Содержание

Предисловие 24

Введение 25

Новые концепции программирования 25
Объектно-ориентированное программирование 25
Унифицированный язык моделирования 26
Языки и платформы разработки 26
Для чего нужна эта книга 27
Новые концепции 27
Последовательность изложения материала 27
Знания, необходимые для чтения этой книги 28
Техническое и программное обеспечение 28
Консольные программы 28
Исходные тексты программ 28
Упражнения 29
Проще, чем кажется 29
Преподавателям 29
Стандартный C++ 29
Унифицированный язык моделирования (UML) 29
Средства разработки программного обеспечения 30
Различия между C и C++ 30
Оптимальный порядок изучения ООП 30
Нововведения в C++ 31
Избыточные возможности 31
Упражнения 31
От издательства 31

Глава 1. Общие сведения 32

Для чего нужно объектно-ориентированное программирование? 32
Процедурные языки 32
Деление на функции 33
Недостатки структурного программирования 33
Неконтролируемый доступ к данным 34
Моделирование реального мира 35
Объектно-ориентированный подход 36
Аналогия 37
ООП: подход к организации программы 38

Характеристики объектно-ориентированных языков 38
Объекты 38
Классы 39
Наследование 40
Повторное использование кода 42
Пользовательские типы данных 42
Полиморфизм и перегрузка 42
C++ и C 43
Изучение основ 44
Универсальный язык моделирования (UML) 44
Резюме 45
Вопросы 46

Глава 2. Основы программирования на C++ 48

Что необходимо для работы 49
Структура программы 49
Функции 49
Операторы 51
Разделяющие знаки 51
Вывод с использованием cout 52
Строковые константы 53
Директивы 53
Директивы препроцессора 54
Заголовочные файлы 54
Директива using 55
Комментарии 55
Синтаксис комментариев 55
Использование комментариев 56
Альтернативный вид комментариев 56
Переменные целого типа 56
Описание переменных целого типа 57
Объявление и определение переменной 58
Имена переменных 59
Операция присваивания 59
Целые константы 59
Оператор вывода 60
Манипулятор endl 60
Другие целые типы 61
Символьные переменные 61
Символьные константы 62
Инициализация 63
Управляющие последовательности 63
Ввод с помощью cin 64
Определение переменных при первом использовании 65
Каскадирование операции << 66
Выражения 66
Приоритеты выполнения операций 66
Вещественные типы 67
Тип float 67

Типы double и long double . . . 68
Вещественные константы . . . 68
Префикс const . . . 69
Директива #define . . . 70
Тип bool . . . 70
Манипулятор setw . . . 71
Каскадирование операции << . . . 72
Множественное определение . . . 72
Файл заголовка IOMANIP . . . 73
Таблица типов переменных . . . 73
Беззнаковые типы данных . . . 73
Преобразования типов . . . 75
Неявные преобразования типов . . . 75
Явные преобразования типов . . . 77
Арифметические операции . . . 78
Остаток от деления . . . 79
Арифметические операции с присваиванием . . . 79
Инкремент . . . 81
Декремент . . . 82
Библиотечные функции . . . 82
Заголовочные файлы . . . 83
Библиотечные файлы . . . 84
Заголовочные и библиотечные файлы . . . 84
Формы директивы #include . . . 84
Резюме . . . 85
Вопросы . . . 87
Упражнения . . . 88

Глава 3. Циклы и ветвления . . . 92

Операции отношения . . . 92
Циклы . . . 94
Цикл for . . . 94
Инициализирующее выражение . . . 96
Условие выполнения цикла . . . 96
Инкрементирующее выражение . . . 96
Число выполнений цикла . . . 97
Несколько операторов в теле цикла . . . 97
Блоки и область видимости переменных . . . 98
Форматирование и стиль оформления циклов . . . 99
Обнаружение ошибок . . . 100
Варианты цикла for . . . 100
Определение счетчика цикла внутри оператора цикла for . . . 101
Несколько инициализирующих выражений и условий цикла . . . 101
Цикл while . . . 102
Несколько операторов в цикле while . . . 104
Приоритеты арифметических операций и операций отношения . . . 105
Цикл do . . . 106
Выбор типа цикла . . . 108
Ветвления . . . 108

Условный оператор if . . . 109
Несколько операторов в теле if . . . 110
if внутри циклов . . . 110
Функция exit() . . . 111
Оператор if else . . . 112
Функция getche() . . . 113
Условия с присваиванием . . . 114
Вложенные ветвления if...else . . . 116
if и else во вложенных ветвлениях . . . 117
Конструкция else...if . . . 119
Оператор switch . . . 119
Оператор break . . . 121
switch и символьные переменные . . . 122
Ключевое слово default . . . 123
Сравнение switch и if...else . . . 124
Условная операция . . . 124
Логические операции . . . 127
Операция логического И . . . 127
Логическое ИЛИ . . . 128
Логическое НЕ . . . 129
Целые величины в качестве булевых . . . 129
Приоритеты операций C++ . . . 130
Другие операторы перехода . . . 131
Оператор break . . . 131
Расширенная таблица символов ASCII . . . 133
Оператор continue . . . 133
Оператор goto . . . 134
Резюме . . . 135
Вопросы . . . 136
Упражнения . . . 138

Глава 4. Структуры . . . 142

Структуры . . . 142
Простая структура . . . 143
Определение структуры . . . 143
Определение структурной переменной . . . 144
Доступ к полям структуры . . . 146
Другие возможности структур . . . 146
Пример применения структур . . . 148
Вложенные структуры . . . 150
Пример карточной игры . . . 154
Структуры и классы . . . 156
Перечисления . . . 156
Дни недели . . . 157
Перечисления и программа подсчета числа слов . . . 159
Пример карточной игры . . . 161
Недостаток перечислений . . . 162
Примеры перечисляемых типов . . . 163
Резюме . . . 163
Вопросы . . . 164
Упражнения . . . 165

Глава 5. Функции 168

Простые функции 169
Объявление функции 170
Вызов функции 171
Определение функции 171
Обычные и библиотечные функции 172
Отсутствие объявления 173
Передача аргументов в функцию 174
Передача констант в функцию 174
Передача значений переменных в функцию 175
Передача аргументов по значению 176
Структурные переменные в качестве аргументов 177
Имена переменных внутри прототипа функции 181
Значение, возвращаемое функцией 181
Оператор return 182
Исключение ненужных переменных 184
Структурная переменная в качестве возвращаемого значения 185
Ссылки на аргументы 186
Передача по ссылке аргументов стандартных типов 187
Усложненный вариант передачи по ссылке 189
Передача структурных переменных по ссылке 191
Замечание о ссылках 192
Перегруженные функции 192
Переменное число аргументов функции 193
Различные типы аргументов 195
Рекурсия 196
Встраиваемые функции 198
Аргументы по умолчанию 200
Область видимости и класс памяти 202
Локальные переменные 203
Глобальные переменные 205
Статические локальные переменные 207
Возвращение значения по ссылке 208
Вызов функции в качестве левого операнда операции присваивания 209
Зачем нужно возвращение по ссылке? 210
Константные аргументы функции 210
Резюме 212
Вопросы 213
Упражнения 215

Глава 6. Объекты и классы 217

Простой класс 217
Классы и объекты 218
Определение класса 219
Использование класса 222
Вызов методов класса 222
Объекты программы и объекты реального мира 224
Детали изделия в качестве объектов 224
Круги в качестве объектов 225

Класс как тип данных . . . 226
Конструкторы . . . 227
Пример со счетчиком . . . 228
Графический пример . . . 231
Объекты в качестве аргументов функций . . . 232
Объекты в качестве аргументов . . . 236
Конструктор копирования по умолчанию . . . 237
Объекты, возвращаемые функцией . . . 239
Аргументы и объекты . . . 240
Пример карточной игры . . . 242
Структуры и классы . . . 244
Классы, объекты и память . . . 245
Статические данные класса . . . 247
Применение статических полей класса . . . 247
Пример использования статических полей класса . . . 247
Раздельное объявление и определение полей класса . . . 248
const и классы . . . 249
Константные методы . . . 250
Константные объекты . . . 252
Зачем нужны классы? . . . 253
Резюме . . . 254
Вопросы . . . 255
Упражнения . . . 257

Глава 7. Массивы и строки . . . 261

Основы массивов . . . 262
Определение массивов . . . 263
Элементы массива . . . 263
Доступ к элементам массива . . . 263
Среднее арифметическое элементов массива . . . 264
Инициализация массива . . . 265
Многомерные массивы . . . 267
Передача массивов в функции . . . 271
Массивы структур . . . 273
Массивы как члены классов . . . 275
Массивы объектов . . . 278
Массивы интервалов . . . 278
Границы массива . . . 280
Доступ к объектам в массиве . . . 280
Массивы карт . . . 281
Строки . . . 284
Строковые переменные . . . 285
Считывание нескольких строк . . . 288
Копирование строк . . . 289
Копирование строк более простым способом . . . 290
Массивы строк . . . 291
Строки как члены классов . . . 292
Определенные пользователем типы строк . . . 294
Стандартный класс string языка C++ . . . 296

Определение объектов класса string и присваивание им значений 296
Ввод/вывод для объектов класса string 298
Поиск объектов класса string 299
Модификация объектов класса string 300
Сравнение объектов класса string 301
Доступ к символам в объектах класса string 302
Другие методы класса string 303
Резюме 304
Вопросы 304
Упражнения 307

Глава 8. Перегрузка операций 312

Перегрузка унарных операций 313
Ключевое слово operator 314
Аргументы операции 315
Значения, возвращаемые операцией 315
Временные безымянные объекты 317
Постфиксные операции 318
Перегрузка бинарных операций 320
Арифметические операции 320
Объединение строк 323
Множественная перегрузка 325
Операции сравнения 325
Операции арифметического присваивания 328
Операция индексации массива ([]) 330
Преобразование типов 334
Преобразования основных типов в основные типы 335
Преобразования объектов в основные типы и наоборот 336
Преобразования строк в объекты класса string и наоборот 338
Преобразования объектов классов в объекты других классов 340
Преобразования: когда что использовать 346
Диаграммы классов UML 346
Объединения 347
Направленность 347
«Подводные камни» перегрузки операций и преобразования типов 348
Использование похожих значений 348
Использование похожего синтаксиса 348
Показывайте ограничение 349
Избегайте неопределенности 349
Не все операции могут быть перегружены 349
Ключевые слова explicit и mutable 349
Предотвращение преобразования типов с помощью explicit 350
Изменение данных объекта, объявленных как const, используя ключевое слово mutable 351
Резюме 353
Вопросы 353
Упражнения 356

Глава 9. Наследование 361

Базовый и производный классы 362
Определение производного класса 364
Обобщение в диаграммах классов в UML 364
Доступ к базовому классу 365
Результат программы COUNTEN 366
Спецификатор доступа protected 366
Недостатки использования спецификатора protected 368
Неизменность базового класса 368
Разнообразие терминов 368
Конструкторы производного класса 368
Перегрузка функций 370
Какой из методов использовать? 372
Операция разрешения и перегрузка функций 372
Наследование в классе Distance 373
Применение программы ENGLEN 374
Конструкторы класса DistSign 375
Методы класса DistSign 375
В поддержку наследования 375
Иерархия классов 376
Абстрактный базовый класс 379
Конструкторы и функции 380
Наследование и графика 380
Общее и частное наследование 383
Комбинации доступа 383
Выбор спецификатора доступа 384
Уровни наследования 385
Множественное наследование 388
Методы классов и множественное наследование 389
Частное наследование в программе EMPMULT 393
Конструкторы при множественном наследовании 393
Конструкторы без аргументов 396
Конструктор со многими аргументами 396
Неопределенность при множественном наследовании 397
Включение: классы в классах 398
Включение в программе EMPCONT 399
Композиция: сложное включение 403
Роль наследования при разработке программ 403
Резюме 404
Вопросы 405
Упражнения 407

Глава 10. Указатели 411

Адреса и указатели 412
Операция получения адреса & 412
Переменные указатели 414
Недостатки синтаксиса 416
Указатели должны иметь значение 416

Доступ к переменной по указателю . . . 417
Указатель на void . . . 420
Указатели и массивы . . . 421
Указатели-константы и указатели-переменные . . . 423
Указатели и функции . . . 424
Передача простой переменной . . . 424
Передача массивов . . . 426
Сортировка элементов массива . . . 428
Расстановка с использованием указателей . . . 428
Сортировка методом пузырька . . . 430
Указатели на строки . . . 432
Указатели на строковые константы . . . 432
Строки как аргументы функций . . . 433
Копирование строк с использованием указателей . . . 434
Библиотека строковых функций . . . 434
Модификатор const и указатели . . . 435
Массивы указателей на строки . . . 436
Управление памятью: операции new и delete . . . 437
Операция new . . . 438
Операция delete . . . 439
Класс String с использованием операции new . . . 440
Указатели на объекты . . . 442
Ссылки на члены класса . . . 443
Другое применение операции new . . . 444
Массив указателей на объекты . . . 445
Действия программы . . . 446
Доступ к методам класса . . . 446
Связный список . . . 447
Цепочка указателей . . . 447
Добавление новых элементов в список . . . 449
Получение содержимого списка . . . 450
Классы, содержащие сами себя . . . 450
Пополнение примера LINKLIST . . . 451
Указатели на указатели . . . 451
Сортируем указатели . . . 453
Тип данных person** . . . 454
Сравнение строк . . . 454
Пример разбора строки . . . 455
Разбор арифметических выражений . . . 456
Программа PARSE . . . 457
Симулятор: лошадиные скачки . . . 459
Разработка лошадиных скачек . . . 460
Моделирование хода времени . . . 463
Уничтожение массива указателей на объекты . . . 463
Функция putch() . . . 464
Диаграммы UML . . . 464
Диаграмма состояний в UML . . . 465
Состояния . . . 466
Переходы . . . 466
От состояния к состоянию . . . 466

Отладка указателей 467
Резюме 467
Вопросы 469
Упражнения 471

Глава 11. Виртуальные функции 476

Виртуальные функции 476
Доступ к обычным методам через указатели 477
Доступ к виртуальным методам через указатели 479
Позднее связывание 481
Абстрактные классы и чистые виртуальные функции 481
Виртуальные функции и класс person 483
Виртуальные функции в графическом примере 485
Виртуальные деструкторы 488
Виртуальные базовые классы 489
Дружественные функции 491
Дружественные функции как мосты между классами 491
Ломая стены 492
Пример с английскими мерами длины 493
Дружественность и функциональная запись 496
Дружественные классы 499
Статические функции 500
Доступ к статическим функциям 501
Инициализация копирования и присваивания 502
Перегрузка оператора присваивания 503
Конструктор копирования 506
Объектные диаграммы UML 510
Эффективное использование памяти классом String 510
Указатель this 516
Доступ к компонентным данным через указатель this 517
Использование this для возврата значений 518
Исправленная программа STRIMEM 520
Динамическая информация о типах 523
Проверка типа класса с помощью dynamic_cast 523
Изменение типов указателей с помощью dynamic_cast 524
Оператор typeid 526
Резюме 527
Вопросы 528
Упражнения 531

