следващите операции за присвояване са правилни:

```

ip = &i;
i = ia[ 0 ] + 1;
ia[ *ip ] = 1024;
*ip = i * 2 + ia[ i ];

```

Стойността на израза отдясно на операцията за присвояване е резултатът от нейното изпълнение. Типът на резултата е типа на левия операнд.

Възможни са изрази, в които участват няколко операции за присвояване, като всяка от променливите получава съответстваща на типа и стойност, т.е. използват се правилата за преобразуване на типове. Например:

```
main()
{
    int i, j;
    i = j = 0; // ок: i и j получават стойност 0
    //...
}
```

i и j получават стойност 0. Операциите се изпълняват отдясно наляво.

Изрази с няколко операции за присвояване могат да се използват за по-компактен запис. За да демонстрираме тази възможност ще препишем конструктора на клас `IntArray` с малки промени. Виж раздел 1.8, където е даден оригиналният вариант.

```
IntArray::IntArray( int sz )
{
    ia = new int[ size = sz ];
    //...
}
```

Използването на такива означения зависи от стила на програмиста. Желателно е компактността на записа да не затруднява разглеждането на програмата. За компактност на записа се използват съкратени означения. Например:

```
int arraySum( int ia[], int sz )
{
    int sum = 0;
    for ( int i = 0; i < sz; ++i )
        sum += ia [ i ];
    return sum;
}
```

Общият вид на съкратените записи е следният:

```
a op= b;
```

където `op=` може да бъде една от следните десет операции: `+=`, `-=`, `*=`, `/=`, `%=`, `<=`, `>=`, `&=`, `~=`, `|=`. Всеки съкратен запис е еквивалентен на:

```
a = a op b;
```

Следователно, вместо горния запис можем да запишем:

```
sum = sum + ia [ i ];
```

Упражнение 2-1. В следващия фрагмент има грешка. В какво се състои тя? Опитайте да я отстраните.

```
main() {  
    int i = j = k = 0;  
}
```

Упражнение 2-2. Открийте грешката в програмния фрагмент и се опитайте да я коригирате.

```
main()  
{  
    int i, j;  
    int *ip;  
    i = j = ip = 0;  
}
```

2.5 Операциите "++" и "--"

Операциите "++" и "--" са компактен запис на действията прибавяне и изваждане на единица. Те могат да се разглеждат като операции за присвояване. Техният операнд трябва да бъде lvalue. Двете операции имат префиксна и постфиксна форма:

```
main()  
{  
    int c;  
    ++c;    // префиксен запис  
    c++;    // постфиксен запис  
}
```

Употребата на префиксната и постфиксната форма на тези операции ще илюстрираме чрез клас IntStack — стек, който запазва цели числа. IntStack изпълнява две операции:

1. поставя стойността на v на върха на стека: push(int v).
2. извлича стойността от върха на стека: pop().

При тези операции са възможни следните критични ситуации:

1. Ситуация overflow: прилагане на push() при пълен стек.
2. Ситуация underflow: прилагане на pop() при празен стек.

IntStack може да се представи като масив от цели числа. Следователно за представяне на класа са необходими следните членове:

```
int size;  
int *ia;  
int top;
```

top ще бъде винаги индексът на най-горния елемент на стека. Следователно, за празния стек top е -1, а за пълния, size — 1.

Функциите isEmpty() и isFull() са тривиални:

```
typedef int Boolean;
extern const int BOS;           // Bottom of Stack (дъно на стека)

Boolean IntStack::isEmpty()     { return (top == BOS); }
Boolean IntStack::isFull()      { return (top == size - 1); }
```

Дефиницията на push() илюстрира префиксната форма на операцията "++". Припомняме, че top указва текущия връх на стека. След извикване на push(), top трябва да сочи новия връх. Затова push() увеличава top с единица.

```
void IntStack::push( int v )
{   // добавя v в стека
    if ( isFull() )
        grow();    // разширява стека
    ia[ ++top ] = v;
}
```

Префиксната форма на операцията "++" добавя единица към стойността на top, преди използването на top като индекс на ia. Тази форма е компактен запис на следните два израза:

```
top = top + 1;
ia[ top ] = v;
```

grow() увеличава размера на стека с някаква предварително зададена величина. Тази функция е дефинирана в раздел 4.1.

Дефиницията на pop() илюстрира постфиксната форма на операцията "--". Тъй като top указва върха на стека, след извличане на стойност, top трябва да се намали.

```
int IntStack::pop()
{   // връща най-горния елемент на стека
    if ( isEmpty() )
        // отчита грешка и изпълнява exit()
        ;
    return ia[ top-- ];
}
```

Постфиксната форма на операцията "--" намалява с единица стойността на top след използването на top като индекс на ia. Тази форма е компактен запис на следните два израза:

```
ia[ top ];
top = top - 1;
```

При изпълнение на return, стойността на top намалява с единица.

Вътрешното представяне на клас `IntStack` е като представянето на дефинирания по-рано клас `IntArray`. Ще използваме дефиницията на `IntArray` за да дефинираме `IntStack` като негов производен клас. За справка виж раздел 1.8.

```
#include "IntArray.h"
typedef int Boolean;
const int BOS = -1;      // Bottom of Stack (дъно на стека)

class IntStack : private IntArray {
public:
    IntStack( int sz = ARRAY_SIZE )
        :IntArray( sz ), top( BOS ) {}
    Boolean isEmpty();
    Boolean isFull();
    void push( int );
    int pop();
    void grow() {}
protected:
    int top;
};
```

В раздел 1.8 дефинирахме клас `IntArrayRC` като подклас на `IntArray`, но използвахме ключовата дума `public` пред името на основния клас. В горния пример пред името на основния клас стои запазената дума `private`. По този начин на обект от основния клас не може да се присвои обект от негов подклас. Например:

```
extern swap( IntArray&, int, int );
IntArray ia;
IntArrayRC ia1;
IntStack is;
ia = ia1;    // ок: основният клас е public
ia = is;     // грешка: основният клас е private

swap( ia1, 4, 7 );    // ок
swap( is, 4, 7 );    // грешка
```

Второ, когато основният клас е `private`, неговите членове от секция `public` се смятат за членове на секция `private` в производния клас. Следователно, членовете на основния клас `IntArray` не са достъпни чрез обект от клас `IntStack`. Например:

```
int bottom = is[ 0 ]; // грешка: operator[] е в секция private на клас IntStack
```

Наследяването може да се използва по два начина:

1. Използване на метод (операция) на основния клас като метод на съответния подклас. Така беше при `IntArrayRC` и `IntArray` в глава 1.
2. Използване на дефиницията на един клас за създаване на нов клас. Така създадохме клас `IntStack`.

`IntStack` не е подклас на `IntArray`. Той не използва методите на `IntArray`. Ключовата дума `private` пред името на основния клас предотвратява случайно прилагане на методите на `IntArray` върху обект от клас `IntStack`. Освен това, ако обект от клас `IntStack` се присвои на

обект от клас `IntArray`, ще се разруши специфичната структура на стека. Това е втората причина да употребим `private` пред името на основния клас.

Следва един пример за използване на клас `IntStack`:

```
#include <stream.h>
#include "IntStack.h"

IntStack myStack;
const DEPTH = 7;

main() {
    for ( int i = 0; i < DEPTH; ++i ) myStack.push(i);
    for ( i = 0; i < DEPTH; ++i ) cout << myStack.pop() << "\t";
    cout << "\n";
}
```

Програмата генерира следния резултат:

```
6 5    4    3    2    1    0
```

Упражнение 2-3. Как мислите, защо `C++` не е наречено `++C` ?

2.6 Операция `sizeof`

Операцията `sizeof` връща размера на израз или тип данни в байтове. Тя се използва в една от следните две форми:

```
sizeof (type_specifier);
sizeof expr;
```

Ето пример за нейното използване:

```
int ia[] = { 0, 1, 2 };
const sz = sizeof ia / sizeof( int );
```

Следващата програма илюстрира употребата на `sizeof` върху различни типове данни.

```
#include <stream.h>
#include "IntStack.h"

main() {
    cout << "int :\t\t" << sizeof( int );
    cout << "\nint* :\t\t" << sizeof( int* );
    cout << "\nint& :\t\t" << sizeof( int& );
    cout << "\nint[3] :\t\t" << sizeof( int[3] );
    cout << "\n";          // за разделяне

    cout << "IntStack :\t\t" << sizeof( IntStack );
    cout << "\nIntStack* :\t\t" << sizeof( IntStack* );
    cout << "\nIntStack& :\t\t" << sizeof( IntStack& );
    cout << "\nIntStack[3] :\t\t" << sizeof( IntStack[3] );
    cout << "\n";
}
```

```
}
```

След компиляция и изпълнение на програмата на конкретна машина получаваме например следното:

```
int :          4
int* :         4
int& :         4
int[3] :       12

IntStack :     12
IntStack* :    4
IntStack& :    4
IntStack[3] :  36
```

2.7 Операция за условен израз (аритметичен if)

Тази операция има следния синтаксис:

```
expr1 ? expr2 : expr3;
```

expr1 се изчислява и ако неговата стойност е различна от нула, се изчислява expr2. В противен случай се изчислява expr3. Следващата програма използва аритметичен if.

```
#include <stream.h>

main() {
    int i = 10, j = 20, k = 30;

    cout    << "\nПо-голямото число от
             << i << "и" << j << "е"
             << ( i > j ? i : j );

    cout    << "\nСтойността на " << i
             << ( i % 2 ? "не е" : "е" )
             << "четно число";

    // Аритметичният if може да се влага,
    // но това може да затрудни четенето на текста.

    // max приема стойността на най-голямата от трите променливи
    int max = ( (i > j)
                ? ((i > k) ? i : k)
                : (j > k) ? j : k);
    cout    << "\nНай-голямото число от
             << i << ", " << j << " и " << k
             << " е " << max << "\n";
}
```

Програмата извежда следния резултат:

```
По-голямото число от 10 и 20 е 20
Стойността на 10 е четно число
```


2.8 Побитови операции

Побитовите операции интерпретират операнда или операндите си като съвкупност от битове. Те позволяват на програмиста да провери или присвои стойности на отделни битове или множество битове.

Операндите на побитовите операции трябва да бъдат от тип `int` (`long`, `short`, `unsigned`) или `char`. Могат да се използват операнди със или без знак, но препоръчителна е употребата на тези без знак. Знаковият бит на числото в побитова операция се третира различно в различните версии на езика. Затова програми, създадени за една версия, могат да не работят при друга.

Първо ще разгледаме как работят съществуващите побитови операции. След това ще дадем пример за използването им. В раздел 6.4 ще дефинираме клас `BitVector`.

Оператор	Предназначение	Употреба
<code>~</code>	допълнение до 1 (NOT)	<code>~ expr</code>
<code><<</code>	изместване наляво	<code>expr1 << expr2</code>
<code>>></code>	изместване надясно	<code>expr1 >> expr2</code>
<code>&</code>	побитово И (AND)	<code>expr1 & expr2</code>
<code>^</code>	изключващо ИЛИ (XOR)	<code>Expr1 ^ expr2</code>
<code> </code>	побитово ИЛИ (OR)	<code>expr1 expr2</code>

Табл. 2.3 Побитови операции

Побитовата операция **NOT** `"~"` инвертира битовете на своя операнд. Всеки бит, със стойност 1 се нулира и обратно, всеки нулев бит получава стойност 1.

`unsigned char bits = 0227;`

1	0	0	1	0	1	1	1
---	---	---	---	---	---	---	---

`bits = ~bits;`

0	1	1	0	1	0	0	0
---	---	---	---	---	---	---	---

Операциите за изместване `"<<"` и `">>"` преместват битовете на левия си операнд с няколко позиции наляво или надясно.

Излишните битове на операнда се премахват. Операцията `"<<"` добавя нулирани битове отляво. Операцията `">>"` добавя нулирани битове отляво, ако операндът е без знак. Ако операндът е със знак, се добавят битове със стойността на знаковия бит или нулирани битове. Последното е машинно зависимо.

`unsigned char bits = 1;`

0	0	0	0	0	0	0	1
---	---	---	---	---	---	---	---

`bits = bits << 1;`

0	0	0	0	0	0	1	0
---	---	---	---	---	---	---	---

bits = bits << 2;

0	0	0	0	1	0	0	0
---	---	---	---	---	---	---	---

bits = bits >> 3;

0	0	0	0	0	0	0	1
---	---	---	---	---	---	---	---

Операцията **побитово И (AND) "&"** изисква два операнда. Ако в двата операнда съответните битове са единици, единица ще е и стойността на съответния бит във резултата. При всеки друг случай този бит се нулира.

unsigned char result;

unsigned char b1 = 0145;

0	1	1	0	0	1	0	1
---	---	---	---	---	---	---	---

unsigned char b2 = 0257;

1	0	1	0	1	1	1	1
---	---	---	---	---	---	---	---

result = b1 & b2;

0	0	1	0	0	1	0	1
---	---	---	---	---	---	---	---

Операцията **изключващо ИЛИ (XOR) "^"** изисква два операнда. Бит от резултата съдържа единица, ако един от съответните битове в операндите е единица. При всеки друг случай този бит се нулира.

unsigned char result;

unsigned char b1 = 0145;

0	1	1	0	0	1	0	1
---	---	---	---	---	---	---	---

unsigned char b2 = 0257;

1	0	1	0	1	1	1	1
---	---	---	---	---	---	---	---

result = b1 ^ b2;

1	1	0	0	1	0	1	0
---	---	---	---	---	---	---	---

Операцията **побитово ИЛИ (OR) "|"** също изисква два операнда. Бит от резултата съдържа единица, когато поне един от съответните битове в операндите е единица. В противен случай този бит се нулира.

unsigned char result;

unsigned char b1 = 0145;

0	1	1	0	0	1	0	1
---	---	---	---	---	---	---	---

unsigned char b2 = 0257;

1	0	1	0	1	1	1	1
---	---	---	---	---	---	---	---

result = b1 | b2;

1	1	1	0	1	1	1	1
---	---	---	---	---	---	---	---

Променлива, инициализирана със съвкупност от отделни битове се нарича маска. Маската е средство за запазване на информация за наличие или отсъствие на множество признаци. Да разгледаме един пример. Преподавател обучава 15 студенти. Всяка седмица той провежда тест, оценката от който е "да" (издържал) или "не" (неиздържал). За запазване на резултатите от всеки тест ще използваме по една маска.

unsigned int quiz1 = 0;

Преподавателят трябва да може да записва 0 или 1 в конкретен бит на цялото число и да проверява съдържанието му. Например, ако десетият студент е издържал теста, десетият бит трябва да се установи в единица. Първо ще покажем как се установява конкретен бит в единица, като останалите битове остават нулирани. Ще приложим операцията за изместване наляво "<<" към константата единица².

1

0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	1

1 << 9

0	0	0	0	0	0	1	0
0	0	0	0	0	0	0	0

Ако между тази стойност и quiz1 се направи побитово ИЛИ, всички битове с изключение на десетия ще останат непроменени. Десетият ще бъде единица.

quiz1 |= 1 << 9;

Представете си, че преподавателят е сбъркал и се оказва, че десетият студент не е издържал теста. Трябва да поправим грешката като нулираме десетия бит. Това може да се направи по следния начин:

1. Всички битове на quiz1 се установят в единица, а десетият се нулира. За целта се инвертира 1 << 9.

~(1 << 9)

1	1	1	1	1	1	0	1
1	1	1	1	1	1	1	1

2. Между quiz1 и ~(1 << 9) се прави побитово И. Всички битове на quiz1 остават непроменени с изключение на десетия бит. Той се нулира.

Действията от точка 1 и точка 2 се изразяват чрез израза

² Броенето на битовете започва от нула и за да установим десетия бит, изместването наляво трябва да е с 9 бита

```
quiz1 &= ~(1 << 9);
```

Накрая ще покажем как се определя съдържанието на конкретен бит от quiz1. Първо да установим този бит в единица — вече знаем как се прави това. Сега да приложим побитово И между quiz1 и числото, в което всички битове са нули в изключение на искания. В следващия пример променливата hasPassed е различна от нула, само когато десетият бит на quiz1 е единица.

```
int hasPassed = quiz1 & (1 << 9);
```

Упражнение 2-4. Дадени са две дефиниции:

```
unsigned int ui1 = 3, ui2 = 7;
```

Какво се получава като резултат от изразите:

- | | |
|----------------|----------------|
| (a) ui1 & ui2 | (c) ui1 ui2 |
| (b) ui1 && ui2 | (d) ui1 ui2 |

Упражнение 2-5. Какво ще се получи, ако на променлива от тип unsigned char се присвои числото 0377? Изобразете съдържанието на битовете.

Упражнение 2-6. Как може да се отдели вторият байт на променлива от тип int чрез побитови операции?

Упражнение 2-7. Степените на 2 могат да се намерят чрез операцията "<<" и константата 1. Генерирайте таблица от първите 16 степени на 2.

2.9 Приоритет на операциите

Приоритетът на операциите определя реда на тяхното изпълнение в сложен израз. Неговото познаване предотвратява много изчислителни грешки в програмите. Например, какъв е резултатът от аритметичния израз

```
6 + 3 * 4 / 2 + 2
```

Ако операциите се изпълняват отляво надясно, резултатът ще е 20. Други възможни резултати са 9, 14 и 36. Кой от тях е верният?

В C++ умножението и делението са с по-висок приоритет от събирането и изваждането. Това означава, че те се изпълняват първи. При това умножението и делението са с еднакъв приоритет. Операции с еднакъв приоритет се изпълняват отляво надясно. Тогава операциите в горния пример ще се изпълнят в следния ред:

1. $3 * 4 \rightarrow 12$
2. $12 / 2 \rightarrow 6$
3. $6 + 6 \rightarrow 12$
4. $12 + 2 \rightarrow 14$

По-долу е даден пример за една коварна грешка. Проблемът е в това, че операцията "!=" има по-висок приоритет от операцията за присвояване.

```
while ( ch = nextChar() != '\0' )
```

Очевидно програмистът има намерение да присвои на `ch` следващия символ и след това да провери дали този символ е различен от `null`. На практика, обаче, първо се проверява дали следващият символ е различен от `null`, а на `ch` се присвоява истина или лъжа, в зависимост от резултата от проверката. `ch` никога не получава стойността на следващия символ.

Приоритетът на операциите може да се отмени чрез скоби, които отделят подизраз от целия израз. При пресмятане на сложен израз, първо се изчисляват всички подизрази. Всеки подизраз се замества с резултата от пресмятането и изчисленията продължават. Ако има вложени скоби, изчисленията започват от най-вътрешните. Например:

```
4 * 5 + 7 * 2 —> 34
4 * ( 5 + 7 * 2 ) —> 76
4 * ( ( 5 + 7 ) * 2 ) —> 96
```

Като използваме скоби можем да поправим грешката от горния пример и да реализираме намерението на програмиста.

```
while ( (ch = nextChar()) != '\0' )
```

В таблица 2.4 са дадени всички операции в C++ поред на техния приоритет. 17R трябва да се чете като "ниво на приоритет 17, асоциативност отлясно наляво". Аналогично 7L означава "ниво на приоритет 7, асоциативност отляво надясно". По-високото ниво означава по-висок приоритет.

Упражнение 2-8. Използвайте таблица 2.4 и определете реда на изпълнение на операциите в изразите:

```
(a) ! ptr == ptr—> next
(b) ~ uc ^ 0377 & ui << 4
(c) ch = buf[ bp++ ] != '\0'
```

Упражнение 2-9. Горните изрази се пресмятат в противоречие с намерението на създателя им. Поставете необходимите скоби, за да може те да се изчисляват правилно.

Упражнение 2-10. Защо при компилация програмният фрагмент предизвиква грешка? Как може да я отстраните?

```
void doSomething();

main() {
    int i = doSomething(), 0;
}
```

2.10 Преобразуване на типове

В компютъра всеки тип се представя като последователност от битове. Информацията за типа е част от рецептата: "Вземи x на брой бита и ги интерпретирай според следния шаблон....".

Преобразуването на един стандартен тип в друг обикновено променя една или две характеристики на типа, но никога шаблона. Размерът може да бъде увеличен или намален, като това предизвиква промяна на интерпретацията.

Някои преобразувания са опасни и компилаторът предупреждава за това. Преобразуване на тип в тип с по-малък размер е опасно. Следват няколко конкретни примера:

```
long lval;  
unsigned char uc;  
  
int (3.14159);  
(signed char) uc;  
short (lval);
```

Следващите две означения служат за преобразуване на тип:

```
type (expr)  
(type) expr
```

Те преобразуват expr в тип type.

Горните примери илюстрират опасностите, които могат да се появят при намаляване на мястото в паметта, отделено за съответния тип. В първия случай се загубва дробната част:

```
// 3.14159 != 3.0  
3.14159 != double ( int (3.14159) );
```

Във втория случай за половината от възможните стойности на uc (128 - 256) се променя интерпретацията на шаблона. Най-левият бит вече се интерпретира като знаков. В третия случай lval, получава неопределена стойност, ако битовете, отделени за тип short са недостатъчни, за да се представи стойността на тази променлива.

Ниво	Операция	Предназначение
17R	:: (унарна)	глобална дефиниция
17L	:: (бинарна)	принадлежност на клас
16L	—>, .	избор на член
16L	[]	индексиране
16L	()	извикване на функция
16L	()	коструиране на типове
16L	sizeof	размер на обект в байтове
15R	++, --	добавяне/изваждане на 1
15R	~	побитово отрицание
15R	!	логическо отрицание
15R	+, -	унарен плюс/минус
15R	*, &	адресни операции
15R	()	преобразуване на тип
15R	new, delete	заемане и освобождаване на памет
14L	->*, .*	избор на член чрез указател
13L	*, /, %	умножение, деление, остатък
12L	+, -	събиране и изваждане
11L	<<, >>	изместване наляво/надясно
10L	<, <=, >, >=	операции за сравнение
9L	=, !=	операции за сравнение
8L	&	побитово И
7L	^	изключващо ИЛИ
6L		побитово ИЛИ
5L	&&	логическо И
4L		логическо ИЛИ
3L	?:	аритметичен IF
2L	=, *=, /=, %=, +=, -=, <<=, >>=, &=, =, ^=	присвояване комбинирано с други операции
1L	,	последователно изпълнение (списък)

Табл. 2.4 Приоритет и асоциативност на операциите

Някои преобразувания са безопасни за един вид компютри, а за друг вид не са. За повечето компютри тип `int` е еквивалентен на тип `short` или на тип `long`. Тогава едно от следващите преобразувания е опасно за всеки компютър, за който `short`, `long` и `int` не са три различни типа.

`unsigned short us;`
`unsigned int ui;`

```
int ( us );  
long ( ui );
```

Резултатът от преобразуването на един тип в друг понякога е твърде озадачаващ. Следващите два подраздела разглеждат автоматичното преобразуване на типове и преобразуване на типовете по начин, указан от програмиста. Раздел 6.5 представя преобразуванията между класове.

Автоматично преобразуване на типове

Автоматичното преобразуване на типове се извършва от компилатора без намесата на програмиста. То се прилага, когато в един и същи израз има смесване на типове. Съществуват правила, по които компилаторът преобразува типовете.

Например, при присвояване на стойност на обект, стойността се преобразува според типа на обекта. По аналогичен начин се преобразува и стойност, която се предава като фактически параметър на функция:

```
void ff( int );  
  
int val = 3.14159; // преобразува в цялото число 3  
ff( 3.14159 );    // преобразува в цялото число 3
```

И в двата случая компилаторът автоматично преобразува константата от тип `double` в константа от тип `int`. При това той предупреждава за отрязване на дробната част.

В аритметичен израз операндите се преобразуват в типа на операнда, чието представяне изисква най-много памет. Например:

```
val + 3.14159;
```

Съгласно казаното по-горе `val` се преобразува автоматично в тип `double`. Подобни преобразувания са известни като преминаване в тип с по-голям обхват (*type promotion*). Числото 3 се преобразува в 3.0 и едва след това се събира с 3.14159. Забележете, че стойността на `val` не се променя. Преобразуването се извършва върху копие на `val`. След използване, преобразуваната стойност не се запазва никъде.

```
val = val + 3.14159;
```

При пресмятане на горния израз се извършват две преобразувания. Първо, `val` се преобразува в `double`. Резултатът от събирането е 6.14159. Тъй като това число трябва да се присвои на променлива от тип `int`, следва отрязване на дробната част. `val` получава стойност 6. Действията на компилатора са същите и за еквивалентния израз

```
val += 3.14159;
```

Да разгледаме още един пример. Резултатът от изпълнението на следващите два оператора не е 20, а 23.


```
int i = 10;  
i *= 2.3    // резултатът е 23, а не 20
```

Резултатът е такъв, защото дробната част се отрязва след умножението.

Преобразуване по желание на потребителя

Истинско разточителство е да се извършват две преобразувания в изрази като

```
i = i + 3.14159;
```

Тъй като резултатът е от тип `int`, по-добре е да се отреже дробната част на 3.14159, отколкото `i` да се превръща в тип `double` и след това сумата да се преобразува в цяло число. От казаното става ясно, че възможна причина за дефиниране на явни преобразувания на типове, е желанието да съкратим някои стандартни преобразувания. Например в израза

```
i = i + int (3.14159);
```

3.14159 се преобразува в 3 и това число се събира със стойността на `i`. Резултатът от събирането се присвоява на `i`.

Стандартните преобразувания позволяват указател от произволен тип да се присвои на указател от тип `void*`. Този указател се използва, когато типът на обекта не е точно определен или ще се променя. Указател от тип `void*` не може да се използва директно, защото не съществува информация за типа и компилаторът не знае как да интерпретира съвкупността от битове. Затова такъв указател първо се превръща в указател от определен тип.

В C++ не съществуват правила за преобразуване на указател от тип `void*` в указател от конкретен тип, защото това не е безопасно. Ако присвоим указател от тип `void*` на указател от стандартен тип, ще получим грешка при компилация. Например:

```
int i;  
void *vp;  
int *ip = &i;  
double *dp;  
  
vp = ip;    // ок  
dp = vp;    // грешка: не е възможно автоматично преобразуване
```

Втора причина за дефиниране на явни преобразувания е желанието да предотвратим грешките, свързани с типа по време на компилация. В C++ чрез собствени преобразувания можем да присвоим произволна стойност на обект от произволен тип. При тези преобразувания компилаторът извежда предупреждения, но позволява използването им като предполага, че програмистът ще ги използва внимателно. Например:

```
dp = (int*)vp;    // ок: използваме собствено преобразуване  
*dp = 3.14;       // неприятност, ако dp адресира i
```

Трета причина за дефиниране на явни преобразувания е желанието да избегнем двусмислията, когато са възможни няколко преобразувания. Ще разгледаме по-подробно тази тема в раздел 4.3.

Упражнение 2-11. Дадени са следните дефиниции:

char ch;	unsigned char unChar;
short sh;	unsigned short unShort;
int intVal;	unsigned int unInt;
long longVal;	float fl;

Определете кои от следните присвоявания не са безопасни, защото намаляват броя на байтовете за представяне на данните или създават проблеми с интерпретацията им.

- | | |
|-----------------------|-----------------------|
| (a) sh = intVal; | (e) longVal = unInt; |
| (b) intVal = longVal; | (f) unInt = f1; |
| (c) sh = unChar; | (g) intVal = unShort; |

Упражнение 2-12. Като използвате дефинициите от упражнение 2-11, определете типа на резултата за следните изрази:

- (a) 'a' - 3
- (b) intVal * longVal - ch
- (c) f1 + longVal / sh
- (d) unInt + (unsigned int) longVal
- (e) ch + unChar + longVal + unInt

2.11 Оператори

Операторите са най-малката изпълнима част в програма на C++. Всеки оператор завършва с ";". Най-простият оператор е празният. Той има следния вид:

```
; // празен оператор
```

Празният оператор е полезен, когато синтаксисът на някакъв оператор изисква присъствие на поне един оператор, а логиката на програмата — не. Това се налага понякога при операторите while и for. Ето един пример:

```
while ( *string++ = *inBuf++ )  
    ; // празен оператор
```

Излишен празен оператор не предизвиква грешка при компилация. Например,

```
int val;; // излишен празен оператор
```

се състои от два оператора: оператор за деклариране int val; и празен оператор.

Декларация, следвана от ";" се нарича оператор за деклариране и е единствения оператор, който може да се използва извън тялото на функция.

Съставни оператори и блокове

Редица конструкции в езика позволяват да се използват само прости оператори. Понякога логиката на програмата изисква да се изпълни последователност от два или повече оператора. Тогава се използват съставни оператори. Например:

```
if ( account.balance - withdrawal <0 )
{ // съставен оператор
  issueNotice( account.number );
  chargePenalty( account.number );
}
```

Съставният оператор е последователност от оператори, заградени във фигурни скоби. Той може да се поставя навсякъде вместо прост оператор. За разлика от простия оператор не завършва с ";".

Съставен оператор, съдържащ един или повече оператори за деклариране, се нарича блок. Блоковете се разглеждат в раздел 3.10.

2.12 Управляващи оператори

По подразбиране операторите в C++ се изпълняват последователно. Всяка програма започва изпълнението си от първия оператор на функцията main(). Поред се изпълняват всички оператори от тялото на тази функция. Изпълнението завършва с последния оператор от main().

С изключение на най-простите програми последователното изпълнение на операторите не съответства на логиката на алгоритмите. Следващите подраздели разглеждат управляващите оператори в C++.

2.13 Оператор if

Оператор if проверява дали дадено условие е вярно. Ако това е така, се изпълнява един оператор — прост или съставен. В противен случай операторът не се изпълнява. Синтаксисът на оператор if е следният:

```
if ( израз ) оператор;
```

Изразът трябва да е заграден с кръгли скоби. Ако неговата стойност е различна от нула, условието се счита за вярно и оператора се изпълнява.

Да разгледаме един пример. Ще създадем функция-член на клас IntArray, която връща минималната стойност на елемент в масива. Ще запазим и броя на срещанията на този елемент. За целта ще използваме два условни оператора:

1. Ако текущата стойност е равна на минималната досега, ще увеличим броя с единица.
2. Ако текущата стойност е по-малка от минималната досега, ще заменим минималната с текущата стойност и ще инициализираме броя с единица.

Обхождането на масива става чрез оператора за цикъл `for`. За да определим минималния елемент, трябва да изследваме всички елементи на масива.

В тялото на функцията ще дефинираме две променливи — за минималната стойност и брояча. Най-голямата възможна стойност на брояча е равна на дължината на масива. Следователно типът на брояча ще е типа на променливата `size`, чиято стойност е дължината на масива.

```
int minVal = ?;    // С какво да я инициализираме?
int occurs = ?;    // С какво да я инициализираме?
```

`minVal` може да се инициализира с първия елемент на масива. С него се сравняват останалите елементи. Друга възможност е `minVal` да се инициализира с най-голямото положително число. `occurs` се инициализира с 0.

Да разгледаме възможните случаи:

- Ако елементите на масива са подредени в нарастващ ред, първият елемент е минимален. Необходимо е само едно присвояване. Това е най-добрият случай.
- Ако елементите на масива са подредени в намаляващ ред, последният елемент е минимален. Необходими са n на брой присвоявания. Това е най-лошият случай.
- Ако масивът не е сортиран, най-често минималният елемент се намира към средата на масива. Тогава са необходими средно $n/2$ на брой присвоявания.

Във всички случаи, ако инициализираме `minVal` със стойността на първия елемент, спестяваме едно присвояване. Ще се изследват само $n-1$ елемента. Като използваме направените бележки да реализираме функцията.

Необходими са два условни оператора:

```
if ( minVal > ia[ i ] ) ...    // актуализиране на minVal
if ( minVal == ia[ i ] ) ...   // ново срещане на minVal
```

При използване на условен оператор често се греша, ако не се използва съставен оператор, когато са необходими няколко оператора. Грешката е трудна за откриване, защото текстът на програмата изглежда коректен. Например:

```
if ( minVal > ia[ i ] )
    minVal = ia[ i ];
    occurs = 1;           // не е част от условния оператор
```

Независимо от това, че операторът

```
occurs = 1;
```

е със същия отстъп навътре, както оператора

```
minVal = ia[ i ];
```

от което личи намерението на програмиста и двата да се изпълнят при вярно условие `if`, на практика се изпълнява само първият оператор. Същият условен оператор трябва да се запише така:

```
if ( minVal > ia[ i ] ) {  
    minVal = ia[ i ];  
    occurs = 1;  
}
```

Вторият условен оператор има вида:

```
if ( minVal == ia[ i ] ) ++occurs;
```

Забележете, че редът на условните оператори е от значение. Ако те са подредени както по-долу, броят на срещанията на `minVal` ще е с 1 по-голям от действителния.

```
if ( minVal > ia[ i ] ) {  
    minVal = ia[ i ];  
    occurs = 1;  
}
```

```
// грешка, ако променливата minVal току-що е получила стойността на ia[ i ]  
if ( minVal == ia[ i ] ) ++occurs;
```

Условията в двата оператора `if` са за един и същ елемент на масива. Това е не само опасно, но и излишно. Един и същ елемент не може да бъде едновременно по-малък и равен на `minVal`. Ако едното условие е вярно, другото може да се игнорира. В такива случаи се използва друг вид на оператора `if` — `if` с клауза `else`.

Синтаксисът на оператора `if-else` е следният:

```
if ( израз )  
    оператор_1;  
else  
    оператор_2;
```

Ако стойността на израза е различна от нула, условието се счита за вярно и се изпълнява **оператор_1**. В противен случай се изпълнява **оператор_2**. Забележете, че ако **оператор_1** не е съставен, той трябва да завършва с `;`. Програмисти с опит в програмирането на Паскал или Ада често забравят тази особеност, защото в тези езици подобно нещо се счита за грешка. Например:

```
if ( minVal == ia[ i ] )  
    ++occurs;    // изисква задължително ";"  
else  
    if ( minVal > ia[ i ] ) {  
        minVal = ia[ i ];  
        occurs = 1;  
    }
```

В примера **оператор_2** е също условен оператор. Ако minVal е по-малко от съответния елемент, няма да последват никакви действия.

В следващия пример се изпълнява винаги само един оператор:

```
if ( minVal < ia[ i ] )
    ; // празен оператор
else
if ( minVal > ia[ i ] ) {
    minVal = ia[ i ];
    occurs = 1;
}
else // minVal == ia[ i ]
    ++occurs;
```

Операторът if-else е потенциален източник на двусмислие. Ако в един оператор има повече оператори if, отколкото клаузи else, възниква въпросът, за кой от операторите if се отнася съответната клауза else. Ето един пример:

```
if ( minVal <= ia[ i ] )
    if ( minVal == ia[ i ] )
        ++occurs;
else {
    minVal = ia[ i ];
    occurs = 1;
}
```

Отстъпите свидетелстват за това, че програмистът отнася клаузата else към първия условен оператор. Но в C++ всяка клауза else е свързана с последния оператор if, който не е свързан с друга клауза else.

В нашия случай действителният резултат е следния:

```
if ( minVal <= ia[ i ] )
{ // действителен ефект
    if ( minVal == ia[ i ] )
        ++occurs;
    else {
        minVal = ia[ i ];
        occurs = 1;
    }
}
```

За да отменим горното правило можем да превърнем последния оператор if в съставен оператор:

```
ff( const char *st ) {
{ // отменяме правилото
    if ( minVal == ia[ i ] ) ++occurs;
}
else {
    minVal = ia[ i ];
    occurs = 1;
}
```

При съмнение за колизия е добре да се използват съставни оператори, както е в посочения пример.

В раздел 1.8 въведохме клас `IntArray`. По-долу е описана функцията `min()`, член на този клас:

```
#include "IntArray.h"

IntArray::min( int &occurs ) {
    int minVal = ia[ 0 ];
    occurs = 1;

    for ( int i = 1; i < size; ++i ) {
        if ( minVal == ia[ i ] ) ++occurs;
        else
            if ( minVal > ia[ i ] ) {
                minVal = ia[ i ];
                occurs = 1;
            }
    }
}
```

За да може `min()` да върне стойността на `minVal` и броя на срещанията на тази стойност, ще използваме аргумент от тип `reference`. Когато формален параметър на функция е от тип `reference`, функцията получава адреса на действителния параметър. Този начин за предаване на параметри се нарича предаване по адрес. Чрез него промените на формалния параметър в тялото на функция са промени и на съответния фактически параметър. Следва един пример за използване на `min()`:

```
#include <stream.h>
#include "IntArray.h"

IntArray myArray;

main() {
    // масив с подредени в намаляващ ред елементи
    for ( int i = myArray.getSize()-2; i = 0; --i ) myArray[ i ] = i;

    // поява на втори елемент с най-малката стойност 0
    myArray[ myArray.getSize()-1 ] = 0;

    int number = 0;
    int low = myArray.min( number );
    cout    << "\nlow: " << low
           << "number: " << number << "\n";
}
```

След компилиране и изпълнение на програмата се получава следния резултат:

```
low: 0 number: 2
```

Раздел 3.7 разглежда тип `reference` и механизма за предаване на параметър по адрес.

Упражнение 2-13. Променете декларацията на `occurs` в списъка от аргументи на `min()`, така че да не е от тип `reference` и изпълнете отново програмата.

2.14 Оператор switch

Многократното влагане на оператори if-else често изисква корекции, защото действителният ефект е различен от очаквания. Съществува вероятност нежеланите свързвания else-if да останат незабелязани. Модифицирането на операторите е трудно. Един алтернативен начин за избор от множество възможности в C++ предлага оператор switch.

Да разгледаме един пример. Трябва да преброим срещанията на всяка от петте гласни в произволен текст.

Ще използваме следния алгоритъм:

- Четем символ по символ от текста до достигане на неговия край.
- Сравняваме всеки прочетен символ с гласните.
- Ако прочетеният символ е гласна, увеличаваме брояча, който съответства на тази гласна.
- Отпечатваме резултатите.

В реализацията се използва един оператор switch. За подобряване на читаемостта на програмата се използва изброимия тип Vowels. Ето и самата програма:

```
#include <stream.h>

enum Vowels { a='a', e='e', i='i', o='o', u='u' };

main()
{
    char ch;
    int aCnt=0, eCnt=0, iCnt=0, oCnt=0, uCnt=0;

    while ( cin >> ch )
        switch ( ch ) {
            case a:
                ++aCnt;
                break;
            case e:
                ++eCnt;
                break;
            case i:
                ++iCnt;
                break;
            case o:
                ++oCnt;
                break;
            case u:
                ++uCnt;
                break;
        }; // край на switch(ch)

    cout << "aCnt: \t" << aCnt << "\n";
    cout << "eCnt: \t" << eCnt << "\n";
    cout << "iCnt: \t" << iCnt << "\n";
    cout << "oCnt: \t" << oCnt << "\n";
    cout << "uCnt: \t" << uCnt << "\n";
}
```

Тази програма има един недостатък. Ще се справи ли тя със следния текст?

U и I няма да бъдат разпознати като гласни. Програмата не разпознава гласните, записани с главни букви. Преди да отстраним допуснатата неточност, да разгледаме особеностите на оператора switch.

Стойността след ключовата дума case се нарича етикет case (case label). След него задължително следва ":". Всеки етикет case е израз, чиято стойност е цяло число. Два етикета не бива да имат еднаква стойност. В противен случай компилаторът сигнализира за грешка.

При изпълнение на switch се изчислява изразът (в примера ch) и получената стойност се сравнява последователно с етикетите case. При съвпадение се изпълняват операторите след съответния етикет. В противен случай не следват никакви действия от switch.

Обикновено се смята, че се изпълняват само операторите за съответния етикет case. В действителност се изпълняват всички оператори между първия оператор за съответния случай и последния оператор в switch. По-долу е даден незначително модифициран вариант на оператора switch от предишната програма:

```
switch ( ch ) {
    case a: ++aCnt;
    case e: ++eCnt;
    case i: ++iCnt;
    case o: ++oCnt;
    case u: ++uCnt;
}; // край на switch(ch)
cout << ch << "\n"; // за илюстрация
```

Ако ch има стойност 'i', ще се изпълнят всички оператори след етикета i. iCnt ще се увеличи с единица, но действието няма да се ограничи с това. Ще се изпълнят всички оператори до края на switch. С единица ще се увеличат още oCnt и uCnt. Ако следващия път ch има стойност 'e', с единица ще се увеличат eCnt, iCnt, oCnt и uCnt.

Съществува възможност програмистът да съобщи на компилатора, че иска да се изпълнят само операторите за съответния етикет. За целта се използва оператор break. Обикновено break е последният оператор за съответния етикет case. break предизвиква излизане от switch. Управлението се предава на първия оператор след switch. В примера това е операторът:

```
cout << ch << "\n"; // за илюстрация
```

Добре е да се постави коментар за съзнателно пропуснат break. Коментар не е необходим, ако контекстът е достатъчен за изясняване на смисъла.

Кога програмистът съзнателно пропуска break? Първо, когато за няколко стойности от множеството трябва да се извършат еднакви действия. Всяка отделна стойност, обаче, трябва да представлява етикет. Да припомним, че нашата програма не може да обработва гласни, записани с главни букви. По-долу този недостатък е отстранен, а реализацията показва случаи, когато break е пропуснат.

```
switch ( ch ) {
    case A:
```

```

case a:
    ++aCnt;
    break;
// ...
case U:
case u:
    ++uCnt;
    break;
};

```

Оператор switch позволява да се реализира безусловна else клауза чрез етикета default. Ако стойността на израза в switch не съвпада с никой от етикетите case и има етикет default, ще се изпълнят операторите след default. Нека добавим етикет default към оператора switch от горния пример. Така ще изчисляваме и броя на съгласните в текста.

```

#include <ctype.h>
//...
switch ( ch ) {
    case A:
    case a:
        ++aCnt;
        break;
    // ...
    case U:
    case u:
        ++uCnt;
        break;
    default:
        if isalpha( ch ) ++conCnt;
        break;
};

```

isalpha() е стандартна функция. Тя връща стойност истина, ако нейният аргумент е буква от английската азбука. За да я използваме, трябва да включим системния заглавен файл ctype.h.

Макар че break не е задължителен за последния етикет case, за сигурност е добре да се използва. Ако допълнително добавим нов етикет в края на switch, липсата на break ще се окаже съществена.

Упражнение 2-14. Модифицирайте програмата, за да преброява и интервалите в текста.

Упражнение 2-15. Модифицирайте програмата, за да преброява и срещанията на следните последователности от символи: ff, fl и fi.

Оператори за цикъл

В програмите често се налага да се изпълнят едни и същи оператори над множество обекти. Да разгледаме програма, написана за банка, която обработва чекове. За всеки чек се изпълнява следната последователност от действия:

- Чекът се прочита.
- Проверява се съществува ли такъв номер на банкова сметка.
- Проверява се дали средствата в съответната банкова сметка са достатъчни, за да покрият сумата от чека.

- Нанасят се необходимите корекции в съответната сметка.
- Операцията се регистрира.

Едва ли е разумно тази последователност от действия да се повтаря в програмата за всеки чек. Езикът C++ позволява да се използват оператори за цикъл. Чрез тях се изпълнява многократно съвкупност от оператори.

Всеки цикъл има тяло. То се изпълнява докато е удовлетворено някакво условие. В програмата за банката условието е:

```
while there exist a check to be processed
```

Според това условие цикълът ще завърши, когато се обработи и последният чек.

В C++ съществуват три вида цикли: while, do и for. Основната разлика между тях е в начина за напускане на цикъла. И за трите оператора съответното условие е вярно, ако стойността му е ненулева. Нулева стойност означава невярно условие.

2.15 Оператор while

Оператор while има следния синтаксис:

```
while ( израз ) оператор;
```

Неговото действие е следното:

1. Изчислява се изразът.
2. Ако стойността на израза е различна от нула, се изпълнява тялото на while. Тялото на цикъла се състои от един съставен или прост оператор. След това се преминава към точка 1.

Ако при първото изчисление на израза се получи нулева стойност, тялото на while не се изпълнява. Управлението се предава на първия оператор след цикъла. В примера от предишния раздел символите се четат един по един до края на текста. Оператор while е подходящ за организиране на четенето и затова го използвахме в програмата:

```
char ch;
while (cin > ch )
    switch ( ch ) {
        ...
    }
```

Тъй като оператор switch е прост оператор, той може да не се загради във фигурни скоби.

Оператор while е подходящ за работа с низове. Например:

```
ff(const char *st) {
    int len = 0;
    const char *tp = st;
    // изчисляване на дължината на st
    while ( *tp++ ) ++len;
```

```

// копиране на st
char *s = new char[ len +1 ];
while ( *s++ = *st++ )
    ; // празен оператор
// ... остатъка на функцията
}

```

Упражнение 2-16. Създайте функция, която определя дали два низа са равни.

Упражнение 2-17. Създайте функция, която връща броя на срещанията на даден символ в низ.

Упражнение 2-18. Създайте функция, която определя дали даден низ се среща в друг низ.

2.16 Оператор for

Оператор for се използва най-често за обхождане на структура с фиксирана дължина, например масив. Ето неговият синтаксис:

```
for ( инициализиращ_оператор; израз_1; израз_2 ) оператор;
```

Инициализиращ_оператор може да бъде декларация или израз. В общия случай той инициализира една или повече променливи, но може и да липсва. Следват примери за правилно зададени инициализиращи оператори:

```

for ( int i = 0; ...
for ( ; /* липсва инициализиращ_оператор */ ...
for ( i = 0; ...
for ( int lo = 0, hi = max, mid = max/2; ...

for ( char *ptr = getStr(); ...
for ( i = 0, ptr = buf, dbl = 0.0; ...

```

Израз_1 е условие, което се проверява при всяко завъртане на цикъла. Ако неговата стойност е различна от нула, се изпълнява тялото на for — един съставен или прост оператор. Ако при първото изчисление на **израз_1** се получи нулева стойност, тялото на цикъла не се изпълнява. Ето няколко примера:

```

( ...; index < arraySize; ... )
( ...; ptr; ... )
( ...; *st1++ = *st2++; ... )
( ...; ch = getNextChar(); ... )

```

Израз_2 се изчислява след всяко завъртане на for. В общия случай той променя стойностите на променливите, инициализирани в инициализиращия оператор. Ако при първото изчисление на **израз_1**, неговата стойност е нула, **израз_2** не се изчислява. По-долу са дадени няколко примера за вида на **израз_2**:

```

( ...; ...; ++i )
( ...; ...; ptr = ptr—>next )
( ...; ...; ++i, --j, ++cnt )

```

```
( ...; ...; )    // израз_2 липсва
```

Да разгледаме следния цикъл for:

```
const int sz = 24;
int ia[ sz ];
for ( int i = 0; i < sz; ++i ) ia[ i ] = i;
```

Неговото действие е следното:

1. Инициализацият оператор се изчислява само веднъж — в началото на цикъла. В случая се дефинира *i* и се инициализира с нула.
2. Изчислява се израз_1. Ако неговата стойност не е нула, се изпълнява тялото на цикъла. В противен случай действието на for се прекратява. Ако първата изчислена стойност на израз_1 е нула, тялото на цикъла не се изпълнява. В примера стойността на *i* се сравнява със стойността на *sz*. Докато *i* е по-малко от *sz* се изпълнява операторът:

```
    ia[ i ] = i;
```

3. Изчислява се израз_2. Обикновено той променя стойностите на променливите, инициализирани чрез инициализация оператор. В примера *i* се увеличава с единица.

Това е едно цялостно завъртане на цикъл for. След стъпка 3 следва отново стъпка 2. Същите действия се реализират от следния оператор while :

```
инициализиращ_оператор;
while ( израз_1 )
{
    оператор;
    израз_2;
}
```

Упражнение 2-19. Създайте функция, която проверява дали два масива са еднакви. Какво означава два масива да бъдат еднакви?

Упражнение 2-20. Създайте функция, която търси елемент с определена стойност в масив. Ако такъв елемент съществува, функцията връща неговия индекс. Какво трябва да върне функцията, ако няма такъв елемент?

2.17 Оператор do

Нека имаме за задача да създадем интерактивна програма, която превръща милите в километри. Схематично програмата ще изглежда така:

```
int more = 1;          // стойност, която гарантира, че цикълът ще се завърти поне веднъж
while ( more )
{
    val = getValue();
    val = convertValue(val);
    printValue(val);
}
```

```

    more = doMore();
}

```

В този пример променливата, която управлява цикъла, се настройва изкуствено, за да може цикълът да стартира. Знаем, че ако първата изчислена стойност на условието в `while` и `for` е нула, цикълът не предизвиква никакви действия. Следователно, програмистът трябва сам да се грижи за стартиране на цикъла. Цикъл `do` гарантира, че неговото тяло ще се изпълни поне веднъж. Ето как изглежда синтаксисът на този оператор:

```

do
    оператор;
while ( израз );

```

Тялото на цикъла — един съставен или прост оператор — се изпълнява преди изчисляване на израза. Ако стойността на израза е нула, цикълът завършва. Горната програма вече може да се реализира така:

```

int more;
do {
    val = getValue();
    val = convertValue(val);
    printValue(val);
    more = doMore(); }
while ( more );

```

Оператори за преход

Операторите за преход предават безусловно управлението в някаква точка на програмата. В C++ такива операторите са `break`, `continue` и `goto`. Следващите раздели разглеждат тези оператори.

2.18 Оператор `break`

Оператор `break` прекъсва изпълнението на `while`, `for`, `do` и `switch`. Ако няколко от изброените по-горе оператори са вложени един в друг, ще се прекрати изпълнението на този, който съдържа `break` непосредствено в тялото си. Изпълнението продължава от първия оператор след прекъснатия. Да разгледаме един пример. Ще създадем функция-член на клас `IntArray`, която проверява дали в масива има елемент с конкретна стойност. Ако открие такъв елемент, връща неговия индекс; иначе връща `-1`. Ето самата функция:

```

#include "IntArray.h"

IntArray::search( int val ) {
    int loc = -1;
    for ( int i = 0; i < size; ++i )
        if ( val == ia[ i ] ) {
            loc = i;
            break;
        }
    return loc;
}

```

Ако елементът е открит, продължаването на цикъла е излишно. `break` прекратява изпълнението на `for`. Тогава се изпълнява оператор `return`, следващ непосредствено след `for`.

2.19 Оператор `continue`

Оператор `continue` прекратява текущото изпълнение (итерация) на `while`, `for` или `do`. При влагане `continue` прекратява текущата итерация на оператора, в чието тяло непосредствено се намира. Ако `continue` е в тялото на `while` или `do`, изпълнението продължава с изчисление на израза-условие. Ако операторът е `for`, се изчислява израз_2. За разлика от `break`, който прекратява целия цикъл, `continue` прекратява само текущата итерация. Да разгледаме един пример:

```
while ( cin >> inBuf ) {  
    if ( inBuf[0] != ' ' )  
        continue; // прекратява текущата итерация  
    // ... обработка на думата ...  
}
```

Този програмен фрагмент чете по една дума от текстов файл и ако думата започва със знак за подчертаване я обработва. В противен случай текущата итерация завършва.

2.20 Оператор `goto`

Ако `break` и `continue` не съществуваха, щеше да има нужда от оператор, чрез който да се излиза от тялото на `for`, `while`, `do` и `switch`. Оператор `goto` реализира тази необходимост, но той се използва рядко в програмирането на C++. `goto` предава управлението в точка от програмата безусловно. Точката, в която се предава управлението, се означава с етикет. Този етикет и `goto` трябва да са в една и съща функция.

Синтаксисът на оператор `goto` е следния:

```
goto етикет;
```

Като етикет може да използва произволен идентификатор. Етикетът трябва да се среща само веднъж в съответната функция и да завършва с ":". Той не може да стои непосредствено пред дясна затваряща скоба "}". Това ограничение се преодолява, като след етикета се постави празен оператор. Например:

```
end: ; // празен оператор  
}
```

Оператор `goto` не може да прескача дефиниция на променлива, която се инициализира явно или неявно, освен ако дефиницията се намира в блок и целият блок се прескочи. В следващата програма е допусната грешка:

```
#include "IntArray.h"  
extern int getValue();  
extern void processArray( IntArray& )  
  
main() {  
    int sz = getValue();  
    if ( sz <= 0)
```

```
        goto end;

    IntArray myArray(sz);
    processArray( myArray );

end: ;
}
```

myArray се инициализира неявно от конструктора на клас IntArray.

Упражнение 2-21. Коригирайте главната програма в горния пример, за да отстраните допуснатата грешка.

ГЛАВА 3

Всяка функция може да се разглежда като дефинирана от програмиста операция. Операндите на функцията се наричат аргументи и се въвеждат чрез списък на аргументите (argument list). В този списък те са отделени със запетаи и заградени в кръгли скоби. Всяка функция се характеризира с тип на резултата (return type). Функция, която не връща резултат, има тип на резултата void. Действията на функцията се описват в нейното тяло. Тялото на всяка функция е заградено във фигурни скоби "{}" и понякога се нарича блок. Ето няколко примера за функции:

```
inline int abs( int i )
{ // връща абсолютната стойност на i
  return( i < 0 ? -i : i );
}

inline int min( int v1, int v2 )
{ // връща по-малкото от двете числа
  return( v1 <= v2 ? v1 : v2 );
}

gcd( int v1, int v2 )
{ // връща най-големия общ делител на две числа
  int temp;
  while ( v2 ) {
    temp = v2;
    v2 = v1 % v2;
    v1 = temp;
  }
  return v1;
}
```

За да се изпълни дадена функция, трябва да се приложи операцията извикване на функция, означена с "()". Ако функцията има аргументи, трябва да и бъдат подадени т.нар. фактически параметри (actual arguments). Те се предават, като се поставят между лявата и дясна скоба на операцията "()". Всеки аргумент се отделя със запетая. В следващия пример main() извиква abs() два пъти, а min() и gcd() по веднъж. Програмният текст се намира във файл main.C.

```
#include <stream.h>
#include "localMath.h"

main() {
  int i, j;
  // получаване на стойности от стандартния вход
  cout << "Стойност: "; cin >> i;
  cout << "Стойност: "; cin >> j;

  cout << "\nМинимум: " << min( i, j ) << "\n";
  i = abs( i ); j = abs( j );
  cout << "НОД: " << gcd( i, j ) << "\n";
}
```

При извикване на функция, може да се случи едно от следните две неща. Ако функцията се декларира като inline, при компилация нейното тяло се копира в точката на

извикване. В противен случай се осъществява обръщение към функцията при изпълнение на програмата. Това означава, че активната в момента функция се спира временно, изпълнява се извиканата функция и след това се продължава с изпълнението на прекъснатата функция. Продължава се от точката непосредствено след извикването. За да се реализира обръщение към функцията се използва програмния стек (run-time stack).

Преди да се използва, всяка функция трябва да се декларира. В противен случай компилаторът ще сигнализира за грешка. Разбира се дефиницията на функция е и нейна декларация. Всяка функция може да се дефинира само веднъж в програмата. Обикновено дефиницията е или в текстовия файл, в който функцията се използва, или в друг текстов файл, съдържащ функции, свързани с тази. Често функциите се използват във файлове, в които не са дефинирани. Това означава, че трябва да има начин за деклариране на функции, когато това е необходимо.

Декларацията на функция включва тип на резултата, име на функцията и списък на аргументите и. Тези три елемента взети заедно се наричат прототип на функцията (function prototype). Прототипът на дадена функция може да се среща многократно в един файл без това да предизвиква грешка при компилация.

За да компилираме main.C, трябва да декларираме abs(), min() и gcd(). В противен случай всяко тяхно извикване ще предизвиква грешка при компилация. Ето декларациите на трите функции:

```
int abs( int );
int min( int, int );
int gcd( int, int );
```

Забележете, че в списъка на аргументите се посочва само техният тип.

Декларациите на трите функции и дефинициите на обявените като inline функции е добре да се запишат в заглавен файл. Този заглавен файл може да се включи във всеки друг файл. Така много файлове могат да ползват общи декларации. Това опростява и коригирането на функциите, включени в заглавния файл.

В нашия пример заглавният файл се нарича localMath.h. Той съдържа:

```
int gcd( int, int );

// дефиниции на inline функции
inline int abs( int i )      { return( i < 0 ? -i : i ); }
inline int min( int v1, int v2 ) { return( v1 < v2 ? v1 : v2 ); }
```

Функциите abs() и min() са дефинирани в заглавния файл и няма да присъстват в програмния текст. Можем да компилираме програмата:

```
$CC main.C gcd.C
```

След нейното изпълнение се получава следния резултат:

```
Стойност: 15
Стойност: 123
```

```
Минимум: 15
```

НОД: 3

3.1 Рекурсия

Функция, която вика себе си директно или посредством друга функция, се нарича рекурсивна. gcd() може да се напише и като рекурсивна функция:

```
rgcd( int v1, int v2 )  
{  
    if ( v2 == 0 ) return v1;  
    return rgcd( v2, v1%v2 );  
}
```

Във всяка рекурсивна функция трябва да присъства условие за край на рекурсията (stopping condition). В противен случай функцията ще се обръща към себе си безкрайно. В горния пример условие за край е проверката за остатък нула при деление на v1 и v2. Обръщението

```
rgcd( 15, 123 )
```

връща резултат 3. В таблица 3.1 е проследено изпълнението на rgcd(15, 123).

Последното обръщение rgcd(3,0) удовлетворява условието за край. То връща НОД = 3. Тази стойност става последователно резултат за всяко предишно извикване. Казва се, че тази стойност се връща на всяко по-горно ниво.

v1	v2	връщан резултат
15	123	rgcd(123, 15)
123	15	rgcd(15, 3)
15	3	rgcd(3, 0)
3	0	3

Табл. 3.1 Изпълнение на **rgcd(15, 123)**

Обикновено рекурсивна функция се изпълнява по-бавно от еквивалентната на нея нерекурсивна функция. Това се дължи на многократните обръщения към функцията и усилията за поддържане на програмния стек.

Да разгледаме още един пример. Ще създадем рекурсивна програма за изчисляване на факториел на естествено число. Под факториел се разбира произведението на всички естествени числа между 1 и даденото число. Например факториела на 5 е 120:

$$1 * 2 * 3 * 4 * 5 = 120$$

Ето рекурсивният вариант на функцията за изчисляване на факториел:

```
unsigned long factorial( int val ) {  
    if ( val > 1 )
```

```

        return val * factorial( val - 1 );
    return val;
}

```

Условието за край на рекурсията сега е проверката дали val е 1.

Упражнение 3-1. Напишете нерекурсивен вариант на функцията factorial().

Упражнение 3-2. Какво ще се получи, ако условието за край на рекурсията в горния пример е:

```
if ( val != 0 )
```

Упражнение 3-3. Защо функцията връща резултат от тип unsigned long, а аргументите и са от тип int?

3.2 Inline-функции

Един въпрос, който досега не сме обсъждали, е защо дефинирахме функцията min(). Явно не за да намалим повторенията на еднакви програмни части. В действителност извикването

```
min( i, j );
```

изисква един символ в повече, отколкото директното използване на аритметичен if:

```
i < j ? i : j;
```

От създаването на min() извличаме следните предимства:

- В общия случай по-лесно се прочита обръщение към min(), отколкото съответния аритметичен if, особено ако i и j са сложни изрази.
- По-лесно се коригира програмният текст в една функция, отколкото навсякъде, където се използва. Ако например решим да проверяваме дали $i \leq j$ откриването и корекцията на всички места в програмата е скучна и предразполагаща към грешки работа.
- При многократно изписване на един и същ програмен текст може да допуснем несъзнателна грешка. Чрез извикване на min(), си осигуряваме, че ще се изпълни това, което сме имали в предвид.
- Ако използваме функцията min(), ние осигуряваме проверка за съответствие на типовете на формалните и фактически параметри. За несъответствие се сигнализира по време на компилация.
- Тази функция може да се използва лесно за други приложения.

Използването на min() притежава един сериозен недостатък. То намалява бързодействието на програмата, защото при всяко извикване на min() аргументите на функцията се копират, регистрите се запазват в стека и едва тогава се стартира самата функция. Директното изписване на кода ускорява изпълнението.

Поставеният проблем се решава чрез дефиниране на `min()` като `inline`-функция. Когато функцията е `inline`, при компилация всяко обръщение към нея се замества с копие на тялото и. От казаното следва, че

```
int minVal2 = min( i, j );
```

по време на компилация се превръща в

```
int minVal2 = ( i <= j ) ? i : j;
```

Следователно, при изпълнение на програмата няма да има обръщение към `min()`, а изпълнение на аритметичния `if`.

`min()` става `inline`-функция, ако при дефинирането и използваме ключовата дума `inline`. Забележете, че спецификацията `inline` е само препоръка за компилатора. Една рекурсивна функция, например, не може да се разгърне изцяло в точката на обръщение. Това може да стане само за първото и извикване. Вероятно функция от 1200 реда също няма да се копира в точката на извикване. В общия случай механизмът `inline` се прилага за малки функции (до няколко реда), използвани достатъчно често.

3.3 Строг контрол на типовете

Функцията `gcd()` изисква два аргумента от тип `int`. Какво ще се случи, ако фактическите параметри са от тип `float` или `char*`? А какво ще се случи, ако предадем един или повече от два аргумента?

Основната операция в `gcd()` е операцията за изчисляване на остатък от делението на две числа. Знаем, че операндите на тази операция трябва да са цели числа. Следователно обръщението

```
gcd( 3.14, 6.29 )
```

вероятно ще предизвика грешка при изпълнение на програмата. По-лошо е, ако програмата изчисли някаква стойност, която няма смисъл за нас. Това е по-лошият случай, защото е трудно да се разбере каква е причината за безсмисления резултат. Необходими са големи усилия за откриване и отстраняване на допуснатата грешка. Какъв е според вас резултатът от следните две обръщения:

```
gcd( "hello", "world" );  
gcd( 24312 );
```

Съобщението за грешка е единственият приемлив резултат от компилирането на горните обръщения. Съобщенията изглеждат приблизително така:

```
// gcd( "hello", "world" );  
error: invalid argument types (char*, char*) — expecting (int, int )  
// gcd( 24312 );  
error: missing value for argument two
```

А какво ще се случи, ако gcd() се извика с параметри от тип double? Съобщение за грешка от смесване на типове би било правилна, но строга мярка. В действителност се извършва автоматично преобразуване на типове. Тъй като double се превръща в int чрез отрязване на значещи цифри, ще се изведе предупреждение. Обръщението към gcd() се преобразува в

```
gcd( 3, 6 );
```

В такъв случай резултатът ще бъде 3.

C++ е език със строг контрол на типовете. При компилация се проверява за съответствие на типовете, както на списъка от аргументи, така и на резултата. Ако има смесване на типове, се прави опит за автоматично преобразуване. Ако преобразуване не може да се направи или броят на аргументите не съответства, се издава съобщение за грешка.

Сега става ясно, защо преди да се използва функция, тя трябва да се декларира. Нейният прототип осигурява информация, която се използва от компилатора за контрол на типовете.

3.4 Резултат на функция

Една функция може да върне резултат от стандартен, произведен или дефиниран от потребителя тип. Ето няколко примера за допустими типове на резултата:

```
double sqrt( double );
char *strcpy( char*, const char* );
IntArray &IntArray::qsort();
TreeNode *TreeNode::inOrder();
void error( const char* ... );
```

Масиви и функции не могат да се използват като тип на резултата. Вместо тях се използват указател към масив и указател към функция. void показва, че функцията не връща резултат. Функция без явен тип на резултата, връща резултат от тип int.

Следващите декларации на isEqual() са еквивалентни. В двата случая функцията връща целочислена стойност:

```
int isEqual( long*, long* );
isEqual( long*, long* );
```

Оператор return прекратява изпълнението на функция. Управлението се предава на извикващата функция. return има две форми:

```
return;
return израз;
```

За функции, които не връщат резултат, оператор return не е необходим. В тях той може да се използва за преждевременно прекратяване на функцията. Тази употреба на return прилича на използването на break в операторите за цикъл. След изпълнение на последния оператор от функцията, се изпълнява неявно и оператора return. Например:

```

void dCopy( double *src, double *dst, int sz )
{ // копира масива src в масива dst
  // за простота предполагаме, че масивите са с еднаква дължина

  // ако някой от масивите е празен
  if ( src == 0 || dst == 0 ) return;

  if ( src == dst ) return; // копиране не е необходимо

  if ( sz <= 0 ) return;      // няма елементи за копиране

  // копиране
  for ( int i = 0; i < sz; ++i ) dst[ i ] = src[ i ];
  // явен оператор return не е необходим
}

```

Вторият вид на оператора return връща резултата от функцията. Изразът може да е произволен и да съдържа дори обръщение към функция. Например, във функцията factorial() се използва следният оператор return:

```

return val * factorial( val - 1 );

```

Функция, която не връща резултат и не е декларирана като void, не предизвиква грешка при компилация. В общия случай се получава само предупреждение. main() е типичен пример за функция, за която оператор return често се забравя. Във всяка точка на прекъсване на функцията трябва да стои съответният оператор return. Например:

```

enum Boolean { FALSE, TRUE };
Boolean isEqual( char *s1, char *s2 )
{ // ако s1 и s2 са еднакви низове, връща TRUE, иначе връща FALSE

  // ако някой от низовете е празен, те не са равни
  if ( s1 == 0 || s2 == 0 ) return FALSE;

  if ( s1 == s2 ) // имаме един и същ низ
    return TRUE;

  while ( *s1 == *s2++ )
    if ( *s1++ == '\0' ) return TRUE;

  // ако стигнем до тук, низовете не са равни
  return FALSE;
}

```

По принцип всяка функция връща само една стойност. Съществуват начини за връщане на няколко стойности:

- Всяка променлива, дефинирана извън тялото на една функция, е глобална за тази функция. Ако в тялото на функцията глобалната променлива получи някаква стойност, тази стойност се връща в извикващата функция. Предимството на този начин е в неговата простота. Негов недостатък е това, че по обръщението към функцията не може да се разбере дали се променя глобална променлива и коя от тях. Това затруднява четимостта на програмата и нейното

коригиране. Промяната на глобална променлива в тялото на функция е известно като страничен ефект (side effect).

- Връщане на резултат, който е съвкупност от няколко стойности. В общия случай за предпочитане е да се използват класове, а не масиви. Ако стойностите са в масив, трябва да се върне указател към масива. Ако се използва клас, може да се върне обект от класа или указател към този обект. При класове типът на резултата може да е и reference.
- Формалните аргументи могат да бъдат от тип reference или указатели. Така получаваме достъп до адресите на фактическите параметри и можем директно да променяме техните стойности. Този метод е разгледан подробно в раздел 3.6.

3.5 Списък от аргументи

Функциите обменят информация по два начина — чрез формалните си параметри или чрез глобални променливи.

Всяка глобална променлива е достъпна за функциите, разположени след нейната обява. Глобалните променливи са удобен начин за комуникация между частите на програмата, но притежават и редица неудобства:

- Функциите, които използват глобални променливи, зависят от тези променливи. Това значително затруднява използването на същите функции в други приложения.
- При модифициране на програмата, силната зависимост от глобалните променливи увеличава вероятността за грешки. Нещо повече: за да се внесат локални промени, трябва да се разгледа цялата програма.
- Ако глобална променлива получи неправилна стойност, за откриване на допуснатата грешка е необходимо цялостно изследване на програмата.
- Използването на глобални променливи затруднява правия ход на рекурсията.

Списъкът от аргументи предоставя алтернативен начин за комуникация между функциите в една програма. Този списък и типът на резултата представляват т.нар. открит интерфейс (public interface) на функцията. Програма, която познава само открития интерфейс на дадена функция, не трябва да се променя при промяна на функцията. Такава функция може да се използва без ограничения в различни приложения.

Пропускът на аргумент или предаването на аргумент с несъответстващ тип са общи източници на грешки. С въвеждане на строг контрол на типовете в C++, тези грешки почти винаги се улавят от компилатора.

С увеличаване броя на аргументите нараства и възможността за грешки. Смята се, че девет е максималният приемлив брой аргументи. Ако някоя функция изисква повече аргументи, сигурно тя извършва твърде много действия. Опитайте се да я разделите на две или повече тясно специализирани функции.

Друга възможност за намаляване броя на аргументите е използването на класове. Тогава съвкупност от параметри може да се предаде чрез един обект от класа. Този начин има следните предимства:

1. Сложността на списъка от аргументи значително намалява.
2. Проверката за коректност на стойностите на аргумента може да се възложи на членовете на класа, вместо на функцията. Това намалява размера на функцията и подобрява нейната прегледност.

Синтаксис на списъка от аргументи

Списъкът от аргументи на дадена функция може да бъде празен, ако функцията няма аргументи. Функция без аргументи се декларира по един от следните два еквивалентни начина:³

```
// еквивалентни декларации
int fork();
int fork(void);
```

Списъкът от аргументи се нарича още сигнатура, защото често служи за различаване на функциите. Името и сигнатурата на една функция я идентифицират еднозначно. Раздел 4.3 разглежда дефинирането на функции с еднакви имена.

Сигнатурата на всяка функция включва типовете на аргументите, разделени със запетаи. Имената на аргументите могат да липсват. Типът на всеки аргумент трябва да се посочи отделно. Например:

```
min( int v1, v2 );      // грешка
min( int v1, int v2 );  // ок
```

Аргументите в сигнатурата трябва да имат различни имена. Тези имена осигуряват достъп до аргументите в тялото на функцията. Затова в декларацията имената са излишни, но могат да се използват за документиране. Например:

```
print( int *array, int size );
```

Езикът позволява използването на различни имена на аргументите в дефиницията и декларациите на една функция. Това не се препоръчва, за да не се загуби прегледността на програмата.

Специалната сигнатура "... " (Ellipses)

Има случаи, когато типът и броят на аргументите на функция не са известни. В тези случаи се използва специалната сигнатура "...".

При тази сигнатура не се извършва проверка за съответствие на типовете. Тя указва на компилатора, че следват нула или повече аргументи и техните типове са неизвестни. Сигнатурата може да се използва в една от следните две форми:

```
foo( arg_list, ... );
foo( ... );
```

В първата форма запетаята, която отделя известните аргументи, не е задължителна.⁴

³ Borland C++ 2.0 изисква задължително ключовата дума void. — бел. прев.

⁴ За компилатора на Borland C++ 2.0 запетаята е задължителна. — бел. прев.

Стандартната функция `printf()` е пример за функция, която използва сигнатурата "...". Първият аргумент на `printf()` е низ. От него зависи дали ще има други аргументи. В този низ метасимволите след символа "%" показват, че има и други аргументи. Например,

```
printf( "hello, world\n" );
```

изисква само един аргумент, докато

```
printf( "hello, %s\n", userName );
```

изисква два аргумента. Символът "%" показва, че има и втори аргумент. Символът `s` означава, че този аргумент трябва да бъде низ. В C++ `printf()` се декларира така:

```
printf( const char*, ... );
```

Тази декларация изисква при всяко обръщение към `printf()`, първият аргумент да бъде от тип `char*`. След него може да следва произволен списък от аргументи.

Следващите две декларации не са еквивалентни:

```
void f();  
void f( ... );
```

В първия случай `f()` е функция без параметри, докато във втория случай `f()` е функция с неизвестен брой параметри. Обръщенията

```
f( someValue );  
f( cnt, a, b, c );
```

са допустими само във втория случай. Обръщението

```
f();
```

е възможно и в двата случая.

Сигнатура с инициализация

Стойност по подразбиране е такава стойност, която е подходяща в най-често срещаните случаи. Използването на стойности по подразбиране освобождава потребителя от необходимостта да се грижи за всичко. Например в системата UNIX текстовите файлове по подразбиране са отворени за четене и запис, а всички останали файлове — само за четене. Съществува механизъм за промяна на установения по подразбиране достъп.

