

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ
ВЫСШЕГО ОБРАЗОВАНИЯ
«СЕВЕРО-КАВКАЗСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»

Методические указания
по выполнению лабораторных работ
по дисциплине «Объектно-ориентированное
программирование» для студентов направления
подготовки
09.03.03 «Прикладная информатика»
Направленность (профиль)
«Прикладная информатика в экономике»

Ставрополь 2026

Лабораторная работа №1. Классы.

Цель работы: изучить базовые понятия (классы, подклассы и методы) Реализовать фундаментальные принципы объектно-ориентированного программирования.

Формируемые компетенции:

1. Теоретическая и 2. Практическая части

Появление объектно-ориентированного программирования

Объектно-ориентированное программирование (ООП) — это технология, возникшая как реакция на очередную фазу кризиса программного обеспечения, когда методы структурного программирования уже не позволяли справляться с растущей сложностью промышленных программных продуктов. Следствия — срыв сроков проектов, перерасход бюджета, урезанная функциональность и множество ошибок.

Существенная черта промышленной программы — ее *сложность*: один разработчик не в состоянии охватить все аспекты системы, поэтому в ее создании участвует целый коллектив. Следовательно, к первичной сложности самой задачи, вытекающей из предметной области, добавляются проблемы управления процессом разработки. Так как сложные системы разрабатываются в расчете на длительную эксплуатацию, то появляются еще две проблемы: *сопровождение* системы (устранение обнаруженных ошибок) и ее *модификация*, поскольку у заказчика постоянно появляются новые требования и пожелания. Часто затраты на сопровождение и модификацию сопоставимы с затратами на собственно разработку системы.

Способ управления сложными системами был известен еще в древности — *divide et impera* (разделяй и властвуй). То есть выход — в *декомпозиции* системы на все меньшие и меньшие подсистемы, каждую из которых можно разрабатывать независимо. Но если в рамках структурного подхода декомпозиция понимается как разбиение алгоритма, когда каждый из модулей системы выполняет один из этапов общего процесса, то ООП предлагает совершенно другой подход.

Суть его в том, что в качестве *критерия декомпозиции* принимается принадлежность элементов системы к различным *абстракциям проблемной области*. Откуда же они берутся? Исключительно из головы программиста, который, анализируя предметную область, вычленяет из нее отдельные объекты. Для каждого из них определяются свойства, существенные для решения задачи. Так из реальных объектов предметной области получаются абстрактные программные объекты.

Почему *объектно-ориентированная декомпозиция* оказалась более эффективной, чем *функциональная* (синонимы — *структурная, процедурная, алгоритмически-ориентированная*) декомпозиция? Тому есть много причин. Чтобы в них разобраться, рассмотрим критерии качества проекта, связанные с его декомпозицией.

Критерии качества декомпозиции проекта

Со сложностью приложения трудно что-либо сделать — она определяется целью создания программы. А вот сложность реализации можно попытаться контролировать. Первый вопрос, возникающий при декомпозиции: на какие компоненты (модули, функции, классы) нужно разбить программу? Очевидно, что с ростом количества компонентов сложность программы растет. Особенно негативны последствия неоправданного разбиения на компоненты, когда оказываются разделенными действия, по сути тесно связанные между собой.

Вторая проблема связана с организацией взаимодействия между компонентами. Взаимодействие упрощается и его легче контролировать, если каждый компонент рассматривается как «черный ящик», внутреннее устройство которого неизвестно, но известны выполняемые им функции, а также «входы» и «выходы». Вход компонента позволяет ввести в него значение некоторой входной переменной, а выход — получить значение выходной переменной. Совокупность входов и выходов черного ящика определяет интерфейс компонента. Интерфейс реализуется как набор функций (или запросов к компоненту), вызывая которые, клиент либо получает какую-то информацию, либо меняет состояние компонента.

Здесь термин «клиент» означает просто компонент, использующий услуги другого компонента, выполняющего в этом случае роль сервера. Взаимоотношение клиент–сервер довольно старо и использовалось уже в рамках структурного подхода, когда функция-клиент пользовалась услугами функции-сервера путем ее вызова.

Подытожим сказанное. Для оценки качества программного проекта нужно учитывать, кроме всех прочих, следующие два показателя.

Сцепление (*cohesion*) внутри компонента — характеризует степень взаимосвязи его отдельных частей. Пример: если внутри компонента решаются две подзадачи, которые можно легко разделить, то он обладает слабым (плохим) сцеплением.

Связанность (*coupling*) между компонентами — описывает интерфейс между компонентом-клиентом и компонентом-сервером. Общее количество входов и выходов сервера есть мера связанности. Чем меньше связанность между компонентами, тем проще понять и отследить их взаимодействие. Так как в больших проектах разные компоненты разрабатываются разными людьми, уменьшать связанность между компонентами очень важно.

Заметим, что описанные показатели имеют относительный характер и пользоваться ими следует благоразумно. Например, стремление максимально увеличить сцепление может привести к дроблению проекта на очень большое количество мелких функций.

Почему в случае функциональной декомпозиции трудно достичь слабой связанности между компонентами? Дело в том, что в сложной программной системе, реализующей структурную модель, практически невозможно обойтись без связи через глобальные структуры данных, а это означает, что любая функция в случае ошибки может их испортить. Такие ошибки очень трудно обнаружить. Добавьте к этому головную боль для сопровождающего программиста, который должен помнить десятки, если не сотни, обращений к общим данным из разных частей проекта. Соответственно, модификация существующего проекта в связи с новыми требованиями заказчика также потребует очень большой работы.

Другой проблемой в проектах с функциональной декомпозицией является «проклятие» общего глобального пространства имен. Члены команды, работающей над проектом, должны тратить немалые усилия по согласованию применяемых имен для своих функций, чтобы они были уникальными в рамках всего проекта.

Что принесло с собой ООП

Первым бросающимся в глаза отличием ООП от структурного программирования является использование классов. *Класс* — это тип, определяемый программистом, в котором объединяются структуры данных и функции их обработки. Конкретные переменные типа данных «класс» называются *экземплярами класса*, или *объектами*. Программы, разрабатываемые на основе концепций ООП, реализуют алгоритмы, описывающие взаимодействие между объектами.

Класс содержит константы и переменные, называемые *полями*, а также выполняемые над ними операции и функции. Функции класса называются *методами*¹. Предполагается, что доступ к полям класса возможен только через вызов соответствующих методов. Поля и методы являются *элементами (членами)* класса.

Эффективным механизмом ослабления связанности между компонентами при объектно-ориентированной декомпозиции является *инкапсуляция* — ограничение доступа к данным и их объединение с методами, обрабатывающими эти данные. Доступ к отдельным частям класса регулируется с помощью ключевых слов *public* (открытая часть), *private* (закрытая часть) и *protected* (защищенная часть). Методы, расположенные в открытой части, формируют *интерфейс* класса и могут свободно вызываться клиентом через соответствующий объект класса. Доступ к закрытой секции класса возможен только из его собственных методов, а к защищенной — из его собственных методов, а также из методов классов-потомков (механизму наследования посвящен следующий семинар).

Инкапсуляция повышает надежность программ, предотвращая непреднамеренный ошибочный доступ к полям объекта. Кроме этого, программу легче модифицировать, поскольку при сохранении интерфейса класса можно менять его реализацию, и это не затронет внешний программный код (код клиента).

С понятием инкапсуляции тесно связано понятие *сокрытия информации*. С другой стороны, это понятие соприкасается с понятием *разделения ответственности* между клиентом и сервером. Клиент не обязан знать, *как реализованы* те или иные методы в сервере. Ему достаточно знать, *что делает* данный метод и как к нему *обратиться*. При хорошем проектировании имена методов обычно отражают суть

¹ Широко используется и другое название — функции-члены, возникшее при подстрочном переводе англоязычной литературы.

выполняемой ими работы, поэтому сопровождающий программист может читать код клиента как художественную литературу.

Заметим, что класс обеспечивает локальную (в пределах класса) область видимости имен. Поэтому методы, реализующие схожие подзадачи в разных классах, могут иметь одинаковые имена. То же относится и к полям разных классов. Таким образом, «проклятия общего глобального пространства имен» здесь попросту нет.

С ООП связаны еще два инструмента, грамотное использование которых повышает качество проектов: наследование классов и полиморфизм. *Наследование* — механизм получения нового класса из существующего путем его дополнения или изменения. Благодаря этому возможно повторное использование кода. Наследование позволяет создать иерархии родственных типов, совместно использующих код и интерфейсы.

Полиморфизм дает возможность создавать множественные определения для операций и функций. Какое именно определение будет использоваться, зависит от контекста программы. Вы уже знакомы с двумя разновидностями полиморфизма — перегрузкой функций и шаблонами функций. ООП дает еще три возможности: перегрузку операций, использование виртуальных методов и шаблоны классов. Перегрузка операций позволяет применять для собственных классов те же операции, что используются для встроенных типов C++. Виртуальные методы обеспечивают возможность выбрать на этапе выполнения нужный метод среди одноименных методов базового и производного классов. Шаблоны классов позволяют описать классы, инвариантные относительно типов данных.

Отметим, что в реальном проекте, разработанном на базе объектно-ориентированной декомпозиции, находится место и для процедурной декомпозиции (например, при реализации сложных методов).

От структуры — к классу

Прообразом класса в C++ является структура в C. В то же время в C++ структура обрела новые свойства и теперь является частным видом класса. Со структурой `struct` в C++ можно делать все, что можно делать с классом. Тем не менее обычно структуры в C++ используют лишь для удобства работы с небольшими наборами данных без какого-либо собственного поведения.

Новые понятия легче изучать, отталкиваясь от уже освоенного материала. Давайте возьмем задачу 6.1 и посмотрим, что можно с ней сделать, применив средства ООП.

Задача 10.1. Поиск в простой базе (массив объектов)

В текстовом файле хранится база отдела кадров предприятия. На предприятии 100 сотрудников. Каждая строка файла содержит запись об одном сотруднике.

Формат записи: фамилия и инициалы (30 поз., фамилия должна начинаться с первой позиции), год рождения (5 поз.), оклад (10 поз.). Написать программу, которая по заданной фамилии выводит на экран сведения о сотруднике, подсчитывая средний оклад всех запрошенных сотрудников

Для начала преобразуем в класс структуру `Man`, которая применялась в листинге 6.1 для хранения сведений об одном сотруднике. Мы предполагаем, что наш новый тип будет обладать более сложным поведением, чем просто чтение и запись его полей:

```
class Man {
    char name[l_name + 1];
    int birth_year;
    float pay; };
```

Все поля класса по умолчанию закрыты (`private`), поэтому, если клиентская функция `main` объявит объект `Man man`, а потом попытается обратиться к какому-либо его полю, например `man.pay = value`, произойдет ошибка компиляции. Поэтому в состав класса надо добавить методы доступа к его полям. Эти методы должны быть общедоступными, или открытыми (`public`).

Но сначала критически рассмотрим определения полей. Поле `name` объявлено как статический массив. Это не очень гибкое решение. Желательно предусмотреть простой механизм настройки на произвольное значение длины `l_name`, чтобы класс `Man` можно было применять в разных приложениях. Используем динамический массив символов (`char* pName`). Возникает вопрос: кто и где будет выделять память под этот массив? Один из принципов ООП гласит, что все объекты должны быть самодостаточными, то есть полностью себя обслуживать.

Таким образом, в состав класса необходимо включить метод, который обеспечит выделение памяти под динамический массив при создании объекта (переменной типа `Man`). Метод, который автоматически вызывается при создании экземпляра класса, называется *конструктором*. Имя конструктора совпадает с именем класса. Парным конструктору является метод, называемый

деструктором, который автоматически вызывается перед уничтожением объекта. Имя деструктора отличается от имени конструктора только наличием предваряющего символа ~ (тильда).

Ясно, что если в конструкторе была выделена динамическая память, то в деструкторе нужно побеспокоиться о ее освобождении. Напомним, что объект, созданный как локальная переменная в некотором блоке, уничтожается, когда при выполнении достигнут конец блока. Если же объект создан с помощью операции new, например, `Man* pMan = new Man;`, то для его уничтожения применяется операция delete. Итак, наш класс принимает следующий вид:

```
class Man {
public:
    Man(int lName = 30) { pName = new char[lName + 1]; }           // конструктор
    ~Man() { delete [] pName; }                                     // деструктор
private:
    char* pName;
    int   birth_year;
    float pay; };           // объявление класса должно завершаться точкой с запятой!
```

Отметим, что здесь методы класса определены как встроенные (inline). При другом способе описания методы только объявляются внутри класса, а их реализация записывается вне определения класса, как показано ниже:

```
//////////////////////////////////// Man.h (интерфейс класса) //////////////////////////////////
class Man {
public:
    Man( int lName = 30 );           // конструктор
    ~Man();                           // деструктор
private:
    char* pName;
    int   birth_year;
    float pay;
};
//////////////////////////////////// Man.cpp (реализация класса) //////////////////////////////////
#include "Man.h"
Man::Man( int lName ) { pName = new char[lName + 1]; }
Man::~~Man() { delete [] pName; }
```

При внешнем определении метода перед его именем указывается имя класса, за которым следует операция доступа к области видимости ::. Выбор способа определения метода зависит в основном от его размера: короткие методы можно определить как встроенные, что может привести к более эффективному коду.

Продолжим процесс проектирования интерфейса нашего класса. Какие методы нужно добавить в класс? С какими сигнатурами²? На этом этапе полезно задаться вопросом: *какие обязанности* должны быть возложены на класс Man? Первую обязанность мы уже реализовали: объект класса хранит сведения о сотруднике. Чтобы ими воспользоваться, клиент должен иметь возможность получить эти сведения, изменить их и вывести на экран. Кроме этого, для поиска сотрудника желательно иметь возможность сравнивать его имя с заданным.

Начнем с методов, обеспечивающих доступ к полям класса. Для *считывания* значений полей добавим методы GetName, GetBirthYear, GetPay. Очевидно, что аргументы им не нужны, а тип возвращаемого значения совпадает с типом поля. Для *записи* значений полей добавим методы SetName, SetBirthYear, SetPay. Чтобы определиться с их сигнатурой, представим себе, как они будут вызываться клиентом. Вспомните, как вводились данные в задаче 6.1 (см. листинг 6.1, цикл, помеченный комментарием 2). Поступив аналогично, получим в теле цикла код вида

```
man[i].SetName( buf );
man[i].SetBirthYear( atoi( &buf[l_name] ) );
man[i].SetPay( atof( &buf[l_name + l_year] ) );
```

Вряд ли такое решение можно признать удачным. Ведь аргументы вида `atoi(&buf[l_name])` фактически являются сущностями типа «как делать», а не «что делать»! Иными словами, крайне нежелательно нагружать клиента избыточной информацией. Детали реализации должны быть спрятаны (инкапсулированы) внутри класса. Поэтому в данной задаче более удачной является сигнатура метода

²Сигнатура, прототип, заголовок функции — синонимы.

void SetBirthYear(const char* fromBuf). Аналогичные размышления можно повторить и для метода SetPay.

СОВЕТ. Распределяя обязанности между клиентом и сервером, инкапсулируйте подробности реализации в серверном компоненте.

Теперь перейдем к функции main, решающей подзадачу поиска сотрудника в базе по заданной фамилии и вывода сведений о нем (*операторы 7–10* листинга 6.1). Напомним, что в задаче 6.1 поиск сотрудника с заданным именем был реализован с помощью следующего цикла (для удобства имя dbase изменено на man):

```
for ( int i = 0; i < n_record; ++i ) {
    if ( strstr( man[i].name, name ) )                // 1
        if ( man[i].name[strlen(name)] == ' ' ) {    // 2
            strcpy( name, man[i].name );              // 3
            cout << name << man[i].birth_year << ' ' << man[i].pay << endl; // 4
            // . . .
        }
}
```

Опять задумаемся над вопросом: какая информация здесь *избыточна для клиента*? Здесь решаются две задачи: сравнение имени в очередной записи с заданным именем name (*операторы 1 и 2*) и вывод информации (*операторы 3 и 4*). Поручим решение первой подзадачи методу CompareName(const char* name), инкапсулировав в него подробности решения, а решение второй подзадачи — методу Print. Проектирование интерфейса класса Man завершено. Посмотрите на текст программы, в которой реализован и использован описанный выше класс Man (листинг 10.1).

Листинг 10.1. Проект Task10_1

```
// ////////////////////////////////////// Man.h //////////////////////////////////////
const int l_name = 30;
const int l_year = 5;
const int l_pay = 10;
const int l_buf = l_name + l_year + l_pay;
class Man {
public:
    Man(int lName = 30);
    ~Man();
    bool CompareName(const char*) const;
    int GetBirthYear() const { return birth_year; }
    float GetPay() const { return pay; }
    char* GetName() const { return pName; }
    void Print() const;
    void SetBirthYear(const char*);
    void SetName(const char*);
    void SetPay(const char*);
private:
    char* pName;
    int birth_year;
    float pay;
};
// ////////////////////////////////////// Man.cpp //////////////////////////////////////
#include <iostream>
#include <cstring>
#include "Man.h"
using namespace std;
Man::Man( int lName ) {
    cout << "Constructor is working... ";
    pName = new char[lName + 1];
}
Man::~~Man() {
    cout << "Destructor is working...";
    delete [] pName;
}
void Man::SetName(const char* fromBuf) {
    strncpy(pName, fromBuf, l_name);
    pName[l_name] = 0;
}
void Man::SetBirthYear(const char* fromBuf) {
    birth_year = atoi(fromBuf + l_name);
}
```

```

void Man::SetPay(const char* fromBuf) {
pay = atof(fromBuf + l_name + l_year);
}
bool Man::CompareName(const char* name) const {
if ((strstr(pName, name)) && (pName[strlen(name)] == ' ')) return true;
else return false;
}
void Man::Print() const {
cout << pName << birth_year << ' ' << pay << endl;
}
////////// Main.cpp //////////
#include <Windows.h> // содержит прототип OemToChar
#include <iostream>
#include <fstream>
#include "man.cpp"
using namespace std;
const char filename[] = "dbase.txt";
int main() {
    const int maxn_record = 10;
    Man man[maxn_record];
    char buf [l_buf + 1];
    char name[l_name + 1];
    setlocale( LC_ALL, "Russian" );
    ifstream fin(filename);
    if (!fin) { cout << "Нет файла " << filename << endl; return 1; }
    int k = 0;
    while (fin.getline( buf, l_buf )) {
        if (k >= maxn_record) { cout << "Слишком длинный файл"; return 1; }
        man[k].SetName( buf );
        man[k].SetBirthYear( buf );
        man[k].SetPay( buf );
        k++;
    }
    int n_record = k;    float mean_pay = 0 , n_man = 0;    while (true) {
        cout << "Введите фамилию или слово end: ";
        cin >> name;
        OemToChar(name, name); // для ввода кириллицы
        if (0 == strcmp(name, "end")) break;
        bool not_found = true;
        for (int i = 0; i < n_record; ++ i ) {
            if (man[i].CompareName(name)) {
                man[i].Print();
                n_man++; mean_pay += man[i].GetPay();
                not_found = false;
                break;
            }
        }
        if (not_found) cout << "Такого сотрудника нет" << endl;
    }
    if (n_man) cout << " Средний оклад: " << mean_pay / n_man << endl;
}

```

Обратите внимание на следующие моменты.

Константные методы. Заголовки методов класса, которые *не должны изменять поля* класса, снабжены модификатором `const` после списка параметров. Если вы по ошибке попытаетесь в теле метода что-либо присвоить полю класса, компилятор не позволит вам это сделать. Другое достоинство ключевого слова `const` — облегчение сопровождения программы. Например, если обнаружено некорректное поведение приложения и выяснено, что «кто-то» портит одно из полей объекта, можно сразу исключить из списка подозреваемых константные методы класса. Поэтому использование `const` в объявлениях методов, не изменяющих объект, считается хорошим стилем программирования.

Отладочная печать в конструкторе и деструкторе. Вывод сообщений типа «Constructor is working», «Destructor is working» очень помогает на начальном этапе освоения классов и при поиске труднодиагностируемых ошибок.

Отладка программы. С технологией создания многофайловых проектов вы уже знакомы по первой части практикума. Создайте проект с именем Task10_1 и добавьте к нему файлы из листинга 10.1.

Поместите в текущий каталог проекта файл с базой данных `dbase.txt`, заполнив его соответствующей информацией, например:

Иванов И.П.	1957	4200
Петров А.Б.	1947	3800
Сидоров С.С.	1938	3000
Ивановский Р.Е.	1963	6200

Скомпилируйте и запустите программу на выполнение. Вы должны увидеть следующий вывод в консольном окне программы:

```
Constructor is working... Constructor is working... Constructor is working...
Constructor is working... Constructor is working... Constructor is working...
Constructor is working... Constructor is working... Constructor is working...
Constructor is working... Введите фамилию или слово end:
```

Благодаря отладочной печати мы можем проконтролировать, как создаются все 10 элементов массива `man`. Продолжим тестирование программы. Введите с клавиатуры текст: «Сидоров». После нажатия `Enter` вы должны увидеть на экране новый текст:

```
Сидоров С.С.      1938    3000 Введите фамилию или слово end:
Введите с клавиатуры текст «Петров» и нажмите Enter. Появятся две строки:
Петров А.Б.      1947    3800 Введите фамилию или слово end:
Введите с клавиатуры текст «end». После нажатия Enter появится вывод:
Средний оклад: 3400 Destructor is working... Destructor is working... Destructor is
working...
Destructor is working... Destructor is working... Destructor is working...
Destructor is working... Destructor is working... Destructor is working...
Destructor is working... Press any key to continue
```

Все верно! После ввода `end` программа покидает цикл `while (true)` и печатает величину среднего оклада для вызванных ранее записей из базы данных. Затем программа завершается, а при выходе из области видимости все объекты вызывают свои деструкторы, и мы это тоже четко наблюдаем.

ПРИМЕЧАНИЕ В реальных программах доступ к полям класса с помощью методов преследует еще одну важную цель — проверку заносимых значений. В качестве упражнения дополните методы `SetBirthYear` и `SetPay` проверками успешности преобразования из строки в число и осмысленности получившегося результата (например, на неотрицательность).

Итак, мы завершили перестройку структурной программы в программу, реализующую концепции ООП. Рекомендуем вам сравнить решение задачи 6.1 с новым, начав со ввода исходных данных. В новой программе все понятно без лишних комментариев. В старой — нужно разбираться на уровне клиента с реализацией более низкого уровня. Перейдем теперь к рассмотрению других аспектов, необходимых для создания хорошо спроектированного класса

Конструктор по умолчанию

Обратите внимание, что в списке параметров конструктора класса `Man` параметр `lName` имеет значение по умолчанию. Если все параметры конструктора имеют значения по умолчанию или если конструктор вообще не имеет параметров, он называется *конструктором по умолчанию*. Такой конструктор имеет несколько специальных областей применения.

Во-первых, он используется при описании массива объектов, например: `Man man[25]`. Здесь каждый из 25 объектов создается путем вызова конструктора по умолчанию. В случае отсутствия конструктора по умолчанию вы *не сможете объявлять массивы* объектов этого класса (если вы не определите в классе вообще ни одного конструктора, то компилятор создаст конструктор по умолчанию самостоятельно). Во-вторых, вы *не сможете использовать* объекты этого класса *в качестве полей другого класса*. Еще одно применение конструкторов по умолчанию относится к механизму наследования классов (см. след. л/р).

Инициализаторы конструктора

Существует альтернативный способ инициализации отдельных частей объекта в конструкторе — с помощью *списка инициализаторов*, расположенного после двоеточия между заголовком и телом конструктора. Каждый инициализатор имеет вид `имя_поля (выражение)` и обеспечивает присвоение полю объекта значения указанного выражения. Если инициализаторов несколько, они разделяются запятыми.

Если полем объекта является объект другого класса, для его инициализации будет вызван соответствующий конструктор.

Рассмотрим класс `Point`, задающий точку на плоскости в декартовых координатах:

```
class Point {
    double x, y; public:
    Point( double _x = 0, double _y = 0 ) { x = _x; y = _y; }
    // Point(double _x = 0, double _y = 0) : x(_x), y(_y) {} // 1
    . . .
};
```

В *операторе 1* записан вариант со списком инициализации. Тело конструктора пустое, а действия по присваиванию начальных значений полям перенесены в инициализаторы. Кстати, в нашем примере все параметры конструктора имеют значения по умолчанию, благодаря чему он может использоваться и как конструктор с параметрами, и как конструктор по умолчанию.

В этом примере может быть выбран любой вариант конструктора, однако есть три ситуации, в которых инициализация полей объекта возможна *только с использованием инициализаторов конструктора*: для констант, для ссылок и для полей, являющихся объектами класса, в котором есть один или более конструкторов, но отсутствует конструктор по умолчанию.

Инициализация элементов класса с помощью списка инициализаторов является более эффективной по быстродействию, чем присваивание начальных значений в теле конструктора. Это объясняется особенностями *порядка инициализации объектов*. Рассмотрим этот порядок на примере. В общем случае класс содержит как поля стандартных типов, так и поля, являющиеся объектами других классов:

```
class Coo {
    int x1;
    double x2;
    Point p1, p2; public:
    Coo( int _x1, double _x2 ) { x1 = _x1; x2 = _x2; }
    . . .
};
```

При создании объекта класса `Coo` сначала вызываются конструкторы объектных полей (конструкторы по умолчанию класса `Point` для полей `p1` и `p2`), а затем — конструктор класса `Coo`.

Если конструктор класса имеет список инициализаторов, то порядок инициализации включенных в него объектов определяется не самим списком, а тем, в каком порядке эти поля объявлены в теле класса. Поэтому во избежание путаницы рекомендуется записывать список инициализаторов в порядке, соответствующем объявлению.

Если элемент класса, являющийся объектом, инициализируется не в списке инициализаторов, а в теле конструктора, то сначала он получает значение при помощи конструктора по умолчанию своего класса, а потом это значение изменяется в теле конструктора (ясно, что эффективность при этом ухудшается).

Конструктор копирования

Конструктор так же, как и остальные методы класса, можно перегружать. Одной из важнейших форм перегружаемого конструктора является *конструктор копирования* (*copy constructor*). Он вызывается в трех ситуациях:

• при инициализации нового объекта другим объектом в операторе объявления; • при передаче объекта в функцию в качестве аргумента по значению; • при возврате объекта из функции по значению.

Если при объявлении класса конструктор копирования не задан, компилятор создает *конструктор копирования по умолчанию*, который дублирует объект, выполняя побайтное копирование его полей. Однако при этом возможно появление проблем, связанных с копированием указателей. Например, если объект, передаваемый в функцию в качестве аргумента по значению, содержит указатель `pMem` на выделенную область памяти, то в копии объекта этот указатель будет ссылаться на ту же самую область. При возврате из функции вызывается деструктор копии объекта, освобождающий память, на которую указывает `pMem`. При выходе из области видимости исходного объекта его деструктор попытается еще раз освободить память, на которую указывает `pMem`. Результат обычно весьма плачевный.

Чтобы исключить нежелательные побочные эффекты в каждой из трех перечисленных ситуаций, необходимо создать конструктор копирования, в котором реализуется необходимый контроль за созданием копии объекта.

Конструктор копирования имеет следующую общую форму:

```
имя_класса( const имя_класса & obj ) {
    // тело конструктора
}
```

Здесь `obj` — ссылка на объект, для которого должна создаваться копия. Например, пусть имеется класс `Сoo`, а `y` — объект типа `Сoo`. Тогда следующие операторы вызовут конструктор копирования класса `Сoo`:

```
Сoo x = y;           // y явно инициализирует x
Func1( y );          // y передается в качестве аргумента по значению
y = Func2();          // y получает возвращаемый объект
```

В двух первых случаях конструктору копирования передается ссылка на `y`. В последнем случае конструктору копирования передается ссылка на объект, возвращаемый функцией `Func2`. Пример реализации конструктора копирования будет показан при решении задачи 10.2.

Перегрузка операций

Любая операция, определенная в C++, за исключением `::`, `?:`, `.`, `.*`, `#`, `##`, может быть перегружена для созданного вами класса. Это делается с помощью функций специального вида, называемых *функциями-операциями* (операторными функциями): возвращаемый_тип operator # (список параметров) { тело функции }

Вместо знака `#` ставится знак перегружаемой операции. Функция-операция может быть реализована либо как функция класса, либо как внешняя (обычно дружественная) функция. В первом случае количество параметров у функции-операции на единицу меньше, так как первым операндом при этом считается сам объект, вызвавший данную операцию.

Покажем два варианта перегрузки операции сложения для класса `Point`:

```
class Point {                                     // Вариант 1: в форме метода класса
    double x, y;
public:
    //...
    Point operator +( Point& );
};
Point Point::operator +( Point& p ) {
    return Point( x + p.x, y + p.y );
}
class Point {                                     // Вариант 2: в форме внешней глобальной функции
    //...
    friend Point operator +( Point&, Point& );
};
Point operator +( Point& p1, Point& p2 ) {
    return Point( p1.x + p2.x, p1.y + p2.y ); }
```

Внешняя глобальная функция, как правило, объявляется дружественной классу, чтобы иметь доступ к его закрытым элементам. Независимо от формы реализации операции «+» теперь можно написать

```
Point p1( 0, 2 ), p2( -1, 5 );
Point p3 = p1 + p2;
```

Важно понимать, что, встретив выражение `p1 + p2`, компилятор в случае первой формы перегрузки вызовет метод `p1.operator +(p2)`, а в случае второй формы перегрузки — глобальную функцию `operator +(p1, p2)`. Результатом выполнения данных операторов будет точка `p3` с координатами `x = -1`, `y = 7`. Заметим, что для инициализации объекта `p3` будет вызван конструктор копирования по умолчанию, но он нас устраивает, поскольку в классе нет полей-указателей.

Какую из двух форм следует выбирать? Используйте перегрузку в форме метода класса, если нет каких-либо причин, препятствующих этому. Например, если первый аргумент (левый операнд) относится к одному из базовых типов (к примеру, `int`), то перегрузка операции возможна только в форме внешней функции.

Перегрузка операций инкремента

Операция инкремента имеет две формы: *префиксную* и *постфиксную*. Для первой формы сначала изменяется состояние объекта в соответствии с операцией, а затем объект используется в выражении. Для второй формы объект используется в том состоянии, которое он имел до начала операции, а потом его состояние изменяется.

Чтобы компилятор смог различить эти две формы, для них используются разные сигнатуры. Покажем реализацию операций инкремента на примере класса `Point`:

```

Point& Point::operator ++() {           // префиксный инкремент
    x++;    y++;
    return *this;
}
Point Point::operator ++( int ) {       // постфиксный инкремент
    Point old = *this;
    x++;    y++;
    return old; }

```

В префиксной операции выполняется возврат результата по ссылке. Это предотвращает вызов конструктора копирования для создания возвращаемого значения и последующий вызов деструктора. В постфиксной операции возврат по ссылке не подходит, поскольку необходимо вернуть первоначальное состояние объекта, сохраненное в локальной переменной `old`. Таким образом, *префиксный инкремент является более эффективной операцией, чем постфиксный инкремент*.

Заметим, что ранее мы спокойно пользовались постфиксной формой инкремента, например, в заголовке цикла `for`, поскольку для переменных встроеного типа разницы в эффективности нет. Когда мы будем применять контейнерные классы стандартной библиотеки, параметр в заголовке цикла часто будет представлять объект-итератор, и для него префиксная форма будет более эффективной.

ПРИМЕЧАНИЕ Все сказанное об операции инкремента относится также и к *операции декремента*.