Глава 12. Потоки и файлы 536

Потоковые классы 536
Преимущества потоков 537
Иерархия потоковых классов 537
Класс ios 539
Класс istream 542
Класс ostream 543
Классы iostream и _withassign 544
Предопределенные потоковые объекты 544

Ошибки потоков . . . 545
Биты статуса ошибки . . . 545
Ввод чисел . . . 546
Переизбыток символов . . . 547
Ввод при отсутствии данных . . . 547
Ввод строк и символов . . . 548
Отладка примера с английскими расстояниями . . . 548
Потоковый ввод/вывод дисковых файлов . . . 551
Форматированный файловый ввод/вывод . . . 551
Строки с пробелами . . . 554
Ввод/вывод символов . . . 555
Двоичный ввод/вывод . . . 557
Оператор reinterpret_cast . . . 558
Закрытие файлов . . . 558
Объектный ввод/вывод . . . 559
Совместимость структур данных . . . 560
Ввод/вывод множества объектов . . . 561
Биты режимов . . . 563
Указатели файлов . . . 564
Вычисление позиции . . . 564
Вычисление сдвига . . . 565
Функция tellg() . . . 567
Обработка ошибок файлового ввода/вывода . . . 567
Реагирование на ошибки . . . 567
Анализ ошибок . . . 568
Файловый ввод/вывод с помощью методов . . . 570
Как объекты записывают и читают сами себя . . . 570
Как классы записывают и читают сами себя . . . 572
Код типа объекта . . . 578
Перегрузка операторов извлечения и вставки . . . 581
Перегрузка cout и cin . . . 581
Перегрузка << и >> для файлов . . . 583
Память как поток . . . 585
Аргументы командной строки . . . 586
Вывод на печатающее устройство . . . 589
Резюме . . . 590
Вопросы . . . 591
Упражнения . . . 592

Глава 13. Многофайловые программы . . . 596

Причины использования многофайловых программ . . . 596
Библиотеки классов . . . 597
Организация и концептуализация . . . 598
Создание многофайловой программы . . . 598
Заголовочные файлы . . . 599
Директории . . . 599
Проекты . . . 600
Межфайловое взаимодействие . . . 600
Взаимодействие исходных файлов . . . 600

Заголовочные файлы . . . 605
Пространства имен . . . 609
Класс сверхбольших чисел . . . 613
Числа как строки . . . 613
Описатель класса . . . 614
Методы . . . 615
Прикладная программа . . . 617
Моделирование высотного лифта . . . 619
Работа программы ELEV . . . 620
Проектирование системы . . . 621
Листинг программы ELEV . . . 623
Диаграмма состояний для программы ELEV . . . 634
Резюме . . . 635
Вопросы . . . 636
Проекты . . . 638

Глава 14. Шаблоны и исключения . . . 640

Шаблоны функций . . . 640
Шаблон простой функции . . . 642
Шаблоны функций с несколькими аргументами . . . 644
Шаблоны классов . . . 647
Контекстозависимое имя класса . . . 651
Создание класса связных списков с помощью шаблонов . . . 653
Хранение пользовательских типов . . . 655
UML и шаблоны . . . 658
Исключения . . . 659
Для чего нужны исключения . . . 660
Синтаксис исключений . . . 661
Простой пример исключения . . . 662
Многократные исключения . . . 666
Исключения и класс Distance . . . 668
Исключения с аргументами . . . 670
Класс bad_alloc . . . 673
Размышления об исключениях . . . 674
Резюме . . . 675
Вопросы . . . 676
Упражнения . . . 678

Глава 15. Стандартная библиотека шаблонов (STL) . . . 681

Введение в STL . . . 682
Контейнеры . . . 682
Алгоритмы . . . 687
Итераторы . . . 688
Возможные проблемы с STL . . . 689
Алгоритмы . . . 690
Алгоритм find () . . . 690
Алгоритм count () . . . 691
Алгоритм sort() . . . 692
Алгоритм search() . . . 692

Алгоритм merge(). 693
Функциональные объекты 694
Пользовательские функции вместо функциональных объектов 695
Добавление _if к аргументам 696
Алгоритм for_each () 697
Алгоритм transform() 697
Последовательные контейнеры 698
Векторы 699
Списки 702
Итераторы 706
Итераторы как интеллектуальные указатели 706
Итераторы в качестве интерфейса 708
Соответствие алгоритмов контейнерам 710
Работа с итераторами 713
Специализированные итераторы 717
Адаптеры итераторов 717
Потоковые итераторы 720
Ассоциативные контейнеры 724
Множества и мультимножества 725
Отображения и мультиотображения 729
Ассоциативный массив 730
Хранение пользовательских объектов 731
Множество объектов person 732
Список объектов класса person 735
Функциональные объекты 738
Предопределенные функциональные объекты 739
Создание собственных функциональных объектов 741
Функциональные объекты и поведение контейнеров. 746
Резюме 746
Вопросы 747
Упражнения 749

Глава 16. Разработка объектно-ориентированного ПО . . 752

Эволюция процесса создания программного обеспечения 752
Процесс просиживания штанов 753
Каскадный процесс. 753
Объектно-ориентированное программирование 753
Современные подходы 754
Моделирование вариантов использования. 755
Действующие субъекты 755
Варианты использования. 756
Сценарии 756
Диаграммы вариантов использования 757
Описания вариантов использования 758
От вариантов использования к классам 758
Предметная область программирования. 759
Рукописные формы. 760
Допущения 762
Программа LANDLORD: стадия развития. 762
Действующие субъекты 762

Варианты использования . . . 762
Описание вариантов использования . . . 763
Сценарии . . . 765
Диаграммы действий UML . . . 765
От вариантов использования к классам . . . 766
Список существительных . . . 766
Уточнение списка . . . 767
Определение атрибутов . . . 768
От глаголов к сообщениям . . . 768
Диаграмма классов . . . 770
Диаграммы последовательностей . . . 770
Написание кода . . . 774
Заголовочный файл . . . 775
Исходные .cpp файлы . . . 780
Вынужденные упрощения . . . 789
Взаимодействие с программой . . . 789
Заключение . . . 791
Резюме . . . 791
Вопросы . . . 792
Проекты . . . 794

Приложение А. Таблица ASCII . . . 796

Приложение Б. Таблица приоритетов операций C++ . . . 803

Таблица приоритетов операций . . . 803
Зарезервированные слова . . . 803

Приложение В. Microsoft Visual C++ . . . 806

Элементы экрана . . . 806
Однофайловые программы . . . 807
Компоновка существующего файла . . . 807
Создание нового файла . . . 808
Ошибки . . . 808
Информация о типах в процессе исполнения (RTTI) . . . 808
Многофайловые программы . . . 809
Проекты и рабочие области . . . 809
Работа над проектом . . . 809
Сохранение, закрытие и открытие проектов . . . 810
Компиляция и компоновка . . . 811
Программы с консольной графикой . . . 811
Отладка программ . . . 811
Пошаговая трассировка . . . 812
Просмотр переменных . . . 812
Пошаговая трассировка функций . . . 812
Точки останова . . . 813

Приложение Г. Borland C++ Builder . . . 814

Запуск примеров в C++ Builder . . . 815
Очистка экрана . . . 815

Создание нового проекта . . . 815
Задание имени и сохранение проекта . . . 817
Работа с существующими файлами . . . 817
Компиляция, связывание и запуск программ . . . 818
Запуск программы в C++ Builder . . . 818
Запуск программы в MS DOS . . . 818
Предварительно скомпилированные заголовочные файлы . . . 818
Закрытие и открытие проектов . . . 818
Добавление заголовочного файла к проекту . . . 819
Создание нового заголовочного файла . . . 819
Редактирование заголовочного файла . . . 819
Определение местонахождения заголовочного файла . . . 819
Проекты с несколькими исходными файлами . . . 820
Создание дополнительных исходных файлов . . . 820
Добавление существующих исходных файлов . . . 820
Менеджер проектов . . . 821
Программы с консольной графикой . . . 821
Отладка . . . 822
Пошаговый прогон . . . 822
Просмотр переменных . . . 822
Пошаговая трассировка функций . . . 822
Точки останова . . . 823

Приложение Д. Упрощенный вариант консольной графики 824

Использование подпрограмм библиотеки консольной графики . . . 825
Функции библиотеки консольной графики . . . 825
Реализация функций консольной графики . . . 827
Компиляторы Microsoft . . . 827
Компиляторы Borland . . . 828
Листинги исходных кодов . . . 828

Приложение Е. Алгоритмы и методы STL . . . 836

Алгоритмы . . . 836
Методы . . . 843
Итераторы . . . 845

Приложение Ж. Ответы и решения . . . 847

Глава 1 . . . 847
Глава 2 . . . 847
Глава 3 . . . 849
Глава 4 . . . 853
Глава 5 . . . 855
Глава 6 . . . 859
Глава 7 . . . 862
Глава 8 . . . 866
Глава 9 . . . 871
Глава 10 . . . 876
Глава 11 . . . 881

Глава 12 886
Глава 13 889
Глава 14 890
Глава 15 894
Глава 16 898

Приложение З. Библиография 899

Углубленное изучение C++ 899
Основополагающие документы 900
UML 900
История C++ 901
И другое 901

Алфавитный указатель 902

Глава 6
Объекты и классы

- ✦ Простой класс
- ✦ Объекты C++ и физические объекты

Теперь мы приступим к изучению того раздела программирования, к которому так долго готовились: *объекты и классы*. Мы предварительно рассмотрели все, что нам будет необходимо: структуры, позволяющие группировать данные, и функции, объединяющие фрагменты программы под одним именем. В этой главе мы соединим эти два понятия и создадим новый элемент программы — класс. Мы начнем с создания самых простых классов, постепенно усложняя наши примеры. Вначале нам будет необходимо сосредоточить свое внимание на частностях, касающихся создания классов и объектов, однако в конце главы мы поговорим об общих аспектах объектно-ориентированного подхода к программированию.

В процессе чтения этой главы рекомендуется при необходимости возвращаться к материалу главы 1 «Общие сведения».

Простой класс

Наш первый пример содержит класс и два объекта этого класса. Несмотря на свою простоту, он демонстрирует синтаксис и основные черты классов C++. Листинг программы SMALLOBJ приведен ниже[1].

```
// smallobj.cpp
// демонстрирует простой небольшой объект
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////
class smallobj               // определение класса
{
  private:
    int somedata;            // поле класса
  public:
```

[1] Все исходные тексты, приведенные в книге, вы найдете по адресу http://www.piter.com/download.

```
    void setdata(int d)      // метод класса, изменяющий значение поля
        { somedata = d; }
    void showdata()          // метод класса, отображающий значение поля
      { cout << "Значение поля равно " << somedata << endl; }
};
////////////////////////////////////////////////////////
int main()
{
  smallobj s1, s2;           // определение двух объектов класса smallobj
  s1.setdata(1066);          // вызовы метода setdata()
  s2.setdata(1776);
  s1.showdata();             // вызовы метода showdata()
  s2.showdata();
  return 0;
}
```

Класс smallobj, определенный в этой программе, содержит одно поле данных и два метода. Методы обеспечивают доступ к полю данных класса. Первый из методов присваивает полю значение, а второй метод выводит это значение на экран (возможно, незнакомые термины привели вас в недоумение, но скоро мы раскроем их смысл).

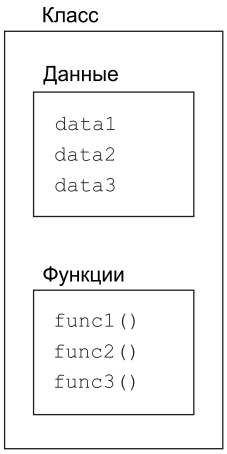

Рис. 6.1. Класс содержит данные и функции

Объединение данных и функций является стержневой идеей объектно-ориентированного программирования. Это проиллюстрировано на рис. 6.1.

Классы и объекты

В главе 1 мы говорили, что объект находится в таком же отношении к своему классу, в каком переменная находится по отношению к своему типу. Объект является экземпляром класса, так же, как автомобиль является экземпляром колесного средства передвижения. Класс smallobj определяется в начале программы SMALLOBJ. Позже, в функции main(), мы определяем два объекта s1 и s2, являющихся экземплярами класса smallobj.

Каждый из двух объектов имеет свое значение и способен выводить на экран это значение. Результат работы программы выглядит следующим образом:

```
Значение поля равно 1076 – вывод объекта s1
Значение поля равно 1776 – вывод объекта s2
```

Рассмотрим подробнее первую часть программы, где происходит определение класса smallobj. Затем мы обратимся к функции main(), в которой задействованы объекты класса smallobj.

Определение класса

Определение класса smallobj в приведенной выше программе выглядит следующим образом:

```
class smallobj              // определение класса
{
  private:
    int somedata;            // поле класса
  public:
    void setdata(int d)      // метод класса, изменяющий значение поля
      { somedata = d; }
    void showdata()          // метод класса, отображающий значение поля
      { cout << "Значение поля равно " << somedata << endl; }
};
```

Определение начинается с ключевого слова class, за которым следует имя класса; в данном случае этим именем является smallobj. Подобно структуре, тело класса заключено в фигурные скобки, после которых следует точка с запятой (;) (не забывайте ставить этот знак. Конструкции, связанные с типами данных, такие, как структуры и классы, требуют после своего тела наличия точки с запятой, в отличие от конструкций, связанных с передачей управления, например функций и циклов).

private и public

Тело класса содержит два не встречавшихся раньше ключевых слова: private и public. Сейчас мы раскроем их смысл.

Ключевой особенностью объектно-ориентированного программирования является возможность *сокрытия данных*. Этот термин понимается в том смысле, что данные заключены внутри класса и защищены от несанкционированного доступа функций, расположенных вне класса. Если необходимо защитить какие-либо данные, то их помещают внутрь класса с ключевым словом private. Такие данные доступны только внутри класса. Данные, описанные с ключевым словом public, напротив, доступны за пределами класса. Вышесказанное проиллюстрировано на рис. 6.2.

Зачем скрывать данные?

Не путайте сокрытие данных с техническими средствами, предназначенными для защиты баз данных. В последнем случае для обеспечения сохранности данных

можно, например, попросить пользователя ввести пароль перед тем, как разрешить ему доступ к базе данных. Пароль обеспечивает защиту базы данных от несанкционированного или злоумышленного изменения, а также копирования и чтения ее содержимого.