Всяка функция може да определи стойности по подразбиране за един или повече свои параметри. За целта се използва сигнатура с инициализация. Нека имаме функция, която симулира екрана, като създава и инициализира двумерен масив от символи. Параметрите за дължина, ширина и фон на екрана могат да получат стойности по подразбиране:

```
char* screenInit( int height = 24, int width = 80, char background = ' ' );
```

Функция, чиито аргументи имат стойности по подразбиране, може да се извика без съответните фактически аргументи. Ако не са посочени фактически аргументи, ще се използват стойностите по подразбиране. В противен случай тези стойности не се вземат в предвид. Следващите обръщения към screenInit() са коректни:

```
char *cursor;

// еквивалентно на screenInit(24,80,' ')
cursor = screenInit();

// еквивалентно на screenInit(66,80,' ')
cursor = screenInit( 66 );

// еквивалентно на screenInit(66,256,' ')
cursor = screenInit(66, 256);
cursor = screenInit(66, 256, '#');
```

Забележете, че е невъзможно да се даде стойност за фон на екрана, без да се дадат стойности на първите два параметъра. При извикване на съответната функция позиционно се решава за кой от аргументите е зададена стойност. При създаване на функции, използващи сигнатура с инициализация, е важно как ще се подредят аргументите. Най-често променящите се аргументи трябва да стоят най-отпред. Например, за screenInit() се предполага, че стойността на height ще се определя най-често от потребителя.

Една функция може да определи стойности по подразбиране за всичките си аргументи или за част от тях. Най-десният неинициализиран аргумент трябва да получи стойност по подразбиране преди всички аргументи наляво от него. Това изискване произтича от споменатия по-горе механизъм за позиционност.

Стойността по подразбиране може да не е константен израз. За конкретен аргумент тя трябва да се определи само веднъж в даден файл. В следващия пример е допусната грешка:

```
ff( int i = 0 );           // в заглавния файл ff.h

#include "ff.h"
ff( int i = 0 ) { ... }    // грешка
```

Добре е стойностите по подразбиране да се задават в заглавния файл, а не в дефиницията на функцията. В следващите декларации на тази функция може да се определят стойности по подразбиране за някои от неинициализираните аргументи. Това е един добър начин за използване на по-обща функция в специфични приложения. Например, функцията chmod() се съдържа в библиотеката със функции за операционната система UNIX. Нейният прототип се намира в системния заглавен файл stdlib.h. и има следния вид:

```
chmod( char *filePath, int protMode );
```

където `protMode` е режимът за достъп до файла, а `filePath` е името на файла и пълен път до него. Някои приложения разрешават само четене на файлове. Те могат да предекларират `chmod()` като дадат на `protMode` стойност по подразбиране:

```
#include <stdlib.h>
chmod( char *filePath, int protMode = 0444 );
```

В заглавния файл `ff.h` е даден следният прототип на функция:

```
ff( int a, int b, int c = 0 ); // във файла ff.h
```

Как да предекларираме тази функция, така че `b` да получи стойност по подразбиране? Следващият опит е некоректен, защото `c` за втори път получава стойност по подразбиране:

```
#include "ff.h"
ff( int a, int b = 0, int c = 0 ); // грешка
```

Макар че следващият вариант също изглежда неправилен, той решава проблема.

```
#include "ff.h"
ff( int a, int b = 0, int c ); // правилно
```

В началната декларация на `ff()` `b` е най-десният неинициализиран аргумент. Следователно правилото за позиционно съпоставяне на стойностите не е нарушено. Можем да предекларираме `ff()` още веднъж:

```
#include "ff.h"
ff( int a, int b = 0, int c ); // правилно
ff( int a = 0, int b, int c ); // правилно
```

3.6 Предаване на параметри

Всяка функция има връзка с място в паметта с определена структура, известно като програмен стек. При обръщение към функция, в стека се запазват данни. Те остават там до излизане от тялото на функцията, когато стекът се изчиства автоматично.

Списъкът от аргументи описва формалните аргументи на функция. В стека се отделя място за всеки формален аргумент. Размерът на това място зависи от типа на аргумента. При обръщение към функция, в скобите стоят нейните фактически параметри. Предаването на параметри е процес на копиране на стойностите на фактическите параметри на съответните места в стека. Този начин за предаване на аргументи е известен като предаване по стойност (*pass by value*). При него функцията няма достъп до фактическите параметри, а до техните локални копия в стека. Следователно, промените на фактически параметър в тялото на функция не се отразят на стойността на оригиналната променлива и програмистът не трябва да се грижи за запазване и възстановяване на оригиналните стойности. Без този механизъм за предаване на параметри, формален аргумент, който не е деклариран като `const`, може да се промени при обръщение към функцията. Това е най-безопасният начин за предаване на параметри. Той изисква най-малко грижи, но понякога не е подходящ. Например:

- Когато се предава обект от даден клас, неговото копиране в стека изисква много време и място.
- Когато искаме да променим стойността на някой аргумент. Например, функцията swap(), трябва да промени стойностите на фактическите аргументи и да запази тези промени извън тялото си. Следващата дефиниция на swap() не реализира целта:

```
void swap( int v1, int v2 ) {
    int tmp = v2;
    v2 = v1;
    v1 = tmp;
}
```

swap() променя само локалните копия на аргументите. Следващата програма показва това:

```
#include <stream.h>
void swap( int, int );

main() {
    int i = 10;
    int j = 20;

    cout    << "Преди извикване на swap():\ti: "
            << i << "\tj: " << j << "\n";
    swap( i, j );
    cout    << "След извикване на swap():\ti: "
            << i << "\tj: " << j << "\n";
}
```

След компилиране и изпълнение на програмата получаваме:

```
Преди извикване на swap():      i: 10    j: 20
След извикване на swap():      i: 10    j: 20
```

Поставената задача може да се реши по два начина. Първият начин е формалните аргументи да бъдат указатели. Тогава swap() изглежда така:

```
void pswap( int *v1, int *v2 ) {
    int tmp = *v2;
    *v2 = *v1;
    *v1 = *tmp;
}
```

main() трябва да се промени като декларира и извиква pswap(), вместо swap(). При обръщение към pswap(), трябва да се предават адресите на двата обекта, а не обектите, както е при обръщение към swap():

```
pswap( &i, &j );
```

След компилиране и изпълнение на модифицираната програма получаваме искания резултат:

Преди извикване на swap():	i: 10	j: 20
След извикване на swap():	i: 20	j: 10

Ако формалният аргумент е указател, за да се намалят разходите на време и място за копиране в стека, а не за да се запазят промените на съответния обект, можем да декларираме указател към константа. Например:

```
void print( const BigClassObject* );
```

По този начин функцията не променя стойността на обекта, който се адресира от указателя BigClassObject.

Втори начин за получаване на желанния резултат е използването на формални аргументи от тип reference. Тогава функцията изглежда така:

```
void rswap( int &v1, int &v2 ) {  
    int tmp = v2;  
    v2 = v1;  
    v1 = tmp;  
}
```

Обръщението към rswap() в main() е като обръщението към swap():

```
rswap( i, j );
```

След компилиране и изпълнение на получената програма се вижда, че стойностите на i и j са разменени.

3.7 Аргументи от тип reference

При използване на аргумент от тип reference, функцията получава адреса на фактическия аргумент. Този механизъм за предаване на параметри е известен като предаване чрез адрес (pass by reference) и води до следните два ефекта:

1. Промяната на аргумент в тялото на функция е промяна на фактическия параметър. Това означава, че в тялото на функцията се работи с оригиналните променливи, а не с техни копия. Затова rswap() дава верен резултат.
2. Без проблеми може да се предаде голям обект от даден клас. Ако обектът се предава по стойност, той се копира в стека, при всяко извикване на функцията. Тази операция изисква много място и при рекурсивни функции води до препълване на стека. Освен това копирането забавя изпълнението.

Ако използваме аргумент от тип reference, но не искаме да променяме стойността на фактическия параметър, можем да декларираме формалния параметър като const. В следващия пример обектът x е деклариран неправилно като const:

```
class X;
```

```
int foo( X& );
int bar( const X& x ) {
    // Обектът x е деклариран като const, а
    // параметърът на foo() не може да бъде const
    return foo( x );    // грешка
}
```

x не може да се предаде като аргумент на функцията foo(), докато нейната сигнатура не се промени на const X& или X.

Ако аргументът от тип reference се отнася за стандартен тип, а съответният фактически параметър не е от същия стандартен тип, може да се получи неочакван резултат. Ще се генерира временна променлива, чийто тип съвпада с декларирания. Тази променлива получава стойността на фактическия аргумент. На функцията се предава временната променлива. Нека да извикаме rswap() с променлива от тип unsigned int:

```
int i = 10;
unsigned int ui = 20;

rswap( i, ui );
```

Ще се извършат следните преобразувания:

```
int T2 = int(ui);
rswap( i, T2 );
```

Това обръщение към rswap() връща следния неочакван резултат:

Преди извикване на swap():	i: 10	j: 20
След извикване на swap():	i: 20	j: 20

Стойността на ui остава непроменена, защото тази променлива не се предава на rswap(). За да не се получи смесване на типове, на rswap() се предава временната променлива T2. Същият резултат ще се получи, ако ui се предаде по стойност. В случаи като този, компилаторът извежда предупреждение.

Употребата на аргументи от тип reference е подходяща при работа с класове. Тогава няма проблеми със съответствието на типове. Освен това размерът на обектите е значителен. Стандартните типове данни не работят добре със този механизъм за предаване на параметри. Когато типът на фактическия аргумент не може да се определи предварително, по-добре pass by reference да не се използва.

В предишния подраздел използвахме указател като аргумент на функция, за да запазим промените в стойността на съответния фактически параметър. Какво да направим, за да променим адреса в този указател? Можем да използваме механизма pass by reference за самия указател:

```
void prswap( int *&v1, int *&v2 ) {
    int *tmp = v2;
    v2 = v1;
    v1 = tmp;
}
```

Декларацията

```
int *&p1;
```

трябва да се чете отлясно наляво, т.е. p1 е адреса на указател към обект от тип int. main() сега изглежда така:

```
#include <stream.h>
void prswap( int *&v1, int *&v2 );

main() {
    int i = 10;
    int j = 20;
    int *pi = &i;
    int *pj = &j;

    cout    << "Преди извикване на swap():\tpi: "
            << *pi << "\tpj: " << *pj << "\n";
    prswap( pi, pj );
    cout    << "След извикване на swap():\tpi: "
            << *pi << "\tpj: " << *pj << "\n";
}
```

След компилиране и изпълнение на програмата получаваме следния резултат:

```
Преди извикване на swap():      pi: 10      pj: 20
След извикване на swap():      pi: 20      pj: 10
```

Една функция може да връща резултат от тип reference или указател. Това е удобно при работа с класове, защото обектите не се копират.

Друга възможна причина за използване на тип reference като тип на резултата, е необходимостта функцията да стои отляво на оператора за присвояване. Този тип на резултата означава, че функцията връща адреса на обекта. Функция с такъв тип на резултата е операцията за индексване на клас IntArray. Тя връща резултат от тип reference, защото трябва да позволява четене и запис на стойност в елемент на масива. Следва дефиницията на функцията и един пример за нейното използване:

```
int& IntArray::operator[]( int index ) {
    return ia[ index ];
}

IntArray myArray( 8 );
myArray[ 2 ] = myArray[ 1 ] + myArray[ 0 ];
```

За справка дефиницията на клас IntArray се намира в раздел 1.8.

3.8 Масив като аргумент на функция

В C++ масивите не се предават по стойност. Вместо масива се предава указател към първия му елемент. Например,


```
void putValues( int[ 10 ] );
```

се възприема от компилатора като

```
void putValues( int* );
```

Размерът на масива няма смисъл при деклариране на формалния аргумент. Затова следващите три декларации са еквивалентни:

```
// три еквивалентни декларации на putValues()
void putValues( int* );
void putValues( int[] );
void putValues( int[ 10 ] );
```

Можем да направим следните заключения:

- putValues() работи с масива, а не с негово копие. Промените на елементи от масива се запазват след излизане от функцията. Ако искаме масивът да остане непроменен, трябва да се погрижим предаването му да става по стойност.
- Размерът на масива не може да се разбере от неговата декларация в списъка от аргументи. Следователно, putValues() не познава дължината на масива. Дължината не е известна и на компилатора. Проверка за излизане от границите на масива не се прави. Например:

```
void putValues( int[ 10 ] ); // разглежда се като int*
main() {
    int i, j[ 2 ];
    putValues( &i );          // ок: int*, но грешка при изпълнение
    putValues( j );           // ок: int*, но грешка при изпълнение
    return 0;
}
```

Компилаторът приема двете обръщения към putValues() за коректни, защото фактическият аргумент е от тип int*. Неизвестната дължина на този аргумент ще породи грешка едва при изпълнение на програмата.

Знаем, че низовете като частен случай на масивите от символи завършват със символа null. Дължината на останалите масиви трябва да се предава на съответната функция. Това се отнася и за масивите от символи, които изискват обработка на символите null, преди края на масива. За целта може да се въведе нов формален аргумент за размера на масива:

```
void putValues( int[], int size );
main() {
    int i, j[ 2 ];
    putValues( &i, 1 );
    putValues( j, 2 );
    return 0;
}
```

putValues() отпечатва елементите на масива във вида:

```
( 10 ) < 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 >
```

където 10 е дължината на масива. Следва дефиницията на putValues():

```
#include <stream.h>
const lineLength = 12; // брой елементи на ред

void putValues( int *ia, int sz ) {
    cout << "( " << sz << " )< ";
    for ( int i = 0; i < sz; ++i ) {
        if ( i % lineLength == 0 && i )
            cout << "\n\t"; // редът е запълнен

        cout << ia[ i ];

        // разделяне на елементите със запетайки
        if ( i % lineLength != lineLength - 1 && i != sz - 1 )
            cout << ", ";
    }
    cout << " >\n";
}
```

Ако формалният аргумент е многомерен масив, в декларацията му трябва да присъстват всички размери с изключение на първия. Например,

```
void putValues( int matrix[][10], int rowSize );
```

декларира matrix като двумерен масив. Всеки ред на matrix се състои от 10 елемента.

```
int (*matrix)[10];
```

е еквивалентна декларация на matrix. Според нея matrix е указател към масив от десет елемента. С израза

```
matrix += 1;
```

се преминава към следващия ред от масива, т.е. към следващия елемент на matrix. От примера става ясно, защо трябва да се знае вторият размер на масива. Във втората декларация на matrix кръглите скоби са задължителни, защото операцията за индексване има по-голям приоритет.

```
int *matrix[ 10 ];
```

декларира matrix като масив от 10 указателя към целочислени променливи.

3.9 Области на действие

Знаем, че всеки идентификатор трябва да бъде уникален. В действителност това не е точно така. Едно и също име може да се използва за различни цели, стига да съществува контекст за различаване. Такъв контекст е сигнатурата на функциите. В предишния раздел използвахме `putValue()` като име на две функции, но всяка от тях беше с уникална сигнатура.

Друг контекст е областта на действие. В C++ съществуват три вида области на действие: глобална област на действие (`file scope`), локална област на действие (`local scope`) и област на действие за клас (`class scope`). Глобалната област на действие е частта от програмния текст, която се намира извън дефинициите на функции и класове. Това е най-външната област на действие. Тя огражда другите два вида области.

Локалната област на действие е тази част от програмния текст, която представлява дефиниция на функция. Всяка функция се свързва със своя локална област на действие. Всеки съставен оператор (или блок) в тялото на функция, включващ оператори за декларация, поддържа своя локална област на действие. В този смисъл локалните области на действие могат да се влагат. Списъкът от аргументи на една функция принадлежи на нейната локална област, а не на ограждащата глобална област.

Раздел 5.8 разглежда подробно понятието `class scope`. Засега ще отбележим само, че:

1. Всеки клас се свързва с различна област на действие.
2. Функциите-членове на даден клас се включват в областта на действие за този клас.

Всяка глобална променлива в C++, която не е явно инициализирана, се инициализира с 0. В следващия пример `i` и `j` получават стойност 0:

```
int i = 0;
int j;
```

Стойността на неинициализирана локална променлива е неопределена.

Едни и същи идентификатори могат да се използват без проблеми в различни области на действие на програмата. Например, в следващата програма се използват четири различни променливи `ia` с едно и също име.

```
void swap( int *ia, int, int );
void sort( int *ia, int sz );
void PutValues( int *ia, int sz );

int ia[ ] = { 4, 7, 0, 9, 2, 5, 8, 3, 6, 1 };
const SZ = 10;

main() {
    int i, j;
    // ...
    swap( ia, i, j );
    sort( ia, SZ );
    PutValues( ia, SZ );
}
```

Всяка променлива се свързва с някаква област на действие, като достъпът до променливата се ограничава от тази област на действие. Например, една локална променлива е достъпна само за функцията, в която е дефинирана. Нейното име може да се използва за

дефиниране на друга променлива в друга област на действие. Всяка променлива се определя еднозначно от своето име и област на действие.

Ако променливата е глобална, тя е видима (достъпна) в цялата програма. Съществуват три случая, когато нейната видимост е ограничена:

1. Ако съществува локална променлива със същото име, глобалната променлива не е видима в тялото на съответната функция. В този случай локалната променлива покрива глобалната. Как да постъпим, ако се нуждаем от стойността на глобалната променлива? Можем да използваме операцията за глобална дефиниция "::".
2. Как да постъпим, ако глобалната променлива е дефинирана в един файл, а искаме да я използваме във втори файл? Проблемът се решава, ако пред декларацията на променливата във втория файл стои ключовата дума `extern`.
3. Да предположим, че в отделни файлове, съдържащи части от програмата, сме дефинирали две глобални променливи с едно и също име. Всеки отделен файл се компилира без проблеми, но при свързването им се поражда конфликт, защото една и съща променлива е дефинирана на две места в програмата. Тази конфликтна ситуация може да се предотврати, ако използваме ключовата дума `static`.

Трите случая се разглеждат подробно в следващите три подраздела.

Операция за глобална дефиниция — "::"

Чрез тази операция се осъществява достъп до стойността на покрита глобална променлива. Следващият пример показва как се използва операцията. Функцията `fibonacci()` изчислява членовете от редицата на Фибоначи. Дефинирани са две променливи с име `max` — глобална и локална. Глобалната променлива определя максимално допустимата стойност на член от редицата. Локалната променлива задава нейната дължина. Напомняме, че формалните аргументи принадлежат на локалната област. И двете променливи са необходими в тялото на функцията, но чрез идентификатора `max` се обръщаме към съответната локална променлива. Достъпът до глобалната променлива се осъществява чрез `::max`. Ето и самия пример:

```
#include <stream.h>
const max = 65000;
const lineLength = 12;

void fibonacci( int max ) {
    if ( max < 2 ) return;
    cout << "0 1 ";

    for ( int i = 3, v1 = 0, v2 = 1, cur; i <= max; ++i ) {
        cur = v1 + v2;
        if ( cur >> ::max ) break;
        cout << cur << " ";
        v1 = v2;
        v2 = cur;
        if ( i % lineLength == 0 ) cout << "\n";
    }
}
```

`main()` изглежда така:

```
#include <stream.h>
void fibonacci( int );

main() {
    cout << "Редица на Фибоначи с 16 члена:\n"
    fibonacci( 16 );
    return 0;
}
```

След компилиране и изпълнение на програмата получаваме:

```
Редица на Фибоначи с 16 члена:
0 1 1 2 3 5 8 13 21 34 55 89
144 233 377 610
```

Външни (extern) променливи

Чрез ключовата дума `extern` декларираме променлива без да я дефинираме. Ефектът е подобен на ефекта от използването на прототип на функция и означава, че някъде в програмата е дефиниран идентификатор от съответния тип. Например,

```
extern int i;
```

показва, че в програмата съществува дефиниция

```
int i;
```

При деклариране на външни променливи, за тях не се заделя памет. Декларацията

```
extern int i;
```

може да се появи на много места в програмата. Обикновено тя се включва в заглавен файл и след това се използва, където е необходимо.

Ключовата дума `extern` може да стои и пред прототип на функция, но това е излишно, защото самият прототип показва, че функцията е дефинирана някъде в програмата. В този смисъл се използва и ключовата дума `extern`. Ето един пример за такава употреба:

```
extern void PutValues( int, int );
```

Ако в декларацията на външна променлива има явна инициализация, тази декларация се възприема като дефиниция на променливата. Следваща дефиниция на същата променлива в същата област на действие ще предизвика грешка. Например:

```
extern const bufSize = 512;           // дефиниция
```

Статични (static) глобални променливи

Всяка статична глобална променлива е невидима извън файла, в който е дефинирана. Като статични се дефинират тези глобални променливи, които представляват интерес само за даден файл. Така се избягват противоречията, ако в друг файл на програмата се дефинира глобална променлива със същото име.

По подразбиране всички inline-функции и дефиниции const се считат за статични.

По-долу е даден пример за статична функция. Името на файла е sort.C. Той съдържа три функции: bsort(), qsort() и swap(). bsort() и qsort() сортират масив в нарастващ ред. swap() се използва от тези две функции, но не е предназначена за общо ползване. Затова тя се декларира като static. (Можем да я декларираме и като inline.)

```
static void swap( int *ia, int i, int j )
{ // разменя местата на два елемента от масива
    int tmp = ia[ i ];
    ia[ i ] = ia[ j ];
    ia[ j ] = tmp;
}

void bsort( int *ia, int sz )
{ // сортиране по метода на мехурчето
    for ( int i = 0; i < sz; i++ )
        for ( int j = i; j < sz; j++ )
            if ( ia[i] > ia[j] ) swap( ia, i, j );
}

1 void qsort( int *ia, int low, int high )
2 {
3     if ( low < high ) { // ако условието не е изпълнено, край на рекурсията
4         int lo = low;
5         int hi = high;
6         int elem = ia[ low ];
7
8         for (;;) {
9             while ( ia[ ++lo ] <= elem );
10            while ( ia[ hi ] > elem ) hi--;
11            if ( lo < hi ) swap( ia, lo, hi );
12            else break;
13        } // край на for(;;)
14
15        swap( ia, low, hi );
16        qsort( ia, low, hi - 1 );
17        qsort( ia, hi + 1, high );
18    } // край на if ( low < high )
19 }
```

qsort() реализира алгоритъма на Хоар за бързо сортиране. Ще разгледаме тази функция подробно. low и high са съответно долната и горната граница на масива. qsort() е рекурсивна функция. Всеки път тя вика себе си за сортиране на по-къс масив. Условието за край се изпълнява, когато долната граница на масива стане по-голяма или равна на горната (ред 3).

elem (ред 6) е разделящият елемент. По-малките от elem елементи се преместват наляво от него, а по-големите надясно. Така масивът се разделя на два подмасива. За всеки от тях (редове 16 - 17) qsort() се прилага рекурсивно.

Разделянето на масива се реализира от оператора for(; ;) (редове 8 - 13). На всяка стъпка от цикъла lo е индекса на първия елемент на ia, който е по-голям от elem (ред 9).

Аналогично `hi` намалява до достигане на елемент, който е по-малък или равен на `elem` (ред 10). Ако `lo` е по-голямо или равно на `hi`, масивът е разделен. Тогава чрез `break` излизаме от тялото на цикъла. В противен случай разменяме местата на елементите с индекси `lo` и `hi` и преминаваме към нова итерация (ред 11).

Макар че масивът е разделен, той не е подреден, защото `elem` е `ia[low]`. Чрез обръщение към `swap()` (ред 15) `elem` получава стойността на `ia[hi]`. След това `qsort()` се прилага за двата подмасива.

Следващата функция `main()` илюстрира работата на `bsearch()` и `qsort()`. `putValues()` отпечатва масивите.

```
#include <stream.h>
#include "sort.h"    /* bsearch(), qsort() */
#include "print.h"   /* putValues() */

// за илюстрация дефинираме следните два масива
int ia1[10] = { 26, 5, 37, 1, 61, 11, 59, 15, 48, 19 };
int ia2[16] = { 503, 87, 512, 61, 908, 170, 897, 275,
               653, 426, 154, 509, 612, 677, 765, 703 };

main() {
    cout << "\nМетод на мехурчето за първия масив";
    bsearch( ia1, 10 );
    putValues( ia1, 10 );

    cout << "\nБързо сортиране за втория масив";
    qsort( ia2, 0, 15 );
    putValues( ia2, 16 );
}
```

След компилация и изпълнение на програмата получаваме следния резултат:

```
Метод на мехурчето за първия масив
( 10 )< 1, 5, 11, 15, 19, 26, 37, 48,
      59, 61 >
```

```
Бързо сортиране за втория масив
( 16 )< 61, 87, 154, 170, 275, 426, 503, 509,
      512, 612, 653, 677, 703, 765, 897, 908 >
```

3.10 Локална област на действие

При обръщение към функция, в програмния стек се заделя място за всяка локална променлива. Ако променливата не е инициализирана, нейната стойност е неопределена. При излизане от функцията, връзката между локалните променливи и техните стойности в стека се прекъсва. По тази причина адресът на локална променлива не бива да се предава извън съответната локална област на действие. Ето един пример:

```
#include <stream.h>
const strLen = 8;
char *globalString;

char *trouble() {
    char localString[ strLen ];
    cout << "Въведете низ: ";
    cin >> localString;
```

```

    return localString;      // опасно връщане
}

main() { globalString = trouble(); ... }

```

globalString получава адреса на локалния масив от символи localString. За нещастие паметта, заделена за localString, след изпълнение на trouble() се освобождава. При връщане в main() globalString вече адресира незаета памет. Това е сериозна програмна грешка, защото съдържанието на адресираната памет е непредсказуемо. Ако там се съдържа валидна комбинация от битове, програмата ще завърши, но ще даде неверен резултат.

Локалните области на действие могат да се влагат. Всеки блок, който съдържа декларации, поддържа своя локална област. В следващия пример има две локални области.

```

const notFound = -1;

// binSearch() извършва двоично търсене в сортиран масив
int binSearch( int *ia, int sz, int val )
{ // локална област ниво #1
    // включва ia, sz, val, low, high
    int low = 0;
    int high = sz - 1;

    while ( low <= high )
    { // локална област ниво #2
        int mid = ( low + high ) / 2;
        if ( val == ia[ mid ] ) return mid;
        if ( val < ia[ mid ] ) high = mid - 1;
        else low = mid + 1;
    } // край на локална област ниво #2

    return notFound;
} // край на локална област ниво #1

```

В binSearch() цикъл while дефинира втора локална област на действие. Тя е вложена в локалната област на функцията. Втората локална област съдържа една променлива — mid от тип int. Ограждащата локална област съдържа формалните аргументи ia, sz и val и локалните променливи high и low. Глобалната област огражда двете локални области. Тя съдържа само константата notFound.

Ако като идентификатор използваме променлива от тип reference, компилаторът проверява дали в текущата област на действие има дефиниция на тази променлива. Ако отговорът е отрицателен, компилаторът проверява ограждащата област на действие. Процесът продължава до откриване на дефиниция или до изчерпване съдържанието на глобалната област. Във втория случай се появява съобщение за грешка. Грешка има и когато дефиницията се среща по-късно. Операцията "::" ограничава търсенето само в глобалната област на действие.

Както знаем, цикъл for позволява да се дефинират променливи в управляващите изрази. Например:

```

for ( int i = 0; i < arrayBound; ++i )

```

Всяка дефинирана по този начин променлива принадлежи на областта, в която се намира оператор for, т.е. горният запис е еквивалентен на следния:


```
int i;
for ( i = 0; i < arrayBound; ++i )
```

Следователно и в двата случая променливата *i* е достъпна след края на цикъла. Ето един пример:

```
const notFound = - 1;

findElement( int *ia, int sz, int val ) {
    for ( int i = 0; i < sz; ++i )
        if ( ia[ i ] == val ) break;
    if ( i == sz ) return notFound;
    return i;
}
```

От казаното следва, че променлива, която се дефинира в управляващите изрази на *for*, не може да се дефинира повторно в същата област на действие. Например:

```
fooBar( int *ia, int sz ) {
    for (int i=0; iz; ++i)...    // дефиниция на i
    for (int i=0; iz; ++i)...    // грешка: предефиниране на i
    for (i=0; iz; ++i)...       // ок
}
```

Статични (static) локални променливи

Когато дадена променлива се използва само в една функция, тя се дефинира като локална за тази функция. Но обикновената локална променлива е неудобна, ако нейната стойност трябва да се запази до следващото извикване на функцията. В този случай се дефинира статична локалната променлива. За нея се отделя постоянно място в паметта. Затова стойността и се запазва до следващото извикване на функцията. Забележете, че променливата е достъпна само във функцията. Да разгледаме един пример. Ще дефинираме отново функцията *gcd()*, като запазваме дълбочината на рекурсията в статична локална променлива:

```
#include <stream.h>
traceGcd( int v1, int v2 )
{
    static int depth = 1;
    cout << "Ниво на рекурсията #"
         << depth++ << "\n";

    if ( v2 == 0 ) return v1;
    return traceGcd( v2, v1%v2 );
}
```

Стойността на *depth* се запазва до следващото извикване на *traceGcd()*. Променливата се инициализира еднократно. Следващата малка програма илюстрира действието на *traceGcd()*:

```
#include <stream.h>
extern traceGcd( int, int );

main() {
    int rslt = traceGcd( 15, 123 );
```

```
cout << "НОД на (15, 123): " << rslt << "\n";  
}
```

След компилация и изпълнение получаваме:

```
Ниво на рекурсията #1  
Ниво на рекурсията #2  
Ниво на рекурсията #3  
Ниво на рекурсията #4  
НОД на (15, 123): 3
```

Регистрови (register) локални променливи

Локалните променливи, които се използват често във функцията, могат да се дефинират като регистри. При възможност компилаторът ще ги зареди в някои от машинните регистри. Ако това е невъзможно, променливата остава в паметта. Например, индексите на масивите често се дефинират като регистри променливи:

```
for ( register int i = 0; i < sz; ++i ) ia[ i ] = i;
```

Формалните аргументи също могат да се декларират като регистри:

```
class iList {  
public:  
    int value;  
    iList *next;  
};  
  
int find( register iList *ptr, int val )  
{ // намира val в свързания списък  
    while ( ptr ) {  
        if ( ptr->value == val ) return 1;  
        ptr = ptr->next;  
    }  
    return 0;  
}
```

Ако избраните регистри променливи се използват достатъчно често, ще се увеличи бързодействието на програмата.

ГЛАВА 4

Тази глава разглежда две основни концепции в програмирането на C++: динамичното разпределение на паметта и създаването на функции с еднакви имена. Под динамично разпределение на паметта се разбира разпределение на паметта при изпълнение на програмата. С този механизъм за управление на паметта се запознахме при създаването на клас `IntArray`. Сега ще го разгледаме подробно.

Функции с еднакви имена се създават, ако те реализират еднакви операции, но над обекти от различен тип. Например, в аритметичните изрази използваме стандартни операции, които притежават тази характеристика. В тази глава ще покажем как се създават функции с еднакви имена.

4.1 Динамично разпределение на паметта

Всяка програма разполага с незаета памет, която може да използва при своето изпълнение. Този резерв от памет се нарича динамична памет (*free storage*). Благодарение на нея клас `IntArray` може да отложи заделянето на памет за масива, който дефинира, до изпълнение на програмата. Да припомним как се прави това:

```
IntArray::IntArray( int sz )
{
    size = sz;
    ia = new int[ size ];
    for ( int i = 0; i < sz; ++i)
        ia[ i ] = 0;
}
```

Клас `IntArray` има два члена-данни: `size` и `ia`. `size` представлява дължината на масива. `ia` е указател от тип `int`. Той адресира разположения в динамичната памет масив. Една особеност на динамичната памет е това, че тя не се свързва с име на променлива. С разположените в нея обекти се работи косвено чрез указатели. Втора особеност на динамичната памет е това, че тя е неинициализирана. Следователно, преди да я използваме, трябва да я инициализираме. В примера елементите на `ia` се инициализират с 0.

Динамична памет се заделя от операцията `new`, приложена към стандартен тип или име на клас. В динамичната памет може да се задели място за отделен обект или за масив от обекти. Например,

```
int *pi = new int;
```

заделя място за един обект от тип `int`. `new` връща указател към обекта и `pi` се инициализира с този указател.

```
IntArray *pia = new IntArray( 1024 );
```

заделя място за един обект от клас `IntArray`. Ако след името на класа има скоби, в тях стоят аргументи за конструктора на класа. В случая `pia` адресира един обект от клас `IntArray`, който представлява масив с 1024 елемента. Ако скобите липсват, класът трябва да притежава конструктор без аргументи или да не притежава конструктори. Например:

```
IntArray *pia2 = new IntArray;
```

Ако след името на типа стои заграден в квадратни скоби израз, new заделя място за масив от обекти. Изразът може да бъде с произволна сложност и се нарича размерност. new връща указател към първия елемент на масива. Например:

```
#include <string.h>
char* copyStr( const char *s )
{
    char *ps = new char[ strlen(s) + 1 ];
    strcpy( ps, s );
    return ps;
}
```

Може да се задели място и за масив, чиито елементи са обекти от даден клас. Например,

```
IntArray *pia = new IntArray[ someSize ];
```

заделя място за масив с дължина someSize, чиито елементи са обекти от клас IntArray.

Разпределението на паметта при изпълнение на програмата, се нарича динамично разпределение на паметта. Прието е да се казва, че pia адресира динамичен масив. Обърнете внимание, че паметта за указателя pia се заделя при компилация. Заделянето на памет при компилация се нарича статично заделяне на памет. Следователно, паметта за pia е заделена статично.

Под период на активност за една променлива се разбира частта от времето за изпълнение на програмата, през което променливата е свързана с конкретно място в паметта. Паметта за глобалните променливи се заделя в началото на програмата и остава свързана с тях до завършване на програмата. Паметта за локалните променливи се заделя при влизане в локалната област и се освобождава при напускането и. Статичните локални променливи имат период на активност като глобалните променливи.

Паметта за динамичните променливи се заделя от операцията new. Заделената по този начин памет остава свързана със съответната променлива докато не се освободи явно от програмиста. Явното освобождаване на памет се извършва от операцията delete, приложена към указателя, който адресира съответната променлива. Да разгледаме един пример.

IntArray::grow() добавя към масива половината от текущия му брой елементи. Най-напред се заделя памет за новия масив. След това елементите на стария масив се копират в новия. Останалите елементи на новия масив се инициализират с 0. Накрая заделената за стария масив памет се освобождава чрез операцията delete:

```
void IntArray::grow()
{
    int *oldia = ia;
    int oldSize = size;

    size += size/2 + 1;
    ia = new int[ size ];

    // копира елементите на стария масив в новия
```

```

    for ( int i = 0; i < oldSize; ++i) ia[ i ] = oldia[ i ];

    // инициализира останалите елементи с 0
    for ( ; i < size; ++i ) ia[ i ] = 0;
    delete oldia;
}

```

Периодът на активност на `oldia` е като на всяка друга локална променлива. Не е така, обаче, с масива, който се адресира от `oldia`. За него е заделена динамична памет. Затова тя е достъпна извън `IntArray::grow()`. Ако не се освободи явно, тя се счита от програмата за заета.

```
delete oldia;
```

освобождава тази памет.

Ако се освобождава памет, заета от масив, чиито елементи са обекти от даден клас, трябва да се посочи дължината на масива. Тя е необходима, за да се определи броят на деструкторите, които ще се извикат. Нека имаме следната дефиниция:

```
IntArray *pia = new IntArray[ size ];
```

Заетата от този масив памет се освобождава така:

```
delete [ size ] pia;
```

`delete` трябва да се използва само за памет, заделена чрез `new`. В противен случай ефектът е непредсказуем. Няма забрана за прилагане на `delete` към указател със стойност 0. Ако стойността на указателя е 0, той е свободен и не адресира нищо. Следват няколко примера за правилно и неправилно използване на операцията `delete`:

```

void f() {
    int i;
    char *str = "dwarves";
    int *pi = &i;
    IntArray *pia = 0;
    double *pd = new double;

    delete str;      // опасна операция
    delete pi;       // опасна операция
    delete pia;      // безопасна операция
    delete pd;       // безопасна операция
}

```

Динамичната памет не е неограничена. Тя може да се изчерпи при изпълнение на програмата. Неуспешното завършване на `delete` ускорява изчерпването. `new` връща нулев указател, ако наличната в момента динамична памет е недостатъчна. Не бива да се пренебрегва възможността `new` да върне указател със стойност 0. Например, функцията `grow()` няма да работи, ако `new` не успее да задели нужната памет. Да припомним как изглежда `grow()`:

```

ia = new int[ size ];
// неприятност в случай, че new върне 0
for ( int i = 0; i < oldSize; ++i) ia[ i ] = oldia[ i ];

```

Програмистът трябва да предотврати изпълнението на цикъла, ако new върне 0. Той може да провери дали new връща 0 непосредствено след тази операция:

```

ia = new int[ size ];
if ( !ia ) error("IntArray::grow(): не достига динамична памет ");
for ( int i = 0; i < oldSize; ++i) ia[ i ] = oldia[ i ];

```

error() е дефинирана от потребителя функция, която отпечатва съобщение за грешка и приключва изпълнението на програмата.

Следващият пример илюстрира действието на grow():

```

#include <stream.h>
#include "IntArray.h"
IntArray ia( 10 );

main()
{
    cout << "дължина: " << ia.getSize() << "\n";
    for ( int i = 0; i < ia.getSize(); ++i)
        ia[ i ] = i * 2; // инициализация

    ia.grow();
    cout << "нова дължина: " << ia.getSize() << "\n";
    for ( int i = 0; i < ia.getSize(); ++i)
        cout << ia[ i ] << " ";
}

```

След компилация и изпълнение получаваме следния резултат:

```

дължина: 10
нова дължина: 16
0 2 4 6 8 10 12 14 16 18 0 0 0 0 0 0

```

Следващият пример демонстрира изчерпване на динамичната памет. Той представлява една рекурсивна функция. Условие за край на рекурсията е получаването на нулев указател при поредното изпълнение на new.

```

#include <stream.h>
void exhaustFreeStore( unsigned long chunk )
{
    static int depth = 1;
    static int report = 0;

    ++depth; // показва дълбочината на рекурсията
    double *ptr = new double[ chunk ];
    if ( ptr ) exhaustFreeStore( chunk );

    // изчерпване на динамичната памет
    delete ptr;
}

```

```

    if ( !report++ )
        cout    << "Не достига динамична памет:"
                << "\tОбем: " << chunk;
                << "\tДълбочина: " << depth;
}

```

При изпълнение на `exhaustFreeStore()` с четири различни стойности на аргумента, получаваме следните резултати:

Не достига динамична памет:	Обем: 1000000	Дълбочина: 4
Не достига динамична памет:	Обем: 100000	Дълбочина: 22
Не достига динамична памет:	Обем: 10000	Дълбочина: 209
Не достига динамична памет:	Обем: 1000	Дълбочина: 2072

В C++ съществува библиотека, която улеснява следенето на динамичната памет.

Един обект може да се разположи на точно определено място в динамичната памет. За целта се използва следния вид на операцията `new`:

```
new (адрес_на_мястото) спецификатор_на_тип;
```

където `адрес_на_мястото` е указател. За да се използва тази форма на операцията `new`, в програмата трябва да се включи заглавният файл `new.h`. Ако предварително заделим място в паметта, по-късно можем да разположим на това място конкретен обект, като използваме специалната форма на `new`. Например:

```

#include <stream.h>
#include <new.h>

const chunk = 16;
class Foo { public: int val; Foo() { val = 0; } };

// предварително заделяне на памет, където
// по-късно ще разположим 16 обекта Foo
char *buf = new char[ sizeof(Foo) * chunk ];

main() {
    // Разполагане на 16-те обекти Foo в buf
    Foo *pb = new (buf) Foo[ chunk ];

    // проверка дали обектите са разположени в buf
    if ( (char*)pb == buf )
        cout << "Операцията new работи!: pb: "
              << pb << "buf: " << (void*)buf << "\n";
}

```

След компилиране и изпълнение на програмата получаваме следния резултат:

```
Операцията new работи!: pb: 0x234cc buf: 0x234cc
```

В програмата има един тънък момент — преобразуването на `buf` в тип `void*`. То е задължително, ако искаме да отпечатаме адреса, на който е разположен даден символен низ.

Ако аргументът на операцията "<<" е указател от тип `char*`, на екрана се извежда символният низ, а не адреса му. Тази особеност е причина за преобразуването на `buf` в тип `void*`. Операцията "<<" е дефинирана два пъти, съответно за аргументи от тип `char*` и `void*`.

Специалната форма на операцията `new` се прилага предимно за класове, но понякога се използва и при стандартните типове данни. Например:

```
#include <new.h>
int *pi = new int;

main() {
    int *pi2 = new (pi) int;
}
```

4.2 Клас `IntList` (линеен списък). Примерна реализация

Ще илюстрираме работата с указатели и използването на операциите `new` и `delete` като дефинираме клас `IntList`. Елементите на списъка включват две стойности: цяло число, наречено ключ и указател, който съдържа адреса на следващия елемент от списъка.

```
int val;           // ключ
ListItem *next;    // указател към следващия елемент
```

Линейният списък е последователност от свързани елементи. Всеки елемент съдържа ключ и указател към следващия елемент. Ако даден елемент е последен в списъка, указателят към следващ елемент има стойност 0. Линейният списък може да бъде празен. Следващият оператор дефинира празен списък:

```
IntList il;    // il е празен списък
```

Добавянето на елемент е една от възможните операции с линеен списък. Функцията `insert()` добавя елемент в началото на списъка:

```
il.insert( someValue );
```

Функцията `append()` добавя елемент в края на списъка:

```
il.append( someValue );
```

Друга възможна операция с линеен списък е отстраняване на елемент. За тази цел се използва функцията `remove()`:

```
il.remove( someValue );
```

`display()` отпечатва елементите на списъка:

```
il.display();
```


Следващата програма илюстрира действието на изброените функции:

```
#include "IntList.h"
const SZ=12;

main()
{
    IntList il;
    il.display();

    for ( int i = 0; i < SZ; ++i ) il.insert(i);
    il.display();

    IntList il2;
    for( int i = 0; i < SZ; ++i ) il2.append(i);
    il2.display();

    return 0;
}
```

След компилация и изпълнение на програмата получаваме следния резултат:

```
( празен списък )
( 11 10 9 8 7 6 5 4 3 2 1 0 )
( 0 1 2 3 4 5 6 7 8 9 10 11 )
```

Да започнем описанието на клас IntList. Неподходящо представяне на класа може да създаде неприятности при неговото използване. Ще сгрешим, ако дефинираме val и next като членове на IntList:

```
class IntList {
public:
    IntList( val = ??? );
    //...
private:
    int val;
    IntList *next;
};
```

Това описание на IntList поражда няколко проблема, причина за които е смесването на обект и член на класа. Например, при тази дефиниция на IntList не може да се създаде празен списък. Въпросителните знаци в сигнатурата на конструктора подчертават този проблем. Не съществува стойност за val, с която да означим празен списък. Други проблеми се появяват при дефиниране на insert() и remove(). Причината е в двойствената интерпретация на обектите от клас IntList — като цял списък и като негов първи елемент. За да ги разграничим можем да дефинираме два класа — IntList и IntItem. По-долу е дадена дефиницията на класа IntItem:

```
class IntList;
class IntItem {
    friend class IntList;
private:
    IntItem(int v=0) { val = v; next = 0; }
    IntItem *next;
```

```
int val;
};
```

IntItem е скрит клас (private class). Само клас IntList може да създава и обработва обекти от клас IntItem. Такъв е смисълът на декларацията:

```
friend class IntList;
```

Раздел 5.5 разглежда дефинирането и употребата на т.нар. приятелски класове. Раздел 5.1 представя последиците от дефинирането на класа като скрит (private) или общодостъпен (public). Следва описанието на клас IntList:

```
class IntItem;
class IntList {
public:
    IntList( int val ) { list = new IntItem( val ); }
    IntList() { list = 0; }
    //...
private:
    IntItem *list;
};
```

Упражнение 4-1. Защо са необходими два конструктора за клас IntList? Защо не се използва по-простата дефиниция

```
IntList ( int val = 0 );
```

Упражнение 4-2. Да добавим още един член към клас IntList

```
int len;    // дължина на списъка
```

който показва текущия брой на елементите на списъка. Какви промени ще предизвика това в дефинициите на функциите-членове на IntList.

Нека сега реализираме споменатите в началото на раздела функции. Първата от тях, insert() добавя нов елемент в началото на списъка:

```
IntList::insert( int val )
{ // добавя елемент в началото на списъка
    IntItem *pt = new IntItem( val );
    pt->next = list;
    list = pt;
    return val;
}
```

append() е по-сложна функция. Тя добавя нов елемент в края на списъка. Ще създадем помощна функция atEnd(), която връща указател към последния елемент на списъка.

```

IntItem* IntList::atEnd()
{ // връща указател към последния елемент на списъка
  IntItem *prv, *pt;
  for ( prv=pt=list; pt; prv=pt, pt=pt->next )
    ; // празен оператор
  return prv;
}

```

append() трябва да провери дали списъкът не е празен. Ето самата функция:

```

IntList::append( int val )
{ // добавя елемент в края на списъка
  IntItem *pt = new IntItem( val );
  if ( list == 0 )
    list = pt;
  else
    (atEnd())->next = pt;
  return val;
}

```

Упражнение 4-3. Какви промени в append() ще предизвика добавянето на нов член

```
IntItem *endList;
```

към класа IntList? Какви предимства и недостатъци има това?

Често се налага елементите на списъка да се отпечатават. За целта се използва функцията display(). Тя изобразява по 16 елемента на ред. Следва дефиниция на функцията:

```

#include <stream.h>
const int lineLength = 16;

IntList::display()
{ // изобразява елементите на списъка
  if ( list == 0 ) {
    cout << "( празен списък )\n";
    return 0;
  }

  cout << "(";
  int cnt = 0;      // брой на отпечатаните елементи
  IntItem *pt = list;

  while ( pt ) {
    if ( ++cnt % lineLength == 1 && cnt != 1 ) cout << "\n";
    cout << pt->val << " ";
    pt = pt->next;
  }

  cout << ")\n";
  return cnt;
}

```

Проверката

```
if ( ++cnt % lineLength == 1 && cnt != 1 )
```

гарантира, че лявата затваряща скоба няма да се отпечата сама на първия ред. Следва подробно описание на заглавния файл `IntList.h`, съобразено с досегашните дефиниции:

```
class IntList;      // дефиницията следва по-нататък
class IntItem {
    friend class IntList;
private:
    IntItem( int v=0 ) { val = v; next = 0; }
    IntItem *next;
    int val;
};

class IntList {
public:
    IntList( int val ) { list = new IntItem( val ); }
    IntList() { list = 0; }
    display();
    insert( int val = 0 );
    append( int val = 0 );
private:
    IntItem *list;
    IntItem *atEnd();
};
```

На потребителя трябва да се позволи да изтрива целия списък или елементи от него. Дали изтриване на елемент от празен списък е грешка, се определя от създателя на класа. Във всички случаи обаче, предикатната функция `isEmpty()` е полезна:

```
class IntItem { /*....*/ }
class IntList {
public:
    isEmpty() { return list == 0; }
    //...
private:
    IntItem *list;
};
```

Ето как може да се изтрие целия списък:

```
IntList::remove()
{ // изтрива целия списък
    IntItem *tmp, *pt = list;
    int cnt = 0;

    while ( pt ) {
        tmp = pt;
        pt = pt->next;
        ++cnt;
        delete tmp;
    }
    list = 0;
```

```

    return cnt;
}

```

Функцията връща броя на изтрите елементи.

Потребителят може да иска да се изтрият всички елементи с определен ключ. Специално се разглежда случая, при който се изтрива първият елемент на списъка, защото list променя стойността си. Ето описание на функцията:

```

IntList::remove( int val )
{ // изтрива всички елементи с определен ключ
  IntItem *prv, *tmp, *pt = list;
  cnt = 0;

  // когато ключа на първия елемент е val
  while ( pt && pt->val == val ) {
    tmp = pt->next;    // запазва указател към следващия елемент
    delete pt;
    ++cnt;
    pt = tmp;
  }

  if ( ( list = pt ) == 0 )
    return cnt;    // списъкът е празен
  prv = pt; pt = pt->next;

  while ( pt ) { // преминаване през списъка
    if ( pt->val == val ) {
      tmp = prv->next = pt->next;
      delete pt;
      ++cnt;
      pt = tmp;
    }
    else {
      prv = pt;
      pt = pt->next;
    }
  } // край на while ( pt )
  return cnt;
}

```

Друга полезна функция е length(). Тя връща броя на елементите в списъка. Ако той е празен, дължината му е 0.

```

IntList::length() {
  int cnt = 0;
  IntItem *pt = list;
  for ( ; pt; pt = pt->next, ++cnt )
    ; // празен оператор
  return cnt;
}

```

Следващата програма тества описаните функции. Промените в заглавния файл IntList.h се оставят на читателя.

```

#include "IntList.h"
#include <stream.h>

```

```

const SZ = 12;
const ODD = 1;

main() {
    IntList il;    // празен списък

    // проверка за коректност на функциите при празен списък
    if ( il.isEmpty() && il.length() == 0 && il.remove() == 0 )
        cout << "Функциите работят за празен списък.\n";

    // всеки нечетен елемент получава ключ 1
    for ( int i = 0; i < SZ; ++i )
        il.append( i%2 == 0 ? i : ODD );
    il.display();

    // илюстрира действието на remove( someValue )
    cout << "Изтрети са " << il.remove( ODD ) << "елемента с ключ "
        << ODD << ": ";
    il.display();

    // илюстрира действието на remove()
    int len = il.length();
    if ( il.remove() == len )
        cout << "Всичките " << len << "елемента са изтрети: ";
    il.display();
    return 0;
}

```

След компилиране и изпълнение получаваме следния резултат:

```

Функциите работят за празен списък.
( 0 1 2 1 4 1 6 1 8 1 10 1 )
Изтрети са 6 елемента с ключ 1: ( 0 2 4 6 8 10 )
Всичките 6 елемента са изтрети: ( празен списък )

```

Упражнение 4-4. Дефинирайте функция `IntList::removeFirst()`, която връща ключа на първия елемент. Не забравяйте да обработвате специално празния списък.

Упражнение 4-5. Дефинирайте функция `IntList::removeLast()`, която връща ключа на последния елемент от списъка. Специално обработвайте празния списък.

Често се налага два списъка да се свържат в един. Тази операция изглежда проста, но при реализирането и лесно се допуска грешка. Следващата реализация вероятно ще предизвика неприятности:

```

#include "IntList.h"

void IntList::concat( IntList& il )
{
    (atEnd())—>next = il.list;
}

```

Проблемът е в това, че една и съща последователност от елементи е достъпна чрез два обекта от клас `IntList`. Двата обекта могат да изтриват по различно време елементи от списъка. Програмата ще се държи странно, ако вторият обект се опита да работи с вече

изтрят от първия обект елемент. Неприятност може да има и когато вторият обект изтрие елемент, който е изтрят вече от първия обект, защото на това място в паметта може да има данни с друго предназначение. За да отстраним недостатъците на тази реализация, ще копираме всеки елемент от втория списък след края на първия. Тогава concat() изглежда така:

```
void IntList::concat( IntList& il )
{ // добавя il.list към списъка, който извиква функцията
  IntItem *pt = il.list;

  while ( pt ) {
    append( pt->val );
    pt = pt->next;
  }
}
```

Интересна операция е обръщането на списък. При нея първите елементи на списъка стават последни и обратно. Функцията, която реализира тази операция, е кратка, но работата с указателите крие опасности и може да доведе до грешки. Следващата дефиниция е правилна:

```
void IntList::reverse()
{
  IntItem *pt, *prv, *tmp;
  prv = 0; pt = list; list = atEnd();

  while ( pt != list ) {
    tmp = pt->next;
    pt->next = prv;
    prv = pt;
    pt = tmp;
  }
  list->next = prv;
}
```

Следващата малка програма тества concat() и reverse():

```
#include " IntList.h"
const SZ = 8;

main() {
  IntList il, il2;
  for ( int i = 0; i < SZ/2; ++i )
    il.append(i);
  for ( i = SZ/2; i < SZ; ++i )
    il2.append(i);

  il.display(); il2.display();
  il.concat( il2 ); il.display();
  il.reverse(); il.display();
  return 0;
}
```

След компилация и изпълнение получаваме:

```
( 0 1 2 3 )  
( 4 5 6 7 )  
( 0 1 2 3 4 5 6 7 )  
( 7 6 5 4 3 2 1 0 )
```

Упражнение 4-6. Създайте функция, която добавя в списъка елемент с даден ключ, така че елементът след него да е първият елемент на списъка с ключ по-голям от дадения.

Упражнение 4-7. Добавете нов член към дефиницията на `IntList`:

```
IntItem *endList;
```

Коригирайте функциите-членове на клас `IntList`, така че трите примерни програми от този раздел да дават очаквания резултат.

4.3 Функции с еднакви имена

В естествените езици съществуват думи с няколко значения. Във всеки отделен случай валидното значение се определя в зависимост от контекста. Някои ключови думи в C++ също се интерпретират различно в различни ситуации. В следващия пример `static` има различен смисъл в зависимост от това, къде се намира дефиницията:

```
static int depth;
```

`depth` може да бъде локална или глобална статична променлива. В следващата глава ще разгледаме още едно значение на ключовата дума `static` — за дефиниране на статичен член на клас. За да определим значението на `static`, трябва да използваме някакъв контекст. Ако такъв липсва, думата предизвиква двусмислие. То е резултат от употреба на дума с две или повече еднакво вероятни значения.

В естествените езици често двусмислието е преднамерено. За разлика от компютъра човешкият мозък е в състояние да обработва двусмислията. За компютъра липсата на контекст или неговата непълнота са признак за грешка. В този раздел ще обсъдим как се разрешават двусмислията при дефиниране на функции с еднакви имена.

Защо са необходими функции с еднакви имена?

В C++ няколко функции могат да получат едно и също име, ако притежават уникална сигнатура, т.е. ако се различават по броя или типа на аргументите си. Например:

```
int max( int, int );  
double max( double, double );  
Complex &max( const Complex&, const Complex& );
```

Функция с име `max()`, може да се създаде за всеки тип. Всяка от тези функции реализира една и съща съвкупност от действия — връща по-големия от своите аргументи.

От гледна точка на потребителя съществува само една операция за определяне на максимална стойност. Как се реализира това за него няма значение. По-важно за него е, че може да използва следните оператори:


```
int i = max( j, k );  
Complex c = max( a, b );
```

Така са реализирани стандартните аритметични операции. Например, в израза

1 + 3

се използва операция за събиране на цели числа, докато в израза

1.0 + 3.0

се използва същата операция, но за числа с плаваща точка. Във всеки конкретен случай компилаторът решава коя операция за събиране да приложи.

Ако в C++ не съществуваше възможност за създаване на функции с еднакви имена, всяка функция трябваше да притежава уникално име. Функциите за максимум щяха да се декларират по следния начин:

```
int max( int, int );  
double fmax( double, double );  
Complex &Cmax( const Complex&, const Complex& );
```

Лексическата сложност в този пример не е очевидна, но в една голяма и сложна програма изискването за уникалност на имената създава затруднения. Дефинирането на функции с еднакви имена премахва ненужната сложност на записите.

Как се интерпретират функции с еднакви имена?

Ако едно и също име на функция се декларира на няколко места в програмата, компилаторът интерпретира втората (и следващите) декларация по следния начин:

- Ако сигнатурата и типът на резултата в двете декларации съвпадат, втората декларация се смята за повторение на първата. Например:

```
// декларира една и съща функция два пъти  
extern void print( int *ia, int sz );  
void print( int *array, int size );
```

Имената на аргументите не оказват влияние при сравняване на сигнатурите.

- Ако сигнатурите в двете декларации съвпадат, но типът на резултата е различен, втората декларация се смята за невярна повторна декларация на първата. Компилаторът съобщава за грешка. Например:

```
unsigned int max( int*, int sz );  
extern int max( int *ia, int );    //грешка
```

Различният тип на резултата не е достатъчно различие за да бъде коректна и втората декларация.

- Ако сигнатурите в двете декларации се различават по броя или типа на аргументите, компилаторът смята, че се декларират функции с едно и също име. Двете могат да съществуват едновременно и компилаторът решава във всеки конкретен случай, коя от тях да извика. Например:

```
extern void print( int*, int );
void print( double *da, int sz );
```

Знаем, че описанието typedef дава ново име на съществуващ тип данни, но не дефинира нов тип данни. Следващите две декларации на search() имат еднакви сигнатури. Втората декларация ще предизвика грешка при компилация, защото сигнатурите съвпадат, за разлика от типа на резултата.

```
// typedef не дефинира нов тип
typedef char *string;

extern int search( string );
extern char *search( char* );           // грешка
```

Кога не се създават функции с еднакви имена?

Създаването на функции с едно и също име не винаги е удачно. Такъв пример представляват функциите, които не реализират сходни операции. Например, следващите функции работят с един и същ абстрактен тип данни. Всяка от тях, обаче, реализира специфична операция за този тип. Затова обединяването им под едно общо име е неправилен подход.

```
void setDate( Date&, int, int, int );
Date& convertDate( char* );
void printDate( const Date& );
```

В случаи като този имената дори се усложняват, за да изразят връзката между действията и данните. Използването на класове в C++ опростява имената. Например, изброените функции могат да станат членове на клас Date:

```
class Date {
public:
    void set( int, int, int );
    Date &convert( char* );
    void print();
    // ...
};
```

Нека сега разгледаме следните функции-членове на клас Screen:

```
Screen& moveHome();
Screen& moveAbs( int, int );
Screen& moveRel( int, int, char *direction );
Screen& moveX( int );
Screen& moveY( int );
```

Всички те осъществяват някакво движение на курсора върху екрана. На пръв поглед изглежда, че е удачно да ги обединим под общото име `move()`. Но ако ги разгледаме внимателно, ще видим, че последните две функции имат еднаква сигнатура и следователно не могат да използват общо име. Можем, обаче, да обединим двете функции в една по следния начин:

```
Screen& move( int, char xy );
```

Така получаваме четири функции с уникална сигнатура. Ако опитите покажат, че движението по `x` е по-вероятно от движението по `y`, вторият аргумент може да получи стойност по подразбиране:

```
Screen& move( int, char xy = 'x' );
```

Същите опити могат да препоръчат някаква стъпка на придвижване, която ще използваме като стойност по подразбиране на първия аргумент:

```
Screen& move( int sz = 1, char xy = 'x' );
```

Сигнатурата, която получаваме сега не е уникална, защото `moveHome()` също се извиква без параметри.

Тук човек може да попита, какво е предимството от обединяването на тези две функции под едно общо име. В случая общото име унищожава информацията за характера на движението. Името `moveHome` дава много повече информация от името `move`. Движението на курсора е обща операция за изброените функции, но за някои от тях движението е специфично. `moveHome()` е специален случай на движение на курсора — движение до определена точка на екрана (неговия център). Може би най-доброто решение е дефиниране на специално име за тази функция, а при реализирането и да използваме `move()`. Ето как може да стане това:

```
inline Screen& Screen::home()
{
    return move( 0, 0 );
}
```

Остават втората и третата функция. Техните сигнатури са уникални. Затова те могат да използват общото име `move()`. Те могат да се обединят и в една функция:

```
move( int, int, char* = 0 );
```

Последният пример показва как да използваме богатите възможности на езика `C++`. Кой от тях ще изберем, зависи от логиката на проблема, който решаваме. Не бива да използваме необосновано всички средства на езика, само защото съществуват.

Разрешаване на двусмислието при функции с общо име

Функциите с едно и също име се различават чрез своята сигнатура. Да декларираме четири различни функции с общо име `print()`:

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

При обръщение към функция с име `print()` се извършва т.нар. съпоставяне на аргументи (argument matching). Това означава, че се сравняват типовете на формалните аргументи на всяка от четирите функции с фактическите параметри на обръщението. Съществуват три възможни резултата:

1. Съответствие.

При съответствие, се осъществява обръщение към конкретна функция. Например:

```
unsigned a;

print( 'a' ); // обръщение към print(char);
print( "a" ); // обръщение към print(char*);
print( a );   // обръщение към print(unsigned);
```

2. Несъответствие.

Несъответствие се получава, ако фактическият аргумент не съответства на формалния аргумент в нито една от функциите с име `print()`. Например:

```
int *ip;
SmallInt si;

print( ip );    // грешка: несъответствие
print( si );    // грешка: несъответствие
```

3. Противоречиво съответствие.

Когато съществува съответствие с повече от една функция, съответствието е противоречиво. Следващото обръщение към `print()` е противоречиво, защото има съответствие с три от изброените функции:

```
unsigned long ul;

print( ul );    // грешка: противоречиво съответствие
```

Съответствие може да се получи по един от следните три начина, изброени поред на техния приоритет:

1. Точно съответствие.

Получава се, когато типовете на формалния и фактическия аргумент съвпадат. Например:

```
extern ff( int );
extern ff( char* );

ff( 0 );      // съответствие с ff(int)
```

Константата 0 е от тип int, затова съществува точно съответствие с ff(int).

2. Съответствие, получено чрез стандартно преобразуване.

Ако точно съответствие не е открито, се прилага стандартно преобразуване за фактическия параметър. Например:

```
class X;
extern ff( X& );
extern ff( char* );

ff( 0 );      // съответствие с ff(char*)
```

3. Съответствие, получено чрез преобразуване, дефинирано от потребителя.

Ако съответствие по точки 1 и 2 не може да се получи, но има подходящо преобразуване, дефинирано от потребителя, компилаторът го прилага и получава съответствие. Например:

```
class SmallInt {
public:
    operator int();
    // ...
}

SmallInt si;
extern ff( int );
extern ff( char* );

ff( si );      // съответствие с ff(int)
```

Операция int() се нарича операция за преобразуване. Чрез нея клас SmallInt дефинира свое "стандартно" преобразуване. Раздел 6.5 разглежда дефинирането на преобразувания от потребителя.

Особености при точно съответствие

Когато фактическият параметър е от тип char, short или float, проверката за точно съответствие преминава през два етапа: Първо се проверява за съвпадение на типовете. Например:

```
ff( char );
ff( long );

ff ( 'a' );    // съответствие с ff(char)
```

Символната константа е от тип `char` и затова получаваме точно съответствие с `ff(char)`.

Ако при първия етап не се получи съответствие, споменатите типове се преобразуват в типове с по-голям обхват:

- Аргумент от тип `char`, `unsigned char` или `short` се преобразува в тип `int`. Аргумент от тип `unsigned short` се преобразува в `int`, ако в компютъра `int` се представя в повече байтове от `short`. В противен случай се преобразува в тип `unsigned int`.
- Аргумент от тип `float` се преобразува в тип `double`.

На втория етап сравнението се осъществява на базата на извършените преобразувания. Например:

```
ff( int );  
ff( long );  
ff( short );  
  
ff ( 'a' );    // съответствие с ff(int)
```

Символната константа `'a'` се преобразува в тип `int`. Така се получава точно съответствие с `ff(int)`. Съответствие със `ff(long)` или `ff(short)` може да се получи, ако се приложи стандартно преобразуване, но с откриване на точно съответствие, търсенето на съответствие завършва.

Точно съответствие няма да се получи, ако фактическият аргумент е от тип `int`, а формалният е от тип `short` или `char`. Положението е аналогично, когато фактическият аргумент е от тип `double`, а формалният от тип `float`. Да разгледаме функциите:

```
ff( long );  
ff( float );
```

Следващото обръщение поражда противоречиво съответствие:

```
ff( 3.14 );    // грешка: противоречиво съответствие
```

Числовата константа `3.14` е от тип `double`, поради което точно съответствие няма. За да получи съответствие, компилаторът прилага стандартно преобразуване. Възможни са две такива преобразувания. Те имат еднакъв приоритет, затова обръщението е противоречиво. Противоречието може да се отстрани като се приложи явно преобразуване. Например:

```
ff( long(3.14));    // съответствие с ff(long)
```

Друга възможност е използване на суфикс `F`, за да се покаже, че `3.14` е от тип `float`:

```
ff( 3.14F );    // съответствие с ff(float)
```

Нека са дадени декларациите:

```
ff( unsigned );  
ff( int );
```

```
ff( char );
```

При обръщение с аргумент от тип `unsigned char`, се получава точно съответствие с `ff(int)`. Съответствие с другите две функции може да се получи чрез стандартни преобразувания.

```
unsigned char uc;  
ff( uc );    // съответствие с ff(int)
```

При съпоставяне на аргументите се прави разлика между константен и "обикновен" указател. Същото важи за аргументи от тип `reference`. Ето един пример:

```
extern void ff( const char* );  
extern void ff( char* );  
  
char *cp;  
const char *pcc;  
  
ff( pcc );    // съответствие с ff( const char* )  
ff( cp );     // съответствие с ff( char* )  
ff( 0 );      // грешка: противоречиво съответствие
```

Последното обръщение е противоречиво. Тъй като константата `0` е от тип `int`, точно съответствие няма. Извършва се стандартно преобразуване, което предизвиква съответствие с двете функции.

Знаем, че всеки изброим тип с име дефинира уникален тип. За да се получи съответствие, фактическият аргумент трябва да бъде конкретен елемент на типа, или променлива от този тип. Например:

```
enum Bool { FALSE, TRUE } found;  
enum Stat { FAIL, PASS };  
  
extern void ff( Bool );  
extern void ff( Stat );  
extern void ff( int );  
  
ff( PASS ); // съответствие с ff( Stat )  
ff( 0 );    // съответствие с ff( int )  
ff( found ); // съответствие с ff( Bool )
```

Точното съответствие може да се избегне, ако се използва явно преобразуване на типовете. Да разгледаме следните две декларации:

```
extern void ff( int );  
extern void ff( void* );
```

Обръщението

```
ff( 0xffbc );
```

е обръщение към `ff(int)`, защото `0xffbc` е целочислена константа, записана в шестнайсетична бройна система. `ff(void*)` може да се извика чрез явно преобразуване:

```
ff( (void *)0xffbc ); // обръщение към ff( void* )
```

Следователно чрез явни преобразувания може да се изпълни точно определена функция от група функции с общо име.

Особености при съответствие чрез стандартно преобразуване

При отсъствие на точно съответствие, компилаторът опитва да получи съответствие чрез стандартно преобразуване. Да разгледаме един пример:

```
extern void ff( char* );
extern void ff( double );

ff( 'a' ); // съответствие с ff( double )
```

Стандартните преобразувания се извършват по следните правила:

1. Всеки числов тип се преобразува във всеки друг числов тип, включително и `unsigned`.
2. Всеки изброим тип се преобразува в произволен числов тип.
3. Числото `0` се преобразува в произволен числов тип или в указател от произволен тип.
4. Указател от произволен тип се преобразува в тип `void*`.

Следващият пример е илюстрация на тези правила:

```
extern void ff( char* );
extern void ff( void* );
extern void ff( double );

main() {
    int i;
    ff( i ); // обръщение към ff( double )
    ff( &i ); // обръщение към ff( void* )
    ff( "a" ); // обръщение към ff( char* )
}
```

Всички стандартни преобразувания са с еднакъв приоритет. Например, преобразуването на тип `char` в тип `unsigned char` не е по-приоритетно от преобразуването на тип `char` в тип `double`. Няма никакво значение, колко близки по смисъл са двата типа. Стандартните преобразувания могат да предизвикат съответствие с няколко функции. Тогава се създава двусмислие, което компилаторът не може да разреши. Той извежда съобщение за грешка. Да разгледаме още един пример:

```
extern void ff( unsigned );
extern void ff( float );
```


Следващите обръщения са противоречиви. Те ще предизвикат грешка при компилация:

```
ff( 'a' );  
ff( 0 );  
ff( 2uL );  
ff( 3.14159 );
```

Двусмислието може да се избегне чрез явни преобразувания.

Съпоставяне на аргументи от тип **reference**

В раздел 3.7 споменахме, че когато аргументът се предава чрез адрес, но типовете на формалния и фактическия аргумент не съвпадат, се дефинира временна променлива. Стандартни преобразувания, които не изискват създаване на временна променлива са с по-висок приоритет от останалите. Да разгледаме един пример:

```
extern ff( char& );  
extern ff( short );  
int i;  
  
ff( i );      // съответствие с ff(short)
```

Преобразуването на тип `int` в тип `short` не изисква генериране на временна променлива, за разлика от преобразуването на `int` в `char&`. Следователно, първото преобразуване е с по-висок приоритет. Затова няма противоречиво съответствие. Ако аргументът на втората функция беше от тип `short&`, щеше да се получи двусмислие, защото никое от стандартните преобразувания нямаше да има приоритет. При точно съответствие, временна променлива не се генерира. Например:

```
ff( 'a' );      // точно съответствие с ff(char&)
```

Съпоставяне на функции с няколко аргумента

При съпоставяне на функции с няколко аргумента, правилата за съпоставяне се прилагат последователно за всеки от аргументите. Избира се функцията, с която има точно съответствие. Ако такава няма, се избира функцията с "най-близка" сигнатура. Да разгледаме един пример:

```
extern ff( char*, int );  
extern ff( int, int );  
ff( 0, 'a' );    // обръщение към ff( int, int )
```

Обръщението

```
ff( 0, 'a' );
```

ще предизвика изпълнение на втората функция, защото:

1. Нейният първи аргумент е по-добър. Константата 0 е от тип `int`, а това съответства точно на типа на първия аргумент на `ff(int, int)`.
2. Вторият аргумент на двете функции е от един и същи тип. 'a' съответства точно с втория аргумент на двете функции.

Съпоставянето може да доведе до противоречие, ако не съществува критерий, който да определи, коя от функциите има "по-близка" сигнатура. Следващият пример е такъв, защото и двете функции `min()` изискват по две стандартни преобразувания:

```
int i, j;  
extern min( long, long );  
extern min( double, double );  
  
// грешка: противоречивост  
min( i, j );
```

Противоречие се получава и в следващия пример:

```
extern foo( int, int );  
extern foo( double, double );  
  
// грешка: противоречивост  
foo( 'a', 3.14F );
```

И за двете функции `foo()` типът на единия аргумент съвпада точно с типа на съответния фактически аргумент, а за другият аргумент се прилага стандартно преобразувание.

Съпоставяне при сигнатура с инициализация

Особеностите на съпоставянето при сигнатура с инициализация, ще изясним чрез пример:

```
extern ff( int );  
extern ff( long, int = 0 );  
main() {  
    ff( 2L );      // съответствие с ff( long, 0 )  
    ff( 0, 0 );    // съответствие с ff( long, int )  
    ff( 0 );       // съответствие с ff( int )  
    ff(3.14 );    // грешка: противоречивост  
}
```

Съответствие може да се получи както при обръщение с пълен брой аргументи, така и при по-малко аргументи. Последното обръщение е противоречиво, защото чрез стандартни преобразувания се получава съответствие с двете функции `ff()`. Изборът на `ff(int)` няма приоритет, независимо от това, че тази функция има толкова аргументи, колкото и обръщението.

Функции с еднакви имена и области на действие

Всички функции с общо име трябва да се декларират в една и съща област на действие, защото в противен случай една от тях може да покрие останалите. Функция, декларирана в

локална област на действие, покрива функцията със същото име, декларирана в глобалната област на действие. Например:

```
extern void print( char* );
extern void print( double );

void fooBar( int val )
{
    // покрива двете глобални функции със същото име
    extern void print( int );
    // грешка: print(char*) е невидима в локалната област
    print( "Value: " );
    print( val );    // ок: print(int);
}
```

Знаем, че всеки клас поддържа своя област на действие. Това означава, че функциите-членове на два различни класа с едно и също име, са валидни само в рамките на своя клас.

Упражнение 4-8. Декларирайте множество функции с общо име `error()`, така че обръщенията

```
error( "Array out of bounds: ", index, upperBound );
error( "Division by zero" );
error( "Invalid selection", selectVal );
```

да са коректни.

Предефиниране на операция `new`

Операцията `new` може да се предефинира от програмиста. Стандартната операция `new` има следния прототип:

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

`size` е размерът на необходимата памет в байтове⁵.