Перегрузка операции присваивания

О перегрузке этой операции следует поговорить особо по нескольким причинам. Во-первых, если вы не определите эту операцию в разрабатываемом классе, *компилятор создаст операцию присваивания по умолчанию*, которая выполняет поэлементное копирование объекта. При этом возможны те же проблемы, что возникают при использовании конструктора копирования по умолчанию (см. выше). Поэтому запомните правило: если в классе требуется определить конструктор копирования, его верной спутницей должна быть операция присваивания, и наоборот. Во-вторых, операцию присваивания *можно определить только в форме метода класса*, и, в-третьих, *она не наследуется*, в отличие от всех остальных операций. Определим операцию присваивания для класса `Man` из задачи 10.1:

```

class Man {           // Man.h (интерфейс класса)
public:
    // . . .
    Man& operator =( const Man& );           // операция присваивания
private:
    char* pName;
    // . . .
};

// Man.cpp (реализация класса)
Man& Man::operator =( const Man& man ) {
    if ( this == &man ) return *this;           // проверка на самоприсваивание
    delete [] pName;                             // уничтожение предыдущего значения
    pName = new char[strlen( man.pName ) + 1];
    strcpy( pName, man.pName );
    birth_year = man.birth_year;
    pay = man.pay;
    return *this;
}

```

Обратите внимание на несколько важных моментов:

убедитесь, что не выполняется присваивание вида `x = x;`. Если левая и правая части ссылаются на один и тот же объект, то делать ничего не надо. Если не перехватить этот особый случай, то следующий шаг уничтожит значение, на которое указывает `pName`, еще до того, как оно будет скопировано;

удалите предыдущие значения полей в динамически выделенной памяти;

выделите память под новые значения полей и скопируйте в нее эти значения;

возвратите значение объекта, на которое указывает `this` (то есть `*this`).

Статические элементы класса

Если необходимо создать переменную, значение которой будет общим для всех объектов данного класса, следует *объявить* в классе поле с модификатором `static` и дать его *определение* в глобальной области видимости программы:

```

class Coo {
    static int count;           // объявление в классе
    // остальной код
};

```

```
int Coo::count = 1;           // определение и инициализация
// int Coo::count;           // по умолчанию инициализируется нулем
```

Просто воспользоваться для этой цели глобальной переменной будет неграмотно, это нарушит принцип инкапсуляции данных. Аналогично статическим полям, могут быть объявлены и статические методы класса (с модификатором `static`). Они могут обращаться непосредственно только к статическим полям и вызывать только другие статические методы класса, потому что им не передается скрытый указатель `this`. Статические методы не могут быть константными (`const`) и виртуальными (`virtual`). Обращение к статическим методам производится так же, как к статическим полям — либо через имя класса, либо, если хотя бы один объект класса уже создан, через имя объекта.

Все рассмотренные выше аспекты найдут применение при решении задачи 10.2.

Задача 10.2. Реализация класса треугольников

Для некоторого множества заданных координатами своих вершин треугольников найти треугольник максимальной площади (если максимальную площадь имеют несколько треугольников, найти первый из них). Предусмотреть возможность перемещения треугольников и проверки включения одного треугольника в другой. Программа должна содержать меню, позволяющее выполнить проверку всех методов класса.

Для реализации этой задачи составить описание класса треугольников на плоскости. Предусмотреть возможность объявления в клиентской программе (`main`) экземпляра треугольника с заданными координатами вершин. Предусмотреть наличие в классе методов, обеспечивающих: 1) перемещение треугольников на плоскости; 2) определение отношения $>$ для пары заданных треугольников (мера сравнения — площадь треугольников); 3) определение отношения включения типа: «Треугольник 1 входит (не входит) в Треугольник 2».

Сейчас мы еще не готовы провести полномасштабную объектно-ориентированную декомпозицию программы: не хватает информации из следующего семинара.

Поэтому применим гибридный подход: разработку главного клиента `main` проведем по технологии функциональной декомпозиции, а функции-серверы, вызываемые из `main`, будут использовать объекты.

Начнем с выявления основных понятий (классов) для нашей задачи. Первый очевидный класс `Triangle` необходим для представления треугольников. Из нескольких способов определения треугольника на плоскости выберем самый простой — через три точки, задающие его вершины. Этот выбор опирается на понятие, отсутствующее в условии задачи, — понятие *точки*. Точку на плоскости также можно представить разными способами; остановимся на представлении парой вещественных чисел, задающих координаты точки по осям x и y .

Таким образом, с понятием точки связывается как минимум пара атрибутов. Если же представить, что можно делать с объектом типа точки — например, перемещать ее на плоскости или определять ее вхождение в заданную фигуру, — то становится очевидной целесообразность определения класса `Point`.

Итак, объектно-ориентированная декомпозиция выявила два класса: `Triangle` и `Point`. Как они должны быть связаны друг с другом? На следующем семинаре взаимоотношения между классами будут рассматриваться подробно, а пока отметим, что чаще всего для двух классов имеет место одно из двух отношений: *наследования* (отношение *is a*) и *агрегации*, или *включения* (отношение *has a*).

Если класс B является частным случаем класса A , то говорят, что B *is a* A (например, класс треугольников есть частный вид класса многоугольников: `Triangle is a Polygon`). Если класс A содержит в себе объект класса B , то говорят, что A *has a* B (например, класс треугольников может содержать в себе объекты класса точек: `Triangle has a Point`).

Займемся теперь основным клиентом — `main`. Здесь мы применим функциональную декомпозицию, или технологию нисходящего проектирования: представим алгоритм как последовательность подзадач, каждой из которых соответствует вызываемая серверная функция. На начальном этапе проектирования тела этих функций могут быть заполнены «заглушками» (отладочной печатью). Если в какой-то серверной функции окажется слабое сцепление, она тоже разбивается на несколько подзадач.

То же самое выполняется и с классами, используемыми в программе: по мере реализации подзадач они пополняются необходимыми для этого методами. Такая технология облегчает отладку и поиск ошибок, сокращая общее время разработки.

В соответствии с описанной технологией представим решение задачи 10.2 как последовательность нескольких этапов. Иногда как бы по забывчивости мы будем допускать некоторые «ляпы», поскольку в процессе поиска ошибок можно глубже понять нюансы программирования с классами.

На первом этапе напишем код (листинг 10.2) для начального представления классов Point и Triangle, достаточный для того, чтобы создать несколько объектов типа Triangle и реализовать первый пункт меню — вывод всех объектов на экран.

Этап 1

Листинг 10.2. Проект Task10_2

```
//////////////////////////////////// Point.h //////////////////////////////////
#ifndef POINT_H
#define POINT_H
class Point {
public:
    double x, y;
    Point(double _x = 0, double _y = 0) : x(_x), y(_y) {}           // Конструктор
    // Другие методы
    void Show() const;
    double DistanceTo(Point) const;                                // Расстояние до другой точки
};
#endif /* POINT_H */
//////////////////////////////////// Point.cpp //////////////////////////////////
#include <iostream>
#include <cmath>
#include "Point.h"
using namespace std;
void Point::Show() const { cout << " (" << x << ", " << y << ")"; }
double Point::DistanceTo(Point sp) const {
    return sqrt((x - sp.x) * (x - sp.x) + (y - sp.y) * (y - sp.y));
}
//////////////////////////////////// Triangle.h //////////////////////////////////
#ifndef TRIANGLE_H
#define TRIANGLE_H
#include "Point.h"
class Triangle {
public:
    static int count;                                             // Количество созданных объектов
    Triangle(Point, Point, Point, const char*);                // Конструктор
    Triangle(const char*);                                       // Конструктор пустого (нулевого) треугольника
    ~Triangle();                                                 // Деструктор
    Point Get_v1() const { return v1; }                        // Получить значение v1
    Point Get_v2() const { return v2; }                        // Получить значение v2
    Point Get_v3() const { return v3; }                        // Получить значение v3
    char* GetName() const { return name; }                     // Получить имя объекта
    void Show() const;                                          // Показать объект
private:
    char* objID;                                                 // Идентификатор объекта
    char* name;                                                  // Наименование треугольника
    Point v1, v2, v3;                                           // Вершины
    double a;                                                    // Сторона, соединяющая v1 и v2
    double b;                                                    // Сторона, соединяющая v2 и v3
    double c;                                                    // Сторона, соединяющая v1 и v3
};
#endif /* TRIANGLE_H */
//////////////////////////////////// Triangle.cpp //////////////////////////////////
// Реализация класса Triangle
#include <cmath>
#include <iostream>
#include <iomanip>
#include <string.h>
#include "Point.cpp"
#include "Triangle.h"
using namespace std;
Triangle::Triangle(Point _v1, Point _v2, Point _v3, const char* ident)
    : v1(_v1), v2(_v2), v3(_v3) {
    char buf[16];
    objID = new char[strlen(ident) + 1];
    strcpy(objID, ident);
    count++;
    sprintf(buf, "Треугольник %d", count);
    name = new char[strlen(buf) + 1];
    strcpy(name, buf);
}
```

```

        a = v1.DistanceTo(v2);
        b = v2.DistanceTo(v3);
        c = v1.DistanceTo(v3);
        cout << "Constructor_1 for: " << objID << " (" << name << ")\n"; // для отладки
    }
Triangle::Triangle(const char* ident) {
    char buf[16];
    objID = new char[strlen(ident) + 1];
    strcpy(objID, ident);
    count++;
    sprintf(buf, "Треугольник %d", count);
    name = new char[strlen(buf) + 1];
    strcpy(name, buf);
    a = b = c = 0;
    cout << "Constructor_2 for: " << objID << " (" << name << ")\n"; // для отладки
}
Triangle::~Triangle() {
    cout << "Destructor for: " << objID << endl;
    delete [] objID;    delete [] name;
}
void Triangle::Show() const {
    cout << name << ":";
    v1.Show();    v2.Show();    v3.Show();
    cout << endl;
}
////////// Main.cpp //////////
#include <iostream>
#include "Triangle.cpp"
using namespace std;
int Menu();
int GetNumber(int, int);
void ExitBack();
void Show(Triangle* [], int);
void Move(Triangle* [], int);
void FindMax(Triangle* [], int);
void IsIncluded(Triangle* [], int);
int Triangle::count = 0; // Инициализация глобальной переменной // -----
----- Главная функция
int main() {
    Point p1(0, 0);    Point p2(0.5, 1);    // Определения точек
    Point p3(1, 0);    Point p4(0, 4.5);
    Point p5(2, 1);    Point p6(2, 0);
    Point p7(2, 2);    Point p8(3, 0);
    Triangle triaA(p1, p2, p3, "triaA");    // Определения треугольников
    Triangle triaB(p1, p4, p8, "triaB");
    Triangle triaC(p1, p5, p6, "triaC");
    Triangle triaD(p1, p7, p8, "triaD");
    // Определение массива указателей на треугольники
    Triangle* pTria[] = { &triaA, &triaB, &triaC, &triaD };
    int n = sizeof (pTria) / sizeof (pTria[0]);
    setlocale( LC_ALL, "Russian" );
    bool done = false;
    while (!done) { // Главный цикл
        switch (Menu()) {
            case 1: Show(pTria, n);    break;
            case 2: Move(pTria, n);    break;
            case 3: FindMax(pTria, n);    break;
            case 4: IsIncluded(pTria, n);    break;
            case 5: cout << "Конец работы." << endl; done = true;    break;
        }
    }
}
int Menu() { // ----- Вывод меню
    cout << "\n===== Главное меню===== " << endl;
    cout << "1 - вывести все объекты\t 3 - найти максимальный" << endl;
    cout << "2 - переместить\t\t 4 - определ. отношение включения" << endl;
    cout << "\t\t 5 - выход" << endl;
    return GetNumber(1, 5);
}

```

```

int GetNumber(int min, int max) { // ----- Ввод целого числа в заданном диапазоне
    int number = min - 1;    while (true) {                cin >> number;
        if ((number >= min) && (number <= max) && (cin.peek() == '\n'))
            break;
        else {
            cout << "Повторите ввод (ожидается число от " << min
                << " до " << max << "):" << endl;
            cin.clear();
            while (cin.get() != '\n') {};
        }
    }
    return number;
}

void ExitBack() { // ----- Возврат в функцию с основным меню
    cout << "Нажмите Enter." << endl;
    cin.get(); cin.get();
}

void Show(Triangle* p_tria[], int k) { // ----- Вывод всех треугольников
    cout << "===== Перечень треугольников =====" << endl;
    for (int i = 0; i < k; ++i) p_tria[i]->Show();
    ExitBack();
}

void Move(Triangle* p_tria[], int k) { // ----- Перемещение
    cout << "===== Перемещение =====" << endl;
    // здесь будет код функции...
    ExitBack();
}

void FindMax(Triangle* p_tria[], int k) { // ----- Поиск максимального треугольника
    cout << "=== Поиск максимального треугольника ===" << endl;
    // здесь будет код функции...
    ExitBack();
}

void IsIncluded(Triangle* p_tria[], int k) { // --- Определение отношения включения
    cout << "===== Отношение включения =====" << endl;
    // здесь будет код функции...
    ExitBack();
}

```

Рекомендуем вам обратить внимание на следующие моменты в проекте Task10_2.

Класс Point (файлы Point.h, Point.cpp). Реализация класса содержит единственный метод Show, назначение которого — показать объект типа Point на экране. Заметим, что при решении реальных задач в какой-либо графической оболочке метод Show действительно нарисовал бы точку. Однако мы связаны стандартом C++, поэтому демонстрация точки сводится к выводу ее координат.

Класс Triangle (файлы Triangle.h, Triangle.cpp).

- Назначение большинства полей и методов очевидно из их имен и комментариев.
- Поле count исполняет роль глобального счетчика создаваемых объектов (свойство глобальности обеспечивается благодаря модификатору static); мы сочли удобным в конструкторах генерировать имена треугольников автоматически: «Треугольник 1», «Треугольник 2», и т. д., используя текущее значение count (возможны и другие способы именования треугольников).
- Поле char* objID введено для целей отладки и обучения; вскоре вы увидите, что благодаря отладочным операторам печати в конструкторах и деструкторе удобно наблюдать за созданием и уничтожением объектов.
- Конструктор пустого (нулевого) треугольника нужен для создания временных объектов, которые могут модифицироваться с помощью присваивания.
- Метод Show — к сожалению, здесь нам тоже не удастся нарисовать треугольник на экране; вместо этого печатаются координаты его вершин.
- Назначение большинства полей и методов очевидно из их имен и комментариев.
- Поле count исполняет роль глобального счетчика создаваемых объектов (свойство глобальности обеспечивается благодаря модификатору static); мы сочли удобным в конструкторах генерировать имена треугольников автоматически: «Треугольник 1», «Треугольник 2», и т. д., используя текущее значение count (возможны и другие способы именования треугольников).

- Поле `char* objID` введено для целей отладки и обучения; вскоре вы увидите, что благодаря отладочным операторам печати в конструкторах и деструкторе удобно наблюдать за созданием и уничтожением объектов.
- Конструктор пустого (нулевого) треугольника нужен для создания временных объектов, которые могут модифицироваться с помощью присваивания.
- Метод `Show` — к сожалению, здесь нам тоже не удастся нарисовать треугольник на экране; вместо этого печатаются координаты его вершин.
- **Основной модуль** (файл `Main.cpp`).
- Инициализация глобальных переменных: обратите внимание на оператор `int Triangle::count = 0;` — если вы его забудете, компилятор обидится.
- Функция `main`:
 - определения восьми точек `p1, ..., p8` сделаны произвольно, но так, чтобы из них можно было составить треугольники;
 - определения четырех треугольников сделаны тоже произвольно, впоследствии на них будут демонстрироваться основные методы класса;
 - далее определяются массив указателей `Triangle* pTria[]` с адресами объявленных выше треугольников и его размер `n`; в таком виде удобно передавать адрес `pTria` и величину `n` в вызываемые серверные функции.
- Функция `Menu` — после вывода на экран списка пунктов меню вызывается функция `GetNumber`, возвращающая номер пункта, введенный пользователем с клавиатуры. Отчего было просто не написать: `cin >> number;`? Но тогда мы не обеспечили бы защиту программы от непреднамеренных ошибок при вводе. Вообще-то вопрос надежного ввода чисел с клавиатуры подробно разбирается на семинаре 13 при решении задачи 13.1.
- Функция `Show` выводит на экран перечень всех треугольников. В завершение вызывается функция `ExitBack`, которая обеспечивает заключительный диалог с пользователем после обработки очередного пункта меню.
- Остальные функции по обработке других пунктов меню выполнены в виде заглушек, выводящих только наименование соответствующего пункта.

Тестирование и отладка первой версии программы.

После компиляции и запуска программы вы должны увидеть на экране следующее:

```
Constructor_1 for: triaA (Треугольник 1)
Constructor_1 for: triaB (Треугольник 2)
Constructor_1 for: triaC (Треугольник 3)
Constructor_1 for: triaD (Треугольник 4)
```

```
===== Г л а в н о е   м е н ю =====
1      - вывести все объекты      3 - найти максимальный
2      - переместить              4 - определить отношение включения
                                   5 - выход
```

Введите с клавиатуры цифру 1³. Программа выведет:

```
1
===== Перечень треугольников =====
Треугольник 1:  (0, 0)  (0.5, 1)  (1, 0)
Треугольник 2:  (0, 0)  (0, 4.5)  (3, 0)
Треугольник 3:  (0, 0)  (2, 1)  (2, 0)
Треугольник 4:  (0, 0)  (2, 2)  (3, 0)
```

Нажмите Enter.

Выбор первого пункта меню проверен. Нажмите Enter. Программа выведет:

```
===== Г л а в н о е   м е н ю =====
```

. . .
Теперь проверим выбор второго пункта меню. После ввода цифры 2 на экране должно появиться:

```
2
===== Перемещение =====
```

Нажмите Enter.

³ После ввода числовой информации всегда подразумевается нажатие Enter.

Все правильно. После нажатия Enter программа возвращается в главное меню. Для проверки ввода ошибочного символа введите с клавиатуры любой буквенный символ и нажмите Enter. Программа должна сообщить:

Повторите ввод (ожидается число от 1 до 5):

Проверяем завершение работы. Введите цифру 5. Программа выведет:

```
5
Конец работы.
Destructor for: triaD
Destructor for: triaC
Destructor for: triaB
Destructor for: triaA
```

Тестирование закончено. Обратите внимание на то, что деструкторы объектов вызываются в порядке, обратном вызову конструкторов.

Этап 2

Добавим в классы Point и Triangle методы, обеспечивающие перемещение треугольников, а в основной модуль — реализацию функции Move. Внесите следующие изменения в тексты модулей проекта.

1. **Point.h.** добавьте объявление операции-функции «+=», которая будет использована для реализации метода Move в классе Triangle:

```
void operator +=( Point& );
```

2. **Point.cpp.** Добавьте код реализации данной функции:

```
void Point::operator +=( Point& p ) { x += p.x; y += p.y; }
```

3. **Triangle.h.** Добавьте объявление метода:

```
void Move( Point );
```

4. **Triangle.cpp.** Добавьте код метода Move:

```
void Triangle::Move(Point dp) {
    v1 += dp;    v2 += dp;    v3 += dp;
}
```

5. **Main.cpp.** В список прототипов функций добавьте сигнатуру: `double GetDouble();`

Добавьте в конец файла новую функцию GetDouble. Она предназначена для ввода вещественного числа и вызывается из функции Move. В ней предусмотрена защита от ввода недопустимых символов аналогично функции GetNumber:

```
double GetDouble() {
    double value;
    while (true) {
        cin >> value;
        if (cin.peek() == '\n') break;
        else {
            cout << "Повторите ввод (ожидается веществ. число):" << endl;
            cin.clear();
            while (cin.get() != '\n') {};
        }
    }
    return value;
}
```

Замените заглушку функции Move следующим кодом:

```
void Move(Triangle* p_tria[], int k) {
    cout << "===== Перемещение =====" << endl;
    cout << "Введите номер треугольника (от 1 до " << k << "): ";
    int i = GetNumber( 1, k ) - 1;
    p_tria[i]->Show();
    Point dp;
    cout << "Введите смещение по x: ";
    dp.x = GetDouble();
    cout << "Введите смещение по y: ";
    dp.y = GetDouble();
    p_tria[i]->Move( dp );
    cout << "Новое положение треугольника:" << endl;
    p_tria[i]->Show();    ExitBack();
}
```

```
}
```

Выполнив компиляцию проекта, проведите его тестирование аналогично тестированию на первом этапе. После выбора второго пункта меню и ввода данных, задающих номер треугольника, величину сдвига по x и величину сдвига по y , вы должны увидеть на экране примерно следующее (жирным шрифтом выделено то, что вводилось с клавиатуры):

2

```
===== Перемещение =====
Введите номер треугольника (от 1 до 4): 1
Треугольник 1: (0, 0) (0.5, 1) (1, 0)
Введите смещение по x: 2.5
Введите смещение по y: -7
Новое положение треугольника:
Треугольник 1: (2.5, -7) (3, -6) (3.5, -7)
Нажмите Enter.
```

Этап 3

На этом этапе мы добавим в класс `Triangle` метод, обеспечивающий сравнение треугольников по их площади, а в основной модуль — реализацию функции `FindMax`. Внесите следующие изменения в тексты модулей проекта:

1. **Triangle.h:** добвьте объявление метода `Area` и функции-операции `>`:

```
double Area() const; // площадь объекта
bool operator >(const Triangle&) const;
```

2. **Triangle.cpp:** добавьте код реализации метода `Area` и функции-операции `>`:

```
double Triangle::Area() const {
    double p = (a + b + c) / 2;
    return sqrt(p * (p - a) * (p - b) * (p - c));
}

bool Triangle::operator >(const Triangle& tria) const {
    if (Area() > tria.Area()) return true;
    else return false; }
```

3. **Main.cpp:** замените заглушку функции `FindMax` следующим кодом:

```
// ----- поиск максимального треугольника
void FindMax(Triangle* p_tria[], int k) {
    cout << "=== Поиск максимального треугольника ===" << endl;
    // Создаем объект triaMax, который по завершению поиска будет идентичен максимальному
    // объекту. Инициализируем его значением 1-го объекта из массива:
    Triangle triaMax("triaMax");
    triaMax = *p_tria[0];
    for (int i = 1; i < 4; ++i)
        if (*p_tria[i] > triaMax) triaMax = *p_tria[i];
    cout << "Максимальный треугольник: " << triaMax.GetName() << endl;
    ExitBack();
}
```

Откомпилируйте программу и запустите. Выберите третий пункт меню. На экране должно появиться:

3

```
=== Поиск максимального треугольника ===
Constructor for: triaMax (Треугольник 5)
Максимальный треугольник: Треугольник 2
Нажмите Enter.
```

Максимальный треугольник найден правильно. После нажатия `Enter` должен появиться текст:

```
Destructor for: triaB
===== Г л а в н о е   м е н ю =====
- вывести все объекты      3 - найти максимальный
- переместить              4 - определить отношение включения
                          5 - выход
```

Чтобы завершить работу, введите цифру 5 и нажмите `Enter`.

Но... вместо нормального завершения программа ломается, а операционная система выдает сообщение об ошибке примерно следующего вида:

```
Debug Assertion Failed!
Program: . . .
```



```

if (&tria == this) return *this;
delete [] name;
name = new char[strlen(tria.name) + 1];
strcpy(name, tria.name);
v1 = tria.v1; v2 = tria.v2; v3 = tria.v3;
a = tria.a; b = tria.b; c = tria.c;
return *this;
}

```

И в конструкторе копирования, и в операции присваивания перед копированием содержимого полей, на которые указывают поля типа `char*`, для них выделяется новая память. Обратите внимание, что в конструкторе копирования после переписи поля `objID` мы добавляем в конец этой строки текст (копия). А в операции присваивания это поле, идентифицирующее объект, вообще не затрагивается и остается в своем первоначальном значении. Все это полезно для целей отладки.

Откомпилируйте и запустите программу. Выберите третий пункт меню. На экране должен появиться текст:

```

3
=== Поиск максимального треугольника ===
Constructor_2 for: triaMax (Треугольник 5)
Assign operator: triaMax = triaA
Assign operator: triaMax = triaB
Максимальный треугольник: Треугольник 2
Нажмите Enter.

```

Обратите внимание на отладочный вывод операции присваивания. Продолжим тестирование. После нажатия `Enter` программа выведет

```

Destructor for: triaMax
===== Г л а в н о е   м е н ю =====
- вывести все объекты      3 - найти максимальный
- переместить              4 - определить отношение включения
                          5 - выход

```

Заметьте, что был вызван деструктор для объекта `triaMax`, а не `triaB`. Продолжим тестирование. Введите цифру 5. Программа выведет

```

5 Конец работы.
Destructor for: triaD
Destructor for: triaC
Destructor for: triaB
Destructor for: triaA

```

Все. Программа работает как часы. Нам осталось решить последнюю задачу — определение отношения включения одного треугольника в другой.

Этап 4

Треугольник DEF входит в треугольник ABC (или ABC содержит в себе DEF), если каждая из вершин, D, E, F, находится внутри треугольника ABC. Известен простой способ определения, попадает ли некоторая точка Y во внутреннюю область треугольника ABC: Если сумма площадей треугольников AYW, BYC и AYC равна площади треугольника ABC, то очевидно, что ABC содержит в себе Y. Чтобы реализовать эту идею решения, внесите следующие дополнения в тексты модулей.

1. **Triangle.h:** добавьте в класс `Triangle` объявления методов:

```

double Area(Point, Point, Point) const; // Площадь треугольника, образованного
// заданными точками
bool Contains(Point) const; // Определяет, находится ли точка внутри треугольника а также
// объявление дружественной функции:
friend bool TriaInTria(Triangle, Triangle); // Определяет,
// входит ли один треугольник во второй

```

2. **Triangle.cpp:** добавьте реализацию методов:

```

double Triangle::Area(Point p1, Point p2, Point p3) const {
    double a1 = p1.DistanceTo(p2);
    double b1 = p2.DistanceTo(p3);
    double c1 = p1.DistanceTo(p3);
    double p = (a1 + b1 + c1) / 2;
    return sqrt(p * (p - a1) * (p - b1) * (p - c1));
}

```

```

bool Triangle::Contains(Point pt) const {
    const double calc_error = 0.001;
    double area1 = Area(pt, v1, v2);
    double area2 = Area(pt, v2, v3);
    double area3 = Area(pt, v3, v1);
    if (fabs(area1 + area2 + area3 - Area()) < calc_error) return true;
    else return false;
}

```

а также добавьте в конец файла реализацию внешней дружественной функции:

```

bool TriaInTria(Triangle trial, Triangle tria2) {
    Point v1 = trial.Get_v1();
    Point v2 = trial.Get_v2();
    Point v3 = trial.Get_v3();
    return (tria2.Contains(v1) && tria2.Contains(v2) && tria2.Contains(v3));
}

```

3. **Main.cpp:** замените заглушку функции IsIncluded следующим:

```

// ----- Определение отношения включения
void IsIncluded(Triangle* p_tria[], int k) {
    cout << "===== Отношение включения =====" << endl;
    cout << "Введите номер 1-го треугольника (от 1 до " << k << "): ";
    int i1 = GetNumber(1, k) - 1;
    cout << "Введите номер 2-го треугольника (от 1 до " << k << "): ";
    int i2 = GetNumber(1, k) - 1;
    if (TriaInTria(*p_tria[i1], *p_tria[i2]))
        cout << p_tria[i1]->GetName() << " - входит в - "
            << p_tria[i2]->GetName() << endl;
    else
        cout << p_tria[i1]->GetName() << " - не входит в - "
            << p_tria[i2]->GetName() << endl;
    ExitBack();
}

```

Модификация проекта завершена. Заметим, что погрешность сравнения площадей в методе Contains задается константой calc_error.

Откомпилируйте и запустите программу. Выберите пункт меню 4. Следуя указаниям программы, введите номера сравниваемых треугольников, например, 1 и 2. Вы должны получить такой результат:

```

4
===== Отношение включения =====
Введите номер 1-го треугольника (от 1 до 4): 1
Введите номер 2-го треугольника (от 1 до 4): 2
Copy constructor for: triaB
Copy constructor for: triaA
Destructor for: triaA(копия)
Destructor for: triaB(копия)
Треугольник 1 - входит в - Треугольник 2
Нажмите Enter.

```

Обратите внимание на отладочную печать: конструкторы копирования вызываются при передаче аргументов в функцию TriaInTria, а перед возвратом из этой функции вызываются соответствующие деструкторы.

Проведите следующий эксперимент: удалите с помощью скобок комментария конструктор копирования в файлах Triangle.h и Triangle.cpp, откомпилируйте проект и повторите тестирование четвертого пункта меню. Убедившись в неадекватном поведении программы, верните проект в нормальное состояние. Протестируйте остальные пункты меню. По завершении тестирования уберите с помощью символов комментария // всю отладочную печать. Повторите тестирование без отладочной печати. Решение задачи 10.2 завершено.

Итоги

1. Использование классов лежит в основе объектно-ориентированной декомпозиции программных систем, которая является более эффективным средством борьбы со сложностью систем, чем функциональная декомпозиция.
2. В результате декомпозиции система разделяется на компоненты (модули, функции, классы). Чтобы обеспечить высокое качество проекта, его способность к модификациям, удобство сопровождения, необходимо учитывать сцепление внутри компонента (оно должно быть сильным) и связанность

между компонентами (она должна быть слабой), а также правильно распределять обязанности между компонентом-клиентом и компонентом-сервером.

3. Класс — это определяемый пользователем тип, лежащий в основе ООП. Класс содержит ряд полей (переменных), а также методов (функций), имеющих доступ к этим полям.
4. Доступ к отдельным частям класса регулируется с помощью ключевых слов: `public` (открытая часть), `private` (закрытая часть) и `protected` (защищенная часть). Последний вид доступа имеет значение только при наследовании классов. Методы, расположенные в открытой части, формируют *интерфейс* класса и могут вызываться через соответствующий объект.
5. Если в классе имеются поля типа указателей и осуществляется динамическое выделение памяти, то необходимо позаботиться о создании конструктора копирования и перегрузке операции присваивания.
6. Для создания полного, минимального и интуитивно понятного интерфейса класса широко применяется перегрузка методов и операций.

3. Ход работы

3.1. Установка и настройка

3.1.1. _____

Для установки Python необходимо выполнить ряд действий в зависимости от ОС:

3.1.2. IDE

В рамках курса возможности разных сред разработки не отличаются, поэтому выбор остается за Вами.

3.2. Выполнение заданий включает несколько этапов

1. *Решение.*

Задания предусматривают написание программы на C++ (программирование) на базе заготовок в соответствии с заданием, инструкциями и подсказками.

Для решения задания по программированию необходимо:

- открыть заготовку и выполнить решение;
- запустить программу и исправить ошибки (до их отсутствия);
- проверить соответствие вывода на экран примеру.

2. *Проверка.*

Задания должны проходить проверку. В процессе проверки программа автоматически проверяется на правильность решения, соответствие стандарту оформления и др.

Для проверки задачи:

- настройте среду разработки;
- запустите команду для проверки и исправьте ошибки (до их отсутствия).

3. *Защита.*

Задания, которые были выполнены и проверены защищаются очно, включая демонстрацию и ответы на дополнительные вопросы.

3.3. Оборудование и материалы

Для выполнения данной лабораторной работы необходим компьютер с установленной операционной системой.

3.4. Указания по технике безопасности.

К выполнению лабораторных работ допускаются студенты, ознакомившиеся с правилами работы в лаборатории, прошедшие инструктаж безопасности.

4. Задания

Вариант 1 - 20

1. При выполнении задания необходимо построить UML-диаграмму классов программного модуля.
2. Тестирование и отладка:
 - 2.1. Внимательно прочитать условие задачи.
 - 2.2. Убедиться, что программа корректно работает на примерах.
 - 2.3. Составить план тестирования (проанализировать классы входных данных).
 - 2.4. Тестировать программу.
 - 2.5. Выполнить декомпозицию программы на отдельные блоки и покрыть каждый из них юнит-тестами

Примечание:

Условие задачи сформулировать внутри файла readme.md. С соответствующим форматированием. Файл поместить в папку проекта.

Диаграммы прикрепить в качестве отдельного файла в папку проекта.

Отладку решения задачи с помощью юнит-тестов, проводить в файле проекта (после отладки юнит-тесты закомментировать). При необходимости применять ввод входных данных из текстового файла .

Вариант 1

Описать класс, реализующий стек. Написать программу, использующую этот класс для моделирования Т-образного сортировочного узла на железной дороге. Программа должна разделять на

два направления состав, состоящий из вагонов двух типов (на каждое направление формируется состав из вагонов одного типа). Предусмотреть возможность формирования состава из файла и с клавиатуры.

Вариант 2

Описать класс, реализующий бинарное дерево, обладающее возможностью добавления новых элементов, удаления существующих, поиска элемента по ключу, а также последовательного доступа ко всем элементам.

Написать программу, использующую этот класс для представления англо-русского словаря. Программа должна содержать меню, позволяющее выполнить проверку всех методов класса. Предусмотреть возможность создания словаря из файла и с клавиатуры.