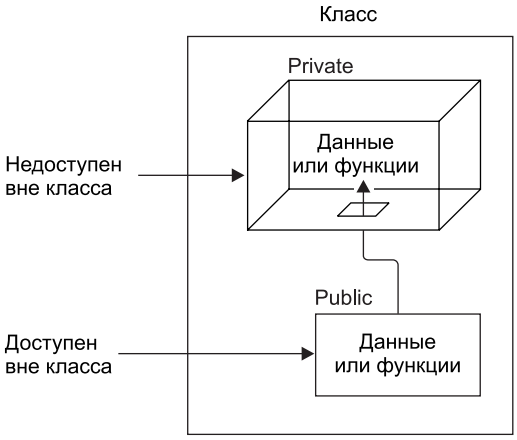

Рис. 6.2. Скрытые и общедоступные классы

Сокрытие данных в нашем толковании означает ограждение данных от тех частей программы, которые не имеют необходимости использовать эти данные. В более узком смысле это означает сокрытие данных одного класса от другого класса. Сокрытие данных позволяет уберечь опытных программистов от своих собственных ошибок. Программисты могут сами создать средства доступа к закрытым данным, что значительно снижает вероятность случайного или некорректного доступа к ним.

Данные класса

Класс smallobj содержит всего одно поле данных somedata, имеющее тип int. Данные, содержащиеся внутри класса, называют *данными-членами* или *полями класса*. Число полей класса, как и у структуры, теоретически может быть любым. Поскольку перед описанием поля somedata стоит ключевое слово private, это поле доступно только внутри класса.

Методы класса

Методы класса — это функции, входящие в состав класса. Класс smallobj содержит два метода: setdata() и showdata(). Тела обоих методов состоят из одного оператора, который записан на одной строке с фигурными скобками, ограничивающими тело функции. Разумеется, можно использовать и более традиционный способ оформления функций:

```
void setdata(int d)
{
  somedata = d;
}
```

и

```
void showdata()
{
  cout << "\n3начение поля равно " << somedata;
}
```

В тех случаях, когда тела методов невелики по размеру, имеет смысл использовать более сжатую форму их записи.

Поскольку методы setdata() и showdata() описаны с ключевым словом public, они доступны за пределами класса smallobj. Мы покажем, каким образом можно получить доступ к этим функциям, чуть позже. На рис. 6.3 показан синтаксис определения класса.

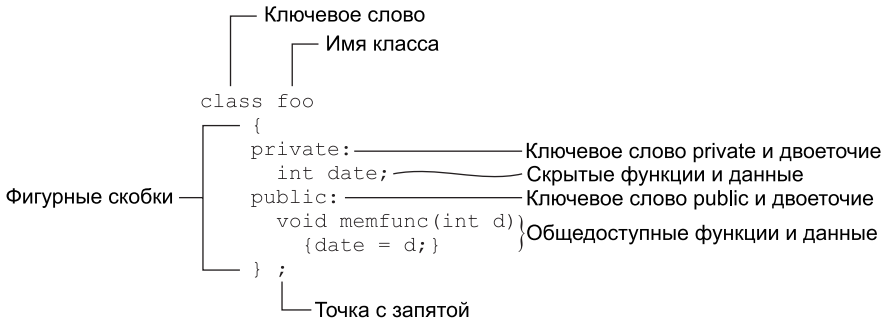

Рис. 6.3. Синтаксис определения класса

Сокрытие данных и доступность функций

Как правило, скрывая данные класса, его методы оставляют доступными. Это объясняется тем, что данные скрывают с целью избежать нежелательного внешнего воздействия на них, а функции, работающие с этими данными, должны обеспечивать взаимодействие между данными и внешней по отношению к классу частью программы. Тем не менее, не существует четкого правила, которое бы определяло, какие данные следует определять как private, а какие функции — как public. Вы можете столкнуться с ситуациями, когда вам будет необходимо скрывать функции и обеспечивать свободный доступ к данным класса.

Методы класса внутри определения класса

Методы класса smallobj выполняют действия, типичные для методов классов вообще: они считывают и присваивают значения полям класса. Метод setdata() принимает аргумент и присваивает полю somedata значение, равное значению аргумента. Метод showdata() отображает на экране значение поля somedata.

Обратите внимание на то, что функции setdata() и showdata() определены внутри класса, то есть код функции содержится непосредственно в определении класса (здесь определение функции не означает, что код функции помещается в память. Это происходит лишь при создании объекта класса). Методы класса, определенные подобным образом, по умолчанию являются встраиваемыми (встраиваемые функции обсуждались в главе 5 «Функции»). Позже мы увидим,

что функции внутри класса можно не только определять, но и объявлять, а определение функции производить в другом месте. Функция, определенная вне класса, по умолчанию уже не является встраиваемой.

Использование класса

Теперь, когда класс определен, давайте рассмотрим, каким образом его можно использовать в функции main(). Мы увидим, как определяются объекты и каким образом организован доступ к методам уже определенных объектов.

Определение объектов

Первым оператором функции main() является

```
smallobj s1, s2;
```

Этот оператор определяет два объекта s1 и s2 класса smallobj. Обратите внимание на то, что при определении класса smallobj не создаются никакие его объекты. Определение класса лишь задает вид будущего объекта, подобно тому, как определение структуры не выделяет память под структурные переменные, а лишь описывает их организацию. Все операции программа производит с объектами. Определение объекта похоже на определение переменной: оно означает выделение памяти, необходимой для хранения объекта.

Вызов методов класса

Следующая пара операторов осуществляет вызов метода setdata():

```
s1.setdata(1066);
s1.setdata(1776);
```

Эти операторы выглядят не так, как обычный вызов функции. Почему имена объектов s1 и s2 связаны с именами функций операцией точки (.)? Такой странный синтаксис объясняется тем, что вызов применим к методу конкретного объекта. Поскольку setdata() является методом класса smallobj, его вызов должен быть связан с объектом этого класса. Например, оператор

```
setdata(1066);
```

сам по себе не имеет смысла, потому что метод всегда производит действия с конкретным объектом, а не с классом в целом. Попытка доступа к классу по смыслу сходна попытке сесть за руль чертежа автомобиля. Кроме бессмысленности такого действия, компилятор расценил бы это как ошибку. Таким образом, доступ к методам класса возможен только через конкретный объект этого класса.

Для того чтобы получить доступ к методу класса, необходимо использовать операцию точки (.), связывающую метод с именем объекта. Синтаксически это напоминает доступ к полям структуры, но скобки позади имени метода говорят о том, что мы совершаем вызов функции, а не используем значение переменной (операцию точки называют *операцией доступа к члену класса*).

Оператор

```
s1.setdata(1066);
```

вызывает метод setdata() объекта s1. Метод присваивает полю somedata объекта s1 значение, равное 1066. Вызов

```
s2.setdata(1776);
```

подобным же образом присваивает полю somedata объекта s2 значение, равное 1776. Теперь мы имеем два объекта с различными значениями поля somedata, как показано на рис. 6.4.

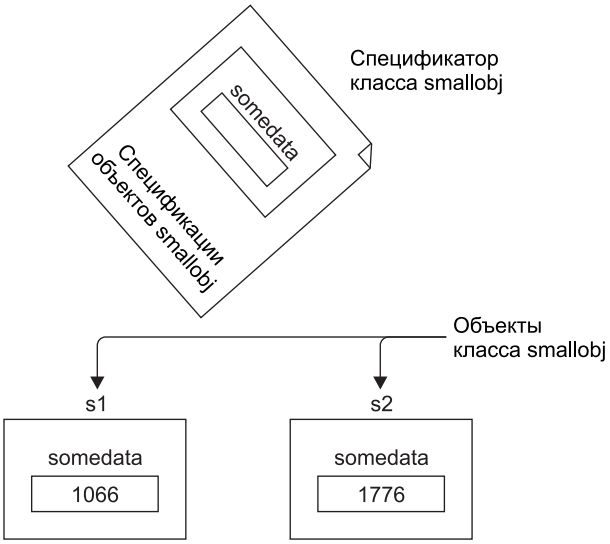

Рис. 6.4. Два объекта класса smallobj

Аналогично, два вызова функции showdata() отобразят на экране значения полей соответствующих объектов:

```
s1.showdata();
s2.showdata();
```

Сообщения

В некоторых объектно-ориентированных языках программирования вызовы методов объектов называют *сообщениями*. Так, например, вызов

```
s1.showdata();
```

можно рассматривать как посылку сообщения объекту s1 с указанием вывести на экран свои данные. Термин *сообщение* не входит в число формальных терминов языка C++, однако его полезно держать в голове в процессе дальнейшего обсуждения. Представление вызова методов в виде сообщений подчеркивает независимость объектов как самостоятельных единиц, взаимодействие с которыми осуществляется путем обращения к их методам. Если обратиться к аналогии со

структурой компании, приведенной в главе 1, то вызов метода будет похож на письмо к секретарю отдела продаж с запросом статистики о рынке компании в каком-либо регионе.

Объекты программы и объекты реального мира

Зачастую объекты, использующиеся в программе, представляют реальные физические объекты. В таких ситуациях проявляется взаимодействие между программой и реальным миром. Мы рассмотрим две подобные ситуации: детали изделия и рисование кругов.

Детали изделия в качестве объектов

Класс smallobj из предыдущего примера содержал только одно поле данных. Теперь мы рассмотрим более интересный пример. Мы создадим класс, основой для которого послужит структура, описывающая комплектующие изделия и ранее использовавшаяся в программе PARTS главы 4 «Структуры». Рассмотрим следующий пример — OBJPART:

```
// objpart.cpp
// детали изделия в качестве объектов
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////
class part                    // определение класса
{
  private:
  int modelnumber;            // номер изделия
  int partnumber;             // номер детали
  float cost;                 // стоимость детали
  public:
    // установка данных
    void setpart(int mn, int pn, float c)
   {
      modelnumber = mn;
      partnumber = pn;
      cost = c;
    }
    void showpart()           // вывод данных
    {
      cout << "Модель " << modelnumber;
      cout << ", деталь " << partnumber;
      cout << ", стоимость $" << cost << endl;
    }
};
////////////////////////////////////////////////////////////
int main()
{
  part part1;                 // определение объекта
                              // класса part
```

```
  part1.setpart(6244, 373, 217.55F); // вызов метода
  part1.showpart();                  // вызов метода
  return 0;
}
```

В этой программе используется класс part. В отличие от класса smallobj, класс part состоит из трех полей: modelnumber, partnumber и cost. Метод класса setpart() присваивает значения всем трем полям класса одновременно. Другой метод, showpart(), выводит на экран содержимое полей.

В примере создается единственный объект класса part с именем part1. Метод setpart() присваивает его полям значения соответственно 6244, 373 и 217.55. Затем метод showpart() выводит эти значения на экран. Результат работы программы выглядит следующим образом:

```
Модель 6244, деталь 373, цена $217.55
```

Этот пример уже ближе к реальной жизни, чем SMALLOBJ. Если бы вы разрабатывали инвентаризационную программу, то, вероятно, создали бы класс, аналогичный классу part. Мы привели пример объекта C++, моделирующего реально существующий объект — комплектующие изделия.

Круги в качестве объектов

В следующем нашем примере мы создадим объект, представляющий собой круг, отображающийся на вашем экране. Круг не столь материален, как деталь, которую легко можно подержать в руке, но тем не менее вы сможете увидеть его изображение, когда запустите программу.

Наша программа будет представлять собой объектно-ориентированную версию программы CIRCSTRC главы 5 (как и в программе CIRCSTRC, вам будет необходимо включить функции консольной графики в ваш проект. Эти файлы можно загрузить с сайта издательства, адрес которого указан во введении к данной книге. Описание файлов содержится в приложении Д «Упрощенный вариант консольной графики». Кроме того, полезную информацию можно также найти в приложениях к вашему компилятору). В программе будут созданы три круга с различными параметрами, а затем они появятся на экране. Ниже приведен листинг программы CIRCLES:

```
// circles.cpp
// круги в качестве объектов
#include "msoftcon.h"       // для функций консольной графики
////////////////////////////////////////////////////////////
class circle                //графический объект "круг"
{
  protected:
    int xCo,yCo;             // координаты центра
    int radius;
    color fillcolor;         // цвет
    fstyle fillstyle;        // стиль заполнения
  public:                    // установка атрибутов круга
    void set(int x, int y, int r, color fc, fstyle fs)
    {
```

```
      xCo = x;
      yCo = y;
      radius = r;
      fillcolor = fc;
      fillstyle = fs;
    }
    void draw()                    // рисование круга
    {
      set_color(fillcolor);        // установка цвета и
      set_fill_style(fillstyle);   // стиля заполнения
      draw_circle(xCo,yCo,radius); // рисование круга
    }
};
////////////////////////////////////////////////////////
int main()
{
  init_graphics();           // инициализация графики
  circle c1;                 // создание кругов
  circle c2;
  circle c3;
  // установка атрибутов кругов
  c1.set(15, 7, 5, cBLUE, X_FILL);
  c2.set(41, 12, 7, cRED, O_FILL);
  c3.set(65, 18, 4, cGREEN, MEDIUM_FILL);
  c1.draw();                 // рисование кругов
  c2.draw();
  c3.draw();
  set_cursor_pos(1,25);      // нижний левый угол
  return 0;
}
```

Результат работы программы CIRCLES такой же, как и для программы CIRCSTRC (см. рис. 5.5). Возможно, вам будет интересно сравнить две программы. В программе CIRCLES каждый из кругов представлен в виде объекта, а не совокупностью структуры и независимой от нее функции circ_draw(), как это было в программе CIRCSTRC. Обратите внимание, что в программе CIRCLES все, что имеет отношение к кругам, то есть соответствующие данные и функции, объединены в единое целое в определении класса.

Кроме функции draw(), класс circle содержит функцию set(), имеющую пять аргументов и задающую параметры круга. Как мы увидим позже, вместо функции set() лучше использовать конструктор.

Класс как тип данных

Здесь мы рассмотрим пример, демонстрирующий применение объектов C++ в качестве переменных типа, определенного пользователем. Объекты будут представлять расстояния, выраженные в английской системе мер, описанной в главе 4. Ниже приведен листинг программы ENGLOBJ:

```
// englobj.cpp
// длины в английской системе в качестве объектов
#include <iostream>
using namespace std;
```

```
////////////////////////////////////////////////////////
class Distance                      // длина в английской системе
{
  private:
    int feet;
    float inches;
  public:
    void setdist( int ft, float in ) // установка значений полей
      { feet = ft; inches = in; }
    void getdist()                  // ввод полей с клавиатуры
    {
      cout << "\nВведите число футов: "; cin >> feet;
      cout << "Введите число дюймов: "; cin >> inches;
    }
    void showdist()                 // вывод полей на экран
      { cout << feet << "\'-" << inches << '\"'; }
};
////////////////////////////////////////////////////////
int main()
{
  Distance dist1, dist2;            // две длины
  dist1.setdist(11,6.25);           // установка значений для d1
  dist2.getdist();                  // ввод значений для dist2
  // вывод длин на экран
  cout << "\ndist1 = "; dist1.showdist();
  cout << "\ndist2 = "; dist2.showdist();
  cout << endl;
  return 0;
}
```

В этой программе класс Distance содержит два поля: feet и inches. Он схож со структурой Distance, рассмотренной в главе 4, однако класс Distance имеет три метода: setdist(), предназначенный для задания значений полей объекта через передаваемые ему аргументы, getdist(), получающий эти же значения с клавиатуры, и showdist(), отображающий на экране расстояние в футах и дюймах.