Всяка нова дефиниция на `new` изисква типът на резултата да е `void*` и типът на първия аргумент да е `long`. В C++ съществува стандартна библиотека, която включва втора дефиниция на операцията `new`⁶. Нейният прототип изглежда така:

```
void *operator new( long size, void *memAddress );
```

Стойността на `size` се определя автоматично от компилатора. Останалите аргументи трябва да се изброят в разделен със запетайки списък, разположен между ключовата дума `new` и спецификатора на тип. Например:

```
#include <new.h>
```

⁵ В Borland C++ 2.0 типът на `size` е `unsigned int` — бел. прев

⁶ Не всички реализации поддържат втората реализация на операцията `new` — бел. прев

```

char buf[ sizeof(IntArray) ];

main() {
    // стандартната операция new
    IntArray *pa = new IntArray( 10 );

    // специалната операция new( long, void* )
    IntArray *pbuf = new (buf) IntArray( 10 );
}

```

4.4 Указател към функция

Искаме да създадем функция за сортиране на масив. В повечето случаи алгоритъмът на Хоар за бързо сортиране е подходящ. Потребителят трябва да запише само:

```
sort( array, lowBound, highBound );
```

В редки случаи бързото сортиране не е най-доброто решение на задачата. Ако масивът е почти подреден, методът на мехурчето е достатъчно добър. Ако мястото в паметта е малко, може да се използва пирамидално сортиране.

Би било добре, ако `sort()` позволява да се задава алтернативна функция за сортиране. Как да осигурим нужната гъвкавост, без излишно усложняване на `sort()`? Обикновено по подразбиране се използва най-често срещания случай. В случая по подразбиране ще използваме функцията `quickSort()`. Освен това искаме да заместим тази функция с друга, без да променяме кода на програмата. Настройването за всеки конкретен случай трябва да става при стартиране на програмата. Тези изисквания може да се осъществят чрез предаване на функция като параметър на друга функция. В случая сортиращата функция ще бъде аргумент на `sort()`. Известно е, че функциите не могат да се предават като параметри. Вместо тях се предават указатели към функции. В този раздел ще дефинираме `sort()` според посочените изисквания.

Тип на указател към функция

Какво представлява указателят към функция? Какъв е неговият тип? Как може да се декларира той? Да разгледаме декларацията на `quickSort()`:

```
void quickSort( int*, int, int );
```

При дефиниране на указател към функция, името на функцията не е от значение. Съществени са типът на резултата и сигнатурата на функцията. Когато дефинираме указател към `quickSort()`, трябва да посочим същия тип на резултата и същата сигнатура:

```
void *pf( int*, int, int );
```

Горната дефиниция изглежда коректна, но в действителност не е. Компиляторът я интерпретира като дефиниция на функцията `pf()` с три аргумента и тип на резултата `void*`. Операцията `"*"` се свързва с типа `void`, а не с `pf`. `pf` ще стане указател към функция, ако използваме скоби, а именно:

```
void (*pf)( int*, int, int );
```

Сега pf е указател към функция с три аргумента и тип на резултата void, т.е. може да се използва като указател към quickSort().

pf може да бъде указател и към други функции. Например:

```
void bubbleSort( int*, int, int );    // метод на мехурчето
void mergeSort( int*, int, int );    // сортиране чрез сливане
void heapSort( int*, int, int );    // пирамидално сортиране
```

pf не може бъде указател към следните функции:

```
int min( int*, int sz );
int max( int*, int sz );
```

Указател за тези функции се дефинира така:

```
int (*pfi)( int*, int );
```

Колкото възможни комбинации между тип на резултат и сигнатура съществуват, толкова са и различните типове указатели към функции. Функционалният тип е комбинация между типа на резултата и сигнатурата на функция.

Инициализация и присвояване

Знаем, че името на масив представлява указател към първия му елемент. Аналогично, името на функция е указател към функция от този тип. Например,

```
quickSort
```

е указател без име от тип:

```
void (*)( int*, int, int );
```

При прилагане на операцията "&" към името на функция, отново получаваме указател към същия функционален тип. Следователно, quickSort и &quickSort представляват едно и също нещо. Указател към функция се инициализира така:

```
void (*pfv)(int*, int, int ) = quickSort;
void (*pfv2)(int*, int, int ) = pfv;
```

Присвояването е аналогично:

```
pfv = quickSort;
pfv2 = pfv;
```

Инициализирането и присвояването са коректни, когато функционалните типове съвпадат. В противен случай при компилация се получава съобщение за грешка. Ето няколко примера:

```
extern int min( int*, int sz );
extern void (*pfv)(int*, int, int ) = 0;
extern int (*pfi)( int*, int ) = 0;

main() {
    pfi = min;      // правилно
    pfv = min;      // грешка
    pfv = pfi;      // грешка
}
```

Указател към функция може да се инициализира с 0. Аналогично, 0 може да се присвои на всеки указател към функция.

Нека имаме няколко функции с еднакво име. За всяка от тях може да се дефинира указател. За коя функция се отнася даден указател, компилаторът определя като търси точно съответствие на функционалните типове. Ако няма съответствие, се появява съобщение за грешка. Например:

```
extern void ff( char );
extern void ff( unsigned );

// точно съответствие с void ff(char)
void (*pf)(char) = ff;

// грешка: липсва точно съответствие
void (*pf2)(int) = ff;
```

Обръщение към функция чрез указател към нея

За да извикаме функция чрез указател към нея, не е необходимо да използваме операцията "*". Директното и индиректното обръщение се записват еднакво. Например:

```
#include <stream.h>
extern min( int*, int );
int (*pf)( int*, int ) = min;

const int iaSize = 5;
int ia[ iaSize ] = { 7, 4, 9, 2, 5 };

main() {
    cout << "Директно обръщение към min: "
          << min( ia, iaSize ) << "\n";
    cout << "Индиректно обръщение към min: "
          << pf( ia, iaSize ) << "\n";
}

min( int* ia, int sz )
{
    int minVal = ia[ 0 ];
    for ( int i = 0; i < sz; ++i )
        if ( minVal > ia[ i ] )
            minVal = ia[ i ];
    return minVal;
}
```

```
}
```

Обръщението

```
pf( ia, iaSize );
```

е по-прост запис на явното обръщение чрез указател:

```
(*pf)( ia, iaSize );
```

Двата записа са еквивалентни.

Масив от указатели към функции

Масив от указатели към функции се дефинира по следния начин:

```
int (*testCases[10])();
```

testCases е масив от 10 елемента. Всеки елемент е указател към функция без аргументи, която връща резултат от тип int. Към елементите на testCases се обръщаме така:

```
extern const SIZE = 10;
extern int (*testCases[SIZE])();
extern int testResults[SIZE];

void runtests() {
    for ( int i = 0; i < SIZE; ++i )
        testResults[ i ] = testCases[ i ];
}
```

Следващият пример показва как се инициализира масив от указатели към функции:

```
extern void quickSort( int*, int, int );
extern void mergeSort( int*, int, int );
extern void heapSort( int*, int, int );
extern void bubbleSort( int*, int, int );

void ( *sortFuncs[] )( int*, int, int ) =
{
    quickSort,
    mergeSort,
    heapSort,
    bubbleSort
};
```

Указател към sortFuncs се дефинира така:

```
void ( **pfSort )( int*, int, int ) = sortFuncs;
```

pfSort е указател към указател или с други думи указател към масив от указатели. Операторът

```
*pfSort;
```

дава адреса на sortFuncs, докато операторът

```
**pfSort;
```

дава адреса на първия елемент на sortFuncs, т.е. на quickSort(). Последният оператор е еквивалентен на:

```
*pfSort[ 0 ];
```

quickSort() може да се извика чрез pfSort така:

```
// еквивалентни записи
pfSort[ 0 ]( ia, 0, iaSize-1);      // съкратен запис
(*pfSort[ 0 ])( ia, 0, iaSize-1);  // пълен запис
```

Указателят към функция като аргумент и резултат от функция

Указател към функция може да се предаде като аргумент на друга функция. Той може да получи и стойност по подразбиране. Да разгледаме функцията sort():

```
extern void quickSort( int*, int, int );
void sort( int*, int, int,
           void (*)( int*, int, int ) = quickSort );
```

Ще използваме следната дефиниция на sort():

```
void sort( int* ia, int low, int high, void (*pf)( int*, int, int ))
{
    if ( !ia ) return;
    if ( !pf ) return;
    if ( high < low + 2 ) return;
    pf( ia, low, high );
}
```

sort() може да се извика по един от следните начини:

```
// обикновено декларациите са в заглавен файл
extern int *ia;
extern const iaSize;
extern void quickSort( int*, int, int );
extern void bubbleSort( int*, int, int );

typedef void (*PFV)( int*, int, int );
```



```
extern void sort( int*, int, int, PFV=quickSort );
extern void setSortPointer( PFV& );

PFV mySort;
void ff() {
    sort( ia, 0, iaSize );
    sort( ia, 0, iaSize, bubbleSort );
    setSortPointer( mySort );
    sort( ia, 0, iaSize, mySort );
}
```

Една функция може да върне като резултат указател към друга функция. Например,

```
int ( *ff(int))( int*, int );
```

декларира ff() като функция с един аргумент от тип int, която връща като резултат указател към функция от тип:

```
int (*)( int*, int );
```

За да опростим горния запис, можем да използваме описанието typedef. Например,

```
typedef int (*PFI)( int*, int );
PFI ff(int);
```

е еквивалентна декларация на ff().

Упражнение 4-9. В раздел 3.9 дефинирахме функция fibonaccі(). Дефинирайте указател към тази функция. Извикайте функцията чрез този указател, за да генерирате редица на Фибоначи с 9 елемента.

Упражнение 4-10. В раздел 3.10 дефинирахме функция binSearch(). Дефинирайте функцията search() така, че да е възможно обръщението:

```
extern int size, val, ia[];
int index = search( ia, size, val, binSearch );
```

_new_handler

_new_handler представлява указател към функция. По подразбиране той е инициализиран с 0. Декларацията

```
void ( *_new_handler )();
```

показва, че _new_handler е указател към функция без параметри, която връща резултат от тип void.

Ако при операция new наличната памет не достига, се проверява дали _new_handler адресира някаква функция. Ако това е така, се изпълнява адресираната от _new_handler функция. В противен случай new връща 0.

Ако потребителят иска да управлява паметта чрез `_new_handler`, трябва да дефинира подходяща функция и да присвои нейния адрес на този указател. `_new_handler` може да се инициализира така:

```
// ще се извика, ако не достига памет
extern void freeStoreException()

// _new_handler получава адреса на freeStoreException
_new_handler = freeStoreException;
```

За инициализиране може да се използва и стандартната функция `set_new_handler()`:

```
// декларацията на set_new_handler() е в new.h
#include <new.h>

// set_new_handler() инициализира _new_handler
set_new_handler( freeStoreException );
```

Функцията, която се адресира от `_new_handler`, се извиква, само ако не достига динамична памет. Ако свържем `_new_handler` с конкретна функция, не е необходимо да проверяваме дали има достатъчно памет за поредната операция `new`.

Една проста дефиниция на функцията `freeStoreException()` изглежда така:

```
#include <stream.h>
#include <stdlib.h>

extern char *progName; // името на текущия файл
enum Exceptions { FS_EXHAUST = 1, /*...*/ };

void freeStoreException() {
    cerr << "Динамичната памет не стига!\n";
    // освобождаване на памет ...
    exit( FS_EXHAUST );
}
```

`freeStoreException()` отпечатва съобщение за грешка и прекратява изпълнението на програмата. Преди завършване на програмата може да се освободи памет.

Упражнение 4-10. Инициализирайте `_new_handler` с адреса на `freeStoreException()`. Стартирайте програмата `exhaustFreeStore()` от раздел 4.1.

4.5 Особености при свързване на програмни модули

Езикът C++ позволява да се дефинират функции с едно и също име. Така се намалява лексическата сложност на програмата. При компилация на програмата се преминава през няколко етапа. Етапите, които работят с преработка на изходния текст, искат всяко име на нестатична функция да бъде уникално. Ако, например, свързващият редактор открие две или повече функции с едно и също име, той не може да ги различи. На този етап на компилация липсва информация, по която да се различат функции с еднакви имена. За да се преодолее този проблем, всяка функция се кодира с уникален вътрешен код. Свързващият редактор работи с тези вътрешни кодове. Как се създават вътрешните кодове не е важно, още повече,

че всяка версия на компилатора го прави различно. Съществуват два случая, когато при свързване, кодираните функции предизвикват грешка:

1. При несъвместимост на декларациите на една и съща функция в два различни файла.
 2. При обръщение към функции, написани на друг език за програмиране, най-често C.
- Настоящият раздел разглежда тези два случая.

Несъвместими декларации

Случва се една и съща функция да се декларира различно в два различни файла. В следващия пример `addToken()` е декларирана различно във файловете `token.C` и `lex.C`. В първия файл нейният аргумент е от тип `unsigned char`, а във втория — от тип `char`.

```
// във файла token.C
addToken( unsigned char tok ) { /*...*/ }

// във файла lex.C
extern addToken( char );
```

Обръщението към `addToken()` във файла `lex.C` предизвиква грешка при свързване. Аргументите от тип `unsigned char` и тип `char` се кодират различно. Декларираната в `lex.C` функция `addToken()`, се обявява за недефинирана. Програмата може да не работи, дори ако се генерира изпълним файл.

Компилираната програма е тествана на AT&T 3B20 и работи коректно, но на VAX 8550 се получава препълване. Дори най-простият тестов пример не работи. Каква е причината? Нека имаме следната декларация на `Tokens`:

```
enum Tokens {
    // ...
    INLINE = 128;
    VIRTUAL = 129;
    // ...
};
```

Да извикаме `addToken()` със следния аргумент:

```
curTok = INLINE;
addToken( curTok );
```

На VAX 8550 всяка променлива от тип `char` има знаков бит, а на AT&T 3B20 е без знаков бит. Следователно, грешката в декларациите няма да се прояви при 3B20. При 8550 всеки фактически аргумент със стойност по-голяма от 127, ще предизвика препълване.

При разделна компилация несъответствието в декларациите на функциите не се забелязва и може да стане причина за сериозна грешка при изпълнение на програмата. Вътрешното кодиране на функциите предотвратява такива грешки. Те се откриват при свързване на обектните кодове.

Неправилните декларации на външни променливи не могат да се уловят при компилация. Грешки като тази в следващия пример най-често сами се издават, като предизвикват неочакван резултат или странно поведение на програмата:

```
// в token.C
unsigned char lastTok = 0;

// в lex.C
extern char lastTok;
```

Отново за заглавните файлове

Използването на заглавни файлове е основен принцип в програмирането на C++. Досадните грешки от предишния подраздел могат да се избегнат, ако всички декларации се обединят в един или няколко заглавни файла. Заглавният файл може да включва декларации на външни променливи, прототипи на функции, дефиниции на класове и inline-функции. Файловете, които използват тези декларации, трябва само да включат съответния заглавен файл.

Заглавните файлове имат две преимущества:

1. Гарантират употребата на едни и същи декларации във всички файлове на програмата.
2. Промяната на съществуваща декларация се извършва само на едно място. Изключва се възможността в някой файл тази декларация да остане непроменена.

За да предотвратим грешките в горния пример, ще създадем заглавен файл с име token.h:

```
// в token.h
enum Tokens { /* ... */ };
extern unsigned char lastTok;
extern addToken( unsigned char );

// в lex.C
#include "token.h"

// в token.C
#include "token.h"
```

При създаване на заглавен файл, трябва да се спазват някои препоръки. Първо, файлът трябва да включва само логически свързани декларации. Всеки включен в програмата заглавен файл изисква време за компилация. Ако той е прекалено дълъг или съдържа излишни декларации, ще се компилира ненужно много време. Следователно, преди да използваме един заглавен файл, трябва да го изчистим от ненужните декларации.

Второ, заглавните файлове не могат да съдържат дефиниции на нестатични функции. Нека два файла на програмата включват заглавен файл с дефинирана в него външна функция. Повечето свързващи редактори "изхвърлят" програмата със съобщение за грешка, защото едно и също име е дефинирано два пъти. Заглавните файлове често включват и различни константи. Всяка константа по подразбиране е статична, затова не създава проблеми при свързване.

Обръщения към функции, написани на друг програмен език

Ако искаме да използваме функция, написана на друг език за програмиране, например C, трябва да използваме т.нар. механизъм за освобождаване от вътрешно кодиране. Той се реализира чрез директива за свързване (linkage directive). Например:

```
extern "C" void exit(int);

extern "C" {
    printf( const char* ... );
    scanf( const char* ... );
}

extern "C" {
    #include <string.h>
}
```

Директивата за свързване започва с ключовата дума `extern`, следвана от символен низ и произволен прототип на функция. Независимо от това, че функцията е написана на друг език за програмиране, при обръщение към нея се прави пълен контрол на типовете. Ако функциите са няколко, може да се използват фигурни скоби. Те служат за разделители и не дефинират нова област на действие.

Директивата за свързване може да стои само в глобалната област на действие. Следващият програмен фрагмент предизвиква грешка при компилация:

```
char *copy( char *src, char *dst ) {
    // грешка: директивата за свързване не може да е тук
    extern "C" strlen( const char* );
    if ( !dst )
        dst = new char[ strlen(src) + 1 ];
    // ...
    return dst;
}
```

След преместване на директивата в глобалната област на действие, функцията се компилира без проблем:

```
extern "C" strlen( const char* );

char *copy( char *src, char *dst )
{
    if ( !dst )
        dst = new char[ strlen(src) + 1 ];
    // ...
    return dst;
}
```

Директивата за свързване обикновено се включва в заглавен файл. Механизмът за освобождаване от вътрешно кодиране се прилага и към функции, написани на езика C++, предназначени за използване в програма на друг език.

Когато има функции с едно и също име, директивата за свързване може да се използва само с една от тях. Затова следващият програмен фрагмент е некоректен:

```
// в string.h
extern "C" strlen( const char* );

// в String.h
extern "C" strlen( const String& );
```

Следващият пример показва типичната употреба на директивата за свързване:

```
class Complex;
class BigNum;

extern Complex& sqrt( Complex& );
extern "C" double sqrt( double );
extern BigNum& sqrt( BigNum& );
```

Функцията sqrt(), написана на езика C е освободена от вътрешно кодиране, за разлика от останалите функции със същото име. Редът на декларациите не е от значение.

Упражнение 4-12. exit(), printf(), malloc(), strcpy() и strlen() са функции от библиотеката на C. Модифицирайте следващата програма на C, за да я компилирате с компилатор на C++.

```
char *str = "hello";

main()
{ /* програмата е написана на C */
  char *s, *malloc(), *strcpy();

  s = malloc( strlen(str)+1 );
  strcpy( s, str );
  printf( "%s, world\n", s );
  exit( 0 );
}
```

Упражнение 4-13. В раздел 3.10 дефинирахме функцията binSearch(). Как можем да я пригодим за използване в програма на C?

Упражнение 4-14. В math.lib на езика C са дефинирани следните функции:

```
double sin( double );
double cos( double );
double tan( double );
```

Какви промени са необходими, за да се декларират същите функции за класовете RealNum и Complex?

ГЛАВА 5

Класовете представляват нови типове данни, дефинирани от потребителя. Тези типове обогатяват възможностите на съществуващ тип, като дефинирания в глава 1 клас `IntArray` или представляват напълно нов тип данни — например класът на комплексните числа, `Complex`. Обикновено класовете описват абстракция, която не може да се изрази чрез стандартните и производните типове данни. В следващите четири глави са дадени много примери на класове.

Класовете притежават четири важни атрибута:

1. Включват съвкупност от членове, които описват класа. Ще наричаме тези членове членове-данни (`data members`) или само данни на класа. Техният брой може да е 0. Типът им е произволен.
2. Включват съвкупност от членове, които обработват обектите от класа. Ще наричаме тези членове функции-членове (`member functions`), методи на класа или интерфейс на класа. Техният брой също може да бъде 0.
3. Всеки член на класа притежава ниво на достъп в програмата. Нивото се задава чрез ключовите думи `private`, `protected` и `public`. Обикновено описанието на класа се намира в секцията с име `private`, а операциите над обектите — в секцията с име `public`. Това разпределение на членовете е известно като скриване на информация (`information hiding`). Поместването на данните на класа в секцията `private` се нарича капсулиране на данните.
4. Името на класа може да се използва като спецификатор на тип. Ако сме дефинирали клас `Screen`, следващите оператори са коректни:

```
Screen myScreen;  
Screen *tmpScreen = &myScreen;  
Screen& copy( const Screen[] );
```

Клас, чието описание е в секцията `private`, а операциите с обектите са в секцията `public`, се нарича абстрактен тип данни (`abstract data type`). В тази и следващата глава ще се запознаем с проектирането, реализирането и използването на абстрактните типове данни.

5.1 Дефиниране на класове

Дефиницията на клас се състои от две части: заглавна част (`class head`) и тяло (`class body`). Заглавната част включва ключовата дума `class` и името на класа. Тялото е заградено във фигурни скоби. След скобите стои символът `;"` или списък от имена на променливи. Например:

```
class Screen { /* ... */ };  
class Screen { /* ... */ } myScreen, yourScreen;
```

В тялото на класа се описват членовете на класа със съответните нива на достъп.

Членове-данни

Членовете, които описват класа, се дефинират като обикновени променливи, но не могат да се инициализират явно. Ще дефинираме клас `Screen`:

```

class Screen {
    short height;    // брой редове
    short width;     // брой колони
    char *cursor;    // текуща позиция на курсора на екрана
    char *screen;    // масив за елементите на екрана
};

```

Членовете от един и същи тип могат да се изброят на един ред. Следващата дефиниция използва тази възможност. Тя е еквивалентна на горната:

```

class Screen {
/*
 * height и width се отнасят за реда и колоната
 * cursor посочва текущата точка от екрана
 * screen адресира масив от точките на екрана
 */
    short height, width;
    char *cursor, *screen;
};

```

Членовете на класа трябва да се декларират в нарастващ ред, ако няма причина да се направи друго. Под нарастващ ред се разбира нарастване на броя байтове за представяне в паметта. Така за повечето компютри се получава оптимално подравняване на членовете в паметта. Типът на членовете може да бъде произволен. Например:

```

class StackScreen {
    int topStack;
    void (*handler());    // управлява изключителните ситуации
    Screen stack[ STACK_SIZE ];
};

```

Името на клас може да се използва като тип на член на друг клас, ако първият клас е дефиниран преди втория. Това означава, че дефиницията на клас StackScreen е коректна, само ако клас Screen е дефиниран по-рано. Ако първият клас не може да се дефинира преди втория, се използва т.нар. предварително деклариране (forward declaration). То позволява в дефиницията на втория клас да се включат членове, които са указатели или алтернативни имена (reference) на обекти от първия клас. Указатели и алтернативни имена могат да се използват, защото те имат фиксиран размер, независимо от типа на променливата, която адресират. Ето как изглежда дефиницията на StackScreen при предварително деклариране на клас Screen:

```

class Screen;    // предварително деклариране на клас Screen
class StackScreen {
    int topStack;
    Screen *stack;    // указател към масив от обекти от клас Screen
    void (*handler());
};

```

Член на класа не може да бъде обект от същия клас. Но указател или алтернативно име на обект от класа могат да бъдат негови членове. Например:


```
class LinkScreen {
    Screen window;
    LinkScreen *next;
    LinkScreen *prev;
};
```

Функции-членове на клас

Потребителите на клас Screen ще имат нужда от множество операции за обектите от класа. Ще са необходими функции за движение на курсора по екрана. Полезни ще са и функциите за проверка и инициализиране на позицията на курсора. Може да се наложи един екран да се копира в друг, да се определят действителните размери на текущия екран и др. Споменатите операции трябва да се декларират в тялото на класа. Операциите с обектите се наричат функции-членове или методи на класа.

Декларацията на функция включва нейния прототип:

```
class Screen {
public:
    void home();
    void move( int, int );
    char get();
    char get( int, int );
    void checkRange( int, int );
    // ...
};
```

Дефинициите на функциите могат да бъдат в тялото на класа:

```
class Screen {
public:
    void home() { cursor = screen; }
    char get(); { return *cursor; }
    // ...
};
```

home() позиционира курсора в горния ляв ъгъл на екрана. get() връща текущата позиция на курсора. Когато функциите са дефинирани в тялото на класа, те се възприемат като inline-функции.

Функции, които се състоят от повече от два оператора, е добре да се дефинират извън тялото на класа. В този случай се използва специален синтаксис, който изразява принадлежност на функцията към даден клас. Да разгледаме дефиницията на checkRange():

```
#include "Screen.h"
#include <stream.h>
#include <stdlib.h>

void Screen::checkRange( int row, int col )
{ // проверка за валидност на координатите
    if ( row < 1 || row > height || col < 1 || col > width )
    {
        cerr << " Координатите ( "
              << row << ", " << col
              << " ) са извън екрана.\n";
    }
}
```

```

        exit( -1 );
    }
}

```

Функция, дефинирана извън тялото на класа, може да стане `inline`, ако това се посочи явно. Например:

```

inline void Screen::move( int r, int c )
{ // премества курсора на абсолютни координати
    checkRange( r, c );           // Валидни ли са координатите?
    int row = ( r - 1 ) * width;
    cursor = screen + row + c - 1;
}

```

Функциите-членове на клас се различават от обикновените функции по следните атрибути:

- Първите имат достъп до всички членове на своя клас, а вторите — само до членовете от секция `public`. В общия случай функциите-членове на един клас нямат привилегии за достъп до членовете на друг клас.
- Функциите-членове се дефинират в областта на действие на своя клас, докато обикновените функции се дефинират в глобалната област на действие. Раздел 5.8 разглежда подробно областите на действие за класове.

Няколко функции-членове на един и същи клас могат да имат еднакво име. Функция със същото име, която не принадлежи на класа, е невидима за обектите от този клас. Можем да дефинираме втора функция `get()`, член на клас `Screen`, която няма отношение с глобална функция `get()` или функция-член на друг клас, носеща същото име.

```

inline char Screen::get( int r, int c )
{
    move( r, c );    // позиционира курсора
    return get();    // извиква другата функция get()
}

```

Скриване на информация

В процеса на работа вътрешното представяне на класа може да се променя. Ако всички екрани, с които се работи, са с размери 24 x 80, може да се използва по-малко гъвкаво, но по-ефективно представяне на клас `Screen`:

```

const Height = 24;
const Width = 80;

class Screen {
    char screen[ Height ][ Width ];
    short cursor[2];
};

```

Новата дефиниция на `Screen` изисква промени във всички функции-членове на този клас, но техните сигнатури и типове на резултата остават същите. Как ще се отрази промяната във вътрешното представяне на класа върху потребителя на този клас?

- Всяка програма, която използва директен достъп до данните на класа, ще откаже да работи. Ще трябва да се открият и поправят всички места в кода, които предизвикват грешки.
- Всяка програма, която използва само методите на класа, не изисква поправки в кода. Необходима е само повторна компилация на програмата.

Механизмът, който ограничава достъпа до членовете на класа, се нарича скриване на информация. Членовете на класа се разделят на три групи: `public`, `private` и `protected`.

- Общодостъпните членове (`public`) се описват в едноименната секция. Класовете, които скриват информация, изискват в тази секция да има само методи на класа.
- Защитените членове (`protected`) са общодостъпни членове (`public`) за всеки произведен клас на дадения. За останалата част на програмата те са скрити членове (`private`). Такъв член използвахме в дефиницията на клас `IntArrayRC` в глава 1. В глава 7 ще разгледаме подробно този тип членове, заедно с механизма на наследяване и дефинирането на производни класове.
- Скритите членове (`private`) са достъпни само за методите на класа и неговите "приятели" (функции и класове). "Приятелските" функции и класове се обсъждат в раздел 5.5. Класовете, които скриват информация, дефинират своите данни в секция `private`.

Следващата дефиниция на `Screen` скрива информация:

```
class Screen {
public:
    void home() { move( 0, 0 ); }
    char get() { return *cursor; }
    char get( int, int );
    inline void move( int, int );
    // ...
private:
    short height, width;
    char *cursor, *screen;
};
```

Ако секция `public` съществува, тя трябва да бъде първа. Секция `private` трябва да е последната секция в тялото на класа.

Тялото на класа може да съдържа няколко секции от един и същ тип. Ако няма явно посочена секция, по подразбиране всички членове на класа са скрити (`private`).

5.2 Обекти

Дефиницията на клас `Screen` не заделя памет за него. Памет се заделя едва при дефинирането на обект от класа. Дефиницията

```
Screen myScreen;
```

заделя памет за четирите члена на `Screen`, които описват класа.

Обекти от един и същи клас могат да се присвояват един на друг. Всеки от тях може да се инициализира с друг обект от класа. При копиране на обект от даден клас, се копират всички негови членове-данни. Например:

```
// bufScreen.height = myScreen.height
// bufScreen.width = myScreen.width
// bufScreen.cursor = myScreen.cursor
// bufScreen.screen = myScreen.screen
Screen bufScreen = myScreen;
```

По подразбиране обектите от произволен клас се предават по стойност, когато са аргументи на функция или резултат от нея. Указател към обект се инициализира или чрез операция new, или чрез операция "&". Например:

```
Screen *ptr = new Screen;
myScreen = *ptr;
ptr = &bufScreen;
```

Извън областта на действие на даден клас, неговите членове са достъпни само чрез операциите за избор на член. Операцията "." се използва с обект или алтернативно име (reference) на обект, а операцията "—>" — с указател към обект. Да разгледаме един пример:

```
#include "Screen.h"

isEqual( Screen& s1, Screen *s2 )
{ // ако обектите не са еднакви, връща 0; иначе връща 1
  if ( s1.getHeight() != s2—>getHeight() || s1.getWidth() != s2—>getWidth() )
    return 0; // не са еднакви
  for ( int i = 0; i < s1.getHeight(); ++i )
  for ( int j = 0; j < s2—>getWidth(); ++j )
    if ( s1.get( i, j ) != s2—>get( i, j ) )
      return 0; // не са еднакви
  // ако стигнем дотук, обектите са еднакви
  return 1;
}
```

isEqual() не е член на клас Screen. Тя определя дали два обекта от класа са еднакви. isEqual() няма достъп до членовете, които описват класа, защото те са в секция private. Функцията може да използва само общодостъпните членове на Screen.

getHeight() и getWidth() са членове на клас Screen. Те използват данните, чрез които се представя класа, но не ги променят. Реализацията им е проста. За по-голяма ефективност ще ги дефинираме като inline-функции:

```
class Screen {
public:
  int getHeight() { return height; }
  int getWidth(); { return width; }
  // ...
private:
  short height, width;
  // ...
};
```

Членовете на класа са достъпни директно в неговата област на действие. Всяка функция-член на класа принадлежи на неговата област на действие, независимо къде е

дефинирана — в тялото на класа или извън него. Всеки метод на класа е достъпен за другите методи от този клас директно. Да разгледаме следния пример:

```
#include "String.h"
#include <string.h>

void Screen::copy( Screen& s )
{ // копира един обект от клас Screen в друг
  delete screen; // освобождава заета памет
  height = s.height;
  width = s.width;

  screen = cursor = new char[ height * width + 1 ];
  strcpy( screen, s.screen );
}
```

Упражнение 5-1. copy() изравнява размерите на двата обекта от клас Screen. Дефинирайте отново тази функция, за да позволите на обектите да имат различни размери.

Упражнение 5-2. За да се използва стандартната функция strcpy(), е направено интересно, но и опасно предположение за символите, които се съдържат в screen. Какво е това предположение? Променете дефиницията на copy(), така че тя да не зависи от това предположение.

5.3 Функции-членове на клас

Функциите-членове на даден клас осигуряват множество операции за обектите от този клас. Операциите се наричат общодостъпен интерфейс (public interface) на класа. Дали класът е добре конструиран, зависи от пълнотата и ефективността на неговите методи. Този раздел е посветен на методите на клас Screen. Той е разделен на четири подраздела. Всеки от тях разглежда специален тип методи. Методите условно се делят на: управляващи, инструментални, помощни методи и методи за достъп. Разделението не е част от езика C++. По-скоро то е начин на мислене при проектирането на класове.

Управляващи методи

Управляващи са методите на класа, които инициализират и присвояват обекти, преобразуват типове и управляват паметта. Обикновено те се извикват автоматично от компилатора.

Функциите, които инициализират обектите, се наричат конструктори. Конструкторите се извикват автоматично от компилатора при всяко дефиниране на обект или заделяне на памет за него чрез операцията new. Името на конструктора трябва да съвпада с името на класа. Ето дефиницията на конструктора на клас Screen:

```
Screen::Screen( int high, int wid, char bkground )
{ // конструктор на клас Screen
  int sz = high * wid;
  height = high; width = wid;
  cursor = screen = new char[ sz + 1 ];

  char *ptr = screen;
  char *endptr = screen + sz;
  while ( ptr != endptr ) *ptr++ = bkground;
  *ptr = '\0'; // края на screen се маркира със символа null
}
```

Декларацията на конструктора в тялото на класа осигурява стойности по подразбиране за трите аргумента high, wid и bkground:

```
class Screen {
public:
    Screen( int=8, int=40, char='#' );
    // ...
};
```

При всяко дефиниране на обект, конструкторът на класа осигурява неговото автоматично инициализиране:

```
Screen s1;                // Screen(8, 40, '#')
Screen *ps = new Screen( 20 ); // Screen(20, 40, '#')
main() {
    Screen s(24, 80, '*'); // Screen(24, 80, '*')
    // ...
}
```

Глава 6 разглежда конструкторите и другите управляващи методи.

Инструментални методи

Инструменталните методи реализират потенциалните възможности на класа. Например, от обектите на клас Screen, се очаква да поддържат движение на курсора. Движението може да бъде до конкретна точка на екрана — функция move() — или напред/назад, нагоре/надолу — съответно функции forward(), back(), up(), down(). Да дефинираме тези функции:

```
inline void Screen::forward()
{
    // премества курсора с една позиция напред
    ++cursor;
    // проверява дали не сме в последната точка на екрана
    if ( *cursor == '\0' ) home();
}
```

forward() премества курсора с една позиция напред. Ако сме достигнали последната точка от екрана, курсорът се позиционира в горния ляв ъгъл.

```
inline void Screen::back()
{ //премества курсора с една позиция назад
    // проверява дали не сме в началото на екрана
    if ( cursor == screen ) bottom();
    else --cursor;
}
```

bottom() премества курсора в края на екрана:

```
inline void Screen::bottom()
```

```
{
    int sz = width * height - 1;
    cursor = screen + sz;
}
```

up() и down() преместват курсора съответно с една позиция нагоре или надолу. При достигане съответно на първия или последния ред, курсорът не се премества. В този случай се издава предупредителен сигнал.

```
const char BELL = "\007";

void inline Screen::up()
{ // премества курсора с един ред нагоре
    if ( row() == 1 )           // на първия ред
        cout.put( BELL );
    else
        cursor -= width;
}

void inline Screen::down()
{ // премества курсора с един ред надолу
    if ( row() == height )      // на последния ред
        cout.put( BELL );
    else
        cursor += width;
}
```

Упражнение 5-3. Напишете функциите wordForward() и wordBack(), които преместват курсора съответно една дума напред или назад, при условие, че думите са разделени с интервали.

Упражнение 5-4. Дефинирайте функция find(), която позиционира курсора в началото на търсена дума, например:

```
myScreen.find( "this" );
```

Помощни методи

Помощни са методите, които реализират спомагателни действия. Обикновено те не се използват директно от потребителя, а от другите функции-членове на класа. В общия случай помощните методи се декларират в секция private. В предишните подраздели дефинирахме една помощна функция — checkRange(). Сега ще дефинираме още четири помощни функции. Първата връща реда, на който се намира курсора:

```
inline Screen::row()
{ // връща текущия ред
    int pos = cursor - screen + 1;
    return (pos + width - 1) / width;
}
```

col() връща колоната, на която е курсора:

```
inline Screen::col()
{ // връща текущата колона
```

```

    int pos = cursor - screen + 1;
    return ((pos + width - 1) % width) + 1;
}

```

remainingSpace() връща броя на позициите, които са надясно от курсора. Позицията на курсора не се включва в този брой:

```

inline Screen::remainingSpace()
{ // позицията на курсора не се включва
    int sz = width * height;
    return( screen + sz - cursor - 1 );
}

```

stats() отпечатва информацията, която връщат предишните три функции.

```

void inline Screen::stats()
{
    cout << "ред: " << row() << "\t";
    cout << "колона: " << col() << "\t";
    cout << "останали позиции: " << remainingSpace() << "\n";
}

```

Следващата малка програма демонстрира действието на описаните помощни функции:

```

#include "Screen.h"
#include <stream.h>

main()
{ // демонстрира движението на курсора
    Screen x(3,3);
    int sz = x.getHeight() * x.getWidth();

    cout << "Обект от клас Screen ( "
        << x.getHeight() << ", " << x.getWidth();
        << " ) ( размер: " << sz << " )\n";

    x.home();
    for ( int i = 0; i <= sz; ++i )
    { // "<—" за да изпробваме преминаването от последната
      // позиция в началото на екрана
        x.stats();
        x.forward();
    }
    return 0;
}

```

След нейното компилиране и изпълнение получаваме следния резултат:

```

Обект от клас Screen ( 3, 3 ) ( размер: 9 )
ред: 1 колона: 1 останали позиции: 8
ред: 1 колона: 2 останали позиции: 7
ред: 1 колона: 3 останали позиции: 6
ред: 2 колона: 1 останали позиции: 5
ред: 2 колона: 2 останали позиции: 4

```



```

ред: 2 колона: 3 останали позиции: 3
ред: 3 колона: 1 останали позиции: 2
ред: 3 колона: 2 останали позиции: 1
ред: 3 колона: 3 останали позиции: 0
ред: 1 колона: 1 останали позиции: 8

```

Методи за достъп

Скриването на информация капсулира вътрешното представяне на класа, като по този начин го предпазва от неконтролирани промени. Данните на класа се променят от едно малко множество функции. При възникване на грешка, тя се търси само в тези функции. Така се улеснява поддръжката на програмите. Методите, които осигуряват потребителския достъп до скритите данни на класа, се наричат методи за достъп. Досега разгледаните методи за достъп на клас Screen четяха данните, но не ги променяха. Сега ще дефинираме две функции с име set(), които записват низ или отделен символ на екрана.

```

void Screen::set( char *s )
{ // записва на екрана низ, който започва от позицията на курсора
  int space = remainingSpace();
  int len = strlen( s );
  if ( space < len ) {
    cerr << "Символният низ ще бъде отсечен: "
          << "място: " << space
          << "дължина на низа: " << len << "\n";
    len = space;
  }
  for ( int i = 0; i < len; ++ i )
    *cursor++ = *s++;
}

void Screen::set( char ch ) {
  if ( ch == '\0' )
    cerr << "Символът null се игнорира.\n ";
  else *cursor = ch;
}

```

За простота допускаме, че обектите от клас Screen не съдържат символ null. Затова set(char) игнорира този символ.

Методите за достъп могат да разширят абстрактния тип данни с множество предикатни операции. Например, клас Screen, може да се разшири с набор от функции с общото име isEqual(). isEqual(char ch) връща истина, ако ch съвпада със символа от текущата позиция на курсора. Дефиницията на тази функция е проста:

```

class Screen {
public:
  isEqual( char ch ) { return (ch == *cursor); }
  // ...
};

```

isEqual(char* s) връща истина, ако от текущата позиция на курсора започва съвкупност от символи, която съвпада с низа s.

```
#include <string.h>
```

```
Screen::isEqual( char *s )
```

```

{ // при равенство връща 1, иначе връща 0
  int len = strlen( s );
  if ( remainingSpace() < len ) return 0;

  char *p = cursor;
  while ( len-- > 0 )
    if ( *p++ != *s++ ) return 0;

  return 1;
}

```

isEqual(Screen&) връща истина, ако два екрана са еднакви т.е. ако техните дължини, ширини и съдържания съвпадат.

```

Screen::isEqual( Screen& s )
{ // ако размерите са различни и екраните са различни
  if ( width != s.width || height != s.height )
    return 0;

  // Не са ли двата екрана един и същ обект?
  char *p = screen;
  char *q = s.screen;
  if ( p == q ) return 1;

  // трябва да внимаваме да не сравним елементи,
  // които са извън двата екрана
  while ( *p && *p++ == *q++ );
  if (*p) // цикълът завършва при откриване на различие
    return 0;
  return 1;
}

```

Упражнение 5-5. Сравнете горната дефиниция на функцията isEqual(Screen&) с тази от раздел 5.2. Защо те са толкова различни? Еквивалентни ли са двете функции?

Константни методи

Знаем, че всеки опит за промяна на стойността на константа от стандартен тип предизвиква грешка при компилация. Например:

```

const char blank = ' ';
blank = '\0';           // грешка

```

Обектите не се променят директно от програмиста. За целта той използва част от методите на класа. Ако дефинираме константен обект, трябва да укажем по някакъв начин кои функции са безопасни за него. Безопасни са функциите, които не променят данните на класа. Например:

```

const Screen blackScreen;

// безопасна операция
blackScreen.display();

// опасна операция
blackScreen.set( '*' );

```

При дефиниране на класа, трябва явно да се укаже кои функции са безопасни. За целта се използва ключовата дума `const`. Тя се поставя между списъка от аргументи и тялото на функцията. Например:

```
class Screen {
public:
    char get() const { return *cursor; }
    // ...
};
```

Константните обекти могат да се обръщат само към константните методи на класа. Ако това изискване е нарушено, някои компилатори предупреждават, но не отчитат грешка.

Методите, дефинирани извън тялото на класа, могат да се използват от константен обект, само ако съдържат ключовата дума `const` и декларацията, и в дефиницията си. Например:

```
class Screen {
public:
    isEqual( char ch ) const;
    // ...
private:
    char *cursor;
    // ...
};

Screen::isEqual( char ch ) const {
    return( ch == *cursor );
}
```

Функция, която променя данните на класа не може да се дефинира като константен метод. В следващата опростена дефиниция на клас `Screen`, `ok()` е дефинирана правилно като константен метод, защото не променя стойността на `cursor`. Тя променя стойността на адресирания от `cursor` обект. Програмистът трябва сам да прецени дали тази промяна е безопасна.

```
class Screen {
public:
    void ok( char ch ) const { *cursor = ch; }
    void error(char *pch ) const { cursor = pch; }
    // ...
private:
    char *cursor;
    // ...
};
```

`error()` променя стойността на `cursor` и следователно не може да бъде константен метод. Нейната декларация предизвиква следното съобщение за грешка:

```
error: assignment to member Screen::cursor of const class Screen
```

Константен и неконстантен метод от един същ клас и еднаква сигнатура могат да имат едно и също име. Например:

```
class Screen {
public:
    char get( int x, int y );
    char get( int x, int y ) const;
    // ...
};
```

Кой от двата метода ще бъде избран зависи от самия обект:

```
const Screen cs;
Screen s;

main() {
    char ch = cs.get( 0, 0 );    // втората функция
    ch = s.get( 0, 0 );         // първата функция
}
```

Конструкторите и деструкторите се различават от останалите методи на класа. Те не трябва да се дефинират като константни, за да се използват от константни обекти. Всеки клас със широка употреба е добре да притежава константни методи.

Упражнение 5-6. Определете кои методи на Screen могат да се дефинират като константни.

5.4 Указател this

Дефинираните досега методи на клас Screen притежават един недостатък. Обикновено с обектите от класа се извършва някаква последователност от действия: изчистване, преместване, инициализиране, изобразяване и т.н. Сега програмистът трябва да кодира всяко действие с отделен оператор:

```
Screen myScreen( 3, 3 ), bufScreen;

main() {
    myScreen.clear();
    myScreen.move(2,2);
    myScreen.set('*');
    myScreen.display();
    bufScreen.reSize(5,5);
    bufScreen.display();
}
```

Задаването на всяко действие с отделен оператор е ненужно многословие. За предпочитане е синтаксис, който позволява обединяване на операторите. Например:

```
main() {
    myScreen.clear().move(2,2).set('*').display();
    bufScreen.reSize(5,5).display();
}
```

В този раздел ще покажем как се реализира този по-удобен синтаксис. Нека още веднъж разгледаме функциите-членове.

Какво представлява указателят `this`

Всеки обект от даден клас поддържа свое копие на данните, които го описват. `myScreen` има свои `width`, `height`, `cursor` и `screen`. Същото е и с `bufScreen`. И двата обекта, обаче, извикват едно и също копие на съответната функция-член на класа. В паметта има единствено копие на всяка от функциите-членове. Този факт създава два проблема:

1. След като в паметта има само едно копие на функциите-членове, това означава, че те не се съхраняват на едно място с данните на класа. Тогава как се размножават тези функции за всеки дефиниран обект?
2. Ако съществува само едно копие на функциите-членове, как данните на отделните обекти се свързват с данните, с които работи съответната функция? Как например, функцията `move()`, която работи с `cursor`, го свързва последователно със съответния `cursor` на `myScreen` и `bufScreen`?

Отговор на тези въпроси дава специалният указател `this`. Всяка функция-член на класа съдържа указател с име `this`. За функциите-членове на `Screen` указател `this` е от тип `Screen*`, докато за функциите-членове на клас `IntList` той е от тип `IntList*`.

Указател `this` съдържа адреса на обекта, с който се извиква съответната функция. Така променливата `cursor`, с която работи функцията `home()`, се свързва с `cursor` на `myScreen` или `bufScreen`.

Ще разберем как се извършва това свързване, ако да разгледаме работата на компилатора на C++. Тя включва следните две стъпки:

1. Транслиране на съответната функция. Всяка функция-член на даден клас се транслира в обикновена функция с уникално име и един допълнителен аргумент — специалния указател `this`. Например:

```
home__Screen( Screen *this )
{
    this->cursor = this->screen;
}
```

2. Транслиране на всяко обръщение към тази функция. Например,

```
myScreen.home();
```

се транслира в

```
home__Screen( &myScreen );
```

Указател `this` може да се използва явно в кода на съответната функция, макар че е глупаво да се напише следното:

```
inline void Screen::home()
{
    this->cursor = this->screen;
```

```
}
```

Съществуват ситуации, когато явното използване на този указател е единственото решение на проблема.

Използване на указател **this**

Указател **this** позволява да се реализира компактният синтаксис, за който стана дума по-горе. Операциите за избор на член "." и "—>" са ляво асоциативни бинарни операции. Те се изпълняват отляво надясно. Най-напред се изпълнява `myScreen.clear()`. За да се изпълни следващата функция `move()`, `clear()` трябва да върне обекта `myScreen`. Всяка функция-член на клас `Screen` трябва да се промени, за да връща обекта, който я е извикал. Указател **this** осъществява достъпа до този обект. `clear()` трябва да се дефинира така:

```
Screen& Screen::clear( char bkground )
{ // позиционира cursor в началото на екрана, след което изчиства екрана
  char *p = cursor = screen;
  while ( *p ) *p++ = bkground;

  return *this;    // връща обекта, който извиква този метод
}
```

В дефинициите на `move()`, `home()`, `set()` и четирите функции за движение на курсора също трябва да се добави оператора

```
return *this;
```

Трябва да се промени и типът на резултата им — от `void` в `Screen&`.

Да разгледаме как могат да се използват тези функции — най-напред от други функции-членове на `Screen` и след това от външни за класа функции.

```
Screen& Screen::lineX( int row, int col, int len, char ch )
{ /* изписва права линия на ред row с дължина len;
   * започва от позиция col и използва символа ch */

  move( row, col );
  for ( int i = 0; i < len; ++i ) set(ch).forward();
  return *this;
}
```

Следващата функция е външна за клас `Screen`. Тя чертае вертикална линия с определена дължина в определена колона от екрана, като използва символа `ch`:

```
Screen& lineY( Screen& s, int row, int col, int len, char ch )
{ // изчертава вертикална линия
  s.move( row, col );
  for ( int i = 0; i < len; ++i ) s.set(ch).down();
  return s;
}
```

Упражнение 5-7. Употребата на указател `this` опростява записите. За да усвоите неговото използване, дефинирайте `lineY()` като член на клас `Screen`.

Като използваме указател `this`, нека дефинираме функцията `display()`:

```
Screen& Screen::display()
{
    char *p;
    for ( int i = 0; i < height; ++i )
    { // за всеки ред
        cout << "\n";
        int offset = width * i; // отместване по редове
        for ( int j = 0; j < width; ++j )
        { // отпечатва текущия ред
            p = screen + offset + j;
            cout.put( *p );
        }
    }
    return *this;
}
```

Функцията може да променя адресирания от указател `this` обект. Такава функция е `reSize()`. Тя създава нов обект от клас `Screen`. Присвояването

```
*this = *ps;
```

замества извиквания обект с новия обект. `ps` е указателя към новия обект. Ето и дефиницията на тази функция:

```
Screen& Screen::reSize( int h, int w, char bkground )
{ // променя размерите на екрана

    Screen *ps = new Screen(h, w, bkground);
    char *pNew = ps->screen;
    // ако има памет за новия екран, съдържанието на стария екран се копира в новия
    if ( screen )
    {
        char *pOld = screen;
        while ( *pOld && *pNew ) *pNew++ = *pOld++;
        delete screen;
    }

    *this = *ps;      // замества извиквания с новия обект
    return *this;
}
```

Работата със свързани списъци също може да наложи използване на указател `this`. Например:

```
class Dlist {          // двойно свързан списък
public:
    void append( Dlist* );
    // ...
}
```

```
private:
    Dlist *prior, *next;
};

void Dlist::append( Dlist *ptr )
{
    ptr->next = next;
    ptr->prior = this;
    next->prior = ptr;
    next = ptr;
}
```

Следващата малка програма илюстрира действието на някои функции-членове на клас Screen:

```
#include "Screen.h"

main() {
    Screen x(3,3);
    Screen y(3,3);

    // ако двата екрана са еднакви, връща 1
    cout << "Обектите са еднакви: (>1<) " << x.isEqual(y) << "\n";
    y.resize(6,6); // удвояване на размерите
    cout << "Обектите не са еднакви: (>0<) " << x.isEqual(y) << "\n";

    // чертае вертикални линии
    lineY( y, 1, 1, 6, '*' ); lineY( y, 1, 6, 6, '*' );

    // чертае хоризонтални линии
    y.lineX(1, 2, 4, '*').lineX(6, 2, 4, '*').move(3,3);

    // писане по екрана и отпечатване
    y.set("hi").lineX(4, 3, 2, '^').display();

    // x и y са с еднакви размери, но с различно съдържание
    x.resize( 6, 6 );
    cout << "\n\nОбектите не са еднакви: (>0<) " << x.isEqual(y) << "\n";

    // x и y вече са еднакви
    x.copy( y );
    cout << "Обектите са еднакви: (>1<) " << x.isEqual(y) << "\n";

    return 0;
}
```

След компилация и изпълнение получаване следния резултат:

```
Обектите са еднакви: (>1<) 1
Обектите не са еднакви: (>0<) 0
```

```

*      *      *      *      *      *

*      #      #      #      #      *

*      #      h      i      #      *
```



```

*      #      ^      ^      #      *
*      #      #      #      #      *
*      *      *      *      *      *

```

Обектите не са еднакви: (>0<) 0

Обектите са еднакви: (>1<) 1

Числата, които са заградени в скоби, показват очакваната стойност, а тези след скобите, стойността, която се връща от функцията isEqual().

5.5 Приятелски декларации

Скриването на информация понякога налага големи ограничения. Използването на приятелски декларации е начин за преодоляване на тези ограничения. Той позволява на външни за класа функции да използват членовете от секция private. Най-напред ще илюстрираме нуждата от приятелски декларации.

Операциите за стандартен вход и изход (">>", "<<") могат да се дефинират и за класове. Това ще ни позволи да отпечатваме обекти от произволен клас, както отпечатваме променливи от стандартните типове. Например:

```

Screen myScreen;
cout << myScreen;
cout << "myScreen: " << myScreen << "\n";

```

Левият операнд на операциите за стандартен вход и изход е име на входно или изходно устройство. Няколко операции за вход или изход могат да се обединят в един оператор, защото всяка операция връща името на входното или изходното устройство. Например:

```
(( (cout << "myScreen: ") << myScreen) << "\n");
```

Всеки от заградените в скоби изрази връща името на устройството за изход cout. То става ляв операнд на следващия израз.

```
cout << myScreen
```

може да се реализира чрез

```
ostream& operator<<( ostream&, Screen& );
```

Този вид на операцията за изход показва, че тя не е член на клас Screen. Ето как изглежда декларацията на същата операция, когато тя е член на класа:

```

class Screen {
public:
    ostream& operator<<( ostream& );

```

```
// ...  
};
```

Всяка функция-член на клас използва като ляв операнд обект или указател към обект от класа. Това е причината в горната декларация да липсва втори аргумент на операцията за изход. Обръщението към тази операция изглежда така:

```
myScreen << cout;
```

Този обърнат синтаксис затруднява работата, защото извеждането на обекти не е аналогично на извеждането на променливи от стандартните типове. Ако използваме първата декларация на операцията, когато тя не е член на клас Screen, няма да имаме достъп до членовете от секция private, а точно това са членовете, които трябва да изведем. В тази безизходна ситуация на помощ идват приятелските декларации.

Приятелска за даден клас може да бъде функция, която не е член на класа или функция-член на дефиниран преди това клас. Нещо повече: цял клас може да бъде приятелски за друг клас. Всяка приятелска функция има достъп до членовете, които са скрити за външни функции. Всички функции на приятелски клас имат достъп до недостъпните членове на другия клас.

Всяка приятелска декларация започва с ключовата дума friend. Приятелските декларации се поставят в тялото на съответния клас. Те не декларираят членове на класа, затова няма значение в коя от трите секции се намират. Групирането им непосредствено след заглавната част на дефиницията на класа се налага като стил на програмиране:

```
class Screen {  
    friend ostream& operator>>( ostream&, Screen& );  
    friend ostream& operator<<( ostream&, Screen& );  
public:  
    // ... останалата част от дефиницията на Screen  
};
```

Да се опитаме да реализираме декларираната по-горе операция за стандартен изход. Ще използваме три от членовете на клас Screen, които го описват, а именно height, width и screen. Извеждането на обект от този клас има следния формат:

```
<height,width>linearScreenDump
```

Реализацията на операцията за изход изглежда така:

```
ostream& operator<<( ostream& os, Screen& s )  
{  
    os << "\n<" << s.height << "," << s.width << ">";  
  
    char *p = s.screen;  
    while ( *p ) os.put( *p++ );  
  
    return os;  
}
```

Следващата малка програма проверява действието на дефинираната операция:

```
#include <stream.h>
#include "Screen.h"

main() {
    Screen x(4,4,'%');
    cout << x;
    return 0;
}
```

След компилация и изпълнение получаваме следния резултат:

```
<4,4>%%%%%%%%%
```

Операцията за стандартен вход чете резултата от операцията за стандартен изход. Нека този резултат се съхранява във файл със име output. За простота при четене на обект позицията на курсора се установява в началната позиция на екрана. Следва дефиницията на операцията за вход. Проверката за коректност на входния формат е пропусната, за да се спести място:

```
istream& operator>>( istream& is, Screen& s );
{ // чете обект от клас Screen, изведен чрез операцията "<<"
    int wid, hi;
    char ch;

    // проверката на входния формат е пропусната
    // входният формат е <hi,wid>screenDump
    is >> ch;          // '<<'
    is >> hi;           // получава height
    is >> ch;           // ','
    is >> wid;          // получава width
    is >> ch;           // '>>'

    delete s.screen;

    int sz = hi * wid;
    s.height = hi; s.width = wid;
    s.cursor = s.screen = new char[ sz + 1 ];

    char *endptr = s.screen + sz;
    char *ptr = s.screen;
    while ( ptr != endptr ) is.get( *ptr++ );
    *prt = '\0';

    return is;
}
```

Следващата малка програма проверява действието на операциите за вход и изход върху конкретен обект от клас Screen:

```
#include <stream.h>
#include "Screen.h"
```

```
main() {
    Screen x(5,5,'?');
    cout << "Начален екран: \t" << x;

    cin >> x;
    cout << "Въведен екран: \t" << x;
    return 0;
}
```

Резултатът от предишната програма се използва като вход в тази програма. След компилация и изпълнение получаваме следния резултат:

```
Начален екран:
<5,5>????????????????????????????
Въведен екран:
<4,4>%%%%%%%%%%%%%
```

Упражнение 5-8. Дефинирайте операцията за стандартен вход, така че тя да проверява коректността на входния формат.

Упражнение 5-9. Реализирайте отново операцията за стандартен изход, като запазвате текущата позиция на курсора. За целта използвайте помощните функции `col()` и `row()`, членове на клас `Screen`.

Упражнение 5-10. предефинирайте операцията за стандартен вход в съответствие с изменения от предишното упражнение формат на операцията за стандартен изход.

Всяка функция от множество функции с еднакво име може да се декларира като приятелска за даден клас. Например:

```
iostream& storeOn( iostream&, Screen& );
BitMap& storeOn( BitMap&, Screen& );

class Screen {
    friend iostream& storeOn( iostream&, Screen& );
    friend BitMap& storeOn( BitMap&, Screen& );
public:
    // ... останалата част от дефиницията на клас Screen
};
```

Ако една функция работи с обекти от два различни класа, тя може да се декларира като приятелска за двата класа или да бъде член на единия клас и приятелска за другия. Ще разгледаме два примера.

В първия пример функцията би трябвало да бъде член на двата класа, но това е невъзможно. Затова тя се декларира като приятелска за двата класа:

```
// предварителни декларации
class Screen;
class Window;
Screen& isEqual( Screen&, Window& );

class Screen {
    friend Screen& isEqual( Screen&, Window& );
```

```
// ... останалата част от дефиницията на клас Screen
};

class Window {
friend Screen& isEqual( Screen&, Window& );
// ... останалата част от дефиницията на клас Window
};
```

Във втория пример логиката изисква функцията да бъде член на единия клас. Тя се обявява за приятелска на втория клас, защото трябва да има достъп и до неговите членове.

```
class Window;

class Screen {
public:
    Screen& copy( Window& );
    // ...останалите членове на Screen
};

class Window {
public:
    friend Screen& Screen::copy( Window& );
    // ...останалите членове на Window
};

Screen& Screen::copy( Window& ) { /* ... */ }
```

Знаем, че един клас може да се декларира като приятелски за друг клас. Ето един пример:

```
class Window;

class Screen {
friend class Window;
public:
    // ...останалите членове на Screen
};
```

Така функциите-членове на Window имат достъп до всички членове на клас Screen.

5.6 Статични членове

Понякога всички обекти от един клас трябва да имат достъп до една и съща променлива. Тя може да бъде флаг или брояч, който се променя динамично в хода на програмата и показва колко обекта от класа съществуват във всяка точка на програмата. Променливата може да бъде указател към функция, която обработва критичните ситуации. Във тези случаи е по-удобно да се използва една и съща променлива за всички обекти, отколкото всеки обект да поддържа свое копие на променливата. Решение в такива случаи е обявяването на съответния член на класа за статичен.

Статичният член действа като глобална променлива, но притежава следните две предимства:

1. Скриването на информация може да се използва и в този случай. Статичният член може да е скрит, докато глобалната променлива винаги е общодостъпна.

2. Статичният член не принадлежи на глобалната област на действие. Това премахва възможността за случайно използване на еднакви имена на променливи.

Член, който се използва за представяне на класа, става статичен, ако декларацията му започва с ключовата дума `static`. Достъпът до статичните данни на класа зависи от секцията, в която са дефинирани. В следващата дефиниция `costPerShare` е статичен скрит (`private`) член на клас `CoOp` от тип `double`:

```
class CoOp {
friend compareCost( CoOp&, CoOp* );
public:
    CoOp( int, char* );
    inline double montlyMain();
    void raiseCost(double incr) { costPerShare += incr; }
    double getCost() { return costPerShare; }
private:
    static double costPerShare;
    int shares;
    char *owner;
};
```

Решението да направим `costPerShare` статичен член на клас `CoOp` цели да запази този член скрит за външни функции, като в същото време ограничи възможностите за грешка. Всеки обект от клас `CoOp` има достъп до `costPerShare`. Макар че текущата стойност на тази променлива е една и съща за всички обекти от класа, тя може да се променя. Следователно, тя не е константа и се декларира като статична, защото не е ефективно всеки обект от класа да поддържа нейно копие. След промяна на нейната стойност, всички обекти използват променената стойност. Ако променливата не беше статична, всяко нейно копие трябваше да се актуализира. Това може да бъде източник на грешки.

Статичните данни на даден клас се инициализират извън неговата дефиниция, както се инициализират променливи, които не са членове на клас. Ето как може да се инициализира `costPerShare`:

```
#include "CoOp.h"
double CoOp::costPerShare = 23.99;
```

Всяка статична данна може да се инициализира само веднъж в програмата. Следователно, тези инициализации не трябва да се включват в заглавен файл. Тяхното място е при дефинициите на тези функции-членове на съответния клас, които не са `inline`. Нивото на достъп до статичните данни на класа се отнася само за тяхното четене и промяна, но не и за инициализирането им. Затова `costPerShare` може да се инициализира в глобалната област на действие.

Достъпът до статичните членове на произволен клас е синтактически еднакъв с достъпа до останалите членове. Например:

```
inline double CoOp::montlyMain()
{
    return( costPerShare * shares );
}
```

// илюстрира достъпа до статичния член на клас `CoOp`, като използва

```
// указател към обект и алтернативно име на обект от този клас
compareCost( CoOp& unit1, CoOp* unit2 )
{
    double maint1, maint2;
    maint1 = unit1.costPerShare * unit1.shares;
    maint2 = unit2—>costPerShare * unit2—>shares;
    // ...
}
```

unit1.costPerShare и unit2—>costPerShare осъществяват достъп до статичния член CoOp::costPerShare.

Възможен е и директен достъп до статичните данни на класа, защото в паметта съществува само едно копие на всеки статичен член:

```
if ( CoOp::costPerShare < 100.00 )
```

Операцията за принадлежност към клас "CoOp::" трябва да се използва, защото статичният член costPerShare е в областта на действие на клас CoOp, а не в глобалната област на действие.

Следващата дефиниция на функцията compareCost е еквивалентна на предишната:

```
compareCost( CoOp& unit1, CoOp* unit2 )
{
    double maint1, maint2;
    maint1 = CoOp::costPerShare * unit1.shares;
    maint2 = CoOp::costPerShare * unit2—>shares;
    // ...
}
```

Двата метода за достъп raiseCost() и getCost() работят само със статичната данна costPerShare. За да се спази синтаксисът на обръщение към функции-членове на клас, всяка от тях трябва да се извика с конкретен обект от класа. Така може да се получи подвеждащ програмен код. Проблемът ще се реши, ако двете функции се декларират като статични:

```
class CoOp {
friend compareCost( CoOp&, CoOp* );
public:
    CoOp( int, char* );
    inline double montlyMain();
    static void raiseCost(double incr);
    static double getCost() { return costPerShare; }
private:
    static double costPerShare;
    int shares;
    char *owner;
};

void CoOp::raiseCost(double incr) {
    costPerShare += incr;
}
```

Статичните методи на класа не поддържат указателя `this`. Всяко явно или неявно използване на този указател ще предизвика грешка при компилация. Достъпът до нестатичен член на класа е неявно използване на указателя `this`. Например, `montlyMain()` не може да се декларира като `static`, защото осъществява достъп до нестатичния член `shares`. Статичните методи се дефинират както всички останали методи на класа.

Обръщението към статичните методи може да се извърши чрез обект или указател към обект от класа, както се извършва обръщението към нестатичните методи. Обръщението може да се бъде и директно. Например:

```
compareCost( CoOp& unit1, CoOp* unit2 )
{
    // еквивалентни обръщения към getCost()
    double maint1, maint2;
    if ( CoOp::getCost() == 0 ) return 0;
    maint1 = unit1.getCost() * unit1.shares;
    maint2 = unit2—>getCost() * unit2—>shares;
    // ...
}
```

Директен достъп и обръщение към статичен член на клас са възможни, дори когато не са декларирани обекти от този клас.

Упражнение 5-11. В дадената дефиниция на клас `Y` има две статични данни и два статични метода:

```
class X {
public:
    X( int i ) { val = i; }
    getVal() { return val; }
private:
    int val;
};

class Y {
public:
    Y( int i );
    static getXval();
    static getCallsXval();
private:
    static X Xval;
    static callsXval;
};
```

Инициализирайте `Xval` с 20, а `callsXval` с 0.

Упражнение 5-12. Реализирайте двата статични метода за достъп на клас `Y`. `callsXval` съдържа броя на обръщенията към функцията `getXval()`.

5.7 Указатели към членове на клас

Указателите и особено указателите към функции са полезно средство в програмирането. Например, потребителите на клас `Screen` може да се нуждаят от функция, която изпълнява `n` пъти определена от тях операция. Една реализация на тази функция изглежда така:


```

Screen &repeat( char op, int times ) {
    switch( op ) {
        case DOWN:    // ...обръщение към Screen::down()
            break;
        case UP:      // ...обръщение към Screen::up()
            break;
        // ...
    }
}

```

Предложената дефиниция притежава редица несъвършенства. Тя се влияе явно от функциите-членове на клас Screen. Всеки път, когато се добави или изтрие функция от класа, repeat() трябва да се актуализира. Вторият проблем е свързан с дължината на repeat(). Ако трябва да проверяваме за всяка функция-член на класа, листингът на repeat() ще изглежда твърде сложен.

Ако заменим op със указател към функции-членове на клас Screen от определен функционален тип, ще получим по-общо решение. Тогава оператор switch е излишен. Темата на следващите подраздели е дефинирането и използването на указатели към членове на произволен клас.

Тип на членовете на клас

В C++ адресът на функция-член на клас не може да се присвои на указател към функция, дори когато функционалните типове съвпадат. Например, pfi е указател към функция без аргументи, която връща резултат от тип int:

```
int (*pfi)();
```

Клас Screen притежава две функции от същия функционален тип — getHeight() и getWidth():

```

inline Screen::getHeight() { return height; }
inline Screen::getWidth()  { return width;  }

```

Да предположим, че извън тялото на класа са дефинирани функции HeightIs() и WidthIs():

```

HeightIs() { return HEIGHT; }
WidthIs()  { return WIDTH;  }

```

Указателят pfi получава коректно адресите на последните две функции:

```

pfi = HeightIs;
pfi = WidthIs;

```

Ако в последните два оператора използваме функциите getHeight() и getWidth(), ще получим несъответствие на типовете и грешка при компилация:

```
// неправилно присвояване: несъответствие на типовете
```

```
pfi = Screen::getHeight;
```

Защо се получава несъответствие на типовете? Причината е, че всяка функция-член на клас притежава като допълнителен атрибут името на класа. Указателят към такава функция трябва да отговаря на три изисквания:

1. Определената за него сигнатура да съвпада със сигнатурата на съответната функция.
2. Определения за него тип на резултата да съответства на типа на връщания от функцията резултат.
3. Да притежава допълнителен атрибут, който съответства на допълнителния атрибут на функцията.

При деклариране на указател към функция-член на клас, се използва разширен синтаксис, който включва и името на този клас. Същото важи и за указателите към данните на класа. Да определим типа на члена `height`. Пълният тип на `Screen::height` е "член на клас `Screen` от тип `short`". Следователно пълният тип на указателя към `Screen::height` е "указател към член на клас `Screen` от тип `short`". На езика C++ последният израз, заграден в кавички, се записва така:

```
short Screen::*
```

Указател към член на клас `Screen` от тип `short` се дефинира така:

```
short Screen::*ps_Screen;
```

`ps_Screen` може да се инициализира с адреса на `height`:

```
short Screen::*ps_Screen = &Screen::height;
```

Аналогично, на този указател може да се присвои адреса на члена `width`:

```
ps_Screen = &Screen::width;
```

`ps_Screen` може да получи адреса на `width` или `height`, защото и двете променливи са членове на клас `Screen` от тип `short`. Всеки опит да се вземе адреса на недостъпен член на класа в съответното място на програмата, ще предизвика грешка при компилация.

Упражнение 5-13. Опишете типа на членовете `screen` и `cursor` на клас `Screen`.

Упражнение 5-14. Дефинирайте и инициализирайте указатели към членовете `Screen::screen` и `Screen::cursor`. Присвоете на дефинираните указатели адреси на подходящи променливи.

В дефиницията на указател към функция-член на клас задължително се задават сигнатурата, типа на резултата и името на класа. Например, указател към функциите `getHeight()` и `getWidth()` се дефинира така:

```
int (Screen::* )()
```

С този запис се означава указател към функция-член на клас Screen, която няма аргументи и връща резултат от тип int. Инициализирането и присвояването на стойност на този указател се извършва така:

```
// всеки указател към функция-член на клас може получи стойност 0
int (Screen::*pmf1)() = 0;
int (Screen::*pmf2)() = Screen::getHeight;

pmf1 = pmf2;
pmf2 = Screen::getWidth;
```

Описанието typedef опростява синтаксиса при дефиниране на указатели към членове на клас. В следващия пример typedef дефинира алтернативно име на указателя за функции-членове на клас Screen, които нямат аргументи и връщат резултат от тип Screen&:

```
Screen& (Screen::* )()
typedef Screen& (Screen::*Action)();

Action default = Screen::home;
Action next = Screen::forward;
```

Упражнение 5-15. Използвайте typedef за да дефинирате алтернативни имена на указателите за всеки различен тип функции-членове на клас Screen.

Указатели към функции-членове на клас може да бъдат аргументи на други функции. В сигнатурата на тези функции указателите към членовете на класа може да получат и стойности по подразбиране. Например:

```
action( Screen&, Screen& (Screen::* )() );
```

action() притежава два аргумента:

1. Първият аргумент е от тип reference за обект от клас Screen.
2. Вторият аргумент е указател към функция-член на клас Screen, която няма аргументи и връща обекта, който я е извикал.

action() може да извика по един от следните начини:

```
Screen myScreen;
typedef Screen& (Screen::*Action)();
Action default = Screen::home;
extern Screen& action( Screen&, Action = Screen::display );

ff()
{
    action( myScreen );
}
```

```

    action( myScreen, default );
    action( myScreen, Screen::bottom );
}

```

Упражнение 5-16. Указателите към членове на класа могат да бъдат данни на този клас. Модифицирайте дефиницията на клас `Screen`, като включите в нея указател към функции от типа на `home()` и `bottom()`.

Упражнение 5-17. Предефинирайте съществуващия конструктор на клас `Screen` (или създайте нов), като добавите още един аргумент — указател към функциите от предишното упражнение. Дайте му стойност по подразбиране. Създайте метод за достъп, който улеснява работата с новия член.

Използване на указатели към членове на клас

Членовете на даден клас са достъпни посредством указатели към тях чрез конкретен обект от класа. Към този обект се прилагат операциите `".*"` или `"—>*"`. Да разгледаме един пример:

```

int (Screen::*pmfi)() = Screen::getHeight;
Screen& (Screen::*pmfS)(Screen&) = Screen::copy;

Screen myScreen, *bufScreen;

// директно обръщение към методите на клас Screen
if (myScreen.getHeight() == bufScreen—>getHeight())
    bufScreen—>copy( myScreen );

// еквивалентно обръщение към същите методи чрез указателите към тях
if ( (myScreen.*pmfi)() == (bufScreen—>*pmfi)() )
    (bufScreen—>*pmfS)(myScreen);

```

Скобите в обръщението

```

(myScreen.*pmfi)()
(bufScreen—>*pmfi)()

```

са задължителни, защото приоритетът на операцията `"()"` (обръщение към функция) е по-висок от приоритета на операциите `".*"` и `"—>*"`. Обръщение без скоби

```
myScreen.*pmfi()
```

се интерпретира по следния начин:

```
myScreen.*(pmfi())
```

Този запис означава обръщение към функцията `pmfi()`, като резултатът от нейното изпълнение се свързва с указател към член на клас `Screen`.

Аналогично се осъществява достъп до данните на класа чрез указатели към тях:

```
typedef short Screen::*ps_Screen;
Screen myScreen, *tmpScreen = new Screen(10, 10);
```

```
ff()
{
    ps_Screen pH = &Screen::height;
    ps_Screen pW = &Screen::width;
    tmpScreen—>*pH = myScreen.*pH;
    tmpScreen—>*pW = myScreen.*pW;
}
```

Инициализацията на pH и pW в тялото на ff() ще предизвика грешка при компилация, ако ff() не е декларирана като приятелска функция за клас Screen, защото height и width са недостъпни за външни функции.