Вариант 3

Построить систему классов для описания плоских геометрических фигур: круг, квадрат, прямоугольник. Предусмотреть методы для создания объектов, перемещения на плоскости, изменения размеров и вращения на заданный угол.

Написать программу, демонстрирующую работу с этими классами. Программа должна содержать меню, позволяющее осуществить проверку всех методов классов.

Вариант 4

Построить описание класса, содержащего информацию о почтовом адресе организации. Предусмотреть возможность раздельного изменения составных частей адреса, создания и уничтожения объектов этого класса.

Написать программу, демонстрирующую работу с этим классом. Программа должна содержать меню, позволяющее осуществить проверку всех методов класса.

Вариант 5

Составить описание класса для представления комплексных чисел. Обеспечить выполнение операций сложения, вычитания и умножения комплексных чисел.

Написать программу, демонстрирующую работу с этим классом. Программа должна содержать меню, позволяющее осуществить проверку всех методов класса.

Вариант 6

Составить описание класса для объектов-векторов, задаваемых координатами концов в трехмерном пространстве. Обеспечить операции сложения и вычитания векторов с получением нового вектора (суммы или разности), вычисления скалярного произведения двух векторов, длины вектора, косинуса угла между векторами.

Написать программу, демонстрирующую работу с этим классом. Программа должна содержать меню, позволяющее осуществить проверку всех методов класса.

Вариант 7

Составить описание класса прямоугольников со сторонами, параллельными осям координат. Предусмотреть возможность перемещения прямоугольников на плоскости, изменение размеров, построение наименьшего прямоугольника, содержащего два заданных прямоугольника, и прямоугольника, являющегося общей частью (пересечением) двух прямоугольников.

Написать программу, демонстрирующую работу с этим классом. Программа должна содержать меню, позволяющее осуществить проверку всех методов класса.

Вариант 8

Составить описание класса для определения одномерных массивов целых чисел (векторов). Предусмотреть возможность обращения к отдельному элементу массива с контролем выхода за пределы массива, возможность задания произвольных границ индексов при создании объекта и выполнения операций поэлементного сложения и вычитания массивов с одинаковыми границами индексов, умножения и деления всех элементов массива на скаляр, вывода на экран элемента массива по заданному индексу и всего массива.

Написать программу, демонстрирующую работу с этим классом. Программа должна содержать меню, позволяющее осуществить проверку всех методов класса.

Вариант 9

Составить описание класса для определения одномерных массивов строк фиксированной длины. Предусмотреть возможность обращения к отдельным строкам массива по индексам, контроль выхода за пределы массива, выполнения операций поэлементного сцепления двух массивов с образованием нового массива, слияния двух массивов с исключением повторяющихся элементов, вывода на экран элемента массива по заданному индексу и всего массива.

Написать программу, демонстрирующую работу с этим классом. Программа должна содержать меню, позволяющее осуществить проверку всех методов класса.

Вариант 10

Составить описание класса многочленов от одной переменной, задаваемых степенью многочлена и массивом коэффициентов. Предусмотреть методы для вычисления значения многочлена для заданного аргумента, операции сложения, вычитания и умножения многочленов с получением нового объекта-многочлена, вывод на экран описания многочлена.

Написать программу, демонстрирующую работу с этим классом. Программа должна содержать меню, позволяющее осуществить проверку всех методов класса.

Вариант 11

Составить описание класса одномерных массивов строк, каждая строка задается длиной и указателем на выделенную для нее память. Предусмотреть возможность обращения к отдельным строкам массива по индексам, контроль выхода за пределы массивов, выполнения операций поэлементного сцепления двух массивов с образованием нового массива, слияния двух массивов с исключением повторяющихся элементов, вывода на экран элемента массива и всего массива.

Написать программу, демонстрирующую работу с этим классом. Программа должна содержать меню, позволяющее осуществить проверку всех методов класса.

Вариант 12

Составить описание класса, обеспечивающего представление матрицы произвольного размера с возможностью изменения числа строк и столбцов, вывод на экран подматрицы любого размера и всей матрицы.

Написать программу, демонстрирующую работу с этим классом. Программа должна содержать меню, позволяющее осуществить проверку всех методов класса.

Вариант 13

Написать класс для эффективной работы со строками, позволяющий форматировать и сравнивать строки, хранить в строках числовые значения и извлекать их. Для этого необходимо реализовать:

- перегруженные операции присваивания и конкатенации;
- операции сравнения и приведения типов; % преобразование в число любого типа; %о форматный вывод строки.

Написать программу, демонстрирующую работу с этим классом. Программа должна содержать меню, позволяющее осуществить проверку всех методов класса.

Вариант 14

Описать класс «домашняя библиотека». Предусмотреть возможность работы с произвольным числом книг, поиска книги по какому-либо признаку (например, по автору или по году издания), добавления книг в библиотеку, удаления книг из нее, сортировки книг по разным полям.

Написать программу, демонстрирующую работу с этим классом. Программа должна содержать меню, позволяющее осуществить проверку всех методов класса.

Вариант 15

Описать класс «записная книжка». Предусмотреть возможность работы с произвольным числом записей, поиска записи по какому-либо признаку (например, по фамилии, дате рождения или номеру телефона), добавления и удаления записей, сортировки по разным полям.

Написать программу, демонстрирующую работу с этим классом. Программа должна содержать меню, позволяющее осуществить проверку всех методов класса.

Вариант 16

Описать класс «студенческая группа». Предусмотреть возможность работы с переменным числом студентов, поиска студента по какому-либо признаку (например, по фамилии, дате рождения или номеру телефона), добавления и удаления записей, сортировки по разным полям.

Написать программу, демонстрирующую работу с этим классом. Программа должна содержать меню, позволяющее осуществить проверку всех методов класса.

Вариант 17

Описать класс, реализующий тип данных «вещественная матрица» и работу с ними. Класс должен реализовывать следующие операции над матрицами:

- сложение, вычитание, умножение, деление (+, -, *, /) (умножение и деление как на другую матрицу, так и на число);
- комбинированные операции присваивания (+=, -=, *=, /=);
- операции сравнения на равенство (равенство);

- операции вычисления обратной и транспонированной матрицы, операцию возведения в степень;
- методы вычисления детерминанта и нормы;
- методы, реализующие проверку типа матрицы (квадратная, диагональная, нулевая, единичная, симметрическая, верхняя треугольная, нижняя треугольная); %о операции ввода-вывода в стандартные потоки.

Написать программу, демонстрирующую работу с этим классом. Программа должна содержать меню, позволяющее осуществить проверку всех методов класса.

Вариант 18

Описать класс «множество», позволяющий выполнять основные операции — добавление и удаление элемента, пересечение, объединение и разность множеств.

Написать программу, демонстрирующую работу с этим классом. Программа должна содержать меню, позволяющее осуществить проверку всех методов класса.

Вариант 19

Описать класс, реализующий стек. Написать программу, использующую этот класс для отыскания прохода по лабиринту.

Лабиринт представляется в виде матрицы, состоящей из квадратов. Каждый квадрат либо открыт, либо закрыт. Вход в закрытый квадрат запрещен. Если квадрат открыт, то вход в него возможен со стороны, но не с угла. Каждый квадрат определяется его координатами в матрице. После отыскания прохода программа печатает найденный путь в виде координат квадратов.

Вариант 20

Описать класс «предметный указатель». Каждый компонент указателя содержит слово и номера страниц, на которых это слово встречается. Количество номеров страниц, относящихся к одному слову, от одного до десяти. Предусмотреть возможность формирования указателя с клавиатуры и из файла, вывода указателя, вывода номеров страниц для заданного слова, удаления элемента из указателя.

Написать программу, демонстрирующую работу с этим классом. Программа должна содержать меню, позволяющее осуществить проверку всех методов класса.

5. Содержание отчета

Отчет по лабораторной работе должен быть выполнен в текстовом редакторе и оформлен согласно требованиям. На проверку необходимо сдать следующие файлы:

1. Исходный файл, содержащий готовый отчет и набранный с помощью текстового редактора (например *.doc, *.docx, *.rtf и др.), а также готовый отчет в открытом формате электронных документов *.pdf.
2. Zip-архив, содержащий файлы программного кода выполненных заданий.

Требования по форматированию: Шрифт TimesNewRoman, интервал – полуторный, поля левое – 3 см., правое – 1,5 см., верхнее и нижнее – 2 см. Абзацный отступ – 1,25. Текст должен быть выровнен по ширине.

Отчет должен содержать титульный лист с темой лабораторной работы, цель работы и описанный процесс выполнения вашей работы. В конце отчета приводятся выводы о проделанной работе.

В отчет необходимо вставлять скриншоты выполненной работы и добавлять описание к ним. Каждый рисунок должен располагаться по центру страницы, иметь подпись (Рисунок 1 – Создание подсистемы) и ссылку на него в тексте.

6. Контрольные вопросы:

6.1. Классы и контейнеры:

1. Что определяет класс? Чем обличается класс от объекта?
2. Можно ли объявлять массив объектов? А массив классов?
3. Разрешается ли объявлять указатель на объект? А указатель на класс?
4. Допускается ли передавать объекты в качестве параметров, и какими способами? А возвращать как результат?
5. Как называется использование объекта одного класса в качестве поля другого класса?
6. Является ли структура классом? Чем класс отличается от структуры?
7. Объясните принцип инкапсуляции.
8. Что такое композиция?
9. Для чего используются ключевые слова public и private?
10. Можно ли использовать ключевые слова public и private в структуре?

11. Существуют ли ограничения на использование `public` и `private` в классе? А в структуре?
12. Обязательно ли делать поля класса приватными?
13. Что такое метод? Как вызывается метод?
14. Может ли метод быть приватный?
15. Как определить метод непосредственно внутри класса? А вне класса? Чем эти определения отличаются?
16. Можно в методах присваивать параметрам значения по умолчанию?
17. Что обозначается ключевым словом `this`?
18. Зачем нужны константные методы? Чем отличается определение константного метода от обычного?
19. Может ли константный метод вызываться для объектов-переменных? А обычный метод — для объектов-констант?
20. Объясните принцип полиморфизма.
21. Сколько места в памяти занимает объект класса? Как это узнать?
22. Каков размер «пустого» объекта?
23. Влияют ли методы на размер объекта?
24. Одинаков ли размер класса и аналогичной структуры?
25. Какие операции нельзя перегружать? Как вы думаете, почему?
26. Можно ли перегружать операции для встроенных типов данных?
27. Можно ли при перегрузке изменить приоритет операции?
28. Можно ли определить новую операцию?
29. Перечислите особенности перегрузки операций как методов класса. Чем отличается перегрузка внешним образом от перегрузки как метода класса?
30. Какой результат должны возвращать операции с присваиванием?
31. Как различаются перегруженная префиксная и постфиксная операции инкремента и декремента?
32. Что означает выражение `*this`? В каких случаях оно используется?
33. Какие операции не рекомендуется перегружать как методы класса? Почему?
34. Какие операции разрешается перегружать только как методы класса?
35. Дайте определение дружественной функции. Как объявляется дружественная функция? А как определяется?
36. Дайте определение конструктора. Каково назначение конструктора? Перечислите отличия конструктора от метода.
37. Сколько конструкторов может быть в классе? Допускается ли перегрузка конструкторов? Какие виды конструкторов создаются по умолчанию?
38. Может ли конструктор быть приватным? Какие последствия влечет за собой объявление конструктора приватным?
39. Приведите несколько случаев, когда конструктор вызывается неявно.
40. Как проинициализировать динамическую переменную?
41. Как объявить константу в классе? Можно ли объявить дробную константу?
42. Каким образом разрешается инициализировать константные поля в классе?
43. В каком порядке инициализируются поля в классе? Совпадает ли этот порядок с порядком перечисления инициализаторов в списке инициализации конструктора?
44. Какие конструкции C++ разрешается использовать в списке инициализации в качестве инициализирующих выражений?
45. Какой вид конструктора фактически является конструктором преобразования типов?
46. Для чего нужны функции преобразования? Как объявить такую функцию в классе?
47. Как запретить неявное преобразование типа, выполняемое конструктором инициализации?
48. Какие проблемы могут возникнуть при определении функций преобразования?
49. Для чего служит ключевое слово `explicit`?
50. Влияет ли наличие целочисленных констант-полей на размер класса?
51. Разрешается ли объявлять массив в качестве поля класса. Как присвоить элементам массива начальные значения?
52. Сколько операндов имеет операция индексирования `[]`? Какой вид результата должна возвращать эта операция?
53. Для чего нужны статические поля в классе? Как они определяются?

54. Как объявить в классе и проинициализировать статический константный массив?
55. Что такое выравнивание и от чего оно зависит? Влияет ли выравнивание на размер класса?
56. Дайте определение контейнера.
57. Какие виды встроженных контейнеров в C++ вы знаете?
58. Какие виды доступа к элементам контейнера вам известны?
59. Чем отличается прямой доступ от ассоциативного?
60. Прямой(`direct access`) - это вроеде через оператор[], а что такое ассоциативный??
61. Перечислите операции, которые обычно реализуются для последовательного доступа к элементам контейнера.
62. Дайте определение итератора.
63. Можно ли реализовать последовательный доступ без итератора? В чем преимущества реализации последовательного доступа с помощью итератора?
64. Что играет роль итератора для массивов C++?
65. Что такое деструктор? Может ли деструктор иметь параметры?
66. Почему для классов-контейнеров деструктор надо писать явным образом?
67. Допускается ли перегрузка деструкторов?
68. Что такое «глубокое копирование» и когда в нем возникает необходимость?
69. Какое копирование осуществляет стандартный конструктор копирования?
70. Чем отличается копирование от присваивания?
71. Объясните, почему в операции присваивания требуется проверка присваивания самому себе?
72. Можно ли в качестве операции индексирования использовать операцию вызова функции ()? В чем ее преимущества перед операцией []?
73. Почему необходимо писать два определения операции индексирования? Чем они отличаются?
74. Дайте определение вложенного класса.
75. Можно ли класс-итератор реализовать как внешний класс? А как вложенный? В чем отличия этих методов реализации?
76. Может ли объемлющий класс иметь неограниченный доступ к элементам вложенного класса? А вложенный класс — к элементам объемлющего?
77. Ограничена ли глубина вложенности классов?
78. Можно ли определить вложенный класс внешним образом? Зачем это может понадобиться?
79. Каким образом вложенный класс может использовать методы объемлющего класса? А объемлющий — методы вложенного?
80. Что такое «запредельный» элемент, какую роль он играет в контейнерах?
81. Объясните, по каким причинам трудно написать универсальный контейнер, элементы которого могут иметь произвольный тип.

7. Список рекомендуемых источников по данной теме:

7.1. Официальная документация.

1. News, Status & Discussion about Standard C++. URL: <https://isocpp.org/>
2. Get a Compiler. URL: <https://isocpp.org/get-started>
3. Standardization. URL: <https://isocpp.org/std>

7.2. C/C++.

1. Страуструп, Б. Тур по C++. Краткий курс. Третье издание / Б. Страуструп. – Издательство: Самиздат, 2022. – 320 с.
2. Страуструп, Б. Язык программирования C++. Специальное издание / Б. Страуструп. – Издательство: Бином, 2008. – 1104 с.
3. Прата, С. Язык программирования C++. Лекции и упражнения / С. Прата. – М.: Издательство: ДиаСофтЮП, 2005. – 1104 с.
4. Архангельский, А.Я. Программирование в C++ Builder / А.Я. Архангельский. – М.: Бином, 2008.
5. Павловская, Т.А. C/ C++. Программирование на языке высокого уровня / Т.А. Павловская. – СПб.: Питер, 2011. – 461 с.
6. Бобровский, С.И. Технологии C++ Builder. Разработка приложений для бизнеса. Учебный курс /С.И. Бобровский. – СПб.: Питер, 2007. – 560 с..

7.3. Прочее.

1. Объектно-ориентированное программирование. URL: https://ru.wikipedia.org/wiki/Объектно-ориентированное_программирование.
2. Коротко об истории объектно-ориентированного программирования. URL: <http://habrahabr.ru/post/107940/>.
3. ООП с примерами (часть 1). URL: <https://megamozg.ru/post/6908/>.
4. ООП с примерами (часть 2). URL: <https://megamozg.ru/post/6910/>.
5. Really Brief Introduction to Object Oriented Design. URL: <http://www.finner.org/tips/General/SoftwareEngineering/ObjectOrientedDesign.shtml>.
6. Java Programming Tutorial. Object-oriented Programming (OOP) Basics. URL: https://www3.ntu.edu.sg/home/ehchua/programming/java/J3a_OOPBasics.html.
7. Define static methods in the following scenarios only. URL: <http://stackoverflow.com/a/5313383/396619>.
8. Practical UML: A Hands-On Introduction for Developers. URL: <http://edn.embarcadero.com/article/31863>.
9. Building Skills in Object-Oriented Design. URL: <http://www.itmaybeahack.com/homepage/books/oodesign.html>.

Лабораторная работа №2.

Наследование

На этом семинаре мы не ограничимся наследованием, а рассмотрим более широкий круг вопросов:

- наследование классов и полиморфизм;
- отношения между классами и диаграммы классов на языке UML;
- введение в использование паттернов проектирования; технология проектирования программ с учетом будущих изменений.

Наследование классов

Классы в объектно-ориентированных программах используются для моделирования концепций реального и программного мира. Концепции или сущности предметной области находятся в различных взаимоотношениях. Одно из таких взаимоотношений — *отношение наследования* (именуемое также отношением *родитель–потомок* или отношением *обобщение–специализация*).

Например, если в условии задачи 1.2 рассматривать не только треугольники, но и четырехугольники, класс Tetragon имел бы общие черты с классом Triangle, поскольку и треугольники, и четырехугольники являются частным (специальным, конкретным) случаем более общего понятия «многоугольник». Поэтому было бы логичным создать класс Polygon, содержащий элементы, общие для классов Triangle и Tetragon, а последние два класса объявить «наследниками» базового (родительского) класса Polygon. Язык C++ позволяет легко это сделать:

```
class Polygon {  
    // . . .  
};  
class Triangle : public Polygon {  
public: Show();  
};  
class Tetragon : public Polygon {  
public: Show();  
};
```

В этом примере производные классы Triangle и Tetragon наследуют все элементы базового класса Polygon, но каждый из них имеет свой собственный метод Show. Иногда отношение наследования называют отношением «*is a*», что можно перевести как «представляет собой». В данном примере Triangle *is a* Polygon, в такой же мере и

Tetragon *is a* Polygon.

Общий синтаксис создания производного класса при *простом наследовании*:

```
class имя : ключ_доступа имя_базового_класса {  
    // тело класса
```

```
};
```

В случае *множественного наследования* после двоеточия перечисляются через запятую все базовые классы со своими модификаторами доступа.

Производный класс, в свою очередь, сам может служить базовым классом. Такой набор связанных классов называется *иерархией классов*. Иерархия чаще всего является деревом, но может иметь и более общую структуру графа.

1. В примерах предыдущего семинара доступ к элементам класса регулировался с помощью двух модификаторов: `private` — закрытая часть класса, `public` — открытая часть класса. Для базовых классов возможно использование еще одного модификатора — `protected`, который определяет *защищенную* часть класса. Ее элементы являются доступными для любого производного класса, но в то же время они недоступны вне классов данной иерархии.

Кроме этого, доступность в производном классе регулируется *ключом доступа к базовому классу*, указываемому в объявлении производного класса. Этот ключ определяет *вид наследования*: открытое (`public`), защищенное (`protected`) или закрытое (`private`).

Открытое наследование сохраняет статус доступа всех элементов базового класса, *защищенное* — понижает статус доступа `public` элементов базового класса до `protected`, и наконец, *закрытое* — понижает статусы доступа `public` и `protected` элементов базового класса до `private`. Заметим, что в C++ отношение между классами «*is a*» имеет место только при открытом наследовании.

Замещение функций базового класса

Иногда в производном классе требуется несколько иная реализация метода, унаследованного из базового класса. Язык позволяет это сделать. *Замещение (переопределение)* метода производится путем объявления в производном классе метода с таким же именем. Если понадобится все-таки вызвать из потомка метод предка, используется операция доступа к области видимости ::, например:

```
#include <iostream>
using namespace std;
class Base {
public:
    void Display() { cout << "Hello, world! "; }
};
class Derived : public Base {
public:
    void Display() {
        Base::Display(); // Вызов метода базового
        cout << "How are you? ";
    }
};
```

```

    }
};
class SubDerived : public Derived {
public:
    void Display() {
        Derived::Display();           // Вызов метода базового класса
        cout << "Bye!" << endl;
    }
};
int main() {
    SubDerived sd;
    sd.Display();                     // Результат: Hello, world!  How are you?  Bye!
}

```

Конструкторы и деструкторы в производном классе

Конструкторы и деструкторы из базового класса не наследуются, поэтому при создании производного класса встает вопрос, нужны ли эти методы и как они должны быть реализованы. Решая эти вопросы, учитывайте следующее.

- Если в базовом классе вообще нет конструктора или есть конструктор по умолчанию, производному классу конструктор нужен, *только* если требуется инициализировать поля, введенные в этом классе.

- Если вы не определили ни одного конструктора, компилятор самостоятельно создаст конструктор по умолчанию, из которого будет вызван конструктор по умолчанию базового класса.

- Если в базовом классе есть конструктор с аргументами, то производный класс должен содержать конструктор со списком аргументов, включающим значения для передачи конструктору базового класса; конструктор базового класса вызывается в списке инициализации.

Необходимость **в деструкторе** для производного класса определяется тем, нужно ли освобождать какие-либо ресурсы, выделенные в конструкторе. Если такой необходимости нет, можно доверить компилятору создать деструктор по умолчанию. В нем обеспечивается вызов деструктора базового класса.

На этапе выполнения программы при создании объекта производного класса сначала вызываются конструкторы базовых классов, начиная с самого верхнего уровня, затем конструкторы объектов-элементов класса и в последнюю очередь — конструктор класса. При уничтожении объекта (например, когда покидается область его видимости) деструкторы вызываются в порядке, обратном вызову конструкторов.

Устранение неоднозначности при множественном наследовании

Предположим, что от базового класса А, имеющего некоторый элемент *x*, наследуются два класса, В и С, а класс D является производным от этих двух классов (множественное наследование). Если попытаться обратиться к элементу

x из методов класса D, компилятор воспримет выражение D::x как неоднозначное¹ и прекратит работу. Для решения этой проблемы в C++ предусмотрен механизм, благодаря которому в класс D будет включена только одна копия класса A. Достигается это добавлением спецификатора virtual перед модификаторами доступа к A в объявлениях классов B и C:

```
class A {
public:    A(int _x = 0) { x = _x; }
protected: int x;
};
class B : virtual public A {           // Виртуальное наследование
public:    void AddB(int y) { x += y; }
};
class C : virtual public A {           // Виртуальное наследование
public:    void AddC(int y) { x += y; }
};
class D : public B, public C {         // Здесь будет только одна копия
полей класса A
public:    void ShowX() { cout << "x = " << x << endl; }
};
int main() {
    D d;
    d.ShowX();                        // Вывод: x = 0
    d.AddB(10);    d.ShowX();         // Вывод: x = 10
    d.AddC(5);    d.ShowX();         // Вывод: x = 15
}
```

Доступ к объектам иерархии

Эффективнее всего работать с объектами одной иерархии через указатель на базовый класс. При открытом (public) наследовании такому указателю можно присваивать адрес объекта как базового класса, так и любого производного класса. Такая возможность представляется вполне логичной: в качестве представителя более общего класса используется объект специализированного класса.

Однако если не принять специальных мер, то при работе с таким указателем независимо от того, какой адрес ему присвоен, оказываются доступными *только элементы базового класса*. Например, рассмотрим следующий код:

```
class Base {
public: void Modify();
};
class Derived : public Base {
public: void Modify();
};
int main() {
```

¹ Поскольку неясно, какой элемент класса A имеется в виду: унаследованный через класс B или унаследованный через класс C.

```

int mode;
Base* pB;
cin >> mode;
if (mode == 1) pB = new Base;
else          pB = new Derived;
pB->Modify(); // на что указывает pB?
}

```

Так как компилятор не может предсказать, какой выбор будет сделан на этапе выполнения, он выбирает метод по типу указателя `pB`, то есть `Base::Modify`. Такая стратегия называется *ранним (статическим) связыванием*. Поэтому, если в производном классе замещен некоторый метод базового класса (например, `Derived::Modify`), он оказывается недоступным (листинг 2.1).

Листинг 2.1. Раннее связывание

```

#include <iostream>
using namespace std;
class Base {
public:
    Base(int _x = 10) { x = _x; }
    void ShowX() { cout << "x = " << x << "; "; }
    void ModifyX() { x *= 2; }
protected:
    int x;
};
class Derived : public Base {
public:
    void ModifyX() { x /= 2; }
};
int main() {
    Base b;      Derived d;
    b.ShowX();   d.ShowX(); // Вывод: x = 10; x = 10;
    Base* pB;
    pB = &b;     pB->ModifyX(); pB->ShowX(); // Вывод: x = 20;
    pB = &d;     pB->ModifyX(); pB->ShowX(); // Вывод: x = 20;
}

```

Как видите, второй вызов метода `ModifyX` происходит также из базового класса.

Виртуальные методы

Проблема доступа к методам, переопределенным в производных классах, через указатель на базовый класс решается в C++ посредством использования *виртуальных методов*. Чтобы сделать некоторый метод виртуальным, надо в базовом классе предварить его заголовок спецификатором `virtual`. После этого он будет восприниматься как виртуальный во всех производных классах иерархии.

По отношению к виртуальным методам компилятор применяет стратегию *позднего (динамического) связывания*. Это означает, что на этапе компиляции он не определяет, какой из методов должен быть вызван, а передает

ответственность программе, принимающей решение на этапе выполнения, когда уже точно известно, каков тип объекта, адресуемого через указатель. Все сказанное относится также к вызову методов *по ссылке* на базовый класс.

Для реализации динамического связывания компилятор создает *таблицу виртуальных методов* (*vtbl*), а к каждому объекту добавляет скрытое поле ссылки (*vptr*) на таблицу *vtbl*. Это несколько снижает эффективность программы, поэтому рекомендуется делать виртуальными только те методы, которые переопределяются в производных классах.

Чтобы увидеть динамическое связывание в действии, добавьте в листинг 2.1 спецификатор `virtual` перед заголовком метода `ModifyX` в базовом классе:

```
virtual void ModifyX() { x *= 2; }
```

Теперь второй вызов метода `ModifyX` происходит из производного класса, и программа должна вывести

```
x = 10; x = 10; x = 20; x = 5;
```

Виртуальный механизм работает только при использовании указателей и ссылок на объекты. Объект, определенный через указатель или ссылку и содержащий виртуальные методы, называется *полиморфным*.

Если базовый класс содержит хотя бы один виртуальный метод, рекомендуется всегда снабжать этот класс *виртуальным деструктором*, даже если он ничего не делает. Наличие виртуального деструктора предотвратит некорректное удаление объектов производного класса, адресуемых через указатель на базовый класс, так как иначе деструктор производного класса вызван не будет².

Абстрактные классы. Чисто виртуальные методы

Иногда, разрабатывая иерархию классов, можно получить базовый класс, для которого создание объекта как бы теряет смысл. Например, иерархия классов геометрических фигур может произрастать из базового класса `Shape`, а в качестве производных будут выступать классы `Polygon`, `Ellipse` и т.д. Одним из методов базового класса будет, видимо, функция `Show` — показать объект. Но как можно показывать объект, не имеющий конкретной формы?

Классы, для которых нет смысла создавать объекты, объявляют как *абстрактные*. Абстрактный класс — это класс, содержащий хотя бы один *чисто виртуальный метод* (метод, объявленный в классе, но не имеющий конкретной реализации). Синтаксис объявления такого метода дополняется конструкцией `= 0`, например:

```
virtual void Show() = 0;
```

Компилятор не допустит создания объекта для абстрактного класса, если вы по забывчивости попытаетесь это сделать. Если в производном классе хотя бы одна чисто виртуальная функция базового класса останется без конкретной

² В небольшой учебной программе это может пройти без последствий. В нетривиальных программах возможно появление труднодиагностируемых ошибок.

реализации, то производный класс также будет абстрактным классом, для которого запрещено создавать объекты.

Отношения между классами. Диаграммы классов на языке UML

Полиморфизм и наследование делают язык C++ чрезвычайно мощным инструментом для реализации объектно-ориентированной технологии проектирования.

Для эффективного восприятия новых технологических идей полезно знать хотя бы основы ставшего уже стандартом *унифицированного языка моделирования UML* (Unified Modeling Language).

Язык UML является визуальным средством представления *моделей программ*, то есть их графического представления в виде различных диаграмм, отражающих связи между объектами в программном коде.

Одной из основных диаграмм языка UML является *диаграмма классов*. Она описывает классы и отношения, существующие между ними. Такая диаграмма чрезвычайно удобна для сопоставления различных вариантов проектных решений. Класс изображается на диаграмме классов UML в виде прямоугольника, состоящего из трех частей. Имя класса указывается в верхней части. В средней части приводится список *атрибутов* с указанием типов (атрибут класса в UML соответствует термину *поле класса* в C++).

В нижней части записывается список *операций* (методов класса), возможно, с указанием списка типов аргументов и типа возвращаемого значения. Имя *абстрактного класса*, так же как и имена *абстрактных операций* (чисто виртуальных методов), выделяются курсивом. Перед именем поля или метода может указываться спецификатор доступа с помощью одного из трех символов: + для public, - для private, # для protected. Для статических элементов класса после спецификатора доступа записывается символ \$.

Во второй и третьей частях могут указываться не все элементы класса, а только те, которые представляют интерес на данном уровне абстракции. Если обе эти части пусты, они могут быть опущены. Например, варианты изображения класса Triangle на различных этапах анализа решения задачи 1.2 показаны на рис. 2.1.

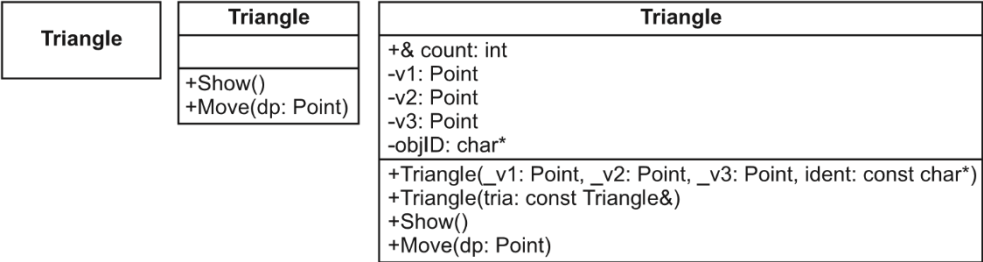


Рис.2.1. Варианты изображения класса Triangle на диаграмме классов UML

Большинство объектно-ориентированных языков поддерживают следующие виды отношений между классами: ассоциация, наследование, агрегация и зависимость. Рассмотрим, что они означают и как изображаются на диаграмме классов.

Ассоциация

Если два класса концептуально взаимодействуют друг с другом, такое взаимодействие называется *ассоциацией*. Например, желая смоделировать торговую точку, мы вводим две абстракции: товары (класс `Product`) и продажи (класс `Sale`). Объект класса `Sale` — это некоторая сделка, в которой продано от 1 до n объектов класса `Product`. На рис. 2.2 ассоциация между этими двумя классами показана соединяющей их линией. Над линией рядом с обозначением класса может быть указана так называемая *кратность* (*multiplicity*), указывающая, сколько объектов данного класса может быть связано с одним объектом другого класса. Символ звездочки `*` обозначает «произвольное количество».

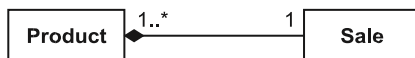


Рис. 2.2. Отношения ассоциации

Ассоциация представляет наиболее абстрактную семантическую связь между двумя классами, выявляемую на ранней стадии анализа. В дальнейшем она, как правило, конкретизируется, переходя в одно из рассматриваемых далее отношений.

Наследование

Отношение *обобщения* (наследования, «*is a*») между классами показывает, что *подкласс* (производный класс) разделяет атрибуты и операции, определенные в одном или нескольких *суперклассах* (базовых классах). На диаграмме классов отношение наследования показывают линией со стрелкой в виде незакрашенного треугольника, которая указывает на базовый класс. Допускается объединять несколько стрелок в одну (см. рис. 2.3), с тем чтобы разгрузить диаграмму.