Таким образом, значения полей объекта класса Distance могут быть заданы двумя способами. В функции main() мы определили две переменные типа Distance: dist1 и dist2. Значения полей для первой из них задаются с помощью функции setdist(), вызванной с аргументами 11 и 6.25, а значения полей переменной dist2 вводятся пользователем. Результат работы программы выглядит следующим образом:

```
Введите число футов: 10
Введите число дюймов: 4.75

dist1 = 11'-6.25" – задано аргументами программы
dist1 = 10'-4.75" – введено пользователем
```

Конструкторы

Пример ENGLOBJ демонстрирует два способа использования методов класса для инициализации полей объекта класса. Как правило, удобнее инициализировать поля объекта автоматически в момент его создания, а не явно вызывать в программе соответствующий метод. Такой способ инициализации реализуется с

помощью особого метода класса, называемого *конструктором*. Конструктор — это метод класса, выполняющийся автоматически в момент создания объекта.

Пример со счетчиком

В качестве примера мы создадим класс, объекты которого могут быть полезны практически для любой программы. Счетчик — это средство, предназначенное для хранения количественной меры какой-либо изменяющейся величины. Счетчик может хранить число обращений к файлу, число раз, которое пользователь нажал клавишу Enter, или количество клиентов банка. Как правило, при наступлении соответствующего события счетчик увеличивается на единицу (инкрементируется). Обращение к счетчику происходит, как правило, для того, чтобы узнать текущее значение той величины, для измерения которой он предназначен.

Допустим, что счетчик, который мы сейчас создадим, будет важной частью нашей программы, и многие из ее функций будут использовать значение этого счетчика. В процедурных языках, таких, как C, счетчик, скорее всего, был бы представлен в виде глобальной переменной. Но, как мы уже говорили в главе 1, использование глобальных переменных усложняет разработку программы и небезопасно с точки зрения несанкционированного доступа со стороны функций. Наш следующий пример, COUNTER, использует такой счетчик, значение которого может быть изменено только с помощью его собственных методов.

```
// counter.cpp
// счетчик в качестве объекта
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////
class Counter
{
  private:
    unsigned int count;         // значение счетчика
  public:
    Counter() : count(0)        // конструктор
      { /* пустое тело */ }
    void inc_count()            // инкрементирование счетчика
      { count++; }
    int get_count()             // получение значения счетчика
      { return count; }
};
////////////////////////////////////////////////////////////
int main()
{
  Counter c1, c2;               // определение с инициализацией
  cout <<"\nc1=" << c1.get_count(); // вывод
  cout <<"\nc2=" << c2.get_count();
  c1.inc_count();               // инкрементирование c1
  c2.inc_count();               // инкрементирование c2
  c2.inc_count();               // инкрементирование c2
  cout <<"\nc1=" << c1.get_count(); // вывод
  cout <<"\nc2=" << c2.get_count();
  cout << endl;
  return 0;
}
```

Класс Counter имеет единственное поле count типа unsigned int, поскольку значение счетчика не может быть отрицательным, и три метода: конструктор Counter(), который мы рассмотрим чуть позже, inc_count(), инкрементирующий поле count, и get_count(), возвращающий текущее значение счетчика.

Автоматическая инициализация

Когда создается объект типа Counter, нам хотелось бы, чтобы его поле count было инициализировано нулевым значением, поскольку большинство счетчиков начинают отсчет именно с нуля. Мы могли бы провести инициализацию с помощью вызова функции set_count() с аргументом, равным нулю, или создать специальный метод zero_count(), обнуляющий значение функции. Недостаток такого подхода заключается в том, что эти функции необходимо вызывать явно каждый раз при создании объекта типа Counter:

```
Counter c1;                 // при определении объекта
c1.zero_count();            // это необходимое действие
```

Подобные действия легко могут привести к неправильной работе всей программы, поскольку программисту для этого достаточно забыть проинициализировать хотя бы одну переменную после ее создания. Если в программе создается множество таких переменных, гораздо проще и надежнее было бы инициализировать их автоматически при создании. В нашем примере конструктор Counter() выполняет эти действия. Конструктор вызывается автоматически при создании каждого из объектов. Таким образом, в функции main() оператор

```
Counter c1, c2;
```

создает два объекта типа Counter. При создании каждого из них вызывается конструктор Counter(), присваивающий полю counter нулевое значение. Таким образом, кроме создания переменных, данный оператор еще присваивает их полям нулевое значение.

Имя конструктора

У конструкторов есть несколько особенностей, отличающих их от других методов класса. Во-первых, имя конструктора в точности совпадает с именем класса (в нашем примере таким именем является Counter). Таким образом, компилятор отличает конструкторы от других методов класса. Во-вторых, у конструкторов не существует возвращаемого значения. Это объясняется тем, что конструктор автоматически вызывается системой, и, следовательно, не существует вызывающей программы или функции, которой конструктор мог бы возвратить значение. Следовательно, указание возвращаемого значения для конструктора не имеет смысла. Отсутствие типа возвращаемого значения у конструкторов является вторым признаком, по которому компилятор может отличить их от других методов класса.

Список инициализации

Одной из наиболее часто возлагаемых на конструктор задач является инициализация полей объекта класса. Для каждого объекта класса Counter конструктор выполняет инициализацию поля count нулем. Вы, вероятно, ожидали, что это

действие будет произведено в теле конструктора приблизительно следующим образом:

```
count()
   { count = 0; }
```

Такая форма записи не рекомендуется, несмотря на то, что она не содержит ошибок. Инициализация в нашем примере происходит следующим образом:

```
count(): count(0)
{   }
```

Инициализация расположена между прототипом метода и телом функции и предварена двоеточием. Инициализирующее значение помещено в скобках после имени поля.

Если необходимо инициализировать сразу несколько полей класса, то значения разделяются запятыми, и в результате образуется *список инициализации*:

```
someClass() : m1(7), m2(33), m2(4)
{   }
```

Причины, по которым инициализация не проводится в теле конструктора, достаточно сложны. Инициализация полей с помощью списка инициализации происходит до начала исполнения тела конструктора, что в некоторых ситуациях бывает важно. Так, например, список инициализации — это единственный способ задать начальные значения констант и ссылок. В теле конструктора, как правило, производятся более сложные действия, чем обычная инициализация.

Результаты работы программы со счетчиком

В функции `main()` рассматриваемой нами программы создаются два объекта класса `Counter` с именами `c1` и `c2`. Затем на экран выводятся значения полей каждого из объектов, которые, согласно нашей задумке, должны быть инициализированы нулевыми значениями. Далее значение счетчика `c1` инкрементируется один раз, а значение счетчика `c2` — два раза, и программа вновь заставляет объекты вывести значения своих полей на экран (что является в данном случае вполне корректным). Результат работы программы выглядит следующим образом:

```
c1=0
c2=0
c1=1
c2=2
```

Для того чтобы убедиться в том, что конструктор функционирует именно так, как мы описали выше, заставим его печатать сообщение во время выполнения:

```
Counter() : count(0)
   { cout << "Конструктор\n"; }
```

Теперь результат работы программы будет выглядеть следующим образом:

```
Конструктор
Конструктор
c1=0
```

```
c2=0
c1=1
c2=2
```

Как можно видеть, конструктор исполняется дважды: первый раз — для переменной c1, второй раз — для переменной c2, во время выполнения оператора

```
Counter c1, c2;
в функции main().
```

Конструкторы и собственные типы данных

Разработчики компиляторов для языков C, VB или C++ должны позаботиться о том, чтобы для любой переменной стандартного типа, которую программист определяет в своей программе, вызывался необходимый конструктор. Например, если в программе встречается определение переменной типа int, где-то должен существовать конструктор, который выделит для этой переменной четыре байта памяти. Таким образом, научившись создавать свои собственные конструкторы, мы можем выполнять задачи, с которыми сталкиваются разработчики компиляторов. Мы сделали еще один шаг на пути к созданию собственных типов данных, в чем мы скоро убедимся.

Графический пример

Давайте модифицируем пример CIRCLES так, чтобы вместо функции set() использовался конструктор. Чтобы инициализировать пять полей класса, конструктору необходимо указать пять имен и пять соответствующих им значений в списке инициализации. Приведем листинг программы CIRCTOR:

```
// circtor.cpp
// графические объекты "круг" и конструкторы
#include "msoftcon.h "            // для функций консольной графики
////////////////////////////////////////////////////////
class circle                      // графический объект "круг"
{
  protected:
    int xCo,yCo;                  // координаты центра
    int radius;
    color fillcolor;              // цвет
    fstyle fillstyle;             // стиль заполнения
  public:
    // конструктор
    circle( int x, int y, int r, color fc, fstyle fs ):
    xCo(x), yCo(y), radius(r), fillcolor(fc), fillstyle(fs)
    { }
    void draw()                   // рисование круга
    {
      set_color(fillcolor);       // установка цвета и
      set_fill_style(fillstyle);  // стиля заполнения
      draw_circle(xCo,yCo,radius);// вывод круга на экран
    }
};
////////////////////////////////////////////////////////
```

```
int main()
{
  init_graphics();                // инициализация графики
  // создание кругов
  circle c1(15, 7, 5, cBLUE, X_FILL);
  circle c2(41, 12, 7, cRED, O_FILL);
  circle c3(65, 18, 4, cGREEN, MEDIUM_FILL);
  c1.draw();                      // рисование кругов
  c2.draw();
  c3.draw();
  set_cursor_pos(1,25);          // левый нижний угол
  return 0;
}
```

Данная программа схожа с программой CIRCLES, за исключением того, что функция set() замещена конструктором. Обратите внимание, как это изменение упростило функцию main(). Вместо двух операторов, используемых для создания объекта и для его инициализации, используется один оператор, совмещающий эти два действия.

Деструкторы

Как мы уже видели, особый метод класса — конструктор — вызывается при создании объекта. Как вы, возможно, догадались, существует другая функция, автоматически вызываемая при уничтожении объекта и называемая *деструктором*. Деструктор имеет имя, совпадающее с именем конструктора (а следовательно, и класса) и предваряющееся символом ~:

```
class Foo
   {
     private:
       int data;
     public:
       Foo() : data(0)       //конструктор (имя такое же, как у класса)
          { }
      ~Foo()                 //деструктор  (то же имя, но с символом ~)
         { }
};
```

Подобно конструкторам, деструкторы не возвращают значения и не имеют аргументов, поскольку невозможно уничтожение объектов несколькими способами. Наиболее распространенное применение деструкторов — освобождение памяти, выделенной конструктором при создании объекта. Более подробно мы рассмотрим эти действия в главе 10 «Указатели», а пока использование деструкторов не будет иметь для нас большого смысла.

Объекты в качестве аргументов функций

В следующей нашей программе мы внесем улучшения в пример ENGLOBJ, а также продемонстрируем несколько новых аспектов создания классов: перегрузку конструкторов, определение методов класса вне класса и, возможно, самое важное —

использование объектов в качестве аргументов функций. Рассмотрим программу ENGLCON:

```
// englcon.cpp
// constructors,adds objects using member function
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////
class Distance               // длина в английской системе
{
  private:
    int feet;
    float inches;
  public:                    // конструктор без аргументов
    Distance() : feet(0),inches(0.0)
      { }
    // конструктор с двумя аргументами
    Distance(int ft,float in) : feet(ft),inches(in)
      { }
    void getdist()           // ввод длины пользователем
    {
      cout << "\nВведите число футов: "; cin >> feet;
      cout << "Введите число дюймов: "; cin >> inches;
    }
    void showdist()          // вывод длины на экран
      { cout << feet << "\'-" << inches << '\"'; }
    void add_dist( Distance, Distance ); // прототип
};
//--------------------------------------------------------
// сложение длин d2 и d3
void Distance::add_dist(Distance d2, Distance d3)
{
  inches = d2.inches + d3.inches;        // сложение дюймов
  feet = 0;                              // с возможным заемом
  if(inches >= 12.0)                     // если число дюймов больше 12.0,
  {                                      // то уменьшаем число дюймов
    inches -= 12.0;                      // на 12.0 и увеличиваем
    feet++;                              // число футов на 1
  }
  feet +=d2.feet + d3.feet;             // сложение футов
}
////////////////////////////////////////////////////////
int main()
{
  Distance dist1, dist3;        // две длины
  Distance dist2(11, 6.25);     // определение и инициализация
  dist1.getdist();              // ввод dist1
  dist3.add_dist(dist1,dist2);  // dist3 = dist1 + dist2
  // вывод всех длин
  cout << "\ndist1 = "; dist1.showdist();
  cout << "\ndist2 = "; dist2.showdist();
  cout << "\ndist3 = "; dist3.showdist();
  cout << endl;
  return 0;
}
```

Основной блок этой программы начинается с присвоения начальных значений полям объекта dist2 класса Distance, после чего производится его сложение

с экземпляром dist1, инициализируемым пользователем. Затем на экран выводятся все три экземпляра класса Distance:

```
Введите число футов: 17
Введите число дюймов: 5.75
dist1 = 17'-5.75"
dist2 = 11'-6.25"
dist3 = 29'-0"
```

Рассмотрим теперь, как выполняются новые действия, включенные в программу.

Перегруженные конструкторы

Было бы удобно производить инициализацию переменных класса Distance в момент их создания, то есть использовать объявления типа

```
Distance width(5, 6.25);
```

где определяется объект width, сразу же инициализируемый значениями 5 и 6.25 для футов и дюймов соответственно.

Чтобы сделать это, вызовем конструктор следующим образом:

```
Distance(int ft, float in) : feet(ft), inches(in)
{ }
```

Мы инициализируем поля feet и inches теми значениями, которые передаются конструктору в качестве аргументов.

Тем не менее, мы хотим сохранить возможность определять переменные класса Distance без инициализации, подобно тому, как мы делали в программе ENGLOBJ.

```
Distance dist1, dist2;
```

В программе ENGLOBJ не было конструктора, но определения работали без ошибок. Почему же они работали без конструктора? Это объясняется тем, что компилятор автоматически встраивает в программу конструктор без параметров, который и создает переменные класса, несмотря на то, что явного определения конструктора мы не делали. Такой конструктор без параметров называется *конструктором по умолчанию*. Если бы конструктор по умолчанию не создавался автоматически, то мы не смогли бы определять переменные классов, в которых отсутствует конструктор.