Сега ще дефинираме отново обсъжданата в началото на раздела функция repeat():

```
typedef Screen& (Screen::*Action)();
Screen& Screen::repeat( Action op, int times )
{
    for ( int i = 0; i < times; ++i ) (this—>*op)();
    return *this;
}
```

Декларацията на функцията осигурява стойности по подразбиране и за двата си аргумента:

```
class Screen {
public:
    Screen& repeat( Action=Screen::forward, int=1 );
    // ...
};
```

Следващите обръщения към repeat() са коректни:

```
Screen myScreen;
myScreen.repeat(); // repeat( Screen::forward, 1 );
myScreen.repeat( Screen::down, 20 );
```

Можем да дефинираме таблица от указатели към членове на даден клас. В следващия пример Menu е таблица от указатели към методите на Screen, които осъществяват движение на курсора. Допълнително е дефиниран изброимият тип CursorMovements. Той се използва за индексирание на Menu.

```
Action Menu[] {
    Screen::home,
    Screen::forward,
    Screen::back,
    Screen::up,
    Screen::down,
```

```

    Screen::bottom
};

enum CursorMovements {
    HOME, FORWARD, BACK, UP, DOWN, BOTTOM
};

```

Ще дефинираме още една функция с име `move()` с един аргумент от тип `CursorMovements`:

```

Screen& Screen::move( CursorMovements cm )
{
    (this—>*Menu[ cm ])();
    return *this;
}

```

Тази функция може да се използва в интерактивна програма, при избор на начина на движение на курсора от меню.

Упражнение 5-18. Създайте нов вариант на функцията `repeat()`, чийто първи аргумент е от тип `CursorMovements`.

Указатели към статични членове

Статичните членове принадлежат на класа, а не на конкретен обект от него. Това е причина декларацията на указател към статичен член на класа да е същата като декларацията на "обикновен" указател. В нея не присъства името на класа. Да разгледаме отново дефиницията на клас `CoOp`:

```

class CoOp {
friend compareCost( CoOp&, CoOp* );
public:
    CoOp( int, char* );
    inline double montlyMain();
    static void raiseCost(double incr);
    static double getCost() { return costPerShare; }
private:
    static double costPerShare;
    int shares;
    char *owner;
};

void CoOp::raiseCost( double incr )
{
    costPerShare += incr;
}

```

Типът на `&costPerShare` е `double*`, а не `double CoOp::*`. Указател към `costPerShare` се дефинира така:

```

// типът не е double CoOp::*
double *pd = &CoOp::costPerShare;

```

Тази дефиниция е аналогична на дефиницията на "обикновен" указател. Той не изисква свързване с конкретен обект от класа. Например:

```
CoOp unit;  
double maint = *pd * unit.shares;
```

Аналогично типът на `getCost()` е `double (*)()`, а не `double (CoOp::*)()`. Дефинирането на указател към тази функция и непрякото обръщение към нея са аналогични на тези при указатели към "обикновени" функции:

```
// типът не е double (CoOp::*)()  
double (*pf)() = CoOp::getCost;  
double maint = pf() * unit.shares;
```

5.8 Област на действие при класове

Един клас може да се намира в глобална или локална област на действие. Ако дефиницията на даден клас се вложи в дефиницията на друг клас, двата класа остават в една и съща област на действие. Вложеният клас не се смята за член на класа, в който е дефиниран. Той няма привилегии за достъп до скритите (`private`) и защитените (`protected`) членове на ограждащия клас. В раздел 4.2 дефинирахме клас `IntList` и скрития (`private`) клас `IntItem`. Сега ще дадем еквивалентна дефиниция на двата класа, като вложим дефиницията на `IntItem` в дефиницията на `IntList`:

```
class IntList {  
    class IntItem {  
        friend class IntList;  
    private:  
        IntItem( int v=0 ) { val = v; next = 0; }  
        IntItem *next;  
        int val;  
    };  
    public:  
        IntList( int val ) { list = new IntItem( val ); }  
        IntList() { list = 0; }  
        // ...  
    private:  
        IntItem *atEnd();  
        IntItem *list;  
};
```

`IntItem` не е член на клас `IntList`. Член на `IntList` е `list`, който е указател към обект от клас `IntItem`. `IntItem` и `IntList` се намират в глобалната област на действие. Влагането на `IntItem` в `IntList` цели да покаже, че `IntItem` се използва само от `IntList`. Аналогично, ако в дефиницията на клас се среща описанието `typedef`, то е видимо там, където е видим и ограждащия го клас. Например:

```
class Foo {  
    typedef int Bar;  
    private:  
        Bar val; // ok  
};  
  
Bar val;        // ok: typedef е видимо и тук
```

Всеки клас поддържа своя област на действие. Имената на неговите членове са локални за тази област на действие. Ако съществува глобална променлива с име, съвпадащо с името на член на класа, глобалната променлива се покрива от този член. Да разгледаме един пример:

```
int height;

class FooBar {
public:
    // FooBar::height ← 0
    FooBar() { height = 0; }
private:
    short height;
};
```

height е декларирана в края на дефиницията на клас FooBar, но е видима навсякъде в тялото на класа. Глобалната променлива height е достъпна чрез явното прилагане на операцията за глобална дефиниция "::". Например:

```
int height;

class FooBar {
public:
    FooBar() { height = ::height; }
private:
    short height;
};
```

Видимостта на локалните променливи във функциите и на членовете в класовете е различна, защото локалните променливи са невидими преди декларирането им в тялото на съответната функция. В следващия пример localFunc() използва две различни променливи height:

```
int height = 66;

localFunc() {
    int hi = height; // ::height
    int height = 24; // ::height вече е покрита
    hi = height;     // hi ← 24
}
```

Следващият пример е още по-объркващ. Как мислите коя от променливите height се използва?

```
int height = 66;

badPractice() {
    int height = height; // за коя height се отнася?
}
```


Една променлива се счита за дефинирана, ако нейното име се появи в оператор за деклариране. В случая локалната променлива се инициализира със себе си. Това означава, че тя получава неопределена стойност. Ето как можем да инициализираме локалната променлива със стойността на глобалната променлива:

```
int height = 66;

bdPrac() {
    int height = ::height;
}
```

Компиляторът няма да сбърка двете променливи `height`, но човек, който разглежда програмата може да сгреша. Препоръчва се да се даде друго име на локалната променлива.

Функции-членове на клас се намират в областта на действие на своя клас. Те поддържат и своя локална област на действие като всяка друга функция. Ако в тялото им се дефинира променлива, чието име съвпада с името на член на класа, членът ще се покрие от локалната променлива. Например:

```
Screen::badPractice() {
    int hi = height;           // Screen::height
    int height = height;       // локалната променлива height
}
```

Достъпът до покрит член на класа се осъществява чрез операцията за принадлежност към клас:

```
Screen::bdPrac()
{
    int height = Screen::height;
}
```

Може да се осъществи и достъп до покритата глобална променлива:

```
Screen::badPractice()
{
    int height = (Screen::height > ::height) ? ::height : Screen::height;
}
```

Определяне областта на действие

Областта на действие на идентификатор, който се намира в тялото на функция-член на клас, се определя по следния алгоритъм:

1. Търси се декларация на този идентификатор в блока, в който се използва. Ако има такава декларация, областта на действие е определена. В противен случай търсенето продължава в ограждащата област на действие.
2. Ако идентификаторът се използва в локален блок на дадената функция, ограждащата област на действие е локалната област на функцията. Търсенето

продължава в нея. Ако там има негова декларация, областта на действие е определена. В противен случай се търси в ограждащата област на действие.

3. Ако идентификаторът се използва в самата функция, ограждащата област е областта на действие на класа. Ако идентификаторът е име на член от класа, неговата област на действие е определена. В противен случай търсенето продължава в ограждащата област на действие.
4. Ако класът не е произведен на друг клас, ограждащата област е глобалната област на действие. Ако в нея съществува търсената декларация, идентификаторът е име на глобална променлива. В противен случай компилаторът съобщава, че се използва недекларирана променлива.

Ако идентификаторът се предшества от операцията за глобална дефиниция или принадлежност към клас, търсенето на декларация се ограничава до търсене в явно зададената област на действие. Да разгледаме един пример. Ще дефинираме множество идентификатори:

```
extern f( int ), ff(), f3();
int i = 1024;

class Example {
public:
    f();          // покрива ::f( int )
    Example( int ii = 0 ) { i = ii; }
private:
    int i;        // покрива ::i
    int ff;       // покрива ::ff()
};
```

Един идентификатор се счита за покрит, ако е деклариран отново в по-вътрешна област на действие. Типовете в различните декларации могат да се различават. Глобалните идентификатори `i`, `ff()` и `f(int)` са невидими в областта на действие на клас `Example`. В `Example::f()` към покритите идентификатори можем да се обърнем чрез операцията за глобална дефиниция

```
#include "Example.h"

Example::f()
{
    int j;

    // грешка: глобалната функция f() е покрита, а Example::f() няма аргументи
    j = f( 1 );

    // ок: явно обръщение към ::f( int )
    // под i се разбира Example::i;
    j = ::f( i );

    // ок: явно обръщение към съответните идентификатори
    ::i = ::f( ::ff() );

    // ок: явно обръщение не се изисква, защото f3() е видима в тялото на клас Example
    return ( f3() );
}
```

В `Example::f()` също можем да дефинираме `i` и `f()`. Те ще бъдат локални и ще покриват съответните членове на класа. Членове са достъпни чрез операцията за принадлежност към клас:

```
#include "Example.h"

Example::f()
{
    // покриваме Example::i
    int i = ff ? Example::i : ( ::ff() ? ::i : 0 );

    // покриваме Example::f
    float f = (float) Example::f();

    return ( i + Example::i + ::i );
}
```

Явното посочване на област на действие ограничава търсенето на декларация в посочената област. Операцията за принадлежност към клас не може да се използва за достъп до глобален идентификатор. Например, обръщението към `Example::f3()` ще предизвика съобщение за грешка, защото `Example::f3()` не е дефиниран.

```
#include "Example.h"

Example::f()
{
    // ок: извиква се ::f3()
    int i = f3();

    // грешка: Example::f3() не е дефиниран
    return( Example::f3() );
}
```

Локални класове

Класове могат да се дефинират и в локална област на действие. В такъв случай името на класа е видимо само в тази локална област. Например:

```
int doString( char *s )
{
    // локален клас, видим само в doString()
    class String { ... };
    String str(s);
}

String str( "gravity" );      // грешка: String е невидим
```

Членовете на локален клас трябва да се дефинират в тялото на класа. В следващия пример дефиницията на една функция е разположена в тялото на друга функция, а това е недопустимо:

```
int doString( char *s )
{
```

```

class String {
public:
    String& operator=(char*);
};

String& String::operator=(char* s) {} // недопустимо
}

```

Същата функция не може да се дефинира и извън тялото на функцията doString, защото клас String е непознат извън doString:

```

int string1(char *s) { class String { ... }; }
int string2(char *s) { class String { ... }; }

// грешка: и двата класа String са невидими
String& String::operator=(char* s) {}

```

Посоченото изискване ограничава размера и сложността на функциите-членове на локален клас до няколко реда. В противен случай кодът на програмата е трудно разбираем.

Всеки локален клас е свързан с локалната област на някаква функция, но дефинираните в тази област променливи са невидими за членовете на класа. Например, в String() обръщението към bufSize е обръщение към глобалната променлива. Същото обръщение във функцията func() е обръщение към локалната променлива.

```

const int bufSize = 1024;

void func() {
    const int bufSize = 512;
    char *ps = new char[bufSize];    // 512

    class String {
    public:
        String() {
            str = new char[ BufSize ]; // 1024
        }
        // ...
    };
}

```

Използването на локални класове позволява да се ограничи видимостта на класа до функцията (или блока), където е дефиниран. В общия случай е разумно да се използва локален клас, когато неговата дефиниция е проста и употребата му се ограничава до една функция.

5.9 Обединения — класове, спестяващи памет

Обединението е специален вид клас. Заделяната за него памет се равнява на паметта, необходима за запис на най-големия му член. Всеки член на обединението се записва на един и същ адрес. Във всеки момент от време на този адрес се намира само един от членовете на обединението. Да разгледаме един пример. За лексическия анализатор на компилатора програмата е последователност от думи. Операторът

```
int i = 0;
```

е последователност от пет думи:

1. Ключовата дума `int`
2. Идентификаторът `i`
3. Операцията `=`
4. Целочислената константа `0`
5. Разделителят `;`

Лексическият анализатор предава тези думи на синтактичния анализатор. Първата негова стъпка е идентификацията на последователността от думи. За да разпознае входния поток от думи като декларация, той се нуждае от допълнителна информация. Трябват му изречения от вида:

```
Type ID Assign Constant Semicolon
```

След това синтактическият анализатор се нуждае от конкретна информация за всяка дума. В случая, той трябва да знае, че

```
Type <==> int
ID <==> i
Constant <==> 0
```

Повече информация за `Assign` и `Semicolon` не е необходима.

Думите ще се представят чрез два члена, `token` и `value`. `token` ще съдържа уникален номер за всяка дума. Например, идентификаторът може да се представи от числото 85, а разделителят `;` от 72. `value` ще съдържа характерната за всяка дума информация. Например, за `ID` `value` може да съдържа низа `"i"`; за `Type` `value` ще бъде кода на тип `int`.

Представянето на `value` създава проблеми. Макар че за всяка дума от входния поток се пази една стойност, `value` трябва да поддържа множество типове. Известно е, че класовете представляват удобно средство за представяне на съвкупност от данни. `value` може да се декларира като клас с име `TokenValue`. `TokenValue` ще съдържа по един член за всеки възможен тип на `value`.

Това представяне е подходящо, но тъй като за всяка дума типът на `value` е различен, `TokenValue` заделя място за всички възможни типове. За предпочитане е `TokenValue` да заделя памет, достатъчна за съхраняване на типа, който изисква най-много място, а не на всички типове едновременно. За целта ще дефинираме обединение с име `TokenValue`:

```
union TokenValue {
    char cval;
    char *sval;
    int ival;
    double dval;
};
```

В дефиницията на `TokenValue`, `double` е типът, който изисква най-много място. За цялото обединение се заделя толкова памет, колкото се заделя за числа от тип `double`. По подразбиране всички членове на обединение са общодостъпни (`public`). Обединенията не

могат да съдържат статични членове или обекти от клас, който притежава конструктор или деструктор. Сега ще покажем как може да се използва `TokenValue`:

```
class Token {
public:
    int tok;
    TokenValue val;
};

lex() {
    Token curToken;
    char *curString;
    int curlval;

    // ...
    case ID:          // идентификатор
        curToken.tok = ID;
        curToken.val.sval = curString;
        break;

    case ICON:        // целочислена константа
        curToken.tok = ICON;
        curToken.val.ival = curlval;
        break;

    // ... и т.н.
}
```

При обединенията съществува опасност от случайно интерпретиране на текущата стойност чрез неподходящ тип на данните. Например, ако последната стойност е записана чрез `ival`, програмистът не бива да се опитва да получи същата стойност чрез указателя `sval`. В противен случай ще предизвика програмна грешка.

За предпазване от подобни грешки, се използва допълнителна променлива, която помни типа на текущата стойност. Допълнителната променлива се нарича дискриминанта на обединението. С тази цел се дефинира члена `tok` на клас `Token`. Да разгледаме един пример:

```
char *idVal;

if ( curToken.tok == ID )
    idVal = curToken.val.sval;
```

Добра практика при работа с обединения е дефинирането на членове, които осигуряват достъп до всеки от възможните типове. Например:

```
char *Token::getString()
{
    if ( tok == ID ) return val.sval;
    error( ... );
}
```

Ако обединението се използва само в конкретен случай, не е задължително то да има име. Това намалява броя на глобалните идентификатори и опасността от поява на еднакви

имена. Следващата дефиниция на Token е еквивалентна на предишната. Единствената разлика е, че в случая обединението е без име:

```
class Token {
public:
    int tok;
    union {
        char cval;
        char *sval;
        int ival;
        double dval;
    } val;
};
```

Обединение, което няма име и не дефинира обект, се нарича анонимно обединение. Следващата дефиниция на клас Token съдържа анонимно обединение:

```
class Token {
public:
    int tok;
    // анонимно обединение
    union {
        char cval;
        char *sval;
        int ival;
        double dval;
    };
};
```

Членовете на анонимно обединение са достъпни директно. Новото описание на lex() е съобразено с последната дефиниция на Token:

```
lex() {
    Token curToken;
    char *curString;
    int curlval;

    // ... определя каква дума представя curToken
    case ID:
        curToken.tok = ID;
        curToken.sval = curString;
        break;

    case ICON: // целочислена константа
        curToken.tok = ICON;
        curToken.ival = curlval;
        break;

    // ... и т.н.
}
```

Анонимните обединения опростяват с едно ниво избора на член, благодарение на това, че имената на техните членове принадлежат на ограждащата област на действие. В анонимно обединение не може да има секции private и protected. Ако анонимното обединение е в глобалната област на действие, то трябва да се декларира като static.

5.10 Битови полета — членове, спестяващи памет

Битовите полета са специални членове на клас. Те се състоят от определен брой битове. Типът им трябва да е целочислен — signed или unsigned. Например:

```
class File {  
    // ...  
    unsigned modified : 1; // битово поле  
};
```

След идентификатора на битово поле следва двоеточие и константен израз, който определя броя на битовете. В примера modified е битово поле, състоящо се от един бит.

Когато е възможно, последователно дефинираните битови полета се съхраняват като съседни битове на променливи от съответния целочислен тип. Така се постига по-голяма компактност. В следващия пример петте битови полета се съхранят като едно число от тип unsigned int, започвайки с полето mode.

```
typedef unsigned int Bit;  
  
class File {  
public:  
    // ...  
private:  
    Bit mode: 2;  
    Bit modified: 1;  
    Bit plot_owner: 3;  
    Bit plot_group: 3;  
    Bit plot_world: 3;  
  
    // ...  
};
```

Достъпът до битовите полета се осъществява, както се осъществява достъпа до останалите данни на класа. Например:

```
File::write()  
{  
    modified = 1;  
    // ...  
}  
  
File::close()  
{  
    if ( modified )    // ... запазва съдържанието  
}
```

Следващият пример показва как се използват битови полета, състоящи се от няколко бита. В него се прилагат разгледаните в раздел 2.8 побитови операции.

```
enum { READ = 01, WRITE = 02 }; // режим за достъп  
  
main() {  
    File myFile;
```



```

myFile.mode |= READ;
if ( myFile.mode & READ )
    cout << "\nmyFile.mode има стойност READ";
}

```

Обикновено стойностите в битовите полета се проверяват от inline-функции. За битовото поле mode такива функции са isRead() и isWrite():

```

inline File::isRead()  { return mode & READ; }
inline File::isWrite() { return mode & WRITE; }

if (myFile.isRead())  /* ... */

```

Операцията за получаване на адрес "&" не може да се приложи към битово поле. Не може да се използва и указател към битово поле. Невъзможно е и декларирането на битово поле като static.

5.11 Предаване на обекти при сигнатура "..."

Обект от клас с дефиниран конструктор или операция за присвояване не може да се предаде като аргумент на функция, в чиято сигнатура не съществува явна декларация на такъв аргумент. Например:

```

extern foo( int, ... );

class Screen {
public:
    Screen( const Screen& );
    // ...
};

void bar( int ival, Screen scrObj )
{
    // грешка: в сигнатурата на foo() не съществува
    // декларация на обект от клас Screen
    foo( ival, scrObj );
}

```

При обръщение към foo(), вместо обект може да се предаде указател към този обект.

ГЛАВА 6

Тази глава разглежда подробно три вида функции-членове на клас.

1. Конструктори и деструктори. Те служат за автоматично инициализиране и освобождаване на обекти.
2. Основни операции с обекти. Част от стандартните операции могат да се дефинират и за обектите на клас. Това опростява синтаксиса. Например, вместо следното явно обръщение към функцията `isEqual()`

```
if( myScreen.isEqual(yourScreen) )
```

можем да използваме операцията за проверка на равенство `"=="`, дефинирана за обектите от клас `Screen`:

```
if ( myScreen == yourScreen )
```

Класовете могат да дефинират собствени операции за управление на динамичната памет.

3. Операции за преобразуване. Те реализират позволените преобразувания на типове за даден клас и се прилагат неявно от компилатора, както се прилагат стандартните преобразувания. Изброените функции се създават за да може с обектите на произволен клас да се работи така лесно, както се работи с вградените типове данни.

6.1 Инициализиране на обекти

Инициализирането на обект представлява инициализиране на неговите данни. Ако всички членове на класа са общодостъпни, обектът може да се инициализира чрез списък, в който съответните стойности са отделени със запетаи. Например:

```
class Word {
public:
    int occurs;
    char *string;
};

Word search = { 0, "resebud" }    // явна инициализация
```

C++ поддържа механизъм за автоматично инициализиране на обекти. При дефиниране на обект или заделяне на памет за него чрез операцията `new`, компилаторът извиква специална функция-член на класа, наречена конструктор. Конструкторът носи името на класа. Ето как изглежда конструкторът на клас `Word`:

```
class Word {
public:
    Word( char*, int=0 );    // конструктор
private:
    int occurs;
    char *string;
```

```
};

#include <string.h>

inline Word::Word( char *str, int cnt ) {
    string = new char[ strlen(str) + 1 ];
    strcpy( string, str );
    occurs = cnt;
}
```

Конструкторът не трябва да посочва тип на резултата. Освен това той не трябва да връща стойност явно. В противен случай неговата дефиниция няма да се различава от дефинициите на другите функции-членове на класа. Конструкторът на клас Word изисква един задължителен аргумент от тип char*. Вторият аргумент се задава по желание. Той е от тип int. Следващите примери показват как се дефинира обект от клас Word, при наличие на конструктор:

```
#include "Word.h"

// Word::Word( "rosebud", 0 )
Word search = Word( "rosebud" );

// Word::Word( "sleigh", 1 )
Word *ptrAns = new Word( "sleigh", 1 );

main()
{ // съкратени записи на обръщението към конструктора
    // Word::Word( "CitizenKane", 0 )
    Word film( "CitizenKane" );

    // Word::Word( "Orson Welles", 0 )
    Word director = "Orson Welles";
}
```

Дефиниране на конструктор

Символните низове са често използвани типове данни. В C++ те са производен тип (масив от символи), без вградени операции. Например, липсват операции за присвояване и сравняване на символни низове. Затова е добре низовете да се представят като абстрактен тип данни. В следващите раздели ще обясним синтаксиса и семантиката на конструкторите, деструкторите и операциите, като използваме низове. Да започнем с конструкторите.

Знаем, че конструкторът носи името на класа. За един клас може да се дефинират няколко конструктора. Например, клас String притежава два члена-данни:

1. указател str от тип char*, който адресира текущия низ.
2. идентификатор len от тип int, който съдържа дължината на низа.

Ще декларираме два конструктора. Първият инициализира str, а вторият len:

```
class String {
public:
    String( int );
    String( char* );
private:
    int len;
    char* str;
};
```

При дефиниране на обект, се заделя място за неговите нестатични данни. Конструкторите се грижат за инициализиране на заделената памет. Ето дефиницията на първия конструктор:

```
String::String( char *s)
{ // #include <string.h>
  len = strlen( s );
  str = new char[ len + 1 ];
  strcpy( str, s );
}
```

Дефиницията не е сложна. Силата на конструктора се състои във възможността той да се извика неявно за всеки дефиниран обект, който трябва да се свърже с конкретен низ. `strlen()` и `strcpy()` са функции, дефинирани в стандартната библиотека на C++. Ще обясним защо не присвоихме на `str` адреса на `s`

```
str = s;
```

а заделихме динамична памет и в нея копирахме `s`. Главната причина е, че не знаем как е заделена паметта за `s`.

- Ако паметта е заделена от програмния стек, при напускане на блока, в който е дефинирана променливата `s`, съдържанието на тази памет е произволно. Следователно, по-нататъшното използване на `str` ще предизвиква грешки. Например:

```
String *readString()
{ // пример за низ, разположен в програмния стек
  char inBuf[ maxLen ];
  cin >> inBuf;
  String *ps = new String( inBuf );
  return ps;
}

String *ps = readString();
```

- Ако паметта е заделена динамично, тя трябва да се освободи, когато обектът излезе от областта на действие. Напомняме, че използването на операцията `delete` за памет, която не е заделена динамично, поражда грешка при изпълнение.

От казаното следва, че присвояването на указатели не е безопасна операция. Тя изисква повишено внимание от програмиста. Отново ще се върнем на този проблем, когато обсъждаме инициализирането и присвояването на обекти.

Ето как изглежда дефиницията на втория конструктор на клас `String`:

```
String::String( int ln ) {
  len = ln;
  str = new char[ len + 1 ];
  str[0] = '\0';
}
```

Решението да пазим дължината на низа като отделен член на класа може да се оспорва. Трябва да се избере между загуба на памет и по-бавно изпълнение на програмата. Направеният избор се основава на два фактора. Първо, дължината е необходима доста често. Затова намаленото време за нейното получаване компенсира увеличаването на памет (това трябва да се потвърди в практиката). Второ, клас String ще се използва не само за низове, но и за поддържане на буфери с фиксирана дължина.

Всяка дефиниция на обект поражда неявно обръщение към конструктор на класа, при което се прави пълен контрол на типовете. Следващите дефиниции на обекти от клас String са неправилни:

```
int len = 1024;
String myString;           // грешка: няма аргументи
String inBuf( &len );      // грешка: типът не може да е int*
String search( "rosebud", 7 ); // грешка: повече аргументи
```

Съществува явен и съкратен начин за предаване на аргументи на конструктор:

```
// явно предаване
String searchWord = String( "rosebud" );

// съкратен запис #1
String commonWord( "the" );

// съкратен запис #2
String inBuf = 1024;

// използването на new изисква явно предаване
String *ptrBuf = new String( 1024 );
```

При използването на операцията new, обръщението към конструктора се извършва след заделянето на памет. Ако няма достатъчно памет, конструкторът не се изпълнява. Указателят към съответния обект получава стойност 0. Например:

```
String *ptrBuf = new String( 1024 );
if ( ptrBuf == 0 ) cerr << "Не достига памет\n";
```

Полезно е да позволим дефиниране на обект без параметри:

```
String tmpStr;
```

За тази цел може да дадем стойности по подразбиране на аргументите на дефинираните конструктори. Друг начин е да дефинираме т.нар. конструктор по подразбиране. За такъв се смята конструкторът без аргументи. Масив от обекти от даден клас, за който има дефинирани конструктори, се инициализира чрез инициализиране на отделните елементи на масива. Ако стойностите в инициализиращия списък не достигат, останалите елементи на масива се инициализират от конструктора без аргументи. Ако няма такъв конструктор, инициализиращият списък трябва да бъде пълен. Следва дефиницията на конструктора без аргументи на клас String:

```
#include "String.h"

String::String()
{ // конструктор по подразбиране
    len = 0;
    str = 0;
}
```

Често допускана грешка е използването на такава дефиниция:

```
String st();
```

Тя не дефинира обект от клас `String`, който да се инициализира от конструктора без аргументи. По този начин се декларира функция без аргументи, с име `st`, която връща резултат от тип `String`. Следващите два оператора дефинират обект с име `st` и го инициализират чрез конструктора по подразбиране:

```
String st;
String st = String();
```

Упражнение 6-1. Дефинирайте конструктор на клас `String`, който позволява следните декларации:

```
String s1( "rosebud", 7 );
String s1( "rosebud", 8 );
String s2( "", 1024 );
String s3( "The Raw and the Cooked" );
String s4;
```

Конструктори и скриване на информация

Нивото на достъп до даден конструктор зависи от това, в коя секция на класа се намира неговата декларация. Ако декларираме `String::String(int)` в секция `private`, ще ограничим използването на клас `String` само за низове. При дефиниране на буфери този конструктор ще бъде достъпен така:

```
class String {
    friend class Buf;
public:
    String( char* );
    String();
private:
    String( int );

    // ... остатък от дефиницията на String
};
```

Само функциите-членове на клас `String` и приятелският клас `Buf` могат да използват конструктора `String::String(int)`. Другите два конструктора се използват без ограничение.

```
f() {
    // ок: String::String(char*) е общодостъпен член
    String search( "rosebud" );

    // грешка: String::String(int) е скрит член
    String inBuf( 1024 );
}

Buf::in()
{
    // ок: String::String(char*) е общодостъпен член
    String search( "rosebud" );

    // ок: String::String(int) е скрит член, но Buf е приятелски клас
    String inBuf( 4096 );
}
```

Скритите класове са класове без общодостъпни конструктори. В раздел 4.2 е дефиниран скритият клас `IntItem`. Негови обекти могат да се дефинират само чрез приятелския клас `IntList`.

Деструктори

C++ поддържа механизъм за автоматично "освобождаване" на обекти. Ако обект от даден клас напусне областта на действие, в която е дефиниран, или указателят към него се освободи чрез операцията `delete`, компилаторът извиква специална функция-член на класа, наречена деструктор. Деструкторът не се извиква, ако областта на действие се напусне от алтернативно име на обекта. Това потвърждава факта, че алтернативното име на дефиниран обект не е самият обект. Следва декларацията на деструктора на клас `String`:

```
class String {
public:
    ~String(); // деструктор
};
```

Деструкторът носи името на класа, предшествано от тилда "~". Той е функция без аргументи. Следователно за един и същи клас не може да се дефинира повече от един деструктор. Аналогично на конструкторите, деструкторът не трябва да има тип на резултата и да връща стойност. Ето една примерна дефиниция на деструктора на клас `String`:

```
String::~String() { delete str; }
```

Ще припомним, че конструкторите не заделят памет за обектите. Те инициализират заделената вече памет, като по този начин я свързват с конкретен обект от класа. Аналогично, деструкторът не освобождава заетата от обекта памет. Той прекъсва връзката между обекта и паметта, преди нейното освобождаване при напускане областта на действие. В нашия случай `str` адресира памет, заета чрез операцията `new`. Деструкторът на `String` освобождава явно тази памет чрез операцията `delete`. Паметта, заделена за `len`, не изисква специално освобождаване.

Деструкторите могат да извършват различни допълнителни действия. Много разпространена практика при тестване на програми е отпечатването на съобщения в определени контролни точки. Следващият деструктор съдържа такова контролно съобщение:

```
String::~String() {
#ifdef DEBUG
    cout << "~String() " << len << " " << str << "\n";
#endif
    delete str;
}
```

Накратко, деструкторите могат да изпълнят всички действия, които трябва да се извършат преди напускане областта на действие на даден обект.

Деструкторът на класа не се извиква автоматично за указатели към обекти. Той може да се изпълни само чрез явно прилагане на операцията delete. Например:

```
#include "String.h"
String search( "rosebud" );

f() {
    // искаме деструктора да не се изпълни за p
    String *p = &search;

    // искаме да приложим деструктора за pp
    String *pp = new String( "sleigh" );

    // ... тялото на f()
    delete pp;      // извикване на String::~String() за pp
}
```

Ако указателят, за който се прилага операцията delete, не съдържа адрес на обект, т.е. има стойност 0, деструкторът не се извиква. Следната проверка е излишна:

```
if( pp != 0 ) delete pp;
```

Съществува случай, когато програмистът трябва да извика явно деструктора на класа. Такъв е случаят, когато той желае да изтрие обекта, но не иска да освободи заеманата от него памет. Такава ситуация възниква, когато обектът е разположен на точно определен адрес в паметта. Например:

```
#include <string.h>
#include <stream.h>
#include <new.h>

struct inBuf {
public:
    inBuf( char* );
    ~inBuf();
private:
    char *st;
    int sz;
};

inBuf::inBuf( char *s ) {
    st = new char[ sz = strlen(s) + 1 ];
    strcpy( st, s );
}
```



```

inBuf::~inBuf() {
    cout << "inBuf::~inBuf: " << st << "\n";
    delete st;
}

char *pBuf = new char[ sizeof( inBuf ) ];

main() {
    inBuf *pb = new (pBuf) inBuf( "free store inBuf #1" );
    pb->inBuf::~inBuf();    // явно обръщение към деструктора

    pb = new (pBuf) inBuf( "free store inBuf #2" );
    pb->inBuf::~inBuf();    // явно обръщение към деструктора

    // ...
}

```

След компилация и изпълнение на тази програма получаваме следния резултат:

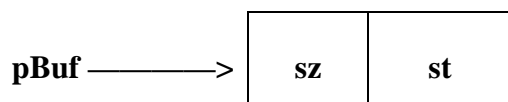
```

inBuf::~inBuf(): free store inBuf #1
inBuf::~inBuf(): free store inBuf #2

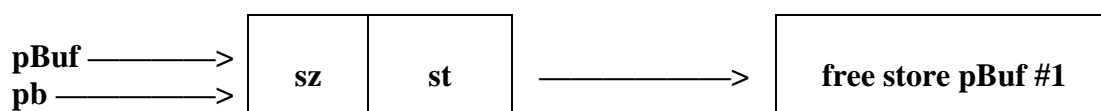
```

Бележка на преводача: За изясняване на процеса, предлагаме на читателя следната схема:

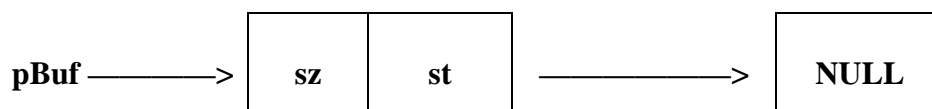
1. Начално състояние:



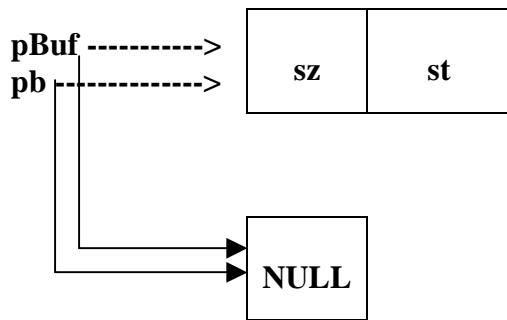
2. След `pb = new(pBuf) inBuf("free store pBuf #1")`



3. Явно извикване: `pb->inBuf::~inBuf()`



4. Ако се използва неявно извикване: `delete pb`, паметта, заделена за `sz` и `st` се освобождава.



Явното обръщение към деструктора на класа изисква пълно описание на името му:

```

pb—>inBuf::~inBuf();    // правилно
pb—>~inBuf();           // неправилно

```

Масиви от обекти

Масиви от обекти се дефинират, както се дефинират "обикновени" масиви. Например, tb1 и tb2 дефинират масиви, чиито 16 елемента са обекти от клас String:

```

const size = 16;
String tb1[ size ];
String *tb2 = new String[ size ];

```

Операцията за индексване осигурява достъп до елементите на масива. Достъпът до членовете на всеки отделен обект се реализира чрез операцията за избор на член, приложена след операцията за индексване. Например:

```

while (cin >> tb1[i]) tb1[i].display();

```

Елементите на масив от обекти се инициализират чрез конструкторите на класа. Аргументите на конструкторите се описват в т.нар. инициализиращ списък. Ако се използва конструктор с няколко аргумента, инициализиращият списък трябва да съдържа пълно описание на конструкторите. В противен случай е възможно пълно описание или изброяване на аргументите:

```

String ar1[] = { "phoenix", "crane" }
String ar2[3] = { String(), String(1024), String("string") };
String ar3[2] = { 1024, String( 512 ) };
Screen as[] = { Screen(24, 80, '#') };

```

Ако класът притежава конструктор без аргументи, той се извиква при непълен инициализиращ списък за инициализиране на останалите елементи на масива. Ако няма конструктор по подразбиране, инициализиращият списък трябва да бъде пълен. Динамичните масиви не могат да се инициализират явно. Класът трябва да има конструктор по подразбиране или въобще да не притежава конструктори.

При напускане областта на действие на tb12, към този указател трябва да се приложи явно операцията delete, за да се върне заделената за него памет в свободната динамична памет. Но простият запис

```
delete tbl2;
```

не е достатъчен, защото предизвиква извикване на деструктора на `String` само за първия елемент на `tbl2`. `delete` не знае, че `tbl2` не адресира един обект от класа, а цял масив от обекти. Програмистът трябва да предаде на `delete` и размера на масива:

```
delete [ size ] tbl2;
```

Сега деструкторът на клас `String` се извиква за всеки елемент на масива.

Членове-обекти на клас

След като дефинирахме клас `String`, ще дефинираме отново клас `Word`, като заменим неговия член от тип `char*` с член от тип `String`. При това искаме да запазим съвместимост с редишния общодостъпен интерфейс на класа.

```
class Word {  
public:  
    Word();  
    Word( char*, int = 0 );  
    Word( String&, int = 0 );  
private:  
    int occurs;  
    String name;  
};
```

За всеки обект от клас `Word` се извикват два конструктора — неговият собствен конструктор и конструкторът за члена от клас `String`. Тогава възникват два въпроса:

1. Съществува ли ред на обръщение към конструкторите? Ако съществува, какъв е той?
2. Как се предават параметри на конструктора, който инициализира член на класа?

Отговорът на първия въпрос е положителен. Конструкторите, които инициализират членове на класа се извикват преди конструктора на този клас. Ако класът съдържа няколко члена-обекти от различни класове, редът на обръщение към съответните конструктори следва реда на декларацията на членовете в тялото на класа. Редът на обръщение към съответните деструктори е обратен.

```
// първо се извиква String::String( char* )  
// след това се извиква Word::Word( char* )  
Word flower( "iris" );
```

Аргументите на конструкторите, които инициализират членове на класа, се предават чрез т.нар. списък за инициализация на членове. Това е списък от разделени със запетайки двойки — име на член и в скоби съответен аргумент. Например:

```
Word::Word( char *s, int cnt ) : name( s ) {  
    occurs = cnt;  
}
```

Списъкът за инициализация на членове следва след сигнатурата на конструктора и е отделен от нея чрез ":". Всеки член може да се включи само веднъж в този списък. Списъкът трябва да присъства в дефиницията на конструктора, а не в неговата декларация. В горния пример на `name` се предава указателя `s`, а след това този указател се използва като аргумент на конструктора на `String`. Данните от стандартен тип, също могат да се включат в списъка за инициализация на членове. В следващия пример `occurs` се инициализира със стойността на `cnt`.

```
Word::Word( char *s, int cnt )
    : name( s ), occurs( cnt ) {}
```

Изпълнението на конструктора се състои от две фази — инициализация и присвояване. Когато тялото на конструктора е празно, втората фаза липсва. Например:

```
class Simple {
public:
    Simple( int, float );
private:
    int i;
    float f;
};

Simple::Simple( int ii, float ff )
    : i(ii), f(ff)           // инициализация
    {}                      // присвояване
```

Фазата присвояване започва с изпълнение на тялото на конструктора. Членовете на класа, които са обекти от клас с дефиниран конструктор без аргументи, се инициализират неявно. Първата фаза се поражда от списъка за инициализация на членове. В следващия конструктор на клас `Simple` няма първа фаза:

```
Simple::Simple( int ii, float ff )
{
    i = ii; f = ff;         // присвояване
}
```

В повечето случаи разликата между двете фази е разбираема за програмиста. При използване на константни данни или данни от тип `reference`, разликата не е ясна, но е много съществена. Такива данни могат да се инициализират само чрез списък за инициализация на членове. Следващата дефиниция на конструктор е неправилна:

```
class ConsRef {
public:
    ConsRef( int ii );
private:
    int i;
    const int ci;
    int &ri;
};

ConsRef::ConsRef( int ii )
```

```
{ // присвояване
    i = ii;    // правилно
    ci = ii;   // грешка: на константа не може да се присвои стойност
    ri = i;    // грешка: ri е неинициализирана
}
```

Когато започне изпълнението на тялото на конструктора, инициализацията на константните данни и данните от тип `reference` трябва да е приключила. Това може да се направи само чрез списъка за инициализация на членове:

```
ConsRef::ConsRef( int ii )
    : ci( ii ), ri( i )      // инициализация
{ // присвояване
    i = ii;
}
```

Инициализиращият аргумент може да е не само прост идентификатор или константа. Той може да е произволен по сложност израз. Например:

```
class Random {
public:
    Random( int i ) : val( seed( i ) ) {}
    int seed( int );
private:
    int val;
};
```

Инициализиращият аргумент на член-обект от клас може да е друг обект от този клас:

```
Word::Word( String &str, int cnt )
    : name( str ), occurs( cnt )
    {}

String msg( "hello" );
Word greetings( msg );
```

Всеки член-обект от даден клас трябва да се включи в списъка за инициализация на членове, ако конструкторът на неговия клас изисква параметри. В противен случай се получава грешка при компилация. В следващия пример клас `SynAntonym` съдържа три члена-обекти: `wd` от клас `Word`, `synonym` и `antonym` от клас `String`. Знаем, че при дефиниране на обект от клас `Word`, трябва да посочим поне един аргумент от тип `char*` или `String&`. Следователно, списъкът за инициализация на членове в конструкторите на `SynAntonym` трябва да включва аргумент за `wd`.

```
class SynAntonym {
public:
    SynAntonym( char* s ) : wd(s) {}
    SynAntonym( char* s1, char* s2, char* s3): wd(s1), synonym(s2), antonym(s3) {}
    ~SynAntonym();
private:
    String synonym;
    Word wd;
```

```

    String antonym;
};

SynAntonym sa1( "repine" );
SynAntonym sa2( "cause", "origin", "effect" );

```

Да разгледаме реда на извикване на конструкторите при дефиниране на обектите sa1 и sa2:

1. Последователно се извикват конструкторите за първия, втория и т.н. членове на SynAntonym.

```

String();           // за члена synonym
String( "repine" ); // за wd.name
Word( "repine" );   // за члена wd
String();           // за члена antonym

// sa1( "repine" );
String( "origin" ); // за члена synonym
String( "cause" );  // за wd.name
Word( "cause" );    // за члена wd
String( "effect" ); // за члена antonym

// sa2( "cause", "origin", "effect" );

```

2. Извиква се конструкторът на клас SynAntonym.

Ако съответният член е обект със членове-обекти, горните две правила се прилагат рекурсивно. Редът на обръщение към деструкторите е противоположен на описания по-горе. Това означава, че най-напред се извиква деструкторът на съответния клас и една след това деструкторите на неговите членове. Ако тялото на класа съдържа няколко члена-обекти, най-напред се извиква деструкторът на последния деклариран член, после на предпоследния и т.н.

Упражнение 6-2. Дадена е следната дефиниция на клас Buffer:

```

#include "String.h"

class Buffer {
public:
    // ...
private:
    String buf;
};

```

Дефинирайте конструктор и деструктор на класа, които позволяват използване на следните декларации:

```

String s1;
Buffer();
Buffer( 1024 );
Buffer( s1 );

```

Двоичното дърво е важен абстрактен тип данни. По-долу е даден скелет на неговата дефиниция. Предполагаме, че във възлите на дървото стоят цели числа. В този и следващите подраздели ще представим реализация на двоично дърво на базата на няколко упражнения.

```
class BinTree;

class Inode {          // класът е скрит (private)
    friend class BinTree;
    int val;
    BinTree *left;
    BinTree *right;
};

class BinTree {
public:
    // ... общодостъпен интерфейс
private:
    Inode *node;
};
```

Двоичното дърво може да не притежава възли (node е 0) или да бъде дърво в истинския смисъл на думата (node е указател към първия му възел). Всеки възел на дървото се описва с три члена: цяло число, което се използва като ключ и два указателя към лявото и дясното поддърво. Всяко поддърво е или празно, или от него излизат нови поддървета.

Упражнение 6-3. Какви са предимствата и недостатъците на дефинирането на Inode като скрит клас.

Упражнение 6-4. Дефинирайте конструктор(и) и деструктор за клас Inode.

Упражнение 6-5. Дефинирайте конструктор(и) и деструктор за клас BinTree.

Упражнение 6-6. Помислете върху дефинициите от упражнение 6-4 и 6-5.

6.2 Инициализация чрез копиране на членове

Инициализацията на току-що дефиниран обект не предизвиква обръщение към конструктор на класа, ако обектът се инициализира с друг обект от същия клас. Например:

```
String vowel( "a" );
String article = vowel;
```

article се инициализира чрез копиране на членовете на vowel в съответните членове на article. Този начин за инициализиране е известен като инициализиране чрез копиране на членове (memberwise initialization) и се осъществява от компилатора чрез дефиниране на служебен конструктор от вида:

```
X::X( const X& );
```

За клас String се дефинира следният служебен конструктор:

```
String::String( const String& s )
{
    len = s.len;
```

```

    str = s.str;
}

```

Съществуват три случая, при които се използва служебен конструктор:

1. При явно инициализиране на един обект с друг:

```

// String::String( char * );
String color ( "blue" );

//инициализиране чрез копиране на членовете
String mood = color;

```

2. При предаване на обект като аргумент на функция:

```

extern int count( String s, char ch );

// членовете на mood —> в членове на локалния обект s
int occurs = count( mood, 'e' );

```

3. При връщане на обект като резултат от изпълнение на функция:

```

extern String sub( String&, char, char );

main()
{
    String river( "mississippi" );
    cout << river << " " << sub( river, 'i', 'l' ) << "\n";
}

```

При предаване на аргумент от тип `reference` или връщане на резултат от този тип, не се извършва инициализиране чрез копиране на членове. Този факт се обяснява с това, че предаването на аргументи чрез адрес не създава локални копия на аргументите, за разлика от предаването чрез стойност. Предаването на аргумент чрез адрес се разглежда в раздел 3.6.

При инициализиране чрез копиране всеки член от стандартен или производен тип на единия обект се копира в съответния член на другия обект. Ако някой от членовете е обект, това правило се прилага рекурсивно. Например, клас `Word` притежава два члена: `occurs` от тип `int` и `name` от клас `String`. Да разгледаме следните две дефиниции на обекти от клас `Word`:

```

Word noun( "book" );
Word verb = noun;

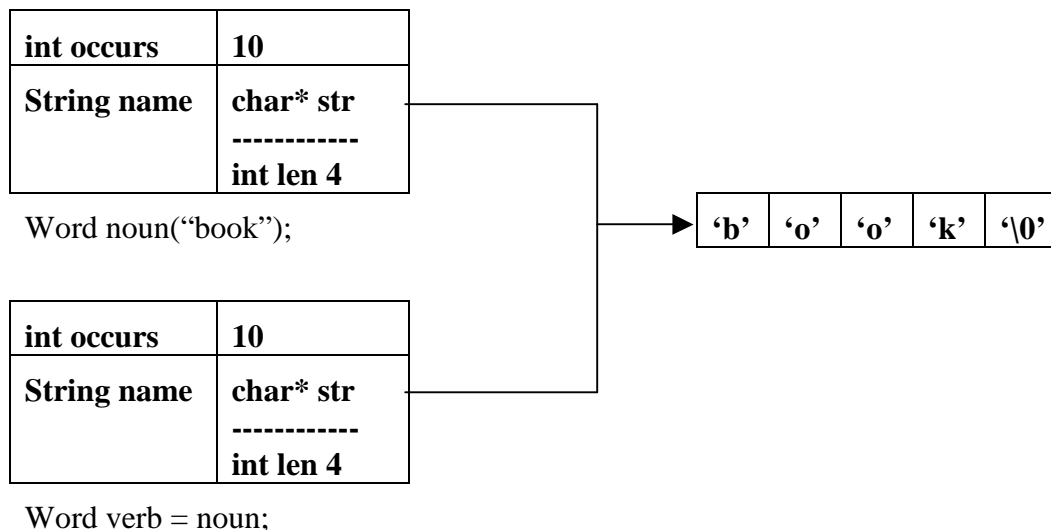
```

`verb` се инициализира на две стъпки:

1. Неговият член `occurs` се инициализира със стойността на `noun.occurs`.
2. Неговият член `name` се инициализира от служебния конструктор на клас `String`.

Дефинираният от компилатора служебен конструктор понякога не е достатъчен. Фиг. 6.1 изобразява паметта, заделена за `noun` и `verb`. Възникват два проблема:

1. Срещанията на noun не трябва да се копират във verb.occurs. На практика двете стойности не са свързани. Дефинираният от компилатора служебен конструктор нарушава семантиката на клас Word.
2. Членовете noun.str и verb.str адресират една и съща памет. Това може да създаде сериозни затруднения, ако двата обекта не напускат едновременно съответната област на действие.



Дефинираният от компилатора служебен конструктор не е достатъчен, когато класовете имат за член указател и следователно притежават деструктор. Деструкторът се извиква за всеки обект от класа, независимо от начина, по който е инициализиран. Паметта, която се използва от два или повече обекта може да се освободи няколко пъти. Това ще предизвика освобождаване на незаета памет или заета по-късно с друга цел памет. И в двата случая програмата ще греша. Решение на проблема е дефинирането на явен конструктор за инициализация чрез копиране.

Дефиниране на явен конструктор X(const X&)

Показахме, че има случаи, когато инициализирането на обекти чрез копиране изисква по-голям контрол от програмиста. Този допълнителен контрол може да се осъществи чрез дефиниране на явен конструктор от вида X(const X&). Когато този конструктор се дефинира в тялото на класа, той се извиква при всяко инициализиране на обект от класа с друг обект от същия клас. Например:

```
String::String( const String& s )
{
    len = s.len;
    str = new char[ len + 1 ];
    strcpy( str, s.str )
}
```

String(const String&) ще се извиква при всяко инициализиране на обект от клас String с друг обект от същия клас. Двата члена str ще адресират различна област от паметта.

Упражнение 6-7. Дефинирайте конструктор Screen(const Screen&) за дефинирания в глава 5 клас Screen. Посочете примери за трите случая, при които се използва този конструктор.

Упражнение 6-8. Дефинирайте конструктор `IntList(const IntList&)` за дефинирания в глава 4 клас `IntList`. Посочете примери за трите случая, при които се използва този конструктор.

X(const X&) и членове-обекти

В този подраздел ще разгледаме два случая на използване на `X(const X&)`, за класове с членове-обекти.

1. Явен конструктор `X(const X&)` за дадения клас не съществува, но такъв е дефиниран за негов член.
2. Даденият клас и класът на негов член притежават явен конструктор `X(const X&)`.

За илюстрация на първия случай ще използваме клас `Word`, за който не съществува конструктор `Word(const Word&)`. `Word` притежава член от клас `String`, а ние вече дефинирахме `String(const String&)`.

```
class Word {
public:
    Word( char *s, int cnt = 0 ): name(s), occurs(cnt) {}
    Word( String& s, int cnt = 0 ): name(s), occurs(cnt) {}
private:
    int occurs;
    String name;
};
```

Инициализирането на обект от клас `Word` с друг обект от същия клас се извършва чрез дефинирания от компилатора конструктор `Word(const Word&)`. Неговият член от клас `String` се инициализира чрез дефинирания от програмиста конструктор `String(const String&)`:

```
String mystery( "rosebud" );
Word resolve( mystery );

extern search( Word wd );
search( resolve );
```

`String(const String&)` се използва за инициализиране на членовете `resolve.name` и `wd.name`. В общия случай служебният конструктор се прилага рекурсивно за всеки член-обект, ако за неговия клас няма явно дефиниран конструктор `X(const X&)`.

Ако дефинираме конструктор `Word(const Word&)`, описаната процедура за инициализиране се променя. Дефинираният от нас конструктор `String(const String&)` не се извиква автоматично. При инициализиране на членовете на клас `Word` се взима в предвид съществуващата дефиниция на `Word(const Word&)`. Преди да покажем коректната дефиниция на този конструктор, да разгледаме следния неправилен вариант:

```
// тази реализация е неправилна
Word::Word( const Word& w )
{
    occurs = 0;
    name = w.name;
}
```

Този конструктор не инициализира коректно члена name. Да разгледаме следния пример:

```
Word weather( "worm" );  
Word feeling = weather;
```

Инициализацията на feeling преминава през следните етапи:

1. Проверява се дали съществува явно дефиниран конструктор Word(const Word&). Ако има такъв, той се извиква. В противен случай се използва служебният конструктор. В нашия случай има явно дефиниран конструктор.
2. Проверява се дали избраният конструктор има списък за инициализация на членове. В нашия случай такъв списък липсва.
3. Проверява се дали класът притежава членове-обекти. В случая name е от клас String.
4. Проверява се дали клас String притежава конструктор без аргументи. Ако не съществува такъв конструктор, се получава грешка при компилация. В противен случай се извиква този конструктор.
5. За инициализация на feeling.name се извиква String().
6. Извиква се Word(const Word&). Извършва се присвояването name = w.name. Подобно на инициализацията присвояването по подразбиране също се извършва чрез копиране. За подробности вижте раздел 6.3. В такъв случай String(const String&) няма да се извика.

В раздел 6.1 посочихме едно различие между фазите присвояване и инициализиране. Там споменахме, че константните членове и членовете от тип reference трябва да се инициализират чрез списъка за инициализация на членове. Тук ще покажем още една разлика между фазите инициализиране и присвояване. Ако искаме да инициализираме name с w.name, трябва да включим name в списъка за инициализация на членове. Коректната дефиниция на Word(const Word&) изглежда така:

```
Word::Word( const Word& w ): name (w.name)    // инициализация  
{ // присвояване  
    occurs = 0;  
}
```

Накрая ще резюмираме съдържанието на този подраздел. Ако даден клас съдържа членове-обекти, но не притежава явно дефиниран конструктор X(const X&), тези членове се инициализират чрез съответния служебен конструктор. Ако за някой член-обект съществува явно дефиниран конструктор X(const X&), той ще се извика за инициализиране на този член. Ако, обаче, даденият клас притежава явен конструктор X(const X&), инициализирането на членовете-обекти става негово задължение, като списъкът за инициализация на членове играе главна роля.

Упражнение 6-9. Дефинирайте Buffer(const Buffer&). (Виж упражнение 6-2.)

Упражнение 6-10. Дефинирайте конструкторите Inode(const Inode&) и BinTree(const BinTree&).

Резюме върху конструктори и деструктори

Конструкторите и деструкторите осъществяват автоматично инициализиране и освобождаване на обекти. Един и същи клас може да притежава няколко конструктора. По този начин за всеки клас се осигурява множество от операции за инициализиране. За по-голяма ефективност конструкторите могат да се дефинират като inline-функции.

Ако дефиницията на един клас съдържа членове-обекти от други класове, първо се извикват конструкторите, които инициализират членовете на класа и едва тогава конструкторът на самия клас. Ако членовете-обекти са няколко, техните конструктори се извикват поред на деклариране на членовете в тялото на класа. Редът на обръщение към деструкторите е обратен.

Конструкторите могат да използват т.нар. списък за инициализация на членове, за да предадат аргументи на конструкторите на съответните членове-обекти. Списъкът може да се използва и за инициализиране на членове, които не са обекти. Той е единственото средство за инициализиране на константни членове и членове от тип reference.

При инициализиране на току-що дефиниран обект с обект от същия клас, не се извиква никой от дефинираните конструктори. Такова инициализиране се нарича инициализиране чрез копиране на членове и се извършва чрез копиране на членовете на единия обект в съответните членове на другия обект. Ако класът притежава членове-обекти, инициализирането чрез копиране се прилага рекурсивно за всеки обект.

Инициализирането чрез копиране може да създаде проблеми за класове с членове, които са указатели. Тогава една и съща област от паметта може да се освободи няколко пъти. Проблемът се решава чрез дефиниране на явен конструктор `X(const X&)`, който управлява особените ситуации. При инициализиране на обект от клас `X` с друг обект от този клас, се извиква явно дефинираният конструктор.

6.3 Дефиниране на операции

В предишния раздел разгледахме механизма за автоматично инициализиране и освобождаване на обекти от клас `String`. От какви допълнителни функционални възможности се нуждае този клас?

Потребителят може да се нуждае от различни проверки за обектите на класа. Например, дали низът е празен или дали два низа са равни. Той може да поиска да провери дали един низ е подниз на друг. Низовете трябва да могат да се въвеждат и извеждат, да се присвояват един на друг, да се слепват. Трябва да може да се определя тяхната дължина, да се индексират техните символи и др. Следващата малка програма илюстрира използването на клас `String`:

```
String inBuf;
```

```
while ( readString( cin, inBuf )) {  
    if ( inBuf.isEmpty() ) return;  
    if ( inBuf.isEqual( "done" )) return;  
    switch ( inBuf.index(0) ) { /* ... */ }  
    cout << "Низът е ";  
    writeString( cout, inBuf );  
}
```

Примерът показва, че клас `String` не се използва толкова лесно, колкото стандартните типове данни. Макар че избраните имена са мнемонични, те се запомнят трудно. Същият програмен фрагмент може да се запише така:

```
String inBuf;

while ( cin >> inBuf ) {
    if ( !inBuf ) return;
    if ( inBuf == "done" ) return;
    switch ( inBuf[0] ) { /* ... */ }
    cout << "Низът е " << inBuf;
}
```

За да използваме такъв стил на програмиране, трябва да предефинираме съответните стандартни операции за работа с обекти от клас String. В този раздел ще дефинираме необходимите за клас String операции.

Начални сведения за предефинирането на операции

При създаване на клас, трябва да се осигурят операции за работа с неговите обекти. Функциите, които реализират тези операции, може да не са членове на класа, но трябва да имат поне един аргумент от този клас. Така се предотвратява опасността операциите да престанат да действат за стандартните типове данни. Операциите са "обикновени" функции с тази разлика, че името им се състои от ключовата дума operator, следвана от знак за стандартна операция. Например:

```
String& String::operator=( const String& s )
{ // операция за присвояване
    len = s.len;
    delete str; // освобождава предишен обект
    str = new char[ len + 1 ];
    strcpy( str, s.str );
    return *this;
}
```

След дефиниране на тази операция, при всяко присвояването на обекти от клас String ще се извършва обръщение към нея.

```
#include "String.h"

String atricle( "the" );
String common;

main() {
    // String::operator=( )
    common = atricle;
}
```

За един и същи клас може да се дефинират няколко операции с общо име, но с различна сигнатура. Например:

```
class String {
public:
    String &operator=( const String& );
    String &operator=( const char* );
    // ...
};
```

strcmp() е стандартна функция, която проверява дали два масива от символи са равни. Тя се използва в дефиницията на операцията за проверка на равенство между два обекта на String:

```
String::operator==( String& st ) {
    // strcmp() връща 0, ако двата низа са равни
    // при равенство на обектите operator== връща 1
    return( strcmp( str, st.str ) == 0 );
}
```

Стандартни операции и класове

Операциите, които се дефинират за класове, трябва да се представят от знак на съществуваща в C++ операция. Например, не може да се създаде операция за повдигане на степен и да се обозначи с "**". В таблица 6.1 е даден списъкът на позволените операции.

Стандартни операции в C++							
+	-	*	/	%	^	&	
~	!	,	=	<	>	<=	>=
++	--	<<	>>	==	!=	&&	
+=	-=	*=	/=	%=	^=	&=	=
<<=	>>=	[]	()	->	->*	new	delete

Табл. 6.1 Стандартни операции в C++

Съществуващите дефиниции на стандартни операции не могат да се отменят. Например, операцията събиране на цели числа не може да се замени с операция, която при събиране на числата проверява за препълване. Не може да се дефинират и допълнителни операции за стандартните типове данни. Например, не може да се дефинира операция за събиране на елементите на целочислен масив. Разрешава се само предефиниране на стандартните операции за класове. Това налага изискването всяка функция, която реализира операция, да притежава като аргумент обект от някакъв клас.

Съществуващият приоритет на операциите (раздел 2.9 разглежда този въпрос) не може да се отмени. Да разгледаме един пример:

```
x == y + z;
```

Въпреки че става дума за класове, първо ще се изпълни operator+, а след това operator==. Редът на изпълнение може да се промени чрез използване на кръгли скоби, както при стандартните типове данни.

Операциите, които се дефинират за класове, трябва да запазят броя на операндите. Например, логическото отрицание NOT ("!") е унарна операция и не може да се дефинира като бинарна за обектите от клас String. Следващата дефиниция на NOT е неправилна:

```
// неправилна дефиниция: "!" е унарна операция
operator!( String st1, String st2 )
```

```
{
    return( strcmp(st1.str, st2.str) != 0 );
}
```

Операциите "+", "-", "*" и "&" са унарни или бинарни. Съответните операции за класове също могат да бъдат унарни или бинарни.

Операциите за увеличаване и намаляване с единица "++" и "--" могат да имат само една дефиниция. Между префиксната и постфиксната им форма не се прави разлика.

Функциите, които реализират съответните операции, могат да не бъдат членове на класа. Да разгледаме операцията конкатенация (слепване) на два символни низа:

```
String st1( "cobble" );
String st2( "stone" );
String st3 = st1 + st2;
```

Тази операция може да не е член на клас String:

```
class String {
    friend String& operator+( String&, String& );
    // ...
};
```

Тя може да се дефинира и като член на класа:

```
class String {
public:
    String& operator+( String& );
    // ...
};
```

Едновременното дефиниране на двете операции води до противоречие:

```
class String {
    friend String& operator+( String&, String& );
public:
    String& operator+( String& );
    // ...
};

String a( "hobby" ), b( "horse" );
String c = a + b;    // грешка: противоречие
```

Всяка операция, която е член на класа, има един аргумент по-малко от съответната "приятелска" операция. Това е така, защото първият операнд на операцията-член е обекта, с който се извиква тя. Например,

```
st1 + st2
```

може да се реализира от функция-член на класа

```
st1.operator+( st2 )
```

или от функция, която не е член на този клас

```
operator+( st1, st2 )
```

Програмистът трябва да избере една от тези дефиниции. В противен случай ще се получи грешка при компилация, защото за компилатора не съществува критерий за избор на една от двете дефинирани функции.

Четири от операциите задължително трябва да бъдат членове на класа. Тези операции са "=" (присвояване), "[]" (индексиране), "()" (обръщение към функция) и "—>" (избор на член). В следващите подраздели ще разгледаме подробно всяка една от тях.

Всяка операция-член на клас изисква като ляв операнд обект от този клас. Операцията не може да е член на класа, ако левият и операнд не е обект от класа. Тя трябва да се декларира като приятелска, ако използва скритите членове на класа. Например:

```
class String {  
    friend ostream& operator<<( ostream& os, String s );  
    { return ( os << s.str); }  
    // ...  
};
```

Упражнение 6-11. Определете прототипа на функцията, която реализира унарната операция "+" — първо, ако тя е член на клас String и второ, ако тя е негова приятелска функция.

Упражнение 6-12. Логическата операция NOT връща 1, ако низът е празен; в противен случай връща 0. Дефинирайте NOT за клас String.

Упражнение 6-13. Бинарната операция "+" слепва два символни низа. Дефинирайте тази операция, но нека тя не е член на клас String.

Упражнение 6-14. Дефинирайте бинарната операция "+" като член на клас String.

Операция за индексиране "[]"

Потребителите на клас String трябва да могат да прочитат и да променят отделни символи на низа. Сега ще създадем средства за достъп до символите чрез индексиране:

```
String sentence( "Ash on an old man's sleeve." );  
String tempBuf( sentence.getLength() );  
  
for ( int i = 0; i < sentence.getLength(); ++i )  
    tempBuf[ i ] = sentence[ i ];
```

String::getLen() е проста функция, която връща дължината на низа:

```
inline String::getLen()  
{  
    return len;
```



```
}
```

Коректната дефиниция на операцията за индексирание трябва да позволява използване на операцията от двете страни на знака "=". За тази цел функцията трябва да връща като резултат адрес от паметта, където се съхранява един символ, т.е. резултатът трябва да е lvalue. Следователно, типът на резултата трябва да бъде reference:

```
inline char& String::operator[]( int elem )
{
    checkBounds( elem );
    return str[ elem ];
}
```

Функцията връща адреса на символа с индекс elem. В този вид тя може да се използва и отляво на операцията за присвояване. Например,

```
String st( "mauve" );
st[ 0 ] = 'M';
```

присвоява на първия символ от низа символната константа 'M'.

checkBounds() проверява дали посочения индекс е в границите на текущия символен низ. Ако индексът не е в необходимите граници, функцията извежда съобщение за грешка.

```
// включва прототипа на exit( int )
#include <stdlib.h>

void String::checkBounds( int elem )
{ // В границите на низа ли е индекса?
    if ( elem < 0 || elem >= len ) {
        cerr << "\nИндексът е извън границите! индекс: "
              << elem << "границите са 0 и "
              << len - 1 << "\n";
        exit( -1 );
    }
}
```

Операция за обръщение към функция "()"

Понякога трябва да се обходят всички елементи на низа. Затова ще създадем функция, която при всяко обръщение връща следващия символ от низа до достигане на неговия край. Следващият програмен фрагмент показва как може да се използва тази функция:

```
String inBuf;

// прочита обект от клас Screen
while ( cin >> inBuf ) {
    char ch;
    // обхожда символите от низа
    while ( ch = inBuf() )
        // ... действия със ch
}
```

В този подраздел ще изясним как могат да се реализират използваните по-горе операции ">>" и "()".

Операцията за вход не е член на клас String. Тя се дефинира така:

```
istream& operator>>( istream& is, String& s )
{
    char inBuf[ STRING_SIZE ];

    is >> inBuf;
    s = inBuf;      // String::operator=( char* )
    return is;
}
```

Операцията ">>" присвоява на обект от клас String съдържанието на променливата inBuf. Присвояването се извършва чрез операцията String::operator=(char*).

```
String& String::operator=( const char *s )
{
    len = strlen( s );
    delete str;
    str = new char[ len + 1 ];
    strcpy( str, s );
    return *this;
}
```

Операцията за обръщение към функция осигурява лесен начин за обхождане на низа. При достигане на неговия край, тя връща стойност 0. Следователно изрази като

```
while ( ch = inBuf() ) // ...
```

са допустими. За да реализираме тази операция, ще въведем още един член на клас String — index, който адресира следващия елемент на низа. Тогава всеки конструктор на класа трябва да инициализира index с 0.

```
char String::operator()() {
    if ( index < len )
        return str[ index++ ];
    // ако стигнем до тук, трябва да спрем обхождането
    return ( index = 0 );
}
```

Следващата малка програма демонстрира употребата на операциите за вход и изход, индексирание и обхождане на низ:

```
#include "String.h"

const LINESIZE = 40;
enum { PERIOD='.', COMMA=',', SEMI=';', COLON=':' };

main() {
    String inBuf( STRING_SIZE );
```

```

int lineSize = 0;

// operator>>( istream&, String& )
while ( cin >> inBuf ) {
    char ch;
    int i = 0;

    // String::operator()()
    while ( ch = inBuf() ) {
        switch (ch) {
            case PERIOD:
            case COMMA:
            case SEMI:
            case COLON:
                // String::operator[](int)
                intBuf[ i ] = '\0'; break;
        };
        ++i; ++lineSize;
    }
    if ( lineSize >= LINESIZE ) {
        cout << "\n";
        lineSize = 0;
    }
    cout << inBuf << " ";
}
cout << "\n";
}

```

Ще използваме следния вход:

```

We were her pride of ten; she named us: benjamin
phoenix, the prodigal, and perspicacious, pacific
Suzanne. Benjamin, hust now. Be still, child.
People are never just.

```

След компилация и изпълнение получаваме такъв резултат:

```

We were her pride of ten she named us
benjamin phoenix the prodigal and perspicacious
pacific Suzanne Benjamin hust now Be still child
People are never just

```

Операция new и операция delete

В общия случай паметта за обектите на клас се заделя от стандартната операция new. (Нейното описание е дадено в раздел 4.1.) Следващата програма прочита от терминала поредица от думи и ги сортира по дължина. Тя използва услугите на клас StringList.

```

#include "String.h"
#include "StringList.h"

// поддържа се масив от указатели
const maxLen = 25;
StringList *stbl[ maxLen ];

main()

```

```

{ // прочита низовете и ги сортира по дължина
  const inBuf = 512;
  char st[ inBuf ];
  StringList *p;

  while ( cin >> st ) {
    p = new StringList( st );
    int sz = p->getLen();
    if ( sz >= maxLen )
      // извежда съобщение за грешка
      continue;
    p->next = stbl[ sz ];
    stbl[ i ] = p;
  }

  for ( int i = maxLen - 1; i > 0; --i ) {
    StringList *tp;
    p = stbl[ i ];
    while ( p != 0 ) {
      cout << *p << "\n";
      tp = p;
      p = p->next;
      delete tp;
    }
  }
}

```

StringList притежава два члена-данни: entry — обект от клас String и next — указател към обект от клас StringList. Конструкторът на класа изисква един аргумент от тип char*. Този аргумент се предава на конструктора на клас String чрез списъка за инициализация на членове.

```
StringList::StringList( char *s ): entry( s ), next( 0 ) { }
```

getLen() е метод за достъп. Той връща дължината на entry:

```
StringList::getLen()
{
  return entry.getLen();
}
```

StringList няма достъп до скритите членове на клас String. Затова неговата функция getLen() използва същата функция на клас String. Следващата дефиниция на StringList::getLen() е неправилна:

```
StringList::getLen()
{
  // грешка: len е скрит член на клас String
  return entry.len;
}
```

Трябва да се дефинира и операция за извеждане на обекти от клас StringList:

```
ostream& operator<<( ostream& os, StringList& s )
{
    return ( os << s.entry );
}
```

Програмата използва следния вход:

```
A class may provide new and delete
operator functions
```

След компилация и изпълнение получаваме такъв резултат:

```
functions
operator
provide
delete
class
and
new
may
A
```

Уверихме се, че програмата работи правилно. Ще се опитаме да я оптимизираме, като подобрим управлението на паметта. Досега след всяко прочитане на дума се заделя памет за един обект от клас `StringList`. Ако намалим броя на обръщенията към `new` — на всяка 24-та дума, например — ще ускорим програмата. Ще създадем нови операции `new` и `delete` за клас `StringList`. При това програмата няма да се промени.

За обектите от клас `StringList` ще заделяме памет на порции. Всяко заделяне на памет осигурява място за `stringChunk` на брой обекта. С тази памет ще работим като със свързан списък. Следва дефиницията на `freeStore` и `stringChunk`:

```
class StringList {
// ...
private:
    enum { stringChunk = 24 };
    static StringList *freeStore;
// ...
};
```

`stringChunk` определя за колко обекта е предназначена заделената наведнъж памет. `freeStore` е остатъкът от заделеното парче памет, след като поредната дума е взела от него. `freeStore` е свързан списък. Той е статичен член на клас `StringList`.

Дефинираната от нас операция `new` трябва да връща резултат от тип `void*` и да има един аргумент от тип `size_t`. Този тип е дефиниран в системния заглавен файл `stddef.h`. Единственият аргумент на `new` се инициализира автоматично от компилатора с размера на класа в байтове. Могатда се дефинират и други варианти на операцията `new`, стига всеки от тях да има уникална сигнатура. В раздел 4.3 е дадена друга дефиниция на тази операция. Когато `new` се прилага към името на клас, компилаторът проверява дали съществува специално дефинирана за класа операция `new`. Ако има такава операция, той я изпълнява. В

противен случай той изпълнява стандартната операция new. Добавяне или премахване на специално дефинирана за класа операция new не променя програмата.