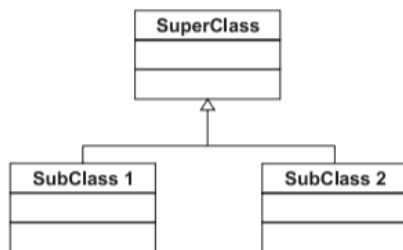


Рис. 2.3 Отношения наследования

Отношение *агрегации* показывает, что один класс содержит в качестве составной части объекты другого класса. Иными словами, это отношение *целое–часть*, или отношение «*has a*», между двумя классами. На диаграмме такая связь обозначается линией со стрелкой в виде незакрашенного ромба, которая указывает на целое. На рис. 2.4 показан пример так называемой *нестрогой* агрегации (или просто агрегации). Действительно, конкретный объект класса СпортЗал может содержать не все компоненты (спортивные снаряды), присутствующие на схеме.

Как распознать агрегацию в программном коде? Для реализации нестрогой агрегации *часть* включается в целое *по ссылке*: на языке C++ это обычно указатель на соответствующий класс. Если этот указатель равен нулю, то компонент отсутствует. В зависимости от решаемой задачи такой компонент может появляться и исчезать динамически в течение жизни объекта *целое*.



Рис. 2.4. Отношения агрегации

Строгая агрегация имеет специальное название — *композиция*. Она означает, что компонент не может исчезнуть, пока объект *целое* существует. Пример такого отношения мы уже встречали при решении задачи 1.2, в которой класс Triangle содержал в себе три объекта класса Point. На диаграмме отношение композиции обозначается линией со стрелкой в виде закрашенного ромба (рис. 2.5).

Проще всего композицию реализовать включением объектов-компонентов *по значению*. В то же время возможна реализация и включением по ссылке, но тогда времена жизни компонентов и объекта *целое* должны совпадать.

Зависимость

Отношение *зависимости* (использования) показывает, что один класс (Client) пользуется услугами другого класса (Supplier), например:

- метод класса Client использует значения некоторых полей класса Supplier;
- метод класса Client вызывает некоторый метод класса Supplier;
- метод класса Client имеет сигнатуру, в которой упоминается класс Supplier. На диаграмме такое отношение изображается пунктирной

линией со стрелкой, указывающей на класс `Supplier`. Например, отношения зависимости существует между классами `Triangle` и `Point` из задачи 1.2 (рис. 2.6).



Рис. 2.5. Отношение композиции

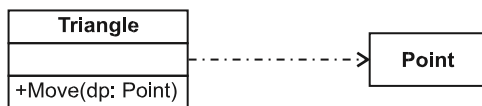


Рис. 2.6. Отношение зависимости (использования)

Паттерны проектирования

Одним из важнейших достижений в области ООП является методология паттернов проектирования, иногда называемых шаблонами проектирования³. Впервые она была представлена в книге [6].

Паттерн — это описание взаимодействия объектов и классов, адаптированных для решения общей задачи проектирования в конкретном контексте. Паттерны выявляются по мере накопления опыта разработок, когда программист использует одну и ту же схему организации и взаимодействия объектов в разных контекстах.

Поначалу паттерны можно было рассматривать как особенно изящный и хорошо продуманный способ решения определенного класса задач. Однако практика показала, что применение паттернов наряду с объектной технологией позволяет вывести процесс проектирования из области интуиции и представить систему на более высоком уровне абстракции — на уровне схем взаимодействия объектов. Любая абстракция исключает частные, второстепенные детали, и обычно при этом удается выделить в проблеме ее переменные и постоянные составляющие.

Основные трудности при разработке элегантной и удобной в сопровождении архитектуры вызывает идентификация так называемого «вектора изменений». Выявление важнейшего фактора перемен в проектируемой системе дает опорную точку для построения дальнейшей архитектуры.

Итак, паттерны предназначены прежде всего для *инкапсуляции изменений*. Если рассматривать их с этой точки зрения, то некоторые паттерны нам уже встречались. Например, наследование тоже может рассматриваться как паттерн, пусть даже реализованный на уровне компилятора. Оно выражает

³ Слово «шаблон» в отношении термина «pattern» не вполне удачно, поскольку вызывает путаницу с термином «template». Поэтому во многих книгах издательства «Питер» используется термин «паттерн».
— Примеч. ред.

различия в поведении (переменная составляющая) объектов, обладающих одинаковым интерфейсом (постоянная составляющая). Впрочем, обычно возможности, напрямую поддерживаемые языком программирования, не принято относить к паттернам.

Один из фундаментальных принципов, провозглашенный в книге [6], гласит: «Отдавайте предпочтение композиции объектов перед наследованием классов». И действительно, иногда композиция радикально упрощает архитектуру, в которой кажется уместным наследование, а полученная архитектура становится более гибкой.

Краткие сведения о паттернах в данном разделе приведены по книге [6]. Разумеется, для более глубокого изучения этой темы нужно обратиться к специальной литературе, например, [5], [6] и [34].

Все паттерны в соответствии с их назначением разделены на три группы: порождающие, структурные и паттерны поведения. *Порождающие паттерны* описывают способы создания объектов, *структурные* — схемы организации классов и объектов, а *паттерны поведения* определяют типичные схемы взаимодействия классов и объектов. В каждой группе можно, в свою очередь, выделить два уровня, определяющих, применяется паттерн к классам или объектам. В табл. 2.1 приведена классификация паттернов.

Описание каждого паттерна в [6] выполнено по одной и той же схеме, включающей его назначение, область применения, структурную схему (в форме диаграммы классов UML), описание входящих в него объектов и их взаимодействий, а также пример реализации. Такой уровень документирования делает возможным использование паттерна в различных конкретных случаях, возникающих при проектировании программных систем.

Таблица 2.1. Классификация паттернов проектирования

Назначение/ Уровень	Порождающие паттерны	Структурные паттерны	Паттерны поведения
Класс	Фабричный метод	Адаптер (класса)	Интерпретатор Шаблонный метод
Объект	Абстрактная фабрика Одиночка Прототип Строитель	Адаптер (объекта)	Итератор
		Декоратор	Команда
		Заместитель	Наблюдатель
		Компоновщик	Посетитель
		Мост	Посредник
		Приспособленец	Состояние
		Фасад	Стратегия
			Хранитель

В качестве примера мы приведем здесь описание только паттерна Стратегия.

Паттерн Стратегия (Strategy)

Этот паттерн инкапсулирует семейство алгоритмов, делая их взаимозаменяемыми. Применять его целесообразно в следующих случаях:

- имеются родственные классы, различающиеся только поведением. Стратегия позволяет гибко конфигурировать класс, задавая одно из возможных поведений;

- требуется иметь несколько разных вариантов алгоритма. Варианты алгоритмов могут быть реализованы в виде иерархии классов, что позволяет вычлнить общую для всех классов функциональность. Инкапсулированные алгоритмы можно затем применять в разных контекстах;

- в классе определено много вариантов поведения, что представлено разветвленными условными операторами. В этом случае проще перенести код из ветвей в отдельные классы стратегий.

Диаграмма классов паттерна приведена на рис. 2.7.

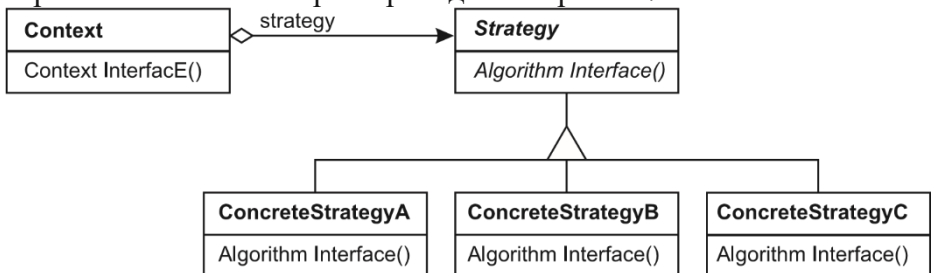


Рис. 2.7. Структура паттерна Стратегия

Паттерн состоит из следующих классов.

- **Strategy** (стратегия) — объявляет общий для всех поддерживаемых алгоритмов интерфейс. Класс Context пользуется этим интерфейсом для вызова конкретного алгоритма, определенного в классе ConcreteStrategy.

- **ConcreteStrategy** (конкретная стратегия) — наследник класса Strategy. Реализует алгоритм, использующий интерфейс, объявленный в классе Strategy.

- **Context** (контекст) — конфигурируется объектом класса ConcreteStrategy. Хранит ссылку на объект класса Strategy и может определять интерфейс, который позволяет объекту Strategy получить доступ к данным контекста.

Пример реализации. Термин «стратегия» означает лишь то, что у проблемы имеется несколько решений. Допустим, что вы забыли, как зовут встреченного вами знакомого. Из неловкого положения можно выйти несколькими способами, как показано в листинге 2.2 (идея примера позаимствована из [38]).

Листинг 2.2. Использование паттерна Strategy

```
#include <iostream>
using namespace std;
class NameStrategy {
public: virtual void greet() = 0;
};
class SayHi : public NameStrategy {
public: void greet() { cout << "Привет! Как дела?" << endl; }
};
class Ignore : public NameStrategy {
public: void greet() { cout << "(Сделать вид, что не заметил человека)" << endl; }
};
class Admission : public NameStrategy {
public: void greet() { cout << "Простите, я забыл Ваше имя." << endl; }
};
class Context { // -----Контекст управляет выбором стратегии:
NameStrategy& strategy;
public:
    Context(NameStrategy& strat) : strategy(strat) {}
    void greet() { strategy.greet(); }
};
int main() {
    SayHi      sayhi;
    Ignore     ignore;
    Admission  admission;
    Context c1(sayhi), c2(ignore), c3(admission);
    c1.greet(); c2.greet(); c3.greet();
}
```

Завершая краткое введение в паттерны проектирования, заметим, что хороший критерий профессионализма — выделение паттернов самостоятельно,

на основании собственного опыта. При решении следующих задач мы покажем, как это делается.

Проектирование программы с учетом будущих изменений

Одним из показателей качества программной системы является ее способность к модификациям в связи с появлением новых требований заказчика. На предыдущем семинаре уже отмечалось, что большое значение имеет правильная декомпозиция системы на компоненты (модули, классы, функции).

Однако тактические решения локальных проблем также влияют на то, сколь сложным окажется добавление новой функциональности к разработанному ранее проекту. Сейчас мы рассмотрим некоторые концепции поиска таких решений.

Одной из причин негибкости процедурно-ориентированных систем является частое использование в них управляющих конструкций на базе оператора `switch`. Рассмотрим типичную ситуацию: некоторая функция `SomeFunc` в зависимости от значения параметра `mode` вызывает одну из обрабатывающих процедур:

```
typedef enum { mode1, mode2, mode3 } ModeType;
void SomeFunc (ModeType mode) {
    switch (mode) {
        case mode1: DoSomething1(); break;
        case mode2: DoSomething2(); break;
        case mode3: DoSomething3(); break;
    }
}
```

Допустим, что перечень значений перечисляемого типа `ModeType` согласован с заказчиком и утвержден техническим заданием. Вся система, таким образом, разрабатывается с использованием утвержденного перечня. Однако через некоторое время заказчик обнаруживает, что необходимо предусмотреть еще одно значение параметра `mode`. Хорошо, если он еще платежеспособен и сможет заказать модификацию системы. Но для разработчиков наступят черные дни, когда придется искать по всем модулям переключателя `switch` и вносить нужные добавления. При этом потребуются полное повторное тестирование всех модулей, использующих эту функцию, так как могут появиться неожиданные побочные эффекты.

Попытаемся найти более удачное решение проблемы типа «переключатель». Одна из рекомендаций, приведенных в [6], звучит так: «Найдите то, что должно или может *измениться* в вашем дизайне, и *инкапсулируйте сущности*, подверженные изменениям!» В нашем случае переменной сущностью являются вызываемые процедуры. Можем ли мы их инкапсулировать? Да — если реализуем эти процедуры как виртуальные методы полиморфных объектов!

Идея заключается в следующем: объект типа «переключатель» (класс `Switch`) должен иметь дело с некоторой абстрактной «процедурой вообще», а это можно реализовать, создав абстрактный класс `AbstractEntity` с чисто виртуальным методом `DoSomething`. Конкретные процедуры в виде одноименных замещенных методов будут содержаться в производных от `AbstractEntity` классах `Entity1`, `Entity2`, и т. д. Чтобы делать «осознанный» выбор, класс `Switch` должен быть осведомлен об адресах объектов `Entity1`, `Entity2`, и т. д. Это можно обеспечить, поместив список адресов в одно из его полей типа `std::vector<AbstractEntity*>`⁴. Описываемая идея поясняется на диаграмме классов (рис. 2.8).

Обратите внимание, что между классами `Switch` и `AbstractEntity` имеется отношение композиции (стрелка с закрашенным ромбиком), так как класс `Switch` содержит поле типа `std::vector<AbstractEntity*>`. В то же время класс `Switch` *использует* класс `AbstractEntity` (пунктирная стрелка), поскольку один из его методов возвращает значение типа `AbstractEntity*`.

Как же все это работает? Клиент обращается с запросом `SelectEntity` к объекту `Switch` (то есть вызывает одноименный метод). Неважно, как реализован метод `SelectEntity`, существенно то, что он возвращает значение типа `AbstractEntity*`, содержащее адрес объекта одного из подклассов класса `AbstractEntity`. После этого через полученный указатель клиент вызывает конкретную процедуру `DoSomething` одного из объектов `Entity1`, `Entity2`, ...

Чем примечательно предлагаемое решение? Необычайной легкостью модификации. Если возникла необходимость дополнить список переключаемых сущностей еще одним *k*-м экземпляром, то достаточно добавить производный класс `EntityK`, объявить объект этого класса и добавить адрес нового объекта в список `vector<AbstractEntity*>`.

Очевидно, что предложенное на рис. 2.8 решение есть не что иное, как еще один *паттерн проектирования*, который мы назвали `Switch`. Пример его реализации приводится в решении задачи 2.1. Понятно, что скорее всего аналогичное решение уже неоднократно применялось программистами на практике, но ведь решение становится паттерном только после его описания на требуемом уровне абстракции!

⁴ Класс `std::vector` содержится в стандартной библиотеке шаблонов STL, которая изучается на семинаре 15. Тем не менее мы будем пользоваться этим удобным инструментом работы с динамическими массивами уже сейчас.

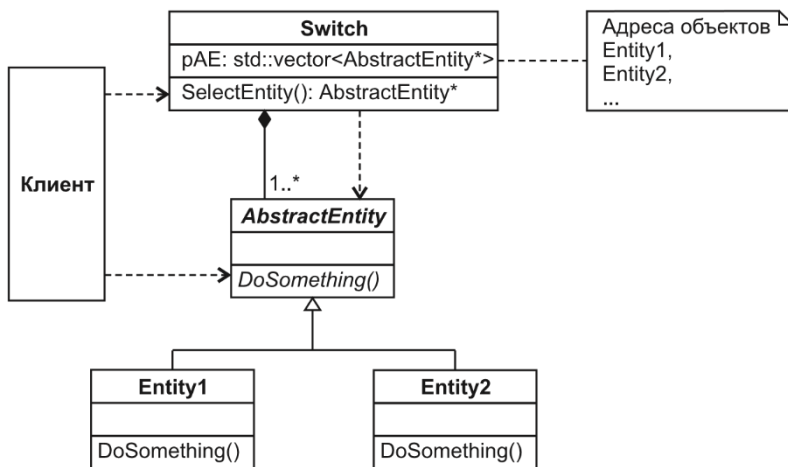


Рис. 2.8. Паттерн Switch (переключатель)

Задача 2.1. Функциональный калькулятор

Разработать программу, имитирующую работу функционального калькулятора, который позволяет выбрать с помощью меню какую-либо из известных ему функций, затем предлагает ввести значение аргумента и, возможно, коэффициентов и после ввода выдает соответствующее значение функции.

В первой версии калькулятора «база знаний» содержит две функции:

- экспоненту $y = e^x$;
- линейную функцию $y = a x + b$.

Решение задачи начнем с выявления понятий (классов) и их взаимосвязей.

Интерфейс пользователя нашего калькулятора, как следует из его описания, должен обеспечить меню для выбора вида функции. У нас уже есть опыт создания меню для консольных приложений, в которых отсутствует оконная графика, — например, в задаче 10.2. Функция `menu` в той программе выводит список выбираемых пунктов с номерами и после ввода пользователем номера пункта возвращает это значение главной функции. Далее следует традиционный переключатель:

```

switch (Menu()) {
    case 1: Show(...); break;
    case 2: Move(...); break;
    ...
    case 5: // Конец работы
}

```

Но вряд ли стоит повторять такое решение. Ведь мы уже знаем, что подобная реализация может резко усложнить модификацию программы. Мы

также вооружены новым паттерном проектирования `Switch`, который дает красивое, легко модифицируемое решение. Попробуем применить этот паттерн на практике. Несложно увидеть, что роль класса `Switch` здесь будет играть класс `Menu`.

Что является изменяемой сущностью в нашей задаче? — Вид функции, для которой нужно выполнить вычисления (в более общем случае — вид некоторого объекта, для которого нужно выполнить некоторую операцию). Следовательно, требуется инкапсулировать эту изменяемую сущность так, чтобы класс `Menu` имел дело с некоторой абстрактной «функцией вообще» (с некоторым «объектом вообще»). Отсюда вывод: необходим абстрактный класс `Function`, обеспечивающий единый унифицированный интерфейс для всех его потомков, в данном случае — для классов `Exp` и `Line`. В результате мы приходим к диаграмме классов на рис. 2.9.

На диаграмме, естественно, показаны не все элементы классов, а только наиболее существенные для данного этапа анализа. Кроме метода `Calculate`, выполняющего роль метода `DoSomething` в паттерне `Switch`, появился метод `GetName`, извлекающий наименование функции. Этим методом будет пользоваться объект `Menu` для вывода перечня выбираемых функций.

Прежде чем привести код программы, скажем несколько слов о работе с классами `std::string` и `std::vector`. Более подробно они изучаются на семинаре 12.

Класс `string` позволяет создать строку `s` типа `string`, с которой очень удобно работать: ее можно инициализировать значением C-строки: `std::string s("C-строка");` или присвоить ей значение другой строки посредством операции присваивания `=`. Память, выделенная для объекта `s`, автоматически освобождается, как только программа покидает область его видимости. Для использования класса необходимо подключить заголовочный файл `<string>`.

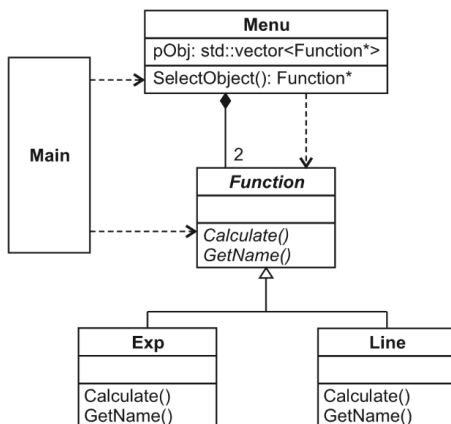


Рис.2.9. Диаграмма классов программы «Функциональный калькулятор»

Контейнер `vector` является шаблонным классом и позволяет создавать динамические массивы (термины «*вектор*» и «*динамический массив*» мы будем использовать как синонимы). Для использования класса необходимо подключить заголовочный файл `<vector>`. В классе есть конструктор по умолчанию, создающий вектор нулевой длины, а также конструктор с инициализацией создаваемого вектора обычным одномерным C-массивом. Можно добавить в конец имеющегося вектора новый элемент с помощью метода `push_back`. Доступ к любому элементу вектора можно выполнять по индексу, как и в обычном массиве. Текущее количество элементов в векторе можно определить методом `size`.

Поскольку `vector` — шаблонный класс, он позволяет создавать конкретные экземпляры классов для любого типа элементов, задаваемого в качестве аргумента в угловых скобках после имени класса. Поясним использование класса на примере, в котором показано использование векторов `v1`, `v2`, `v3`, предназначенных для хранения элементов типа `char`, `int` и `double` соответственно:

```
#include <iostream>
#include <vector>
using namespace std;
int main() {
    int a[5] = { 5, 4, 3, 2, 1 };
    double b[] = { 1.1, 2.2, 3.3, 4.4 };
    vector<char> v1;
    v1.push_back('A'); v1.push_back('B'); v1.push_back('C');
    for (int i = 0; i < v1.size(); ++i) cout << v1[i] << ' '; cout
<< endl;
    vector<int> v2(a, a + 5);
    for (int i = 0; i < v2.size(); ++i) cout << v2[i] << ' '; cout
<< endl;
    vector<double> v3(b, b + sizeof(b) / sizeof(double));
    for (int i = 0; i < v3.size(); ++i) cout << v3[i] << ' '; cout
<< endl; }
```

Эта программа должна вывести на экран:

```
A B C
5 4 3 2 1
2.2 3.3 4.4
```

Обратите внимание на два способа инициализации вектора одномерным массивом. Первый (для вектора `v2`) используется, когда длина массива задана константой. Второй (для вектора `v3`) оказывается единственным возможным, когда длина массива определяется на стадии компиляции. В листинге 2.3 приведена реализация программы «Функциональный калькулятор».

Листинг 2.3. Функциональный калькулятор

```

////////////////////////////////////Function.h
#ifndef FUNCTION_H
#define FUNCTION_H
#include <string>
class Function {
public:

    virtual ~Function() {}
    virtual const std::string& GetName() const = 0;           // 1
    virtual void Calculate() = 0;
protected:
    double x;          // аргумент
};
#endif /* FUNCTION_H */
// ////////////////////////////////////// Exp.h
#include "Function.h"
class Exp : public Function { // ----- Класс для представления
функции  $y = e^x$ 
public:
    Exp() : name("e ^ x") {}
    const std::string& GetName() const { return name; }
    void Calculate();
protected:
    std::string name;    // математическое обозначение функции
};
extern Exp f_exp;
// ////////////////////////////////////// Exp.cpp
#include <iostream>
#include <math.h>
#include "Exp.h"
using namespace std;
void Exp::Calculate() {
    cout << "Calculation for function  $y =$ " << name << endl;
    cout << "Enter x = ";
    cin >> x; cin.get();
    cout << " $y =$ " << exp(x) << endl; cin.get();
}
Exp f_exp;    // Глобальный объект
// ////////////////////////////////////// Line.h
#include "Function.h"
class Line : public Function { // --- Класс для представления
функции  $y = a * x + b$ 
public:
    Line() : name("a * x + b") {}
    const std::string& GetName() const { return name; }
    void Calculate();
protected:
    std::string name;    // математическое обозначение функции
    double a, b;
};

```

```

extern Line f_line;
// //////////////////////////////////////// Line.cpp
#include <iostream>
#include "Line.h"
using namespace std;
void Line::Calculate() {
    cout << "Calculation for function y = " << name << endl;
    cout << "Enter a = ";    cin >> a;
    cout << "Enter b = ";    cin >> b;
    cout << "Enter x = ";    cin >> x;
    cin.get();
    cout << "y = " << (a * x + b) << endl;
    cin.get();
}
Line f_line;    // Глобальный объект
// //////////////////////////////////////// Menu.h
#include <vector>
#include "Function.h"
class Menu {
public:
    Menu(std::vector<Function*>);
    Function* SelectObject() const;
private:

    int SelectItem(int) const;
    std::vector<Function*> pObj;
};
// //////////////////////////////////////// Menu.cpp
#include <iostream>
#include "Menu.h"
using namespace std;
Menu::Menu(vector<Function*> _pObj) : pObj(_pObj) {
    pObj.push_back(0);    // для выбора пункта 'Exit'
}
Function* Menu::SelectObject() const {
    int nItem = pObj.size();
    cout << "=====\n";
    cout << "Select one of the following function:\n";
    for (int i = 0; i < nItem; ++i) {
        cout << i+1 << ". ";    // номер пункта меню на 1 больше
индекса массива pObj
        if (pObj[i]) cout << pObj[i]->GetName() << endl;
        else cout << "Exit" << endl;
    }
    int item = SelectItem(nItem);
    return pObj[item - 1];
}
int Menu::SelectItem(int nItem) const {
    cout << "-----\n";
    int item;

```

```

while ( true ) {
    cin >> item;
    if ((item > 0) && (item <= nItem) && (cin.peek() == '\n')) {
        cin.get(); break;
    }

    else {
        cout << "Error (must be number from 1 to " << nItem <<
        "):" << endl;
        cin.clear();
        while ( cin.get() != '\n' ) {};
    }
    return item;
}
}
//////////////////////////////////// Main.cpp
#include <iostream>
#include "Exp.cpp"
#include "Line.cpp"
#include "Menu.cpp"
using namespace std;
Function* pObj[] = { &f_exp, &f_line };
vector<Function*> funcList(pObj, pObj + sizeof(pObj) /
sizeof(Function*));
int main() {
    Menu menu(funcList);
    while ( Function* pObj = menu.SelectObject() )
        pObj->Calculate();
    cout << "Bye!\n";
}

```

Несколько пояснений по тексту программы.

В абстрактном классе `Function` конструктор отсутствует. Класс содержит *виртуальный* деструктор (см. рекомендацию, приведенную в разделе «Виртуальные методы»). Существует рекомендация размещать в абстрактном классе *только методы*, а все поля объявлять в производных классах (то есть делать класс чисто интерфейсным). Этим гарантируется, что любые изменения и дополнения не затронут абстрактный класс. В данном случае мы все-таки описали поле `x` для значения аргумента, поскольку рассматриваем только функции с одним аргументом и при этом полагаем, что аргумент всегда имеет тип `double`. А вот значения коэффициентов содержатся в полях производных классов.

Обратите внимание на сигнатуру метода `GetName` в классе `Function` (*оператор 1*). Метод объявлен константным, так как он не должен изменять состояние полей класса. Кроме того, чтобы исключить вызов конструктора копирования, мы хотим вернуть значение по ссылке: `std::string&`. Однако

сигнатуру `virtual std::string& GetName const` не пропустит компилятор, так как возврат по ссылке `string&` противоречит модификатору `const`. Выход из этого противоречия — возврат по константной ссылке: `const std::string&`.

Конкретные методы `Calculate` в классах `Exp` и `Line` обеспечивают ввод исходных данных (аргумент и, по необходимости, значения коэффициентов) и вычисляют значение соответствующей функции. Отметим, что в модуле `Exp.cpp` объявлен глобальный объект `Exp f_exp`, а в модуле `Line.cpp` — глобальный объект `Line f_line`. Конструктор класса `Menu` обеспечивает инициализацию поля `pObj` вектором объектов типа `Function*`, передаваемым через его аргумент. После копирования аргумента в поле `pObj` (в инициализаторе конструктора) к вектору с помощью метода `pObj.push_back(0)` добавляется нулевой указатель, который используется для реализации пункта меню `Exit` (см. метод `Menu::SelectObject`).

В методе `Menu::SelectObject` количество пунктов меню определяется вызовом `pObj.size`. После вывода оператором `for` списка пунктов меню вызывается метод `SelectItem`, обеспечивающий защищенный от ошибок ввод пользователем номера выбираемого пункта. Заметим, что метод `SelectItem` объявлен в секции `private` класса `Menu`, так как он используется только внутри класса.

В главном модуле `main.cpp` объявлен глобальный массив `pObjs[]`, содержащий список адресов объектов `&f_exp`, `&f_line`. Этим массивом инициализируется вектор `funcList`.

Функция `main` лаконична: в ней объявлен объект `menu` класса `Menu`, которому передается вектор `funcList`, а затем идет цикл `while`, работа которого очевидна. Метод `menu.SelectObject` возвращает адрес конкретного объекта (`f_exp` или `f_line`), через который вызывается метод `Calculate` соответствующего класса. При возврате нулевого адреса цикл завершается, и программа завершается.

Необходимо также пояснить, по какому принципу размещаются директивы `using` в тексте модулей. Напомним, что благодаря директиве `using namespace std` становятся видимыми (известными компилятору) все имена из пространства имен `std` стандартной библиотеки C++. В нашей программе это, например, имена `cin`, `cout`, `string`, `vector`.

Есть и другие способы сделать видимым некоторое конкретное имя: с помощью объявления `using` (например, `using std::string` — действует до конца блока, в котором оно сделано) или с помощью квалификатора `std::` (например, `std::string`). Проще всего использовать директиву `using`, но при этом возникают две опасности: возможность конфликта имен из пространства `std` с вашими именами и, что более неприятно, возможность неоднозначной трактовки кода компилятором при беспорядочном употреблении директив

using. В этом вопросе мы солидарны с Гербом Саттером [25] и приводим ниже его рекомендации.

СОВЕТ

Правило №1. *Не используйте директивы using, равно как и объявления using, в заголовочных файлах.* Именно в этом случае возможно появление указанных выше проблем. Поэтому в заголовочных файлах используйте только квалификатор std:: перед конкретным именем.

Правило №2. *В файлах реализации директивы и объявления using не должны располагаться перед директивами include.* Иначе использование using может изменить семантику заголовочного файла из-за введения неожиданных имен.

Откомпилируйте и проверьте работу программы. Еще раз сравните код функций main в этой программе и в задаче 10.2, чтобы увидеть элегантность нового решения. А теперь представим, что от заказчика поступило предложение сделать наш калькулятор более «мощным», добавив в него новую функцию — параболу. Доработка сведется к добавлению в проект класса Parabola:

```
// //////////////////////////////////////// Parabola.h
#include "Function.h"
// Класс для представления функции  $y = a * x^2 + b * x + c$ 
class Parabola : public Function {
public:
    Parabola() : name("a * x^2 + b * x + c") {}
    const std::string& GetName() const { return name; }
    void Calculate();
protected:
    std::string name; // мат. обозначение функции
    double a, b, c;
};
extern Parabola f_parabola;

// //////////////////////////////////////// Parabola.cpp
#include <iostream>
#include "Parabola.h"
using namespace std;
void Parabola::Calculate() {
    cout << "Calculation for function y = " << name << endl;
    cout << "Enter a = ";    cin >> a;
    cout << "Enter b = ";    cin >> b;
    cout << "Enter c = ";    cin >> c;
    cout << "Enter x = ";    cin >> x;
    cin.get();
    cout << "y = " << (a * x * x + b * x + c) << endl;
    cin.get();
}
```

```
Parabola f_parabola;    // Глобальный объект
```

Кроме этого, в основном модуле `main.cpp` необходимо добавить директиву `#include "Parabola.h"`; а в списке инициализации массива `pObjs` — адрес нового объекта `&f_parabola`. Все! Весь прежний код проекта остается без изменений. Новая версия калькулятора готова.

Задача 2.2. Продвинутый функциональный калькулятор

Наш заказчик никак не уgomонится. Теперь ему пришла идея расширить набор операций, которые способен выполнять функциональный калькулятор: пусть он либо вычисляет значение заданной функции для некоторого аргумента, либо выполняет табуляцию функции в заданном интервале с заданным шагом.

Идея неплохая. Это позволит резко повысить спрос на рынке на наш калькулятор, поскольку калькуляторы других фирм не умеют это делать. За работу!

Нам нужно решить две проблемы. Во-первых, видимо, придется добавить в базовый класс `Function` новый чисто виртуальный метод `Tabulation`, а также заместить его во всех подклассах. То есть придется вносить изменения в код всех классов иерархии `Function`, а это плохо. Во-вторых, в продвинутом функциональном калькуляторе необходимо реализовать двухуровневое меню: на первом уровне выбирается вид функции, на втором уровне — вид операции. В предыдущей задаче мы справились с проблемой *легко модифицируемого кода* одноуровневого меню с помощью паттерна проектирования `Switch`. Неужели на втором уровне придется писать по старинке гирлянду `case'ов` в блоке оператора `switch`? Как мы знаем, это плохо для дальнейшей модификации программы.

Попробуем найти более удачное решение. А что, если попытаться развить идею одноуровневого переключателя (паттерн `Switch`) для случая двух уровней? Например, переключатель такого типа мог бы выбирать на первом уровне некоторую конкретную сущность `EntityA_I` из иерархии абстрактного базового класса `AbstractEntityA`, а затем, на втором уровне, — некоторую конкретную сущность `EntityB_J` из иерархии абстрактного базового класса `AbstractEntityB`, после чего клиенту предоставляется возможность реализовать требуемую ассоциацию между выбранными сущностями⁵.