Зачастую нам хотелось бы, чтобы начальные значения полям объекта присваивались также и в конструкторе без параметров. Если возложить эту функцию на конструктор по умолчанию, то мы не сможем узнать, какими значениями были инициализированы поля. Если же для нас важно, какими значениями будут инициализироваться поля объекта класса, то нам следует явно определить конструктор. В программе ENGLCON мы поступаем подобным образом:

```
Distance() : feet(0), inches(0.0)  //конструктор по умолчанию
     { }                           //тело функции пусто, никаких действий не производится
```

Члены класса инициализируются константными значениями, в данном случае целым значением 0 для переменной feet и вещественным значением 0.0 для переменной inches. Значит, мы можем использовать объекты, инициализиру-

емые с помощью конструктора без параметров, будучи уверенными в том, что поля объекта имеют нулевые, а не другие, случайные, значения.

Теперь у нас имеется два явно определенных конструктора с одним и тем же именем Distance(), и поэтому говорят, что конструктор является *перегруженным*. Какой из этих двух конструкторов исполняется во время создания нового объекта, зависит от того, сколько аргументов используется при вызове:

```
Distance length;            //вызывает первый конструктор
Distance width(11, 6.0);    //вызывает второй конструктор
```

Определение методов класса вне класса

До сих пор мы всегда определяли методы класса внутри самого класса. На самом деле это не является обязательным. В примере ENGLCON метод add_dist() определен вне класса Distance(). Внутри определения класса содержится лишь прототип функции add_dist():

```
void add_dist(Distance, Distance);
```

Такая форма означает, что функция является методом класса, однако ее определение следует искать не внутри определения класса, а где-то в другом месте листинга.

В примере ENGLCON функция add_dist() определена позже, чем класс Distance(). Ее код взят из программы ENGLSTRC главы 4:

```
void Distance::add_dist(Distance d2, Distance d3)
{
  inches = d2.inches + d3.inches;  // сложение дюймов
  feet = 0;                        // с возможным заемом
  if(inches >= 12.0)               // если число дюймов больше 12.0,
  {                                // то уменьшаем число дюймов
    inches -= 12.0;                // на 12.0 и увеличиваем
    feet++;                        // число футов на 1
  }
  feet += d2.feet + d3.feet;       // сложение футов
}
```

Заголовок функции содержит не встречавшиеся нам ранее синтаксические элементы. Перед именем функции add_dist() стоит имя класса Distance и новый символ ::. Этот символ является знаком *операции глобального разрешения*. Такая форма записи устанавливает взаимосвязь функции и класса, к которой относится эта функция. В данном случае запись Distance::add_list() означает, что функция add_dist() является методом класса Distance. Вышесказанное проиллюстрировано на рис. 6.5.

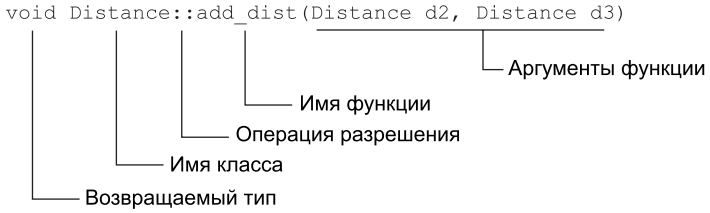

Рис. 6.5. Операция разрешения

Объекты в качестве аргументов

Рассмотрим, каким образом исполняется программа ENGLCON. Объекты dist1 и dist3 создаются при помощи конструктора по умолчанию (конструктора без аргументов), а объект dist2 — при помощи конструктора, принимающего два аргумента, значения которых инициализируют поля объекта dist2. Значения для инициализации объекта dist1 вводятся пользователем при помощи метода getdist().

Теперь мы хотим сложить значения dist1 и dist2 и присвоить результат объекту dist3. Это делается с помощью вызова функции

```
dist3.add_dist(dist1, dist2);
```

Величины, которые мы хотим сложить, передаются в качестве аргументов методу add_dist(). Синтаксис передачи объектов в функцию такой же, как и для переменных стандартных типов: на месте аргумента указывается имя объекта. Поскольку add_dist() является методом класса Distance, он имеет доступ к любым полям любого объекта класса Distance, используя операцию точки (.), например dist1.inches и dist2.feet.

Если внимательно изучить функцию add_dist(), то мы увидим несколько важных деталей, касающихся методов класса. Методу класса всегда предоставлен доступ к полям объекта, для которого он вызван: объект связан с методом операцией точки (.). Однако на самом деле методу класса доступны и другие объекты. Если задаться вопросом, к каким объектам имеет доступ метод add_dist(), то, взглянув на вызов

```
dist3.add_dist(dist1, dist2);
```

можно заметить, что, кроме объекта dist3, из которого был вызван метод add_dist(), он имеет доступ также и к объектам dist1 и dist2, поскольку они выступают в качестве его аргументов. Можно рассматривать объект dist3 как псевдоаргумент функции add_dist(): формально он не является аргументом, но функция имеет доступ к его полям. Смысл приведенного вызова можно сформулировать так: «выполнить метод add_dist() объекта dist3». Когда внутри функции происходит обращение к полям inches и feet, это означает, что на самом деле обращение происходит к полям dist3.inches и dist3.feet.

Обратите внимание на то, что функция не возвращает значения. Типом возвращаемого значения для функции add_dist() является void. Результат автоматически присваивается объекту dist3. На рис. 6.6 приведена иллюстрация сложения значений dist1 и dist2 с сохранением результата в переменной dist3.

Подводя итог вышесказанному, мы можем утверждать, что каждый вызов метода класса обязательно связан с конкретным объектом этого класса (исключением является вызов статической функции, как мы увидим позже). Метод может прямо обращаться по имени (feet и inches в данном примере) к любым, открытым и закрытым, членам этого объекта. Кроме того, метод имеет непрямой (через операцию точки, например dist1.inches и dist2.feet) доступ к членам других объектов своего класса; последние выступают в качестве аргументов метода.

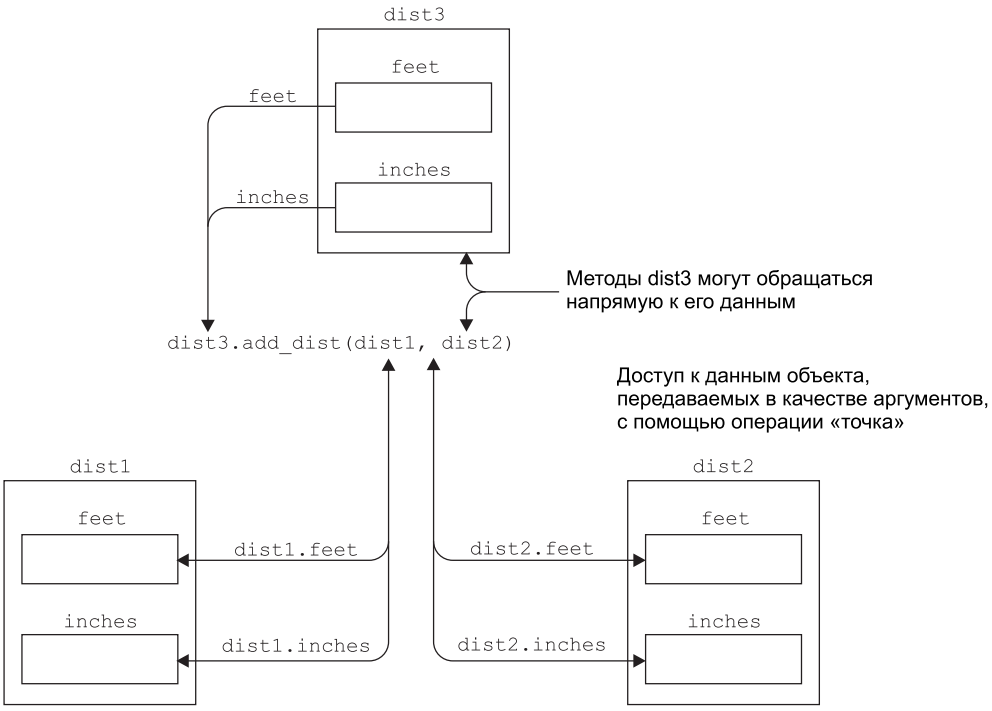

Рис. 6.6. Результат

Конструктор копирования по умолчанию

Мы рассмотрели два способа инициализации объектов. Конструктор без аргументов может инициализировать поля объекта константными значениями, а конструктор, имеющий хотя бы один аргумент, может инициализировать поля значениями, переданными ему в качестве аргументов. Теперь мы рассмотрим третий способ инициализации объекта, использующий значения полей уже существующего объекта. К вашему возможному удивлению, для этого не нужно самим создавать специальный конструктор, поскольку такой конструктор предоставляется компилятором для каждого создаваемого класса и называется *копирующим конструктором по умолчанию*. Копирующий конструктор имеет единственный аргумент, являющийся объектом того же класса, что и конструктор. Программа ECOPYCON демонстрирует использование копирующего конструктора по умолчанию:

```
// ecopycon.cpp
// инициализация объектов с помощью копирующего конструктора
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////
class Distance               // длина в английской системе
```

```
{
  private:
    int feet;
    float inches;
  public:
    // конструктор без аргументов
    Distance() : feet(0), inches(0.0)
      { }
    // конструктора с одним аргументом нет!
    // конструктор с двумя аргументами
    Distance(int ft, float in) : feet(ft), inches(in)
      { }
    void getdist()           // ввод длины пользователем
    {
      cout << "\nВведите число футов "; cin >> feet;
      cout << "Введите число дюймов: "; cin >> inches;
    }
    void showdist()          // вывод длины
      { cout << feet << "\'-" << inches << '\"'; }
};
//////////////////////////////////////////////////////////
int main()
{
  Distance dist1(11, 6.25);  // конструктор с двумя аргументами
  Distance dist2(dist1);     // два конструктора с одним аргументом
  Distance dist3 = dist1;
  // вывод всех длин
  cout <<"\ndist1 = "; dist1.showdist();
  cout <<"\ndist2 = "; dist2.showdist();
  cout <<"\ndist3 = "; dist3.showdist();
  cout << endl;
  return 0;
}
```

Мы инициализировали объект dist1 значением 11'-6.25" при помощи конструктора с двумя аргументами. Затем мы определяем еще два объекта класса Distance с именами dist2 и dist3, оба из которых инициализируются значением объекта dist1. Возможно, вам покажется, что в данном случае должен был вызваться конструктор с одним аргументом, но поскольку аргументом являлся объект того же класса, что и инициализируемые объекты, были предприняты иные действия. В обоих случаях был вызван копирующий конструктор по умолчанию. Объект dist2 инициализирован при помощи оператора

```
Distance dist2(dist1);
```

Действие копирующего конструктора по умолчанию сводится к копированию значений полей объекта dist1 в соответствующие поля объекта dist2. Как это ни удивительно, но идентичные действия для пары объектов dist1 и dist3 выполняются при помощи оператора

```
Distance dist3 = dist1;
```

Можно подумать, что данный оператор выполняет операцию присваивания, но это не так. Здесь, как и в предыдущем случае, вызывается конструктор копирования по умолчанию. Оба оператора выполняют одинаковые действия и рав-

ноправны в использовании. Результат работы программы выглядит следующим образом:

```
dist1 = 11'-6.25"
dist2 = 11'-6.25"
dist3 = 11'-6.25"
```

Видим, что объекты dist2 и dist3 действительно имеют те же значения, что и объект dist1. В главе 11 «Виртуальные функции» мы расскажем, каким образом можно создать свой собственный копирующий конструктор с помощью перегрузки копирующего конструктора по умолчанию.

Объекты, возвращаемые функцией

В примере ENGLCON мы видели, что объекты можно передавать в функцию в качестве аргументов. Теперь мы рассмотрим, каким образом функция может возвратить объект в вызывающую программу. Мы модифицируем пример ENGLCON и в результате получим программу ENGLRET:

```
// englret.cpp
// возвращение функцией значения типа Distance
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////
class Distance               // длина в английской системе
{
  private:
    int feet;
    float inches;
  public:                    // конструктор без аргументов
    Distance() : feet(0), inches(0.0)
      { }                    // конструктор с двумя аргументами
    Distance(int ft,float in) : feet(ft),inches(in)
      { }
    void getdist()           // ввод длины
    {
      cout << "\nВведите число футов: "; cin >> feet;
      cout << "Введите число дюймов: "; cin >> inches;
    }
    void showdist()          // вывод длины
      { cout << feet << "\'-" << inches << '\"'; }
    Distance add_dist(Distance);    // сложение
};
//--------------------------------------------------------
// сложение данного объекта с d2, возврат суммы
Distance Distance::add_dist(Distance d2)
{
  Distance temp;                    // временная переменная
  temp.inches = inches + d2.inches; // сложение дюймов
  if(temp.inches >= 12.0)           // если сумма больше 12.0,
  {                                 // то уменьшаем ее на
    temp.inches -= 12.0;            // 12.0 и увеличиваем
```

```
    temp.feet = 1;                  // число футов на 1
  }
  temp.feet += feet + d2.feet;      // сложение футов
  return temp;
}
//////////////////////////////////////////////////////////
int main()
{
  Distance dist1, dist3;            // две длины
  Distance dist2(11, 6.25);         // определение и инициализация dist2
  dist1.getdist();                  // ввод dist1 пользователем
  dist3 = dist1.add_dist(dist2);    // dist3 = dist1 + dist2
  // вывод всех длин
  cout << "\ndist1 = "; dist1.showdist();
  cout << "\ndist2 = "; dist2.showdist();
  cout << "\ndist3 = "; dist3.showdist();
  cout << endl;
  return 0;
}
```

Программа ENGLRET очень похожа на программу ENGLCON, однако различия между ними указывают на важные аспекты работы функций с объектами.

Аргументы и объекты

В примере ENGLCON два объекта были переданы в качестве аргументов в функцию add_dist(), а результат был сохранен в объекте dist3, методом которого и являлась вызванная функция add_dist(). В программе ENGLRET в качестве аргумента в функцию add_dist() передается лишь один аргумент: объект dist2. dist2 складывается с объектом dist1, к которому относится вызываемый метод add_dist(), результат возвращается в функцию main()и присваивается объекту dist3:

```
dist3 = dist1.add_dist(dist2);
```

Действия этого оператора аналогичны действиям соответствующего оператора программы ENGLCON, однако форма записи первого более естественна, поскольку использует операцию присваивания самым обычным образом. В главе 8 «Перегрузка операций» мы увидим, как с помощью арифметической операции + можно добиться еще более простой формы записи этих же действий:

```
dist3 = dist1 + dist2;
```

Функция add_dist() из примера ENGLRET выглядит так:

```
// сложение данного объекта с d2, возврат суммы
Distance Distance::add_dist(Distance d2)
{
  Distance temp;                    // временная переменная
  temp.inches = inches + d2.inches; // сложение дюймов
  if(temp.inches >= 12.0)           // если сумма больше 12.0,
  {                                 // то уменьшаем ее на
    temp.inches -= 12.0;            // 12.0 и увеличиваем
    temp.feet = 1;                  // число футов на 1
  }
```

```
  temp.feet += feet + d2.feet;      // сложение футов
  return temp;
}
```

Сравнив эту функцию с одноименной функцией программы ENGLCON, вы можете заметить несколько тонкостей, отличающих одну функцию от другой. В функции из программы ENGLRET создается временный объект класса Distance. Этот объект хранит значение вычисленной суммы до тех пор, пока она не будет возвращена вызывающей программе. Сумма вычисляется путем сложения двух объектов класса Distance. Первый из объектов — dist1, по отношению к которому функция add_dist() является методом; второй объект — dist2, переданный в функцию в качестве аргумента. Обращение к его полям из функции выглядит как d2.feet и d2.inches. Результат сложения хранится в объекте temp и обращение к его полям выглядит как temp.feet и temp.inches. Значение объекта temp возвращается в вызывающую программу с помощью оператора

```
return temp;
```

Вызывающая программа main() присваивает значение, возвращенное функцией, объекту dist3. Обратите внимание на то, что значение переменной dist1 не изменяется, а всего лишь используется функцией add_dist(). На рис. 6.7 показано, каким образом значение объекта возвращается в вызывающую программу.