Ще дефинираме new като член на клас StringList. Операцията проверява дали във freeStore има място за поредния обект от класа. Ако отговорът е положителен, тя връща адреса на обекта. В противен случай стандартната операция new заделя памет за stringChunk на брой обекта, т.е. генерира ново парче памет freeStore.

```
#include <stddef.h>

StringList *StringList::freeStore = 0;

void *StringList::operator new( size_t size )
{
    register StringList *p;
    // ако заделената памет не е достатъчна, заделя ново парче памет
    if ( !freeStore ) {
        long sz = stringChunk * size;
        freeStore = p =
            // използва се стандартната операция new
            (StringList *)new char[ sz ];

        // freeStore се инициализира
        for ( ; p != &freeStore[ stringChunk - 1 ]; p—>next = p + 1, p++ );
        p—>next = 0;
    }
    p = freeStore;
    freeStore = freeStore—>next;
    return p;
}
```

Операцията delete ще бъде също член на клас StringList. Тя връща заетата от обект от класа памет към freeStore:

```
void StringList::operator delete( void *p, size_t )
{
    // връща p на freeStore
    ((StringList*)p)—>next = freeStore;
    freeStore = (StringList *)p;
}
```

Първият аргумент на delete трябва да е от тип void*. Може да се зададе и втори аргумент, чиито тип е size_t (тогава трябва да се включи файла stddef.h). Ако има втори аргумент, той се инициализира неявно с дължината на обекта, адресиран от първия аргумент. Дължината се задава в байтове. delete трябва да връща резултат от тип void.

Току-що дефинираната операция new се извиква само за отделен обект от клас StringList, но не и за масив от обекти. Например,

```
StringList *p = new StringList;
```

извиква дефинираната от нас операция new, докато

```
StringList *pia = new StringList[10];
```

извиква стандартната операция new, която резервира памет за масив от обекти. Забележете още, че следващата дефиниция на обект

```
StringList s;
```

не предизвиква обръщение към никоя от операциите new.

Ако има дефинирана операция new, но за конкретен обект искаме да използваме стандартната операция new, трябва да посочим нейната област на действие. Например:

```
StringList *ps = ::new StringList;
```

Аналогично,

```
::delete ps;
```

извиква стандартната операция delete.

Операциите new и delete се дефинират като статични членове на съответния клас и затова се подчиняват на ограниченията за този вид членове. Ще припомним, че статичните методи не поддържат указателя this и имат достъп само до статичните данни на класа. За подробности виж раздел 5.6. Тези операции се дефинират като статични, защото се извикват или преди съществуването на обекта (операция new), или след неговото унищожаване (операция delete).

X::Operator=(const X&)

Ако на един обект от даден клас се присвои друг обект от същия клас, нестатичните членове на единия обект се копират в съответните членове на другия обект. Използва се същият механизъм като при инициализиране чрез копиране (виж раздел 6.2). За целта компилаторът генерира служебна операция за присвояване от вида:

```
X& X::operator=( const X& );
```

Тази операция се използва по подразбиране, ако на един обект се присвои друг обект от същия клас. Нека имаме следните две дефиниции на обекти от клас String:

```
String article( "the" );  
String common( "For example" );
```

Присвояването

```
common = article;
```

се реализира чрез служебната операция за присвояване. Ето нейната дефиниция:

```
String& String::operator=( const String& s )
{
    len = s.len;
    str = s.str;
    index = s.index;
}
```

Използването на тази операция поражда няколко проблема:

1. Двата обекта `article` и `common` заемат една и съща област от паметта. Техните деструктори ще се прилагат за една и съща памет.
2. Заделената за "For example" памет никога няма да се освободи изцяло. Част от нея ще се загуби завинаги при операцията за присвояване.
3. Семантиката на клас `String` изключва копиране на всички данни на класа. Дефинираният в раздел 6.3 член `index`, който се използва при обхождане на низа, трябва да има стойност 0 за обекта отляво на `"="`. Следователно, в него не трябва да се копира стойност. Служебната операция за присвояване нарушава семантиката на клас `String`.

Тези проблеми могат да се решат, ако за клас `String` се създаде явна операция за присвояване. Например:

```
String& String::operator=( const String& s )
{
    index = 0;
    len = s.len;
    delete str;
    str = new char[ len + 1 ];
    strcpy( str, s.str );
    return *this;
}
```

Дефинираният в предишния раздел клас `StringList` има член, който е обект от клас `String`. `StringList` не притежава явно дефинирана операция за присвояване. Ако един обект от `StringList` се присвои на друг обект от този клас, за члена с име `entry` се извиква неявно `String::operator=(const String&)`. Например:

```
#include "StringList.h"

main() {
    StringList sl1( "horse" );
    StringList sl2( "carriage" );

    // sl1.next = sl2.next
    // sl1.entry.String::operator=(sl2.entry)
    sl1 = sl2;
}
```

Ако `StringList` притежава собствена операция за присвояване, `String::operator=(const String&)` се извиква само ако в тялото на тази операция съществува явно обръщение към операцията на клас `String`. Например:


```
StringList& StringList::operator=( const StringList& s )
{
    // String::operator=(const String&)
    entry = s.entry;
    next = 0;
    return *this;
}
```

Ако липсва присвояването

```
entry = s.entry;
```

String::operator=(const String&) няма да се извика. entry няма да се промени.

Начинаещите програмисти често не правят разлика между инициализация и присвояване. Това може да доведе до създаване на неефективни програми. Да разгледаме следните два класа:

```
class X {
public:
    X();
    X( int );
    X( const X& );
    X& operator=( const X& );
    // ...
};

class Y {
public:
    Y();
private:
    X x;
};
```

Следващата проста реализация на конструктора на клас Y предизвиква обръщение към два конструктора на клас X и към операцията за присвояване X::operator=(const X&):

```
Y::Y() { x = 0; }
```

Да разгледаме този пример по-подробно:

1. Преди конструкторът на клас Y да инициализира своя член-обект x, ще се извика конструктора по подразбиране X::X(). На x не може да се присвои директно 0, защото клас X не разполага с операция за присвояване с аргумент от тип int. Присвояването се извършва на две стъпки.
2. Числото 0 се преобразува в обект от клас X чрез конструктора

```
X::X( int )
```

Използването на конструктори за преобразуване на типове е тема на раздел 6.5.

3. Новосъздаденият обект от клас X се присвоява на обекта x чрез обръщение към:

```
X::operator=( const X& )
```

Втората и третата стъпка са излишни, ако конструкторът на клас Y се дефинира така:

```
Y::Y() : x( 0 ) {}
```

Тогава всяко обръщение към конструктора Y::Y() ще извиква само

```
X::X( int )
```

Операция за избор на член "—>"

Операцията "—>" е унарна операция на своя ляв операнд. Този операнд може да бъде обект или алтернативно име на обект. Операцията трябва да връща указател към обект или обект, за който такава операция е дефинирана. В следващия пример операцията "—>" връща указател към обект от клас String:

```
#include "String.h"

class Xample {
public:
    String *operator—>();
    // ...
private:
    String *ps;
    // ...
};
```

Ще разгледаме една примерна реализация на тази операция. След обработка на адресирания от ps обект, операцията връща ps.

```
String* Xample::operator—>() {
    if ( ps == 0 )    // ... ps се инициализира
    // ... обработка на адресирания от ps обект
    return ps;
}
```

Тази операция може да се извика от обект от клас Xample или от алтернативно име на такъв обект. Например:

```
void ff( Xample x, Xample &xr, Xample *xp ) {
    int i;

    // обръщение към String::getLen()
    i = x—>getLen();        // ok: x.ps—>getLen()
    i = xr—>getLen();        // ok: xr.ps—>getLen()
    i = xp—>getLen();        // грешка: няма Xample::getLen()
}
```

Операцията " \longrightarrow " не може да се извика с указател към обект от клас `Xample`, както е в третия случай, защото компилаторът не различава стандартната операция за избор на член и тази, дефинирана от потребителя.

Определяне на необходимите операции

За всеки клас са дефинирани само операции за извличане на адрес "&" и присвояване "=". Всяка друга операция, която има смисъл за класа, трябва да се дефинира явно от потребителя. Какви операции ще бъдат дефинирани зависи от предназначението на класа.

При създаване на клас се започва от общодостъпния интерфейс. След определяне на общодостъпните функции-членове на класа, може да се прецени какви операции са необходими.

Всяка операция се свързва със стандартно действие. Например, бинарният "+" се идентифицира като операция за събиране. При дефиниране на операция "+" за клас, тя трябва да се свърже с действие, аналогично на събирането. Например, конкатенацията на низове добавя един низ към друг и затова е подходящо разширение на "+".

След дефиниране на общодостъпния интерфейс на класа, трябва да се потърси съответствие между смисъла на стандартните операции и действията, реализирани от функциите-членове. Например, `isEmpty()` съответства на логическото отрицание NOT, `operator!()`; `isEqual()` съответства на операцията за проверка на равенство "=", `operator==()`; `copy()` съответства на операцията за присвояване, `operator=()` и т.н.

За стандартните типове данни има операции, които са съкратен запис на две последователни операции. Операции за съкратен запис може да се дефинират и за класове, но това трябва да стане явно. Нека клас `String` притежава операцията конкатенация и присвояване чрез копиране на членове:

```
String s1( "C" );  
String s2( "++" );  
  
s1 = s1 + s2;      // s1 ← "C++"
```

Наличието на "+" и "=" не означава, че може да се използва операцията "+=". Следният съкратен запис е неправилен:

```
s1 += s2;
```

Ако искаме да използваме този съкратен запис, трябва да дефинираме явно операцията "+=".

Упражнение 6-15. Кои функции-членове на дефинирания в глава 5 клас `Screen` са подходящи за реализиране като операции.

Упражнение 6-16. Реализирайте определените в предишното упражнение функции като операции.

Упражнение 6-17. В раздел 4.2 дефинирахме клас `IntList`. Кои негови функции-членове са подходящи за реализиране като операции.

Клас `BinTree` трябва да реализира функциите: `isEmpty()`, `isEqual()`, `print()`, `addNode()`, `build()`, `copy()` и `deleteNode()`.

Упражнение 6-18. Кои от тези функции могат да се реализират като операции?

Упражнение 6-19. Дефинирайте изброените по-горе функции за клас BinTree.

Упражнение 6-20. При работа с обекти от клас BinTree се генерират обекти от клас INode. Като използвате операциите new и delete реализирайте по-добро управление на паметта за клас INode.

6.4 Примерна реализация на клас BitVector

В раздел 2.8 дадохме определение на маска. Маската е поредица от битове, в която всеки бит пази информация за наличие или отсъствие на някакъв признак. В предишния раздел използвахме клас String, за да се запознаем с дефинирането на операции. Сега ще дефинираме клас BitVector, за да покажем как се избират функциите, които се реализират като операции. Вътрешното представяне на клас BitVector се описва така:

```
typedef unsigned int BitVecType;
typedef BitVecType *BitVec;

class BitVector {
private:
    BitVec bv;
    unsigned short size;
    short wordWidth;
};
```

size съдържа броя битове, от които е съставен текущият обект BitVector. bv адресира поредицата битове. Тя представлява последователност от едно или повече числа от тип unsigned int. wordWidth определя броя на числата unsigned int, адресирани от bv. Например, ако unsigned int заема 32 бита, а обектът от клас BitVector е съставен от 16 бита, size и wordWidth ще получат следните стойности:

```
// брой на битовете в текущия BitVector
size = 16;

// брой на числата unsigned int за текущия BitVector
wordWidth = 1;
```

BitVector, който се състои от 107 бита, не може да се представи с по-малко от четири числа unsigned int. size и wordWidth ще имат следните стойности:

```
size = 107;
wordWidth = 4;
```

В двата случая bv се инициализира с адреса на масив от тип BitVecType с дължина wordWidth:

```
bv = new BitVecType[ wordWidth ];
```

Обикновено стандартните типове данни се представят с различен брой битове и байтове при различните машини. Затова е добре да изчистим кода на програмите от тези машинно-зависими стойности:

```
#ifdef vax
const BITSPERBYTE = 8;
const BYTESPERWORD = 4;
#endif

#ifdef sun
const BITSPERBYTE = 8;
const BYTESPERWORD = 2;
#endif

// брой битове за променлива от тип int
const WORDSIZE = BITSPERBYTE * BYTESPERWORD;
```

Ако желае, потребителят може да избере дължина на маската. Нека по подразбиране стойността на `size` е равна на броя битове за число от тип `unsigned int`. Следва дефиницията на конструктора на клас `BitVector`:

```
enum { OFF, ON };

// стойности по подразбиране: sz —> WORDSIZE, init —> OFF
BitVector::BitVector( int sz, int init ) {
    size = sz;
    wordWidth = (size + WORDSIZE - 1) / WORDSIZE;
    bv = new BitVecType[ wordWidth ];

    // инициализира bv изцяло с 0-ли или 1-ци
    if ( init != OFF ) init = ~0;
    // присвоява 0 или 1 на всяка дума от bv
    for ( int i = 0; i < wordWidth; i++ )
        *(bv + i) = init;
}
```

Потребителят трябва да може да установи отделен бит в 0 или 1 чрез функциите `unset()` и `set()`. Вместо тези функции ще дефинираме бинарни операции с операнди обект от клас `BitVector` и цяло число, показващо бита, който трябва да се промени. За да установим конкретен бит в 1, може да използваме следния запис:

```
BitVector bvec;

// възможни операции за set()
bvec | 12;
bvec |= 12;
bvec + 12;
bvec += 12;
```

За установяване на даден бит в единица е по-просто да се използва операцията събиране "+", отколкото операцията побитово ИЛИ. Как да изберем, обаче, между "+" и "+=" ? А може би е по-добре да реализираме и двете операции? Какво в действителност правят те?

Ако `set()` не е реализирана като операция, към нея се обръщаме така:

```
bvec.set(12);
```

След това обръщение 12-тият бит на `bvec` получава стойност 1. Следователно, левият операнд на израза съхранява резултата от него. Това не е присъщо за операцията "+", но отговаря на поведението на операцията "+=". Така направихме избор на операция. Ето нейната реализация:

```
void BitVector::operator+=( int pos )
{ // установява в 1-ца бита с номер pos
  checkBounds( pos );
  int index = getIndex( pos );
  int offset = getOffset( pos );

  // установява в 1-ца бита с номер offset в думата с номер index
  *(bv + index) |= ( ON << offset );
}
```

`getIndex()` и `getOffset()` са скрити (private) помощни функции на клас `BitVector`. `getIndex()` връща номера на думата, в която се намира даден бит. Например, за 16-тия бит тя връща 0; за 33-тия бит 1; за 107-мия бит 3. Ето нейната дефиниция:

```
inline BitVector::getIndex( int pos );
{
  // връща номера на думата, в която е бита с номер pos
  return( pos / WORDSIZE );
}
```

`getOffset()` връща позицията на бита в определената от предишната функция дума. Например, за 16-тия бит тя връща 16; за 33-тия бит 1; за 107-мия бит 11. Нейната реализация изглежда така:

```
inline BitVector::getOffset( int pos );
{
  // връща позицията на бита в думата, определена от getIndex()
  return( pos % WORDSIZE );
}
```

Дефиницията на `BitVector::operator-=(int)` е аналогична на дефиницията на операция "+=". Разликата е в начина на нулиране на съответния бит:

```
// битът в думата с номер index на позиция offset получава стойност 0
*(bv + index) &= ~( ON << offset );
```

На потребителя не трябва да се разрешава да излиза от границите на текущия обект от клас `BitVector`. Например, следващото присвояване трябва да се маркира като недопустимо:

```
BitVector bvec( 16 );
```

```
bvec += 18;          // грешка: излизане извън границите
```

Една примерна реализация на функцията `checkBounds()` изглежда така:

```
void BitVector::checkBounds( int index )
{
    // проверява дали index е в границите на текущия обект от клас BitVector
    if ( index < 0 || index >= size )
    {
        cerr << " \nИндексът е извън границите: "
              << "<< " << index << ", дължина: "
              << size << " >\n";
        exit(-1); // декларирана в stdlib.h
    }
}
```

Потребителите на клас `BitVector` трябва да могат да проверяват каква е стойността на даден бит. `isOn()` и `isOff()` са бинарни операции, чиито ляв операнд е обект от клас `BitVector`. Десният операнд е цяло число. То показва номера на съответния бит. Тези две функции се изразяват съответно с операциите `"= "` и `"!= "`. Например, дали 17-тият бит е 1 може да се провери така:

```
BitVector bvec;
if ( bvec == 17 )
    // ...
```

Следва реализацията на тази операция.

```
BitVector::operator==( int pos )
{
    // връща истина, ако битът на позиция pos е 1
    checkBounds( pos );
    int index = getIndex( pos );
    int offset = getOffset( pos );
    return( *(bv + index) & ( ON << offset ) );
}
```

Дефиницията на операцията `"!= "` използва операцията `"= "`:

```
BitVector::operator!=( int pos )
{
    // връща истина, ако битът на позиция pos е 0
    return ( !(*this == pos) );
}
```

Обектите от клас `BitVector` трябва да се извеждат, както се извеждат стойностите на променливи от стандартните типове данни. За целта за клас `BitVector` ще дефинираме операция за изход `"<< "`. Тогава може да използваме такива изрази:

```
cout << "my BitVector: " << bv << "\n";
```

Ако 7, 12, 19, 27 и 34 бит имат стойност 1, на екрана ще се появи следният печат:

< 7 12 19 27 34 >

Когато няма бит със стойност 1, ще получим следния резултат:

< (none) >

Нека дефинираме операцията "<<":

```
ostream& operator<<( ostream& os, BitVector& bv )
{
    int lineSize = 12;
    os << "\n\t ";
    for ( int cnt = 0, i = 0; i < bv.size; ++i )
    {
        // BitVector::operator==(int)
        if ( bv == i )
        { // следи за форматиран изход на екрана
            int lineCnt = cnt++ % lineSize;
            if ( lineCnt == 0 && cnt != 1 ) os << "\n\t ";
            os << i << " ";
        }
    }
    if ( cnt == 0 ) os << "(none) ";
    os << ">\n";
    return os;
}
```

Има операции, които трябва да останат "обикновени" функции. Например, за да използваме отново даден обект от клас BitVector, трябва да инициализираме всичките му битове с нули — функцията reset(). reset() може да се дефинира като унарна операция с един операнд от клас BitVector. Подходяща е операцията логическо отрицание:

```
BitVector& BitVector::operator!()
{ // инициализира всички битове на обекта с 0
    for ( int i = 0; i < wordWidth; i++ )
        *( bv + i ) = 0;
    return *this;
}
```

Съществува вероятност тази операция да се използва неправилно, защото:

- Стандартната операция за логическо отрицание връща истина, ако нейният операнд има стойност 0. Тя не променя своя операнд.
- Дефинираната от нас операция за логическо отрицание връща своя операнд. Тя присвоява на всички негови битове стойност 0.

Следващият пример показва една вероятна грешка. Употребата на дефинираната от нас операция "!" е сбъркана с употребата на стандартна операция "!". testResults() връща като

резултат от тестови проби обект от клас BitVector. Конкретен бит от този обект има стойност 1, ако съответният тест се е провалил.

```
if ( !(bvec = testResults() ) )
    cout << "Всички тестове са извършени!\n";
else
    reportFailures( bvec );
// ...
```

Вместо операцията "!" по-добре да се използва функцията reset(). Тя е член на клас BitVector и има един аргумент, който при обръщение към нея стои отляво.

Обектите от клас BitVector се използват от оптимизатора на компилаторите за анализ на данните в програмите. При този процес често се използват операциите побитово И и побитово ИЛИ. Те са основни за клас BitVector. Да разгледаме тяхната реализация. За простота ще приемем, че двата обекта се състоят от еднакъв брой битове.

```
#include "BitVector.h"

BitVector BitVector::operator|(BitVector& b)
{ // за простота двата обекта са с еднаква дължина
  BitVector result(size, OFF);
  BitVec tmp = result.bv;
  for ( int i = 0; i < wordWidth; i++ )
      *(tmp + i) = *(bv + i) | *(b.bv + i);
  return result;
}
```

Дефиницията на побитово И "&" е аналогична. Единственият различен оператор е:

```
*(tmp + i) = *(bv + i) & *(b.bv + i);
```

Фигура 6.2 съдържа малка интерактивна програма, която използва клас BitVector, за да съхрани атрибутите на тип. Ако въведем

```
unsigned const char * ;
```

четири бита на обекта typeFlag ще получат стойност 1. За да опростим програмата, всеки низ ще завършва със символа ";", предшестван от един интервал. След компилация и изпълнение на програмата получаваме следния резултат:

```
Типът в декларацията завършва със символа ";",
предшестван от интервал. Например,
'unsigned const char * ;'
За изход от програмата използвайте ctrl-d.
```

```
unsigned const char * ;
```

стойност 1 имат флаговете за:

```
unsigned
const
*
```

char

Фигура 6.3 съдържа описание на заглавния файл BitVector.h.

```
#include <string.h>
#include "BitVector.h"

const int MAXBITS = 8;

enum { ERROR, CHAR, SHORT, INT, PTR,
      REFERENCE, CONST, UNSIGNED };

static char *typeTbl[] {
    "OOPS, няма тип за индекс 0",
    "char", "short", "int",
    "*", "&", "const", "unsigned" };

static char *msg =
    "Типът в декларацията завършва със символа ";", \
    \nпредшестван от интервал. Например, \
    \n\t'unsigned const char *;'\ \
    \nЗа изход от програмата използвайте ctrl-d.\n";

main() {
    BitVector typeFlag( MAXBITS );
    char buf[ 1024 ];

    cout << msg;
    while ( cin >> buf ) {
        for ( int i = 0; i < wordWidth; i++ )
            if ( strcmp(typeTbl[i], buf) == 0 )
                { // BitVector::operator+=(int)
                  typeFlag += i; break;
                }

        if ( buf[0] = ';' ) { // край на входа
            cout << "\nстойност 1 имат флаговете за:\n\t";

            for( i = MAXBITS - 1; i > 0; i-- )
                // BitVector::operator==(int)
                if ( typeFlag == i )
                    cout << typeTbl[i] << "\n\t";
            cout << "\n";

            // повторна инициализация: BitVector::reset()
            typeFlag.reset();
        }
    }
}
```

Фиг. 6.2 Примерна програма с клас **BitVector**

```
#ifndef BITVECTOR_H
#define BITVECTOR_H

#ifdef vax
const BITSPERBYTE = 8;
const BYTESPERWORD = 4;
#endif
```

```

#ifdef sun
const BITSPERBYTE = 8;
const BYTESPERWORD = 2;
#endif

const WORDSIZE = BITSPERBYTE * BYTESPERWORD;
enum { OFF, ON };
typedef unsigned int BitVecType;
typedef BitVecType *BitVec;

#include <iostream.h>

class BitVector {
friend ostream&
    operator<<(ostream&, BitVector&);
public:
    BitVector( int = WORDSIZE, int = OFF );
    ~BitVector() { delete [ wordWidth ] bv; }
    void operator+=( int pos ); // бит pos става 1
    void operator-=( int pos ); // бит pos става 0
    BitVector operator & (BitVector& );
    BitVector operator | (BitVector& );
    operator == ( int pos );      // 1 ли е бит с номер pos?
    operator != ( int pos );      // 0 ли е бит с номер pos?
    void reset();                 // всички битове отново стават 0
private:
    void checkBounds( int);
    inline getIndex( pos );
    inline getOffset( pos );
private: // Ÿel°Гë-© ĩ°Г±l Ÿi-Г
    BitVec bv;
    unsigned short size;
    short wordWidth;
};

```

Фиг. 6.3 Съдържание на заглавния файл **BitVector.h**

Упражнение 6-21. Модифицирайте програмата от фигура 6.2, за да може да обработва и следния вход:

```
const char * const;
```

Упражнение 6-22. Дефинирайте `BitVector::operator=(const X&)` и `BitVector::X(const X&)`.

Упражнение 6-23. Дефинирайте операция, която проверява дали два обекта от клас `BitVector` са еднакви.

Упражнение 6-24. Дефинирайте отново операцията "&", като отстраните ограничението за еднаква дължина на двата обекта.

Упражнение 6-25. Дефинирайте отново операцията "|", като отстраните ограничението за еднаква дължина на двата обекта.

Упражнение 6-26. Модифицирайте дефинираната за клас `BitVector` операция ">>", така че вместо следния печат

```
< 0, 1, 2, 3, 4, 5, 6, 7 >
```

< 0, 2, 3, 4, 7, 8, 9, 12 >

да получаваме

< 0 - 7 >

< 0, 2 - 4, 7 - 9, 12 >

6.5 Операции за преобразуване

Стандартните преобразувания за стандартните типове данни предотвратяват създаването на голям брой функции и операции с еднакви имена за различни типове данни. Например, без стандартните аритметични преобразувания всяка от следващите шест операции щеше да се нуждае от своя дефиниция:

```
char ch;  
short sh;  
int ival;  
  
// без стандартни преобразувания трябва да се  
// дефинират шест операции за събиране  
ch + ival;      ival + ch;  
ch + sh;        sh + ch;  
ival + sh;      sh + ival;
```

В този пример стандартно преобразуване превръща типа на всеки операнд в `int`. Това означава, че е достатъчно да се дефинира само една операция "+", която събира числа от тип `int`. Всеки път, когато е необходимо, стандартните преобразувания се изпълняват неявно от компилатора.

В този раздел ще разгледаме как се дефинират преобразувания за произволен клас. При необходимост, те също се извикват неявно от компилатора. За илюстрация ще използваме клас `SmallInt`.

Клас `SmallInt` представя осембитови числа без знак, т.е. числа от тип `unsigned char`. Допълнително свойство на неговите обекти е възможността да се следи за препълване отгоре или отдолу. С обектите от този клас е добре да се работи като с "обикновени" числа от тип `unsigned char`. Например, трябва да можем да събираме и изваждаме два обекта от този клас, да събираме и изваждаме обект и променлива от стандартните числови типове. За да осигурим тези операции, трябва да реализираме шест функции:

```
class SmallInt {  
    friend operator+( SmallInt&, int );  
    friend operator-( SmallInt&, int );  
    friend operator-( int, SmallInt& );  
    friend operator+( int, SmallInt& );  
private:  
    operator+( SmallInt& );  
    operator-( SmallInt& );  
    // ...  
};
```

Операциите са шест, защото всеки стандартен аритметичен тип се преобразува в тип `int` от стандартно преобразуване. Например, изразът

```
SmallInt si( 3 );  
si + 3.14159
```

се пресмята на две стъпки:

1. Константата от тип `double` 3.14159 се преобразува в числото 3.
2. Извиква се `operator+(si, 3)`. Той връща резултат 6.

Ако трябва да създадем побитови и логически операции, операции за отношение и др. операции за клас `SmallInt`, техният брой ще е невъобразимо голям. По-добре да дефинираме една операция, която преобразува обект от този клас в число от тип `int`.

В C++ съществува механизъм, който позволява всеки клас да дефинира преобразувания за своите обекти. Например, за клас `SmallInt` ще дефинираме преобразуване на обектите в тип `int`. Ето дефиницията на това преобразуване:

```
class SmallInt {  
public:  
    // операция за преобразуване: SmallInt —> int  
    operator int() { return value; }  
    // ...  
private:  
    int value;  
};
```

Дефинираните от потребителя преобразувания за даден клас са единствените, които могат да се приложат към обектите на класа. Тяхната дефиниция указва на компилатора какво означава съответното преобразуване.

След дефиниране на `SmallInt::operator int()`, обектите от клас `SmallInt` могат да се използват навсякъде, където се използват числа от тип `int`. Например, изразът

```
SmallInt si( 3 );  
si + 3.14159;
```

се пресмята на две стъпки:

1. Извиква се дефинираната от потребителя операция за преобразуване на обект от клас `SmallInt` в число от тип `int`. Тя преобразува `si` в числото 3.
2. Цялото число 3 се превръща в числото 3.0 и се добавя към константата от тип `double` 3.14159. Получава се 6.14159.

Следващата програма илюстрира използването на клас `SmallInt`:

```
#include <stream.h>  
#include "SmallInt.h"  
  
SmallInt si1, si2;  
  
main() {  
    cout << "Въведете SmallInt: ";  
    while ( cin >> si1 )  
    {  
        cout << "\nПрочетена е стойността " << si1 << "\nТя е ";
```

```

// SmallInt::operator int() се извиква два пъти
cout << (( si1 > 127 )
? "по-голяма от "
: (( si1 <= 127 )
? "по-малка от "
: "равна на ")) << "127\n";

cout << "\nВъведете SmallInt \
(ctrl-d за край): ";
}
cout << "Край!";
}

```

След компилация и изпълнение получаваме следния резултат:

```

Въведете SmallInt: 127

Прочетена е стойността 127
Тя е равна на 127

Въведете SmallInt (ctrl-d за край): 126

Прочетена е стойността 126
Тя е по-малка от 127

Въведете SmallInt (ctrl-d за край): 128

Прочетена е стойността 128
Тя е по-голяма от 127

Въведете SmallInt (ctrl-d за край): 256
***числото 256 е извън границите***

```

Ето как изглежда пълното описание на клас SmallInt:

```

#include <stream.h>

class SmallInt {
    friend istream& operator>>(istream& is, SmallInt& s);
    friend ostream& operator<<(ostream& os, SmallInt& s)
    { return( os << s.value ); }
public:
    SmallInt(int = 0) : value( rangeCheck(i) ) { }
    int operator=(int i) { return( value = rangeCheck(i) ); }
    operator int() { return value; }
private:
    enum { ERRANGE = -1 };
    int rangeCheck( int );
    int error( int );
    int value;
};

```

Извън тялото на класа са дефинирани следните функции:

```

istream& operator>>(istream& is, SmallInt& s)
{

```

```

int i;

is >> i;
s = i;    // SmallInt::operator=(int)
return is;
}

SmallInt::error( int i )
{
    cerr << "\n***числото " << i << " е извън границите***\n";
    return ERRANGE;
}

SmallInt::rangeCheck( int i )
{
    // ако след първите 8 бита, има бит със стойност 1, числото е извън границите
    if ( i & ~0377 ) exit( error(i) );    // stdlib.h
    return i;
}

```

Упражнение 6-27. Защо за клас SmallInt операцията ">>" не е дефинирана така:

```

istream& operator>>( istream& is, SmallInt& si )
{
    return ( is >> si.value );
}

```

Конструктори като операции за преобразуване

Конструкторите с един аргумент дефинират неявно преобразуване за типа на своя аргумент. Например, конструкторът на клас SmallInt, SmallInt(int) дефинира неявно преобразуване. Той преобразува цели числа в обекти от клас SmallInt. Ако е необходимо, преди обръщение към конструктора, може да се извърши и стандартно преобразуване.

```

extern f( SmallInt );
int i;
// i трябва да се преобразува в обект от клас SmallInt
// SmallInt(int) извършва това преобразуване
f( i );

```

При обръщението f(i), конструкторът SmallInt(int) преобразува i в обект от клас SmallInt. Компиляторът създава временен обект от този клас и го предава като аргумент на f().

```

// създава се временен обект от клас SmallInt
{
    SmallInt temp = SmallInt(i);
    f(temp);
}

```

Фигурните скоби показват периода на активност на временно генерирания обект.

Преди да се извика конструкторът SmallInt(int), може да се извърши стандартно преобразуване. Например,

```
double d;  
f( d );
```

се преобразува в

```
{  
    // предупреждение: отсичат се значещи цифри  
    SmallInt temp = SmallInt( (int)d );  
    f(temp);  
}
```

Операция за преобразуване се използва, ако не е възможно друго преобразуване. Нека съществува втора дефиниция на `f()` с аргумент от тип `double`. Тогава `SmallInt(int)` няма да се извика.

```
f( SmallInt );  
f( double );  
int ival;  
  
// съответствие с f(double) чрез стандартно преобразуване  
f( ival );
```

Аргумент на конструктора може да бъде обект от друг клас:

```
class Token {  
public:  
    // ... общодостъпни членове  
private:  
    SmallInt tok;  
    // ... останалата част от дефиницията  
};  
  
// преобразува обект от клас Token в обект от клас SmallInt  
SmallInt::SmallInt( Token& t ) { /* ... */ }  
extern f( SmallInt& );  
Token curTok;  
  
main() {  
    // обръщение към SmallInt( Token& )  
    f( curTok );  
}
```

Операции за преобразуване

Всяка операция за преобразуване е член на клас. Тя преобразува неявно обект от класа в някакъв друг тип. Например, за клас `SmallInt` може да се дефинира операция, която преобразува обект от класа в число от тип `unsigned int`:

```
SmallInt::operator unsigned int()  
{  
    return( (unsigned)value );  
}
```


Ще дефинираме няколко операции за преобразуване за клас Token:

```
#include "SmallInt.h"

class Token {
public:
    Token( char *nm, int vl ) : val( vl ), name( nm ) { }
    operator SmallInt() { return val; }
    operator char*() { return name; }
    operator int() { return val; }
    // ... другите общодостъпни членове
private:
    SmallInt val;
    char *name;
};
```

Забележете, че Token притежава две еквивалентни операции — operator SmallInt() и operator int(). Във втората val се преобразува неявно чрез операцията SmallInt::operator int(). Да разгледаме един пример:

```
#include "Token.h"

void f( int i ) { cout << "\nf(int) : " << i; }
Token t1( "integer constant", 127 );

main() {
    Token t2( "friend", 255 );

    f( t1 ); // t1.operator int()
    f( t2 ); // t2.operator int()
}
```

След компилация и изпълнение получаваме:

```
f(int) : 127
f(int) : 255
```

Операциите за преобразуване имат следния общ вид:

```
operator <type> ();
```

<type> се замества със стандартен, производен или потребителски тип. (Преобразувания в масиви и функции не се разрешават.) Операцията за преобразуване трябва да бъде член на клас. Тя не бива да посочва тип на резултата и нейния списък от аргументи трябва да бъде празен. Следващите декларации на операции за преобразуване са некоректни:

```
operator int( SmallInt& );    // грешка: не е член на клас

class SmallInt {
public:
    int operator int();        // грешка: посочва тип на резултата
    operator int( int );      // грешка: има аргумент
```

```
};
```

Разрешава се явно обръщение към операция за преобразуване, като се използва едно от двете възможни означения:

```
#include "Token.h"
Token tok( "function", 78 );

// явно обръщение към operator SmallInt()
SmallInt tokVal = SmallInt( tok );

// явно обръщение към operator char*()
char *tokString = (char*)tok;
```

Преобразуванията, дефинирани от потребителя, се прилагат неявно от компилатора, когато това е необходимо. Например:

```
#include "Token.h"
extern void f( char * );
Token tok( "enumeration", 8 )
enum { ENUM };          // използва се като token стойност

main() {
    f( tok );            // tok.operator char*();

    // явно обръщение към operator int()
    switch ( int(tok) ) { // tok.operator int()
        case ENUM:
            { // tok.operator SmallInt();
                SmallInt si = tok;
                // ...
            }
    }
}
```

Нека необходимият тип не може да се получи чрез никое преобразуване, дефинирано от потребителя. Все пак ще се изпълни ли някое от тези преобразувания?

- Да, ако за получаване на типа може да се приложи стандартно преобразуване. Например:

```
extern void f( double );
Token tok( "constant", 44 );

// извиква се tok.operator int()
// int —> double чрез стандартно преобразуване
f( tok );
```

- Не, ако към резултата от първото преобразуване трябва да се приложи второ преобразуване, дефинирано от потребителя. По тази причина клас Token притежава както operator SmallInt(), така и operator int(). Ако Token не притежава operator int(), следващото обръщение към f() е некоректно:

```
extern void f(int);
Token tok( "pointer", 37 );

// ако Token::operator int() не съществува, следващото
// обръщение ще предизвика грешка при компилация
f( tok );
```

Липсата на `Token::operator int()` ще наложи изпълнение на две преобразувания, дефинирани от потребителя:

```
Token::operator SmallInt();
```

ще преобразува `tok` в обект от клас `SmallInt`.

```
SmallInt::operator int();
```

ще довърши преобразуването до тип `int`.

Тъй като съществува правило да се изпълнява само едно преобразуване, дефинирано от потребителя, обръщението `f(tok)` ще предизвика грешка при компилация.

Противоречивост

При неявно обръщение към операциите за преобразуване, може да възникне противоречие. Например:

```
extern void f( int );
extern void f( SmallInt );
Token tok( "string constant", 3 );

f( tok );    // грешка: противоречие
```

Обектите от клас `Token` могат да се преобразуват в обект от клас `SmallInt` или в число от тип `int`. Двете преобразувания са еднакво възможни. Обръщението `f(tok)` е противоречиво, защото компилаторът не притежава критерий за избор на едно от двете преобразувания. В резултат ще се получи грешка при компилация. Програмистът може да разреши противоречието чрез явно преобразуване:

```
// явното преобразуване разрешава противоречието
f( int(tok) );
```

Ако операцията `Token::operator int()` не е дефинирана, обръщението `f(tok)` няма да породи противоречие. Тогава ще се изпълни операцията `Token::operator SmallInt()`. Фактът, че обект от клас `SmallInt` може да се преобразува в число от тип `int`, не променя ситуацията. Това означава, че се разглеждат само преобразуванията от първо ниво.

Противоречие няма, когато могат да се изпълнят две от дефинираните преобразувания, но едното изисква и стандартно преобразуване. Ще се изпълни преобразуването, което не изиска стандартно преобразуване. Например:

```

class Token {
public:
    operator float();
    operator int();
};

Token tok;
float ff = tok;      // ок: operator float()

```

Ако двете преобразувания имат еднакъв приоритет, ще се получи грешка. Например:

```

// грешка: двете операции са еднакво приложими
long lval = tok;

```

Противоречие може да се появи, когато два класа се преобразуват взаимно. Например:

```

SmallInt::SmallInt( Token& );
Token::operator SmallInt();

extern void f( SmallInt );
Token tok( "pointer to class member", 197 );

// грешка: две преобразувания са еднакво приложими
f( tok );

```

В този пример съществуват два начина за преобразуване на tok в обект от клас SmallInt. Обръщението f(tok) е противоречиво, защото компилаторът не може да избере един от тези два начина. Противоречието може да се отстрани чрез явно преобразуване. Например:

```

// явното преобразуване премахва противоречието
// извиква се Token::operator SmallInt()
f( (SmallInt)tok );

```

Друг начин за разрешаване на противоречието е явното обръщение към Token::operator SmallInt():

```

// явното преобразуване премахва противоречието
f( tok.operator SmallInt() );

```

Забележете, че следващото явно преобразуване не отстранява противоречието:

```

// грешка: противоречието се запазва
f( SmallInt(tok) );

```

SmallInt(Token&) и Token::operator SmallInt() са възможни интерпретации на това явно преобразуване.

Когато разглеждаме появата на противоречия при преобразуване на типове, трябва да се спрем на въпроса за читаемост на програмата. Преобразуването на обект от клас SmallInt в число от тип int има ясен смисъл. Аналогична е ситуацията, когато обект от клас String се

преобразува в тип `char*`. В двата случая съществува еднозначно съответствие между стандартния тип данни и вътрешното представяне на класа. Ако липсва логическо съответствие между тях, използването на обекти от класа може да породи противоречия в смисъла на програмата. Ето един пример:

```
class Date {
public:
    // кой член се връща?
    operator int();
private:
    int month, day, year;
};
```

Каква стойност връща операцията, която преобразува обект от клас `Date` в число от тип `int`? Какъвто и избор да направим, използването на обектите от клас `Date` ще бъде противоречиво за този, който разглежда програмата, защото не съществува еднозначно логическо съответствие. В случая е по-добре да не се дефинира операция за преобразуване.

Упражнение 6-28. Какви операции за преобразуване са необходими, за да може обектите от клас `String` да се използват по този начин:

```
extern int strlen( const char* );
const maxlen = 32;

String st = "a string";
int len = strlen( st );
if ( st >= maxlen ) ...
String st2 = maxlen;
```

Упражнение 6-29. Какви са предимствата и недостатъците от дефиниране на преобразуване, което позволява да се използва оператора

```
if ( st >= maxlen ) ...
```

Упражнение 6-30. Интересно преобразуване за клас `BinTree` е формирането на обект от клас `IntArray` от стойността на единствения член на `BinTree`. (Дефиницията на клас `IntArray` е дадена в раздел 1.8.) Дефинирайте преобразуванията:

```
BinTree::BinTree( IntArray& );
BinTree::operator IntArray();
```

Забележете, че редът на обхождане на дървото трябва да бъде еднакъв за двете операции. Може да се използва следния общ вид на обхождане в ширина:

```
BinTree::preOrder( /* ??? */ )
{
    if ( !node ) return;
    node->visit( /* ??? */ ); // някаква обработка

    if( left ) left->preOrder( /* ??? */ );
    if( right ) right->preOrder( /* ??? */ );
}
```

Упражнение 6-31. Предишното упражнение позволява съвместно използване на обекти от класовете `IntArray` и `BinTree`. Например:

```
extern BinTree bt;  
extern IntArray ia;  
  
if ( ia == bt ) ...
```

Сравнението се осъществява на две стъпки:

1. Извиква се операцията за преобразуване на обект от клас `BinTree` в обект от клас `IntArray`.
2. Извиква се операцията за проверка на равенство между два обекта от клас `IntArray`.

Какво ще стане, ако клас `IntArray` притежава следните две преобразувания:

```
IntArray::IntArray( BinTree& );  
IntArray::operator BinTree();
```

ГЛАВА 7

Обектно-ориентираното програмиране позволява на абстрактните типове данни да встъпват в отношения помежду си чрез механизма на наследяване. Този механизъм позволява един клас да наследи избрани членове от други класове. Наследяването се осъществява чрез използване на производни класове (derivated classes).

В тази глава ще разгледаме следните въпроси: Защо взаимоотношенията между класовете са важни? Какви проблеми може да предизвика наследяването? Какъв е смисълът на термините обектно-ориентирано програмиране и наследяване? Нека разгледаме отново клас `Screen`, който дефинирахме в глава 5. Дефиницията на този клас отговаря на остаряла представа за описание на екран. Съвременните работни станции и терминали поддържат т.нар. "прозорци" на екрана. Всеки прозорец представлява отделен екран с допълнителни възможности:

- Прозорецът може да се разширява или свива.
- Прозорецът може да се премества от едно място на друго.
- На един и същи екран може да има повече от един прозорец.

Всеки прозорец има някакво предназначение. Един от тях може да се използва като редакторски екран за въвеждане текста на тази книга. Във втори прозорец може да се създават и редактират примерите в книгата, а в трети тези примери да се компилират и изпълняват. Четвърти прозорец може да служи за четене на електронната поща. Всички прозорци могат да се използват едновременно по начина, по който се използва обикновеният екран.

За всеки прозорец трябва да се знаят размерите и позицията му на екрана. Това означава, че освен данните на клас `Screen`, новият клас `Window` трябва да има и допълнителни данни — екранни координати. Част от методите на клас `Screen` може да се използват директно от клас `Window`, а други да се реализират отново.

Клас `Window` разширява възможностите на клас `Screen`. Той може да се разглежда като разширен подтип на независимия абстрактен тип данни `Screen`. Двата класа имат общи данни и функции, затова един от членовете на `Window` може да бъде обект от клас `Screen`:

```
class Window {
public:
    // ...
private:
    Screen base;
    // ... допълнителни данни
};
```

Тази дефиниция на `Window` създава един проблем. Тъй като данните на клас `Screen` са негови скрити (private) членове, те не са достъпни директно за обектите от клас `Window`. Ако има готова програма, която използва съществуващата дефиниция на `Screen`, по-добре `Window` да се дефинира като приятелски клас.

За да се използват директно членовете на `Screen`, `base` трябва да бъде общодостъпен член на `Window`:

```
Window w;
w.base.clear();
```

Това обстоятелство ще наруши скриването на информация. Друго затруднение ще предизвика фактът, че трябва да се знае кои методи на Screen са дефинирани отново за обектите от клас Window и кои се използват наготово. От това зависи синтаксисът на обръщение към функциите. Например, ако home() се дефинира отново, следващите две обръщения извикват различни функции:

```
w.home();  
w.base.home();
```

Различният синтаксис може да стане причина за програмни грешки.

За да се запази скриването на информация и единният синтаксис, Window може да дефинира по една общодостъпна функция за всяка активна функция от Screen. За да използваме

```
w.clear();
```

трябва да дефинираме

```
inline void Window::clear() { base.clear(); }
```

Този начин на програмиране е твърде скучен, но приложим.

Ако обогатим Window с нови възможности, сложността на подобна реализация поставя сериозно ограничение пред този стил на програмиране. Ако например дефинираме специален вид прозорец, клас Menu, той ще притежава свои данни и функции. От друга страна той ще ползва и част от методите на Window, а чрез тях и част от функциите на Screen. По-добро решение предоставя механизмът на наследяване.

7.1 Обектно-ориентирано програмиране

Наследяването позволява на един клас да използва членовете на друг клас като свои членове. Ако Window наследи членовете на Screen, всички възможни действия с обектите на Screen могат да се изпълнят и с обектите на Window. Остава да се дефинират функциите, които реализират допълнителните възможности на Window и функциите, които заместват съществуващи в Screen функции.

Наследяването е основна характеристика на обектно-ориентираното програмиране. В C++ то се осъществява чрез производни класове. Ако Window е производен клас на Screen, той наследява членовете на Screen. Window получава достъп до общодостъпните членове на Screen и може да ги използва като свои членове. Например:

```
Window w;  
w.clear().home();
```

За да дефинираме Window като производен клас, трябва да разширим дефиницията на класове с още два детайла:

1. Заглавната част на дефиницията на производния клас се разширява със списък на класовете, от които се наследяват членове:

```
class Window : public Screen { ... }
```

Двоеточието показва, че Window е производен клас на един или няколко предварително дефинирани класове. Клас Screen се нарича основен или базов клас (base class). Видимостта на наследените членове в производния клас Window зависи от ключовата дума пред името на основния клас — public или private.

2. В тялото на основния клас може да се въведе още едно ниво на достъп — protected. Членовете от секция protected са достъпни за членовете на производния клас, а в останалата част на програмата не са достъпни. Ако Window използва част от скритите членове на Screen, тези членове трябва да се преместят в секция protected. Тогава те ще могат да се използват от членовете и "приятелите" на Window.

Всеки производен клас може да бъде основен за друг клас. Например, Menu може да бъде производен клас на Window:

```
class Menu : public Window { ... };
```

По този начин Menu ще наследи членове от Window и Screen и трябва да дефинира само специфичните функции и тези, които заместват функции от предните два класа. Всички функции се извикват по един и същ начин:

```
Menu m;  
m.display();
```

Между основен клас и производни класове съществуват множество стандартни преобразувания. Например, производен клас може да се присвои на основен клас, ако основният клас е public.

```
Menu m;  
Window &w = m;    // ok  
Screen *ps = &m;   // ok
```

Тези специални отношения между основен клас и производни класове насърчават т.нар. "родов" стил на програмиране. При него действителният тип клас е неизвестен. Следващата декларация на функцията dumpImage() потвърждава това:

```
extern void dumpImage( Screen *s );
```

dumpImage() може да се извика с адреса на обект от всеки клас, за който Screen е основен клас от тип public. При това няма значение колко далеч е този клас в наследствената верига на Screen. Ако се използва клас, който не наследява Screen, ще се получи грешка при компилация. Например:

```
Screen s;
```

```

Window w;
Menu m;
BitVector bv;

// ok: Window е вид екран
dumpImage( &w );

// ok: Menu е вид екран
dumpImage( &m );

// ok: типът на аргумента съвпада
dumpImage( &s );

// грешка: BitVector не произлиза от Screen
dumpImage( &bv );

```

Всеки от класовете Screen, Window и Menu притежава своя функция dumpImage(). В зависимост от типа на действителния аргумент горната функция dumpImage() извиква една от трите функции със същото име. Ако не се използва обектно-ориентиран стил на програмиране, създаването на общата функция dumpImage() изисква усилие. Ето една примерна дефиниция:

```

#include "Screen.h"
extern void error( const char*, ... );

enum { SCREEN, WINDOW, MENU };
// не се използва обектно-ориентирано програмиране
void dumpImage( Screen *s )
{
    // определя типа на действителния аргумент
    switch( s—>typeOf() )
    {
        case SCREEN:
            s—>dumpImage();
            break;
        case WINDOW:
            ((Window *)s)—>dumpImage();
            break;
        case MENU:
            ((Menu *)s)—>dumpImage();
            break;
        default:
            error( "unknown type", s—>typeOf() );
            break;
    }
}

```

В този случай определянето на действителния тип на аргумента се оставя изцяло на програмиста. При обектно-ориентирания стил на програмиране, програмистът се освобождава от това задължение. Типа на действителния аргумент се определя от компилатора. Извършва се т.нар. динамично свързване (dynamic binding). Динамичното свързване опростява значително дефиницията на dumpImage():

```

void dumpImage( Screen *s )
{
    s—>dumpImage();
}

```

При извикване на тази функция се извършва автоматично обръщение към нужната функция със същото име. Това съкращава сложността и размера на програмата и улеснява нейното разширяване. По-нататък може да се дефинира още едно звено във веригата на наследяване, клас `Border`, без да се променя съществуващият код. Програмата трябва да се компилира отново. При извикване на `dumpImage()` с аргумент от клас `Border`, ще задейства нужната функция. Динамичното свързване в C++ се осъществява от виртуални функции (virtual member functions). Една функция е виртуална, ако нейната декларация в тялото на класа започва с ключовата дума `virtual`:

```
class Screen {
public:
    virtual void dumpImage();
    // ...
};
```

Една функция се дефинира като виртуална по две причини:

1. Ако се очаква класът да бъде базов клас.
2. Ако дефиницията на функцията зависи от тип на аргумент.

В нашия пример дефиницията на трите функции `dumpImage()` — за класовете `Screen`, `Window` и `Menu` — зависи от типа на аргумента. Тази зависимост налага дефинирането на `Screen::dumpImage()` като виртуална функция.

Отношенията между основен и произведен клас са известни като наследствена йерархия (inheritance hierarchy). Наследяването на свойства от няколко основни класа се нарича наследствен граф (inheritance graph).

Обектно-ориентираното програмиране се основава на разширена концепция за абстрактен тип данни, която поддържа идеята за наследствена йерархия. В този смисъл `Simula`

беше първият обектно-ориентиран език, но терминът обектно-ориентирано програмиране се появи за първи път в езика `Smalltalk`.

7.2 Клас `ISortArray`

Масив, който се използва за справочен индекс, трябва да бъде сортиран. Най-лошият случай при търсене на стойност в неподреден масив се получава, когато трябва да се претърси целият масив. В средния случай трябва да се разгледат половината от елементите на масива. Когато броят на елементите е голям, времето за търсене на елемент е прекалено голямо. По-добро време може да се получи, ако се използва алгоритъм на двоично търсене, но масивът трябва да бъде подреден. Средното и най-лошото време за откриване на елемент тогава е сравнимо с $\log n$, където n е дължината на масива.

Наследственият механизъм освобождава програмиста от необходимостта за цялостна реализация на клас сортиран масив. `ISortArray` може да се дефинира като произведен клас на `IntArray`. Тогава е достатъчно да се дефинират само различните функции за двата класа. Еднаквите данни и функции могат да се използват без проблеми от обектите на `ISortArray`.

Динамичното свързване не изисква допълнителни корекции в кода на програмата. Функциите, които се използват различно за двата класа, се декларират като виртуални. Така сортирани масиви могат да се използват в съществуващи приложения, без промени в кода на програмите. Забележете, че програмите трябва да се компилират отново.

Знаем, че една функция е виртуална, ако нейната декларация в тялото на класа започва с ключовата дума `virtual`. Данните и функциите, които не са виртуални, могат да се използват без проблеми от сортиран масив. Изключение правят конструкторите и операцията за присвояване чрез копиране. Новата дефиниция на клас `IntArray` изглежда така:

```
const int ArraySize = 24;    // дължина по подразбиране
class IntArray {
public:
    IntArray( int sz = ArraySize );
    IntArray( const IntArray& );
    virtual ~IntArray() { delete ia; }
    IntArray& operator=( const IntArray& );
    int getSize() { return size; }
    void grow();
    virtual int& operator[] ( int );
    virtual void sort( int, int );
    virtual int find( int );
    virtual int min();
    virtual int max();
protected:    // видими само за членовете и производните класове на IntArray
    int size;
    int *ia;
};
```

Функцията `sort()` е член на `IntArray`, защото тя е полезна операция за всеки масив. Разликата между `IntArray` и `ISortArray` е в това, че вторият клас представя само сортирани масиви — главна предпоставка за осъществяване на справки. Справките, а не сортирането на масива са причина за използване на производен клас.

`IntArray::sort()` е функцията `qsort()` от раздел 3.9. Незначителните разлики се дължат на това, че сега тази функция е член на клас `IntArray`. Опитайте да модифицирате `qsort()` като `IntArray::sort()`.

Всяка виртуална функция може да се дефинира отново в производен клас, но това не е задължително. Една функция се декларира като виртуална, ако съществува вероятност в някой производен клас да се дефинира функция със същото име. Всеки производен клас може да използва виртуална функция на базовия клас. Производният клас наследява всяка виртуална функция на базовия клас, за която няма собствена дефиниция. Например, клас `ISortArray` наследява виртуалния деструктор на клас `IntArray`. По-подробно виртуалните деструктори са разгледани в раздел 8.2. `ISortArray` наследява и сортиращата функция на клас `IntArray`. Следва дефиницията на `ISortArray`:

```
class ISortArray : public IntArray {
public:
    ISortArray( int sz = ArraySize )
        : IntArray(sz) { dirtyBit = 0; }
    int& operator[] ( int );
    int find( int );
    int min() { checkBit(); return ia[0]; }
    int max() { checkBit(); return ia[size-1]; }
private:
    void checkBit();
    char dirtyBit;    // ако е 1, необходимо е сортиране
};
```

Конструкторът на основния клас се извиква преди конструктора на производния клас. Списъкът за инициализация на членове може да предаде аргументи на конструктора на основния клас. ISortArray няма членове, които са указатели. Освен това неговата семантика позволява копиране на dirtyBit. Следователно, служебната операция за присвояване чрез копиране е достатъчна. Когато се присвоява или инициализира един обект от клас ISortArray с друг обект от същия клас, за управление на базовия клас се извиква съответно

```
IntArray( const IntArray& )
```

или

```
operator=( const IntArray& )
```

Ако производният клас дефинира функция, която е обявена за виртуална в основния клас, ключовата дума `virtual` не е необходима. Типът на резултата, името и сигнатурата на тази функция трябва да съвпадат с тези на функцията от основния клас.

Да разгледаме дефиницията на функцията `checkBit()`. Тя проверява дали има промяна в масива. Ако това е така, масивът се сортира отново. `checkBit()` се извиква от всяка справочна функция.

```
#include "Isort.h"
void ISortArray::checkBit()
{ // проверява дали масивът е променен
    if ( dirtyBit ) {
        sort( 0, size-1 );
        dirtyBit = 0;
    }
}
```

Функцията `find()` е преработената функция `binSearch()` от раздел 3.10. Незначителните промени в дефиницията се оставят на читателя.

Остава да дефинираме операцията за индексирание. Чрез нея масивът може да се промени и преди поредната справка да се наложи сортиране. Ето една възможна реализация на операцията за индексирание:

```
int& ISortArray::operator[]( int indx )
{
    dirtyBit = 1;
    return IntArray::operator[]( indx );
}
```

В тази дефиниция използвахме явно обръщение към операцията за индексирание на клас `IntArray`. При явно обръщение към виртуална функция чрез операцията `::`, още при компилация е известно коя функция ще се извика. Същото важи и за обръщение към виртуална функция чрез обект от клас.

Упражнение 7-1. В реализираната операция за индексирание на клас `ISortArray`, `dirtyBit` получава безусловно стойност 1. Защо това не е ефективно? Предложете по-добра реализация на тази операция.

Упражнение 7-2. Дефинирайте отново клас `ISortArray`, като позволите да се запазва броя на справките за обект от класа, преди неговото поредно сортиране.

Упражнение 7-3. Предефинирайте `ISortArray`, за да запазвате броя на сортиранията на всеки обект, сравнен с броя на обръщенията към справочните функции.

Упражнение 7-4. Променете дефиницията на `sort()`, така че функцията да определя дали масивът се нуждае от сортиране. Предложете начин за запазване на процента ненужни обръщения към `sort()`.

Упражнение 7-5. Дефинирайте клас `IStatSort` като производен клас на `ISortArray`, с цел новият клас да поддържа статистическите данни от предишните три упражнения.

7.3 Описание на зоологическа градина

Ще илюстрираме отношенията между основни и производни класове с един пример. Ще опишем животните в една зоологическа градина.

Лин-Лин, голяма китайска панда, ще бъде взета назаем в местната зоологическа градина за шест месеца. През тези месеци се очаква четирикратно увеличение на посетителите. Поради липса на достатъчно персонал, ръководството предлага в информационния център на зоологическата градина да се инсталира терминал, който да отговаря на въпросите на посетителите.

Животните в зоологическа градина съществуват на различни нива на абстракция. Има отделни животни, като Лин-Лин. Всяко животно, обаче, спада към конкретен вид. Например, Лин-Лин е голяма панда. Видовете са членове на някакъв род. Например, гигантската панда спада към рода на Мечките. Всеки род е член на животинското царство. В случая това царство се ограничава с животните от зоологическата градина.

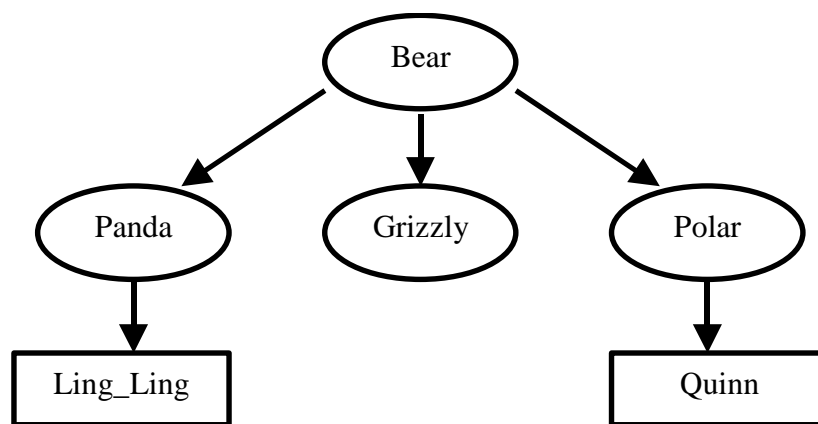
Таксономията на животните е типичен пример за наследствена йерархия. Фигура 7.1 илюстрира връзките в рода Мечки. Наследствената йерархия в този случай се нарича просто наследяване, защото всеки производен клас произлиза непосредствено от един основен клас. Таксономията на животните не осигурява тяхното пълно описание. Например, пандата е мечка, но освен това тя е застрашена от изчезване и следователно е защитен вид. Полярната мечка и мечката гризли също са мечки, но не са застрашени от изчезване. Леопардът, който принадлежи към семейство Котки също е защитен.

"Защитен" е независим абстрактен тип данни, който разваля структурата на таксономията. Простото наследяване не може да опише това свойство на някои животински видове. За тази цел се използва множествено наследяване. Фигура 7.2 илюстрира този тип наследяване. То се изобразява във формата на наследствен граф.

Множественото наследяване дефинира връзки между независими класове. Може да се смята, че в този случай производният клас е съставен от няколко основни класове. Езикът C++ позволява да се дефинира както просто, така и множествено наследяване.

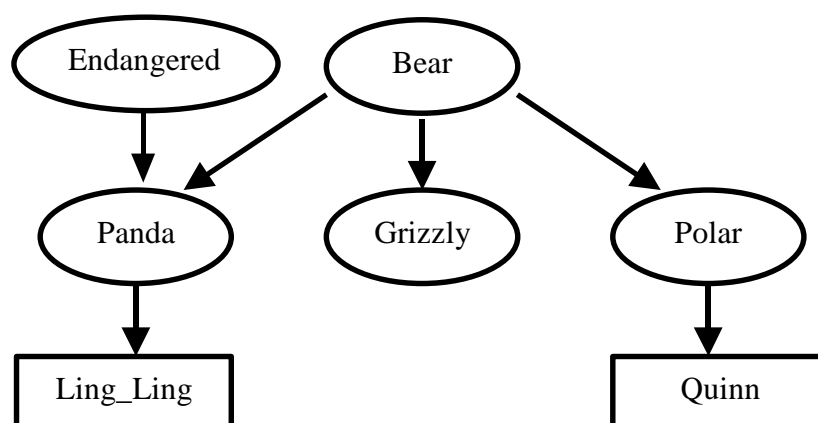
Фигура 7.3 показва три нива от наследствената йерархия на животните в зоологическата градина.

Клас `ZooAnimal` представлява абстрактен тип данни. От него произлизат други класове. Той може да се смята за недовършен клас, чиято дефиниция се довършва от производните му класове.



Фиг. 7.1 Просто наследяване

ZooAnimal дефинира общите свойства за всички животни от зоологическата градина. Неговите производни класове дефинират свойства, които са специфични за конкретен род животни. Производните класове от второ ниво доопределят съответния вид животни. Например, клас Bear дефинира специфичните черти на рода Мечки, а клас Panda описва конкретен вид мечка.



Фиг. 7.2 Множествено наследяване

При изграждане на наследствената йерархия, дефиницията на ZooAnimal многократно ще се променя и разширява. Създаването на всеки клас започва с формиране на абстракцията и завършва с нейната реализация, но пътят между тези две точки не е права линия, а цикличен процес.

Класът, който е корен на йерархията, се нарича абстрактен суперклас (abstract superclass). ZooAnimal е абстрактен суперклас за всички класове животни.

Абстрактният суперклас е основа за изграждане на наследствената йерархия. Независимо колко сложна е тя и какви нива на влагане съдържа, отношенията между класовете се поддържат от наследените членове на абстрактния суперклас.

Класът на защитените животни, Endangered, не характеризира всяко животно от зоологическата градина. Затова ZooAnimal не е негов производен клас. По същата причина обект от клас Endangered не може да стане член на ZooAnimal. Endangered е допълнителен основен клас. Той е независим от клас ZooAnimal. Множественото наследяване позволява да се дефинират класове, които наследяват членове от няколко основни класа. Такъв клас е Panda. В тази и следващата глава ще разгледаме как това представяне може да се опише на езика C++.

Упражнение 7-6. В схеми и чертежи се използват различни геометрични фигури — триъгълници, квадрати, окръжности и др. Начертайте наследствената йерархия на геометричните фигури.

Упражнение 7-7. Къде са местата на квадрата и правоъгълника в тази наследствена йерархия? Къде поставихте различните видове триъгълници (правоъгълен, равнобедрен и др.)?

7.4 Производни класове

Наследственият граф от фигура 7.3 се описва схематично по следния начин:

```
// основни класове
class ZooAnimal { ... };
class Endangered { ... };
class Carnivore { ... };
class Herbivore { ... };

// Bear: просто наследяване
class Bear : public ZooAnimal { ... };

// Cat, Panda : множествено наследяване
class Cat : public ZooAnimal, Carnivore { ... };
class Panda: private Endangered, public: Bear, private Herbivore { ... };
```

Bear, Panda и Cat са производни класове. Клас Bear е пример за просто наследяване, защото произлиза от един основен клас, ZooAnimal. Пред името на основния клас стои ключовата дума public. Cat и Panda са примери за множествено наследяване. Panda произлиза от три основни класа — единият е public (Bear), а другите два private (Endangered и Herbivore). Cat произлиза от два основни класа — единият е public (ZooAnimal), а другият private (Carnivore). Ако атрибутът public/private е изпуснат, базовият клас се смята за private. В следващия пример основният клас Carnivore е private:

```
class Cat : public ZooAnimal, Carnivore { ... };
```

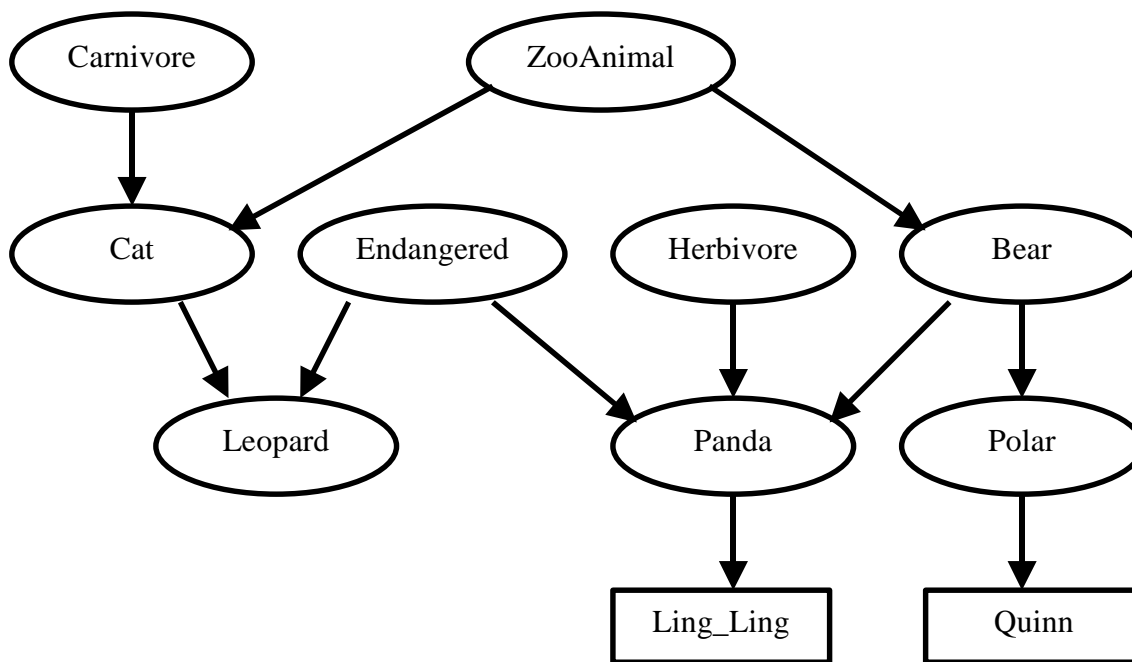
Лин-Лин е конкретен обект от клас Panda. Декларацията на обект от производен клас не се различава от декларацията на другите обекти:

```
Panda Ling_Ling; // декларация на обект от производен клас
```

Производен клас може да бъде основен за друг клас. Например, клас Bear произлиза от клас ZooAnimal, но е основен клас за клас Panda.

В дефиницията на основен клас има две особености:

1. Членовете на основния клас, които не бива да бъдат общодостъпни, но трябва да са достъпни за производните класове, се дефинират в секцията protected. В "обикновените" класове същите членове се намират в секцията private.
2. Тези функции-членове на основен клас, чиято реализация зависи от неизвестни детайли в описанието на производните класове, се декларират като виртуални.



Фигл 7.3 Наследствен граф на животните от зоопарка

По-нататък ще използваме следната проста дефиниция на основния клас `ZooAnimal`:

```

class ZooAnimal {
public:
    ZooAnimal( char*, char*, short );
    virtual ~ZooAnimal();
    virtual void draw();
    void locate();
    void inform();
protected:
    char *name;
    char *infoFile;
    short location;
    short count;
};
  
```

Упражнение 7-8. Членовете на абстрактния суперклас са общи за цялата йерархия. Кои са общите данни и функции за йерархията на геометричните фигури? Определете виртуалните функции.

Упражнение 7-9. Дефинирайте клас `Shape` (геометрична фигура) като използвате създадената от предишното упражнение схема. Забележете, че деструкторът на абстрактния суперклас е винаги виртуален.

Дефиниране на производни класове

В този подраздел ще разгледаме подробно дефинирането на производни класове. Ето една проста предварителна дефиниция на производния клас `Bear`:

```

#include "ZooAnimal.h"
  
```

```

class Bear : public ZooAnimal {
public:
    Bear( char*, char*, short, char, char );
    ~Bear();
    void locate( int );
    int isOnDisplay();
protected:
    char beingFed;
    char isDanger;
    char onDisplay;
    char epoch;
};

```

Дефинициите на "обикновен" и производен клас се различават само по заглавната част:

```

class Bear : public ZooAnimal

```

Двоеточието показва, че присъства списък от основни класове, от които произлиза дадения клас. Броят на основните класове може да бъде произволен, но всеки основен клас трябва да се среща веднъж в списъка. Основните класове се дефинират предварително.

Пред името на всеки основен клас стои една от думите `public`, `private`. Когато, никой от тези две думи не стои пред името на класа, класът се смята за `private`. Например, в следващите две декларации `ZooAnimal` е основен клас с атрибут `private`:

```

// двете декларации са еквивалентни
class Bear : ZooAnimal
class Bear : private ZooAnimal

```

Въпреки еквивалентността на двете декларации, явната декларация е за предпочитане, защото предпазва от грешки, особено когато има няколко основни класа. (Повечето компилатори извеждат съобщение за грешка, когато пред името на основен клас не стои никой от двете запазени думи.)

При множествено наследяване всеки основен клас трябва да притежава свой атрибут `public/private`. Този атрибут не се предава на следващия основен клас в списъка на основните класове. В следващия пример съществува голяма вероятност за грешно интерпретиране на `Endangered` като основен клас с атрибут `public`:

```

// по подразбиране Endangered е private
class Panda: public Bear, Endangered

```

Всеки основен клас може да бъде `private` за един производен клас и `public` за друг. Например, в декларацията на `Bear` и `Cat` `ZooAnimal` е основен клас с атрибут `public`, но в декларацията на клас `Rodent` `ZooAnimal` може да бъде основен клас с атрибут `private`.

Смисълът на атрибута `public/private` и причините за избор на една от двете възможности са разгледани в раздел 7.6.

Упражнение 7-10. Дефинирайте клас, който описва геометричната фигура окръжност — `Circle`.

Упражнение 7-11. Дефинирайте клас, който описва геометричната фигура квадрат — Box.

Упражнение 7-12. Дефинирайте клас Rectangle (правоъгълник), като производен клас на клас Box.

Достъп до наследени членове

Клас Bear наследява от ZooAnimal членовете: name, location, count, infoFile, locate() и inform(). (В глава 8 ще обсъдим действието на виртуалните функции draw() и ~ZooAnimal().) Достъпът до наследените членове не се отличава от достъпа до собствените членове. Например:

```
void objectExample( Bear& ursus ) {  
    if ( ursus.isOnDisplay() ) {  
        ursus.locate(HIGH);    // HIGH е константа  
        if ( ursus.beingFed )  
            cout << ursus.name << " в момента яде\n";  
        ursus.inform();  
    }  
}
```

Достъп до наследените членове може да се получи и чрез операцията за принадлежност към клас. Например:

```
if ( ursus.beingFed )  
    cout << ursus.ZooAnimal::name;  
ursus.ZooAnimal::inform();
```

В повечето случаи използването на тази операция е излишно. Компиляторът може да открие наследения член без допълнителна информация. Съществуват, обаче, два случая, когато допълнителната информация е необходима:

1. Когато името на наследен член съвпада с името на член на производния клас.
2. Когато два или повече основни класа притежават членове с едно и също име, достъпни в производния клас.

Ако името на наследен член съвпада с името от член на производния клас, наследеният член се покрива. Същата ситуация възниква при дефиниране на локална променлива, ако вече съществува глобална променлива със същото име. В двата случая се избира идентификаторът, който е дефиниран в непосредствената област на действие. Например, Bear::locate(int) покрива наследения член от клас ZooAnimal със същото име. Операторът

```
ursus.locate();
```

се възприема като обръщение към Bear::locate(int) и предизвиква грешка при компилация, защото липсва аргумента на функцията. Обръщението към покритата функция със същото име изглежда така:

```
ursus.ZooAnimal::locate();
```

Ако два или повече основни класа притежават членове с едно и също име, неадресираното използване на това име от техен производен клас предизвиква грешка при компилация. Възникналото противоречие може да се разреши чрез операцията "::". Например:

```
class Endangered      { public: highlight(short); };
class Herbivore       { public: highlight(short); };
class Panda : public Bear, public Endangered, public Herbivore { public locate(); };

// покрива ZooAnimal::locate() и Bear::locate(int)
Panda::locate() {
    if ( isOnDisplay() ) {
        Bear::locate(LOW); // LOW е константа
        highlight( location ); // грешка: противоречие
        Endangered::highlight( location ); // ок
    }
}
```

Разрешаване на противоречието чрез явно посочване на класа, от чийто член се интересуваме, притежава два недостатъка:

1. Ако така се извика виртуална функция, тя няма да се отличава от останалите функции. Следователно, ако функцията `highlight()` е виртуална, тя няма да се използва като такава.
2. Противоречивостта се наследява и от следващите класове в наследствената верига. При създаване на наследствена верига трябва да се спазва едно правило: всеки производен клас трябва да познава реализацията само на класовете, от които непосредствено произлиза. Предложеният начин за разрешаване на противоречията нарушава това правило.