Паттерн проектирования `DoubleSwitch`, реализующий эту идею, имеет диаграмму классов, показанную на рис. 2.10.

Обратите внимание, что метод `SelectEntityA` класса `Switch` возвращает указатель `AbstractEntityA*`, который передается далее в качестве аргумента методу `SelectEntityB`, осуществляющему выбор на втором уровне уже для конкретной сущности первого уровня. Абстрактный класс второго уровня

⁵ Красиво сказано, не правда ли?

AbstractEntityB позволяет клиенту абстрагироваться от вида выполняемой операции: он имеет дело с «операцией вообще» Operate.

После выбора на втором уровне благодаря полиморфизму клиент может вызвать метод Operate, относящийся к конкретной сущности второго уровня. Передача ссылки AbstractEntityA* методу Operate разрешает последнему иметь доступ к атомарным операциям AtomOperate1, AtomOperate2 и т. п., определенным в конкретном объекте иерархии AbstractEntityA. Совокупность этих примитивов должна быть достаточной для реализации любого метода Operate в иерархии AbstractEntityB. Как правило, это операции, обеспечивающие доступ ко всем защищенным или закрытым полям объектов иерархии AbstractEntityA или выполняющие обработку информации, специфичную для данного класса.

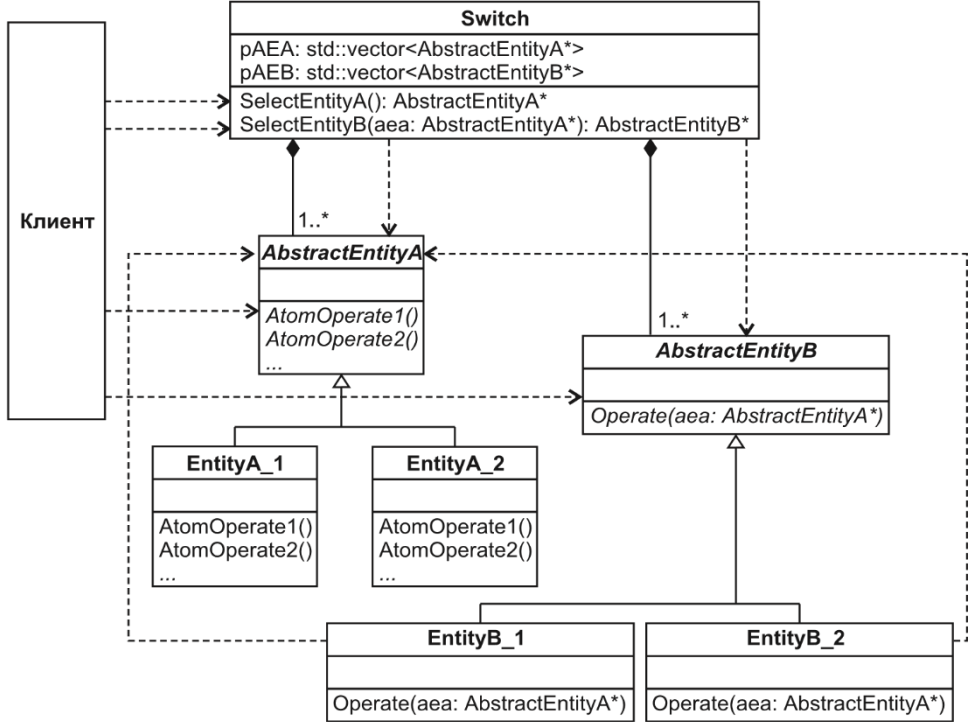


Рис. 2.10. Паттерн DoubleSwitch

Теперь попытаемся применить этот паттерн к нашей задаче. Ясно, что в качестве класса Switch будет выступать класс Menu, как и в предыдущей задаче, а в качестве класса AbstractEntityA — класс Function. Роль класса AbstractEntityB мы поручим новому абстрактному классу Action, обеспечивающему унифицированный интерфейс для конкретных классов Calculation, Tabulation, AnyAction. Последний класспустышку мы добавим

просто для иллюстрации того, что список выполняемых операций может легко расширяться.

Роль метода `AbstractEntityB::Operate(AbstractEntityA*)` будет возложена на метод `Action::Operate(Function*)`. Ну и, наконец, метод `Calculate` перекоちует из иерархии базового класса `Function` в конкретный класс `Calculation`, принадлежащий иерархии базового класса `Action`. Взамен мы должны пополнить иерархию `Function` такими атомарными операциями, как `SetArg` — установить значение аргумента, `SetCoeff` — установить значения коэффициентов, `GetVal` — получить значение функции.

Мы не будем приводить здесь диаграмму классов предлагаемого решения, оставляя это в качестве упражнения читателю. Текст программы приведен в листинге 2.4.

Листинг 2.4. Продвинутый функциональный калькулятор

[illegible]

```

class Line : public Function { // --- Класс для представления
функции  $y = a * x + b$ 
public:
    Line() : name("a * x + b") {}
    const std::string& GetName() const { return name; }
    void SetCoeff();
    double GetVal() const { return (a * x + b); }
private:
    std::string name;    // мат. обозначение функции
    double a, b;
};

extern Line f_line;
// //////////////////////////////////////// Line.cpp
#include <iostream>
#include "Line.h"
using namespace std;
void Line::SetCoeff() {
    cout << "Enter a = ";    cin >> a;
    cout << "Enter b = ";    cin >> b;
}
Line f_line;    // Глобальный объект
// //////////////////////////////////////// Action.h
#ifndef ACTION_H
#define ACTION_H
#include "Function.h"
class Action {
public:
    virtual ~Action() {}
    virtual void Operate(Function*) = 0;
    virtual const std::string& GetName() const = 0;
};
#endif /* ACTION_H */
// //////////////////////////////////////// Calculation.h
#include "Action.h"
class Calculation : public Action {
public:
    Calculation() : name("Calculation") {}
    const std::string& GetName() const { return name; }
    void Operate(Function*);
private:
    std::string name;    // обозначение операции
};

extern Calculation calculation;
// //////////////////////////////////////// Calculation.cpp
#include <iostream>
#include "Calculation.h"
using namespace std;
void Calculation::Operate(Function* pFunc) {
    cout << "Calculation for function y = " << pFunc->GetName() <<
endl;

```

```

    pFunc->SetCoeff();
    double x;
    cout << "Enter x = ";    cin >> x;
    cin.get();
    pFunc->SetArg(x);
    cout << "y = " << pFunc->GetVal() << endl;
    cin.get();
}
Calculation calculation;    // Глобальный объект
// ////////////////////////////////////// Tabulation.h
#include "Action.h"
class Tabulation : public Action {
public:
    Tabulation() : name("Tabulation") {}
    const std::string& GetName() const { return name; }
    void Operate(Function*);
private:
    std::string name;    // обозначение операции
};
extern Tabulation tabulation;
// ////////////////////////////////////// Tabulation.cpp
#include <iostream>
#include <iomanip>
#include "Tabulation.h"
using namespace std;
void Tabulation::Operate(Function* pFunc) {
    cout << "Tabulation for function y = ";
    cout << pFunc->GetName() << endl;
    pFunc->SetCoeff();
    double x_beg, x_end, x_step;
    cout << "Enter x_beg = ";    cin >> x_beg;
    cout << "Enter x_end = ";    cin >> x_end;
    cout << "Enter x_step = ";    cin >> x_step;
    cin.get();
    cout << "-----" << endl;
    cout << "      x          y" << endl;
    cout << "-----" << endl;
    for (double x = x_beg; x <= x_end; x += x_step) {
        pFunc->SetArg(x);
        cout << setw(6) << x << setw(14) << pFunc->GetVal() << endl;
    }
    cin.get();
}
Tabulation tabulation;    // Глобальный объект
// ////////////////////////////////////// AnyAction.h
#include "Action.h"
class AnyAction : public Action {
public:
    AnyAction() : name("Any action") {}
    const std::string& GetName() const { return name; }

```

```

        void Operate(Function*);
private:
    std::string name;    // обозначение операции
};
extern AnyAction any_action;
// //////////////////////////////////////// AnyAction.cpp
#include <iostream>
#include "AnyAction.h"
using namespace std;
void AnyAction::Operate(Function*) {
    cout << "Здесь могла бы быть Ваша реклама!" << endl;
    cin.get();
}
AnyAction any_action;    // Глобальный объект
// //////////////////////////////////////// Menu.h
#include <vector>
#include "Function.h"
#include "Action.h"
class Menu {
public:
    Menu(std::vector<Function*>, std::vector<Action*>);
    Function* SelectObject() const;
    Action* SelectAction(Function*) const;
private:
    int SelectItem(int) const;
    std::vector<Function*> pObj;
    std::vector<Action*> pAct;
};
// //////////////////////////////////////// Menu.cpp
#include <iostream>
#include "Menu.h"
using namespace std;
Menu::Menu(vector<Function*> _pObj, vector<Action*> _pAct)
    : pObj(_pObj), pAct(_pAct) {
    pObj.push_back(0);    // для выбора пункта 'Exit'
}
Function* Menu::SelectObject() const {
    // Возьмите код аналогичной функции из листинга 2.3
}
Action* Menu::SelectAction(Function* pObj) const {
    int nItem = pAct.size();
    cout << "=====\n";
    cout << "Select one of the following Action:\n";
    for (int i = 0; i < nItem; ++i) {
        cout << i + 1 << ". " << pAct[i]->GetName() << endl;
    }
    int item = SelectItem(nItem);
    return pAct[item - 1];
}
int Menu::SelectItem(int nItem) const {

```

```

    // Возьмите код аналогичной функции из листинга 2.3
}
// ////////////////////////////////////// Main.cpp
#include <iostream>
#include "Exp.cpp"
#include "Line.cpp"
#include "Calculation.cpp"
#include "Tabulation.cpp"
#include "AnyAction.cpp"
#include "Menu.cpp"
using namespace std;
Function* pObj[] = { &f_exp, &f_line };
vector<Function*> funcList(pObj, pObj +
sizeof(pObj)/sizeof(Function*));
Action* pActs[] = { &calculation, &tabulation, &any_action };
vector<Action*> operList(pActs, pActs + sizeof(pActs) /
sizeof(Action*));
int main() {
    Menu menu(funcList, operList);
    while ( Function* pObj = menu.SelectObject() ) {
        Action* pAct = menu.SelectAction(pObj);
        pAct->Operate(pObj);
    }
    cout << "Bye!\n";
}

```

Обратите внимание на следующее:

- метод `Exp::SetCoeff` имеет пустое тело, то есть ничего не делает, поскольку для вычисления экспоненты коэффициенты не требуются;

- в связи с тем, что все методы класса `Exp` — встроенные, файл реализации `Exp.cpp` содержит только объявление глобального объекта `f_exp`.

Отметим, что предложенное решение позволяет легко модифицировать наш калькулятор, решая проблемы добавления как новых функций, так и новых операций.

Задача 2.3. Работа с объектами символьных и шестнадцатеричных строк

Написать программу, демонстрирующую работу с объектами двух типов: символьная строка (`SymbString`) и шестнадцатеричная строка (`HexString`), для чего создать систему соответствующих классов.

Каждый объект должен иметь как минимум два атрибута: идентификатор и значение, представленные в виде строк символов. В поле значения объекты `SymbString` могут хранить произвольный набор символов, в то время как объекты `HexString` — только изображение шестнадцатеричного числа.

Клиенту (функции main) должны быть доступны следующие операции: создать объект, удалить объект, показать значение объекта (строку символов), показать изображение эквивалентного десятичного числа (только для объектов HexString), показать изображение эквивалентного двоичного числа (только для объектов HexString). Предусмотреть меню, позволяющее демонстрировать эти операции.

При решении этой задачи мы воспользуемся еще одним из видов диаграмм UML — *диаграммой видов деятельности (Activity Diagram)*, чтобы расширить наш кругозор в сфере современных методов проектирования.

Диаграмма видов деятельности UML очень похожа на старые блок-схемы алгоритмов. В ней точками принятия решений и переходами описывается последовательность шагов, именуемых в UML *видами деятельности (Activity)*. Каждый вид деятельности, ассоциируемый обычно с некоторой процедурой, изображается прямоугольником с закругленными углами, а переходы между ними — стрелками. Точки принятия решений можно изображать одним из двух способов: либо просто показываются все возможные переходы после завершения некоторого вида деятельности, либо изображается переход к маленькому ромбику, похожему на блок ветвления, а затем все возможные пути выходят из этого ромбика.

Анализируя условие задачи, мы пытаемся, во-первых, выявить объекты/классы предметной области и, во-вторых, сконструировать основной алгоритм, работающий с этими объектами. В данном случае представляется естественным алгоритм, изображенный в виде диаграммы видов деятельности на рис. 2.11.

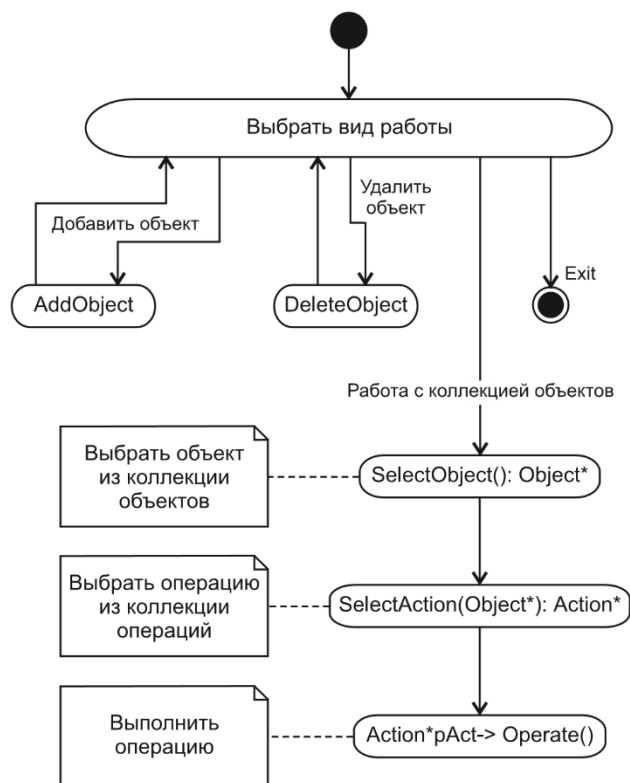


Рис. 2.11. Диаграмма видов деятельности для задачи 2.3

Начальная точка алгоритма обозначается в виде закрашенного кружка, конечная — в виде «глазка». При необходимости на диаграмме можно размещать комментарии в виде прямоугольника с текстом, напоминающего листок бумаги с отогнутым углом.

Графическое представление алгоритма позволяет более четко выявить решаемые проблемы. Так, меню в нашей программе должно обеспечить выбор на трех уровнях: 1) выбор вида работы (добавить объект к коллекции, удалить объект из коллекции, работать с коллекцией объектов, выйти из программы); 2) выбор объекта из коллекции объектов; 3) выбор операции из коллекции операций.

Опираясь на опыт реализации двухуровневого меню в предыдущей задаче, подумаем о способе реализации трехуровневого меню. Первой приходит мысль: а не создать ли нам паттерн проектирования `ThreeLevelSwitch`, развивающий идеи паттерна `DoubleSwitch`? Однако с каждым повышением размерности переключателя код будет становиться все сложнее для понимания. Поэтому важно вовремя остановиться, помня, что любую идею можно довести до абсурда.

Принимаем следующий постулат: перечень действий (работ), выбираемых на первом уровне, тщательно продуман заказчиком, и вероятность его изменения крайне мала. В связи с этим первый уровень меню мы реализуем без использования паттернов — в блоке оператора `switch`, а вот для второго и третьего уровней применим уже испытанный паттерн `DoubleSwitch`, который позволит в будущем легко добавлять как типы обрабатываемых объектов, так и виды применяемых операций.

Теперь можно поговорить о классах. Два из них: `SymbString` и `HexString` — продиктованы условием задачи. Именно объекты этих классов будут пополнять коллекцию объектов, с которыми в дальнейшем можно выполнять те или иные операции (в нашем случае — показать значение объекта, показать изображение эквивалентного десятичного числа и т. д.). Для реализации паттерна проектирования `DoubleSwitch` нам потребуется абстрактный базовый класс `AString`, наследниками которого будут классы `SymbString` и `HexString`, а также абстрактный базовый класс `Action`, имеющий в качестве дочерних классы `ShowStr`, `ShowDec`, `ShowBin`.

Для реализации меню создадим, как и в предыдущей задаче, класс `Menu`, содержащий аналогичные методы `SelectObject` и `SelectAction`, но к ним в компанию добавим метод `SelectJob`. Поскольку список операций, применяемых к объектам, заранее известен (эти операции инкапсулированы в классах `ShowStr`, `ShowDec` и `ShowBin`), этот список, как и раньше, разместим в поле `pAct`. А вот со списком объектов ситуация более сложная, чем в задаче 2.2, так как эти объекты должны создаваться и уничтожаться динамически — в процессе работы программы. Пока отложим этот вопрос, мы вскоре к нему вернемся.

Какие сущности в решаемой задаче не охвачены перечисленными классами? В условии задачи упоминаются операции: *создать объект* и *удалить объект*. Конечно, можно поручить эти операции функции `main`, разместив заодно в ней и коллекцию обрабатываемых объектов, но это было бы в высшей степени нехорошо! Почему? — Потому, что такое решение понизило бы сцепление внутри главного клиента `main`.

СОВЕТ

Прилагайте максимум усилий к тому, чтобы каждая часть кода — каждый модуль, класс, функция, — отвечала за выполнение одной четко определенной задачи.

Последуем этому совету и создадим класс `Factory`, отвечающий за создание и удаление объектов из иерархии `AString`. Класс будет содержать поле `std::vector<AString*> pObj` для хранения коллекции объектов, и два метода: `AddObject` и `DeleteObject`.

Пора переходить к кодированию. Предлагаемое решение приведено в листинге 2.5.

Листинг 2.5. Работа с объектами символьных и шестнадцатеричных строк

```
// //////////////////////////////////////// AString.h
#pragma once
#ifdef ASTRING_H
#define ASTRING_H
#include <string>
class AString {
public:
    virtual ~AString() {}
    virtual const std::string& GetName() const = 0;
    virtual const std::string& GetVal() const = 0;
    virtual int GetSize() const = 0;
};
#endif //ASTRING_H
// //////////////////////////////////////// SymbString.h
#pragma once
#include <string>
#include "AString.h"
class SymbString : public AString {
public:
    SymbString(std::string _name) : name(_name) {}
    SymbString(std::string _name, std::string _val) : name(_name),
val(_val) {}
    const std::string& GetName() const { return name; }
    const std::string& GetVal() const { return val; }
    int GetSize() const { return val.size(); }
private:
    std::string name, val;
};
// //////////////////////////////////////// HexString.h
#pragma once
#include <string>
#include "AString.h"
const std::string alph = "0123456789ABCDEF";
```

```

bool IsHexStrVal(std::string);
class HexString : public AString {
public:
    HexString(std::string _name) : name(_name) {}
    HexString(std::string, std::string);
    const std::string& GetName() const { return name; }
    const std::string& GetVal() const { return val; }
    int GetSize() const { return val.size(); }
private:
    std::string name, val;
};
// ////////////////////////////////////////// HexString.cpp
#include <iostream>
#include "HexString.h"
using namespace std;
bool IsHexStrVal(string _str) { // 1
    for (int i = 0; i < _str.size(); ++i)
        if (-1 == alph.find_first_of(_str[i]))
            return false;
    return true;
}
HexString::HexString(string _name, string _val) : name(_name) {
    if (IsHexStrVal(_val)) val = _val;
}
// ////////////////////////////////////////// Action.h
#pragma once
#ifdef ACTION_H
#define ACTION_H
#include "AString.h"
class Action {
public:
    virtual ~Action() {}
    virtual void Operate(AString*) = 0;
    virtual const std::string& GetName() const = 0;
protected:
    long GetDecimal(AString* pObj) const;
};
#endif /* ACTION_H */
// ////////////////////////////////////////// Action.cpp
#include <iostream>
#include "Action.h"
#include "HexString.h"
using namespace std;
long Action::GetDecimal(AString* pObj) const {
    if (dynamic_cast<HexString*>(pObj)) {
        long dest;
        string source = pObj->GetVal();
        sscanf(source.c_str(), "%lX", &dest);
        return dest;
    }
}

```

```

        else { cout << "Action not supported." << endl; return -1; }
    }
// ////////////////////////////////////// ShowStr.h
#pragma once
#include "Action.h"
class ShowStr : public Action {
public:
    ShowStr() : name("Show string value") {}
    void Operate(AString*);
    const std::string& GetName() const { return name; }
private:
    std::string name;    // обозначение операции
};
extern ShowStr show_str;
// ////////////////////////////////////// ShowStr.cpp
#include <iostream>
#include "ShowStr.h"
using namespace std;
void ShowStr::Operate(AString* pObj) {
    cout << pObj->GetName() << ": " << pObj->GetVal() << endl;
    cin.get();
}
ShowStr show_str;    // Глобальный объект//
// ////////////////////////////////////// ShowDec.h
#pragma once
#include "Action.h"
class ShowDec : public Action {
public:
    ShowDec() : name("Show decimal value") {}
    void Operate(AString*);
    const std::string& GetName() const { return name; }
private:
    std::string name;    // обозначение операции
};
extern ShowDec show_dec;
// ////////////////////////////////////// ShowDec.cpp
#include <iostream>
#include "ShowDec.h"
#include "HexString.h"
using namespace std;
void ShowDec::Operate(AString* pObj) {
    cout << pObj->GetName() << ": ";
    long decVal = GetDecimal(pObj);
    if (decVal != -1) cout << GetDecimal(pObj) << endl;
    cin.get();
}
ShowDec show_dec;    // Глобальный объект
// ////////////////////////////////////// ShowBin.h
#pragma once
#include "Action.h"

```

```

class ShowBin : public Action {
public:
    ShowBin() : name("Show binary value") {}
    void Operate(AString*);
    const std::string& GetName() const { return name; }
private:
    std::string GetBinary(AString*) const;
    std::string name;    // обозначение операции
};

extern ShowBin show_bin;
// ////////////////////////////////////// ShowBin.cpp
#include <iostream>
#include "ShowBin.h"
#include "ShowDec.h"
#include "AString.h"
using namespace std;
void ShowBin::Operate(AString* pObj) {
    cout << pObj->GetName() << ": " << GetBinary(pObj) << endl;
    cin.get();
}
string ShowBin::GetBinary(AString* pObj) const {
    int nBinDigit = 4 * pObj->GetSize();
    char* binStr = new char[nBinDigit + 1];
    for (int k = 0; k < nBinDigit; ++k)
        binStr[k] = '0';
    binStr[nBinDigit] = 0;
    long decVal = GetDecimal(pObj);
    if (-1 == decVal) return string("");
    int i = nBinDigit - 1;
    while ( decVal > 0 ) {
        binStr[i--] = 48 + (decVal % 2);
        decVal >>= 1;
    }
    string temp(binStr);
    delete [] binStr;
    return temp;
}

ShowBin show_bin;    // Глобальный объект
// ////////////////////////////////////// Factory.h
#pragma once
#ifndef FACTORY_H
#define FACTORY_H
#include <vector>
#include "AString.h"
class Factory {
    friend class Menu; public:
    Factory() {}
    void AddObject();
    void DeleteObject();
private:

```

```

        std::vector<AString*> pObj;
};
#endif //FACTORY_H
// //////////////////////////////////////// Factory.cpp
#include <iostream>
#include "Factory.h"
#include "Menu.h"
#include "SymbString.h"
#include "HexString.cpp"
using namespace std;
#define MAX_LEN_STR 100
void Factory::AddObject() {
    cout << "-----\n";
    cout << "Select object type:\n";
    cout << "1. Symbolic string" << endl;
    cout << "2. Hexadecimal string" << endl;
    int item = Menu::SelectItem(2);
    string name;
    cout << "Enter object name: ";
    cin >> name; cin.get();

    cout << "Enter object value: ";
    char buf[MAX_LEN_STR];
    cin.getline(buf, MAX_LEN_STR);
    string value = buf;
    AString* pNewObj;
    switch (item) {
        case 1: pNewObj = new SymbString(name, value); break;
        case 2: if (!IsHexStrVal(value)) {
                    cout << "Error!" << endl;
                    return;
                }
                pNewObj = new HexString(name, value); break;
    }
    pObj.push_back(pNewObj);      cout << "Object added." << endl;
}

void Factory::DeleteObject() {
    int nItem = pObj.size();
    if (!nItem) {
        cout << "There are no objects." << endl; cin.get();
        return;
    }
    cout << ".....\n";
    cout << "Delete one of the following Object:\n";
    for (int i = 0; i < nItem; ++i)
        cout << i + 1 << ". " << pObj[i]->GetName() << endl;
    int item = Menu::SelectItem(nItem);
    string objName = pObj[item - 1]->GetName();
    pObj.erase(pObj.begin() + item - 1);
    cout << "Object " << objName << " deleted." << endl;
}

```

```

        cin.get();
    }
// //////////////////////////////////////// Menu.h
#pragma once
#include <vector>
#include "AString.h"
#include "Action.h"
#include "Factory.h"
typedef enum { AddObj, DelObj, WorkWithObj, Exit } JobMode;
class Menu {
public:
    Menu(std::vector<Action*>);
    JobMode    SelectJob() const;
    AString*    SelectObject(const Factory&) const;
    Action*    SelectAction(const AString*) const;
    static int SelectItem(int);
private:
    std::vector<Action*> pAct;
};
// //////////////////////////////////////// Menu.cpp
#include <iostream>
#include "AString.h"
#include "SymbString.h"
#include "HexString.h"
#include "Menu.h"
using namespace std;
Menu::Menu(vector<Action*> _pAct) : pAct(_pAct) {}
JobMode Menu::SelectJob() const {
    cout << "=====\n";
    cout << "Select one of the following job mode:\n";
    cout << "1. Add object" << endl;
    cout << "2. Delete object" << endl;        cout << "3. Work with
object" << endl;
    cout << "4. Exit" << endl;
    int item = SelectItem(4);
    return (JobMode)(item - 1);
}
AString* Menu::SelectObject(const Factory& fctry) const {
    int nItem = fctry.pObj.size();
    if (!nItem) {
        cout << "There are no objects." << endl; cin.get();
        return 0;
    }
    cout << ".....\n";
    cout << "Select one of the following Object:\n";
    for (int i = 0; i < nItem; ++i) {
        cout << i + 1 << ". " << fctry.pObj[i]->GetName() << endl;
    }
    int item = SelectItem(nItem);
    return fctry.pObj[item - 1];
}

```

```

}
Action* Menu::SelectAction(const AString* pObj) const {
    if (!pObj) return 0;
    int nItem = pAct.size();
    cout << ". . . . .\n";
    cout << "Select one of the following Action:\n";
    for (int i = 0; i < nItem; ++i) {
        cout << i + 1 << ". " << pAct[i]->GetName() << endl;
    }
    int item = SelectItem(nItem);
    return pAct[item - 1];
}
int Menu::SelectItem(int nItem) {
    // Возьмите код аналогичной функции из листинга 2.3
}
// ////////////////////////////////////// Main.cpp
#include <iostream>
#include "Action.cpp"
#include "ShowStr.cpp"
#include "ShowDec.cpp"
#include "ShowBin.cpp"
#include "Factory.cpp"
#include "Menu.cpp"
using namespace std;
Action* pActs[] = { &show_str, &show_dec, &show_bin };
vector<Action*> actionList(pActs, pActs +
sizeof(pActs)/sizeof(Action*));
int main() {
    Factory factory;
    Menu menu(actionList);
    JobMode jobMode;
    while ( (jobMode = menu.SelectJob()) != Exit ) {
        switch (jobMode) {
            case AddObj: factory.AddObject(); break;
            case DelObj: factory.DeleteObject(); break;
            case WorkWithObj:
                AString* pObj = menu.SelectObject(factory);
                Action* pAct = menu.SelectAction(pObj);
                if (pAct) pAct->Operate(pObj); break;
        }
        cin.get();
    }
    cout << "Bye!\n";
}

```

Обратим внимание на наиболее интересные моменты реализации.

В модуле HexString.cpp размещена глобальная функция IsHexStrVal, проверяющая, соответствует ли аргумент _str изображению шестнадцатеричного числа. Проверка выполняется с помощью метода find_first_of класса string, который возвращает индекс вхождения символа,

заданного аргументом, в строку `alph`. Последняя объявлена в файле `HexString.h` и содержит набор допустимых символов для изображения шестнадцатеричного числа. Если символ не найден, метод `find_first_of` возвращает значение `-1`. Функция `IsHexStrVal` используется в конструкторе класса `HexString`, предотвращая присваивание полю `val` некорректного значения, а также в методе `AddObject` класса `Factory`, блокируя ошибочный ввод информации пользователем. В базовом классе `Action` реализован метод `GetDecimal(AString* pObj)`, возвращающий значение десятичного числа для передаваемого через указатель `pObj` изображения шестнадцатеричного числа. Почему в базовом классе? — Потому, что это значение необходимо получать как в методе `ShowDec::Operate`, так и в методе `ShowBin::GetBinary`.

В методе `Action::GetDecimal` применена технология использования информации о типе на этапе выполнения, сокращенно *RTTI (Run-Time Type Information)*. Дело в том, что заранее (на этапе компиляции) неизвестно, объекты каких типов будут передаваться данному методу как аргумент. Возможно появление объекта любого производного класса иерархии `AString`. В то же время из условия задачи известно, что получение десятичного и двоичного представлений корректно только для шестнадцатеричных строк. Поэтому мы используем операцию динамического приведения типов `dynamic_cast`, выполняющую *понижающее преобразование* из типа базового класса к типу производного класса. Операция возвращает адрес объекта производного класса, указанного в угловых скобках: `<HexString*>`, если преобразование возможно (если `pObj` действительно является адресом объекта класса `HexString`), либо `0` в противном случае.

В методе `ShowBin::GetBinary` сначала вычисляется десятичное значение `decVal` для шестнадцатеричной строки, а затем в цикле `while` формируется символьный массив `binStr`, содержащий символы `'0'` и `'1'` для двоичного изображения числа. Очередной символ вычисляется извлечением младшего бита из `decVal` операцией `decVal % 2` и последующим его преобразованием к коду ASCII, для чего к значению бита (0 или 1) прибавляется 48. После извлечения двоичное представление `decVal` сдвигается на один разряд вправо.

В классе `Menu` метод `SelectItem` объявлен как статический. Дело в том, что подзадачу ввода целого числа для выбора пункта меню нужно решать не только в классе `Menu`, но и в классе `Factory`. Чтобы не дублировать код, мы сделали метод `SelectItem` статическим и теперь можем вызывать его из другого класса, разумеется, предваряя операцией доступа к области видимости `Menu::`.

В методе `Factory::AddObject` добавление нового объекта к коллекции `pObj` производится вызовом метода `push_back` класса `vector`.

В методе `Factory::DeleteObject` удаление объекта с заданным адресом осуществляется с помощью метода `erase` класса `vector`. Так как значение `item` на

единицу больше, чем индекс элемента в контейнере `vector`, то адрес удаляемого объекта вычисляется выражением `pObj.begin() + item - 1`, где `begin` — метод, возвращающий адрес начального элемента в контейнере.

Итоги

1. Наследование и полиморфизм — важнейшие механизмы ООП. Наследование позволяет объединять классы в семейства связанных типов, что дает им возможность совместно использовать общие поля и методы.

2. В языке C++ родительский класс называется *базовым*, а дочерний класс — *производным*. Отношения между родительскими классами и их потомками называются *иерархией наследования*.

3. Полиморфное использование объектов из одной иерархии базируется на двух механизмах: а) возможности использования указателя или ссылки на базовый класс для работы с объектами любого из производных классов, б) технологии *динамического связывания* при выборе виртуального метода через указатель на базовый класс. Фактический выбор операции, которая будет вызвана, происходит во время выполнения программы в зависимости от типа выбранного объекта.