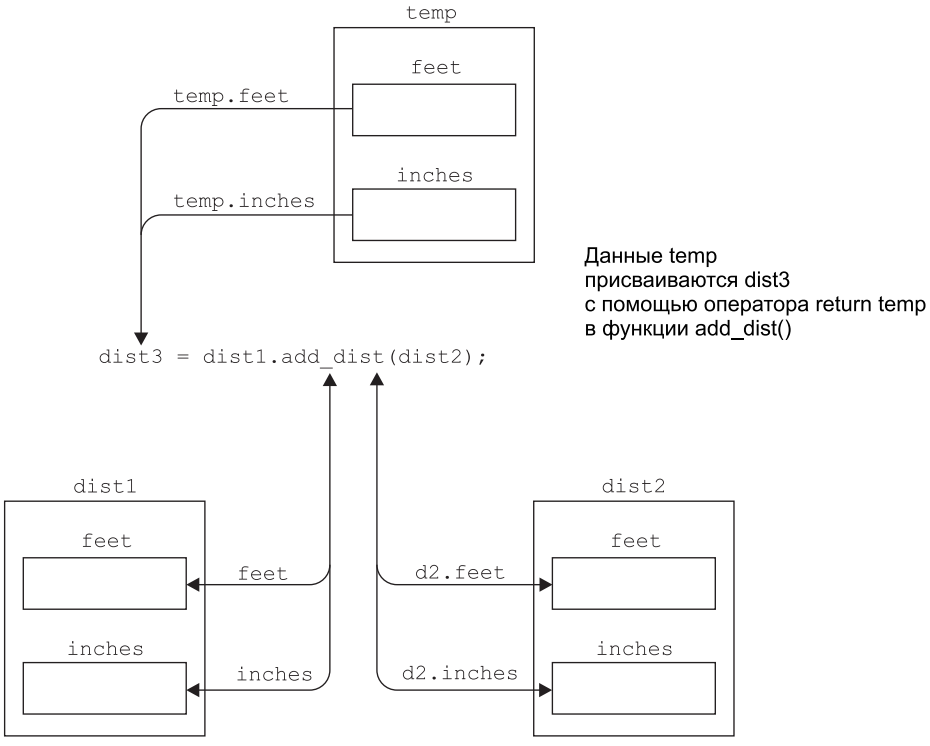

Рис. 6.7. Результат, возвращаемый из временного объекта

Пример карточной игры

В качестве примера моделирования объектами реально существующего мира рассмотрим модификацию программы CARDS, созданную в главе 4. В новой ее версии мы будем использовать объекты. Никаких дополнительных действий в программу мы вводить не будем, но сохраним практически все основные возможности старой версии.

Как и программа CARDS, ее версия CARDOBJ создает три карты фиксированной масти и достоинства, «перемешивает» их, раскладывает и предлагает вам угадать их очередность. Разница между программами состоит лишь в том, что каждая карта представлена объектом класса card.

```
// cardobj.cpp
// игральные карты в качестве объектов
#include <iostream>
using namespace std;
enum Suit { clubs, diamonds, hearts,spades };
const int jack = 11;          // именованные достоинства карт
const int queen = 12;
const int king = 13;
const int ace = 14;
////////////////////////////////////////////////////////
class card
{
  private:
    int number;               // достоинство карты
    Suit suit;                // масть
  public:
    card ()                   // конструктор без аргументов
      { }
                  // конструктор с двумя аргументами
    card (int n, Suit s) : number(n), suit(s)
      { }
    void display();           // вывод карты на экран
    bool isEqual(card);       // результат сравнения карт
};
//--------------------------------------------------------
void card::display()          // вывод карты на экран
{
  if( number>=2 && number<=10 )
    cout << number ;
  else
    switch(number)
    {
      case jack:   cout << "Валет ";  break;
      case queen:  cout << "Дама ";   break;
      case king:   cout << "Король "; break;
      case ace:    cout << "Туз ";    break;
    }
  switch(suit)
  {
    case clubs:     cout << "треф"   ;break;
    case diamonds:  cout << "бубен"  ;break;
```

```
    case hearts:    cout << "червей" ;break;
    case spades:    cout << "пик"    ;break;
  }
}
//-----------------------------------------------------
bool card::isEqual(card c2)  // сравнение двух карт
{
  return (number==c2.number && suit==c2.suit)? true : false;
}
////////////////////////////////////////////////////////
int main()
{
  card temp, chosen, prize;  // 3 карты
  int position;
  card card1(7, clubs);      // определение и инициализация card1
  cout << "\nКарта 1: ";
  card1.display();           // вывод card1
  card card2(jack, hearts);  // определение и инициализация card2
  cout <<  "\nКарта 2: ";
  card2.display();           // вывод card2
  card card3(ace, spades);   // определение и инициализация card3
  cout <<  "\nКарта 3: ";
  card3.display();           // вывод card3
  prize = card3;             // карту prize будет необходимо угадать
  cout << "\nМеняем местами карты 1 и 3...";
  temp = card3; card3 = card1; card1 = temp;
  cout << "\nМеняем местами карты 2 и 3...";
  temp = card3; card3 = card2; card2 = temp;
  cout << "\nМеняем местами карты 1 и 2...";
  temp = card2; card2 = card1; card1 = temp;
  cout << "\nНа какой позиции (1,2 или 3) теперь ";
  prize.display();           // угадываемая карта
  cout << "?";
  cin >> position;           // ввод позиции игроком
  switch (position)
  {                          // chosen - карта на позиции,
    case 1:chosen = card1; break; // выбранной игроком
    case 2:chosen = card2; break;
    case 3:chosen = card3; break;
  }
  if(chosen.isEqual(prize))       // сравнение карт
    cout << "Правильно! Вы выиграли!";
  else
    cout << "Неверно. Вы проиграли.";
   cout << " Вы выбрали карту ";
  chosen.display();               // вывод выбранной карты
  cout << endl;
  return 0;
}
```

Класс card содержит два конструктора. Первый из них не имеет аргументов и используется функцией main() для создания объектов temp, chosen и prize, которые не инициализируются при создании. Другой конструктор имеет два аргумента, и с его помощью создаются объекты card1, card2 и card3, инициализируемые определенными значениями. Кроме конструкторов, класс card содержит два метода, оба из которых определены вне класса.

Функция display() не имеет аргументов; ее задача заключается в отображении на экране параметров той карты, к которой она применяется. Так, оператор

```
chosen.display();
```

отображает карту, которую ввел пользователь.

Функция isEqual() проверяет, совпадает ли карта, введенная пользователем, с той, которая передается функции в качестве аргумента. Для сравнения используется условная операция. Функция могла бы быть реализована с помощью ветвления if else:

```
if(number==c2.number && suit==c2.suit)
  return true;
else
  return false;
```

однако условная операция обеспечивает более краткую форму записи.

Название c2 аргумента функции isEqual() указывает на то, что сравниваются две карты: первая из карт представлена объектом, методом которой является функция isEqual(). Выражение

```
if ( chosen.isEqual(prize) )
```

в функции main() сравнивает карту chosen с картой prize.

Если игрок не угадал карту, то результат работы программы будет приблизительно таким:

```
Карта 1: 7 треф
Карта 2: валет червей
Карта 3: туз пик
Меняем местами карты 1 и 3j
Меняем местами карты 2 и 3j
Меняем местами карты 1 и 2j
На какой позиции (1,2 или 3) теперь туз пик? 1
Неверно. Вы проиграли. Вы выбрали карту 7 треф
```

Структуры и классы

Все примеры, рассмотренные нами до сих пор, подтверждали негласный принцип: структуры предназначены для объединения данных, а классы — для объединения данных и функций. На самом деле, в большинстве ситуаций можно использовать структуры так же, как и классы. Формально разница между структурами и классами заключается лишь в том, что по умолчанию все члены класса являются скрытыми, а все члены структуры — открытыми.

Формат, который мы использовали при определении классов, выглядит примерно так:

```
class foo
{
  private:
    int data1;
```

```
  public:
    void func();
};
```

Поскольку ключевое слово private для членов классов подразумевается по умолчанию, указывать его явно не обязательно. Можно определять класс более компактным способом:

```
class foo
{
    int data1;
  public:
    void func();
};
```

В этом случае поле data1 сохранит свою закрытость. Многие программисты предпочитают второй стиль из-за его краткости. Мы же придерживаемся первого стиля, поскольку он более понятен.

Если вы хотите при помощи структуры выполнять те же действия, что и с использованием класса, вы можете отменить действие принятого по умолчанию ключевого слова public словом private и расположить открытые поля структуры до слова private, а закрытые поля — после слова private:

```
struct foo
{
  void func();
  private:
  int data1;
};
```

Тем не менее, чаще всего программисты не используют структуры таким образом, а придерживаются правила, о котором шла речь в начале: структуры предназначены для объединения данных, а классы — для объединения данных и функций.

Классы, объекты и память

Возможно, что в процессе изучения классов и объектов у вас сформировалось представление об объекте как о копии класса с точки зрения внутренней структуры. Это представление адекватно лишь в первом приближении. Оно акцентировано на том, что объекты являются самодостаточными программными единицами, созданными по образцу, представленному определением класса. Возможно, здесь уместна аналогия с машинами, сходящими с конвейера, каждая из которых построена согласно стандарту своей модели. Машины в этом смысле схожи с объектами, а модели — с классами.

На самом деле все устроено несколько сложнее. Верно, что каждый объект имеет собственные независимые поля данных. С другой стороны, все объекты одного класса используют одни и те же методы. Методы класса создаются и помещаются в память компьютера всего один раз — при создании класса. Это

вполне логически оправданно: нет никакого смысла держать в памяти копии методов для каждого объекта данного класса, поскольку у всех объектов методы одинаковы. А поскольку наборы значений полей у каждого объекта свои, поля объектов не должны быть общими. Это значит, что при создании объектов каждый из наборов данных занимает определенную совокупность свободных мест в памяти. Иллюстрация вышесказанного приведена на рис. 6.8.

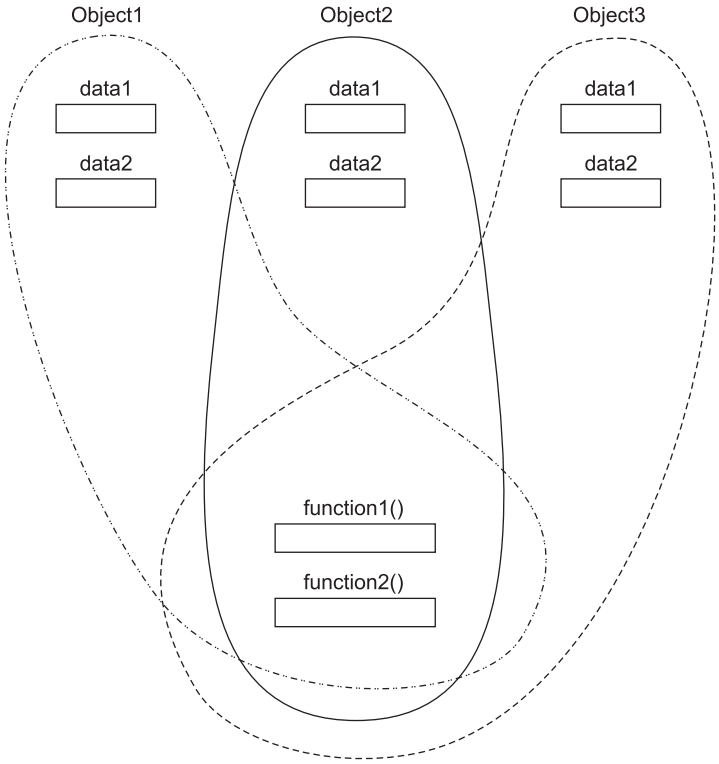

Рис. 6.8. Объекты, данные, функции и память

В примере SMALLOBJ, приведенном в начале этой главы, были созданы два объекта типа smallobj, что означало наличие двух полей с именем somedata в памяти. При этом функции setdata() и showdata() располагались в памяти в единственном экземпляре и совместно использовались обоими объектами класса. Между объектами не возникает конкуренции за общий ресурс (по крайней мере, в системах с последовательной обработкой), поскольку в любой момент времени выполняется не более одной функции.

В большинстве ситуаций вам не нужно использовать тот факт, что для всех объектов класса методы существуют в единственном экземпляре. Разумеется, гораздо проще представить себе объекты, которые включают в себя как данные, так и функции. Однако в некоторых ситуациях, например при оценке размера программы, необходимо знать, какова ее внутренняя организация.

Статические данные класса

Познакомившись с тем, что каждый объект класса содержит свои собственные данные, теперь мы должны углубить свое понимание данной концепции. Если поле данных класса описано с ключевым словом static, то значение этого поля будет одинаковым для всех объектов данного класса. Статические данные класса полезны в тех случаях, когда необходимо, чтобы все объекты включали в себя какое-либо одинаковое значение. Статическое поле по своим характеристикам схоже со статической переменной: оно видимо только внутри класса, но время его жизни совпадает со временем жизни программы. Таким образом, статическое поле существует даже в том случае, когда не существует ни одного объекта класса (обсуждение статических переменных функции можно найти в главе 5). Тем не менее, в отличие от статической переменной функции, предназначенной для сохранения значения между вызовами, статическая переменная класса используется для хранения данных, совместно используемых объектами класса.

Применение статических полей класса

Для того чтобы понять, зачем могут использоваться статические переменные класса, представьте себе следующую ситуацию. Допустим, вам необходимо, чтобы объект вашего класса располагал информацией, сколько еще объектов этого же класса существует на данный момент времени в памяти. Другими словами, если вы моделируете автомобильную гонку, и объекты — это гоночные машины, то вам нужно, чтобы каждый гонщик знал о том, сколько всего автомобилей участвует в гонке. В этом случае можно включить в класс статическую переменную count. Эта переменная будет доступна всем объектам класса, и все они будут видеть одно и то же ее значение.