Ако наследените членове с еднакво име са функции, добро решение на проблема е да се дефинира функция със същото име, която е член на производния клас. Например:

```
void Panda::highlight()
{
    // капсулира наследените членове, които пораждат противоречие
    Endangered::highlight( location );
    Herbivore::highlight( location );
}
```

`Panda::highlight()` покрива наследените от `Endangered` и `Herbivore` членове със същото име, като същевременно обединява тяхното действие. Противоречието се разрешава по лесно разбираем за потребителя начин.

Упражнение 7-13. В много случаи класовете се нуждаят от функция `debug()`, която отпечатва текущите стойности на техните данни. Дефинирайте такава функция за всеки клас от йерархията на геометричните фигури.

Инициализиране на основни класове

Конструкторът на основния клас получава параметри чрез списъка за инициализация на членове. Забележете, че списъкът за инициализация на членове може да стои само в дефиницията, но не и в декларацията на съответния конструктор. Ето как изглежда списъкът за инициализация на членове в дефиницията на конструктора на клас `Bear`:

```

Bear::Bear( char *nm, char *fil, short loc, char danger, char age )
    : ZooAnimal( nm, fil, loc), epoch( age ), isDanger( danger )
    {}          // тялото нарочно е празно

```

В случай на множествено наследяване, в списъка за инициализация на членове може да присъства името на всеки основен клас. Например:

```

// PANDA, MIOCENE, CHINA, BAMBOO са константи
Panda::Panda( char *nm, short loc, char sex )
    : Bear( "Ailuropoda melaoleuca", "Panda", PANDA, MIOCENE ),
      Endangered( CHINA ), Herbivore( BAMBOO ),
      name( nm ), cell( loc ), gender( sex )
    {}          // тялото нарочно е празно

```

Основен клас без конструктор или с конструктор, който няма аргументи, не е необходимо да се включва в списъка за инициализация на членове. Основен клас може да се инициализира чрез списък от аргументи за съответния конструктор (горните два примера) или направо чрез обект. Този обект трябва да е от основния клас. Например:

```

Bear::Bear( ZooAnimal& z )
    : ZooAnimal( z )      { /* ... */ }

```

Ако основният клас на даден производен клас е public, обектът може да бъде и от производния клас. Например:

```

Bear::Bear( const Bear& b )
    : ZooAnimal( b )      { /* ... */ }

```

Следващият пример показва, че обектът може да няма никакво отношение към основния клас. Ако съществува подходяща операция за преобразуване, грешка няма да има:

```

class GiantSloth : public Extinct {
public:
    operator Bear() { return b; }
protected:
    Bear b;
};

// използва GiantSloth::operator Bear()
Panda::Panda(GiantSloth& gs) : Bear(gs) {...}

// използва GiantSloth::operator Bear()
Bear::Bear(GiantSloth& gs) : ZooAnimal(gs) {...}

```

Конструкторите на основните класове се извикват преди конструктора на производния клас. Ето защо списъкът за инициализация на членове на производния клас не трябва да съдържа явна инициализация на наследен член; в този момент наследените членове са вече инициализирани. Ето един пример за неправилна инициализация на наследен член:

```
// не се позволява явно инициализиране на наследения член ZooAnimal::name
Bear( char *nm ) : name( nm ) { /* ... */ }
```

Упражнение 7-14. Дефинирайте конструктор(и) за клас Shapes.

Упражнение 7-15. Дефинирайте конструктор(и) за клас Circle.

Упражнение 7-16. Дефинирайте конструктори за класовете Box и Rectangle.

7.5 Производни класове и скриване на информация

Появата на производни класове налага въвеждане на допълнителна секция в дефиницията на основните класове. Секциите `private` и `public` не са достатъчни, за да отразят по-сложното взаимодействие между класовете. В новата секция се дефинират членове, които са достъпни за производните класове и техните "приятели", но в останалата част от програмата са скрити. В същото време производните класове нямат никакъв достъп до скритите членове на основния клас. Да разгледаме едно опростено описание на таксономията при животните:

```
class ZooAnimal {
    friend void setPrice( ZooAnimal& );
public:
    isOnDisplay();
protected:
    char *name;
    char onDisplay;
private:
    char forSale;
};

class Primate : public ZooAnimal {
    friend void canCommunicate( Primate* );
public:
    locate();
protected:
    short zooArea;
private:
    char languageSkills;
};
```

Производният клас наследява всички членове на основните класове, но няма достъп до техните скрити членове. Например, `Primate` няма достъп до `ZooAnimal::forSale`, докато приятелската функция `setPrice()` има достъп до всички членове на `ZooAnimal`. Но какво е нейното отношение към членовете на производния клас `Primate`? Да разгледаме един пример:

```
void setPrice( ZooAnimal &z ) {
    Primate *pr;
    if ( pr—>forSale                // позволен достъп
        && pr—>languageSkills )    // непозволен достъп
        // ...
}
```

`Primate` е частен случай на `ZooAnimal` и наследява неговите членове. `setPrice()` има достъп до тези членове, защото те са членове на `ZooAnimal`. Следователно, членът `ZooAnimal::forSale` е достъпен в горната дефиниция на `setPrice()`.

Приятелските функции и членовете на ZooAnimal имат достъп само до тези членове на Primate, които са наследени от клас ZooAnimal. Следователно, членът Primate::languageSkills не е достъпен в тялото на setPrice().

Ако ZooAnimal е приятелски клас за Primate, всички членове на ZooAnimal ще имат достъп до скритите членове на Primate. Но и в този случай ограниченият достъп на функцията setPrice() до членовете на Primate се запазва. Единственият начин за разширяване на този достъп е явното деклариране на функцията като приятелска функция за клас Primate. Ето един пример, когато членовете на основния клас и неговите "приятели" имат различно поведение спрямо производния клас.

Функцията canCommunicate() е приятелска функция за Primate и затова има пълен достъп до членовете на този клас. Ако Primate е основен клас за клас Orangutan, canCommunicate() ще има достъп само до членовете на Orangutan, които са наследени от основния клас Primate. А какъв е достъпът на canCommunicate() до членовете на ZooAnimal? Да разгледаме следния пример:

```
void canCommunicate( Primate *pr ) {  
    ZooAnimal za;  
    if ( pr->onDisplay          // позволен достъп  
        && za.onDisplay        // непозволен достъп  
        && pr->forSale )        // непозволен достъп  
        // ...  
}
```

В общия случай приятелските функции имат същите права на достъп като членовете на класа, за който те са приятелски. Членовете на Primate нямат достъп до скритите членове на ZooAnimal. Следователно, canCommunicate() има достъп до onDisplay, но няма достъп до forSale.

Производните класове имат достъп само до наследените членове от секциите public и protected на основните класове и то посредством собствените си обекти. Да разгледаме още един пример:

```
Primate::locate() {  
    ZooAnimal za;  
    Primate pr;  
  
    if ( onDisplay          // позволен достъп  
        && pr.onDisplay      // позволен достъп  
        && za.onDisplay )    // непозволен достъп  
        // ...  
}
```

Primate няма достъп до член на ZooAnimal чрез обект от основния клас, ако този член не е общодостъпен. Същото правило важи и за "приятелите" на Primate. Изразът

za.onDisplay

в canCommunicate() представлява отново непозволен достъп.

Наследяването не е еквивалентно на приятелството. Наследяването разширява възможностите на даден тип. Primate е специален случай на ZooAnimal. Той приема

характеристиките на базовия клас, но добавя и свои атрибути, които се отнасят само за него. Приятелството осигурява достъп до скритите членове на даден клас. То не задава отношения между типовете. Наследяването не е специален вид приятелство — приятелство, което осигурява достъп до членовете от секциите `public` и `protected` на конкретен обект от основния клас. Единствено "приятелите" и членовете на клас `ZooAnimal` имат достъп до всички членове на този клас, използвайки конкретен негов обект.

Упражнение 7-17. Трябва ли функцията `debug()` от упражнение 7-13 да се реализира за всеки от трите вида данни — `public`, `protected` и `private`? Защо?

7.6 Смисъл на атрибута `public/private` за основен клас

Ако атрибутът на основен клас е `public`, наследените от него членове запазват своето ниво на достъп и в производния клас. Това означава, че `name` и `onDisplay`, които са `protected` в дефиницията на `ZooAnimal` ще се смятат за такива и в клас `Primate`. Ако `Baboon` произхожда от `Primate`, той ще наследи тези членове със същото ниво на достъп. В общия случай в наследствена йерархия, където всеки базов клас е `public`, всеки клас от съответната верига на наследявания има достъп до всички `public` и `protected` членове на предходните базови класове. Например, `Panda` има достъп до всички `public` и `protected` членове на класовете `Bear`, `ZooAnimal`, `Endangered` и `Herbivore`.

В наследствена йерархия, където основните класове са `private`, всеки наследен `public` или `protected` член става `private` член на производния клас. Този факт е причина за следните два ефекта:

1. Общодостъпните членове на основния клас не са достъпни чрез обект от производния клас.
2. Членове `public` и `protected` на основния клас остават изолирани за следващите класове от този клон на йерархията.

Следователно, когато основният клас е `private`, неговият общодостъпен интерфейс остава недостъпен за производните класове. Освен това, не съществува неявно преобразуване на обект от производния клас в обект от основния клас. В раздел 2.5 дефинирахме клас `IntStack`, като производен клас на `IntArray`. В тази дефиниция основният клас `IntArray` е `private`. По този начин `IntStack` може да използва вътрешното представяне на `IntArray`, но не и неговите операции и функции. Наследяването в този случай цели да създаде нов клас като използва реализацията на съществуващ клас.

Сега ще разгледаме още един пример. Администрацията на зоологическата градина иска да съхрани информация за популацията на гризачите, но не иска тя да е общодостъпна. За тази цел ще дефинираме клас `Rodent` като производен на клас `ZooAnimal`, но `ZooAnimal` ще е `private`:

```
class Rodent : private ZooAnimal {
public:
    void reportSighting( short loc );
    // ...
};
```

Тази дефиниция превръща всички членове на `ZooAnimal` в `private`-членове на `Rodent`. Следователно, наследените членове са достъпни само за членовете и "приятелите" на `Rodent`. Тогава, следващото обръщение към общодостъпната функция на клас `ZooAnimal`, `locate()`, е неправилно:


```
Rodent pest;  
pest.locate();           // грешка: ZooAnimal::locate() е скрит член в клас Rodent
```

Когато базовият клас е `private`, обект от производния клас не може да се присвои на обект от основния клас. Ако това е позволена операция, наследените членове няма да бъдат вече `private`. Нека имаме следните две декларации:

```
extern void locate( ZooAnimal& );  
ZooAnimal za;
```

Тогава употребата на `pest` в следващите оператори е неправилна и води до грешка при компилация:

```
locate( pest );          // грешка  
za = pest;               // грешка  
za.locate();             // ок
```

Всички класове, за които клас `Rodent` е основен клас, нямат достъп до членовете на `ZooAnimal`. Те имат достъп само до `public` и `private` членовете на `Rodent`. Да разгледаме един пример:

```
class Rat : public Rodent { ... };  
Rat anyRat;  
anyRat.reportSighting( CAFETERIA ); // ок  
  
// грешка: ZooAnimal::locate() е невидима за клас Rat  
anyRat.locate();
```

Ако базовият клас е `private`, програмистът може да освободи отделни негови членове от ефекта, който им оказва този факт. Например:

```
class Rodent : private ZooAnimal {  
public:  
    // за да запазите нивото на достъп, не определяйте тип на резултата и сигнатура  
    ZooAnimal::inform;  
};  
  
pest.locate();           // грешка: ZooAnimal::locate() е private  
pest.inform();           // ок: ZooAnimal::inform() е public
```

Появата на `ZooAnimal::inform` в секция `public` на клас `Rodent` позволява наследяването на тази функция да стане с нейното действително ниво на достъп. Следващите класове във веригата от наследявания също ще наследяват `inform()` като общодостъпен член:

```
class Rat : public Rodent { ... };  
Rat anyRat;  
  
// грешка: ZooAnimal::locate() е невидима за класа Rat  
anyRat.locate();
```

```
// ок: inform() е общодостъпен член на Rat
anyRat.inform();
```

Производните класове могат да поддържат наследеното ниво на достъп за членовете на основния клас, но не могат да го променят. Например, `onDisplay` е `protected` член на `ZooAnimal` и не може да се появи в секция `public` на клас `Rodent`. Аналогично, `inform()` не може да се декларира в секция `protected` на клас `Rodent`.

7.7 Стандартни преобразувания при производни класове

Между основен клас с атрибут `public` и негов производен клас съществуват четири стандартни преобразувания:

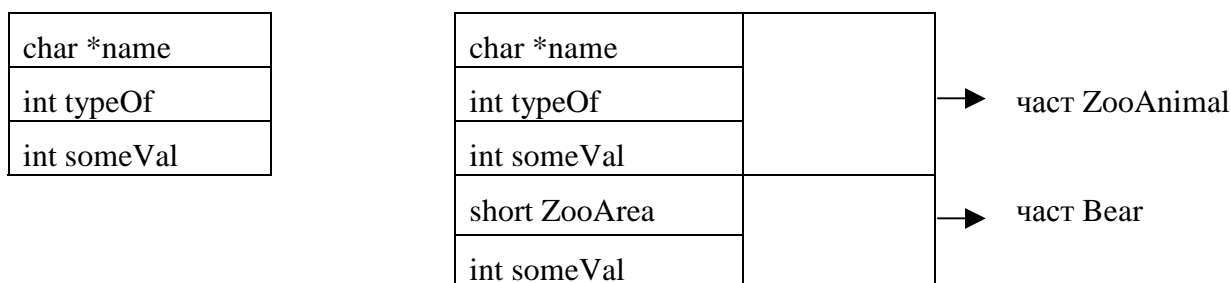
1. Обект от производния клас се преобразува неявно в обект от основния клас.
2. Алтернативно име на обект от производния клас се преобразува неявно в алтернативно име на обект от основния клас.
3. Указател към обект от производния клас се преобразува неявно в указател към обект от основния клас.
4. Указател към член на основния клас се преобразува неявно в указател към член на производния клас.

В допълнение знаем, че указател към обект от произволен клас може да се преобразува неявно в указател от тип `void*`. (Указател от тип `void*` изисква явно преобразуване за да се присвои на указател от друг тип.)

Да разгледаме следните прости дефиниции на класове. Дадена е и функция `f()`, с един аргумент, който е алтернативно име на обект от клас `ZooAnimal`.

```
class Endangered      { ... } endang;
class ZooAnimal       { ... } zooanim;
class Bear : public ZooAnimal { ... } bear;
class Panda : public Bear   { ... } panda;
class ForSale : private ZooAnimal { ... } forsale;
```

```
extern void f( ZooAnimal& );
```



ZooAnimal za;

Bear b;

Фиг. 7.4 Вътрешно представяне на производен клас

`f()` може да се извика без проблеми с обект от клас `ZooAnimal`, `Bear` или `Panda`. В две от следващите обръщения се прилага неявно стандартно преобразуване:

```
f( zooanim );      // ок: точно съответствие
f( bear );        // ок: Bear —> ZooAnimal
f( panda );       // ок: Panda —> ZooAnimal
```

Ето два примера за неправилни обръщения към f(), които предизвикват грешка при компилация. Появява се съобщение за смесване на типовете (type mismatch).

```
f( forsale );      // грешка: ZooAnimal е private
f( endang );       // грешка: ZooAnimal не е основен клас
```

Фигура 7.4 показва вътрешното представяне на обект от произведен клас, като използва следните дефиниции на ZooAnimal и Bear:

```
class ZooAnimal {
public:
    isA();
    // ...
protected:
    char *name;
    int typeOf();
    int someVal;
};

class Bear : public ZooAnimal {
public:
    locate();
    // ...
protected:
    short ZooArea;
    int someVal;
};

Bear b, *pb = &b;
ZooAnimal *pz = pb;
```

Всеки обект от произведен клас се състои от една или повече "основни части" и част, в която се намират неговите собствени членове. Всеки обект, независимо дали е от основен или произведен клас съдържа "основна част". Само производният обект има "производна част". На фигура 7.4 са дадени структурите на основния клас ZooAnimal и производния клас Bear.

Преобразуването на обект от произведен клас в обект от основен клас е безопасно, защото обектите от производния клас включват в себе си основна част. Обратното преобразуване, обаче, не е безопасно. То може да се извърши само явно. Например:

```
// pb и pz адресират един и същ обект b от клас Bear
// ок: Bear::locate()
pb—>locate();

// грешка: locate() не е член на ZooAnimal
pz—>locate();

pb—>someVal;      // ок: Bear::someVal
pz—>someVal       // ок: ZooAnimal::someVal
```

С изключение на случаите, когато се извиква виртуална функция, за осъществяване на достъп до производната част на произведен клас чрез обект от основен клас, се изисква явно преобразуване. Същото важи, ако достъпът се осъществява чрез алтернативно име или указател към обект от основен клас. Да разгледаме един пример:

```
// не се използва обектно-ориентирано програмиране
void locate( ZooAnimal *pZ ) {
    switch ( pZ—>isA() ) {
        case Bear:    ((Bear *) pZ)—>locate();
        case Panda:   ((Panda *) pZ)—>locate();
    }
}
```

Забележете, че втората двойка скоби в преобразуването на pZ е задължителна. Операторът

```
// извиква ZooAnimal::locate() и полученият резултат преобразува в Bear*
(Bear *) pZ—>locate();
```

се опитва да извика ZooAnimal::locate() и след това да преобразува получения резултат в Bear*.

Присвояването и инициализирането на обект от произведен клас с обект от негов основен клас също изисква явно преобразуване на типовете. Например:

```
Bear *pb = new Panda;           // ок: неявно преобразуване
Panda *pp = new Bear;           // грешка: няма явно преобразуване
Panda *pp = (Panda *) new Bear; // ок
```

Всеки обект от клас Panda съдържа членовете на Bear. Обектът от клас Bear може да е, но може и да не е Панда, в зависимост от последното си присвояване. Преобразуването на обект от основен клас (или указател към такъв обект) в обект (указател) от производния клас е опасна операция. Например, следващите оператори присвояват 24 на позиция в паметта, която не се свързва с конкретна променлива:

```
Bear b;
Bear *pb = new Panda;           // ок: неявно преобразуване
pb = &b;                        // pb вече не адресира обект от клас Panda
Panda *pp = (Panda *) pb;       // няма част за Panda
pp—>cell = 24;                  // бедствие! няма клетка Panda::cell
```

Указателите към членове имат обратно поведение. Това означава, че няма проблеми, ако на указател към член на произведен клас се присвои член на основен клас с атрибут public. Не е за препоръчване, обаче, на указател към член на основен клас да се присвоява член на произведен клас. Причината е в това, че когато се използва указател към член на клас, този указател винаги се свързва с обект от този клас. Например, pm_Zoo е указател към член на клас ZooAnimal, инициализиран с isA():

```
int (ZooAnimal::*pm_Zoo)() = ZooAnimal::isA;
```

Извикването на този указател се осъществява чрез свързването му с обект от клас `ZooAnimal`:

```
ZooAnimal z;  
(z.*pm_Zoo)();
```

`isA()` има достъп до членовете на `ZooAnimal`. Всеки обект от клас `Bear`, съдържа част със членовете на основния клас `ZooAnimal`, затова ако на указател към член на `Bear` се присвои адресът на `isA()`, няма да има проблеми:

```
int (Bear::*pm_Bear)() = ZooAnimal::isA();  
Bear b;  
(b.*pm_Bear)();    // ок
```

Нека сега на `pm_Zoo` присвоим адреса на `Bear::locate()`. Това присвояване не е безопасно и следователно изисква явно преобразуване:

```
pm_Zoo = (int (ZooAnimal::*) ()) Bear::locate();
```

`pm_Zoo` се извиква като се свърже с обект от клас `ZooAnimal`. `locate()` има достъп до членове на `Bear`, които не се съдържат в този обект. Дали извикването на `pm_Zoo` ще се "провали", зависи от действителния тип на обекта от основния клас. Например:

```
ZooAnimal *pz = &b;        // pz адресира обект от клас Bear  
(pz->*pm_Zoo)(); // ок  
  
pz = &z;  
(pz->*pm_Zoo)(); // бедствие!
```

Упражнение 7-18. Нека имаме следната наследствена йерархия:

```
class Node { ... };  
class Type      : public Node { ... };  
class Statement : public Type { ... };  
class Expression : public Type { ... };  
class Identifier : public Expression { ... };  
class Function   : public Expression { ... };
```

Кои от следващите инициализации са неправилни? Обяснете защо.

- (a) `Node *pn = new Identifier("i");`
- (б) `Identifier *pi = pn;`
- (в) `Expression e = *pi;`
- (г) `Statement s = e;`
- (д) `Type *pt = &e;`
- (е) `Identifier &ri = e;`
- (ж) `Function &rf = ri;`
- (з) `Node n = rf;`

Упражнение 7-19. Дефинирайте функция `debug()`, която не е член на класовете от наследствената йерархия на геометричните фигури, чийто аргумент може да бъде обект от всеки такъв клас. Тя трябва да извиква съответната на този клас функция `debug()`. (В глава 8 ще дефинираме `debug()` като виртуална функция.)

Упражнение 7-20. Дефинирайте указатели към членове на класовете `Shape`, `Circle`, `Box` и `Rectangle`. Посочете позволени присвоявания и присвоявания, които изискват явно преобразуване.

7.8 Области на действие и производни класове

Наследяването предизвиква влагане на области на действие. От фигура 7.5 се вижда, че производният клас е ограден от областите на действие на своите базови класове.

Всеки клас и външна функция са видими в глобалната област на действие. Техните области на действие са изобразени като правоъгълници със запълване от точки. Всяка функция-член на клас е видима в областта на действие на своя клас. Нейната собствена област на действие е изобразена като правоъгълник без запълване. Стрелките свързват всяка област на действие с нейната ограждаща област на действие. При множествено наследяване съществуват няколко ограждащи области на действие. На една и съща стъпка на търсене на идентификатор, в процеса на компилация, трябва да се прегледа всяка от тези области на действие. Например, ако идентификаторът се среща в клас `Panda` и не принадлежи на неговата област на действие, на една и съща стъпка трябва да се прегледат областите на действие на трите основни класа — `Bear`, `Endangered` и `Herbivore`. Фигура 7.5 използва следните дефиниции на класове:

```
extern inform();

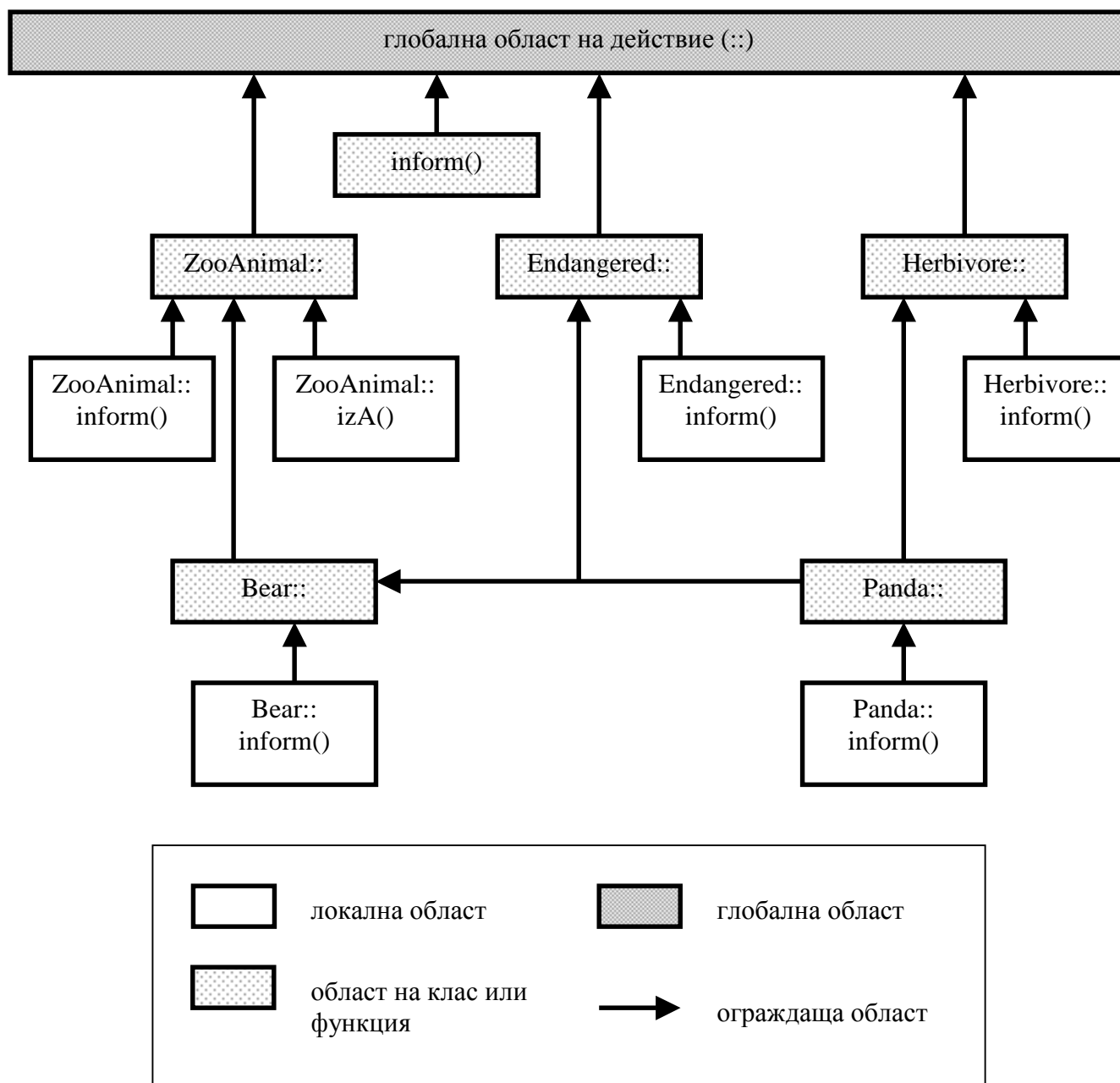
class ZooAnimal {
public:
    inform();
    isA();
};

class Endangered      { public: inform( ZooAnimal* ); };
class Herbivore       { public: inform( ZooAnimal* ); };
class Bear : public ZooAnimal { public: inform( short ); };
class Panda : public Bear, public Endangered, public Herbivore
                { public: inform( char* ); };
```

Ако в `Panda::inform()` се появи обръщение към `isA()`, търсенето на функцията, която трябва да се извика, преминава през следните три стъпки:

1. Проверява се дали `isA()` не е декларирана в `Panda::inform()`. Ако има такава декларация, процесът на търсене спира. В противен случай се извършва предвижване по посока на стрелките. Виж фигура 7.5.
2. Проверява се дали клас `Panda` не съдържа член `isA()`. Ако в дефиницията на `Panda` има такъв член, процесът на търсене спира. В противен случай отново се извършва предвижване в посока на стрелките.
3. Проверява се дали клас `Panda` не е наследил `isA()` от един от своите три основни класа — `Bear`, `Endangered` и `Herbivore`. Ако това е така, процесът на търсене спира. Ако повече от един основни класа имат член `isA()`, се получава противоречие. Извежда се съобщение за грешка. В противен случай се проверява дали някой от тези три основни класа не е също производен клас. Ако има такъв клас, търсенето

продължава в неговите основни класове. Ако основните класове на Panda не произлизат от други класове, търсенето продължава в глобалната област на действие. Описаният в тази точка процес се повтаря до откриване на клас с търсения член или до попадане в глобалната област на действие. Ако и в глобалната област на действие не се открие търсеният идентификатор, се извежда съобщение за грешка.



Фиг. 7.5 Област на действие и производни класове

В нашия конкретен пример, обръщението към `isA()` в `Panda::inform()` се отнася за члена, наследен от клас `ZooAnimal`.

Производни класове и функции с еднакви имена

Клас `Bear` наследява от `ZooAnimal` член `locate()` с различна сигнатура от тази на собственият му член `Bear::locate()`. Програмисти без опит в програмирането на C++ често

предполагат, че различната сигнатура е достатъчна, за да се различат тези две функции и да може да се извика всяка една от тях. Дали това е така показва следващият пример:

```
Bear ursus;  
ursus.locate( 1 );    // ок: извиква се Bear::locate(int)  
usus.locate();        // грешка: функцията ZooAnimal::locate() е покрита
```

`ZooAnimal::locate()` се покрива от `Bear::locate(int)`. Обръщението `ursus.locate()` се смята за обръщение към `Bear::locate(int)` и следователно предизвиква грешка при компилация, заради неправилна сигнатура.

За да можем да извикаме всяка една от функциите с еднакво име, трябва всички те да бъдат дефинирани в една и съща област на действие. В горния пример това не е така, но можем да дефинираме втора функция-член на клас `Bear` с име `locate` и без аргументи. Чрез тази функция ще осъществим достъп до покритата функция `ZooAnimal::locate()`:

```
inline void Bear::locate() { ZooAnimal::locate(); }
```

7.9 Инициализация и присвояване при производни класове

Според раздел 6.2 присвояване и инициализация на обект с друг обект от същия клас се извършва чрез копиране на членове. Същото правило е в сила и за обектите на производен клас. То се прилага най-напред за основните класове поред на тяхното деклариране в списъка на основните класове. След това се прилага за членове на производния клас, които са обекти.

Знаем, че ако инициализацията чрез копиране използва служебния конструктор `X(const X&)`, ще възникнат проблеми, когато класът има членове-указатели. В този случай деструкторът на класа ще се извика два пъти, а това означава, че ще се освободи свободна или заета с друга цел памет. Затова се налага да се дефинира собствен конструктор от вида `X(const X&)`. При наличие на такъв конструктор, компилаторът ще извика него вместо служебния конструктор.

В този раздел ще се спрем на особеностите при присвояване и инициализация чрез копиране за производни класове. В частност ще разгледаме следните три случая:

1. Производният клас не притежава собствен конструктор от вида `X(const X&)`, но един или повече основни класове имат такъв конструктор.
2. Нито производният клас, нито базовите класове имат свой конструктор `X(const X&)`.
3. Производният клас и всички основни класове притежават собствен конструктор `X(const X&)`.

Ако производният клас няма свой конструктор `X(const X&)`, при инициализация на обект от този клас с друг обект от същия клас, се прилага служебният конструктор `X(const X&)`. Преди това, обаче, се инициализират основните класове поред на тяхното деклариране в списъка с основни класове. За всеки основен клас със собствена дефиниция на `X(const X&)`, се прилага именно този конструктор. За останалите базови класове се използва служебният конструктор. Да разгледаме един пример:

```
class Carnivore { public: Carnivore(); };  
class ZooAnimal {  
public:  
    ZooAnimal();  
    ZooAnimal( const ZooAnimal& );
```



```
};

class Cat : public ZooAnimal, public Carnivore { public: Cat(); };

Cat Felix;
Cat Fritz = Felix;
```

Когато Fritz се инициализира с Felix, най-напред се извиква ZooAnimal(const ZooAnimal&). Основният клас Carnivore и членовете на Cat, които са обекти, се инициализират от съответния служебен конструктор.

Ако производният клас притежава собствен конструктор X(const X&), той се използва за инициализиране на обект от класа с друг обект от същия клас. Наследените членове не се инициализират чрез копиране. От конструктора X(const X&) на производния клас зависи как се инициализират те. Например:

```
class Bear : public ZooAnimal {
public:
    Bear();
    Bear( const Bear& );
};

Bear Yogi;

Bear Smokey = Yogi;
```

При инициализиране на Smokey с обекта Yogi се извикват последователно следните два конструктора:

```
ZooAnimal();
Bear(const Bear&);
```

Знаем, че конструкторите на основните класове се извикват винаги преди конструктора на производния клас. Това правило е в сила и за конструкторите от вида X(const X&). Ако конструкторът на основния клас изисква параметър, такъв може да се осигури чрез списъка за инициализация на членове.

Тъй като в нашия пример клас ZooAnimal има собствен конструктор от вида X(const X&), за предпочитане е именно той да се извика. Това може да се осъществи по следния начин:

```
Bear::Bear( const Bear& b )
    // извиква се ZooAnimal(const ZooAnimal&)
    : ZooAnimal(b)
    { /* ... */ }
```

В този случай инициализацията на обекта Smokey с обекта Yogi предизвиква обръщение към следната последователност от конструктори:

```
ZooAnimal(const ZooAnimal&);
Bear(const Bear&);
```

Ако основният клас не притежава свой конструктор за инициализиране чрез копиране, ще се използва служебният конструктор `X(const X&)`.

Ако обект от произведен клас се присвоява на обект от същия клас, действат аналогични правила. Когато производният клас не притежава собствена операция `operator=(const X&)`, базовите класове и членовете-обекти със собствена дефиниция на тази операция, използват своята операция. Останалите използват служебната операция.

Ако производният клас притежава своя операция `operator=(const X&)`, именно тя се прилага. Присвояването за съответните базови класове и членове-обекти зависи от дефиницията на тази операция.

Какво ще се получи, ако на обект от клас `ZooAnimal` се опитаме да присвоим обект от неговия произведен клас `Bear`? Коя от двете операции `operator=(const X&)` ще се извика, когато и двете съществуват?

```
Bear ursus;  
ZooAnimal onLoan;  
  
onLoan = ursus;
```

За да отговорим на поставените въпроси, трябва да напомним правилото, че типът на левия операнд определя типа на присвояването. В нашия случай "основната" част на `ursus` ще се присвои на `onLoan`. Следователно това присвояване може да се разглежда като присвояване между два обекта от клас `ZooAnimal`. Следователно, се извиква операцията за присвояване на клас `ZooAnimal`. Същото правило важи и за инициализирането на обекти. Например, декларацията

```
Bear Fozzie;  
ZooAnimal On_Loan = Fozzie;
```

предизвиква обръщение към `ZooAnimal(const ZooAnimal&)`, а не към `Bear(const Bear&)`.

В общия случай, ако класът има своя дефиниция за `X(const X&)` или `operator=(const X&)`, е добре да притежава и двете. Собствени дефиниции са необходими в следните два случая:

1. Семантиката на даден клас не позволява да се копират един или повече негови членове. Например, такъв е членът `index` на клас `String`. При всяко копиране този член трябва да се нулира.
2. Класът притежава член-указател и деструктор.

Упражнение 7-21. Дефинирайте `X(const X&)` и `operator=(const X&)` за клас `Shape`. Нека клас `Circle` не притежава такива дефиниции. Кои конструктори се извикват в следващия пример?

```
Circle c1;  
Circle c2 = c1;
```

Упражнение 7-22. Дефинирайте `X(const X&)` и `operator=(const X&)` за клас `Box`. Нека клас `Rectangle` не притежава такива дефиниции. Кои конструктори се извикват в следващия пример?

```
Rectangle r1;  
Box b1 = r1;  
Rectangle r2 = r1;
```

7.10 Ред на инициализиране при производни класове

Нека имаме следната проста дефиниция на клас Panda:

```
class Panda : public Bear, public Endangered, public Herbivore  
{  
public:  
    Panda();  
    // ... останалите общодостъпни членове  
protected:  
    BitVector status;  
    String name;  
    // ... останалите членове на клас Panda  
};  
  
Panda yinYang;
```

При дефиниране на обекта yinYang се извикват последователно следните конструктори:

1. Конструкторите на основните класове, поред на тяхното деклариране в списъка на основните класове:

```
ZooAnimal();    // ZooAnimal е основен клас за Bear  
Bear();  
Endangered();  
Herbivore();
```

2. Конструкторите на класове, чиито обекти са членове на Panda. Редът на извикване съответства на реда на деклариране на тези членове в тялото на Panda.

```
BitVector();  
String();
```

3. Конструкторът на производния клас:

```
Panda();
```

Деструкторите за yinYang се извикват в обратна последователност.

Макар че фиксираният ред на извикване на конструкторите за основните класове и членовете-обекти обикновено не е от значение за потребителя, целостта на някои приложения зависи от този ред. Например, библиотека, която позволява на произволно сложни потребителски типове данни да се записват и четат от диска, трябва да е сигурна, че обектите се конструират по един и същ начин, независимо от компютъра и компилатора. C++ гарантира единственост, като фиксира реда на извикване на конструкторите и деструкторите.

ГЛАВА 8

Обектно-ориентираното програмиране се характеризира с механизмите на наследяване и динамично свързване. Наследяването се реализира чрез производни класове, които бяха тема на предишната глава. Динамичното свързване се реализира от виртуални функции.

Между класовете в една йерархия съществуват отношения от вида тип/подтип. Например, Panda е подтип на Bear, а Bear е подтип на ZooAnimal. Аналогично сортираният масив и масивът, който проверява границите, са подтипове на IntArray. Виртуалните функции дефинират операции, които зависят от типа на аргумента. Този аргумент може да бъде обект от произволен клас в йерархията. Примери за виртуални функции са draw() за йерархията на животните и операцията за индексване за различните типове масиви. Виртуалните функции капсулират детайлите в реализацията на класовете от йерархията.

В тази глава ще се спрем подробно на виртуалните функции. Ще разгледаме и един специален вид наследяване — това на виртуалните (поделени) основни класове. Първо, обаче, ще обсъдим функциите с еднакви имена, чиито аргументи са обекти от някакъв клас.

8.1 Функции с еднакви имена с аргумент обект от клас

Често функции с еднакви имена се създават за обектите на даден клас, за да дефинират действия, които вече съществуват за други типове данни. Например за клас Complex може да се дефинира операция за пресмятане на квадратен корен от комплексно число:

```
extern Complex& sqrt( Complex& );
```

Раздел 4.3 разглежда подробно функциите с еднакви имена, но изключва случая, когато аргументите са обекти от някакъв клас. В този раздел ще се спрем именно на този случай. Последователно ще разгледаме получаването на точно съответствие, съответствие чрез стандартно преобразуване или чрез преобразуване, дефинирано от потребителя.

Точно съответствие

Точно съответствие се получава само, ако формалният и фактическият параметър са обекти от един и същ клас. Например:

```
ff( Bear& );  
ff( Panda& );  
  
Panda yinYang;  
ff(yinYang);           // точно съответствие с ff(Panda&)
```

Ситуацията е аналогична, когато формалният аргумент е указател към обект от определен клас. Точно съответствие се получава само, ако фактическият аргумент е указател към обект от същия клас.

Алгоритъмът за търсене на съответствие не прави разлика между обект и алтернативно име на обект. Макар че в следващия пример са декларирани две различни функции, обръщението към всяка една от тях е противоречиво и ще предизвика грешка при компилация:

```
// двете функции не се различават от алгоритъма за търсене на съответствие
```

```
ff( Panda );
ff( Panda& );

int (*pf) ( Panda& ) = ff;    // ок: ff( Panda& )

ff( yinYang );                // грешка: противоречие
pf( yinYang );                // правилно обръщение
```

Стандартни преобразувания

Когато не съществува точно съответствие между формалния и фактическия аргумент, се прави опит за постигане на съответствие чрез стандартно преобразуване. Възможни са следните стандартни преобразувания:

- Всеки обект, указател към обект или алтернативно име на обект от произведен клас неявно се преобразува в обект, указател към обект или алтернативно име на обект от основен клас. Основният клас трябва да бъде с атрибут `public`. Например:

```
ff( ZooAnimal& );
ff( Screen& );

ff( yinYang );    // съответствие с ff(ZooAnimal&)
```

- Указател към обект от произволен клас неявно се преобразува в указател от тип `void*`.

```
ff( Screen& );
ff( void* );

ff( yinYang );    // съответствие с ff( void* );
```

Обект, указател към обект или алтернативно име на обект от основен клас не може да се преобразува в обект, указател към обект или алтернативно име на обект от негов произведен клас. Например:

```
ff( Bear& );
ff( Panda& );

ZooAnimal za;

ff( za );    // грешка: съответствие не може да се получи
```

Когато е възможно стандартно преобразуване към няколко непосредствени основни класове, възниква противоречие. Например, `Panda` е пряк наследник на два класа — `Bear` и `Endangered`. Никое от двете възможни стандартни преобразувания няма приоритет над другото. Затова се получава противоречие:

```
ff( Bear& );
ff( Endangered& );
ff( yinYang );    // грешка: противоречие
```

Противоречието се разрешава чрез явно преобразуване:

```
ff( Bear(yinYang));
```

Всеки производен клас е по-близък до базовия клас, от който непосредствено произлиза, отколкото до базовите класове от по-горните нива на йерархията. Следващото обръщение не е противоречиво, макар че и в двата случая са възможни стандартни преобразувания. Причина за това е фактът, че Panda е най-напред вид мечка и едва след това някакво животно.

```
ff( ZooAnimal& );  
ff( Bear& );  
  
ff( yinYang );      // съответствие с ff( Bear& )
```

Същото правило важи и при работа с указатели от тип void*. Да разгледаме следната двойка функции с еднакво име:

```
ff( void* );  
ff( ZooAnimal* );
```

При обръщение към ff() с аргумент от тип Panda*, се получава съответствие с ff(ZooAnimal*).

Преобразувания, дефинирани от потребителя

Потребителските преобразувания се осъществяват от конструктори с един аргумент или от явни операции за преобразуване. Те се прилагат само в краен случай — когато няма точно съответствие и такова не може да се получи чрез стандартно преобразуване. Нека за илюстрация клас ZooAnimal притежава две потребителски преобразувания:

```
class ZooAnimal {  
public:  
    ZooAnimal( long );      // преобразува long —> ZooAnimal  
    operator char*();      // преобразува ZooAnimal —> char*  
    //...  
};
```

Да разгледаме следната двойка функции с еднакво име:

```
ff( ZooAnimal& );  
ff( Screen& );
```

Ако извикаме ff() с аргумент от тип long, ще получим съответствие чрез дефинираното от потребителя преобразуване на число от тип long в обект от клас ZooAnimal:

```
long lval;  
ff( lval );      // съответствие с ff( ZooAnimal& )
```

Какво ще се случи, ако `ff()` се извика с аргумент от тип `int`? Например:

```
ff( 1024 ); // ???
```

В този случай няма точно съответствие. Съответствие не може да се получи и чрез стандартно преобразуване. Почти съществува приложимо потребителско преобразуване. Проблемът е в това, че конструкторът на клас `ZooAnimal` изисква аргумент от тип `long`, а не от тип `int`. Затова първо се използва стандартно преобразуване и едва след това се прилага потребителското преобразуване. Следователно, числото 1024 се преобразува в тип `long` и едва след това се предава на конструктора `ZooAnimal(long)`.

Вече споменахме, че преобразуванията, дефинирани от потребителя, се прилагат само в краен случай. Следователно, ако съществува функция `ff()` с аргумент от стандартен тип, потребителските преобразувания за клас `ZooAnimal` не се използват. Например:

```
ff( ZooAnimal& );  
ff( char );  
  
long lval;  
ff( lval ); // съответствие с ff(char)
```

В този случай потребителското преобразуване може да се изпълни само явно:

```
ff( ZooAnimal( lval ) ); // съответствие с ff(ZooAnimal&)
```

В следващия пример обектът от клас `ZooAnimal` се преобразува в тип `char*` чрез потребителско преобразуване, защото не съществува стандартно преобразуване на обект от основен клас в обект от производен клас.

```
ff( char* );  
ff( Bear& );  
ZooAnimal za;  
  
ff( za ); // съответствие с ff(char*) чрез преобразуване на za в тип char*
```

Алгоритъмът за търсене на съответствие може да получи съответствие като приложи стандартно преобразуване към резултата от потребителско преобразуване. Например:

```
ff( Panda* );  
ff( void* );  
  
// za —> char* —> void*  
ff( za ); // съответствие с ff(void*)
```

Операциите за преобразуване (но не и конструкторите) се наследяват от производните класове. Следователно, `Bear` и `Panda` наследяват от клас `ZooAnimal` операцията, която преобразува обект в тип `char*`. Да разгледаме един пример:

```

ff( char* );
ff( Bear* );

Bear yogi;
Bear *pBear = &yogi;

ff( yogi );           // съответствие с ff(char*)
ff( pBear );          // съответствие с ff(Bear*)

```

Ако могат да се приложат няколко потребителски преобразувания, обръщението е противоречиво и поражда грешка при компилация. Конструкторите за преобразуване и операциите за преобразуване имат един и същ приоритет. Ще разгледаме един пример за противоречиво обръщение. Клас `Endangered` разполага с операция, която преобразува негов обект в тип `int`. Тази операция е дефинирана от потребителя:

```

class Endangered {
public:
    operator int(); // преобразува Endangered —> int
    // ...
};

```

Нека освен това клас `Extinct` притежава конструктор с аргумент `Endangered&`:

```

class Extinct {
public:
    Extinct( Endangered& ); // преобразува Endangered —> Extinct
    // ...
};

```

Следващото обръщение е противоречиво, защото операцията за преобразуване `Endangered::operator int()` и конструкторът `Extinct::Extinct(Endangered&)` са еднакво приложими.

```

ff( Extinct& );
ff( int );

Endangered e;
ff( e ); // противоречиво обръщение

```

Ще разгледаме още един пример за противоречиво обръщение, породено от възможността да се приложат няколко потребителски преобразувания. В този случай еднакво приложими са конструкторите на класовете `BitVector` и `SmallInt`.

```

class SmallInt {
public:
    SmallInt( int ); // преобразува int —> SmallInt
    // ...
};

class BitVector {
public:

```



```

    BitVector( unsigned long );    // преобразува unsigned long —> BitVector
    // ...
};

ff( SmallInt& );
ff( BitVector& );

ff( 1 );    // противоречиво обръщение

```

8.2 Виртуални функции

Виртуална функция е специална функция-член на клас, която се извиква с алтернативно име или указател към обект от основен клас. Основният клас трябва да е с атрибут `public`. Например, `draw()` е виртуална функция. Всеки един от класовете `ZooAnimal`, `Bear`, `Panda`, `Cat` и `Leopard` притежава собствена дефиниция на тази функция. Коя от функциите `draw()` ще се изпълни зависи от действителния тип на аргумента. Ето как може да се дефинира функцията, която вика `draw()` за някой от петте класа:

```

inline void draw( ZooAnimal& z ) { z.draw(); }

```

Тази функция не е член на клас. Ако тя се извика с аргумент, който адресира обект от клас `Panda`, операторът `z.draw()` ще извика `Panda::draw()`. Ако следващият път тя се извика с аргумент, който адресира обект от клас `Cat`, ще се изпълни `Cat::draw()`. Компиляторът решава коя функция ще изпълни в зависимост от действителния тип на аргумента.

Преди да покажем как се декларира и използват виртуалните функции, нека накратко разгледаме основанията за тяхното използване.

Динамичното свързване — вид капсулиране

Последният екран от представянето на животните е картина на всички животни, от които потребителят се интересува. Този екран е интересен за малките посетители на зоологическата градина и обикновено се превръща в атракция. За да го нарисуваме, ще поддържаме списък от указатели към избраните от потребителя животни. При натискане на бутона `QUIT`, списъкът се подава на функцията `finalCollage()`. Тя се грижи за изрисуване на животните в подходящи размери и разположение на екрана.

Свързаният списък от животни се създава просто, защото всеки указател от тип `ZooAnimal` може да адресира обект от негов произведен клас. Динамичното свързване позволява да се определи типът на конкретното животно и в зависимост от това да се извика съответната функция `draw()`. `finalCollage()` може да се дефинира така:

```

void finalCollage( ZooAnimal *pz ) {
    for ( ZooAnimal *p = pz; p; p = p—>next )
        p—>draw();
}

```

Езици, които не поддържат динамично свързване, оставят на програмиста грижата да определи действителния тип на аргумента, адресиран от `pz`. Тогава типът на аргумента се определя като за всеки клас се дефинира член `isA()` и се използват операторите `if-else` или `switch`. В зависимост от типа се извиква съответната функция `draw()`. Дефиницията на `finalCollage()` изглежда така:

```
// дефиниция на finalCollage(), ако не се поддържа динамично свързване
void finalCollage( ZooAnimal *pz ) {
    for ( ZooAnimal *p = pz; p; p = p—>next )
        switch( p—>isA() ) {
            case BEAR:      ((Bear*) p)—>draw(); break;
            case PANDA:     ((Pabda*) p)—>draw(); break;
            // по същия начин за останалите производни класове
        }
    }
}
```

Написаните по този начин програми зависят от детайлите в реализацията на йерархията. При всяка промяна на тези детайли, програмата може да спре да работи и ще трябва да се коригира.

След като пандата напусне зоологическата градина и се върне в Китай, клас Panda трябва да изчезне. Когато от Австралия пристигнат коали, трябва да се добави нов клас Koala. Всяка промяна в съществуващата йерархия на животните изисква откриване на операторите if-else и switch, които проверяват типа на обекта. Това означава, че програмите ще се променят при всяка промяна в йерархията.

От друга страна даже средно големи оператори switch увеличават значително размера на програмите. Дори простите действия стават трудни за разбиране, когато програмите съдържат много проверки. В заключение става ясно, че такива програми са не само трудни за разбиране, но и за поддръжка.

Потребители, които искат да разширят йерархията или да направят приложения по поръчка, трябва да имат достъп до текстовете на програмите, а това затруднява и оскъпява разпространението и поддръжката на системата.

Динамичното свързване капсулира детайлите в реализацията на йерархията. Вече не се налага проверка на типа. Текстовете на програмите се опростяват. Промени се налагат много по-рядко. Разширяването на йерархията не създава проблеми. Общата функция draw() не е необходимо да знае производните класове, които ще се появят в бъдеще. Програмите не зависят от промените в йерархията. Те могат да се разпространяват като .EXE файлове. Промени ще има само в заглавните файлове, които се използват от системата.

Опростява се и създаването на приложения по поръчка. Тъй като реализацията на класовете и йерархията се капсулират, с минимални промени в програмата може да се създаде променена система, която да задоволи изискванията на новия клиент.

Дефиниране на виртуални функции

За виртуална се смята функция, чиято декларация започва с ключовата дума virtual. Само функции, които са членове на клас, могат да бъдат виртуални. Ключовата дума virtual може да се среща само в тялото на класа. Например:

```
class Foo {
public:
    virtual bar();    // декларира виртуална функция
};

int Foo::bar() { ... }
```

В следващата декларация на клас ZooAnimal се декларират четири виртуални функции: debug(), locate(), draw() и isOnDisplay().

```
#include <stream.h>

class ZooAnimal {
public:
    ZooAnimal( char *whatIs = "Зоологическа градина" )
    : isa( whatIs ) { }
    void isA() { cout << "\n\t" << isa << "\n"; }
    void setOpen( int status ) { isOpen = status; }
    virtual isOnDisplay() { return isOpen; }
    virtual void debug();
    virtual void draw() = 0;
protected:
    virtual void locate() = 0;
    char *isa;
    char isOpen;
};

void ZooAnimal::debug() {
    isA();
    cout << "\tОтворена ли е? — " << ( isOnDisplay() ) ? "да" : "не" ) << "\n";
}
```

Функциите `debug()`, `locate()`, `draw()` и `isOnDisplay()` са членове на клас `ZooAnimal`, защото осигуряват процедури, които са общи за цялата йерархия на животните. Те са виртуални, защото действителната им реализация зависи от типа на класа, а той на този етап е неизвестен. Дефинициите на виртуалните функции на основен клас се използват в случаите, когато произведен клас няма дефиниция на съответната виртуална функция.

Част от виртуалните функции, дефинирани в основния клас на йерархията, често не са предназначени да се извикват с обекти от този клас — в нашия пример такива са `locate()` и `draw()`. Програмистът показва, че виртуалната функция е недефинирана за даден абстрактен клас, като инициализира нейната декларация с 0:

```
virtual void draw() = 0;
virtual void locate() = 0;
```

`draw()` и `locate()` се наричат чисти виртуални функции. Клас, който съдържа поне една чиста виртуална функция, може да се използва само като основен клас за цялата йерархия. Обекти от такъв клас не могат да се дефинират. Следващите две дефиниции ще предизвикат грешка при компилация:

```
ZooAnimal *pz = new ZooAnimal;    // грешка
ZooAnimal za;                     // грешка
```

Следователно, само абстрактен клас, който няма да създава свои обекти, може да декларира чиста виртуална функция.

Първият клас, който декларира дадена виртуална функция, трябва да и осигури дефиниция или да я декларира като чиста виртуална функция.

- Ако функцията получи дефиниция, тя се използва по подразбиране за следващите класове в йерархията, които нямат своя дефиниция за тази функция.

- Ако функцията е декларирана като чиста виртуална функция, следващите класове в йерархията трябва да я декларират също като чиста виртуална функция или да и осигурят дефиниция.⁷

Според второто правило клас Bear трябва да дефинира draw() и locate() или да предекламира двете функции като чисти виртуални функции.

Как да постъпим, ако искаме да дефинираме обекти от клас Bear, но в същото време желаем за различните видове мечки, например Panda и Grizzly, да се изпълнява различна функция draw()? Ясно е, че не може да декларираме Bear::draw() като чиста виртуална функция, защото няма да можем да дефинираме обекти от клас Bear. По-долу са представени три алтернативни решения на този проблем:

1. Bear::draw() може да има празно тяло:

```
class Bear : public ZooAnimal {
public:
    void draw() { }
    // ...
};
```

2. Извикването на Bear::draw() може да генерира вътрешна грешка:

```
void Bear::draw() {
    error( INTERNAL, isa, "draw()" );
}
```

3. Дефиницията на Bear::draw() може да регистрира неочакваните ситуации при работа на системата. За всяко изключение може да се изпраща по един запис във файл за регистрация на неочакваните ситуации. По този начин системата няма да приключи работа поради грешка, а във файла за регистрация ще се натрупа информация, която може да се използва за управление на появилите се неочаквани ситуации.

Всеки производен клас може да притежава своя дефиниция на дадена виртуална функция или да наследи дефиницията на тази виртуална функция от основен клас. От друга страна производният клас може да дефинира нови виртуални функции. Например, клас Bear предефинира debug(), locate() и draw(), наследява ZooAnimal::isOnDisplay() и дефинира две нови виртуални функции hibernates() и feedingHours():

```
class Bear : public ZooAnimal {
public:
    Bear( char *whatIs = "Мечка" )
        : ZooAnimal( whatIs ), feedTime( "2:30" )
    {} // тялото на конструктора нарочно е празно
    void draw(); // замества ZooAnimal::draw()
    void locate(); // замества ZooAnimal::locate()
    virtual char *feedingHours() { return feedTime; }
protected:
    void debug(); // замества ZooAnimal::debug()
    virtual hibernates() { return 1; }
    char *feedTime;
```

⁷ Това правило е променено в стандарт 2.1. Там по подразбиране чистите виртуални функции се наследяват.

```
};
```

Когато производен клас предефинира виртуална функция, името, сигнатурата и типът на резултата, трябва да съответстват точно на тези в основния клас. Ключовата дума `virtual` не е необходима, но ако потребителят желае може да я използва. Дефиницията на такава функция не се различава от дефиницията на обикновена функция-член на клас:

```
void Bear::debug()
{
    isA();
    cout << "\tВреме за ядене: " << feedingHours() << "\n";
}

void Bear::draw()      { /* ... тяло на функцията */ }
void Bear::locate()    { /* ... тяло на функцията */ }
```

Целият виртуален механизъм се управлява неявно от компилатора. Програмистът трябва да постави само ключовата дума `virtual` пред първата дефиниция на желаната функция. Ако името, сигнатурата или типът на резултата на предекларирана виртуална функция не съответстват точно на тези в първата дефиниция на функцията, тя няма да е виртуална за съответния производен клас. Ако клас `Bear` декларира `debug()` по един от следните два начина

```
void *Bear::debug() {...}    // различен тип на резултата
void Bear::debug( int ) {...} // различна сигнатура
```

`debug()` няма да бъде виртуална за клас `Bear`. Например:

```
Bear b;
ZooAnimal &za = b;
za.debug(); // извиква ZooAnimal::debug()
```

Въпреки това класовете, за които `Bear` е основен клас, могат да дефинират `debug()` като своя виртуална функция. Например:

```
class Panda : public Bear {
public:
    void debug();    // виртуална функция
    // ...
};
```

Декларацията на `Panda::debug()` съответства точно на декларацията на `ZooAnimal::debug()`, затова `Panda::debug()` е виртуална функция:

```
Panda p;
ZooAnimal &za = p;
za.debug();    // Panda::debug()
```

Забележете, че нивото на достъп за две от виртуалните функции се различава в основния и производния клас. `ZooAnimal::locate()` се намира в секция `protected`, а `Bear::locate()` е общодостъпен член на `Bear`. Аналогично, `ZooAnimal::debug()` е общодостъпен член на `ZooAnimal`, а `Bear::debug()` е декларирана в секцията `protected`. Какво е действителното ниво на достъп за тези две функции? Да разгледаме следната функция със общо предназначение:

```
void debug( ZooAnimal& z )
{
    z.debug();    // компилаторът решава коя конкретна функция ще извика
}
```

След като `Bear::debug()` се намира в секция `protected`, позволено ли е следващото обръщение?

```
main()
{
    Bear ursus;
    ursus.debug();
}
```

Отговорът е отрицателен: `ursus.debug()` е неправилно обръщение, защото нивото на достъп за виртуалните функции се определя от типа на указателя или алтернативното име чрез които се осъществява обръщението към функцията. Тъй като `debug()` е `protected`-член на клас `Bear`, посоченото обръщение е неправилно. Ще разгледаме още един пример. Забележете, че в него `ZooAnimal` не трябва да съдържа чисти виртуални функции.

```
main()
{
    ZooAnimal *pz = new Bear;
    pz->debug(); // извиква Bear::debug()
    pz = new ZooAnimal;
    pz->setOpen(1); // отваря зоологическата градина
    pz->debug(); // извиква ZooAnimal::debug()
}
```

След компилация и изпълнение на програмата получаваме следния резултат:

```
Мечка
Време за ядене: 2:30

Зоологическа градина
Отворена ли е ? — да
```

Извикването на виртуалната функция `debug()` чрез указател или алтернативно име на обект от клас `Bear` ще предизвика грешка. Следващото обръщение е опит за достъп до скрит член на клас `Bear` и води до грешка при компилация:

```
main()
{
    ZooAnimal *pz = new Bear;
```

```

pz—>debug(); // извиква Bear::debug()

Bear *pb = (Bear *) new ZooAnimal;    // опасно преобразуване
// неправилно обръщение: main() няма достъп до скритите членове на клас Bear
pb—>debug();
}

```

Аналогично, locate() е скрит член на ZooAnimal и общодостъпен член на Bear. Следователно тази функция може да се извика с указател или алтернативно име на обект от клас Bear, но не може да се извика чрез указател или алтернативно име на обект от клас ZooAnimal. Следващото обръщение е неправилно. Грешката може да се избегне, ако дефинираната по-долу функция locate() е приятелска за клас ZooAnimal.

```

void locate( ZooAnimal *pz )
{
    // locate() няма достъп до скритите членове на ZooAnimal, ако не е приятелска функция
    pz—>locate();    // грешка: непозволен достъп
}

```

Горните два примера показват, че виртуалните функции се подчиняват на правилата за скриване на информация. Разликата е в това, че тяхното ниво на достъп се определя от типа на класа, чрез който се извършва обръщението.

Bear предефинира три от четирите виртуални функции на ZooAnimal и дефинира две нови виртуални функции. Panda наследява шестте виртуални функции, видими в основния клас Bear.

Три виртуални функции на ZooAnimal се предефинират в Bear, но те остават наследени членове за клас Bear и могат да се извикат явно. Например, Bear::debug() може да се реализира така:

```

void Bear::debug() {
    ZooAnimal::debug();
    cout << "\tВреме за ядене: " << feedTime << "\n";
}

```

Клас Panda произлиза от три основни класа — Bear, Endangered и Herbivore. Endangered и Herbivore дефинират по две виртуални функции.

```

class Endangered {
public:
    virtual adjustPopulation( int );
    virtual highlight( short );
    // ...
};

class Herbivore {
public:
    virtual inform( ZooAnimal& );
    virtual highlight( short );
    // ...
};

```

Panda наследява общо десет виртуални функции от трите основни класа. Panda може да предефинира някои от тези функции и да дефинира нови виртуални функции.

Panda наследява две виртуални функции с име highlight(). Ако highlight() се вика чрез указател или алтернативно име на обект от клас Panda, или Panda е основен клас за други класове, ще възникне проблем. В двата случая обръщението към highlight() е противоречиво. Например:

```
RedPanda : public Panda { ... };
RedPanda pandy;
Panda &rp = pandy;
// ...
rp.highlight(); // грешка: противоречиво обръщение
```

За да предотвратим опасността от противоречиво обръщение, в раздел 7.4 дефинирахме Panda::highlight(). Ето една дефиниция на клас Panda:

```
class Panda : public Bear, public Endangered, public Herbivore
{
public:
    Panda( char *whatIs = "Панда" ) : Bear( whatIs ) { }
    virtual onLoan();
    inform( ZooAnimal& );
    void draw();
    int debug();
    void locate();
    hibernates() { return 0; }
protected:
    highlight( short );
    short cell;
    short onLoan;
};
```

Основният клас знае за следващите дефиниции на своите виртуални функции. Коя конкретна функция ще се извика, зависи от действителния тип на аргумента, с който се извършва обръщението. Базовият клас, обаче, не знае за виртуалните функции, които производните му класове въвеждат. Например, hibernates() не може да се извика чрез указател или алтернативно име на обект от клас ZooAnimal, дори ако действителният тип на аргумента е от клас Bear. Следващият пример е доказателство за това:

```
int hibernates( ZooAnimal &za )
{
    return za.hibernates();           // грешка: hibernates не е член на ZooAnimal
}
```

Bear е основен клас за виртуалната функция hibernates(). Това означава, че само Bear и производните му класове имат достъп до тази функция. Ако hibernates() трябва да бъде активна за по-широко множество от класове, тя трябва да се придвижи напред в йерархията (в нашия случай да стане член на ZooAnimal), или цялата йерархия да се направи отново. При всички случаи hibernates() трябва да бъде достъпна за всички класове в йерархията, за които зимният сън е обща черта.

В таблица 8.1 е даден списък на активните виртуални функции за клас Panda. Втората колона от таблицата съдържа класовете, в които се намира активната дефиниция на

съответната функция. Третата колона съдържа класовете, в които виртуалните функции са дефинирани за първи път. Създаването на добра йерархия е трудна работа. Обикновено успех се постига при неколккратно коригиране на първоначалния вариант. Създаването на йерархични системи е област, в която се търсят непрекъснато нови решения, но все още не може да се каже, че съществуват определени правила.

Виртуални функции	Активна дефиниция	Първа дефиниция
isOnDisplay()	ZooAnimal	ZooAnimal
locate()	Panda	ZooAnimal
draw()	Panda	ZooAnimal
debug()	Panda	ZooAnimal
feedingHours()	Bear	Bear
hibernates()	Bear	Bear
adjustPopulation(int)	Endangered	Endangered
highlight(short)	Panda	Endangered/Herbivore
inform(ZooAnimal&)	Panda	Herbivore
onLoan()	Panda	Panda

Табл. 8.1 Виртуални функции за клас **Panda**

Упражнение 8-1. В дефиницията на клас Bear има виртуална функция, чието място не е в този клас. Коя е тази виртуална функция? Къде би трябвало да се постави тя? Защо?

Упражнение 8-2. В раздел 7.4 дадохме работни дефиниции на ZooAnimal, Bear и Panda, които илюстрират механизма на наследяване и не представляват пример за добра наследствена йерархия. Дефинирайте отново тези класове, като включите нужните виртуални функции, конструктори от вида X(const X&) и операции operator=(const X&).

Упражнение 8-3. Променете дефинициите на класовете от йерархията на геометричните фигури, за да дефинирате описаната в раздел 7.4 функция debug() като виртуална.

Упражнение 8-4. Дефинирайте отново функцията debug() от упражнение 7-19 (раздел 7.7), като използвате резултата от упражнение 8-3.

Упражнение 8-5. Дефинирайте виртуална функция draw() за йерархията на геометричните фигури.

Упражнение 8-6. Дефинирайте виртуална функция reSize() за йерархията на геометричните фигури.

Виртуални деструктори

Свързаният списък от животни, който се предава на функцията finalCollage(), не е необходим след нейното изпълнение. Затова в края на функцията се добавя цикъл for за освобождаване на заетата от този списък памет:

```
for ( ZooAnimal *p = pz—>next; p; pz = p, p = p—>next )
    delete pz;
```

За нещастие, този начин за освобождаване на памет в случая не върши работа. Явното извикване на delete за указателя pz ще предизвика обръщение към деструктора на клас ZooAnimal. Но всеки обект от списъка принадлежи на произведен клас на ZooAnimal, например Bear. Това означава, че трябва да се изпълни деструкторът на съответния клас, а не деструкторът на клас ZooAnimal. Ако се използва явно обръщение към действителния деструктор, трябва да се познават детайлите в йерархията на животните.

```
for ( ZooAnimal *p = pz—>next; p; pz = p, p = p—>next )
    switch ( pz—>isA() )
    {
        case BEAR:
            // пряко обръщение към съответния деструктор
            ((Bear*) pz)—> Bear:: ~Bear(); break;
        case PANDA:
            // непряко обръщение към съответния деструктор
            delete (Panda *) pz; break;
        // ... останалите случаи
    }
```

Макар че деструкторите не носят еднакви имена, те могат да се декларират като виртуални. Деструкторът на произведен клас е виртуален, ако деструкторът на негов основен клас е виртуален. Ако ZooAnimal::~~ZooAnimal() се декларира като виртуален деструктор, деструкторите на всички класове от йерархията на животните ще бъдат също виртуални. Например:

```
class ZooAnimal {
public:
    virtual ~ZooAnimal();
    // ... останалата част от дефиницията
};

// ok: сега ще се извиква необходимият деструктор
for ( ZooAnimal *p = pz—>next; p; pz = p, p = p—>next )
    delete pz;
```

Като обявим за виртуални деструкторите в йерархията, ще използваме подходящия деструктор при прилагане на операцията delete към указател, който адресира обект от основен клас. Затова деструкторът на абстрактния суперклас трябва да бъде виртуален.

Упражнение 8-7. Дефинирайте деструкторите от йерархията на геометричните фигури като виртуални.

Извикване на виртуална функция

Виртуалните функции се извикват чрез указател или алтернативно име на обект от някакъв клас. Множеството от виртуални функции, които може да се извикат при всяко обръщение, включва следните функции:

- Функцията, дефинирана за класа, чрез който се извършва обръщението.
- Функциите, които са дефинирани в неговите производни класове.

Виртуалната функция, която се изпълнява, се определя от действителния тип на аргумента.

За да илюстрираме извикването на виртуални функции, ще използваме следната опростена дефиниция на клас `ZooAnimal`. `ZooAnimal` дефинира две виртуални функции, `print()` и `isA()`. Деструкторът на класа също е виртуален:

```
#include <stream.h>
enum ZooLocs { ZOOANIMAL, BEAR, PANDA };

class ZooAnimal {
public:
    ZooAnimal( char *s = "ZooAnimal" );
    virtual ~ZooAnimal() { delete name; }
    void link( ZooAnimal* );
    virtual void print( ostream& );
    virtual void isA( ostream& );
protected:
    char *name;
    ZooAnimal *next;
};

#include <string.h>
ZooAnimal::ZooAnimal( char *s ) : next( 0 ) {
    name = new char[ strlen(s) + 1 ];
    strcpy( name, s );
}
```

Ще построим списък от животни от зоологическата градина. Отделните елементи от списъка се свързват посредством члена `next`. Действителният тип на всеки елемент е без значение, защото механизмът на виртуалните функции определя автоматично този тип.

`ZooAnimal::link()` изгражда свързания списък. Тя изисква аргумент от тип `ZooAnimal*` и го присвоява на `next`. Ето нейната дефиниция:

```
void ZooAnimal::link( ZooAnimal *za ) {
    za->next = next;
    next = za;
}
```

`isA()` и `print()` са виртуални функции. Всеки производен клас ще дефинира свои функции със същите имена. `isA()` обявява името на класа. `print()` описва представянето на съответния клас. Всяка от тях изисква аргумент от тип `ostream&`. Следват техните дефиниции:

```
void ZooAnimal::isA( ostream& os )
{
    os << "Име на животно: " << name << "\n";
}

void ZooAnimal::print( ostream& os )
{
    isA(os); // извикване на виртуална функция
}
```

Нашата цел е да изведем информация за всеки клас от йерархията на животните. Затова трябва да дефинираме операцията за стандартен изход и за аргумент от тип `ZooAnimal&`:

```
#include <stream.h>

ostream& operator <<( ostream& os, ZooAnimal& za )
{
    za.print(os);
    return os;
}
```

Сега програмистът може да използва операцията за стандартен изход за всеки член от йерархията на животните. Механизмът на виртуалните функции се грижи за извикване на подходящата функция print(). Ще дефинираме класовете Bear и Panda и ще дадем един пример. Забележете, че дефинираната от нас операция "<<" не е необходимо да бъде приятелска за клас ZooAnimal, защото тя използва само общодостъпни членове на ZooAnimal. Сега ще дефинираме и клас Bear:

```
class Bear : public ZooAnimal {
public:
    Bear( char *s = "Мечка", ZooLocs loc = BEAR, char *sci = "Ursidae" );
    ~Bear() { delete sciName; }
    void print( ostream& );
    void isA( ostream& );
protected:
    char *sciName; // научно наименование
    ZooLocs zooArea;
};

#include <string.h>
Bear::Bear( char *s, ZooLocs loc, char *sci ) : ZooAnimal( s ), zooArea( loc )
{
    sciName = new char[ strlen(sci) + 1 ];
    strcpy( sciName, sci );
}
```

Bear притежава два допълнителни члена-данни:

- sciName е научното име на съответното животно.
- zooArea е района в зоологическата градина, където живее то.

Дефинициите на Bear::isA() и Bear::print() отразяват представянето на клас Bear:

```
void Bear::isA( ostream& )
{
    ZooAnimal::isA( os );           // статично обръщение
    os << "\tНаучно име:\t";
    os << sciName << "\n";
}

static char *locTable[] = {
    "Цялата площ на зоологическата градина",           // ZOOANIMAL
    "Северозапад: B1: Кафявата област",                 // BEAR
    "Северозапад: B1.P: Областта кафяво петно" // PANDA
    // ... и т.н.
};

void Bear::print( ostream& )
{
}
```

```

ZooAnimal::print( os ); // статично обръщение
os << "\tМестонахождение:\n\t";
os << locTable[ zooArea ] << "\n";
}

```

Съществуват три случая, при които обръщението към виртуална функция се решава статично по време на компилация:

1. Когато виртуалната функция се извиква чрез обект. В следващия програмен фрагмент виртуалната функция isA() се извиква чрез обект за от клас ZooAnimal и това обръщение е статично. Забележете, че извикването чрез обект може да стане и чрез указател, който адресира този обект. За целта към името на указателя се прилага операцията "*". В този случай, обаче, извикването на виртуалната функция не е статично:

```

#include "ZooAnimal.h"
main() {
    ZooAnimal za;
    ZooAnimal *pz;
    // ...
    za.isA( cout );           // статично обръщение
    (*pz). isA( cout );       // виртуално обръщение
}

```

2. Когато виртуалната функция се извиква чрез указател или алтернативно име на обект, но конкретната функция се посочва явно, посредством операцията "::". Например:

```

#include <stream.h>
#include "Bear.h"
#include "ZooAnimal.h"
main() {
    Bear yogi( "Малка мечка", BEAR, "ursus cartoonus" );
    ZooAnimal circus( "Циркови животни" );
    ZooAnimal *pz;

    pz = &circus;
    cout << "Виртуално обръщение: ZooAnimal::print():\n";
    pz->print( cout );

    pz = &yogi;
    cout << "\nВиртуално обръщение: Bear::print():\n";
    pz->print( cout );

    cout << "\nСтатично обръщение: ZooAnimal::print():\n";
    cout << "Забележете, че isA() се извиква виртуално\n";
    pz->ZooAnimal::print( cout );
}

```

След компилация и изпълнение на програмата ще получим:

```

Виртуално обръщение: ZooAnimal::print()
Име на животно: Циркови животни

Виртуално обръщение: Bear::print():

```

Име на животно: Малка мечка
Научно име: ursus cartoonus
Местонахождение:
Северозапад: B1: Кафявата област

Статично обръщение: ZooAnimal::print():
Забележете, че isA() се извиква виртуално
Име на животно: Малка мечка
Научно име: ursus cartoonus

3. Когато виртуалната функция се извиква в тялото на конструктор или деструктор на основен клас. Това е така, защото обектът от производния клас още не е създаден или вече е разрушен.