4. Наряду с отношением наследования, классы могут находиться и в других отношениях: *ассоциации*, *агрегации*, *композиции* и *зависимости* (использования). Удобным средством отображения взаимоотношений классов является *диаграмма классов* на языке UML.

5. Современное программирование базируется не только на идеях ООП, но и на применении паттернов проектирования, аккумулирующих наиболее удачные решения типичных проблем, неоднократно возникавших при разработке ПО.

Задания

Вариант 1 - 20

1. При выполнении задания необходимо построить UML-диаграмму классов программного модуля.
2. Тестирование и отладка:
 - 2.1. Внимательно прочитать условие задачи.
 - 2.2. Убедиться, что программа корректно работает на примерах.
 - 2.3. Составить план тестирования (проанализировать классы входных данных).
 - 2.4. Тестировать программу.
 - 2.5. Выполнить декомпозицию программы на отдельные блоки и покрыть каждый из них юнит-тестами

Примечание:

Условие задачи сформулировать внутри файла `readme.md`. С соответствующим форматированием. Файл поместить в папку проекта.

Диаграммы прикрепить в качестве отдельного файла в папку проекта.

Отладку решения задачи с помощью юнит-тестов, проводить в файле проекта (после отладки юнит-тесты закомментировать). При необходимости применять ввод входных данных из текстового файла .

Общая часть заданий для вариантов 1–20

Написать программу, демонстрирующую работу с объектами двух типов, t_1 и t_2 , для чего создать систему соответствующих классов. Каждый объект должен иметь идентификатор (в виде произвольной строки символов) и одно или несколько полей для хранения состояния (текущего значения) объекта.

Клиенту (функции `main`) должны быть доступны следующие основные операции (методы): создать объект, удалить объект, показать значение объекта и прочие дополнительные операции (зависят от варианта). Операции по созданию и удалению объектов инкапсулировать в классе `Factory`. Предусмотреть меню, позволяющее продемонстрировать заданные операции.

При необходимости в разрабатываемые классы добавляются дополнительные методы (например, конструктор копирования, операция присваивания и т.п.) для обеспечения надлежащего функционирования этих классов.

Варианты 1–10

В табл. 2.2 и 2.3 перечислены возможные типы объектов и возможные дополнительные операции над ними. Рассматриваются только целые положительные числа.

Таблица 2.2. Перечень типов объектов

Класс	Объект
<code>SymbString</code>	Символьная строка (произвольная строка символов)
<code>BinString</code>	Двоичная строка (изображение двоичного числа)
<code>OctString</code>	Восьмеричная строка (изображение восьмеричного числа)
<code>DecString</code>	Десятичная строка (изображение десятичного числа)
<code>HexString</code>	Шестнадцатеричная строка (изображение шестнадцатеричного числа)

Таблица 2.3. Перечень дополнительных операций (методов)

Операция (метод)	Описание
<code>ShowBin()</code>	Показать изображение двоичного значения объекта
<code>ShowOct()</code>	Показать изображение восьмеричного значения объекта
<code>ShowDec()</code>	Показать изображение десятичного значения объекта
<code>ShowHex()</code>	Показать изображение шестнадцатеричного значения объекта

operator +(T& s1, T& s2) ⁶	для объектов SymbString — конкатенация строк s1 и s2; для объектов прочих классов — сложение соответствующих численных значений с последующим преобразованием к типу T
operator -(T& s1, T& s2)	для объектов SymbString — если s2 содержится как подстрока в s1, результатом является строка, полученная из s1 удалением подстроки s2; в противном случае возвращается значение s1; для объектов прочих классов — вычитание соответствующих численных значений с последующим преобразованием к типу T

Примечание. Первые четыре операции могут применяться к объектам любых классов, за исключением класса SymbString.

Таблица 2.4 содержит спецификации вариантов 1–10.

Таблица 2.4. Спецификации вариантов 1–10

Вариант	T1	T2	Операции (методы)
1	SymbString	BinString	ShowOct(), ShowDec(),ShowHex()
2	SymbString	BinString	operator +(T&, T&)
3	SymbString	BinString	operator -(T&, T&)
4	SymbString	OctString	operator +(T&, T&)
5	SymbString	OctString	operator -(T&, T&)
6	SymbString	DecString	ShowBin(), ShowOct(), ShowHex()
7	SymbString	DecString	operator +(T&, T&)
8	SymbString	DecString	operator -(T&, T&)
9	SymbString	HexString	operator +(T&, T&)
10	SymbString	HexString	operator -(T&, T&)

Варианты 11–20

В табл. 2.5 и 2.6 перечислены возможные типы объектов и дополнительные операции над ними. Таблица 2.7 содержит спецификации вариантов 11–20.

Таблица 2.5. Перечень типов объектов

Класс	Объект	Класс	Объект
Triangle	Треугольник	Tetragon	Четырехугольник
Quadrate	Квадрат	Pentagon	Пятиугольник
Rectangle	Прямоугольник		

Таблица 2.6. Перечень дополнительных операций (методов)

Операция (метод)	Описание
------------------	----------

⁶ Здесь T — любой из типов, T1 или T2.

Move()	Переместить объект на плоскости
Compare(T& ob1, T& ob2)	Сравнить объекты ob1 и ob2 по площади
IsIntersect(T& ob1, T& ob2)	Определить факт пересечения объектов ob1 и ob2 (есть пересечение или нет)
IsInclude(T& ob1, T& ob2)	Определить факт включения объекта ob2 в ob1

Таблица 2.7. Спецификации вариантов 11–20

Вариант	T1	T2	Операции (методы)
11	Triangle	Quadrate	Move(), Compare(T&, T&)
12	Quadrate	Pentagon	Move(), IsIntersect(T&, T&)
13	Triangle	Rectangle	Move(), Compare(T&, T&)
14	Triangle	Rectangle	Move(), IsIntersect(T&, T&)
15	Rectangle	Pentagon	Move(), IsInclude(T&, T&)
16	Triangle	Tetragon	Move(), Compare(T&, T&)
17	Triangle	Tetragon	Move(), IsIntersect(T&, T&)
18	Triangle	Tetragon	Move(), IsInclude(T&, T&)
19	Triangle	Pentagon	Move(), Compare(T&, T&)
20	Triangle	Pentagon	Move(), IsIntersect(T&, T&)

Лабораторная работа №3

Шаблоны классов. Обработка исключительных ситуаций

Шаблоны классов

Шаблоны классов наряду с шаблонами функций поддерживают парадигму обобщенного программирования, то есть программирования с использованием типов в качестве параметров. Механизм шаблонов в C++ допускает применение абстрактного типа в качестве параметра при определении класса или функции. После того как шаблон класса определен, он может использоваться для определения конкретных классов. Процесс генерации компилятором определения конкретного класса по шаблону и его аргументам называется инстанцированием шаблона (template instantiation).

Рассмотрим, например, точку на плоскости. Для ее представления в задаче 3.2 мы разработали класс Point, в котором положение точки задавалось координатами `x` и `y` — полями типа `double`. Представим, что в другом приложении требуется задавать точки для целочисленной системы координат, то есть использовать поля типа `int`. Можно вообразить себе системы, в которых координаты точки имеют тип `short` или `unsigned char`. Так что же — определять для каждой из этих задач новый класс Point? Бьерна Страуструпа очень раздражала такая перспектива, и он добавил в C++ поддержку того, чем мы будем заниматься на этом семинаре.

Определение шаблона класса

Определение шаблонного (обобщенного, родового) класса имеет вид `template <параметры_шаблона> class имя_класса { /* ... */ };`

Например, определение шаблонного класса Point будет выглядеть так:

```
template <class T> class Point {
public:
    Point( T _x = 0, T _y = 0 ) : x( _x ), y( _y ) {}
    void Show() const { cout << "    (" << x << ", " << y << ")" <<
endl; };
private:
    T x, y;
};
```

Вы заметили, чем отличается это определение от определения обычного класса? Префикс `template <class T>` указывает, что объявлен шаблон класса, в котором `T` — некоторый абстрактный тип. То есть ключевое слово `class` в этом контексте задает вовсе не класс, а означает лишь то, что `T` — это параметр шаблона. Вместо `T` может использоваться любое имя. После объявления `T` используется внутри шаблона точно так же, как имена других типов. Язык позволяет вместо ключевого слова `class` перед параметром шаблона использовать другое — `typename`:

```
template < typename T> class Point { /* ... */ };
```

В литературе встречаются оба стиля, но первый более распространен, так как ключевое слово `typename` появилось в языке C++ сравнительно недавно.

Определение встроенных методов внутри шаблона класса практически не отличается от записи в обычном классе. Но если определение метода выносится за пределы класса, синтаксис его заголовка усложняется. Покажем это на примере метода

Show:

```
template <class T> class Point {      //Версия с внешним определением
                                     метода Show public:
    Point( T _x = 0, T _y = 0 ) : x( _x ), y( _y ) {}
    void Show() const;
private:
    T x, y;
};
template <class T> void Point<T>::Show() const {
    cout << "    (" << x << ", " << y << ")" << endl;
}
```

Обратите внимание на появление того же префикса `template <class T>`, который предварял объявление шаблона класса, а также на более сложную запись операции квалификации области видимости для имени Show: если раньше мы писали `Point::`, то теперь пишем `Point<T>::`, добавляя к имени класса список параметров шаблона в угловых скобках (в данном случае — один параметр `T`). На первых порах это сложно запомнить (что является причиной постоянных ошибок компиляции), но, написав пару десятков шаблонных классов, вы привыкнете...

Использование шаблона класса

При включении шаблона класса в программу никакие классы на самом деле не генерируются до тех пор, пока не будет создан экземпляр шаблонного класса, в котором вместо абстрактного типа `T` указывается некоторый конкретный тип. Такая подстановка приводит к *актуализации*, или *инстанцированию*, шаблона. Как и для обычного класса, экземпляр создается либо объявлением объекта, например:

```
Point<int> aPoint( 13, -5 );
```

либо объявлением указателя на актуализированный шаблонный тип с присваиванием ему адреса, возвращаемого операцией `new`, например:

```
Point<double>* pPoint = new Point<double>( 9.99, 3.33 );
```

Встретив такие объявления, компилятор генерирует код соответствующего класса.

Организация исходного кода

В многофайловом проекте определение шаблона класса обычно выносится в отдельный файл. В то же время для инстанцирования компилятором конкретного экземпляра шаблона класса необходимо, чтобы определение шаблона находилось в *одной единице трансляции* с данным экземпляром.

В связи с этим принято размещать *все определение* шаблонного класса в некотором заголовочном файле, например `Point.h`, и подключать его к нужным файлам с помощью директивы `#include`. Для предотвращения повторного

включения этого файла, которое может возникнуть в многофайловом проекте, обязательно используйте «стражи включения».

Продemonстрируем эту технику на примере нашего класса Point (листинг 3.1).

Листинг 3.1. Шаботонный класс Point

```
//////////////////////////////////// Point.h //////////////////////////////////
#ifndef POINT_H
#define POINT_H
using namespace std;
template <class T> class Point {
public:
    Point(T _x = 0, T _y = 0) : x(_x), y(_y) {}
    void Show() const;
private:
    T x, y;
};
template <class T> void Point<T>::Show() const {
    cout << "  (" << x << ", " << y << ")" << endl;
}
#endif /* POINT_H */
//////////////////////////////////// Main.cpp //////////////////////////////////
#include <iostream>
#include "Point.h"
using namespace std;
int main() {
    Point<double> p1;                // 1
    Point<double> p2(7.32, -2.6);    // 2
    p1.Show(); p2.Show();
    Point<int> p3(13, 15);           // 3
    Point<short> p4(17, 21);         // 4
    p3.Show(); p4.Show();
}
```

Обратите внимание на использование шаботонного класса Point клиентом main: в *строках 1 и 2* шаботон инстанцируется в конкретный класс Point с подстановкой вместо T типа double, в *строке 3* — в класс Point с подстановкой типа int, в *строке 4* — с подстановкой типа short.

Заметим, что наиболее широкое применение шаботоны классов нашли при создании контейнерных классов стандартной библиотеки шаботонов (STL), предназначенных для работы с такими стандартными структурами, как вектор, список, очередь, множество и т. д. Кстати, на семинаре 11 мы уже воспользовались одним из этих классов, а именно классом vector. Так что сейчас вы можете бросить беглый ретроспективный взгляд на пройденный материал, испытывая чувство глубокого удовлетворения от более тонкого понимания новой материи — шаботонных классов.

Параметры шаботонов

Параметрами шаботонов могут быть абстрактные типы или переменные встроенных типов, таких как int. Стандарт C++ допускает также использование

параметров шаблонов, но это реализовано далеко не во всех компиляторах. *Абстрактные типы* в качестве параметров мы уже рассмотрели: при инстанцировании на их место подставляются аргументы либо встроенных, либо описанных программистом типов.

Переменные встроенных типов используются как параметры шаблонов, когда для шаблона предусматривается его настройка некоторой константой. Например, можно создать шаблон для массивов, содержащих n элементов типа T :

```
template <class T, int n> class Array { /*...*/ };
Array<Point, 20> ap;           // Создание массива из 20 элементов типа
Point
```

Приведем менее тривиальный пример использования параметров второго вида:

```
void f1() { cout << "I am f1()." << endl; }           // Директивы
// препроцессора для
void f2() { cout << "I am f2()." << endl; }           // краткости
// опущены
template<void (*pf)()> struct A { void Show() { pf(); } };
int main() {
    A<&f1> aa;
    aa.Show();    // вывод: I am f1().
    A<&f2> ab;
    ab.Show();    // вывод: I am f2().
}
```

Здесь параметр шаблона имеет тип указателя на функцию. При инстанцировании класса в качестве аргумента подставляется адрес соответствующей функции (адрес функции также является константой, создаваемой компилятором).

Естественно, у шаблона может быть несколько параметров. Например, ассоциативный контейнер `map` из библиотеки `STL` имеет следующее определение:

```
template <class Key, class T, class Compare = less<Key> > class map
{ /*...*/ };
```

Из последнего примера видно, что параметры шаблона так же, как и параметры обычных функций, могут иметь значения по умолчанию. Обратите внимание на наличие пробела между двумя последними символами `>`. Если не поставить пробел, компилятор воспримет последовательность `>>` как операцию сдвига вправо и выдаст сообщение об ошибке, текст которого слабо проясняет ее истинную причину.

Специализация

Иногда возникает необходимость определить специализированную версию шаблона для некоторого конкретного типа одного из его параметров. Например, невозможно создать обобщенный алгоритм, проверяющий отношение `<` (меньше) для двух аргументов типа `T`, который одновременно подходил бы и для числовых типов, и для традиционных Строк, завершающихся нулевым байтом. В таких случаях применяется специализация шаблона,

имеющая следующую общую форму: `template <> class имя_класса
<имя_специализированного_типа> { /* ... */ };`

Например:

```
template <class T> class Sample {           // Общий шаблон
    bool Less(T) const; /*...*/ };
template <> class Sample<char*> {           // Специализация для char*
    bool Less(char*) const; /*...*/ };
```

Если у класса-шаблона несколько параметров, возможна *частичная специализация*:

```
template <class T1, class T2> class Pair { /*...*/ }; // Общий шаблон
// специализация, где для T2 установлен тип int:
template <class T1> class Pair <T1, int> { /*...*/ };
```

В любом случае общий шаблон должен быть объявлен прежде любой специализации.

Иногда есть смысл специализировать не весь класс, а какой-либо из его методов:

```
// обобщенный метод:
template <class T> bool Sample<T>::Less(T ob) const { /*...*/ };
// специализированный метод:
void Sample<char*>::Less(char* ob) const { /*...*/ };
```

Использование функциональных объектов для настройки шаблонных классов

Напомним, что *функциональным объектом* называется объект, для которого определена операция вызова функции `operator()`. Соответственно, класс, экземпляром которого является этот объект, называется *классом функционального объекта* или просто *функциональным классом*. Приведем пример (листинг 3.2).

Листинг 3.2. Использование функционального класса

```
#include <iostream>
using namespace std;
struct LessThan { // функциональный класс с единственной операцией
    operator()
        bool operator() ( const int x, const int y ) { return x < y; }
};
int main() {
    LessThan lt // объявлен объект lt функционального класса LessThan
    int a = 5, b = 9;
    if ( lt( a, b ) ) cout << a << " less than " << b << endl; // 2
    if ( LessThan()( a, b ) ) cout << a << " less than " << b <<
    endl; // 3
}
```

Как вы помните, `struct` — это вид класса, в котором все элементы по умолчанию открыты. Объект `lt` используется в операторе `if` (*оператор 2*) для вызова функции `operator()`, передавая ей аргументы `a` и `b`.

В *операторе 3* демонстрируется способ использования функционального объекта без его предварительного объявления. Запись `LessThan()` означает создание анонимного экземпляра класса `LessThan`, то есть функционального

объекта. Запись `LessThan()` (`a`, `b`) означает вызов функции `operator()` с передачей ей аргументов `a` и `b`. Результаты выполнения обоих операторов `if` одинаковы.

Возможно, вы удивитесь, какой смысл использовать здесь класс `LessThan`, вместо того чтобы просто написать `if (a < b)`, и будете абсолютно правы. Но ситуация резко меняется, когда один из параметров шаблонного класса используется для настройки класса на некоторую стратегию.

Рассмотрим пример. Вас вызвал шеф и дал задание создать шаблонный класс `PairSelect`, предназначенный для выбора одного значения из пары значений, хранящихся в этом классе. Выбор выполняется в соответствии с критерием (стратегией), который передается как параметр шаблона. Параметр должен называться `class Compare`. Для начала нужно реализовать две стратегии: `LessThan` (первое значение меньше второго) и `GreaterThan` (первое значение больше второго).

Анализируя постановку задачи, вы приходите к заключению, что реализация стратегий в виде глобальных функций `LessThan()` и `GreaterThan()` быстро заведет в тупик, так как компилятор не позволит передать адреса функций на место параметра `class Compare`. Остается единственное решение — использовать функциональные классы (листинг 3.3).

Листинг 3.3. Шаблонный класс `PairSelect`

```
#include <iostream>
using namespace std;
template <class T> struct LessThan {
    bool operator() ( const T& x, const T& y ) { return x < y; }
};
template <class T> struct GreaterThan {
    bool operator() ( const T& x, const T& y ) { return x > y; }
};
template <class T, class Compare>
class PairSelect {
public:
    PairSelect( const T& x, const T& y ) : a( x ), b( y ) {}
    void OutSelect() const {
        cout << ( Compare()( a, b ) ? a : b ) << endl;
    }
private:
    T a, b;
};

int main() {
    PairSelect<int, LessThan<int> > ps1( 13, 9 );
    ps1.OutSelect(); // вывод: 9
    PairSelect<double, GreaterThan<double> > ps2( 13.8, 9.2 );
    ps2.OutSelect(); // вывод: 13.8
}
```

Обратите внимание на использование функционального объекта `Compare()(a, b)` в методе `OutSelect`, а также на настройку шаблонного класса `PairSelect` при его инстанцировании. При первом инстанцировании (объявление

объекта ps1) второму аргументу класса PairSelect передается функциональный класс LessThan<int>. При втором инстанцировании (объявление ps2) передается GreaterThan<double>.

Разработка шаблонного класса для представления разреженных массивов

Для закрепления материала разработаем шаблонный класс, предназначенный для представления *разреженных массивов* — массивов, в которых не все элементы фактически используются, присутствуют на своих местах или нужны.

Эта структура данных появилась в процессе решения научных и инженерных задач, в которых нужны многомерные массивы очень большого объема, но в то же время на каждом этапе обработки информации фактически используется только незначительная часть элементов массива. Например, это приложения, связанные с анализом матриц, или *электронные таблицы*, использующие матрицу для хранения формул, значений и строк, ассоциируемых с каждой из ячеек. Благодаря использованию разреженных массивов память для хранения каждого из элементов выделяется из пула свободной памяти только по мере надобности.

Для простоты изложения будем рассматривать одномерные разреженные массивы, так как все обсуждаемые идеи несложно применить и для реализации многомерных массивов. В среде приличных людей, так или иначе связанных с разреженными массивами, в ходу два термина: «*логический массив*» и «*физический массив*». Логический массив является воображаемым, а физический массив — это массив, фактически существующий в системе. Например, если вы определите разреженный массив:

```
SparseArr sa(1000000);
```

логический массив будет состоять из 1 000 000 элементов даже в случае, если этот массив в системе физически не существует. Если же фактически используется 100 элементов массива, то только они и будут занимать физическую память компьютера.

Обычно каждый элемент физического разреженного массива содержит как минимум два поля: логический индекс элемента и его значение. Для хранения физического массива, как правило, используют одну из динамических структур данных.

Задача 3.1. Шаблонный класс для разреженных массивов

Разработать шаблонный класс для представления разреженных одномерных массивов. Размер логического массива передавать через аргумент конструктора.

Класс должен обеспечивать хранение данных любого типа T, для которого предусмотрены конструктор по умолчанию, конструктор копирования и операция присваивания. Класс должен содержать операцию индексирования, возвращающую ссылку на найденный элемент. Если элемент с

заданным индексом не найден, операция должна создать новый элемент с этим индексом и поместить его в массив. При необходимости добавить в класс другие методы. В клиенте `main` продемонстрировать использование этого класса.

Заметим, что согласно условию задачи нужно разработать очень примитивный класс с минимальной функциональностью. В реальных приложениях такой класс будет, конечно, содержать и другие методы, например, удаление из физического массива элемента с заданным индексом.

Выше было сказано, что каждый элемент физического массива должен содержать два поля: логический индекс элемента и его значение. Поэтому начнем с разработки серверного класса для представления одного элемента физического массива. Этот класс также должен быть шаблонным, иначе как же он же будет использоваться клиентом — шаблонным же классом разреженного массива? Результатом проработки этого вопроса может быть, например, следующий класс:

```
template <class DataT> class SA_item {
public:
    SA_item( long i, DataT d ) : index( i ), info( d ) {}
    long index;
    DataT info;
};
```

Второй вопрос, который нужно решить до начала кодирования, — какую структуру данных мы выберем для хранения физического массива и какими средствами ее реализуем? Чаще всего используются линейные списки, бинарные деревья или структуры данных с хешированием индексов.

Линейный список имеет наихудшие показатели по времени поиска информации с заданным ключом (индексом), что может иметь существенное значение, если количество элементов в физическом массиве достаточно велико, но в то же время он наиболее прост для программирования. Более того, чтобы не отвлекаться на детали реализации и написать компактный код, мы воспользуемся контейнерным классом `list` из STL. Вообще-то контейнерным классам STL будет посвящен семинар 15, но у нас уже есть опыт использования класса `vector` на семинаре 11; так же и здесь, приведя минимально необходимые сведения о классе `list`, мы сможем воспользоваться многими его преимуществами.

Контейнерный класс `list` является шаблонным классом и реализован в STL в виде двусвязного списка, каждый узел которого содержит ссылки на последующий и предыдущий элементы. Для использования класса необходимо подключить заголовочный файл `<list>`. В классе есть конструктор по умолчанию, создающий список нулевой длины. Можно добавить в конец имеющегося списка новый элемент с помощью метода `push_back`. Доступ к любому элементу списка осуществляется через *итератор* — переменную типа

`list<T>::iterator`. Поскольку с понятием итератора вы еще не знакомы, скажем о нем несколько слов.

Проще всего рассматривать итератор как указатель на элемент списка. Он используется для просмотра списка в прямом или обратном направлении. В первом случае к итератору применяется операция инкремента, во втором — декремента.

В классе `list` есть два метода, позволяющие организовать просмотр всех элементов списка: `begin` возвращает указатель на первый элемент, `end` возвращает указатель на элемент, следующий за последним. Текущее значение итератора в цикле сравнивается со значением, полученным от метода `end`, с помощью операции `!=`, так как из-за произвольного размещения в памяти соседних элементов списка операция `<` для адресов элементов теряет смысл. Поясним применение класса `list` на примере:

```
#include <iostream>
#include <list>
using namespace std;
int main() {
    list<char> vl;
    vl.push_back( 'A' );
    vl.push_back( 'B' );
    vl.push_back( 'C' );
    list<char>::iterator i = vl.begin();
    list<char>::iterator n = vl.end();
    for ( ; i != n; ++i )
        cout << *i << ' '; // содержимое ячейки памяти, на которую
                             // указывает i
    cout << endl;
}
```

Мы можем позволить себе не комментировать этот пример, так как уровень ваших знаний на текущий момент, безусловно, достаточен, чтобы понять все с полувзгляда. Теперь все готово, чтобы привести возможное решение задачи (листинг 3.4).

Листинг 3.4. Шаблонный класс для разреженных массивов

```
//////////////////////////////////// SparseArr.h //////////////////////////////////
#ifndef SPARSE_ARR_H
#define SPARSE_ARR_H
#include <list>
using namespace std;
template <class DataT> class SA_item {
public:
    SA_item() : index( -1 ), info( DataT() ) {}
    SA_item( long i, DataT d ) : index( i ), info( d ) {}
    long index;
    DataT info;
};
template <class T> class SparseArr {
public:
```

```

        SparseArr( long len ) : length( len ) {}
        T& operator [] ( long ind );
        void Show( const char* );
private:
        std::list<SA_item<T> > arr;                                // 1
        long length;
};
template <class T>
void SparseArr<T>::Show( const char* title ) {
    cout << "==== " << title << "====\n";
    typename list<SA_item<T> >::iterator i = arr.begin();
    typename list<SA_item<T> >::iterator n = arr.end();
    for ( ; i != n; ++i )
        cout << i->index << "\t" << i->info << endl;
}
template <class T>
T& SparseArr<T>::operator[] ( long ind ) {
    if ( ( ind < 0 ) || ( ind > length-1 ) ) {
        cerr << "Error of index: " << ind << endl;
        static SA_item<T> temp;
        return temp.info;
    }
    typename list<SA_item<T> >::iterator i = arr.begin();
    typename list<SA_item<T> >::iterator n = arr.end();
    for ( ; i != n; ++i )
        if ( ind == i->index ) return i->info;    // элемент найден
        // Элемент не найден, создаем новый элемент
arr.push_back( SA_item<T>( ind, T() ) );        // 2
    i = arr.end();
    return (--i)->info;
}
#endif    /* SPARSE_ARR_H */
//////////////////////////////////// Main.cpp //////////////////////////////////
#include <iostream>
#include <string>
#include "SparseArr.h"
using namespace std;
int main() {
    SparseArr<double> sa1( 2000000 );                // 3
    sa1[127649] = 1.1;                              // 4
    sa1[38225] = 1.2;                                // 5
    sa1[2004056] = 1.3;                              // 6
    sa1[1999999] = 1.4;                              // 7
    sa1.Show("sa1");                                // 8
    cout << "sa1[38225] = " << sa1[38225] << endl; // 9
    sa1[38225] = sa1[93];                            // 10
    cout << "After the modification of sa1:\n";
    sa1.Show( "sa1" );
    SparseArr<string> sa2( 1000 );                    // 11
    sa2[333] = "Nick";
    sa2[222] = "Peter";
    sa2[444] = "Ann";

```

```

sa2.Show( "sa2" );
sa2[222] = sa2[555];
sa2.Show( "sa2" );
}

```

Обратите внимание на следующие моменты. Поле `arr` (*оператор 1*) в шаблонном классе `SparseArr`, объявленное как объект шаблонного класса `list`, актуализированного шаблонным же параметром `SA_item<T>`, предназначено для хранения физического массива. Аргумент `T` здесь совпадает с параметром шаблонного класса `SparseArr`.

В операции индексирования `operator[]()` прежде всего проверяется, не выходит ли значение индекса `ind` за допустимые границы. Если выходит, формируется сообщение об ошибке, выводимое в поток `cerr`¹, после чего нужно либо прервать выполнение программы, либо вернуть некоторое приемлемое значение. В данном случае выбран второй вариант: создается временный объект `temp` с вызовом конструктора по умолчанию `SA_item()` и возвращается ссылка на его поле `info`.

Теперь обратим наши взоры на конструктор по умолчанию в классе `SA_item` — в первоначальном варианте класса, рассмотренном выше, этого конструктора не было. Здесь же он добавлен именно для использования в нештатных ситуациях: конструктор создает клон объекта физического массива со значением индекса, равным `-1`. Этим гарантируется, что созданный элемент нигде и никогда не будет использован. Таким образом, после вывода сообщения об ошибке программа продолжит свое выполнение.

Вряд ли стоит считать такое решение идеальным: ведь память будет замусориваться неиспользуемыми объектами. С другой стороны, терминальное прерывание работы программы вызовом `exit` тоже не назовешь эстетичным решением. Короче говоря, перед нами — классическая ситуация, требующая генерации и последующей обработки исключения, но эту тему мы рассмотрим чуть ниже. А пока вернемся к анализу работы операции индексирования.

Если с индексом все в порядке, то далее в цикле `for` ищется элемент физического массива с заданным индексом. В случае успеха возвращается ссылка на поле `info` найденного элемента. В случае неуспеха создается новый элемент физического массива и добавляется к списку `arr` (*оператор 2*). При этом полю `info` присваивается значение, сформированное конструктором по умолчанию `T()`. Затем вызывается метод `arr.end`, возвращающий указатель (итератор) на элемент, следующий за последним элементом списка. Чтобы получить доступ к последнему элементу, применяется операция *префиксного*

¹ сегг — объект класса `ostream`, представляющий стандартное устройство для вывода сообщений об ошибках, который аналогично объекту `cout` направляет выводимые данные на терминал пользователя. Различие состоит в том, что вывод объекта `segг` не буферизируется и отображается немедленно.

декремента `--i`. Полученное значение итератора позволяет вернуть из функции ссылку на поле `info` нового элемента.

В класс добавлен метод `Show`, который просто выводит в поток `cout` перечень элементов (индексы и значения) физического массива.

В клиенте `main` показано два варианта инстанцирования класса `SparseArr`. В первом случае (*оператор 3*) создается логический массив `sal` из 2 000 000 элементов типа `double`. Поначалу он не содержит физических элементов. Выполнение *оператора 4* начнется с вызова операции `sal.operator[](127649)`. Так как элемента с указанным индексом в массиве нет, будет создан элемент, имеющий индекс 127649 и значение 0.0 (значение для типа `double` по умолчанию), и добавлен в конец массива. Операция индексирования вернет ссылку на поле `info` со значением 0.0, выполняемая следом операция присваивания изменит это значение на 1.1. Выполнение *операторов 5* и *7* аналогично, а вот при выполнении *оператора 6* будет обнаружена ошибка индексирования.

В *операторе 9* проверяется обращение к существующему элементу массива для его вывода в поток `cout`. Интересным является *оператор 10* — в результате его выполнения элемент `sal[38225]` получит значение 0. Если вы внимательно читали предыдущие пояснения, такой результат не вызовет у вас вопросов. В итоге вывод на экран первой части программы (работа с массивом `sal`) будет следующим:

```
Error of index: 2004056
===== sal =====
127649    1.1
38225     1.2
1999999   1.4
sal[38225] = 1.2
After the modification of sal:
===== sal =====
127649    1.1
38225     0
1999999   1.4
93        0
```

Во втором случае (*оператор 11*) создается логический массив `sa2` из 1000 элементов типа `string`. Действия по проверке его использования аналогичны предыдущим. Вывод на экран второй части программы будет следующим:

```
===== sa2 =====
333 Nick
222 Peter
444 Ann
===== sa2 =====
333 Nick
222
444 Ann
555
```

Обработка исключительных ситуаций

При решении предыдущей задачи мы столкнулись с проблемой — что делать методу класса (или, например, операции индексирования), если он обнаруживает некоторую ошибку, вызванную некорректным обращением клиента к методу (недопустимое значение индекса)? Инструментарий, которым мы пользовались до сих пор, начиная с семинара 7, позволял предпринять одно из следующих решений:

- прервать выполнение программы;
- вернуть значение, означающее «ошибка»;
- вывести сообщение об ошибке в поток `cerr` и вернуть вызывающей программе некоторое приемлемое значение, которое позволит ей продолжить работу.

Первый вариант решения не понравится пользователю программы, так как в самый неподходящий момент она будет «ломаться» без всякого уведомления о причине своей гибели. Второй вариант возможен лишь тогда, когда возвращаемое значение функции предназначено именно для кодирования статуса ее завершения. Так бывает далеко не всегда: например, в операции индексирования `operator[]()` класса `SparseArr` возвращаемое значение есть ссылка на поле `info` объекта. Ну и наконец, третий вариант решения тоже часто связан с труднорешаемыми вопросами: что есть «приемлемое значение» и почему программа продолжает свою работу независимо от обнаруженной ошибки?