Пример использования статических полей класса

Следующий пример, STATDAT, иллюстрирует простое применение статического поля класса:

```
// statdata.cpp
// статические данные класса
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////
class foo
{
  private:
    static int count;         // общее поле для всех объектов
                              // (в смысле "объявления")
  public:
    foo()                     // инкрементирование при создании объекта
      { count++; }
    int getcount()            // возвращает значение count
      { return count; }
};
//--------------------------------------------------------
```

```
int foo::count = 0;           // *определение* count
////////////////////////////////////////////////////////
int main()
{
  foo f1, f2, f3;             // создание трех объектов
  // каждый объект видит одно и то же значение
  cout << "Число объектов: " << f1.getcount() << endl;
  cout << "Число объектов: " << f2.getcount() << endl;
  cout << "Число объектов: " << f3.getcount() << endl;
  return 0;
}
```

В этом примере класс foo содержит единственное поле count, имеющее тип static int. Конструктор класса инкрементирует значение поля count. В функции main() мы определяем три объекта класса foo. Поскольку конструктор в этом случае вызывается трижды, инкрементирование поля count также происходит трижды. Метод getcount() возвращает значение count. Мы вызываем этот метод для каждого из объектов, и во всех случаях он возвращает одну и ту же величину:

```
Число объектов: 3
Число объектов: 3
Число объектов: 3
```

Если бы мы использовали не статическое, а автоматическое поле count, то конструктор увеличивал бы на единицу значение этого поля для каждого объекта, и результат работы программы выглядел бы следующим образом:

```
Число объектов: 1
Число объектов: 1
Число объектов: 1
```

Статические поля класса применяются гораздо реже, чем автоматические, однако существует немало ситуаций, где их использование удобно. Сравнение статических и автоматических полей класса проиллюстрировано на рис. 6.9.

Раздельное объявление и определение полей класса

Определение статических полей класса происходит не так, как для обычных полей. Обычные поля объявляются (компилятору сообщается имя и тип поля) и определяются (компилятор выделяет память для хранения поля) при помощи одного оператора. Для статических полей эти два действия выполняются двумя разными операторами: объявление поля находится внутри определения класса, а определение поля, как правило, располагается вне класса и зачастую представляет собой определение глобальной переменной.

Для чего используется такая двойственная форма? Если бы определение статического поля класса находилось внутри класса (как и предполагалось в ранних версиях C++), то это нарушило бы принцип, в соответствии с которым определение класса не должно быть связано с выделением памяти. Поместив определение статического поля вне класса, мы обеспечили однократное выделение памяти под это поле до того, как программа будет запущена на выполнение и статиче-

ское поле в этом случае станет доступным всему классу. Каждый объект класса уже не будет обладать своим собственным экземпляром поля, как это должно быть с полями автоматического типа. В этом отношении можно усмотреть аналогию статического поля класса с глобальными переменными.

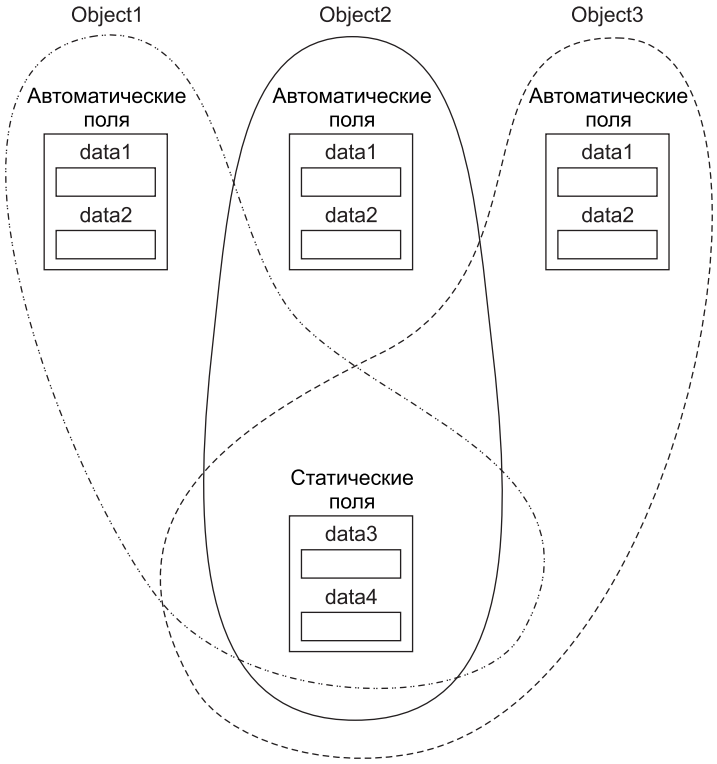

Рис. 6.9. Статические и автоматические поля класса

Работая со статическими данными класса, легко совершить ошибки, которые компилятор будет не в силах распознать. Если вы объявите статическое поле класса, но забудете его определить, компилятор не выдаст предупреждающего сообщения. Ваша программа будет считаться корректной до тех пор, пока редактор связей не обнаружит ошибку и не выдаст сообщение о том, что вы пытаетесь обратиться к необъявленной глобальной переменной. Это произойдет в том случае, если вы забудете указать имя класса при объявлении, например `foo::` в программе `STATDATA`.

const и классы

Мы рассмотрели несколько примеров использования модификатора `const` для того, чтобы защитить значения переменных от изменения. В главе 5 мы также использовали ключевое слово `const` для того, чтобы функция не могла изменить

значение аргумента, передаваемого ей по ссылке. Теперь, когда мы уже немного знакомы с классами, рассмотрим еще один способ применения модификатора const: с методами класса, их аргументами, а также с объектами класса.

Константные методы

Константные методы отличаются тем, что не изменяют значений полей своего класса. Рассмотрим это на примере под названием CONSTFU:

```
// constfu.cpp
// применение константных методов
class aClass
{
  private:
    int alpha;
  public:
    void nonFunc()           // неконстантный метод
      { alpha = 99; }        // корректно
    void conFunc()const      // константный метод
      { alpha = 99; }        // ошибка: нельзя изменить значение поля
};
```

Обычный метод nonFunc() может изменить значение поля alpha, а константный метод conFunc() не может. Если со стороны последнего будет предпринята попытка изменить поле alpha, компилятор выдаст сообщение об ошибке.

Для того чтобы сделать функцию константной, необходимо указать ключевое слово const после прототипа функции, но до начала тела функции. Если объявление и определение функции разделены, то модификатор const необходимо указывать дважды — как при объявлении функции, так и при ее определении. Те методы, которые лишь считывают данные из поля класса, имеет смысл делать константными, поскольку у них нет необходимости изменять значения полей объектов класса.

Использование константных функций помогает компилятору обнаруживать ошибки, а также указывает читающему листинг, что функция не изменяет значений полей объектов. С помощью константных функций можно создавать и использовать константные объекты, как мы увидим позже.

Пример класса Distance

Для того чтобы не включать множество нововведений в одну программу, мы до сих пор не использовали в наших примерах константные методы. Однако существует немалое количество ситуаций, где применение константных методов было бы весьма полезным. Например, метод showdist() класса Distance, неоднократно фигурировавшего в наших примерах, следовало сделать константным, потому что он не изменяет (и не должен изменять!) полей того объекта, для которого он вызывается. Он предназначен лишь для вывода текущих значений полей на экран.

Аналогично, метод add_dist() программы ENGLRET не должен изменять данных, хранящихся в объекте, из которого он вызывается. Значение этого объекта должно складываться с тем значением, которое метод принимает в качестве аргумента, а полученный результат возвращается вызывающей программе. Мы внесли изме-

нения в программу ENGLRET, сделав два указанных метода константными. Обратите внимание на то, что модификаторы const появились как в объявлениях, так и в определениях методов.

```
// engConst.cpp
// константные методы и константные аргументы
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////
class Distance                  // длина в английской системе
{
  private:
    int feet;
    float inches;
  public:                       // конструктор без аргументов
    Distance() : feet(0), inches(0.0)
      { }                       // конструктор с двумя аргументами
    Distance(int ft, float in) : feet(ft), inches(in)
      { }
    void getdist()              // ввод длины пользователем
    {
      cout << "\nВведите число футов: "; cin >> feet;
      cout << "Введите число дюймов: "; cin >> inches;
    }
    void showdist()const        // вывод длины
      { cout << feet << "\'-" << inches <<'\"'; }
    Distance add_dist(const Distance&) const; // сложение
};
//-----------------------------------------------------------
// сложение данного объекта с d2, возврат суммы
Distance Distance::add_dist(const Distance& d2) const
{
  Distance temp;                // временная переменная
  // feet = 0;                  // Ошибка: нельзя изменить поле
  // d2.feet = 0;               // Ошибка: нельзя изменить d2
  temp.inches = inches + d2.inches; // сложение дюймов
  if( temp.inches >= 12.0 )     // если сумма превышает 12.0,
  {                             // то уменьшаем ее на 12.0
    temp.inches -= 12.0;        // и увеличиваем число футов
    temp.feet = 1;              // на 1
  }
  temp.feet += feet + d2.feet;  // сложение футов
  return temp;
}
////////////////////////////////////////////////////////////
int main()
{
  Distance dist1, dist3;        // две длины
  Distance dist2(11, 6.25);     // определение и инициализация dist2
  dist1.getdist();              // ввод dist1
  dist3 = dist1.add_dist(dist2);//dist3 = dist1 + dist2
  // вывод всех длин
  cout << "\ndist1 = "; dist1.showdist();
  cout << "\ndist2 = "; dist2.showdist();
  cout << "\ndist3 = "; dist3.showdist();
  cout << endl;
  return 0;
}
```

В этом примере обе функции showdist() и add_dist() являются константными. В теле функции add_dist() первый из закомментированных операторов, feet=0, демонстрирует, что компилятор выдаст ошибку при попытке изменения константной функцией полей объекта, из которого она вызывалась.

Константные аргументы методов

В главе 5 мы говорили о том, что если вы хотите передать аргумент в функцию по ссылке и в то же время защитить его от изменения функцией, необходимо сделать этот аргумент константным при объявлении и определении функции. Методы классов в этом отношении не являются исключениями. В программе ENGCONST мы передаем аргумент в функцию add_dist() по ссылке, но хотим быть уверенными в том, что функция не изменит значения этого аргумента, в качестве которого при вызове выступает переменная dist2 функции main(). Для этого параметр d2 функции add_dist() указывается с модификатором const в ее объявлении и определении. Второй из закомментированных операторов показывает, что компилятор выдаст ошибку при попытке функции add_dist() изменить значение своего аргумента.

Константные объекты

В нескольких наших примерах мы видели, что ключевое слово const можно применять для защиты от изменения значений переменных стандартных типов, таких, как, например, int. Оказывается, аналогичным способом мы можем применять модификатор const и для объектов классов. Если объект класса объявлен с модификатором const, он становится недоступным для изменения. Это означает, что для такого объекта можно вызывать только константные методы, поскольку только они гарантируют, что объект не будет изменен. В качестве примера рассмотрим программу CONSTOBJ.

```
//constObj.cpp
//constant Distance objects
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////
class Distance               // длина в английской системе
{
  private:
    int feet;
    float inches;
  public:                    // конструктор с двумя аргументами
    Distance(int ft, float in) : feet(ft), inches(in)
      { }
    void getdist()           // неконстантный метод
    {
      cout << "\nВведите число футов: "; cin >> feet;
      cout << "Введите число дюймов: "; cin >> inches;
    }
void showdist() const        // константный метод
  { cout << feet << "\'-" << inches << '\"'; }
};
```

```
//////////////////////////////////////////////////////
int main()
{
  const Distance football(300,0);
  //football.getdist();      // ошибка: метод getdist() неконстантный
  cout << " Длина поля равна ";
  football.showdist();       // корректно
  cout << endl;
  return 0;
}
```

Футбольное поле в американском футболе имеет длину ровно 300 футов. Если бы нам было необходимо использовать в программе величину, выражающую длину футбольного поля, то для нас имело бы смысл сделать эту величину константной. Именно это и сделано в программе CONSTOBJ в отношении объекта football. Теперь для этого объекта можно вызывать только константные методы, например showdist(). Вызовы неконстантных методов, таких, как getdist(), который присваивает полям объекта значения, вводимые пользователем, являются некорректными. Таков механизм, обеспечивающий требуемую константность объекта football.

Когда вы создаете класс, всегда является хорошим стилем объявлять константными функции, не изменяющие полей объектов класса. Это позволяет тому, кто использует данный класс, создавать константные объекты класса. Из этих объектов могут вызываться только константные функции. Использование модификатора const упрощает компилятору задачу оказания вам своевременной помощи в процессе создания программы.

Зачем нужны классы?

Теперь, когда вы имеете представление о классах и объектах, вас, наверное, интересует ответ на вопрос, какова выгода от их использования, тем более что на наших примерах мы убеждались: для выполнения определенных действий программе не обязательно использовать классы — идентичные действия выполнялись при помощи процедурного подхода.

Первым достоинством объектно-ориентированного программирования является то, что оно обеспечивает значительное сходство между объектами реального мира, моделируемыми программой, и объектами классов C++. Мы видели, как объекты классов представляли такие реально существующие вещи, как < >, игральные карты, геометрические фигуры и т. д. Вся информация, касающаяся < > вообще, описана при создании класса; там же описаны и все методы, то есть действия, которые можно производить с этой информацией. Это облегчает концептуализацию решаемой задачи. Вы выделяете в задаче те элементы, которые целесообразно представлять в виде объектов, а затем помещаете все данные и функции, связанные с этим объектом, внутрь самого объекта. Если объект представляет игральную карту, то с помощью описания класса вы наделяете объект полями для хранения достоинства карты, ее масти, а также функциями,

выполняющими действия по заданию этих величин, их изменению, выводу на экран, сравнению с заданными значениями и т. д.

В процедурной программе, напротив, моделирование объектов реального мира возлагается на глобальные переменные и функции. Они не способны вместе образовать легко воспринимаемую модель.

Иногда при решении задачи бывает не столь очевидно, какие элементы реального мира следует представить в качестве объектов. Например, если вы пишете компьютерный вариант игры в шахматы, то что нужно представить в качестве объекта: шахматистов, клетки шахматной доски или игровую позицию целиком?

Небольшие программы, такие, какими являются большинство наших примеров, можно создавать методом «проб и ошибок». Можно представить задачу в виде объектов каким-либо одним способом и написать определения классов для этих объектов. Если такое представление устроит вас, то вы можете продолжить разработку созданных классов. В противном случае следует выбрать другой способ представления задачи в виде совокупности объектов и повторить операцию создания классов. Чем больше опыта вы наберете, создавая объектно-ориентированные программы, тем проще вам будет представлять новые задачи в терминах ООП.