Дефиницията на клас Panda съдържа още два члена-данни: indName е името на конкретно животно, а cell е номерът на клетката, в която живее то. Следва дефиницията на Panda:

```
#include <stream.h>
class Panda : public Bear {
public:
    Panda( char *nm, int room, char *s = "Панда",
           char *sci = "Ailuropoda Melaoleuca",
           ZooLocs loc = PANDA );
    ~Panda() { delete indName; }
    void print( ostream& );
    void isA( ostream& );
protected:
    char *indName; // име на отделно животно
    int cell;
};

#include <string.h>
Panda::Panda( char *nm, int room, char *s, char *sci, ZooLocs loc )
    : Bear( s, loc, sci ), cell( room ) {
    indName = new char[ strlen(nm) + 1 ];
    strcpy( indName, nm );
}
```

Panda::isA() и Panda::print() отразяват представянето на клас Panda:

```
void Panda::isA( ostream& os ) {
    Bear::isA( os );
    os << "\tВикаме нашия приятел:\t";
    os << indName << "\n";
}

void Panda::print( ostream& os ) {
    Bear::print( os );
    os << "\tКлетка номер:\t";
    os << cell << "\n";
}
```

Сега ще покажем как се извикват виртуални функции. Първият пример показва виртуално обръщение към print() чрез алтернативно име на обект от клас ZooAnimal. Всеки

обект се предава на операцията "<<", която дефинирахме в този подраздел. В нейната дефиниция съществува обръщението:

```
za.print( os );
```

В зависимост от действителния тип на za се извиква съответната функция print().

```
#include <iostream.h>
#include "Bear.h"
#include "ZooAnimal.h"
#include "Panda.h"

ZooAnimal circus( "Циркови животни" );
Bear yogi( "Малка мечка", BEAR, "ursus cartoonus" );
Panda yinYang( "Yin Yang", 1001, "Голяма панда" );

main() {
    cout << "Извикване чрез обект от клас ZooAnimal:\n" << circus << "\n";
    cout << "\nИзвикване чрез обект от клас Bear:\n" << yogi << "\n";
    cout << "\nИзвикване чрез обект от клас Panda:\n" << yinYang << "\n";
}
```

След компилация и изпълнение на програмата получаваме:

Извикване чрез обект от клас ZooAnimal:

Име на животно: Циркови животни

Извикване чрез обект от клас Bear:

Име на животно: Малка мечка

Научно име: ursus cartoonus

Местонахождение:

Северозапад: B1: Кафявата област

Извикване чрез обект от клас Panda:

Име на животно: Голяма панда

Научно име: Ailuropoda Melaoleuca

Викаме нашия приятел: Yin Yang

Местонахождение:

Северозапад: B1.P: Областта кафяво петно

Клетка номер: 1001

Следващият пример показва използването на указатели към обекти. В него се извиква виртуалната функция isA().

```
#include <iostream.h>
#include "Bear.h"
#include "ZooAnimal.h"
#include "Panda.h"

ZooAnimal circus( "Циркови животни" );
Bear yogi( "Малка мечка", BEAR, "ursus cartoonus" );
Panda yinYang( "Yin Yang", 1001, "Голяма панда" );

main() {
    ZooAnimal *pz;
```

```

pz = &circus;
cout << "Виртуално обръщение: ZooAnimal::isA():\n";
pz->isA( cout );

pz = &yogi;
cout << "\nВиртуално обръщение: Bear::isA():\n";
pz->isA( cout );

pz = &yinYang;
cout << "\nВиртуално обръщение: Panda::isA():\n";
pz->isA( cout );
}

```

След изпълнение и компилация на тази програма получаваме следния резултат:

```

Виртуално обръщение: ZooAnimal::isA():
Име на животно: Циркови животни
Виртуално обръщение: Bear::isA():
Име на животно: Малка мечка
    Научно име:                ursus cartoonus

Виртуално обръщение: Panda::isA():
Име на животно: Голяма панда
    Научно име:                Ailuropoda Melaoleuca
    Викаме нашия приятел:      Yin Yang

```

В следващия пример действителният тип на обекта, адресиран от указателя pz, е известен още при компилация. Затова съответните функции се извикват статично:

```

#include <iostream.h>
#include "Bear.h"
#include "ZooAnimal.h"
#include "Panda.h"

ZooAnimal circus( "Циркови животни" );
Bear yogi( "Малка мечка", BEAR, "ursus cartoonus" );
Panda yinYang( "Yin Yang", 1001, "Голяма панда" );

main() {
    ZooAnimal *pz = &yinYang;
    cout << "Статично обръщение към Panda::isA():\n";
    ( (Panda*) pz )->Panda::isA( cout );

    pz = &yogi;
    cout << "\nСтатично обръщение към Bear::isA():\n";
    ( (Bear*) pz )->Bear::isA( cout );
}

```

Статично обръщение към Panda::isA() чрез указател от клас ZooAnimal е възможно само чрез явно преобразуване. След изпълнение на програмата получаваме следния печат:

```

Статично обръщение към Panda::isA():
Име на животно: Голяма панда
    Научно име:                Ailuropoda Melaoleuca
    Викаме нашия приятел:      Yin Yang

```


Статично обръщение към Bear::isA():
Име на животно: Малка мечка
Научно име: ursus cartoonus

Последният пример отпечатва списъка от животни. За целта се дефинира още една функция print(), която не е член на клас. Тя осъществява достъп до скрития член next на клас ZooAnimal и затова трябва да бъде приятелска за този клас.

```
#include <iostream.h>
#include "ZooAnimal.h"

void print( ZooAnimal *pz, ostream &os = cout )
{
    while ( pz ) {
        pz->print( os );
        os << "\n";
        pz = pz->next;
    }
}
```

Нуждаем се от един указател от тип ZooAnimal*, който ще сочи началото на списъка:

```
ZooAnimal *headPtr = 0;
```

Тогава програмата ще изглежда така:

```
#include <stream.h>
#include "ZooAnimal.h"

extern ZooAnimal *makeList( ZooAnimal* );
ZooAnimal *headPtr = 0;

main() {
    cout << "Програмата илюстрира виртуалните функции.\n";
    headPtr = makeList( headPtr );
    print( headPtr );
}
```

Ще използваме следната дефиниция на makeList():

```
#include <stream.h>
#include "Bear.h"
#include "ZooAnimal.h"
#include "Panda.h"

ZooAnimal circus( "Циркови животни" );
Bear yogi( "Малка мечка", BEAR, "ursus cartoonus" );
Panda yinYang( "Yin Yang", 1001, "Голяма панда" );
Panda rocky( "Rocky", 943, "Червена панда", "Ailurus fulgens" );

ZooAnimal *makeList( ZooAnimal *ptr )
{
    // за простота списъкът се създава "ръчно"
```

```

ptr = &yinYang;
ptr->link( &circus );
ptr->link( &yogi );
ptr->link( &rocky );
return ptr;
}

```

След компилация и изпълнение на тази програма получаваме следния печат:

Програмата илюстрира виртуалните функции.

Име на животно: Голяма панда

Научно име: Ailuropoda Melaoleuca

Викаме нашия приятел: Yin Yang

Местонахождение:

Северозапад: B1.P: Областта кафяво петно

Клетка номер: 1001

Име на животно: Червена панда

Научно име: Ailurus fulgens

Викаме нашия приятел: Rocky

Местонахождение:

Северозапад: B1.P: Областта кафяво петно

Клетка номер: 943

Име на животно: Малка мечка

Научно име: ursus cartoonus

Местонахождение:

Северозапад: B1: Кафявата област

Име на животно: Циркови животни

С изключение на makeList() програмата не се интересува от детайлите в реализацията на класовете в йерархията. Макар и прост, примерът отразява стила на обектно-ориентираното програмиране.

Упражнение 8-8. Създайте функция draw(), която не е член на клас и има един аргумент от тип Shape*. Начертайте окръжност, правоъгълен триъгълник и правоъгълник.

Упражнение 8-9. Дефинирайте функция reSize(), която не е член на клас и има аргумент от тип Shape& (необходим е и втори аргумент за размера). Изпълнете draw(), reSize() и отново draw() за окръжност, равностранен триъгълник и квадрат.

Упражнение 8-10. Създайте виртуална функция save(), която описва обект от йерархията на геометричните фигури на екрана и виртуална функция restore(), която прочита информацията, изведена от save().

8.3 Виртуални основни класове

Макар че всеки клас може да присъства само веднъж в даден списък на основни класове, той може да се появи в няколко такива списъка. Това може да стане причина за възникване на противоречия. Например, вече повече от 100 години се спори дали вид Panda е от семейството на мечките (Bear) или от семейството на миещите мечки (Raccoon). От гледна точка на информатиката най-доброто решение е Panda да бъде произведен клас и на двата класа:

```
class Panda : public Bear, public Raccoon { ... }
```

Panda наследява основния клас ZooAnimal от два класа — от Bear и Raccoon. Следователно, всеки обект от клас Panda има две части, които съответстват на основния клас ZooAnimal. При деклариране на обект от клас Panda се извикват два конструктора на клас ZooAnimal в следния ред:

```
ZooAnimal();    // ZooAnimal е основен клас за Bear
Bear();         // първият основен клас за Panda
ZooAnimal();    // ZooAnimal е основен клас за Raccoon
Raccoon();      // вторият основен клас за Panda
Panda();        // конструкторът на производния клас е последен
```

За илюстрация ще въведем следната дефиниция на клас ZooAnimal:

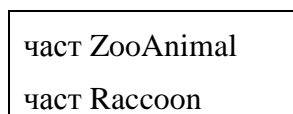
```
class ZooAnimal {          // опростена дефиниция
public:
    void locate();
protected:
    short zooArea;
};

class Bear : public ZooAnimal    { /* ... */ }
class Raccoon : public ZooAnimal { /* ... */ }
```

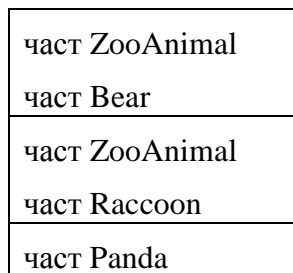
Panda наследява два пъти ZooAnimal::zooArea — от Bear и от Raccoon. Фигура 8.1 показва структурата на обектите от клас Panda.



class Bear: public ZooAnimal



class Raccoon: public ZooAnimal



class Panda: public Bear, public Raccoon

Фиг. 8.1 Многократно включване на основен клас

В клас Panda съществуват две копия на zooArea. Затова този член не е достъпен директно в клас Panda. Чрез операцията "::" може да се посочи едно от двете копия.

Забележете, че записът `ZooAnimal::zooArea` не е по-малко противоречив от простия запис `zooArea`. Трябва да се посочи един от двата члена `ZooAnimal::zooArea` — `Bear::zooArea` или `Raccoon::zooArea`:

```
Panda::getArea() {  
    // BEAR е константа  
    return ( isA() == BEAR ) ? Bear::zooArea : Raccoon::zooArea ;  
}
```

Забележете също, че клас `Panda` притежава само едно копие на `ZooAnimal::locate()`. `locate()` и `ZooAnimal::locate()` извикват единственото копие на наследената функция, но въпреки това съществува противоречие. Защо?

`Panda` съдържа два обекта от основния клас `ZooAnimal`. Кой от тях трябва да се свърже с извикваната функция? (В раздел 5.4 се разглежда свързването на функция-член на клас с извикващия я обект чрез указателя `this`.) Грижата да посочи един от двата обекта се оставя на програмиста. Отново може да се използва операцията `::`. Например:

```
locate( Panda *p )  
{    // BEAR и RACCOON са константи  
    switch ( p—>isA() ) {  
        case BEAR:      p—>Bear::locate(); break;  
        case RACCOON:    p—>Raccoon::locate(); break;  
    }  
}
```

В някои специални приложения такова многократно включване на основен клас може да е необходимо, но в нашия пример не е така. За нас се появява излишна противоречивост. За да се предпазим от нея, трябва да следим много неща. Освен това второто копие на базовия клас заема излишна памет.

`Panda` се нуждае само от едно копие на базовия клас `ZooAnimal`. Тогава йерархията може да се представи като граф без цикли, където `ZooAnimal` е поделен основен клас. Стандартният механизъм за наследяване се представя като дърво, където всяка поява на клас в списък на основните класове генерира копие на този клас. Двете ситуации са представени на фигура 8.2.

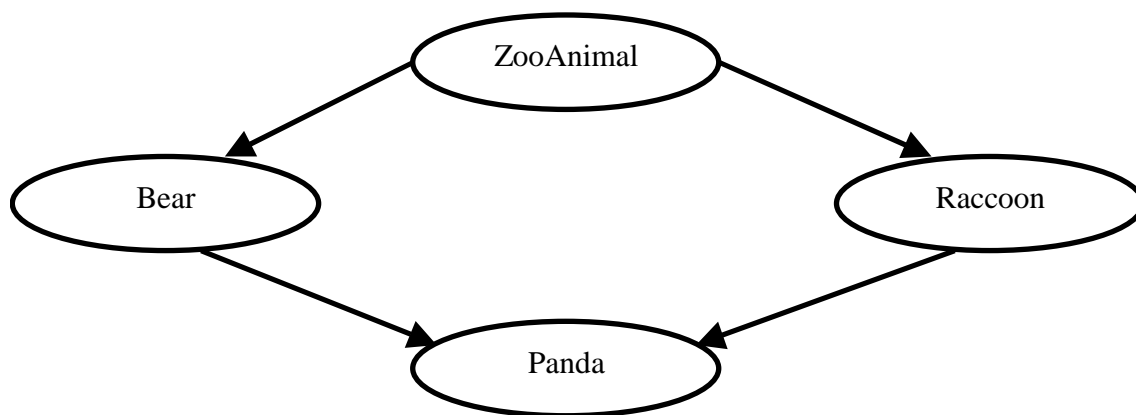
Трябва да се намери начин за отменяне на стандартния механизъм за наследяване. Целта е да се позволи дефиниране на поделени основни класове. Виртуалните класове са решение на поставения въпрос. Когато един клас е виртуален, независимо от участието му в няколко списъка на основни класове, се генерира само едно негово копие. В нашия случай `Panda` ще съдържа един поделен основен клас `ZooAnimal`. Обръщението към членовете на този клас не е противоречиво.

Дефиниране на виртуални основни класове

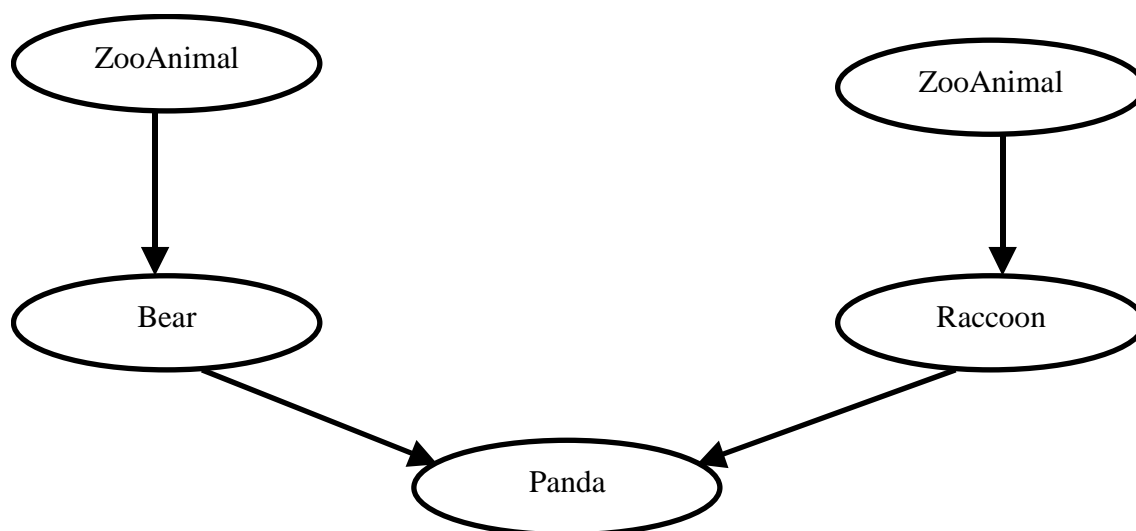
Един основен клас е виртуален, ако в неговата декларация присъства ключовата дума `virtual`. В следващия пример `ZooAnimal` се декларира като виртуален основен клас за `Bear` и `Raccoon`:

```
// редът на думите public и virtual е без значение  
class Bear : public virtual ZooAnimal    { /* ... */ }  
class Raccoon : virtual public ZooAnimal { /* ... */ }
```

Ако един виртуален клас притежава конструктори, той трябва да има и конструктор без аргументи или конструктор, за който всички аргументи имат стойности по подразбиране.⁸



Ацикличен граф с поделен основен клас



Стандартен механизъм за наследяване

Фиг. 8.2 Схеми на двата механизма за многократно наследяване

Ако ZooAnimal е виртуален клас, неговата дефиниция не се нуждае от промени. Ще използваме следната работна дефиниция на ZooAnimal:

```
class ZooAnimal { // опростена дефиниция
public:
    ZooAnimal() { zooArea = 0; name = 0; }
    ZooAnimal( char*, short );
```

⁸ Според сегашните правила на езика, ако се изисква конструктор по подразбиране, но такъв не съществува, компилаторът извежда съобщение за грешка. В по-ранни версии се прави опит за избягване на такива грешки.

```

    void locate();
protected:
    short zooArea;
    char *name;
};

```

Обектите от класовете Bear и Raccoon се използват по същия начин. Виртуалният основен клас се инициализира като "обикновен" основен клас:

```

Bear::Bear( char *nm ) : ZooAnimal( nm, BEAR ) { ... }
Raccoon::Raccoon( char *nm ) : ZooAnimal( nm, RACCOON ) { ... }

```

Декларацията на клас Panda също остава непроменена:

```

class Panda : public Bear, public Raccoon { ... };

```

Тъй като Bear и Raccoon декларират своя основен клас като виртуален, всеки обект от клас Panda съдържа само едно копие от членовете на ZooAnimal.

Обикновено производните класове могат да инициализират явно само базовите класове, от които непосредствено произлизат. Това означава, че името ZooAnimal не може да се включи в списъка за инициализация на конструктор на клас Panda. Виртуалните класове, обаче, правят изключение. Това е така, защото Panda съдържа едно копие от членовете на клас ZooAnimal, поделено между класовете Bear и Raccoon. Bear и Raccoon инициализират явно ZooAnimal, но единственото копие в Panda не може да се инициализира два пъти.

Виртуалните основни класове се инициализират от най-низкия в йерархията произведен клас. В нашия случай такъв клас е Panda. Затова той може да инициализира явно ZooAnimal. Ако ZooAnimal не се инициализира явно, ще се използва неговият конструктор по подразбиране. Bear и Raccoon няма да инициализират единственото копие на ZooAnimal в обектите от клас Panda. Ето една примерна дефиниция на конструктора на клас Panda:

```

Panda::Panda( char *nm )
    : ZooAnimal( nm, PANDA ),
    Bear( nm ), Raccoon( nm ) { ... }

```

Следващата дефиниция на същия конструктор инициализира ZooAnimal чрез неговия конструктор по подразбиране:

```

Panda::Panda( char *nm )
    : Bear( nm ), Raccoon( nm ) { ... }

```

Достъп до членове на виртуален основен клас

Обектите от класовете Bear и Raccoon притежават свое множество от наследени членове от клас ZooAnimal. Тези членове са достъпни по същия начин, както са достъпни наследените членове от "обикновен" основен клас. Разлика в използването на обектите няма. Разликата е в заделянето на памет за частта на основния клас.

В йерархия, в която няма виртуални основни класове, всеки производен клас съдържа непрекъснатата "основна" и "производна" част (разгледайте фигура 7.4 от раздел 7.7). В йерархия с виртуални основни класове всеки производен клас съдържа "производна" част и указател към частта на виртуалния базов клас. На фигура 8.3 е дадена структурата на класовете Bear, Raccoon и Panda, когато ZooAnimal е виртуален основен клас.

Достъпът до членовете, наследени от виртуален основен клас, се подчинява на известните правила за достъп до наследени членове:

- Членовете, които са наследени от виртуален основен клас с атрибут public, запазват своето ниво на достъп в производния клас.
- Членовете, които са наследени от виртуален основен клас с атрибут private, стават скрити (private) членове на производния клас.

Какво ще се получи, ако Panda включи основния клас ZooAnimal веднъж с атрибут public, а втори път с атрибут private? Например:

```
class ZooAnimal {
public:
    void locate();
protected:
    short zooArea;
};

class Bear : public virtual ZooAnimal    { /* ... */ }
class Raccoon : private virtual ZooAnimal { /* ... */ }
class Panda : public Bear, public Raccoon { /* ... */ }
```

Тъй като ZooAnimal е виртуален основен клас, Panda наследява само едно копие на zooArea и locate(). Въпросът е дали Panda има достъп до тези два члена. Ако наследяването е по линия на клас Bear, Panda ще има достъп до тях. Ако наследяването е по линия на клас Raccoon, Panda няма да има достъп. Кое от двете наследявания има приоритет? Правилно ли е да се запише

```
Panda p;
p.locate();
```

Отговорът на последния въпрос е положителен. Наследяването по линия на основния клас с атрибут public има приоритет. Това означава, че Panda има достъп до zooArea и locate(). В общия случай, ако сред срещанията на виртуален клас в дадена йерархия има срещане с атрибут public, поделеният основен клас също се счита за public.

ZooAnimal притежава член locate() и той се наследява от Raccoon. Тогава обръщението

```
Raccoon r;
r.locate(); // наследената от ZooAnimal функция
```

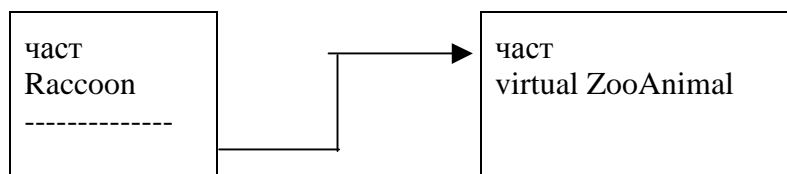
извиква ZooAnimal::locate(). Представете си, че Bear дефинира своя функция locate(). Обръщението

```
Bear b;
b.locate(); // извиква Bear::locate()
```

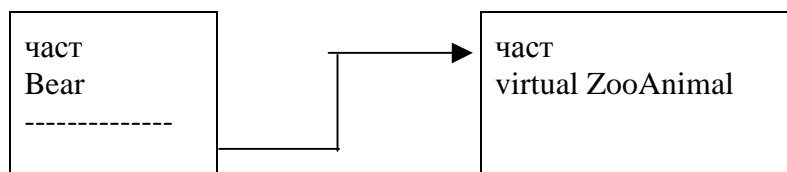
извиква `Bear::locate()`, а не `ZooAnimal::locate()`. В йерархия без виртуални класове обръщението

```
Panda p;
p.locate();    // за коя функция locate() се отнася?
```

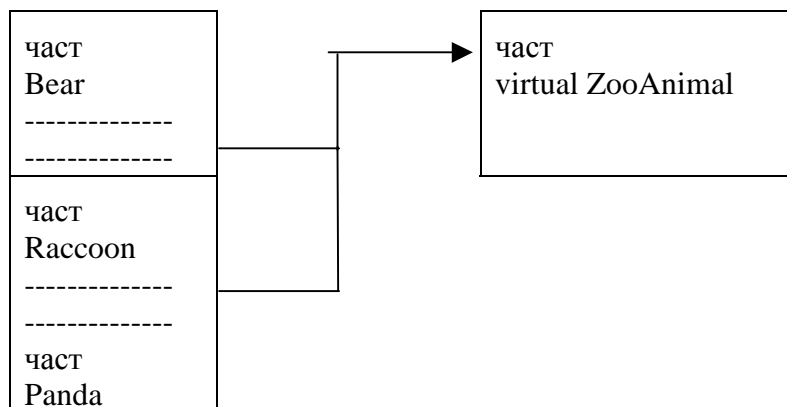
е противоречиво, ако `Panda` не притежава своя функция `locate()`. Но ако `ZooAnimal` е виртуален основен клас за `Bear` и `Raccoon`, ще се извика `Bear::locate()`. Противоречие няма да има. В йерархия с виртуални класове доминира функцията, която е член на най-низкия в йерархията произведен клас. В нашия случай `Bear::locate()` доминира над наследения от `Raccoon` член `ZooAnimal::locate()`.



```
class Raccoon: public virtual ZooAnimal
```



```
class Bear: public virtual ZooAnimal
```



```
class Panda: public virtual ZooAnimal
```

Фиг. 8.3 Представяне на класове с виртуален основен клас

Ред на извикване на конструкторите и деструкторите

Виртуалните основни класове се инициализират преди останалите основни класове, независимо от тяхното място в списъка на основните класове. Ако даден клас произхожда от два основни класа — единият е виртуален, а другият не — първо се извиква конструкторът

на виртуалния клас. Аналогично, деструкторът на виртуалния клас се извиква последен. Например:

```
class TeddyBear : public Bear, public virtual ToyAnimal { ... }  
TeddyBear pooh;
```

pooh се инициализира от конструкторите, извикани в този ред:

```
ToyAnimal();    // виртуален основен клас  
ZooAnimal();    // основен клас за Bear  
Bear();         // основен клас, който не е виртуален  
TeddyBear();
```

Редът на извикване на деструкторите е обратен.

Ако класът произлиза от няколко виртуални класа, техните конструктори се извикват по реда на декларациите на тези класове в списъка на основните класове.

По-сложен е случаят, когато виртуалният клас е вложен в йерархията. Например, Panda наследява от Bear и Raccoon поделения основен клас ZooAnimal. Откриването на виртуален основен клас може да изисква обхождане на целия наследствен граф. Например, дефиницията на Panda не предполага присъствие на виртуален основен клас:

```
class Panda : public Endangered, public Herbivore(),  
              public Raccoon, public Bear { ... };  
  
Panda yinYang;
```

yinYang се инициализира от конструкторите, викани в този ред:

```
ZooAnimal();    // виртуален основен клас  
// по реда на декларациите на основните класове  
Endangered();  
Herbivore();  
Raccoon();  
Bear();  
Panda();
```

Йерархията се претърсва за виртуални основни класове по реда на декларациите на основните класове. За клас Panda претърсването става в следния ред:

- Endangered
- Herbivore
- Raccoon
- Bear

Ако предположим, че Endangered съдържа виртуален основен клас, редът на обръщение към конструкторите на виртуалните основни класове следва реда на претърсване на йерархията. Например, конструкторът на виртуалния основен клас, който е свързан с Endangered, ще се извика преди конструктора на ZooAnimal, който е свързан с Raccoon и Bear. Деструкторите се извикват в обратен ред.

Смесване на виртуални и "обикновени" класове

Когато в йерархията един клас се използва едновременно като виртуален и "обикновен" основен клас, се създава по един обект за всеки обикновен случай и един общ обект за всички виртуални случаи. Например, ако в дефиницията на `Endangered ZooAnimal` е "обикновен" основен клас, `Panda` ще съдържа две копия на `ZooAnimal` — едното, наследено от `Bear` и `Raccoon`, а другото, наследено от `Endangered`. Отново се получават две (или повече) копия от членовете на `ZooAnimal`. Противоречията могат да се разрешат чрез операцията "::".

Резюме

По подразбиране наследственият механизъм дефинира йерархия, която се представя чрез дърво. В този случай всяко срещане на основен клас създава копие на неговите членове. Виртуалните основни класове са начин за отменяне на стандартния наследствен механизъм. Независимо от броя на срещанията на виртуален основен клас, в йерархията се създава само едно негово копие. Виртуалният клас може да се разглежда като поделен клас. В нашия пример `ZooAnimal` е виртуален основен клас за `Bear` и `Raccoon`. `Panda` произлиза от `Bear` и `Raccoon`, но съдържа само едно копие на `ZooAnimal`.

Приложение А

Входно-изходна библиотека в C++

Входно-изходните операции в C++ се реализират от стандартна библиотека. Тя се нарича `iostream` и се разпространява със стандарт 2.0 на езика C++. `iostream` замества по-ранната версия на същата библиотека, известна като библиотека `stream`. Програми, които използват по-ранната версия на библиотеката обикновено са съвместими с новата версия.

На ниско ниво файловете се интерпретират като последователност или поток (`stream`) от байтове. `iostream` управлява предаването на тези байтове. На потребителско ниво файловете съдържат различни типове данни — символи, числови данни и обекти от различни класове. `iostream` управлява и взаимодействието между тези две нива.

Библиотеката `iostream` съдържа операции за четене и запис на стандартните типове данни. Програмистът може да дефинира част от тези операции за потребителски типове. В това приложение ще разглеждаме тези два компонента на входно-изходната библиотека.

Входно-изходните операции се осъществяват от два класа — `istream` и `ostream`. Клас `istream` е производен на тези два класа. Чрез него може да се извършва както въвеждане, така и извеждане на информация. Операцията за изход "<<" добавя стойност към изходния поток. Операцията за вход ">>" извлича стойност от входния поток. Може да се смята, че тези две операции "сочат" посоката на движение на данните. Например,

```
>> x
```

поставя стойност в `x`, докато

```
<< x
```

извлича данните от `x`.

Потребителят може да използва следните четири обекта:

1. `cin`, обект от клас `istream`, свързан със стандартния вход.
2. `cout`, обект от клас `ostream`, свързан със стандартния изход.
3. `cerr`, обект от клас `ostream`, свързан със стандартния изход за грешки. Той се използва за небуфериран изход.
4. `clog`, обект от клас `ostream`, свързан със стандартния изход за грешки. Той се използва за буфериран изход.

Всяка програма, която използва библиотеката `iostream`, трябва да включва заглавния файл `iostream.h`. Програмите, които използват библиотеката `stream`, не е необходимо да се променят. Директивата, която включва заглавния файл `stream.h`, се възприема като алтернатива на директивата за заглавния файл `iostream.h`.

Библиотеката `iostream` поддържа и операции с файлове. Даден файл може да се свърже с програма чрез дефиниране на обект от един от следните три класа:

1. `ifstream`, производен на клас `istream`, който дефинира файлове за четене.
2. `ofstream`, производен на клас `ostream`, който дефинира файлове за запис.

3. `fstream`, произведен на клас `iostream`, който дефинира файлове за четене и запис.

Библиотеката `iostream` поддържа също операции за работа с масиви от символи. Използват се два класа:

1. `istream`, произведен на клас `istream`, чрез който се извличат символи от масив от символи.
2. `ostream`, произведен на клас `ostream`, чрез който се запазват символи в масив от символи.

Операциите с файлове и масиви от символи се разглеждат по-нататък в това приложение.

А.1 Извеждане на информация

Стандартен начин за извеждане на информация е прилагане на операцията `<<` към `cout`. Например,

```
#include <iostream.h>

main() {
    cout << "I am an output string\n";
}
```

извежда на потребителския терминал

```
I am an output string
```

Операцията `<<` е дефинирана за всички стандартни типове данни, включително и за тип `char*`. По-нататък в това приложение ще разгледаме как се дефинира тази операция за класове.

Всеки сложен израз може да се отпечата с помощта на няколко операции за стандартен изход, като се раздели на подизрази с определен тип на резултата. Например,

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

main() {
    cout << "Дължината на низа 'Одисей' е:\t";
    cout << strlen( "Одисей" );
    cout << "\n";

    cout << "Размерът в байтове на низа 'Одисей' е:\t";
    cout << sizeof( "Одисей" );
    cout << "\n";

    return 0;
}
```

ще изведе на потребителския терминал съобщенията:

```
Дължината на низа 'Одисей' е:      6
Размерът в байтове на низа 'Одисей' е:  7
```

Няколко операции за стандартен изход могат да се обединят в един оператор. Например, последната програма може да изглежда така:

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

main()
{ // няколко операции "<<" могат да се обединят в един оператор
  cout << "Дължината на низа 'Одисей' е:\t"
        << strlen( "Одисей" ) << "\n"
        << "Размерът в байтове на низа 'Одисей' е:\t"
        << sizeof( "Одисей" ) << "\n";
  return 0;
}
```

Операцията за стандартен изход е дефинирана и за указатели. По този начин се извеждат адреси на обекти. По подразбиране адресите се извеждат като шестнадесетични числа. Например,

```
#include <iostream.h>

main() {
  int i = 1024;
  int *pi = &i;

  cout << "i: " << i << "\t"
        << "&i:\t" << &i << "\n";

  cout << "*pi: " << *pi << "\t"
        << "pi:\t" << pi << "\n\t\t"
        << "&pi:\t" << &pi << "\n";
  return 0;
}
```

извежда на потребителския терминал следното:

```
i: 1024      &i: 0x7ffff0b4
*pi: 1024    pi: 0x7ffff0b4
             &pi: 0x7ffff0b0
```

Извеждането на адреси в десетичен формат изисква явно преобразуване на аргумента в тип long. Например, последната програма може да се модифицира така:

```
<< "&i:\t" << long( &i ) << "\n";
<< "pi:\t" << long( pi ) << "\n\t\t"
<< "&pi:\t" << (long) &pi << "\n";
```

Модифицираната програма отпечатва следното:

```
i: 1024      &i: 2147479732
*pi: 1024    pi: 2147479732
```

Следващата програма трябва да отпечата адреса, който се съдържа в pstr:

```
#include <iostream.h>

char str[] = "vermeer";

main() {
    char *pstr = str;
    cout << "Адресът на pstr е: "
        << pstr << "\n";
    return 0;
}
```

След изпълнение на програмата неочаквано получаваме следния печат:

Адресът на pstr е: vermeer

Проблемът в случая е в това, че тип char* се интерпретира като низ от символи, а не като адрес на този низ. За да отпечатаме адреса, който се съдържа в pstr, трябва да преобразуваме явно pstr в тип void*:

```
<< (void*) pstr << "\n";
```

Тогава след компилация и изпълнение на програмата ще получим:

Адресът на pstr е: 0x116e8

Програмите са по-прегледни, когато се използват мнемонични имена за литералните символни константи. По-долу са дадени символни дефиниции на често използваните литерални константи. Те са включени в заглавния файл ioconst.h.

```
#ifndef IOCONST_H
#define IOCONST_H

const char *const sep =          " ";
const char *const nltab =       "\n\t";
const char *const prefix1 =     "[ ";
const char *const suffix1 =     " ]";
const char *const prefix2 =     "{ ";
const char *const suffix2 =     " }";

const char bell =                '\007';
const char nl =                  '\n';
const char tab =                  '\t';
const char sp =                   ' ';
const char comma =                ',';
const char lbrace =               '<';
const char rbrace =               '>';
```

```
#endif
```

За да се използват тези символни константи, трябва да се включи заглавния файл `ioconst.h`. Например:

```
#include <iostream.h>
#include "ioconst.h"

main() {
    cout << bell
        << "I am an output string" << nl;

    return 0;
}
```

Функцията `put()` е член на клас `ostream`. Тя осигурява друг начин за отпечатване на символи. Например,

```
// изпраща nl на стандартен изход
cout.put( nl );
```

отпечатва символа за нов ред.

Функцията `put()` има един аргумент от тип `unsigned char` или `char`. Тя връща обекта от клас `ostream`, чрез който се извиква. Следователно, няколко функции `put()` могат да се извикат последователно в един оператор:

```
cout.put( bell ).put( bell ).put( nl );
```

Функцията `write()` също е член на клас `ostream`. Тя осигурява друг начин за отпечатване на символни низове. Нейната сигнатура изглежда така:

```
write( const char *str, int length )
```

`length` е броя на символите, които се извеждат, като се започне от първия символ на низа. `write()` също връща обекта от клас `ostream`, чрез който се извиква. Това означава, че няколко функции `write()` могат да се извикат последователно в един оператор:

```
#include <iostream.h>
#include "ioconst.h"
#include <string.h>

inline void putString( char *s ) {
    cout.put(tab).write(s, strlen( s )).put(nl);
}

main() {
    putString( "Statly, plump Buck Mulligan" );
    putString( "Mr.Leopold Bloom" );
    return 0;
}
```

Програмата извежда следния печат:

Stately, plump Buck Mulligan
Mr. Leopold Bloom

Функцията `write()` може да се използва още за печат на несимволни данни в двоичен вид, т.е. като последователност от байтове. Следващата програма отпечатва цели числа като последователност от символи:

```
#include <iostream.h>
#include "ioconst.h"

const char byteSize = 8;
int ival = 'abcd';

main() {
    for ( int i = 0; i < sizeof(int); ++i ) {
        cout << "Стойността на ival като цяло число: "
              << ival << nl << "ival в двоичен вид: ";

        // забележете преобразуването на адреса в тип char*
        cout.write( (char*)&ival, sizeof(int) );

        // преместване на числото с един байт
        ival <<= byteSize;
        cout.put(nl).put(nl);
    }
    return 0;
}
```

След компилиране и изпълнение на програмата получаваме следния резултат:

Стойността на ival като цяло число: 1633837984
ival в двоичен вид: dcba

Стойността на ival като цяло число: 1650680832
ival в двоичен вид: dcb

Стойността на ival като цяло число: 1667497984
ival в двоичен вид: dc

Стойността на ival като цяло число: 1677721600
ival в двоичен вид: d

`cout` се използва за буфериран изход. Стойностите, които се извеждат, преди да се отпечата, се натрупват в буфер. При отпечатване на множество стойности буферирането е по-ефективно от отделното отпечатване на всяка стойност. Но когато отпечатването трябва да се извърши веднага, буферирането създава проблеми. Такава ситуация възниква при:

1. Интерактивни програми, в които се извършва непрекъснато въвеждане и извеждане на информация.
2. Програми, които проследяват изпълнението на други програми.

Информацията може да се отпечата незабавно, ако се предизвика "изпразване" на буфера. Буферът може да се изпразни, ако към изходния поток се добави специалната стойност `flush`:

```
cout << "Hi. What is your name?\n" << flush;
```

`flush` предизвиква извеждане на всички стойности, които се намират в буфера.

`flush` се нарича манипулатор (управляващ символ), защото неговото въвеждане предизвиква определено действие за изходния поток. Много полезен управляващ символ е `endl` (край на ред). Той добавя нов ред към изходния поток и изпразва буфера. Неговото действие може да се изрази със следния оператор:

```
cout << j << "\n" << flush;
```

което е еквивалентно на оператора:

```
cout << j << endl;
```

Следващата програма трябва да отпечата по-голямото от две числа:

```
#include <iostream.h>
#include "ioconst.h"

char *str1 = "От числата ";
char *str2 = "по-голямото е ";

main() {
    int i = 10, j = 20;
    cout << str1 << i << " и " << j;
    cout << str2 << ( i > j ) ? i : j << nl;
    return 0;
}
```

След компилиране и изпълнение на програмата се получава неверен резултат:

От числата 10 и 20 по-голямото е 0

Причина за този резултат е по-високият приоритет на операцията "`<<`" в сравнение с операцията аритметичен `if`. Операцията за стандартен изход има приоритета на побитовата операция изместване наляво. За да получим верен резултат, трябва да променим реда на изпълнение на двете операции като използваме скоби. Целият аритметичен `if` трябва да се огради в скоби, защото иначе се извежда стойността на израза

`(i > j)`

а тя в случая е лъжа, т.е. 0. Ето как трябва да се запише аритметичният `if`, за да се получи верен резултат:

```
<< ( i > j ? i : j ) << nl;
```

Сега коригираната програма работи правилно:

От числата 10 и 20 по-голямото е 20

Стойността на един израз се извежда правилно, ако целият израз се огради в скоби. Тази бележка се отнася и за изрази, в които участва операцията за присвояване:

```
cout << ( i += j );
```

А.2 Дефиниране на операцията "<<" за класове

В езика C++ операцията за стандартен изход не може да се използва за дефинирани от потребителя типове, ако програмистът не е дефинирал тази операция за съответния клас. Например,

```
#include <iostream.h>
#include "ioconst.h"

class WordCount {
    friend ostream& operator<<(ostream&, WordCount&);
public:
    WordCount( char*, int = 1 );
    // ...
private:
    char *str;
    int occurs;
};

ostream& operator<<( ostream& os, WordCount& wd ) {
    os << lbrace << wd.occurs
        << rbrace << sp << wd.str << endl;
    return os;
}
```

Дефинираната операция за стандартен изход на обекти от клас WordCount може да се използва заедно с операциите за стандартните типове данни. Например,

```
#include <iostream.h>
#include "ioconst.h"
#include "WordCount.h"

main() {
    WordCount wd( "sadness", 12 );
    cout << "wd: " << nl << wd << endl;
    return 0;
};
```

отпечатва следното:

```
wd:
<12> sadness
```

В раздели 5.5, 6.3 и 6.4 операцията "<<" е дефинирана съответно за класовете Screen, String и BitVector.

Операцията за стандартен изход е бинарна и връща резултат от тип ostream&. Общият вид на нейната дефиниция изглежда така:

```
ostream& operator << ( ostream& os, <ClassType>& )
{
    // ...операции, предизвикани от логиката на ClassType
    os << // ... членове, които се печатат
    return os;      // връща обект от клас ostream
}
```

Първият аргумент на тази операция трябва да бъде от тип ostream&, а вторият — от тип reference за съответния клас. Тъй като първият аргумент е от тип ostream&, операцията "<<" не трябва да бъде член на класа. (Този въпрос е разгледан подробно в раздел 5.5.) Ако операцията осъществява достъп до "скрити" членове на класа, тя трябва да се декларира като приятелска за този клас.

Loc е клас, който запазва номерата на реда и колоната за всяко срещане на дадена дума. Ето неговата дефиниция (заглавните файлове са пропуснати):

```
class Loc {
    friend ostream& operator<<(ostream&, Loc&);
public:
    Loc( int l, int c ) : line(l), column(c) {}
    Loc *next;
private:
    short line;
    short column;
};

ostream& operator << ( ostream& os, Loc& lc )
{
    // обектите от клас Loc се извеждат във формат <10,37>
    os << lbrace << lc.line
        << comma << lc.column << rbrace;
    return os;
}
```

Нека дефинираме отново клас WordCount. В новата дефиниция ще използваме два члена, които са обекти — съответно от клас String и клас Loc. Тези два класа притежават операции за стандартен изход.

```
class WordCount {
    friend ostream& operator<<(ostream&, WordCount&);
    friend istream& operator>>(istream&, WordCount&);
public:
    WordCount() : occurs(0), occurList(0) {}
    WordCount(char *s) : str(s), occurs(0), occurList(0) {}
    WordCount(char* s, int l, int c, int o = 1) : str(s), occurs(o)
        { occurList = new Loc(l,c); }
```

```

    WordCount(String&, Loc&, int = 1);
    void found( int, int );
private:
    String str;
    int occurs;
    Loc *occurList;
};

```

found() добавя нов обект от клас Loc към списъка от позиции, на които се среща дадена дума. Ще използваме следната дефиниция на found():

```

void WordCount::found( int l, int c ) {
    Loc *tmp = new Loc(l, c);
    ++occurs;
    tmp->next = occurList;
    occurList = tmp;
}

```

Два от членовете на клас WordCount са обекти от класове, които притежават операции за стандартен изход. Затова ще дефинираме отново операцията "<<" за клас WordCount:

```

ostream& operator << (ostream& os, WordCount& wd) {
    Loc *tptr = wd.occurList;

    os << lbrace << wd.occurs << rbrace << sp
        << wd.str << nl;    // os << String

    int tabCnt = 0, onLine = 5;
    while ( tptr ) {
        os << tab << *tptr;    // os << Loc
        if ( ++tabCnt >= onLine ) {
            os << nl; tabCnt = 0;
        }
        tptr = tptr->next;
    }
    return os;
}

```

Следващата програма илюстрира новата дефиниция на WordCount. За простота срещанията на думата са фиксирани.

```

#include <iostream.h>
#include "ioconst.h"
#include "WordCount.h"

main() {
    WordCount search( "rosebud" );

    // за простота шестте срещания на думата са фиксирани
    search.found( 7, 12 );          search.found( 7, 18 );
    search.found( 14, 2 );          search.found( 34, 36 );
    search.found( 49, 17 ); search.found( 67, 51 );

    cout << "Срещания: " << nl << search;
    return 0;
}

```

При изпълнение на тази програма се получава следния резултат:

Срещания:

```
<6> rosebud
      <67,51> <49,17> <34,36> <14,2> <7,18>
      <7,12>
```

Нека резултатът от програмата се съхранява във файл с име output. В раздел A.4 този файл се използва за четене на информация.

Операциите за стандартен изход, които са дефинирани за класове, не се наследяват. Това е така, защото те не са членове на съответните класове. Освен това те не могат да станат виртуални функции директно. В раздел 8.2 е показано как може да се осигури тяхната виртуалност без да се променя синтаксисът им.

A.3 Въвеждане на информация

Въвеждането на информация е аналогично на извеждането, но сега се използва операцията ">>". Тази операция е дефинирана за всички стандартни типове данни, включително и за тип char*. Операциите за стандартен вход също могат да се конкатенират. Например,

```
#include <iostream.h>
#include "ioconst.h"
#include "Loc.h"

main() {
    int line, col;

    cout << "Въведете две цели числа: " << flush;
    cin >> line >> col;
    Loc loc( line, col );
    cout << loc << endl;
    return 0;
}
```

отпечатва на потребителския терминал

```
Въведете две цели числа: 20 27
<20,27>
```

Често извличането на данни от входния поток се използва като условие в цикъл while. Например,

```
char ch;
while ( cin >> ch ) ...
```

чете по един символ преди всяко завъртане на цикъла. Ако този символ е символът за край на файл (end-of-file), условието в цикъла получава стойност лъжа и цикълът завършва.

За операцията ">>" поредицата от символи

```
ab c
d      e
```

е последователност от пет символа ('a', 'b', 'c', 'd', 'e'). Празните позиции (интервали, нови редове и табулации) са само разделители във входния поток и не се броят за символи. Ако потребителят иска да прочете празните позиции като нормални символи, трябва да използва някоя от функциите `get()`, `getline()` и `read()`. Тези функции са членове на клас `istream`. По-нататък ще ги разгледаме по-подробно.

Операцията `>>` може да се използва и за четене на символни низове от входния поток:

```
char inBuf[ wordLength ];
while (cin >> inBuf ) ...
```

Операцията `>>`, дефинирана за тип `char*`, разглежда символният низ като последователност от символи, завършваща с празна позиция. Дори когато има кавички, празните позиции в оградения с кавички израз не са част от един голям символен низ. Например,

```
"A fine and private place"
```

е последователност от пет символни низа:

```
"A
fine
and
private
place"
```

При въвеждане на тези низове в края на всеки от тях се добавя символа `null`. Следователно, размерът на символния масив, в който се помещава низа, трябва да бъде с единица по-голям от дължината на низа. Например,

```
char inBuf[ 4096 ];
while ( cin >> inBuf ) {
    char *pstr = new char[ strlen(inBuf) + 1 ];
    strcpy( pstr, inBuf );
    ...
}
```

Следващата програма чете низове от входния поток и определя низа с най-голяма дължина:

```
#include <iostream.h>
#include "ioconst.h"
#include "String.h"
#include <string.h>

char *msg[]={
```

```

        "Броят на прочетените думи е ",
        "Дължината на най-дългата дума е ",
        "Най-дългата дума е " };

const bufSize = 24;

main() {
    char buf[ bufSize ];
    String largest;

    int curLen, max = -1, cnt = 0;           // за статистика
    while ( cin >> buf ) {
        curLen = strlen( buf );
        ++cnt;

        // ако е открита нова най-дълга дума, тя се запазва
        if ( curLen > max ) {
            max = curLen;
            largest = buf;
        }
    }

    cout << msg[0] << cnt << nl;
    cout << msg[1] << max << nl;
    cout << msg[2] << largest << nl;
    return 0;
}

```

Като вход за програмата ще използваме първите няколко изречения от английския текст на новелата "Моби Дик":

```

Call me Ishmael. Some years ago, never mind
how long precisely, having little or no money
in my purse, and nothing particular to interest
me on shore, I thought I would sail about a little
and see the watery part of the world. It is a
way I have of driving off the spleen, and
regulating the circulation.

```

След изпълнение на програмата получаваме следния печат:

```

Броят на прочетените думи е 58
Дължината на най-дългата дума е 12
Най-дългата дума е circulation.

```

Програмата съхранява всеки прочетен низ в `buf`, който е масив с дължина 24. Ако дължината на някой от прочетените низове е по-голяма или равна на 24, `buf` ще се препълни. При изпълнение на програмата ще възникне грешка.

За да се предпазим от препълване, можем да използваме манипулатора `setw()`. Програмата трябва да се коригира така:

```

while ( cin >> setw( bufSize ) >> buf )

```

bufSize е дължината на масива buf. setw() нахвърля низовете с дължина по-голяма или равна на bufSize на два или повече низа с максимална дължина bufSize - 1. В края на всеки нов низ се поставя символа null. За да се използва setw(), трябва да се включи заглавният файл iomanip.h.

Ако видимата декларация на масива не задава дължина, към идентификатора на масива може да се приложи операцията sizeof:

```
char buf[] = "An unrealistic example";
while ( cin >> setw(sizeof( buf )) >> buf )
```

Употребата на sizeof предизвиква неочаквано държание на следващата програма:

```
#include <iostream.h>
#include <iomanip.h>
#include "ioconst.h"

const bufSize = 24;
char buf[ bufSize ];

main() {
    char *pbuf = buf;

    // всеки низ, който е по-дълъг от sizeof(char*),
    // се разделя на два или повече низа
    while ( cin >> setw(sizeof(pbuf)) >> pbuf )
        cout << pbuf << nl;
    return 0;
}
```

След компилиране и изпълнение на програмата получаваме неверен резултат:

```
$a.out
The winter of our discontent

The
win
ter
of
our
dis
con
ten
t
```

setw() получава размера на указателя към масива от символи, а не размера на самия масив. На компютъра, на който е изпълнена програмата, този указател заема четири байта. Затова низовете са нахвърляни на низове с максимална дължина от три символа. Следващият опит за коригиране предизвиква още по-сериозна грешка:

```
while ( cin >> setw(sizeof(*pbuf)) >> pbuf )
```


Целта на тази поправка е `setw()` да получи размера на масива, който се адресира от `rbuf`. Но означението `*rbuf` се възприема като идентификатор от тип `char`. Сега `setw()` получава стойност 1. При всяко завъртане на цикъла в масива, адресиран от `rbuf`, се поставя символа `null`. Входният поток не се прочита. Цикълът се изпълнява безкрайно.

Функциите `get()` и `getline()` могат да се използват, ако програмистът не иска да пропуска празните позиции във входния поток. Функцията `get()` има две форми:

1. `get(char& ch)` извлича отделен символ от входния поток и го присвоява на `ch`. Функцията връща обекта от клас `istream`, чрез който се извиква. Например:

```
#include <iostream.h>

main() {
    char ch;
    while (cin.get(ch)) cout.put(ch);
    return 0;
}
```

2. `get()` извлича и връща отделна стойност от входния поток. Тази стойност може да бъде и EOF (край на файл). Стойността EOF е дефинирана в `istream.h`. Да разгледаме един пример:

```
#include <iostream.h>
main() {
    int ch;
    while ( (ch = cin.get()) != EOF ) cout.put(ch);
    return 0;
}
```

Най-често стойността на EOF е -1, за да се различава от стойностите на символите. Ако `ch` може да е символ или EOF, типът на тази променлива не бива да бъде `char`. Работата с тип `signed char` зависи от компютъра. Ако `ch` е от тип `char`, при сравнението на `ch` с EOF може да се получи неверен резултат. Затова в примера `ch` е от тип `int`.

`get()` прочита следната последователност от символи за седем итерации:

```
a b c
d
```

Седемте прочетени символи са: 'a', интервал, 'b', интервал, 'c', нов ред и 'd'. На осмата итерация се прочита EOF. Операцията ">>" прочита същата последователност от символи за четири итерации, защото пропуска празните позиции.

Функцията `getline()` има следната сигнатура:

```
getline( char *Buf, int Limit, char Delim ='\n');
```

Тя извлича от входния поток поредица от символи и ги запазва в масива, който се адресира от `Buf`. `getline()` извлича най-много `Limit - 1` на брой символи, защото в края на `Buf` се добавя символа `null`. При срещане на край на файл (end-of-file) или на символа, който се съдържа в

променливата `Delim`, `getline()` извлича по-малко от `Limit - 1` на брой символи. По подразбиране стойността на `Delim` е символа за нов ред. `Delim` не се помества в `Buf`.

Функцията `gcount()`, която е член на клас `istream`, връща действителния брой символи, извлечени от входния поток при последното извикване на `getline()`. Следващият пример илюстрира употребата на `gcount()` и `getline()`:

```
#include <iostream.h>
#include "ioconst.h"

const lineSize = 1024;
main() {
    int lcnt = 0;          // брой на прочетените редове
    int max = -1;         // дължина на най-дългия ред
    char inBuf[lineSize];

    // прочитат се 1024 символа или всички символи до края на реда
    while (cin.getline(inBuf, lineSize)) {
        // определя действителния брой прочетени символи
        int readin = cin.gcount();

        // статистика за броя на редовете и за най-дългия ред
        ++lcnt;
        if ( readin >> max ) max = readin;
        cout << "Ред #" << lcnt << tab
              << "Прочетени символи: " << readin << nl;
        cout.write(inBuf, readin).put(nl);
    }
    cout << "Общ брой прочетени редове: " << lcnt << nl;
    cout << "Дължина на най-дългия ред: " << max << nl;
    return 0;
}
```

Ако изпълним програмата за първите няколко изречения от новелата "Моби Дик" получаваме следния резултат:

Ред # 1 Прочетени символи: 45
Call me Ishmael. Some years ago, never mind

Ред # 2 Прочетени символи: 46
how long precisely, having little or no money

Ред # 3 Прочетени символи: 48
in my purse, and nothing particular to interest

Ред # 4 Прочетени символи: 50
me on shore, I though I would sail about a little

Ред # 5 Прочетени символи: 47
and see the watery part of the world. It is a

Ред # 6 Прочетени символи: 43
way I have of driving off the spleen, and

Ред # 7 Прочетени символи: 28
regulating the circulation.

Общ брой прочетени редове: 7
Дължина на най-дългия ред: 50

Функцията `read()` е в известен смисъл обратна на функцията `write()`. `read()` е член на клас `istream`. Нейната сигнатура изглежда така:

```
read( char* addr, int size );
```

Тази функция извлича `size` на брой последователни байта от входния поток и ги запазва в масива, адресиран от `addr`. `gcount()` може да върне и броя на извлечените байтове при последното обръщение към `read()`. Най-често функцията `read()` се използва за прочитане на информация, която е изведена чрез функцията `write()`.

Потребителят трябва да познава още три функции:

```
putback( char c );    // връща символ обратно във входно-изходния поток
peek();              // връща следващия символ или EOF, но не го извлича

// прескача Limit на брой символа или
// всички символи до символа, който е стойност на Delim
ignore( int Limit=1, int Delim=EOF );
```

Следващият програмен фрагмент илюстрира употребата на тези три функции:

```
char ch, next, lookahead;
while ( cin.get(ch) ) {
    switch (ch) {
        case '/':
            // Проверява дали редът е коментар. Ако това е така, игнорира всичко до края на реда
            next = cin.peek();
            if( next == '/' ) cin.ignore( lineSize, '\n' );
            break;
        case '>':
            // Търси >=
            next = cin.peek();
            if( next == '>' ) {
                lookahead = cin.get();
                next = cin.peek();
                if ( next != '=' )
                    cin.putback(lookahead);
            }
            break;
    }
}
```

А.4 Дефиниране на операцията ">>" за класове

Операцията ">>" се дефинира аналогично на операцията "<<". На фиг. А.1 е дефинирана операцията ">>" за клас `WordCount`. Тя илюстрира последиците от възможните грешки при работа с входно-изходния поток, които най-общо са следните:

1. Четене от входния поток, което не може да се осъществи заради неправилен формат, трябва да се маркира като невалидно. Изразът

```
is.clear( ios::badbit | is.rdstate() )
```

прави точно това. Да обясним какво означава той. Възможните грешки във входно-изходния поток се управляват чрез поредица от битове. Функцията `rdstate()` връща тази поредица от битове. `clear(ios::badbit)` установява един от битовете за грешка в зависимост от значението на `badbit` и нулира останалите битове. Ние искаме да запазим предишното състояние на битовете за грешки. Затова извършваме побитово ИЛИ между `ios::badbit` и предишното състояние на тези битове:

```
is.clear( ios::badbit | is.rdstate() )
```

2. В случай на грешка функциите, които работят с входно-изходния поток, не извършват никакви действия. Например, ако `is` е в състояние на грешка

```
while ((ch = is.get()) != lbrace)
```

ще зацikli. Ето защо преди всяко извикване на `get()` е добре да се провери състоянието на `is`:

```
while ( is && (ch = is.get()) != lbrace ) // за да се убедим, че не е възникнала грешка
```

При възникване на грешка условието в `while` получава стойност 0.

Следващата програма чете обект от клас `WordCount`, който е изведен по-рано чрез операцията "<<", дефинирана за същия клас в раздел А.2:

```
#include <iostream.h>
#include <stdlib.h>
#include "ioconst.h"
#include "WordCount.h"

main() {
    WordCount readIn;

    if ( (cin >> readIn) == 0 ) {
        cerr << "Грешка при четене на обект от клас WordCount" << nl;
        exit( -1 );
    }
    cout << readIn << nl;
    return 0;
}
```

Изразът

```
if ( (cin >> readIn) == 0 )
```

проверява дали операцията ">>" за клас `WordCount` е прочела успешно обекта. След компилация и изпълнение на програмата получаваме следния резултат:

```
<6> rosebud
```

<7,12> <7,18> <14,2> <34,36> <49,17>
<67,51>

Следва фигура А.1:

```
#include <iostream.h>
#include "ioconst.h"
#include "WordCount.h"

istream& operator >> (istream& is, WordCount& wd) {
    /* формат за четене на обекти от клас WordCount:
     * <2> низ
     * <7,3> <12,36> */

    int ch;
    if((ch = is.get()) != lbrace) { // неправилен формат
        is.clear( ios::badbit | is.rdstate() );
        return is;
    }

    is >> wd.occurs;
    while ( is && (ch = is.get()) != rbrace ) ;
    const bufSize = 512;
    char buf[ bufSize ];
    is >> buf;
    wd.str = buf;

    // четене на срещанията на думата; всяко срещане е записано във формат <l,c>
    for ( int j = 0; j < wd.occurs; ++j ) {
        Loc *tptr;
        int l, c;

        // извличане на стойностите
        while ( is && (ch = is.get()) != lbrace ) ;
        is >> l;

        while ( is && (ch = is.get()) != comma ) ;
        is >> c;

        while ( is && (ch = is.get()) != rbrace ) ;

        // създаване на списъка на срещанията на прочетената дума
        tptr = new Loc( l, c );
        tptr->next = wd.occuList;
        wd.occuList = tptr;
    }
    return is;
}
```

Фигура А.1 Дефиниция на операцията ">>" за клас WordCount

А.5 Четене и запис във файл

За да свържем един файл с програма за четене и/или запис на информация, освен заглавния файл `iostream.h` трябва да включим и заглавния файл `fstream.h`:

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

За да отворим файл само за запис, трябва да дефинираме обект от клас ofstream:

```
ofstream outFile( "copy.out", ios::out );
```

Аргументите, които се предават на конструктора на клас ofstream са последователно име на файла, който се отваря и режим за работа с него. Когато се отваря файл за запис, възможните режими са два: ios::out (отваря файл за запис на информация) и ios::app (отваря файл за добавяне на информация).

Ако съществуващ файл се отвори в режим ios::out, всички данни, които се съдържат в него изчезват. За да се запише информация в съществуващ файл, файлът трябва да се отвори в режим ios::app. Тогава новите данни се добавят в неговия край. При отваряне на несъществуващ файл и в двата режима се създава нов файл.

Преди да се записва или чете от даден файл, е добре да се провери дали файлът е отворен успешно. Например:

```
if ( !outFile ) { // при неуспешно отворен файл
    cerr << "Не мога да отворя `copy.out` за запис\n";
    exit( -1 );
}
```

Клас ofstream е произведен на клас ostream. Следователно, всички функции-членове на клас ostream са достъпни за обекти от клас ofstream. Например,

```
outFile.put( '1' ).put( ' ' ).put( sp );
outFile << "1 + 1 = " << (1 + 1) << nl;
```

въвежда във файла copy.out

```
1) 1 + 1 = 2
```

Следващата програма записва извлечените от стандартния вход символи във файла copy.out:

```
#include <iostream.h>
#include <fstream.h>
#include "ioconst.h"
#include <stdlib.h>

main() {
    ofstream outFile( "copy.out", ios::out ); // отваря за запис файл с име copy.out

    if ( !outFile ) { // при неуспешно отворен файл
        cerr << "Не мога да отворя `copy.out` за запис\n";
        exit( -1 );
    }
}
```

```

char ch;
while ( cin.get( ch )) outFile.put( ch );
return 0;
}

```

Дефинираните от потребителя операции "<<" също могат да се използват с обект от клас ofstream. Следващата програма извиква операцията "<<", дефинирана за клас WordCount:

```

#include <iostream.h>
#include <fstream.h>
#include "ioconst.h"
#include "WordCount.h"

main() {
    ofstream outFile( "word.out", ios::out );    // отваря за запис файл с име word.out
    // проверка за успешно отваряне на файла ...

    // създаване списък на срещанията на дадената дума
    WordCount artist( "Renoir" );
    artist.found( 7, 12 );
    artist.found( 34, 18 );

    // обръщение към operator<<(ostream&, WordCount&);
    outFile << artist;
    return 0;
}

```

За да отворим файл само за четене, трябва да дефинираме обект от клас ifstream. Клас ifstream е производен на клас istream и наследява всички негови членове. Следващата програма чете файла copy.out и извежда прочетеното на стандартния изход:

```

#include <iostream.h>
#include <fstream.h>

main() {
    ifstream inFile( "copy.out", ios::in );    // отваря за четене файл с име copy.out
    // проверка за успешно отваряне на файла ...

    char ch;
    while ( inFile.get( ch )) cout.put( ch );
    return 0;
}

```

Константата ios::in означава, че файлът copy.out е отворен за четене.

Следващата програма копира информацията от един файл в друг, като разполага всеки низ на отделен ред:

```

#include <iostream.h>
#include <fstream.h>
#include <iomanip.h>
#include "ioconst.h"

char *infile = "in.fil";
char *outfile = "out.fil";

```

```

main() {
    ifstream iFile( infile, ios::in );
    ofstream oFile( outfile, ios::out );

    // проверка за успешно отваряне на двата файла ...

    int cnt = 0;
    const bufSize = 1024;
    char buf[ bufSize ];

    while ( iFile >> setw( bufSize ) >> buf ) {
        oFile << buf << nl;
        ++cnt;
    }

    // извежда общия брой на низовете, записани във втория файл
    cout << "[ " << cnt << " ]" << nl;
    return 0;
}

```

Обекти от клас `ifstream` и `ofstream` могат да се дефинират без да се задава име на файл. По-нататък в програмата чрез функцията `open()` с такъв обект може да се свърже конкретен файл. Например:

```

ifstream curFile;
// ...
curFile.open( filename, ios::in );
if ( !curFile )      // при неуспешно отваряне на файла
// ...

```

Тук `filename` е от тип `char*`.

Връзката между даден файл и програмата, която работи с него, може да се прекъсне чрез функцията `close()`. Например:

```

curFile.close();

```

В следващата програма се отварят и затварят пет файла, като се използва само един обект от клас `ifstream`:

```

#include <iostream.h>
#include <fstream.h>

const fileCnt = 5;
char *fileTab[ fileCnt ] = { "Melville", "Joyce", "Musil", "Proust", "Kafka" };

main() {
    ifstream inFile; // не се свързва с конкретен файл
    for ( int i = 0; i < fileCnt; ++i ) {
        inFile.open( fileTab[i], ios::in );
        // ... проверка за успешно отваряне
        // ... работа с отворения файл
        inFile.close();
    }
    return 0;
}

```



```
}
```

Един файл може да се отвори за четене или запис чрез дефиниране на обект от клас `fstream`. Клас `fstream` е произведен на клас `iostream`. В следващия пример файлът `word.out` най-напред се прочита. След това в него се записва информация. За целта се използва обект от клас `fstream`. Файлът `word.out` е създаден по-рано в този раздел. Той съдържа обект от клас `WordCount`.

```
#include <iostream.h>
#include <fstream.h>
#include "ioconst.h"
#include "WordCount.h"

main() {
    WordCount wd;
    fstream file;

    file.open( "word.out", ios::in );
    file >> wd;
    file.close();
    cout << "Прочетеният обект е: " << wd << endl;

    file.open( "word.out", ios::app );
    file << endl << wd;
    file.close();
    return 0;
}
```

Чрез обект от клас `fstream` един файл може да се отвори едновременно за четене и запис. Например, със следващата дефиниция файлът `word.out` се отваря едновременно за четене и добавяне на информация:

```
fstream io( "word.out", ios::in | ios::app );
```

Побитовата операция ИЛИ се използва за задаване на няколко режима за един и същи файл.

Функциите `seekg()` и `seekp()` позволяват преместване на "абсолютен" адрес в даден файл или преместване с определен брой байтове от определена позиция. Те притежават следните два аргумента:

```
typedef long streampos;
seekg( streampos p, seek_dir d=ios::beg );
```

където `seek_dir` е изброим тип с три елемента:

1. `ios::beg`, начало на файл
2. `ios::cur`, текуща позиция във файл
3. `ios::end`, край на файл

При обръщение с два аргумента се извършва преместване с определен брой байтове от определена позиция. В следващия пример на всяка итерация от цикъла се позиционираме на *i*-тия въведен `Record`:

```
for( int i = 0; i < recordCnt; ++i )
    readFile.seekg( i * sizeof(Record), ios::beg );
```

Първият аргумент може да има и отрицателна стойност. В следващия пример се премества с 10 байта назад от края на файла:

```
readFile.seekg( -10, ios::end );
```

Ако тези функции се извикат само с един аргумент, преместването ще бъде с n байта от началото на файла. Следователно, ако вторият аргумент е `ios::beg`, той може да се пропусне. `tellg()` връща текущата позиция във файла. Например:

```
streampos mark = writeFile.tellg(); // маркиране на текущата позиция
// ...
if ( cancelEntry )
    writeFile.seekg( mark ); // връщане на маркираната позиция
```

Ако искаме да се придвижим с един `Record` напред от текущата позиция във файла, можем да използваме един от следните два начина:

```
// два еквивалентни начина за придвижване с един Record напред
readFile.seekg( readFile.tellg() + sizeof(Record) );

// този начин се смята за по-ефективен
readFile.seekg( sizeof(Record), ios::cur);
```

Нека разгледаме подробно един цялостен пример. Даден е текстов файл за четене. Трябва да се пресметне неговата дължина в байтове и да се запише в края на файла. Освен това при срещане на нов ред в края на файла трябва да се запише текущия брой байтове, като се брой и символа за нов ред. Ако е даден текстов файл

```
abcd
efg
hi
j
```

програмата трябва да създаде текстовия файл

```
abcd
efg
hi
j
5 9 12 14 24
```

Ето един начален вариант на програмата:

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

```

#include "ioconst.h"
#include <fstream.h>

main() {
    // отваряне на файла едновременно за четене и добавяне на информация
    fstream inOut( "copy.out", ios::in|ios::app );

    int cnt = 0; // брояч на байтове
    char ch;

    while ( inOut.get( ch ) ) {
        cout.put( ch ); // печат на терминала
        ++cnt;
        if ( ch == nl ) {
            inOut << cnt;
            inOut.put( sp );
        }
    }

    // извеждане на крайния брой байтове
    inOut << cnt; inOut.put(nl);
    cout << "[ " << cnt << " ]\n";
    return 0;
}

```

inOut е обект от клас fstream, свързан с файл с име copy.out, който е отворен едновременно за четене и добавяне на информация. Файл, който е отворен за добавяне на информация, записва новите данни в своя край.

При всяко прочитане на символ, включително и празна позиция, но не и маркер за край на файл, cnt се увеличава, а символът се отпечатва на терминала. Отпечатването се използва като непосредствена проверка за коректността на програмата.

При всяко срещане на нов ред, текущата стойност на cnt се записва в inOut. При прочитане на край на файл, цикълът завършва. Тогава в inOut и на екрана се записва окончателния брой байтове във файла, пресметнати в cnt. Програмата завършва. Тя изглежда коректна. Нека файлът съдържа първите няколко изречения от новелата "Моби Дик", поместени в началото на това приложение. След изпълнение на програмата получаваме следния резултат

[0]

След като не се отпечатват никакви символи, явно програмата смята, че файла е празен. Причина за този резултат е режима на добавяне, с който е отворен файла. При такъв режим текущата позиция във файла е неговия край. Когато се изпълнява

```
inOut.get( ch )
```

се прочита маркера за край на файл и цикъл while приключва. Стойността на cnt остава 0. Можем да решим възникналия проблем, ако преди четенето на символи, се позиционираме в началото на файла. Следващият оператор извършва точно това:

```
inOut.seekg( 0, ios::beg )
```

Отново компилираме и изпълняваме програмата. Сега получаваме следния печат:

```
Call me Ishmael. Some years ago, never mind [ 45 ]
```

Програмата отпечатва първия ред от текстовия файл и броя на байтовете, които съответстват на този ред, като очевидно смята, че това е целия файл. Какъв е проблемът сега?

Отново проблемът е в това, че файлът е отворен за добавяне на информация. След първото отпечатване на cnt, отново се позиционираме в края на файла. Следващото обръщение към get() прочита маркера за край на файл. Затова цикълът завършва преждевременно.