С подобными проблемами часто сталкиваются разработчики промышленных библиотек классов. Автор библиотечного класса может обнаружить ошибки времени выполнения, но в общем случае не знает, как должен на них реагировать клиент. Для решения подобных проблем в C++ были введены средства генерации и обработки *исключений* (*exception*). Заметим, что такими средствами пользуются не только при разработке библиотек. Например, в процессе выполнения конструктора класса может возникнуть какая-то нештатная ситуация (скажем, нехватка памяти). Поскольку конструктор не имеет возвращаемого значения, единственным способом для него уведомить об этом клиента также является генерация исключения.

Определение исключений

Для того чтобы работать с исключениями, необходимо:

- выделить *контролируемый блок* — составной оператор, перед которым записано ключевое слово `try`:

```
try {  
    // фрагмент кода  
}
```

- предусмотреть генерацию одного или нескольких исключений операторами `throw` внутри блока `try` или внутри функций, вызываемых из этого блока;

- разместить сразу за блоком `try` один или несколько обработчиков `catch`. Оператор `throw`, предназначенный для *генерации исключения*, имеет вид `throw выражение;`. Тип выражения определяет тип порождаемого исключения. При генерации исключения выполнение текущего блока прекращается, происходит поиск соответствующего обработчика и передача ему управления.

Синтаксис *обработчиков исключений* напоминает определение функции с одним параметром и именем `catch`:

```
catch (/* ... */) {  
    // действия по обработке исключения  
}
```

Объявление параметра обработчика возможно в одной из трех форм:

```
catch (Type) {           // Форма 1: обработка исключения типа Type  
}  
catch (Type info) {      // Форма 2: обработка исключения типа Type  
                          // с использованием значения info  
}  
catch (...) {            // Форма 3: обработка исключений всех типов  
}
```

После обработки исключения управление передается первому оператору, находящемуся непосредственно за обработчиками. Туда же, минуя все обработчики, передается управление, если исключение в `try`-блоке не было сгенерировано.

Если обработчик не в состоянии полностью обработать ошибку, он может сгенерировать исключение повторно с помощью оператора `throw` без параметров. В этом случае предполагается наличие внешних объемлющих блоков, в которых может находиться другой обработчик для этого типа исключения.

Перехват исключений

Когда с помощью `throw` генерируется исключение, функции исполнительной библиотеки C++ выполняют следующие действия:

- создают копию параметра `throw` в виде статического объекта, который сохраняется до тех пор, пока исключение не будет обработано;
- в поисках подходящего обработчика *раскручивают стек* (об этом чуть ниже);
- передают объект и управление обработчику, имеющему параметр, совместимый по типу с этим объектом.

А что такое подходящий разработчик? Рассмотрим такой пример:

```
try {    throw E();  
} catch ( H ) {  
    // когда мы сюда попадем?
```

```
}
```

Обработчик будет вызван, если

- `н` и `е` — одного типа;
- `н` является открытым базовым классом для `е`;
- `н` и `е` — указатели либо ссылки и для них справедливо одно из

предыдущих утверждений.

Следующая маленькая программка демонстрирует перехват исключений:

```
class A {
public:
    A() { cout << "Constructor of A\n"; }
    ~A() { cout << "Destructor of A\n"; }
};
class Error {};
class ErrorOfA : public Error {};
void foo() {
    A a;
    throw 1;
    cout << "This message is never printed" << endl;
}
int main() {
    try {
        foo();
        throw ErrorOfA();
    }
    catch( int )          { cerr << "Catch of int\n"; }
    catch( ErrorOfA )     { cerr << "Catch of ErrorOfA\n"; }
    catch( Error )        { cerr << "Catch of Error\n"; }
}
```

Она выдаст следующий результат:

```
Constructor of A
Destructor of A
Catch of int
```

Первым генерируется исключение `throw 1` внутри функции `foo`. Исполнительная система создает копию объекта «целая константа 1» и начинает поиск подходящего обработчика. Поскольку внутри `foo` ничего подходящего нет, поиск переносится вовне — в клиент `main`. Но (важная деталь!) перед тем, как покинуть тело функции `foo`, система вызывает деструкторы всех ее локальных объектов! Это и есть *раскрутка стека*.

В функции `main` нужный обработчик обнаруживается и дает о себе знать сообщением «Catch of int». Таким образом, генерация второго исключения `throw ErrorOfA()` здесь невозможна.

Поменяйте местами строки в блоке `try` и запустите программу снова. Теперь исключение `throw ErrorOfA()` будет сгенерировано и вы получите результат:

```
Catch of ErrorOfA
```

Интересно, что класс `ErrorOfA` совершенно пустой. Но этого достаточно, чтобы использовать его экземпляр (в данном случае анонимный) для генерации исключения. А теперь закомментируйте строку с обработчиком `catch(ErrorOfA)`. Программа выдаст:

Catch of Error

Неперехваченные исключения

Если исключение сгенерировано, но не перехвачено, вызывается стандартная функция `std::terminate`. Она будет вызвана и в том случае, если механизм обработки исключения обнаружит, что стек разрушен, или если деструктор, вызванный во время раскрутки стека, пытается завершить свою работу при помощи исключения.

По умолчанию `terminate` вызывает функцию `abort`.

Чтобы увидеть, что происходит при вызове этих функций, скомпилируйте и выполните следующую программу:

```
int main() {  
    throw 5;    // произвольное значение  
    return 0;  
}
```

На нашем компьютере (Windows XP и Visual Studio 2005) на экран вываливается окно с сообщением об аварийном завершении программы:

Debug Error!.. This application has requested the Runtime to terminate it in an unusual way.

Вы можете заменить, если захотите, вызов функции `abort` вызовом своего обработчика. Это делается с помощью функции `set_terminate`. Например:

```
void SoftAbort() {  
    cerr << "Program is terminated." << endl;  
    exit( 1 );  
}  
int main() {  
    set_terminate( SoftAbort );  
    throw 5;  
    return 0;  
}
```

Проверьте, что завершение программы теперь действительно окажется более мягким.

Классы исключений. Иерархии исключений

Отметим, что, хотя язык позволяет генерировать исключения любого встроенного типа (как, например, типа `int` в рассмотренных выше примерах), в реальных программных системах такие исключения используются редко. Гораздо удобнее создавать специальные классы исключений (такие, как `ErrorOfA`) и использовать в операторе `throw` либо объекты этих классов, либо анонимные экземпляры (через вызов конструктора класса).

При необходимости в этих классах можно передавать через параметры конструктора и сохранять для последующей обработки любую информацию о состоянии программы в момент генерации исключения. Другое преимущество

этого подхода — возможность создания иерархии исключений. Например, исключения для математической библиотеки можно организовать следующим образом:

```
class MathError { /* ... */ };          // Базовый класс обработки ошибок
class Overflow : public MathError { /* ... */ }; // Класс ошибки
                                                    переполнения
class ZeroDivide : public MathError { /* ... */ }; // Класс ошибки
                                                    "деление на 0"
```

Если не полениться и добавить в базовый класс виртуальный метод `ErrProcess`, заместив его в производных классах версиями `ErrProcess`, ориентированными на обработку конкретного типа ошибки, это позволит задавать после блока `try` единственный обработчик `catch`, принимающий объект базового класса:

```
try {
    // генерация любых исключений MathError, Overflow, ZeroDivide, ...
}
catch (MathError& me) {
    me.ErrProcess()                // Обработка любого исключения
}
```

На этапе выполнения такой обработчик `catch` будет обрабатывать как исключения типа `MathError`, так и исключения любого производного типа, причем благодаря полиморфизму каждый раз будет вызываться версия метода `ErrProcess`, соответствующая именно данному типу исключения.

Организация исключений в виде иерархий исключительно важна, если ставится цель разработать легко модифицируемое программное обеспечение. Ведь при структуре кода, которую мы рассмотрели, добавление в систему нового вида исключения, входящего в существующую иерархию, вообще не потребует изменять написанные ранее фрагменты кода, предназначенные для перехвата и обработки исключений.

Спецификации исключений

В заголовке функции можно задать список типов исключений, которые она может прямо или косвенно порождать. Этот список приводится в конце заголовка и предваряется ключевым словом `throw`. Например: `void Func(int a) throw (Foo1, Foo2);`

Такое объявление означает, что `Func` может сгенерировать только исключения `Foo1`, `Foo2` и исключения, являющиеся производными от этих типов, но не другие. Заголовок является интерфейсом функции, поэтому такое объявление дает пользователям функции определенные гарантии. Это очень важно при использовании библиотечных функций, так как определения таких функций не всегда доступны.

Что произойдет, если функция вдруг нарушит взятые на себя обязательства и в ее недрах будет возбуждено исключение, не соответствующее списку спецификации исключений? Тогда система вызовет обработчик `std::unexpected`, который может попытаться сгенерировать свое исключение (это зависит от реализации), и если оно не будет противоречить спецификации, то

продолжится поиск подходящего обработчика. В противном случае вызывается `std::terminate`.

Если вас не устраивает поведение `unexpected` по умолчанию, вы можете установить собственную функцию, которую он будет вызывать. Для этого нужно воспользоваться функцией `set_unexpected`.

Если спецификация исключений задана в виде `throw()`, это означает, что функция вообще не генерирует исключений. Отсутствие спецификации исключений в заголовке функции, то есть запись заголовка в привычном нам виде, означает, что функция может сгенерировать любое исключение.

Если заголовок функции содержит спецификацию исключений, то каждое объявление этой функции (включая определение) должно иметь спецификацию исключений с точно таким же набором типов исключений. Виртуальная функция может быть замещена в производном классе функцией с не менее ограничительной спецификацией исключений, чем ее собственная.

Спецификация исключений не является частью типа функции, поэтому `typedef` не может ее содержать. Например:

```
// typedef int (*PF)() throw( A ); // ошибка!
```

Исключения в конструкторах

Исключения предоставляют единственную возможность передать информацию об обломе (срыве, крахе, фиаско), случившемся в процессе создания нового объекта. Рассмотрим пример кода (листинг 3.5), написанного, мягко говоря, некорректно, но именно это позволяет смоделировать ситуации, которые могут иметь место в реальной программе.

Листинг 3.5. Исключения в конструкторе

```
#include <iostream>
using namespace std;
class Vect {
public:
    Vect(char);
    ~Vect() { delete [] p; }
    int& operator [] (int i) { return p[i]; }
    void Print();
private:
    int* p;
    char size;
};
Vect::Vect(char n) : size(n) {
    p = new int[size]; // 1
    for (int i = 0; i < size; ++i) p[i] = int();
}
void Vect::Print() {
    for (int i = 0; i < size; ++i) cout << p[i] << " ";
```

```

    cout << endl;
} int main() {
    int x = -1;
    cout << "x = " << hex << x << endl;    // вывод в
                                              // шестнадцатеричном
                                              // формате

    Vect a(3);
    a[0] = 0;  a[1] = 1;  a[2] = 2;  a.Print();
    Vect a1(200);
    a1[10] = 5;  a1.Print();
}

```

В программе реализован класс `Vect`, предназначенный для создания и использования одномерных массивов типа `int` произвольного размера (размер передается через параметр конструктора класса).

Сначала несколько слов о возможной предыстории появления этого кода на свет. Руководитель проекта, для которого предназначен класс `Vect`, заявил, что размер массива никогда не превысит число 256. Проект (система) разрабатывается для «железа» с жуткими ограничениями на размер оперативной памяти. Программисты ведут борьбу за каждый байт! Человек, которому поручили реализацию класса `Vect`, решил хранить информацию о размере массива в поле `char size`. Ведь для типа `char` выделяется один байт, а диапазон возможных значений для восьмиразрядного двоичного числа составляет 0 ... 255.

В функции `main` созданный класс тестируется. Предварительно листинг 5 должен быть откомпилирован в отладочной конфигурации (Debug) проекта. Если запустить программу под отладчиком (клавиша F5), то программа выводит

```
x = ffffffff 0 1 2
```

после чего серьезно «ломается» — среда выполнения сообщает о недопустимом размере запрашиваемой памяти: 4 294 967 295 байт.

Выполнение по шагам показывает, что крах происходит при создании объекта `a1` (*оператор 2*). Если выполнить трассировку этого оператора, нажав клавишу F11, то можно достичь *оператора 1* в конструкторе `Vect` и увидеть, что значение `size`, задающее размер запрашиваемой памяти, равно -56.

Откуда такое странное значение? Обычно для начинающих программистов непросто выяснить причину... Наш герой, видимо, тоже был начинающим программистом. Более того, скорее всего, он невнимательно изучил Учебник [21], где на страницах

24–25 объясняется, что `char` — сокращенное обозначение типа `signed char`, в котором старший бит используется для представления знака числа. Поэтому диапазон представимых чисел для типа `char` составляет -128 ... +127.

Как же будет интерпретироваться в этом случае десятичное число 200? Преобразовав его в двоичный эквивалент, получаем 11001000. Вы видите, что старший разряд равен единице?.. Это означает, что компьютер воспримет его как -1001000.

Обратный перевод в десятичную систему будет несколько сложнее. Напомним, что отрицательные целые числа хранятся в памяти компьютера в *дополнительном коде*, который получается инверсией всех разрядов с последующим прибавлением единицы. Значит, для обратного перевода нужно сначала вычесть единицу, а потом инвертировать все разряды. В итоге получим число -56 .

Если теперь выполнить трассировку *оператора 1*, нажав клавишу F11, то отладчик остановится перед заголовком библиотечной реализации операции `new: void *__CRTDECL operator new[](size_t count)`

а значение аргумента `count` будет равно уже не -56 , а $4\,294\,967\,295$. Если перевести это десятичное число в шестнадцатеричный эквивалент, получим `ffff ffff`. Для типа `size_t`, эквивалентного типу `unsigned int`, это максимально возможное значение 32-разрядного двоичного числа. Для типа `signed int` это значение представляет -1 (см. результат вывода программы для переменной `x`).

Далее операция `new` завершается аварийно, будучи не в состоянии выделить $4\,294\,967\,295$ байт.

Итак, мы нашли ошибку в реализации класса. Исправить ошибку несложно — нужно все `char` заменить на `unsigned char` (проверьте это). Но не будем спешить. Дело в том, что мы имеем прекрасную модель ненормативного поведения операции `new` и для дальнейших экспериментов она нам еще пригодится.

Приведенный пример показал, что, если вызываемые в теле конструктора класса функции или операции могут завершиться аварийно, необходимо информировать об этом клиента (функцию, в теле которой создается объект класса). Использование механизма исключений дает элегантное решение проблемы. Изменим код конструктора класса на следующий:

```
Vect::Vect(char n) : size( n ) {
    try { p = new int[size]; }
    catch(...) {
        throw "ErrorVectConstr"; }
    for ( int i = 0; i < size; ++i ) p[i] = int();
}
```

Внесите также изменения в функцию `main`, использующую объекты класса `Vect`:

```
int main() {
    try {
        int x = -1;
        cout << "x = " << hex << x << endl;
        Vect a(3);
        a[0] = 0; a[1] = 1; a[2] = 2; a.Print();
        Vect a1(200);
        a1[10] = 5; a1.Print();
    }
    catch( const char* except_name ) { cout << "Error: " <<
except_name << endl; }
}
```

Для проверки работы новой версии программы необходимо откомпилировать листинг 3.5 в выпускной конфигурации (Release) проекта. Дело в том, что по непонятным причинам операция `new` в среде Visual Studio 2005 генерирует исключения только в выпускной конфигурации проекта.

Исключения в деструкторах

Если деструктор, вызванный во время раскрутки стека, попытается завершить свою работу при помощи исключения, система вызовет функцию `terminate`. На этапе отладки программы это допустимо, но в готовом продукте появление окон сообщений об ошибках должно быть сведено к минимуму.

Отсюда наиважнейшее требование к деструктору: *ни одно из исключений, которое могло бы появиться в процессе работы деструктора, не должно покинуть его пределы!* Чтобы выполнить это требование, придерживайтесь двух правил.

1. Никогда не генерируйте исключения в теле деструктора с помощью `throw`.

2. Если финальные действия в деструкторе связаны с вызовом других функций, относительно которых у вас нет гарантий отсутствия генерации исключений, инкапсулируйте эти действия в одном методе, например `Destroy`, и вызывайте его в блоке `try`:

```
T::Destroy() {  
    // код, который может генерировать исключения  
}  
T::~~T() { // деструктор  
    try { Destroy(); }  
    catch(...) { /* ... */ }  
}
```

Мы завершили рассмотрение вопросов, связанных с обработкой исключений. Рекомендуем вам доработать листинг 3.4, предусмотрев в реализации операции индексирования генерацию исключения для ошибочного значения индекса.

Теперь рассмотрим решение задачи, в которой требуется использовать и шаблонные классы, и обработку исключений.

Задача 3.2. Шаблонный класс векторов

Разработать шаблонный класс `Vect` для представления динамических одномерных массивов (векторов). Класс должен обеспечивать хранение данных любого типа `T`, для которого предусмотрены конструктор по умолчанию, конструктор копирования и операция присваивания. Класс должен содержать:

- конструктор по умолчанию, создающий вектор нулевого размера;
- конструктор, создающий вектор заданного размера;
- операцию индексирования, возвращающую ссылку на соответствующий элемент вектора;

- метод, добавляющий элемент в произвольную позицию вектора;
- % метод, добавляющий элемент в конец вектора; % метод, удаляющий элемент из конца вектора.

При необходимости добавить в класс другие методы. Предусмотреть генерацию и обработку исключений для возможных ошибочных ситуаций. В клиенте `main` продемонстрировать использование этого класса.

Одним из принципиальных вопросов при разработке контейнерного класса является вопрос реализации хранения элементов контейнера, или, другими словами, выбор подходящей структуры данных (массив, список, бинарное дерево и т. п.). В данном случае мы остановим наш выбор на динамическом массиве, размещаемом в памяти посредством операции `new`, поскольку это наиболее простое решение. После размещения адрес первого элемента вектора будет запоминаться в поле `T* first`, а адрес элемента, следующего за последним, — в поле `T* last`. С учетом того, что используемый в классе метод `size` возвращает количество элементов в векторе, размещение контейнера в памяти поясняется на рис. 3.1

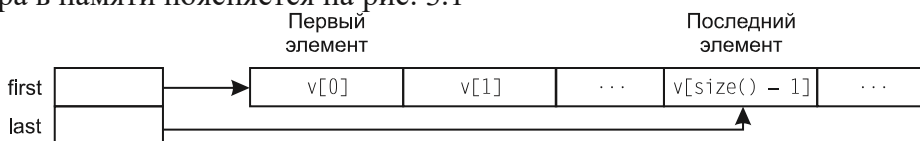


Рис. 3.1. Размещение вектора `v` в памяти

При решении этой задачи мы постараемся сделать класс `Vect` как можно более похожим (с точки зрения его интерфейса и, частично, реализации) на контейнерный класс `std::vector` из библиотеки STL (с учетом ограничений, вытекающих из постановки задачи). Поэтому в состав интерфейса будут включены методы:

- `void insert(T* _P, const T& _x)` — вставка элемента в позицию `_P`;
- `void push_back(const T& _x)` — вставка элемента в конец вектора;
- `void pop_back()` — удаление элемента из конца вектора;
- `T* begin()` — получение указателя на первый элемент;
- `T* end()` — получение указателя на элемент, следующий за последним;
- `size_t size()` — получение размера вектора (тип `std::size_t` является синонимом типа `unsigned int`).

Эти методы имитируют интерфейс класса `std::vector`. Уточним, что вместо типа `iterator`, имеющегося в `std::vector`, здесь используется указатель `T*`, но концептуально это одно и то же. Целью такого подражания является

психологическая подготовка наших читателей к более легкому восприятию контейнерных классов STL (семинар 15).

К двум конструкторам, требующимся по заданию, добавим конструктор копирования, чтобы обеспечить возможность передачи объектов класса в качестве аргументов для любой внешней функции.

По некоторым параметрам наш класс `Vect`, однако, будет превосходить класс `std::vector`. Для целей отладки и обучения мы хотим визуализировать работу таких методов класса, как конструктор копирования и деструктор, путем вывода на терминал соответствующих сообщений. Чтобы эти сообщения были более информативны, мы снабдим каждый объект класса так называемым отладочным именем (поле `markName`), которое позволит распознавать источник сообщения.

Трудно переоценить, какую пользу приносят эти сообщения начинающим программистам, помогая прочувствовать скрытые механизмы функционирования класса. Для более опытных читателей эти средства могут оказаться полезными при поиске неочевидных ошибок в программе.

Задание требует также *предусмотреть генерацию и обработку исключений для возможных ошибочных ситуаций. Наиболее очевидными являются: ошибка индексирования, когда клиент пытается получить доступ к несуществующему элементу вектора, и ошибка удаления несуществующего элемента из конца вектора — когда вектор пуст.* В принципе возможна еще одна нештатная ситуация, связанная с нехваткой памяти, когда операция `new` завершается аварийно. Но мы рассмотрели выше, как решать эту проблему, и сейчас ради краткости изложения будем полагать, что обладаем неограниченной памятью.

Для обработки исключительных ситуаций создадим иерархию классов во главе с базовым классом `VectError`. Производный класс `VectRangeErr` будет предназначен для обработки ошибок индексирования, а производный класс `VectPopErr` — для обработки ошибки удаления несуществующего элемента. В листинге 3.6 приводится код предлагаемого решения задачи.

Листинг 3.6. Шаблонный класс векторов

```
//////////////////////////////////// VectError.h //////////////////////////////////////
#ifndef _VECT_ERROR_
#define _VECT_ERROR_
#include <iostream>
#define DEBUG
class VectError {
public:
    VectError() {}
    virtual void ErrMsg() const { std::cerr << "Error with Vect
object.\n"; }
    void Continue() const {
#ifdef DEBUG
        std::cerr << "Debug: program is being continued.\n";
#else
```

```

        throw;
#endif
    }
};

class VectRangeErr : public VectError {
public:
    VectRangeErr(int _min, int _max, int _actual) :
        min(_min), max(_max), actual(_actual) {}
    void ErrMsg() const {
        std::cerr << "Error of index: ";
        std::cerr << "possible range: " << min << " - " << max << ",
";
        std::cerr << "actual index: " << actual << std::endl;
        Continue();
    }
private:
    int min, max;
    int actual;
};

class VectPopErr : public VectError {
public:
    void ErrMsg() const { std::cerr << "Error of pop\n";
Continue(); }
};
#endif /* _VECT_ERROR */
//////////////////////////////////// Vect.h //////////////////////////////////
#ifndef _VECT_
#define _VECT_
#include <iostream>
#include <string>
#include "VectError.h"
using namespace std;
template<class T> class Vect {          // -----
Template class Vect
public:
    explicit Vect() : first(0), last(0) {}
    explicit Vect(size_t _n, const T& _v = T()) {
        Allocate( _n );
        for (size_t i = 0; i < _n; ++i) *(first + i) = _v;
    }
    Vect( const Vect& );                  // конструктор копирования
    Vect& operator =(const Vect&);        // операция присваивания
    ~Vect() {
#ifdef DEBUG
        cout << "Destructor of " << markName << endl;
#endif
        Destroy();
        first = 0, last = 0;
    }
    void mark(std::string& name) { markName = name; } // установить
отладочное имя

```

```

    std::string mark() const { return markName; }    // получить
отладочное имя
    size_t size() const;                            // получить размер вектора
    T* begin() const { return first; }             // получить указатель на
1-й элемент
    T* end() const { return last; }                // получить указатель на
элемент,
                                                    // следующий за последним
    T& operator[](size_t i);                       // операция индексирования
    void insert(T* _P, const T& _x);               // вставка элемента в
позицию _P
    void push_back(const T& _x); // вставка элемента в конец вектора
    void pop_back();                               // удаление элемента из конца
вектора
    void show() const;                             // вывод в cout содержимого вектора
protected:
    void Allocate( size_t _n ) {
        first = new T [_n * sizeof(T)];
        last = first + _n;
    }
    void Destroy() {
        for ( T* p = first; p != last; ++p )    p->~T();
        delete [] first;
    }
    T* first;                                     // указатель на 1-й элемент
    T* last;                                     // указатель на элемент, следующий за последним
    std::string markName;
};
template<class T>
Vect<T>::Vect(const Vect& other) {    // -----
Конструктор копирования
    size_t n = other.size();
    Allocate( n );
    for (size_t i = 0; i < n; ++i) *(first + i) = *(other.first +
i);
    markName = string("Copy of ") + other.markName;
#ifdef DEBUG
    cout << "Copy constructor: " << markName << endl;
#endif
}
template<class T>
Vect<T>& Vect<T>::operator =(const Vect& other) { // --- Операция
присваивания
    if (this == &other) return *this;
    Destroy();
    size_t n = other.size();
    Allocate(n);
    for (size_t i = 0; i < n; ++i) *(first + i) = *(other.first +
i);
    return *this;
}
template<class T>

```

```

size_t Vect<T>::size() const { // -----Получение размера вектора
    if (first > last) throw VectError();    return (0 == first ? 0
: last - first);
}
template<class T>
T& Vect<T>::operator[](size_t i) { // Операция доступа по индексу
    if(i < 0 || i > ( size() - 1 ))
        throw VectRangeErr(0, last - first - 1, i);
    return ( *( first + i ) );
}
template<class T> // Вставка элемента со значением _x в позицию _P
void Vect<T>::insert(T* _P, const T& _x) {
    size_t n = size() + 1; // новый размер
    T* new_first = new T [n * sizeof(T)];
    T* new_last = new_first + n;    size_t offset = _P - first;
    for (size_t i = 0; i < offset; ++i) *(new_first + i) = *(first +
i);
    *( new_first + offset ) = _x;
    for (size_t i = offset; i < n; ++i) *(new_first + i + 1) =
*(first + i);
    Destroy();
    first = new_first;
    last = new_last;
}
template<class T>
void Vect<T>::push_back(const T& _x) { // ---- Вставка элемента в
конец вектора
    if (!size()) { Allocate(1); *first = _x; }
    else insert( end(), _x );
}
template<class T>
void Vect<T>::pop_back() { // --- Удаление элемента из конца вектора
    if(last == first) throw VectPopErr();
    T* p = end() - 1;
    p->~T();
    last--;
}
template<class T>
void Vect<T>::show() const { // --- Вывод в cout содержимого вектора
    cout << "\n===== Contents of " << markName << "===== " << endl;
    size_t n = size();
    for (size_t i = 0; i < n; ++i) cout << *(first + i) << " ";
    cout << endl;
}
#endif /* _VECT */
//////////_////////// Main.cpp //////////
#include "Vect.h"
#include <iostream>
#include <string>
using namespace std;
template<class T> void SomeFunction( Vect<T> v ) {
    std::cout << "Reversive output for " << v.mark() << ":" << endl;

```

```

    size_t n = v.size();
    for (int i = n - 1; i >= 0; --i) std::cout << v[i] << " ";
    std::cout << endl;
}
int main() {
    std::string vv1 = "v1";
    std::string vv2 = "v2";
    std::string vv3 = "v3";
    std::string vv4 = "v4";
    try {
        string initStr[5] = { "first", "second", "third", "fourth",
"fifth" };
        Vect<int> v1(10); v1.mark(vv1);
        size_t n = v1.size();
        for (int i = 0; i < n; ++i) v1[i] = i + 1;
        v1.show();
        SomeFunction(v1);
        try {
            Vect<string> v2(5);
            v2.mark(vv2);
            size_t n = v2.size();
            for (int i = 0; i < n; ++i) v2[i] = initStr[i];
            v2.show();
            v2.insert(v2.begin() + 3, "After_third");
            v2.show(); cout << v2[6] << endl;
            v2.push_back("Add_1"); v2.push_back("Add_2");
v2.push_back("Add_3");
            v2.show();
            v2.pop_back(); v2.pop_back();
            v2.show();
        }
        catch (VectError& vre) { vre.ErrMsg(); }
        try {
            Vect<int> v3;
v3.mark(vv3);
            v3.push_back(41); v3.push_back(42); v3.push_back(43);
            v3.show();
            Vect<int> v4;
            v4.mark(vv4);
            v4 = v3;
            v4.show();
            v3.pop_back(); v3.pop_back();
            v3.pop_back(); v3.pop_back();
            v3.show();
        }
        catch (VectError& vre) { vre.ErrMsg(); }
    }
    catch (...) { cerr << "Epilogue: error of Main().\n"; }
}

```

Наши комментарии будут следовать в порядке размещения текста модулей. **Модуль** VectError.h содержит иерархию классов исключений. В

базовом классе `VectError` определены два метода. Виртуальный метод `ErrMsg` обеспечивает вывод по умолчанию сообщения об ошибке (в производных классах этот метод замещается конкретными методами). Метод `Continue` определяет стратегию продолжения работы программы после обнаружения ошибки. Стратегия зависит от конфигурации программы. Информация о конфигурации кодируется наличием или отсутствием единственной директивы в начале файла: `#define DEBUG`.

Если лексема `DEBUG` определена, программа компилируется в *отладочной конфигурации*, если не определена — в *выпускной конфигурации*. Для метода `Continue` это означает, что

- в отладочной конфигурации выводится сообщение «Debug: program is being continued», после чего работа программы продолжается;
- в выпускной конфигурации повторно возбуждается (оператор `throw`) исключение, которое было первопричиной цепочки событий, завершившихся вызовом данного метода.

В производных классах `VectRangeErr` и `VectPopErr` метод `ErrMsg` переопределяется с учетом вывода информации о конкретной ошибке. После вывода вызывается метод `Continue` базового класса.

Модуль `Vect.h` содержит определение шаблонного класса `Vect`. Отметим наиболее интересные моменты.

Для управления выделением и освобождением ресурсов класс снабжен методами

`Allocate` и `Destroy`. Они размещены в защищенной части класса, так как относятся к его реализации. Напомним, что операция `new T [n]` (n — количество элементов, которое мы хотим поместить в массив) не только выделяет память, но и инициализирует элементы путем вызова `T()` — конструктора по умолчанию для типа `T`. Поэтому в методе `Destroy` сначала в цикле вызываются деструкторы `~T()` всех элементов массива, а потом операцией `delete` освобождается память.

Обратите внимание, что в заголовке цикла `for` условие его продолжения записано в виде `p != last`, а не `p < last`. Здесь мы следуем традиции STL. Подобный способ проверки обусловлен тем, что для произвольной структуры данных (например, для списка) соседние элементы не обязаны занимать подряд идущие ячейки памяти, а могут быть разбросаны в памяти как угодно.

Деструктор и конструктор копирования содержат вывод сообщений типа «Здесь был я!», дополненных печатью содержимого `markName`. Весь этот вывод осуществляется *только в отладочной конфигурации* программы. Операция присваивания (верная спутница конструктора копирования) реализована стандартным способом.

В операции доступа по индексу `operator[]()` служба внутренней охраны (оператор `if`) проверяет, не несет ли индекс с собой что-либо взрывоопасное (значение, лежащее вне пределов допустимого диапазона). Если индекс

оказывается неблагонадежным, служба бьет тревогу, вызывая исключение типа `VectRangeErr`. Вызов происходит через анонимный экземпляр класса с передачей конструктору трех аргументов: минимальной и максимальной границы и текущего значения индекса.

В методе получения размера вектора `size` предусмотрена проверка на лояльность взаимоотношений между `first` и `last`. Вообще-то непонятно, с каких это дел `first` может оказаться больше, чем `last`? Да, в нормально функционирующем классе это невозможно. Но в случае каких-либо ошибок или сбоев указанная проверка весьма полезна. При несоблюдении указанного условия генерируется исключение типа

`VectError`.

Метод `insert` (вставка элемента со значением `_x` в произвольную позицию `_p`) работает следующим образом. Создается новый вектор размером на единицу больше существующего вектора; адреса размещения нового вектора сохраняются в локальных переменных `new_first`, `new_last`. Затем в новый вектор копируется первая часть существующего вектора, начиная с первого элемента и заканчивая элементом, предшествующим элементу с адресом `_p`.