При создании больших программ метод проб и ошибок оказывается малоприменимым. Для анализа поставленной задачи и создания классов и объектов, подходящих для ее решения, используется объектно-ориентированная разработка (ООР). Мы обсудим объектно-ориентированную разработку в главе 16, которая целиком посвящена аспектам ее применения.

Некоторые преимущества объектно-ориентированного программирования не столь очевидны. Вспомните о том, что ООП было создано, чтобы бороться со сложностью больших программ. Программы небольшого размера, подобные созданным нами, в меньшей степени нуждаются в объектно-ориентированной организации. Чем больше программа, тем ощутимей эффект от использования ООП. Но даже для программ маленького размера применение объектно-ориентированного подхода способно принести определенную выгоду: например, по сравнению с процедурными программами увеличивается эффективность компилятора при поиске и выявлении ошибок концептуального уровня.

Резюме

Класс представляет собой образ, определяющий структуру своих объектов. Объекты включают в себя как данные, так и функции, предназначенные для их обработки. И данные, и функции могут быть определены как закрытые, что означает их доступность только для членов данного класса, и как открытые, то есть доступные любой функции программы. Закрытость члена класса задается ключевым словом `private`, а открытость — ключевым словом `public`.

Методом класса называется функция, являющаяся членом этого класса. Методы класса, в отличие от других функций, имеют доступ к закрытым членам класса.

Конструктор — это метод класса, имя которого совпадает с именем класса и который выполняется каждый раз при создании нового объекта. Конструктор не имеет типа возвращаемого значения, однако может принимать аргументы. Часто конструкторы применяются для инициализации создаваемых объектов класса. Конструкторы допускают перегрузку, поэтому возможна инициализация объекта несколькими способами.

Деструктор — это метод класса, именем которого является имя класса, предваренное символом ~. Вызов деструктора производится при уничтожении объекта. Деструктор не имеет ни аргументов, ни возвращаемого значения.

В памяти компьютера каждый объект имеет свои собственные участки, хранящие значения полей этого объекта, но методы класса хранятся в памяти в единственном экземпляре. Поле класса также можно сделать единым для всех объектов данного класса, описав его при определении класса с ключевым словом static.

Одной из главных причин использования объектно-ориентированного программирования является его возможность с высокой степенью точности моделировать объекты реального мира. Иногда бывает непросто представить решаемую задачу в терминах объектов и классов. Для небольших программ можно последовательно перебрать возможные способы такого представления. Для более серьезных и больших проектов применяются другие методы.

Вопросы

Ответы на перечисленные ниже вопросы можно найти в приложении Ж.

1. Для чего необходимо определение класса?
2. _____ имеет такое же отношение к _____, как стандартный тип данных к переменной этого типа.
3. В определении класса члены класса с ключевым словом private доступны:
   а) любой функции программы;
   б) в случае, если вам известен пароль;
   в) методам этого класса;
   г) только открытым членам класса.
4. Напишите определение класса leverage, включающего одно закрытое поле типа int с именем crowbar и одним открытым методом с прототипом void pry().
5. Истинно ли следующее утверждение: поля класса должны быть закрытыми?
6. Напишите оператор, создающий объект lever1 класса leverage, описанного в вопросе 4.
7. Операция точки (операция доступа к члену класса) объединяет следующие два элемента (слева направо):

а) член класса и объект класса;

б) объект класса и класс;

в) класс и член этого класса;

г) объект класса и член этого класса.

8. Напишите оператор, который вызовет метод pry() объекта lever1 (см. вопросы 4 и 6).
9. Методы класса, определенные внутри класса, по умолчанию _____.
10. Напишите метод getcrow() для класса leverage (см. вопрос 4), который будет возвращать значение поля crowbar. Метод следует определить внутри определения класса.
11. Конструктор вызывается автоматически в момент _____ объекта.
12. Имя конструктора совпадает с именем _____ .
13. Напишите конструктор, который инициализирует нулевым значением поле crowbar класса leverage (см. вопрос 4). Конструктор следует определить внутри определения класса.
14. Верно или неверно следующее утверждение: класс может иметь более одного конструктора с одним и тем же именем?
15. Методу класса всегда доступны данные:

а) объекта, членом которого он является;

б) класса, членом которого он является;

в) любого объекта класса, членом которого он является;

г) класса, объявленного открытым.

16. Предполагая, что метод getcrow(), описанный в вопросе 10, определен вне класса, объявите этот метод внутри класса.
17. Напишите новую версию метода getcrow(), описанного в вопросе 10, которая определяется вне класса.
18. Единственным формальным различием между структурами и классами в C++ является то, что _____.
19. Пусть определены три объекта класса. Сколько копий полей класса содержится в памяти? Сколько копий методов функций?
20. Посылка сообщения объекту эквивалентна _____.
21. Классы полезны потому, что:

а) не занимают памяти, если не используются;

б) защищают свои данные от доступа со стороны других классов;

в) собирают вместе все аспекты, касающиеся отдельной вещи;

г) адекватно моделируют объекты реального мира.

22. Истинно ли следующее утверждение: существует простой, но очень точный метод, позволяющий представлять решаемую задачу в виде совокупности объектов классов?

23. Константный метод, вызванный для объекта класса:
    а) может изменять как неконстантные, так и константные поля;
    б) может изменять только неконстантные поля;
    в) может изменять только константные поля;
    г) не может изменять как неконстантные, так и константные поля.
24. Истинно ли следующее утверждение: объект, объявленный как константный, можно использовать только с помощью константных методов?
25. Напишите объявление (не определение) функции типа `const void` с именем `aFunc()`, которая принимает один константный аргумент `jerry` типа `float`.

Упражнения

Решения к упражнениям, помеченным знаком *, можно найти в приложении Ж.

*1. Создайте класс `Int`, имитирующий стандартный тип `int`. Единственное поле этого класса должно иметь тип `int`. Создайте методы, которые будут устанавливать значение поля, равным нулю, инициализировать его целым значением, выводить значение поля на экран и складывать два значения типа `Int`.

    Напишите программу, в которой будут созданы три объекта класса `Int`, два из которых будут инициализированы. Сложите два инициализированных объекта, присвойте результат третьему, а затем отобразите результат на экране.

*2. Представьте пункт для взимания платежей за проезд по автостраде. Каждая проезжающая машина должна заплатить за проезд 50 центов, однако часть машин платит за проезд, а часть проезжает бесплатно. В кассе ведется учет числа проехавших машин и суммарная выручка от платы за проезд.

    Создайте модель такой кассы с помощью класса `tollBooth`. Класс должен содержать два поля. Одно из них, типа `unsigned int`, предназначено для учета количества проехавших автомобилей, а второе, имеющее тип `double`, будет содержать суммарную выручку от оплаты проезда. Конструктор должен инициализировать оба поля нулевыми значениями. Метод `payingCar()` инкрементирует число машин и увеличивает на 0,50 суммарную выручку. Другой метод, `nopayCar()`, увеличивает на единицу число автомобилей, но оставляет без изменения выручку. Метод `display()` выводит оба значения на экран. Там, где это возможно, сделайте методы константными.

    Создайте программу, которая продемонстрирует работу класса. Программа должна предложить пользователю нажать одну клавишу для того, чтобы сымитировать заплатившего автолюбителя, и другую клавишу, чтобы сымитировать недобросовестного водителя. Нажатие клавиши `Esc` должно привести к выдаче текущих значений количества машин и выручки и завершению программы.

*3. Создайте класс с именем time, содержащий три поля типа int, предназначенные для хранения часов, минут и секунд. Один из конструкторов класса должен инициализировать поля нулевыми значениями, а другой конструктор — заданным набором значений. Создайте метод класса, который будет выводить значения полей на экран в формате 11:59:59, и метод, складывающий значения двух объектов типа time, передаваемых в качестве аргументов.

   В функции main() следует создать два инициализированных объекта (подумайте, должны ли они быть константными) и один неинициализированный объект. Затем сложите два инициализированных значения, а результат присвойте третьему объекту и выведите его значение на экран. Где возможно, сделайте методы константными.

4. Создайте класс employee, используя упражнение 4 главы 4. Класс должен включать поле типа int для хранения номера сотрудника и поле типа float для хранения величины его оклада. Методы класса должны позволять пользователю вводить и отображать данные класса. Напишите функцию main(), которая запросит пользователя ввести данные для трех сотрудников и выведет полученную информацию на экран.

5. Взяв в качестве основы структуру из упражнения 5 главы 4, создайте класс date. Его данные должны размещаться в трех полях типа int: month, day и year. Метод класса getdate() должен принимать значение для объекта в формате 12/31/02, а метод showdate() — выводить данные на экран.

6. Расширьте содержание класса employee из упражнения 4, включив в него класс date и перечисление etype (см. упражнение 6 главы 4). Объект класса date будет использоваться для хранения даты приема сотрудника на работу. Перечисление будет использовано для хранения статуса сотрудника: лаборант, секретарь, менеджер и т. д. Последние два поля данных должны быть закрытыми в определении класса employee, как и номер и оклад сотрудника. Вам будет необходимо разработать методы getemploy() и putemploy(), предназначенные соответственно для ввода и отображения информации о сотруднике. Возможно, при создании методов вам понадобится ветвление switch для работы с перечисляемым типом etype. Напишите функцию main(), которая попросит пользователя ввести данные о трех сотрудниках, а затем выведет эти данные на экран.

7. В морской навигации координаты точки измеряются в градусах и минутах широты и долготы. Например, координаты бухты Папити на о. Таити равны 149 градусов 34.8 минут восточной долготы и 17 градусов 31.5 минут южной широты. Это записывается как 149°34.8' W, 17°31.5' S. Один градус равен 60 минутам (устаревшая система также делила одну минуту на 60 секунд, но сейчас минуту делят на обычные десятичные доли). Долгота измеряется величиной от 0 до 180 градусов восточнее или западнее Гринвича. Широта принимает значения от 0 до 90 градусов севернее или южнее экватора.

Создайте класс angle, включающий следующие три поля: типа int для числа градусов, типа float для числа минут и типа char для указания направления (N, S, E или W). Объект этого класса может содержать значение как широты, так и долготы. Создайте метод, позволяющий ввести координату точки, направление, в котором она измеряется, и метод, выводящий на экран значение этой координаты, например 179°59.9' E. Кроме того, напишите конструктор, принимающий три аргумента. Напишите функцию main(), которая сначала создает переменную с помощью трехаргументного конструктора и выводит ее значение на экран, а затем циклически запрашивает пользователя ввести значение координаты и отображает введенное значение на экране. Для вывода символа градусов (°) можно воспользоваться символьной константой '\xF8'.

8. Создайте класс, одно из полей которого хранит «порядковый номер» объекта, то есть для первого созданного объекта значение этого поля равно 1, для второго созданного объекта значение равно 2 и т. д.

   Для того чтобы создать такое поле, вам необходимо иметь еще одно поле, в которое будет записываться количество созданных объектов класса (это означает, что последнее поле должно относиться не к отдельным объектам класса, а ко всему классу в целом. Вспомните, какое ключевое слово необходимо при описании такого поля.). Каждый раз при создании нового объекта конструктор может получить значение этого поля и в соответствии с ним назначить объекту индивидуальный порядковый номер.

   В класс следует включить метод, который будет выводить на экран порядковый номер объекта. Создайте функцию main(), в которой будут созданы три объекта, и каждый объект выведет на экран свой порядковый номер, например: Мой порядковый номер: 2 и т. п.

9. На основе структуры fraction из упражнения 8 главы 4 создайте класс fraction. Данные класса должны быть представлены двумя полями: числителем и знаменателем. Методы класса должны получать от пользователя значения числителя и знаменателя дроби в форме 3/5 и выводить значение дроби в этом же формате. Кроме того, должен быть разработан метод, складывающий значения двух дробей. Напишите функцию main(), которая циклически запрашивает у пользователя ввод пары дробей, затем складывает их и выводит результат на экран. После каждой такой операции программа должна спрашивать пользователя, следует ли продолжать цикл.

10. Создайте класс с именем ship, который будет содержать данные об учетном номере корабля и координатах его расположения. Для задания номера корабля следует использовать механизм, аналогичный описанному в упражнении 8. Для хранения координат используйте два поля типа angle (см. упражнение 7). Разработайте метод, который будет сохранять в объекте данные о корабле, вводимые пользователем, и метод, выводящий данные о корабле на экран. Напишите функцию main(), создающую три объекта класса ship, затем запрашивающую ввод пользователем информации о каждом из кораблей и выводящую на экран всю полученную информацию.

11. Модифицируйте калькулятор, созданный в упражнении 12 главы 5 так, чтобы вместо структуры fraction использовался одноименный класс. Класс должен содержать методы для ввода и вывода данных объектов, а также для выполнения арифметических операций. Кроме того, необходимо включить в состав класса функцию, приводящую дробь к несократимому виду. Функция должна находить наибольший общий делитель числителя и знаменателя и делить числитель и знаменатель на это значение. Код функции, названной lowterms(), приведен ниже:

```
void fraction::lowterms()    // сокращение дроби
{
long tnum, tden, temp, gcd;
tnum = labs(num);            // используем неотрицательные
tden = labs(den);            // значения (нужен cmath)
if( tden == 0 )              // проверка знаменателя на 0
  { cout << "Недопустимый знаменатель!"; exit(1); }
else if( tnum == 0 )         // проверка числителя на 0
  { num=0; den = 1; return; }
// нахождение наибольшего общего делителя
while(tnum !=0)
{
  if( tnum < tden )          // если числитель больше знаменателя,
    { temp=tnum; tnum=tden; tden=temp; } //меняем их местами
  tnum = tnum - tden;        // вычитание
}
gcd = tden;                  // делим числитель и знаменатель на
num = num / gcd;             // полученный наибольший общий делитель
den = den / gcd;
}
```

Можно вызывать данную функцию в конце каждого метода, выполняющего арифметическую операцию, либо непосредственно перед выводом на экран результата. Кроме перечисленных методов, вы можете включить в класс конструктор с двумя аргументами, что также будет полезно.

12. Используйте преимущество ООП, заключающееся в том, что однажды созданный класс можно помещать в другие программы. Создайте новую программу, которая будет включать класс fraction, созданный в упражнении 11. Программа должна выводить аналог целочисленной таблицы умножения для дробей. Пользователь вводит знаменатель, а программа должна подобрать всевозможные целые значения числителя так, чтобы значения получаемых дробей находились между 0 и 1. Дроби из получившегося таким образом набора перемножаются друг с другом во всевозможных комбинациях, в результате чего получается таблица следующего вида (для знаменателя, равного 6):

```
1/61/31/22/35/6
-----------------------------------------------------------
1/61/361/181/121/95/36
1/31/181/91/62/95/18
1/21/121/61/41/35/12
2/31/92/91/34/95/9
5/65/365/185/125/925/36
```