Решението този път е след всеки запис на cnt да се позиционираме там, където сме били преди добавянето на cnt. Това се реализира от следващите два допълнителни оператора:

```
streampos mark = inOut.tellg();    // маркиране на текущата позиция
inOut << cnt << sp;
inOut.seekg( mark );               // връщане на маркираната позиция
```

След ново компилиране и изпълнение на програмата очакваният печат излиза на терминала, но ако разгледаме получения файл, ще открием друг проблем. Макар че окончателният брой байтове се печата на терминала, той не се записва във файла. Очевидно операцията "<<" след цикъл while не се изпълнява. Това е така, защото след приключване на цикъла се намираме върху маркера за край на файл. Докато се намираме в тази позиция операциите за четене и запис не се изпълняват. Състоянието, в което се намира файла, трябва да се "изчисти". За целта може да се използва следния оператор:

```
inOut.clear();    // нулира всички флагове на състоянието
```

Модифицираната програма изглежда така:

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

main() {
    fstream inOut( "copy.out", ios::in | ios::app );
    int cnt = 0;
    char ch;

    inOut.seekg(0, ios::beg);
    while ( inOut.get( ch ) ) {
        cout.put( ch );
        ++cnt;
        if ( ch == '\n' ) {
            // маркиране на текущата позиция
            streampos mark = inOut.tellg()
            inOut << cnt << ' ';
            inOut.seekg( mark );    // връщане на маркираната позиция
        }
    }

    inOut.clear();
    inOut << cnt << '\n';
    cout << "[ " << cnt << " ]\n";
}
```

```

    return 0;
}

```

След компилиране и изпълнение на този вариант на програмата най-после всичко е наред.

А.6 Флагове на състоянието

Обектите от класовете в библиотека `iostream` поддържат множество флагове, чрез които може да се управлява състоянието на входно-изходния поток. Могат да се използват четири предикатни функции-членове:

1. `eof()`, която връща стойност истина, при срещане на маркер за край на файл (end-of-file). Например:

```
if ( inOut.eof() ) inOut.clear();
```

2. `bad()`, която връща стойност истина, при опит за изпълнение на неправилна операция, като търсене на информация след маркера за край на файл.
3. `fail()`, която връща стойност истина, при неуспешно завършване на операция или ако `bad()` връща стойност истина. Например:

```
ifstream iFile( filename, ios::in );
if ( iFile.fail() )      // файлът не може да се отвори
    error(...);
```

4. `goot()`, която връща стойност истина, ако предишните три функции връщат стойност лъжа. Например:

```
if ( inOut.goot() )
```

Всички класове от библиотека `iostream` притежават логическата операция NOT ("!"). Изразът

```
if ( !inOut )
```

е съкратен запис на по-сложния израз

```
if ( inOut.fail() )
```

Съществува възможност за проверка дали даден файл е отворен вече. Например,

```
ifstream ifile;
```

дефинира обект от клас `ifstream`, който не е свързан с конкретен файл. Преди да запишем

```
ifile << "hello, world\n";
```

е добре да проверим дали ifile е свързан с конкретен файл. Тъй като не са изпълнявани никакви операции, проверката

```
if ( ifile.good() )      // ...
```

връща стойност истина, даже когато файлът все още не е отворен. Следващият предикат проверява дали ifile е отворен:

```
if ( ifile.rdbuf()—>is_open == 0 ) {  
    // няма връзка с конкретен файл  
    ifile.open( filename, ios::in );  
    // ...
```

rdbuf() е функция, която има достъп до вътрешния буфер за връзка с файла.

А.7 Флагове за формат

За всеки обект от класовете в библиотека iostream се поддържат флагове за формат. Има флаг, който определя бройната система за представяне на целите числа или флаг, който определя цифрите след десетичната точка за числа с плаваща точка. Програмистът може да установява в единица или да нулира флаговете за формат чрез функциите setf() и unsetf(). Освен това съществуват няколко манипулатора за промяна на формата.

По подразбиране целите числа се записват и четат в десетичен формат. Бройната система може да се промени на осмична или шестнадесетична, а след това да се върне отново на десетична. Използват се манипулаторите oct, hex и dec. Те се добавят между обекта от който се чете или записва информация и стойността, която ще се чете или записва. Например:

```
#include <iostream.h>  
  
main() {  
    int i = 512;  
    cout << "Във формат по подразбиране: " << i << "\n";  
  
    // oct сменя основата на осмична  
    cout << "В осмична бройна система: " << oct << i;  
    cout << "\t" << i << "\n";  
  
    // hex сменя основата на шестнадесетична  
    cout << "В шестнадесетична бройна система: " << hex << i;  
    cout << "\t" << i << "\n";  
  
    cout << "В десетична бройна система: " << dec << i;  
    cout << "\t" << i << "\n";  
    return 0;  
}
```

След компилиране и изпълнение на програмата получаваме:

```
Във формат по подразбиране:          512
```

В осмична бройна система:	1000	1000
В шестнадесетична бройна система:	200	200
В десетична бройна система:	512	512

Използването на един от манипулаторите `oct`, `hex` или `dec` променя стандартното интерпретиране на целите числа за всички следващи цели стойности, които се четат или записват. Вторият печат в програмата след всяко използване на манипулатор доказва това.

За правилно прочитане на изведените стойности, трябва да се приложи подходящ манипулатор. Например:

```
#include <iostream.h>
main() {
    int i1, i2, i3, i4, i5, i6, i7;
    cin >> i1
        >> oct >> i2 >> i3
        >> hex >> i4 >> i5
        >> dec >> i6 >> i7;
}
```

По подразбиране реалните числа се извеждат с шест цифри след десетичната точка. Този брой може да се промени чрез функцията `precision(int)`. `precision()` връща текущия брой цифри след десетичната точка. Например:

```
#include <iostream.h>
#include "ioconst.h"
#include <math.h>

main() {
    cout << "Брой цифри след десетичната точка: "
    << cout.precision() << nl
    << sqrt(2.0) << nl;

    cout.precision(12);
    cout << "\nБрой цифри след десетичната точка: "
    << cout.precision() << nl
    << sqrt(2.0) << nl;
    return 0;
}
```

След компилиране и изпълнение на програмата се извежда следния печат:

```
Брой цифри след десетичната точка: 6
1.41421
```

```
Брой цифри след десетичната точка: 12
1.41421356237
```

Може да се използва и манипулаторът `setprecision()`, но тогава трябва да се включи заглавния файл `iomanip.h`. Например:

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

```
#include <iomanip.h>
#include "ioconst.h"
#include <math.h>

main() {
    cout << "Брой цифри след десетичната точка: "
    << cout.precision() << nl << sqrt(2.0) << nl
    << setprecision( 12 )
    << "\nБрой цифри след десетичната точка: "
    << cout.precision() << nl << sqrt(2.0) << nl;
    return 0;
}
```

Функцията `setf()` установява определен флаг за формат. Съществуват две различни функции с това име:

```
setf( long );
setf( long, long );
```

Първият аргумент може да бъде флаг за формат или поле за формат. Таблица А.1 съдържа списък на флаговете за формат, които ще разгледаме в този раздел.

Флаг	Значение
<code>ios::showbase</code>	показва бройната система
<code>ios::showpoint</code>	показва нулите и дес. точка
<code>ios::dec</code>	десетична бройна система
<code>ios::hex</code>	шестнадесетична бройна система
<code>ios::oct</code>	осмична бройна система
<code>ios::fixed</code>	фиксиран формат
<code>ios::scientific</code>	формат с експонента

Табл. А.1 Флагове за формат

Вторият аргумент е поле за формат, свързано с определено множество флагове за формат. Таблица А.2 съдържа двете полета за формат, които ще разгледаме в този раздел.

Ако искаме да видим бройната система на извежданите на стандартен изход цели числа, можем да запишем следното:

```
cout.setf( ios::showbase );
```


Поле	Значение	Флаг
ios::basefield	управлява извеждането на целите числа	ios::dec ios::hex ios::oct
ios::floatfield	управлява извеждането на числата с плаваща точка	ios::fixed ios::scientific

Табл. А.2 Полета за формат

Аналогично, за да направим основата осмична, можем да използваме:

```
cout.setf( ios::oct, ios::basefield );
```

setf(long, long) първо нулира цялото поле за формат, а след това установява в единица флага, който съответства на първия аргумент. Функцията връща предишното състояние като число от тип long. Следователно, предишното състояние може да се запази и да се възстанови покъсно, ако се предаде като първи аргумент на същата функция. Например:

```
#include <iostream.h>
#include "ioconst.h"

main() {
    int ival = 1024;
    long oldbase;

    cout.setf( ios::showbase );
    cout.setf( ios::oct, ios::basefield );
    cout << "ival: " << ival << nl;

    oldbase = cout.setf( ios::hex, ios::basefield );    // запазва текущата основа
    cout << "ival: " << ival << nl;

    cout.setf( oldbase, ios::basefield );              // възстановява предишната основа
    cout << "ival: " << ival << nl;

    return 0;
}
```

След компилиране и изпълнение на програмата получаваме:

```
ival: 02000
ival: 0x400
ival: 02000
```

Тук не използвахме по-простия запис

```
cout.setf( ios::hex );
```

защото `setf(long)` не изчиства предишното състояние. Ако преди това `ios::basefield` има стойност `ios::dec`, по този начин ще се установят два флага — `ios::hex` и `ios::dec`. Когато в полето `ios::basefield` са установени няколко флага за бройна система или всички флагове са нулирани, по подразбиране извеждането става в десетичен формат.

По подразбиране в числата с плаваща точка нулите след точката, след които няма други значещи цифри, не се извеждат. Например,

```
cout << 10.70 << nl;
```

отпечатва

```
10.7
```

Следващият оператор не извежда не само нулата след десетичната точка, но и самата точка:

```
cout << 10.0 << nl;
```

За да се изведат нулите и десетичната точка в горните примери, трябва да се установи флага `ios::showpoint`:

```
cout.setf( ios::showpoint );
```

Полето `ios::floatfield` управлява визуалното представяне на числата с плаваща точка. С това поле са свързани следните два флага:

1. `ios::fixed`, изобразява стойностите във фиксиран формат.
2. `ios::scientific`, изобразява стойностите във формат с експонента.

Да разгледаме един пример:

```
#include <iostream.h>
#include <iomanip.h>
#include "ioconst.h"
#include <math.h>

main() {
    cout << setprecision( 8 )
        << "\nБрой цифри след десетичната точка: "
        << cout.precision() << nl
        << "10.0 във формат по подразбиране: " << 10.0 << nl
        << "pow(18,10) във формат по подразбиране: "
        << pow( 18,10 ) << nl;

    cout.setf( ios::fixed, ios::floatfield );

    cout << "\nБрой цифри след десетичната точка: "
        << cout.precision() << nl
        << "10.0 във фиксиран формат: " << 10.0 << nl
        << "pow(18,10) във фиксиран формат: "
        << pow( 18,10 ) << nl;
```

```

cout.setf( ios::scientific, ios::floatfield );
cout << "\nБрой цифри след десетичната точка: "
    << cout.precision() << nl
    << "10.0 във формат с експонента: " << 10.0 << nl
    << "pow(18,10) във формат с експонента: "
    << pow( 18,10 ) << nl;
return 0;
}

```

След компилиране и изпълнение на програмата получаваме следния печат:

```

Брой цифри след десетичната точка: 8
10.0 в формат по подразбиране: 10
pow(18,10) във формат по подразбиране: 3.5704672e+12

```

```

Брой цифри след десетичната точка: 8
10.0 във фиксиран формат: 10.00000000
pow(18,10) във фиксиран формат: 3570467226624.00012000

```

```

Брой цифри след десетичната точка: 8
10.0 във формат с експонента: 1.00000000e+1
pow(18,10) във формат с експонента: 3.57046723e+12

```

A.8 Операции с масиви от символи

Библиотеката `iostream` поддържа множество операции с масиви от символи. Чрез обекти от клас `ostrstream` символите от входния поток се поместват в масив от символи, а чрез обекти от клас `istrstream` се извличат символи от някакъв масив и се предават на изходния поток. За да могат да се дефинират обекти от тези два класа, програмата трябва да включва заглавния файл `strstream.h`. Например, в следващия програмен фрагмент се прочита файла `words` и се помества в масив от символи, заделен динамично чрез `buf`.

```

#include <iostream.h>
#include <fstream.h>
#include <strstream.h>

ifstream ifile("words", ios::in);
ostrstream buf;

processFile() {
    char ch;
    while (buf && ifile.get(ch)) buf.put(ch);
    // ...
}

```

Функцията `str()` връща указател към масив от символи, свързан с конкретен обект от клас `ostrstream`. Например,

```
char *bp = buf.str();
```

инициализира `bp` с адреса на масива от символи, свързан с `buf`. Сега чрез `bp` с масива може да се работи като с "обикновен" масив от символи.

С извикването на `str()` масивът "се предава" на програмиста. Обръщението към `str()` има ефект на освобождаване на масива. По-нататъшна работа с този масив чрез операциите и функциите на клас `ostrstream` е невъзможна. Следователно, `str()` трябва да се извиква след като запълването на масива от символи е завършило.

Когато обектът от клас `ostrstream` напусне своята област на действие и функцията `str()` не е извикана, масивът се изтрива автоматично от паметта чрез деструктора на класа. В противен случай освобождаването на паметта е грижа на програмиста. Затова преди напускане на областта на действие на `bp`, масивът, който се адресира от него, трябва да се изтрие явно:

```
delete bp;
```

Да разгледаме един пример. В следващата програма `inStore` е обект от клас `ostrstream`. Чрез него се съхранява съдържанието на файла `words`. (`words` съдържа няколко изречения от "Моби Дик".) За разлика от предишния пример, файлът се съхранява като поредица от низове, завършващи със символа `null`. `ends` е манипулатор, който добавя символа `null`.

```
#include <iostream.h>
#include <fstream.h>
#include <strstream.h>
#include <iomanip.h>
#include <string.h>

ifstream ifile("words", ios::in);
ostrstream inStore;
const len = 512;

main() {
    char inBuf[len];
    int wordCnt = 0;

    while ( inStore && ifile >> setw(len) >> inBuf ) {
        ++wordCnt;
        inStore << inBuf << ends;
    }

    char *ptr = inStore.str();    // поемане на управлението над масива
    char *tptr = ptr;            // пази началото на масива от символи

    // words е масив от указатели към низове с дължина
    // равна на броя на прочетените низове от ifile
    char **words = new char*[wordCnt];
    wordCnt = 0;

    // инициализира всеки елемент на words с адреса на низа в inStore
    while ( *ptr ) {
        words[ wordCnt++ ] = ptr;
        ptr += strlen(ptr) + 1;
    }

    // отпечатва низовете в обратен ред
    // на всеки ред се печатат по lineLength думи
    const lineLength = 8;
    for ( int i = wordCnt - 1, cnt = 0; i >= 0; --i ) {
        if ( cnt++ % lineLength == 0 ) {
            cout << endl;
            cnt = 1;
        }
    }
}
```

```

    }
    cout << words[i] << " ";
}
cout << endl;
delete tptr;
}

```

След компилиране и изпълнение на програмата получаваме следния печат:

```

circulation. the regulating and spleen, the off driving
of have I way a is It world.
the of part watery the see and little
a about sail would I though I shore,
on me interest to particular nothing and purse,
my in money no or little having precisely,
long how mind never ago, years Some Ishmael.
me Call

```

Възможно е да се дефинира обект от клас `ostream` като се използва предварително заделена памет за масива от символи. Тогава се задават три аргумента:

1. Указател `sr` от тип `char*` към предварително дефиниран масив от символи.
2. Размер на масива в байтове от тип `long`.
3. Режим, който има две възможни стойности — `ios::app` и `ios::out`. Ако режимът е `ios::app`, `sr` е низ, който завършва със символа `null`. Вмъкването на символи започва от мястото на символа `null`. В другия режим вмъкването започва от началото на низа.

Следващата програма преобразува число, записано като масив от символи, първо в шестнадесетичен формат, а след това в осмичен. `seekp()` се използва за позициониране в началото на масива преди второто вмъкване на стойност в него.

```

#include <iostream.h>
#include <sstream.h>

const bufLen = 25;
main() {
    char buf[bufLen];
    long lval = 1024;
    ostream oss(buf, sizeof(buf), ios::out);
    oss.setf( ios::showbase );

    oss << hex << lval << ends;
    cout << "buf: " << buf << endl;

    oss.seekp( ios::beg );
    oss << oct << lval << ends;
    cout << "buf: " << buf << endl;
}

```

След изпълнение на програмата получаваме следния резултат:

```

buf: 0x400
buf: 02000

```

Вторият пример преобразува число в низ. Програмата проверява първия и втория символ на числото, за да определи основата на бройната система, в която е записано то.

```
#include <iostream.h>
#include <sstream.h>
#include <string.h>

const bufLen = 25;
main() {
    char buf[bufLen];
    long lval = 1024;
    ostringstream oss(buf, sizeof(buf), ios::out);

    oss.setf( ios::showbase );
    oss << hex << lval << ends;

    cout << "lval: " << lval << endl;
    if ( strlen(buf) > 1 ) {
        switch ( buf[0] ) {
            case '0':
                if ( buf[1] == 'x' || buf[1] == 'X' ) cout << "шестнадесетичен вид: ";
                else cout << "осмичен вид: ";
                break;
            default:
                cout << "десетичен вид: ";
        }
    }
    cout << "buf: " << buf << endl;
}
```

Обектите от клас `istream` се използват за извличане на символи от масив. За да се дефинира обект от този клас, трябва да се зададат два аргумента:

1. Указател от тип `char*` към предварително дефиниран масив от символи.
2. Размер на масива в байтове.

Позволено е дефиниране на анонимен обект от клас `istream`. Например, следващата програма преобразува символен низ в цяло число чрез анонимен обект от клас `istream`:

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

char s[] = "400"
main() {
    int hexVal;
    istringstream(s, sizeof(s)) >> hex >> hexVal;
    cout << "hexVal: " << hexVal << endl;
}
```

В последния пример се преобразува низ в число, записано в шестнадесетична, осмична и десетична бройна система чрез наименован обект от клас `istream`. Преди второто и третото извличане на символи от масива, трябва да се позиционираме отново в началото на масива.

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

```
#include <strstream.h>

char s[] = "400"
main() {
    int hexVal;
    int octVal;
    int decVal;

    istrstream iss(s,sizeof(s));
    iss >> hex >> hexVal;
    iss.seekg(ios::beg); iss >> oct >> octVal;
    iss.seekg(ios::beg); iss >> dec >> decVal;

    cout << "hexVal: " << hexVal << endl;
    cout << "octVal: " << octVal << endl;
    cout << "decVal: " << decVal << endl;
}
```

След компилиране и изпълнение на програмата получаваме:

```
hexVal: 1024
octVal: 256
decVal: 400
```

Строг контрол на типовете

За функциите от библиотека `iostream` се извършва строг контрол на типовете. Например, четене на информация чрез обект от клас `ostream` или извличане на стойности чрез обект от клас `ofstream` и предаването им на стандартния вход ще предизвикат грешка при компилация. Да разгледаме следните декларации:

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

extern istream& operator>>( istream&, Screen& );
extern void print( ostream& );
ifstream inFile;
```

Следващите оператори предизвикват грешка при компилация, поради смесване на типовете:

```
main() {
    Screen myScreen;

    // грешка: очаква аргумент от тип ostream&
    print( cin >> myScreen );

    // грешка: очаква операцията >>
    inFile << "грешка: очаква операция за вход";
}
```

A.9 Резюме

Входно-изходните операции в C++ се реализират от стандартна библиотека. В стандарт 2.0 на езика тази библиотека се нарича `iostream`. Нейните възможности бяха разгледани в това приложение.

Приложението не описва цялата библиотека — в частност то не показва как могат да се създадат от потребителя производни класове на съществуващи в библиотеката класове. Това не беше целта на направеното изложение. Авторът е предпочел да наблегне на основните възможности, които предоставя библиотеката, чието познаване е необходимо за реализиране на входно-изходните операции в програмите на C++.

Библиотека `iostream` е изградена на базата на множествена йерархия и виртуални основни класове. Кодовете на функциите от тази библиотека са прекрасно средство за обучение. Препоръчваме на тези от вас, които имат достъп до кодовете на библиотеката, да ги разгледат добре.

Приложение Б

Насоки в развитието на C++

C++ е нов език. През 1985 година се появи неговата първа работеща версия. Бързото разпространение на езика стимулира неговото развитие. През 1986 г. в дефиницията на класовете беше въведена секция `protected`. Позволено беше да се дефинират указатели към членове на клас. В началото на 1988 г. беше въведено експериментално множествено наследяване. Позволено беше да се дефинират за класове операциите `new` и `delete`.

Тъй като използването на C++ продължава да расте, сега се обмислят две нови насоки за развитие на езика:

1. Параметризирани типове.
2. Обработка на изключителни ситуации.

Параметризираните типове улесняват дефинирането на класове "контейнери", каквито са масивите и свързаните списъци. Освен това те позволяват да се създават широко приложими функции, като `sort()` и `max()`. Улеснява се и създаването на големи класове и библиотеки от алгоритми.

Обработката на изключителните ситуации трябва да осигури стандартен начин за управление на възможните аномалии в програмите — деление на нула, препълване при изпълнение на аритметични операции и работа с масиви, изчерпване на динамичната памет. Чрез този механизъм размерите и сложността на програмите ще се съкрати значително.

Теоретично всяка дефиниция на клас осигурява проверка за грешки, свързани с класа. Без стандартен механизъм за обработка на изключителните ситуации е трудно да се създадат и използват голям брой библиотеки за различни класове.

Трудността при реализиране на стандартен начин за обработка на изключителните ситуации се дължи на необходимостта от развитие на програмния стек. Синтаксисът, който реализира управлението на изключителните ситуации все още не е публикуван. Затова в това приложение тази тема не е разгледана подробно.

В приложението се представя публикуван експериментален начин за дефиниране на параметризирани типове. Следващите разглеждания не са напълно завършени и могат да се променят. Нещо повече: няма гаранция, че параметризираните типове или механизмът за управление на изключителните ситуации в бъдеще ще станат част от езика C++.

При описанието на параметризираните типове авторът се основава на своята дискусия с Bjarne Stroustrup и на следващата публикация:

Parameterized Types for C++, Bjarne Stroustrup, Proceedings of the Usenix C++ Conference, Denver, 1988.

Б.1 Параметризирани типове

Ако програмистът иска да създаде функция, която връща по-голямата от две стойности, той трябва да дефинира няколко функции с еднакво име за различните типове данни. При това алгоритъм, който определя по-голямата стойност в различните функции е един и същ. Това, което се променя е типът данни и вероятно смисълът на операцията ">>":

```

int max( int i, int j )
    { return( (i > j) ? i : j ); }
double max( double i, double j )
    { return( (i > j) ? i : j ); }
Complex max( Complex i, Complex j )
    { return( (i > j) ? i : j ); }
String max( String i, String j )
    { return( (i > j) ? i : j ); }

```

Ако типовете на аргументите и резултатът от функцията могат да се параметризират, ще бъде достатъчно да се дефинира само една функция `max()`. Компиляторът ще се грижи за извикване на функцията, която съответства на даден тип. Например:

```

// параметризирана дефиниция на max()
// Type замества конкретен тип
template <class Type> Type max( Type i, Type j ) {
    return( (i > j) ? i : j );
}

Complex a, b;
Complex c = max( a, b );    // изисква обръщение към max<Complex>
int i;
int j = max( 10, i );       // изисква обръщение към max<int>

```

Ползата от въвеждането на параметризирани типове е очевидна при параметризиране на класове. Ако програмистът иска да създаде масив, чийто елементи са от произволен тип, той трябва да дефинира различни класове за всеки възможен тип на елементите. Такива класове могат да бъдат `ArrayInt`, `ArrayComplex`, `ArrayDouble` и др. Параметризираните типове осигуряват по-добро решение:

```

// параметризирана дефиниция на клас Array
// Type замества конкретен тип
template <class Type> class Array {
public:
    Array( int );
    Type& operator[]( int );
private:
    Type *ap;
    int size;
};

```

Ето как се дефинира обект от клас `Array` с елементи от конкретен тип:

```

// дефиниция на обект от клас Array с елементи от тип int
Array<int> ia( 24 );

// дефиниция на обект от клас Array с елементи от клас BitVector
Array<BitVector> ba( SZ );

```

Дефинираните по този начин обекти `ia` и `ba` се използват както се използват обектите на непараметризирани класове. Следващият раздел разглежда подробно предложения синтаксис.

Б.2 Параметризирани класове

Параметризираните типове използват нова запазена дума `template`. Общата форма на параметризиран тип е

```
template <class Parameter_tag_name>
```

където `Parameter_tag_name` може да бъде произволен идентификатор. Параметризираната дефиниция на клас се различава от дефиницията на "обикновен" клас по един допълнителен ред от горния вид. Ето как изглежда дефиницията на параметризиран списък:

```
template <class PT> class List {
public:
    List( PT *pt = 0, List<PT> *lpt = 0 ) : val( pt ), next( lpt ) { }
    PT *append( List<PT>* );
private:
    PT *val;
    List<PT> *next;
};
```

`PT` е име на параметризиран тип, който замества конкретен тип. Например, потребителят може да дефинира списък, чийто елементи са цели числа или обекти от клас `String`.

`val` е указател към променлива от тип `PT`. Ако елементите на дефинирания списък са числа от тип `int`, типът на `val` е `int*`. Ако елементите на списъка са обекти от клас `String`, типът на `val` е `String*`. По същия начин типът на резултата на функцията `append()` зависи от типа на елементите на списъка.

Името на параметризирания клас `List` се състои от две имена `List` и `PT`. Името на параметризирания тип се огражда в скоби. `List<PT>` е името на параметризирания клас `List`. `lpt` е указател към друг параметризиран клас `List`. Ако се дефинира списък от цели числа, името на конкретния клас `List` е `List<int>`. Аналогично, ако се дефинира списък, чийто елементи са обекти от клас `String`, името на конкретния клас `List` е `List<String>`.

Понякога се налага да се дефинира функция-член на параметризиран клас, която се отнася за конкретен тип данни. В нея се използват специфични знания, които ускорят или оптимизират нейната реализация за конкретния тип данни. Общата дефиниция на функцията не може да осигури това. Ето как изглежда синтаксисът на функцията `append()`, дефинирана за списък, чийто елементи са числа от тип `int`:

```
// дефиниция на append() за конкретен тип елементи на списъка
int *List<int>::append( List<int> *pt )
{ /* ... */ }
```

Дефиницията на `List<PT>` не означава, че могат да се дефинират обекти от такъв клас. Параметризираният клас `List` е само едно общо описание на конкретен клас `List`. Следващата дефиниция е невъзможна:

```
List<PT> myList;    // грешка
```

Следващите две дефиниции са допустими, защото дефинират обекти от конкретен клас List:

```
List<String> slist, *Pslist;  
List<int> ilist, *Pilist;
```

Ето още няколко примера за правилни дефиниции на обекти от конкретен клас List:

```
List<BitVector> blist;  
List<double> dlist;  
List<String> stList;
```

Ако елементите на списъка са обекти от някакъв клас, този клас трябва да е дефиниран предварително.

С обект от конкретен клас List се работи като с обект от "обикновен" клас. Например:

```
Pilist—>append( &ilist );
```

Описанието typedef може да направи името на конкретен клас List по-естествено:

```
typedef List<Screen> ScreenList;  
typedef List<int> iList;  
  
ScreenList *ps;  
iList hdr;
```

Параметризирани класове и наследяване

Всеки параметризиран клас може да бъде произведен или основен клас в някаква наследствена йерархия. Например:

```
template <class Type>  
class SortedArray : public Array<Type>  
    { /* ... */ }
```

Следващите дефиниции дефинират обекти от производния клас:

```
// производният клас е SortedArray<String>  
// основният клас е Array<String>  
SortedArray<String> sa( 100 );  
  
// производният клас е SortedArray<int>  
// основният клас е Array<int>  
SortedArray<int> ia( ARRAY_SIZE );
```

Б.3 Параметризираны функции

Дефиницията на параметризирана функция започва с ключовата дума `template`, следвана от параметър, заграден в двойка ъглови скоби. Например:

```
template <class Type> Type& min( Array<Type>& a ) {  
    // връща най-малкия елемент на масива  
    Type smallest = a[ 0 ];  
    for ( int i = 1; i < a.size; ++i )  
        if ( a[i] < smallest ) smallest = a[i];  
    return smallest;  
}
```

`min()` е параметризирана функция. Тя връща най-малкия елемент на масив, чийто елементи са от тип `Type`. Нейният аргумент е обект от параметризирания клас `Array`.

Параметризираната функция `min()` замества множество функции с име `min()` за различен тип данни. Тя трябва да се извиква само за типове, за които е дефинирана операцията "`<<`". Това означава, че `Type` може да бъде `int`, `double` или `String`, но не може да бъде `BitVector` или `Screen` преди да се дефинира операцията "`<<`" за тези два класа.

Обръщението към параметризираната функция `min()` не се различава от обръщението към "обикновена" функция. Нека имаме следните дефиниции на обекти:

```
Array<int> ia( 24 );  
Array<String> sa( 1024 );  
Array<double> da( ARRAY_SIZE );
```

`min()` може да се извика по следния начин:

```
int i = min( ia );           // int& min(Array<int>);  
String s(min(sa));          // String& min(Array<String>);  
double d = sqrt(min(da));   // double& min(Array<double>);
```

За всяка конкретна стойност на `Type` може да се дефинира указател към параметризираната функция `min()`. Например:

```
int& (*pt)(Array<int>) = &min;
```

Дадена е следната декларация на параметризираната функция `print()`:

```
template <class Type> void print( Array<Type>&, Type& (*) (Array<Type>) );
```

Дефинираният по-горе указател `pf` може да се предаде на `print()` при обръщение към тази функция:

```
print( ia, pf )
```

Приложение В

Съвместимост между С и С++

Една от основните причини за бързото разпространение на езика С++ е лесното му използване съвместно с езика С. Между двата езика съществува съвместимост в две направления: първо, С++ може да се използва в системи, основани на С и второ, преходът от С към С++ се осъществява лесно.

Програмисти на С, които изучават С++ за първи път, могат да го използват като езика С с допълнителна възможност за строг контрол на типовете. Един ден те ще започнат да програмират действително на С++.

- Като начало може да се използва стандартната библиотека за вход/изход, вместо въведената в С++ библиотека `iostream`. За програмиста е по-просто да използва `stdio.h`.
- Вместо класовете в С++ може да се използват структурите в С. Следващите две декларации са еквивалентни в С++:

```
class foo_1 {
public:
    /* ... */
};

struct foo_2 {
    /* ... */
};
```

- Всички библиотеки в езика С могат да се използват от програма на С++, след като се включат необходимите заглавни файлове.

Програмистите на С++ използват по-рядко предпроцесора на С (`cpp`) — главно за условна компилация и включване на заглавни файлове. Въвеждането на `inline`-функции прави ненужно използването на макроси. Константните данни обезсмислят употребата на директивата `#define` за дефиниране на константи. Проблем при използване на `cpp` с езика С++ е невъзможността на предпроцесора да различи като коментар текста след двете наклонени черти `///
//`. Предпроцесорът на С интерпретира коментарите в С++ като част от израза, с който се замества името на макроса. Например,

```
#define BUFSIZ 1024 // стойност по подразбиране за дължина на буфера
```

замества `BUFSIZ` със израза

```
1024 // стойност по подразбиране за дължина на буфера
```

Ако по стар навик използвате директивата `#define`, записвайте коментарите, така както е възприето в езика С:

```
#define BUFSIZ 1024 /* стойност по подразбиране за дължина на буфера */
```

Основните разлики между C++ и C, въведени в езика C++ са:

- Възможност за дефиниране на функции с еднакви имена.
- Предаване на аргументи чрез адрес (pass-by-reference), в допълнение на стандартното предаване по стойност в езика C.
- Операции за заемане и освобождаване на динамична памет (new и delete), вместо директно обръщение към malloc() и free().
- Дефиниране на абстрактни типове данни (класове) със следните възможности:
 1. Скриване на информация.
 2. Автоматично инициализиране.
 3. Дефиниране на операции с обектите на класа.
 4. Дефиниране на потребителски преобразувания.
- Обектно-ориентирано програмиране, основано на динамично свързване и наследяване.

В.1 Прототип на функция в C++

Езикът C++ използва като основа езика C, описан от Kernighan и Ritchie. Повечето от промените в C, свързани със стандарта ANSI C, се поддържат в C++. Всяка програма, написана върху общо подмножество на езиците C и C++, се интерпретира еднакво от компилаторите на двата езика. Повечето програми, написани на ANSI C също се възприемат като програми на C++. Най-забележимата разлика между оригиналния език C и C++ се изразява в декларирането на външни функции. Следващият пример показва декларация на една и съща функция в C и C++:

```
extern min();           // в C
extern min( int*, int ); // в C++
```

В C++ декларацията на външна функция задължително включва типа на всеки аргумент. Следователно, декларацията съдържа т.нар. прототип на функция. Прототипът на функция включва име на функцията, тип на резултата и списък от аргументи. Като използва прототипа на функцията, компилаторът открива грешка в следващите обръщения към min(). В езика C и двете обръщения се приемат от компилатора. Откриването на грешката е грижа на програмиста:

```
int i, j, ia[10];
main() {
    min( i,j );    // типът на първия аргумент не съответства
    min( ia );    // липсва втори аргумент
}
```

При преобразуване на програма, написана на езика C, в програма на C++ всички декларации на външни функции трябва да се преобразуват. (ANSI C поддържа прототип на функции както в C++. Следователно, основната разлика между C и C++ не се проявява, при работа с компилатор за ANSI C.)

Разпространените системи на езика C++ осигуряват множество заглавни файлове, в които функциите са декларирани в съответствие с изискванията на C++. Например в заглавния файл `stdio.h`, който се разпространява със C++, функцията `printf()` се декларира така:

```
int printf(const char* ...)
```

Програма на C, която не използва стандартните заглавни файлове, за деклариране на стандартни функции няма да се компилира с компилатор на езика C++. Например:

```
struct node {
    int val;
    struct node *left, right;
};

struct node *get_node( val )
int val;
{ /* в C++ това означава, че malloc() няма аргументи */
    char *malloc();
    struct node *ptr;

    /* проверката дали malloc() не връща 0 е пропусната съзнателно */
    ptr = ( struct node * )malloc(sizeof(*ptr) );
    ptr->val = val;
    ptr->left = ptr->right = 0;
    return ptr;
}

main() {
    struct node *p = get_node( 1024 );
    /* ... останалата част от програмата ... */
}
```

Тази програма няма да се компилира на езика C++, заради явната декларация на `malloc()` в тялото на функцията `get_node()`. `malloc()` е декларирана с празен списък от аргументи, а на езика C++ това означава, че тя няма аргументи. Тъй като в обръщението към функцията се използва един аргумент, компилаторът ще открие грешка. Проблемът може да се реши, ако локалната декларация на `malloc()` се замени с директива, която включва стандартния заглавен файл `malloc.h`. Тогава програмата ще се компилира и на двата езика.

Следващата програма е еквивалентна на горната, но тя е написана на езика C++. Този вариант на програмата е по-кратък и по-ясен.

```
/* еквивалентна програма на C++
 * операцията new замества обръщението към malloc()
 * конструкторът на клас Node замества get_node()
 */
class Node {
public:
    Node( int i = 0 ) {
        left = right = 0;
        val = i;
    }
    int val;
    Node *left, *right;
};
```



```
main() {
    Node *p = new Node( 1024 );
    /* ... останалата част от програмата ... */
}
```

C++ и ANSI C интерпретират различно празния списък от аргументи. В ANSI C празният списък означава, че функцията има нула или повече аргументи. Ако използваме декларацията

```
char *malloc();
```

следващите неправилни обръщения към malloc() няма да се забележат от компилатора на ANSI C:

```
char ac[] = "any old string";

/* следващите две обръщения са неправилни */
malloc( ac );
malloc();
```

В C++ празният списък от аргументи означава, че функцията няма аргументи. В ANSI C липсата на аргументи се означава с ключовата дума void.

```
extern ff( void );    // ff() е функция без аргументи
```

В C++ ключовата дума void в списъка от аргументи също означава липса на аргументи. Програми, които ще се компилират с компилатори на C++ и ANSI C, трябва да използват ключовата дума void в списъка от аргументи.

Програмистите на C могат да се освободят от строгия контрол на типовете, ако използват специалната сигнатура "..." (ellipses) вместо празния списък от аргументи. Следващата програма не може да се компилира от компилатора на C++, заради несъответствието между декларацията и обръщението към maxVal():

```
extern maxVal();
main() {
    int ia[ 10 ];
    int max;
    /* ... */
    max = maxVal( ia, 10 );
}
```

Ако променим декларацията на maxVal() по следния начин

```
extern maxVal( ... );
```

програмата ще се компилира и на C++, благодарение на подтиснатия контрол на типовете. В общия случай е по-добре да се създаде заглавен файл, в който функциите са декларирани със съответния списък от аргументи.

В.2 Разлики между C и C++

ANSI C и C++ се различават по сравнително малко характеристики. В предишния раздел беше посочена разликата при интерпретиране на празен списък от аргументи в декларацията на функция. Строгий контрол на типовете и по-голямата функционалност на C++ пораждаят относително малки разлики между двата езика. Този раздел е посветен именно на тези разлики.

Разлики, породени от по-голямата функционалност на C++

Ще изброим последователно всички разлики между C++ и ANSI C, които са предизвикани от по-голямата функционалност на езика C++.

- C++ притежава допълнителни запазени думи:

catch	new	public
class	operator	template
delete	overload	this
friend	private	try
inline	protected	virtual

- В C++ типът на литералните символни константи е `char`, докато в C техният тип е `int`. Например, за компютър с четири байтова дума

```
sizeof( 'a' );
```

връща 1 в C++ и 4 в C. Това различие в C++ е въведено заради функциите с едно и също име, имащи аргумент от тип `char`.

- В C++ името на локална структура покрива обект, функция или променлива със същото име от ограждащата област на действие. Например:

```
int ia[ iaSize ];
void fooBar() {
    struct ia { int i; char a; };

    // в C++ се връща размерът на локалната структура
    // в ANSI C се връща размерът на ::ia
    int sz = sizeof( ia );
    int ln = sizeof( ::ia );    // еквивалентен запис на C++
}
```

Тази разлика е предизвикана от това, че в C++ имената на структури се тълкуват като идентификатори на променливи.

- В C++ елементите на изброим тип са локални за областта на структурата, в която са дефинирани. В ANSI C структурите не поддържат собствена област на действие. Например,

```

struct iosream {
    enum { bad, fail, good };
    /* ... */
};

// в C++ няма грешка
// в ANSI C предефиниране на идентификатор в една и съща област на действие
double fail;

```

- В C++ оператор `goto` не може да прескача декларация с явно или неявно (чрез конструктор) инициализиране, освен ако тя се намира в блок и целият блок се прескочи. Това ограничение не съществува в ANSI C.
- В C++ декларациите на променливи без ключовата дума `extern` се считат за дефиниции на тези променливи. Неколкократно дефиниране на една и съща променлива е недопустимо. В ANSI C много дефиниции на една и съща променлива се разглеждат като единствена дефиниция на тази променлива. Например:

```

struct stat { /* ... */ };

// в ANSI C няма грешка
// в C++ има грешка: срещат се две дефиниции на stat
struct stat stat;
struct stat stat;

```

- В C++ по подразбиране константите са статични данни (`static`). В ANSI C константите са външни (`extern`) данни. Явната обява на константите като `static` или `extern` осигурява желаната семантика:

```

class Boolean { /* ... */ };

extern const Boolean TRUE(1);    // по подразбиране в ANSI C
static const Boolean FALSE(0);   // по подразбиране в C++

```

Разлики, породени от строгия контрол на типовете в C++

Ще изброим основните разлики между C++ и ANSI C, които се дължат на строгия контрол на типовете в C++:

- В C++ не може да се извика недеклаирана функция. В C това е позволено. Например:

```

main() {
    // в C++ има грешка: printf() не е декларирана
    // в C няма грешка: по подразбиране int print()
    printf(3.14159);
}

```

- В C++ указател от тип `void*` може да се присвои на указател от произволен друг тип само чрез явно преобразуване, защото такова преобразуване се смята за опасно. В ANSI C преобразуването се извършва неявно. Например:

```

main() {
    int i = 1024;
    void *pv = &i;

    // в C++ има грешка: необходимо е явно преобразуване
    // в ANSI C всичко е наред: явно преобразуване не е необходимо
    char *pc = pv;
    int len = strlen( pc );    // негоден указател
}

```

- C++ и ANSI C интерпретират различно константен низ, който се присвоява на масив от символи. В C++ следващата инициализация е недопустима, защото няма място за символа null, с който завършва всеки низ:

```

// в C++ има грешка
// в ANSI C няма грешка
char ch[2] = "hi";

```

В ANSI C инициализирането с този низ е съкратен запис на инициализирането на елементите на масива със символите 'h' и 'i'. Символът null се поставя в масива, само ако има място за него. Следващият по-общ начин за инициализиране създава масив от три елемента и в двата езика:

```
char ch[] = "hi";
```

Ако програмата ще се компилира на двата езика, трябва да се използва по-общия запис.

В.3 Свързване на модули, написани на С и C++

Програмистите на С могат да се освободят от строгия контрол на типовете в C++, като заменят празния списък от аргументи със специалната сигнатура "... " (ellipses):

```

struct node *get_node( val );
int val;
{
    /* освобождаване от строгия контрол на типовете */
    char *malloc( ... );
}

```

С тази поправка програмата ще се компилира от компилатора на C++, но няма да се свърже успешно в изпълнима програма. Причина за това е вътрешното кодиране на имената на всички функции в C++. Програмистът трябва да освободи явно името на дадена функция от вътрешно кодиране. За целта се използва директивата за свързване (виж раздел 4.5):

```
extern "C" char *malloc( unsigned );
```

Директивата за свързване показва на компилатора да не кодира името на дадена функция. Сега malloc() ще се свърже със стандартната библиотека на езика С. Препоръчва се предварително да се включи заглавният файл malloc.h.

Ако програмистът иска да създаде програма, която да се изпълни както на C, така и на C++, може да използва макроса `__cplusplus`:

```
#include <stdio.h>
#ifdef __cplusplus
    extern "C" char *malloc(long);
#endif

struct node *get_node( val );
int val;
{
    struct node *ptr;
#ifdef __cplusplus
    char *malloc(long);
#endif
    /* ... */
}
```

Макросът `__cplusplus` позволява да се комбинират двата стила на деклариране на функции, като се използват условните директиви на предпроцесора `#ifdef` и `#ifndef`.⁹

В.4 Преминаване от C на C++

Важен фактор за бързото разпространение на езика C++ е неговото лесно използване съвместно с езика C. Съвместимостта между двата езика се разглежда в две направления:

1. Възможност за използване на C++ със съществуващи системи, основани на езика C.
2. Лесното усвояване и ефективно прилагане на новият език от програмистите на C.

Обикновено начинаещите в програмирането на C++ програмират основно на C, а C++ използват само за да осигурят нови функционални възможности на съществуващи програми. По-долу са изброени основните принципи при съвместно използване на двата езика, които трябва да се знаят и помнят:

- Запазване на ясна и понятна връзка между частите, написани на двата езика.
- Няма смисъл да се преобразува съществуваща система, ако целта не е да се развият допълнителни функционални възможности.
- Новите възможности на C++ трябва да се използват умерено. Ако този принцип не се спазва може да се получи т.нар. код "спагети", т.е. заплетен код.
- Ефективното използване на C++ идва от ефективната структура на класовете. Ще имате голяма полза ако обмислите добре структурата на основните за вашето приложение класове или се доверите на човек с по-голям опит в конструирането на класове.

⁹ Преди стандарт 2.0 на езика C++, името на макроса е `c_plusplus`. То беше променено на `__cplusplus` за съвместимост с ANSI C. В стандарт 2.0 се поддържат и двете макродефиниции.

Приложение Г

Съвместимост със стандарт 1.2

В това приложение ще бъдат разгледани подробно разликите между стандарт 1.2 и 2.0 на езика C++. То представлява интерес за читатели, които нямат достъп до стандарт 2.0 или имат програми за стандарт 1.2 и искат да ги трансформират за стандарт 2.0.

Разликите между двата стандарта могат да се групират в две категории:

1. Разширения на езика, въведени в стандарт 2.0. Тук се включват такива характеристики като множествено наследяване, виртуални основни класове, механизъм за безопасно свързване, константни и статични функции-членове на клас. Разширение представлява и възможността да се дефинират операции за управление на динамичната памет (`new` и `delete`) за класове.
2. Различия в семантиката на отделни елементи в двата стандарта на езика. Например, алгоритъмът за търсене на съответствие при функции с еднакви имена не зависи от реда на деклариране на функциите; присвояването и инициализирането на обект от даден клас с друг обект от същия клас се извършва чрез копиране на членове, а не чрез побитово копиране.

Причините за тези разлики са няколко. Първо, подобренията целят да се улесни създаването на големи библиотеки. Параметризираните типове и обработката на изключителните ситуации от приложение Б, ако се въведат в бъдеще, също ще улеснят създаването на големи библиотеки. Второ, различията се дължат на стремежа да се отстранят съществуващи или вероятни проблеми в програми, написани на стандарт 1.2. Инициализирането и присвояването чрез копиране на членове са примери за такива промени.

Г.1 Характеристики, които не се поддържат в стандарт 1.2

Следващият списък съдържа новите характеристики на езика C++, които се поддържат само в стандарт 2.0:

- Множествено наследяване.
- Виртуални основни класове.
- Чисти виртуални функции.
- "Безопасно" свързване.
- Статични и константни функции-членове на клас.
- Явно инициализиране на статични членове.
- Възможност за дефиниране на операциите `new` и `delete` за класове.
- Възможност за дефиниране от потребителя на операциите `—>` и `..`
- Освобождаване на обекти от даден клас чрез явно обръщение към деструктора на класа.

В следващите няколко подраздела ще разгледаме как отсъствието на тези характеристики се отразява на програмите, написани на стандарт 1.2.

Множествено наследяване

Седма и осма глава разглеждат създаването на производни класове и механизма за наследяване. Ако читателят не е добре запознат с материала от тези две глави, по-добре най-напред да разгледа тях.

В стандарт 1.2 всеки клас може да произлиза само от един основен клас. Освен това правилата за деклариране и инициализиране на основен клас са по-ограничителни от тези в стандарт 2.0.

1. Потребителят не може да декларира явно основния клас като `private`. В стандарт 1.2 това се осъществява неявно по следния начин:

```
class Bear : ZooAnimal {}; // допуска се в двата стандарта на езика
```

В стандарт 2.0 същото нещо може да се реализира явно:

```
class Bear : private ZooAnimal {}; // допуска се само в стандарт 2.0
```

Използването на ключовата дума `private` премахва възможността за неправилно интерпретиране на атрибута на основния клас.

2. Името на основния клас не може да се използва явно в списъка за инициализация на членове в конструктора на производния клас. В стандарт 1.2 основният клас се инициализира по следния начин:

```
class Bear : public ZooAnimal { /* ... */ };  
  
// name е член на клас Bear  
Bear::Bear( char *s, short ZooArea )  
: (ZooArea), name(s) {} // допуска се в двата стандарта на езика
```

Списъкът от аргументи, който няма име, се отнася за основния клас. В стандарт 2.0 името на основния клас може да се посочи явно:

```
Bear::Bear( char *s, short ZooArea )  
: ZooAnimal(ZooArea), name(s) {} // допуска се само в стандарт 2.0
```

3. В стандарт 1.2 основният клас може да се инициализира с обект от същия клас, само ако класът притежава конструктор от вида `X(const X&)`. В стандарт 2.0 не съществува такова ограничение. Например:

```
class ZooAnimal {  
public:  
    ZooAnimal( short );  
    // ...  
};  
  
class Bear : public ZooAnimal { /* ... */ };  
  
Bear::Bear( short ZooArea )  
: (ZooArea) {} // допуска се в двата стандарта на езика
```

```

Bear::Bear( ZooAnimal& z )
    : ZooAnimal( z ) {}          // допуска се само в стандарт 2.0

Bear::Bear( const Bear& b )
    : ZooAnimal( b ) {}          // допуска се само в стандарт 2.0

```

4. Стандарт 1.2 не поддържа виртуални основни класове. Присъствието на ключовата дума `virtual` в списъка на основните класове е недопустимо:

```

class Base { ... };

// в стандарт 2.0 така се декларира виртуален основен клас
// в стандарт 1.2 декларацията е недопустима
class Derived : virtual public Base { ... };

```

5. В стандарт 1.2 не могат да се дефинират чисти виртуални функции. Абстрактните класове, като `ZooAnimal`, трябва да притежават дефиниция на всяка виртуална функция. Например:

```

class ZooAnimal {
public:
    // в стандарт 2.0 може да има чисти виртуални функции
    virtual void draw() = 0;

    // в стандарт 1.2 виртуалните функции трябва да се дефинират
    virtual void draw() {};
};

```

Освен посочените разлики и възможността за множествено наследяване, останалата част от разглежданията в седма и осма глава остава в сила и за стандарт 1.2.

Свързване с модули, написани на други езици

Този подраздел разглежда свързването с модули, написани на други езици в стандарт 1.2 на езика C++. Посочени са част от причините за въвеждане на "безопасно" свързване в стандарт 2.0. Свързването с други езици има връзка с поддържания в C++ механизъм за създаване на функции с еднакви имена. Читателят трябва да е запознат с материала в раздел 4.3 преди да продължи да чете по-нататък. В раздел 4.5 се обсъжда реализирането на "безопасно" свързване в стандарт 2.0.

Създаването на функции с еднакви имена представлява лексическо удобство за програмиста, но създава проблеми на свързващия редактор, който не допуска дублиране на имена. Ако например свързващият редактор открие няколко дефиниции на идентификатора `print`, той ще прекрати работа и ще изведе съобщение за грешка. В стандарт 1.2 този проблем се решава като всички функции с изключение на една от функциите с еднакво име получават уникално вътрешно име.

Ако една от функциите с еднакво име е библиотечна функция от друг език за програмиране, например C, възниква проблем със свързването. Ако например клас `Complex` притежава функция с име `abs()`, как може да се получи достъп до функцията `abs()` от библиотеката `math.lib` на езика C? Преди стандарт 2.0 този въпрос се решава на два етапа:

1. За да се дефинира функцията, чието име съвпада с името на съществуваща функция, трябва да се използва ключовата дума `override`:

```
override abs;
```

2. Първата от множеството функции с еднакво име не получава уникално вътрешно име. Следователно, за да дефинира функцията, чието име съвпада с името на функцията, написана на езика C, програмистът трябва да се увери, че функцията на C е първа. Например:

```
override abs;

#include <math.h>
#include <complex.h>      // съдържа декларацията Complex& abs(Complex&);
BigNum& abs( BigNum& );
```

В стандарт 2.0 не са необходими тези две изисквания. За съвместимост с предишните версии се поддържа ключовата дума `override`, макар и без ефект.

Стандарт 1.2 позволява дефиниране на функцията със съществуващо вече име, само когато това име стои след ключовата дума `override`. При това `override` не бива да бъде след декларация на функцията с това име. В следващия програмен фрагмент посоченото изискване не е спазено:

```
#include <math.h>
override abs;
#include <complex.h>
```

След ключовата дума `override` може да стои списък от функции с еднакво име, ако всички те връщат резултат от един и същ тип. Например:

```
override double sqrt ( double ), sqrt( Complex );
override f( int ), f( double ), f( Screen );
```

Реализираният в стандарт 1.2 начин за създаване на функции с едно и също име притежава два недостатъка. Първият от тях е свързан с реда на деклариране на функциите. Ако функцията, написана на езика C, не се декларира най-напред, тя не може да се свърже с програмата. Вторият недостатък проличава при използването на няколко библиотеки. Създаването на библиотеки за отделните класове е основен принцип в C++. Тези библиотеки могат да се комбинират по различен начин в различни приложения. Следващият пример показва в какво се състои проблема. Нека `complex.h` съдържа следния фрагмент

```
override sqrt;
#include <math.h>
Complex sqrt( Complex );
```

а `3d.h` съдържа

```
overload atan;  
#include <math.h>
```

Ако програмистът запише

```
#include <complex.h>  
#include <3d.h>
```

операторът

```
overload atan;
```

ще предизвика грешка, защото чрез `complex.h` се включва заглавният файл `math.h`, който съдържа декларация на `atan()`. По-горе беше подчертано, че ключовата дума `overload` трябва да предшества декларациите на всички функции с името, посоченото след нея. Подобен проблем възниква, ако се разменят местата на заглавните файлове `complex.h` и `3d.h`. Сега проблеми създава функцията `sqrt()`. Този пример показва, че програмистът трябва да разгледа всички заглавни файлове, които включва, за да открие имената на функциите, които могат да се дублират. За да се избегнат грешките, често се създава нов заглавен файл:

```
// mylib.h  
overload sqrt;  
overload atan;  
#include <3d.h>  
#include <complex.h>
```

Ако програмистът иска да използва `mylib.h` в друго приложение, отново трябва да провери дали няма да възникне грешка.

В стандарт 2.0 свързването с други езици за програмиране се решава на три стъпки:

1. Всички функции получават уникално вътрешно име.
2. Създаването на функции с еднакви имена не изисква използване на ключовата дума `overload`. Всяка следваща декларация на функция с дадено име и различен списък от аргументи се смята за една от възможните функции с това име.
3. Свързването с други езици за програмиране се осъществява чрез общоприложим механизъм за освобождаване от вътрешно кодиране:

```
extern "C" void exit( int );  
extern "C" {  
    strlen( const char* );  
    char *strcpy( char*, const char* );  
}
```

Статични членове

В стандарт 1.2 не могат да се дефинират статични функции-членове на клас. Освен това не се допуска явно инициализиране на статични членове-данни. Следващият програмен фрагмент е създаден за стандарт 2.0. Той е неприложим за стандарт 1.2:

```

class X { public: X(int); };
class Y {
public:
    // в стандарт 2.0 може да се дефинира статична функция-член на клас
    // в стандарт 1.2 това не е позволено
    static getVal() { return val; }
private:
    static X xx;
    static int val;
};

// в стандарт 2.0 е позволено явното инициализиране на статичен член
// в стандарт 1.2 явното инициализиране е невъзможно
X Y::xx(1024);
int Y::val = 1;

```

Г.2 Характеристики, променени в стандарт 2.0

В следващия списък са изброени характеристиките на езика C++, които са променени в стандарт 2.0:

- Инициализирането чрез побитово копиране е заменено от инициализиране чрез копиране на членове. Същото важи и за присвояването.
- Ключовата дума `overload` не е необходима за създаване на функции с еднакви имена.
- Алгоритъмът за търсене на съответствие не зависи от реда на деклариране на функциите.
- Алгоритъмът за търсене на съответствие различава константен и "обикновен" указател. Същото важи за аргумент от тип `reference`. Той различава още тип `int` и целите типове с по-малък размер. Типовете `float` и `double` също се различават.
- Литералните символни константи са от тип `char`, а не от тип `int`.
- Променливите, които са дефинирани в глобалната област на действие, може да се инициализират не само с константни изрази.
- Функциите не могат да се използват преди да се декларират.
- Аргументите на функции могат да получат стойност по подразбиране само веднъж в даден файл.
- Елементите на изброим тип, дефиниран в областта на действие на даден клас, са локални за тази област на действие.
- Анонимните обединения, дефинирани в глобалната област на действия трябва да се декларират като статични.

Изброените модификации се разглеждат подробно в следващите подраздели.

Инициализиране и присвояване чрез побитово копиране

За да разберете този подраздел, трябва да познавате материала в пета и шеста глава. Особено важна е информацията от раздели 6.2 и 6.3 на глава 6.

Съществува случай, когато обект от даден клас не се инициализира от конструктор на класа. Това се случва, когато обектът се инициализира с друг обект от същия клас. Преди стандарт 2.0 инициализирането се извършваше чрез побитово копиране. В стандарт 2.0 се

използва копиране на членове. Разликата между двата начина на копиране е осезателна, когато се инициализира обект от произведен клас или обект, който притежава член-обект от друг клас. И в двата случая програмистът може да отмени служебния механизъм за инициализиране чрез копиране като дефинира специален конструктор от вида:

```
X::X(const X& );
```

Аналогично, стандартният механизъм за присвояване чрез копиране и в двата случая може да се отмени чрез дефиниране на операция от вида:

```
X& X::operator=( const X& );
```

Стандартният механизъм за инициализиране и присвояване чрез копиране трябва да се отмени и в двата случая, ако класът притежава член, който е указател.

Побитовото копиране на обект не разглежда обекта като съвкупност от членове. Извършва се копиране на битовете на единия обект в съответните битове на другия обект.

Да разгледаме един пример. Нека дефиницията на клас Word съдържа член-обект от клас String. Клас String притежава конструктор String::String(const String&). Той се грижи за инициализиране на неговия член от тип char*.

```
class Word {
public:
    Word(char* s) : name(s), occurs(0) {}
    Word(String& s) : name(s), occurs(0) {}
    // ...
private:
    String name;
    int occurs;
};

Word color( "blue" );
Word mood = color;
```

При побитово копиране color се копира изцяло. Това означава, че членът на клас String от тип char*, който отговаря на обекта mood, адресира същия масив от символи като съответния член, който отговаря на обекта color. Точно това, което конструкторът String::String(const String&) предотвратява, не може да се избегне при побитово копиране, защото този конструктор не се извиква.

Казаното по-горе остава вярно и ако String е основен клас за Word. При копиране на членове, ако основният клас или класовете на членовете-обекти притежават конструктори от вида X::X(const X&), тези конструктори се извикват.

Дефиниране на функции с едно и също име

В стандарт 1.2 могат да се дефинират функции с едно и също име, но техните декларации трябва да се предшества от оператор, включващ ключовата дума overload, следвана от името на функциите. Например:

```
overload sqrt;
#include <math.h>      // съдържа декларацията double sqrt(double);
```

```
Complex& sqrt( Complex& );
```

Следващият програмен фрагмент ще предизвика грешка, защото ключовата дума `overload` е след декларацията на `sqrt()` в `math.h`.

```
#include <math.h>
overload sqrt;
```

Стандарт 1.2 позволява по-кратък запис на спецификацията `overload`, ако функциите връщат резултат от един и същ тип:

```
overload void print( int ), print( char& ), print( double );
```

В стандарт 2.0 не е необходимо да се използва ключовата дума `overload`. Всеки две функции с еднакво име и различна сигнатура се смятат за две различни функции с едно и също име.

В стандарт 2.0 алгоритъмът за търсене на съответствие е променен. (Раздел 4.3 обсъжда подробно този алгоритъм.) Ще разгледаме всички разлики в алгоритъма, въведени в стандарт 2.0.

В стандарт 1.2 алгоритъмът за търсене на съответствие не прилага стандартно преобразуване, когато може да се получи съответствие чрез отрязване на дробната част на число с плаваща точка. В резултат на това се получава различно действие, в зависимост от това дали съществуват няколко функции с еднакво име или функцията е единствена. Освен това възниква несъвместимост с езика C. В модифицирания алгоритъм за търсене на съответствие отрязването на дробна част е позволено, но се извежда предупреждение. Например:

```
class X;

overload f;    // използваме заради стандарт 1.2
f( X& );
f( int );

// в стандарт 1.2 няма съответствие
// в стандарт 2.0 има съответствие с f(int), но се извежда предупреждение
f( 3.14159 );
```

В стандарт 1.2 алгоритъмът за търсене на съответствие не различава константен и "обикновен" указател. Същото се отнася и за аргумент от тип `reference`. Следователно, в стандарт 1.2 следващото обръщение е противоречиво. В стандарт 2.0 ще се извика функцията с константен аргумент.

```
const String str;
overload ff;
ff( const String& );
ff( String& );

// в стандарт 1.2 има грешка: противоречиво съответствие
// в стандарт 2.0 съответствие с ff( const String& )
ff( str );
```

В стандарт 1.2 алгоритмът за търсене на съответствие избира първата от две или повече функции, за които има съответствие чрез стандартно преобразуване. Този избор зависи от реда на деклариране на функциите. В модифицирания алгоритъм тази зависимост е отстранена. В такива случаи обръщението се счита за противоречиво. Да разгледаме един пример:

```
overload ff; // използваме заради стандарт 1.2
// от "a.h"
ff( double );

// от "b.h"
ff( char* );

// В стандарт 1.2, ако "a.h" е включен преди "b.h", получаваме
// съответствие с ff(double); иначе съответствие с ff(char*)
// В стандарт 2.0 обръщението е противоречиво
ff( 0 );
```

Противоречието в стандарт 2.0 може да се избегне чрез явно преобразуване:

```
ff( double(0) );
```

В стандарт 1.2 алгоритмът за търсене на съответствие счита типовете char, unsigned char, short и unsigned short за еквивалентни на тип int. Например,

```
overload ff; // използваме заради стандарт 1.2
ff( char );
ff( int );
```

дефинира две функции с име ff(). При обръщение към ff() с аргумент от тип char, unsigned char, short, unsigned short или int, се получава точно съответствие и с двете функции. (При деклариране на втората функция се извежда предупреждение.) Когато се получава точно съответствие с няколко функции, алгоритмът за търсене на съответствие избира първата от тях. Float и double също се считат за еквивалентни типове.

В модифицирания алгоритъм за търсене на съответствие се преминава към тип с по-голям размер, само когато няма формален аргумент със съответния размер. Това по-точно различаване на типовете може да попречи на изпълнението на програми, които са работили на стандарт 1.2. Например:

```
overload f; // използваме заради стандарт 1.2
void f(int);
void f(float);

void foo() {
    int a = 0;
    float b = 0.0;

    f(a); // съответствие с f(int)
    f(b); // съответствие с f(float)
    f(0); // съответствие с f(int)
```

```

// две обръщания с аргумент от тип double
f(0.0)      // в стандарт 1.2 няма грешка; в стандарт 2.0 има грешка
f(1.2)      // в стандарт 1.2 няма грешка; в стандарт 2.0 има грешка
}

```

В стандарт 1.2 може да се дефинира само една функция с аргумент от тип float или от тип double. Втората функция не се признава, защото float и double са еквивалентни типове. Обръщанията към f() с константи от тип double предизвикват точно съответствие с f(float).

В стандарт 2.0 алгоритъмът за търсене на съответствие различава двата типа float и double. Аргумент от тип double не предизвиква точно съответствие с формален аргумент от тип float. Обръщението

```
f(0.0)
```

изисква стандартно преобразуване за получаване на съответствие с f(float). Тъй като чрез стандартно преобразуване се получава съответствие и с функцията f(int), обръщението е противоречиво.

Как да преобразуваме този програмен фрагмент, за да няма противоречие?

- Може да дефинираме функция f(double). В стандарт 1.2 дефиницията на f(float) покрива двата типа аргументи. В стандарт 2.0 е необходима отделна дефиниция за всеки от тях.
- Може да преобразуваме явно константите от тип double в тип float:

```

f(0.0f)      // в стандарт 2.0 вече няма грешка
f(float(1.2)) // в стандарт 2.0 вече няма грешка

```

- Ако използваме не само числа от тип float, а произволни числа с плаваща точка, f(float) може да се замени с f(double). Тогава при обръщение към f() с аргумент float или double ще се получи точно съответствие с f(double).

Литерални символни константи

В стандарт 2.0 типът на литералните символни константи е char, а в стандарт 1.2 — int. Тази разлика предизвиква две несъответствия:

1. Размерът на символните константи в стандарт 2.0 е 1 байт. В стандарт 1.2 той е равен на размера на тип int за съответния компютър.
2. В стандарт 2.0 операцията "<<" извежда символните константи. Същата операция в стандарт 1.2 извежда представянето на символа като число от тип int. Например,

```

#include <stream.h>
main() {
    cout << 'a' << '\n';
}

```

в стандарт 1.2 извежда следния печат:

```
$a.cut
```

9710\$

В стандарт 2.0 се генерира такъв печат:

```
$a.cut  
a  
$
```

където с \$ е означен системния промпт.

В стандарт 1.2 символните константи се отпечатват с функцията `put()`:

```
main() {  
    cout.put('a').put('\n');  
}
```

Друга възможност на отпечатване на символни константи е заграждането им в кавички:

```
main() {  
    cout << "a" << "\n";  
}
```

Инициализиране на глобални променливи

В стандарт 1.2 идентификаторите, декларирани в глобалната област на действие, могат да се инициализират само с константни изрази (изрази, които могат да се пресметнат при компилация). В стандарт 2.0 тези идентификатори могат да се инициализират с произволен израз. Например:

```
#include <math.h>  
extern f(double)  
  
// в стандарт 2.0 няма грешка; в стандарт 1.2 има грешка  
double e = exp( 1 );  
int i = f( e );  
int j = i;
```

Ограничения, свързани с декларациите на функции

В стандарт 1.2 функциите могат да се извикват преди да се декларират. Тогава се смята, че те имат следния прототип:

```
int foo( ... );
```

Ще се изведе предупреждение за използване на недекларирана функция, но програмата ще се компилира.

Следващите две обръщания към `ff()` може да са коректни, но може и да не са. Компиляторът не може да гарантира правилно извикване на недекларирана функция. В стандарт 2.0 функциите могат да се използват само след като се декларират. Този програмен фрагмент ще предизвика грешка при компилация.

```
void f1() { double d = ff(); }  
void f2() { unsigned char ch = ff(1); }
```

Стойности по подразбиране за аргументите на функции

Правилата за задаване на стойности по подразбиране на аргументите на функция не са строго дефинирани в оригиналните ръководства за езика C++. В стандарт 2.0 аргументите на функция могат да получат стойност по подразбиране само веднъж в даден файл. В стандарт 1.2 аргументите могат да получат стойности по подразбиране няколко пъти в един и същи файл. Единственото изискване е тези стойности да се представят с константни изрази. Ето един пример:

```
int val;  
extern foo( int i = 0 );  
extern bar( int i = val );  
  
// допуска се в стандарт 1.2; грешка в стандарт 2.0  
foo( int i = 0 ) { return i*2; }  
  
// не се допуска и в двата стандарта  
bar( int i = val ) { return i*2; }
```

Обикновено аргументите получават стойности по подразбиране в дефинициите на класове или декларации на функции, поместени в някакъв заглавен файл.

Изброими типове, локални за даден клас

Поведението на изброимите типове, които са локални за даден клас, е дефинирано формално едва в стандарт 2.0. Елементите на такъв изброим тип принадлежат на областта на действие на класа. Те могат да бъдат `public`, `private` или `protected` членове на класа, в зависимост от това в коя секция се намира дефиницията на изброимия тип. Извън класа те са достъпни само чрез операцията за принадлежност към клас — `::`.

Преди стандарт 2.0 поведението на изброимите типове, които са локални за даден клас, зависеше от конкретната реализация на съответния стандарт на езика.

Анонимни обединения в глобалната област на действие

В стандарт 2.0 анонимните обединения в глобалната област на действие трябва да се дефинират като статични. Така се генерират вътрешни имена за елементите на това обединение. По този начин се избягва възможността да се получат грешки при разделна компилация.

```
// допуска се в стандарт 1.2;  
// в стандарт 2.0 не се допуска: по подразбиране обединението е extern  
union { int i; double d; };  
  
// допуска се в двата стандарта на езика
```

```
static union { int i; double d; };
```

Правила в стандарт 2.0, по-строги от правилата в стандарт 1.2

До края на този раздел ще се спрем на по-строгите правила, в стандарт 2.0 на езика C++. Те не са модификация на езика. Чрез тях се регулират някои по-свободни за интерпретиране елементи. По-строгите правила могат да променят действието на съществуващи програми, затова тяхното познаване би било от полза за програмиста.

Ниво на достъп за операции, които са членове на клас

В стандарт 1.2 нивото на достъп до операциите, които са членове на клас, не се следи. Това може да доведе до компилиране на програми, в които има грешка. Например:

```
class SmallInt {
public:
    SmallInt( int );
    // ...
private:
    operator+( int );
    int val;
};

SmallInt myInt( 127 );

main() {
    // в стандарт 1.2 грешката остава невидима
    // в стандарт 2.0 грешката се открива: operator+ е скрит член на класа
    int i = myInt + 1;
    // ...
}
```

Ако програмистът иска да има достъп до операцията `SmallInt::operator+(int)`, трябва да я премести в секция `public` на клас `SmallInt`.

Интерпретиране на константите

В стандарт 2.0 константите са статични. Ако е необходимо една и съща константа да се използва във всички файлове на програмата, тя трябва да се обяви явно като `extern`. Например:

```
extern const TRUE = 1;
```

В стандарт 1.2 константен указател и константен обект от даден клас се считат за външни променливи. В стандарт 2.0 тези изключения са премахнати. Да разгледаме един пример:

```
// в стандарт 1.2 обектът и указателят се считат за extern
// в стандарт 2.0 обектът и указателят се считат за статични
const X xObject( 2 );
char *const cp = "abc";

// в стандарт 1.2 обектът и указателят са външни (extern)
```

```
// в стандарт 2.0 обектът и указателят са външни (extern)
extern const X xObject( 2 );
extern char *const cp = "abc";
```

Забележете, че следващият оператор не дефинира константен указател, а указател към константен масив от символи. `pc` се счита за външна (`extern`) променлива:

```
const char *pc = "abc";
```

Константен аргумент от тип `reference`

В стандарт 1.2 проверката за константен аргумент не е пълна. Следващото обръщение към `bar()` не би трябвало да се изпълни, защото формалният аргумент на `bar()` не е константен:

```
extern bar( int& );

int foo( const int &r ) {
    // в стандарт 1.2 обръщението се изпълнява
    // в стандарт 2.0 се открива грешка: r е константен аргумент
    return bar( r );
}
```

Справочник

- [1] Carlo Ghezzi and Mehdi Jazayeri: Programming Language Concepts, John Wiley & Sons, NY, 1992.
- [2] Adele Goldberg and David Robson: SMALLTALK-80: The Language and Its Implementation, Addison-Wesley, Reading, MA, 1983.
- [3] Samuel P. Harbison and Guy L. Steele, Jr.: C: A Reference Manual, 2nd edition, Prentice-Hall, Englewood Cliffs, NJ, 1987.
- [4] Ralph Johnson and Brian Foote: Designing Reusable Classes, Journal of OOProgramming, June/July 1988, pp.22-35.
- [5] Brian W. Kernigan and Dennis M. Ritchie: The C Programming Language, 2nd edition, Prentice-Hall, Englewood Cliffs, NJ, 1988.
- [6] Andrew Koenig: C Traps and PitFalls, Addison-Wesley, Reading, MA, 1989.
- [7] Andrew Koenig: What is the C++, Anyway?, Journal of OOProgramming, April/May 1988, pp.48-52.
- [8] Andrew Koenig: An Example of Dynamic Binding in C++, Journal of OOProgramming, August/Sep. 1988, pp.60-62.
- [9] Stanley Lippman and Barbara Moo: C++: From Research to Practice, Proc. of the USENIX C++ Conference, Denver, CO, October 17-21, 1988, pp. 123-136.
- [10] Stanley Lippman and Bjarne Stroustrup: Pointers to Class Members in C++, Proc. of the USENIX C++ Conference, Denver, CO, October 17-21, 1988, pp. 305-326.
- [11] Stanley Lippman: What is this? The C++ Report, March 1989, pp. 6-7.
- [12] Barbara Liskov and John Guttag: Abstraktion and Specification in Programming Development, McGraw-Hill, NY, 1986.
- [13] Ravi Sethi: Programming Language Concepts and Constructs, Addison-Wesely, Reading, MA, 1989. (Two Chaptrs on C++).
- [14] Bjarne Stroustrup: The C++ Programming Language, Addison-Wesley, Reading, MA, 1986.
- [15] Bjarne Stroustrup: Multiple Inheritance for C++, Proc. EUUG Spring'87 Conference, Helsinki, May, 1987.
- [16] Bjarne Stroustrup:: What is OOProgramming?, Proc. of the USENIX C++ Workshop, Santa Fe, NM, Nov. 9-10, 1987, pp.159-180.
- [17] Bjarne Stroustrup: The Evolution of C++ 1985 to 1987, Proc. of the USENIX C++ Workshop, Santa Fe, NM, Nov. 9-10, 1987, pp. 1-21.
- [18] Bjarne Stroustrup: Possible Directions for C++, Proc. of the USENIX C++ Workshop, Santa Fe, NM, Nov. 9-10, 1987, pp. 399-416.
- [19] Bjarne Stroustrup: Type-safe Linkage for C++, Proc. of the USENIX C++ Conference, Denver, CO, Oct. 17-21, 1988, pp. 193-211.
- [20] Bjarne Stroustrup: Parameterized Types for C++, Proc. of the USENIX C++ Conference, Denver, CO, Oct. 17-21, 1988, pp. 1-18.

Бележки по отпечатването

David Beckdorff, Dag Bruck, John Eldridge, Jim Humelsine, Dave Jordan, Ami Klienman, Andy Koenig, Danny Lippman, Clovis Tondo и Steve Vinoski откриха множество грешки в предишното издание на тази книга. Благодаря за всичко това.

Стенли Липман е ученик на създателя на езика C++ Барни Строуструп. Неговата книга “Езикът C++ в примери” е издавана неколkokратно в САЩ. Тя е един добър учебник по програмиране. Примерите от тази книга са използвани като тестове при разработката на стандарт 2.0 на езика C++ от фирма AT&T Bell Laboratories.