Поскольку доступ к элементам осуществляется через смещение относительно указателя `first`, перед копированием требуется вычислить `offset` — смещение, соответствующее адресу `_p`². Затем элементу с адресом `new_first + offset` присваивается значение `_x`. После этого оставшаяся часть существующего вектора копируется в новый вектор, начиная с позиции `new_first + offset + 1`. Вот и все. Осталось освободить ресурсы, занятые существующим вектором (`Destroy`), и назначить полям `first` и `last` новые значения.

Метод `push_back` (вставка элемента со значением `_x` в конец вектора) получился очень простым благодаря использованию серверной функции `insert`. Если вектор пуст, то выделяются ресурсы для хранения одного элемента и этому элементу присваивается значение `_x`. В противном случае вызывается `insert` для вставки элемента в позицию, определенную с помощью `end`.

В методе `pop_back` (удаление элемента из конца вектора) проверяется, не пуст ли вектор. В пустом векторе `first = last = 0`. Если это так, удалять нечего и, значит, клиент ошибся, затребовав такую операцию. Следовательно, надо ему об этом сообщить: генерируется исключение типа `VectPopErr`. В противном случае определяется указатель на последний элемент вектора, для него вызывается деструктор `~T()` и значение `last` уменьшается на единицу (учитывая арифметику указателей).

Модуль `Main.cpp` содержит определение глобальной шаблонной функции `SomeFunction` и определение функции `main`.

² Напомним, что разность двух указателей — это разность их значений, деленная на размер типа в байтах. Например, в применении к массивам разность указателей на второй и седьмой элементы равна 5.

Функция `SomeFunction` обеспечивает реверсивный вывод содержимого вектора в поток `cout`. Попутно она позволяет проверить передачу объектов конкретного класса `Vect` в качестве аргументов функции.

Функция `main` предназначена для тестирования класса `Vect`. Советуем внимательно изучить ее многослойную архитектуру: внешний блок `try` содержит в себе два внутренних блока `try`. И это не случайно! Первый внутренний блок `try` предназначен для тестирования операций с объектом `v2`, среди которых есть операция, вызывающая ошибку индексирования. Второй блок `try` работает с объектом `v3`, причем одна из операций вызывает ошибку удаления несуществующего элемента.

Обратите внимание, что обработчики `catch` в обоих случаях имеют параметром объект базового класса `VectError`, но благодаря полиморфизму они будут перехватывать и исключения производных классов `VectRangeErr` и `VectPopErr`, так что в результате будет вызываться конкретный метод `ErrMsg` для данного типа ошибки! Вспомним, что после вывода сообщения об ошибке метод `ErrMsg` вызывает метод `Continue`, работа которого зависит от конфигурации программы.

В отладочной конфигурации `Continue` сообщает о продолжении работы программы и возвращает управление клиенту. В выпускной конфигурации `Continue` возбуждает исключение повторно. Именно для того, чтобы его поймать, и служит внешний блок `try` с обработчиком `catch(...)` в основной программе.

Откомпилируйте и выполните программу, чтобы увидеть, как она работает. Исходный текст настроен на работу в отладочной конфигурации. Всмотритесь внимательно в отладочные сообщения конструктора копирования и деструктора. А теперь закомментируйте директиву `#define DEBUG` и выполните программу после компиляции заново. Почувствуйте разницу!

Итоги

1. Шаблоны классов поддерживают парадигму обобщенного программирования — программирования с использованием типов в качестве параметров. Определение конкретного класса по шаблону класса и аргументам шаблона называется инстанцированием (актуализацией) шаблона.

2. Параметрами шаблонов могут быть абстрактные типы или переменные встроенных типов.

3. Для настройки шаблонного класса на некоторую стратегию возможно использование классов функциональных объектов.

4. Обработка исключительных ситуаций позволяет разделить проблему обработки ошибок на две фазы: генерацию исключения в случае нарушения каких-то заданных условий и последующую обработку сгенерированного исключения.

5. Фазы генерации и обработки исключений обычно разнесены в программе по разным компонентам (модулям, функциям). Это дает

существенные преимущества, так как в месте возникновения ошибки (серверная функция) часто нет возможности корректно ее обработать, а клиентская функция такими возможностями обычно располагает.

6. Для конструктора класса использование исключений является единственным способом сообщить клиенту об ошибках или сбоях, случившихся в процессе конструирования объекта.

7. Для работы с исключениями необходимо: а) описать контролируемый блок; б) предусмотреть генерацию одного или нескольких исключений внутри блока или внутри функций, вызываемых из этого блока; в) разместить сразу за блоком один или несколько обработчиков исключений.

8. В качестве типов исключений, генерируемых оператором `throw`, целесообразно использовать определенные программистом классы исключений. Эти классы полезно выстраивать в иерархию, что позволяет создавать программы, для которых модификация в связи с обработкой новых типов ошибок оказывается менее трудоемкой.

9. Не генерируйте исключения в теле деструктора. Если это могут сделать другие вызываемые функции, примите все меры, чтобы исключение не покинуло деструктор.

Задания

Требуется создать шаблон некоторого целевого класса `A`. В каждом варианте уточняются требования к реализации — указанием на применение некоторого серверного класса `B`. Это означает, что объект класса `A` используется как элемент класса `B`. В качестве серверного класса может быть указан либо класс, созданный программистом в рамках того же задания, либо класс стандартной библиотеки.

Варианты целевых и серверных классов, создаваемых программистом, приведены в табл. 3.1. Информацию о работе с динамическими структурами данных см. в Учебнике (с. 114–127) и в семинаре 09.

Таблица 3.1. Варианты целевых или серверных классов

Имя класса	Назначение
<code>Vect</code>	Одномерный динамический массив
<code>List</code>	Двунаправленный список
<code>Stack</code>	Стек
<code>BinaryTree</code>	Бинарное дерево
<code>Queue</code>	Односторонняя очередь
<code>Deque</code>	Двусторонняя очередь (допускает вставку и удаление из обоих концов)
<code>Set</code>	Множество (повторяющиеся элементы в множество не заносятся; элементы в множестве хранятся отсортированными)
<code>SparseArray</code>	Разреженный массив

Если вместо серверного класса указан динамический массив, это означает, что для хранения элементов в целевом классе используется массив, размещаемый с помощью операции `new`. Во всех вариантах необходимо предусмотреть генерацию и обработку исключений для возможных ошибочных ситуаций.

Во всех вариантах показать в клиенте `main` использование созданного класса, включая ситуации, приводящие к генерации исключений. Показать инстанцирование шаблона для типов `int`, `double`, `std::string`. Варианты заданий приведены в табл. 3.2.

Таблица 3.2. Варианты заданий

Вариант	Целевой шаблонный класс	Реализация с применением
1	Vect	std::list
2	List	—
3	Stack	Динамический массив
4	Stack	Vect
5	Stack	List
6	Stack	std::vector
7	Stack	std::list
8	BinaryTree	—
9	Queue	Vect
10	Queue	List
11	Queue	std::list
12	Deque	Vect
13	Deque	List
14	Deque	std::list
15	Set	Динамический массив
16	Set	Vect
17	Set	List
18	Set	std::vector
19	Set	std::list
20	SparseArray	

Лабораторная работа №4. Стандартные потоки

Потоковые классы

Программа на C++ представляет ввод и вывод как поток байтов. При вводе она читает байты из потока ввода, при выводе вставляет байты в поток вывода. Понятие потока позволяет абстрагироваться от того, с каким устройством ввода-вывода идет работа. Например, байты потока ввода могут поступать либо с клавиатуры, либо из файла на диске, либо из другой программы. Аналогично, байты потока вывода могут выводиться на экран, в файл на диске или на вход другой программы.

Обычно ввод-вывод осуществляется через *буфер* — область оперативной памяти, накапливающую какое-то количество байтов, прежде чем они пересылаются по назначению. При обмене, например, с диском это значительно повышает скорость передачи информации. При вводе с клавиатуры буферизация обеспечивает другое удобство для пользователя — возможность исправления ошибок набора символов, пока они не отправлены из буфера в программу.

Буфер ввода обычно очищается при нажатии клавиши Enter. *Буфер вывода* на экран очищается либо при появлении в выходном потоке символа новой строки '\n', либо при выполнении программой очередной операции ввода с клавиатуры.

Язык C++ предоставляет возможности ввода-вывода на низком уровне (*неформатированный* ввод-вывод) и на высоком уровне (*форматированный* ввод-вывод). В первом случае передача информации выполняется блоками байтов без какого-либо преобразования данных. Во втором случае байты группируются для представления таких элементов данных, как целые и вещественные числа, строки символов и так далее.

Для поддержки потоков библиотека C++ содержит иерархию классов, построенную на основе двух базовых классов — `ios` и `streambuf`. Класс `ios` содержит базовые средства управления потоками, являясь родительским для других классов ввода-вывода. Класс `streambuf` обеспечивает общие средства управления буферами потоков и их взаимодействие с физическими устройствами, являясь родительским для других буферных классов. Связь между этими двумя классами реализует поле `bp` в классе `ios`, являющееся указателем на `streambuf`.

Классы стандартных потоков

Потоки, связанные с консольным вводом-выводом (клавиатура–экран), называются *стандартными*. Стандартному потоку ввода соответствует класс

`istream`, стандартному потоку вывода — класс `ostream`. Оба класса являются потомками класса `ios`. Следующим в иерархии является класс `iostream`, наследующий классы `istream` и `ostream` и обеспечивающий общие средства потокового ввода-вывода.

Заголовочные файлы библиотеки ввода-вывода C++

Заголовочные файлы стандартной библиотеки C++ по стандарту указываются без расширения, например `<iostream>`. Все имена стандартной библиотеки принадлежат пространству имен `std`. Для упрощения доступа к именам в программах часто используется директива `using namespace std;`.

Объекты и методы стандартных потоков ввода-вывода

По директиве `#include <iostream>` становятся доступными объекты:

- `cin` — объект класса `istream`, соответствующий стандартному потоку ввода;
- `cout` — объект класса `ostream`, соответствующий стандартному потоку вывода.

Оба потока являются буферизованными. Благодаря объектам `cin` и `cout` становятся доступными и методы соответствующих классов.

Форматированный ввод-вывод реализуется через две перегруженные операции: операция сдвига влево `'<<'` перегружена для вывода в поток и называется *операцией вставки (вывода)* в поток; операция сдвига вправо `'>>'` перегружена для ввода из потока и называется *операцией извлечения (ввода)* из потока. Например, в классе `ostream` определены операции

```
ostream& operator<<( short );  
ostream& operator<<( int );  
ostream& operator<<( double );           // и так далее
```

Рассмотрим, что произойдет, если в программе встретится оператор

```
cout << x;           // ранее в программе:  
int x = 5;
```

Сначала компилятор определит, что левый аргумент имеет тип `ostream`, а правый — тип `int`. Далее он найдет в заголовочном файле `ostream` прототип метода `ostream& operator<<(int)`. В конечном итоге будет вызван метод `cout.operator<<(x)`, в результате выполнения которого мы увидим на экране 5.

Заметим, что в случае вывода значений встроенных типов, таких как `int` или `double`, класс `ostream` обеспечивает их преобразование из внутреннего двоичного представления в строку символов. О форматировании можно не

заботиться, так как класс `ostream` поддерживает форматирование вывода по умолчанию, задаваемое соответствующими полями базового класса `ios`.

Аналогично происходит работа со стандартным потоком ввода через объект `cin`, но направление движения информации противоположное: из потока — в программу.

Если форматирование по умолчанию вас не устраивает, можно воспользоваться либо соответствующими *методами* класса `ios`, либо *манипуляторами* ввода-вывода, представляющими собой функции, которые можно включать прямо в поток. Они определены частично в файлах `istream` и `ostream` и частично в файле `iomanip`. В табл. 4.1 приведены наиболее часто употребляемые манипуляторы¹.

Таблица 4.1. Основные манипуляторы и методы управления форматом

Манипулятор	Метод класса <code>ios</code>	Ввод	Вывод	Описание
<code>endl</code>	—		+	Включить в поток символ новой строки и выгрузить буфер
<code>dec</code>	<code>flags(ios::dec)</code>	+	+	Перейти в десятичную систему счисления
<code>hex</code>	<code>flags(ios::hex)</code>	+	+	Перейти в шестнадцатеричную систему счисления
<code>oct</code>	<code>flags(ios::oct)</code>	+	+	Перейти в восьмеричную систему счисления
<code>setprecision(n)</code>	<code>precision(n)</code>		+	Установить количество отображаемых знаков
<code>setw(n)</code>	<code>width(n)</code>		+	Установить ширину поля вывода
<code>setfill(c)</code>	<code>fill(c)</code>		+	Установить символ заполнения <code>c</code>

Пример вывода целой переменной `x` в шестнадцатеричной форме:

```
cout.flags( ios::hex ); cout << x; // способ 1: использование метода
cout << hex << x; // способ 2: использование манипулятора
```

Остановимся немного подробнее на манипуляторе `setprecision(n)` и соответствующем методе `precision(n)`, поскольку их воздействие на формат вывода подчиняется довольно сложным правилам. И тот, и другой изменяют значение поля `ios::x_precision` (по умолчанию это значение равно шести). Данное поле управляет точностью вывода вещественных чисел,

¹ Полный список манипуляторов и методов форматирования см. в Учебнике, с. 271.

причем его интерпретация зависит от значений других полей, управляющих форматом вывода: `% ios::scientific` — «научный» формат (с плавающей точкой); `% ios::fixed` — формат вывода с фиксированной точкой.

Если установлен хотя бы один из этих форматов (например, с помощью метода `flags`), то `x_precision` задает количество цифр *после десятичной точки*. Если не установлен ни тот, ни другой и вывод идет в так называемом *автоматическом формате*, то значение `x_precision` задает *общее количество значащих цифр*.

Учтите, что большинство параметров форматирования сохраняют свое состояние вплоть до следующего вызова функции, изменяющей это состояние. Исключение — манипулятор `setw(n)` и соответствующий ему метод `width(n)`: их действие распространяется только на ближайшую операцию вывода, после чего восстанавливается ширина поля вывода по умолчанию.

По признаку наличия аргумента манипуляторы подразделяются на *простые* (без аргумента) и *параметризованные* (с аргументом). Для использования последних необходимо подключить заголовочный файл `iomanip`.

С точки зрения *реализации* манипуляторы можно разделить на три группы:

- 1) манипуляторы без аргумента, выводющие в поток управляющий символ (`endl`, `ends`) или очищающие буфер потока (`flush`);
- 2) манипуляторы без аргумента, изменяющие значения полей базового класса `ios`, задающих текущую систему счисления (`dec`, `hex`, `oct`);
- 3) манипуляторы с аргументом.

Кроме рассмотренных, класс `ios` предоставляет и другие методы, обеспечивающие неформатированный ввод-вывод, а также связь с буфером потока. Наиболее часто употребляемые методы ввода-вывода приведены в табл. 4.2.

Таблица 4.2. Некоторые методы ввода-вывода класса `ios`

Метод	Описание
<code>get()</code>	Возвращает код извлеченного из потока символа или EOF, если достигнут конец файла
<code>get(c)</code>	Присваивает код извлеченного из потока символа аргументу <code>c</code>
<code>get(buf, num, lim="\n")</code>	Читает из потока символы, пока не встретится символ <code>lim</code> (по умолчанию <code>"\n"</code>) или пока не будет прочитано <code>num-1</code> символов. Извлеченные символы размещаются в символьный массив <code>buf</code> , к ним добавляется нулевой байт. Символ <code>lim</code> остается в потоке

<code>getline(buf, num, lim="\n")</code>	Выполняется аналогично предыдущему методу, но символ <code>lim</code> удаляется из потока (в массив <code>buf</code> он также не записывается)
<code>peek()</code>	Возвращает код следующего (готового для извлечения) символа в потоке, но не извлекает его; или EOF, если достигнут конец файла
<code>read(buf, num)</code>	Считывает <code>num</code> символов из потока в символьный массив <code>buf</code>
<code>gcount()</code>	Возвращает количество символов, прочитанных последним вызовом функции неформатированного ввода
<code>rdbuf()</code>	Возвращает указатель на буфер типа <code>streambuf</code> , связанный с данным потоком
<code>put(c)</code>	Выводит в поток символ <code>c</code>
<code>write(buf, num)</code>	Выводит в поток <code>num</code> символов из массива <code>buf</code>

Обработка ошибок потоков

Библиотека ввода-вывода C++ обеспечивает гораздо более надежный ввод-вывод, чем старые функции библиотеки C. Это достигается перегрузкой операций извлечения и вставки для всех встроенных типов, что исключает путаницу с типами, которая была возможна при использовании функций `scanf` и `printf` и которая приносила массу неприятностей программистам, поскольку компилятор никак не реагировал на ошибки в спецификации формата.

И все же проблема ошибок практически не волнует нас только при выводе информации в поток `cout`. При вводе никто не может запретить пользователю ввести вместо ожидаемого программой целого числа произвольную строку символов, и если не принять специальных мер, программа «сломается». Для отслеживания ошибок потоков в базовом классе `ios` определено поле `state`, отдельные биты (флаги) которого отображают состояние потока, как показано в табл. 4.3.

Таблица 4.3. Флаги состояния потока

Имя флага	Интерпретация
<code>ios::goodbit</code>	Нет ошибок
<code>ios::eofbit</code>	Достигнут конец файла
<code>ios::failbit</code>	Ошибка форматирования или преобразования
<code>ios::badbit</code>	Серьезная ошибка, после которой пользоваться потоком невозможно

Получить текущее состояние потока можно с помощью метода `rdstate`, который возвращает значение типа `int`. Есть и другие, более удобные для анализа методы:

- `int eof()` — возвращает ненулевое значение, если установлен флаг `eofbit`;
`% int fail()` — возвращает ненулевое значение, если случилась любая ошибка ввода-вывода (но не конец файла). Этому условию соответствует установка либо флага `badbit`, либо флага `failbit`;
- `int bad()` — возвращает ненулевое значение, если случилась серьезная ошибка ввода-вывода (то есть установлен флаг `badbit`);
- `int good()` — возвращает ненулевое значение, если сброшены все флаги ошибок. Если произошла ошибка ввода, в результате которой установлен только флаг `failbit`, то после этого можно продолжать работу с потоком, но сначала нужно сбросить все флаги ошибок, для чего предназначен метод `void clear(int = 0)`. Для сброса флагов достаточно сделать вызов `clear()`, так как аргумент по умолчанию равен нулю. Этот же метод можно использовать для установки соответствующих флагов поля `state`, если передать ему ненулевой аргумент (обычно это комбинация флагов из табл. 4.3, объединенных операцией `|`).

Есть и другие приемы диагностирования ошибочных ситуаций. Мы используем их при решении задачи 4.1.

Перегрузка операций извлечения и вставки для типов, определенных программистом

Одно из удобств C++ — возможность перегрузить операции извлечения и вставки для любого класса `MyClass`, созданного программистом. После этого для любого объекта `obj` класса `MyClass` можно записывать операторы вида `cin >> obj`; `cout << obj`. Удобно, не правда ли?

Однако операции `>>` и `<<` не могут быть элементами класса `MyClass`. Причина в том, что у любой функции-операции класса левым операндом подразумевается объект этого класса, вызывающий данную функцию. Но для операций извлечения (вставки) левый операнд должен быть потоком ввода-вывода. Поэтому эти функции всегда объявляются *дружественными* классу, для которого они создаются.

Общая форма пользовательской *функции извлечения*: `istream& operator >>( istream& is, MyClass& obj )` где `is` — ссылка на входной поток, `obj` — ссылка на объект, принимающий ввод. В теле функции последний оператор должен возвращать объект `is`. Общая форма пользовательской *функции вставки*: `ostream& operator <<( ostream& os, MyClass& obj )`

где `os` — ссылка на выходной поток, а `obj` — ссылка на объект, к которому применяется операция. Последний оператор функции должен возвращать объект `os`. В принципе, возможна и другая сигнатура операции вставки с передачей второго аргумента по значению (`MyClass obj`). Но в этом случае, во-первых, при вызове функции в стеке будет создаваться копия объекта `obj`, а во-вторых, в классе `MyClass` должен быть предусмотрен конструктор копирования.

Рассмотрим теперь задачу, в которой используются стандартные средства вводавывода для потоков и, кроме этого, перегружаются операции извлечения и вставки для пользовательского типа данных.

Задача 4.1. Первичный ввод и поиск информации в базе данных

Написать программу, которая обеспечивает первичный ввод информации в базу данных отдела кадров предприятия численностью до 100 человек (без записи в файл) и поиск информации по заданному критерию. Каждая запись базы данных содержит следующие сведения о сотруднике: фамилия и инициалы; год поступления на работу; оклад. Критерий поиска: сотрудники с окладом, превышающим некоторую заданную величину.

Решение задачи начнем с выявления понятий (классов) и их фундаментальных взаимосвязей.

В данном случае первым понятием является «база данных», и, следовательно, для моделирования этого понятия понадобится класс, который мы назовем `DBase`. Объект типа `DBase` должен содержать некоторую совокупность, или коллекцию, других объектов, соответствующих записям базы данных. Для моделирования понятия «запись базы данных» введем класс `Man`. Очевидно, что взаимоотношение между указанными классами относится к типу «`DBase has a Man`».

На втором этапе необходимо уточнить классы, определив основные поля и набор операций над ними.

Начнем с класса `DBase`. Первый вопрос: какую структуру данных целесообразно использовать для хранения коллекции записей. Поскольку объем базы данных небольшой, выберем самое простое решение — массив объектов типа `Man`. Очевидно, что в конструкторе класса необходимо выполнить выделение памяти для требуемого количества объектов типа `Man`, а в деструкторе — освобождение этой памяти. Адрес начала массива объектов представим полем `Man* pMan`.

Ясно также, что в классе необходимо иметь метод `InitInput` для первичного ввода информации в базу данных и метод `SearchPayNotLess` для поиска сотрудников с окладом, превышающим некоторую заданную величину. Для контроля правильности ввода исходных данных нам пригодится еще один метод — `Show`, выполняющий вывод на экран содержимого базы данных.

Теперь разберемся с классом `Man`. Для хранения информации об одном сотруднике потребуются следующие поля:

- `char* pName` — адрес строки, содержащей фамилию и инициалы;
- `int take_job_year` — год поступления на работу; `% double pay` — величина оклада.

Конструктор класса должен выделять память для хранения указанной строки, а деструктор — освобождать эту память. Для решения второй подзадачи (поиск информации) добавим в класс метод доступа `GetPay`. И наконец, для класса `Man` нужно предусмотреть перегрузку операции извлечения, чтобы обеспечить первичный ввод информации с клавиатуры в методе `InitInput` класса `DBase`, и операцию вставки, которая будет использована в методе `Show` класса `DBase`. Обе операции должны быть реализованы как внешние дружественные функции.

Иногда при решении задачи удобно использовать внешние функции, не являющиеся элементами классов. Обычно они выполняют какую-то рутинную работу и могут быть вызваны как из методов классов, так и из основной функции.

Типичный пример — ввод значений из стандартного потока `cin` с защитой от непреднамеренных ошибок пользователя. Начнем с «наивной» реализации перегруженной операции `>>` для класса `Man`:

```
istream& operator >> ( istream& in, Man& obj ) {  
    // . . . . .  
    in >> obj.take_job_year;      // 1  
    in >> obj.pay;  
    return in;  
}
```

Если при выполнении *оператора 1* пользователь введет вместо целого числа строку символов, программа «сломается». Вашему заказчику наверняка это не понравится. Такая же ситуация возможна и при вводе вещественного числа в следующем операторе. Для решения этих проблем в программе будут использованы функции `GetInt` и `GetDouble`, обеспечивающие надежный

ввод целых и вещественных чисел соответственно. Так как эти функции универсальные и внеклассовые, их код целесообразно разместить в отдельном модуле.

Решение задачи, в котором реализованы рассмотренные концепции, представляет собой многофайловый проект, содержащий файлы DBase.h, DBase.cpp, Man.h, Man.cpp, GetFunc.h, GetFunc.cpp и Main.cpp (листинг 4.1).

Листинг 4.1. Многофайловый проект

```

//////////////////////////////////////// DBase.h
#pragma once
class DBase {
public:
    DBase(int);
    ~DBase();
    void InitInput();
    void Show();
    void SearchPayNotLess(double);
private:
    Man* pMan;
    int nRecords;
};
//////////////////////////////////////// DBase.cpp
#include "Man.h"
#include "DBase.h"
DBase::DBase(int nRec) : nRecords(nRec),
pMan(new Man[nRec]) {}
DBase::~DBase() { if (pMan) delete [] pMan; }
void DBase::InitInput() {
    for ( int i = 0; i < nRecords; i++ ) cin >> *( pMan + i ); // 1
}
void DBase::Show() {
    cout << "=====" << endl;
    cout << "Содержимое базы данных:" << endl;
    for (int i = 0; i < nRecords; i++) cout << *( pMan + i ); // 2
}
void DBase::SearchPayNotLess(double anyPay) {
    bool not_found = true;
    for (int i = 0; i < nRecords; i++)
        if (( pMan + i )->GetPay() >= anyPay) {
            cout << *(pMan + i);
            not_found = false;
        }
    if (not_found) cout << "Таких сотрудников нет." << endl;
}
//////////////////////////////////////// Man.h
#pragma once

```

```

#include <iostream>
#include <iomanip>
using namespace std;
const int l_name = 30;
class Man {
public:
    Man(int lName = 30);
    ~Man();
    double GetPay() const;
    friend istream& operator >>(istream&, Man&); // Операция
извлечения (ввода)

    friend ostream& operator <<(ostream&, Man&); // Операция вставки
(вывода)
private:
    char* pName;
    int take_job_year;
    double pay;
};
////////////////////////////////////// Man.cpp
#include "windows.h" // для работы в среде Windows с кириллицей
#include "Man.h"
#include "GetFunc.h"
Man::Man(int lName) { pName = new char[lName + 1]; }
Man::~Man() { if (pName) delete [] pName; }
double Man::GetPay() const { return pay; }
istream& operator >> (istream& in, Man& ob) { //Операция извлечения
(ввода)
    cout << "\nВведите данные в формате" << endl;
    cout << "Фамилия И.О. <Enter> Год поступления <Enter>";
    cout << " Оклад <Enter>:" << endl;
    in.getline(ob.pName, l_name);
    OemToChar(ob.pName, ob.pName); // для работы в среде Windows с
кириллицей
    ob.take_job_year = GetInt(in); // 3
    ob.pay = GetDouble(in); // 4
    return in;
}
ostream& operator << (ostream& out, Man& ob) { //Операция вставки
(вывода)
    out << setw( 30 ) << setiosflags( ios::left );
    out << ob.pName << " ";
    out << ob.take_job_year << " ";
    out << ob.pay << endl;
    return out;
}
////////////////////////////////////// GetFunc.h
#pragma once

```

```

int GetInt(istream&); // Ввод целого числа
double GetDouble(istream&); // Ввод вещественного числа
//////////////////////////////////// GetFunc.cpp
#include "Man.h"
#include "GetFunc.h"
int GetInt(istream& in) { // ----- ввод целого числа
    int value;
    while ( true ) {
        in >> value; // 5
        if (in.peek() == '\n') { // 6
            in.get(); // 7
            break; }
        else {
            cout << "Повторите ввод (ожидается целое число):" <<
endl; // 8
            in.clear(); // 9
            while ( in.get() != '\n' ) {}; // 10
        }
    }
    return value;
}

double GetDouble(istream& in) { // ----- ввод вещественного числа
    double value;
    while ( true ) {
        in >> value;
        if (in.peek() == '\n') { in.get(); break; }
        else {
            cout << "Повторите ввод (ожидается вещественное число):"
<< endl;
            in.clear();
            while ( in.get() != '\n' ) {};
        }
    }
    return value;
}

//////////////////////////////////// Main.cpp
#include "Man.cpp"
#include "GetFunc.cpp"
#include "DBase.cpp"
int main() {
    const int nRecord = 10; // количество записей в базе данных
    double any_pay;
    setlocale(LC_ALL, "Russian"); // только для работы в среде
Windows
    DBase dBase( nRecord );
    dBase.InitInput();
    dBase.Show();
    cout << "Ввод данных завершен." << endl;

```

```

cout << "===== " << endl;
cout << "Поиск сотрудников, чей оклад не меньше заданной
величины." << endl;
cout << "Поиск завершается при вводе -1." << endl;
while ( true ) {
    cout << "\nВведите величину оклада или -1: ";
    any_pay = GetDouble(cin);
    if (any_pay == -1) break;
    dBase.SearchPayNotLess(any_pay);
}
}

```

Обратите внимание на следующие моменты.

В реализации метода `InitInput` перегруженная для класса `Man` операция извлечения применяется к объекту, задаваемому выражением $*(pMan + i)$, то есть к объекту, адрес которого есть `pMan + i` (*оператор 1*).

Аналогичная адресация объекта для операции вставки использована в методе `Show` (*оператор 2*).

В реализации перегруженной операции извлечения (файл `Man.cpp`) обратите внимание на *операторы 3 и 4*, в которых вызываются функции `GetInt` и `GetDouble`, предназначенные для ввода из стандартного потока целых и вещественных чисел соответственно.

Реализация функций `GetInt` и `GetDouble` находится в файле `GetFunc.cpp`. Рассмотрим подробно первую из них. Ввод целого числа организован внутри бесконечного цикла `while` следующим образом.

Информация читается из входного потока *оператором 5*. Если в буфере типа `streambuf`, связанном с потоком `in`, находится изображение целого числа, завершаемое символом перевода строки `'\n'`, то по завершении операции извлечения в буфере останется только символ `'\n'`. *Оператор 6* проверяет это условие, используя метод `peek`. Если проверка завершилась успешно, *оператор 7* очищает входной буфер от символа `'\n'`, после чего происходит выход из цикла `while` с последующим возвратом из функции значения `value`. Если же введенная информация не является корректным изображением целого числа, то выполняются три действия:

- *оператор 8* выводит сообщение об этом, предлагая повторить ввод;
- сбрасываются флаги ошибок для потока `in` (*оператор 9*);
- с помощью внутреннего цикла `while` из входного буфера извлекаются все символы, вплоть до символа `'\n'` (*оператор 10*). Внешний цикл `while` повторяется сначала.

Функция `GetDouble` работает аналогично.

Функция `main` (файл `Main.cpp`) особых пояснений не требует. Алгоритм прозрачный и воспринимается легко благодаря выразительным именам методов.

В заключение отметим, что рассмотренная задача очень похожа на задачу 1.1. Мы советуем читателю сравнить решение этих двух задач, чтобы увидеть преимущества технологии ООП. Особенно впечатляет сравнение кодов функции `main`: положите рядом тексты листингов 1.1 и 4.1, и вы будете поражены совершенством, лаконичностью и красотой второго решения по сравнению с первым.

Итоги

1. Ввод и вывод представляются в программе как поток байтов и обычно выполняются через буфер. Потоки поддерживаются в библиотеке C++ с помощью иерархии классов, которая реализует безопасный ввод-вывод как стандартных, так и пользовательских типов данных.

2. Ввод-вывод бывает форматированный и неформатированный. Для форматированного ввода-вывода используются перегруженные операции `<<` и `>>`, для неформатированного — методы стандартных классов.

3. Управление форматированием выполняется с помощью манипуляторов и методов стандартных классов.

4. Для вывода объектов пользовательских типов данных следует с помощью дружественных функций перегрузить операции чтения и извлечения.

5. Для обеспечения безопасного ввода необходимо диагностировать возможные ошибки, используя флаги состояния потока и (или) методы `peek`, `get` и `clear`.

Задания

Вариант 1

1. Определить класс с именем `STUDENT`, содержащий следующие поля: фамилия и инициалы; номер группы; успеваемость (массив из пяти элементов).

Определить методы доступа к этим полям и перегруженные операции извлечения и вставки для объектов типа `STUDENT`.

2. Написать программу, выполняющую следующие действия:

- ввод с клавиатуры данных в массив, состоящий из десяти объектов типа `STUDENT`; записи должны быть упорядочены по возрастанию номера группы;

- вывод на дисплей фамилий и номеров групп для всех студентов, включенных в массив, если средний балл студента больше 4.0;
- если таких студентов нет, вывести соответствующее сообщение.

Вариант 2

1. Определить класс с именем STUDENT, содержащий следующие поля: фамилия и инициалы; номер группы; успеваемость (массив из пяти элементов).

Определить методы доступа к этим полям и перегруженные операции извлечения и вставки для объектов типа STUDENT.

2. Написать программу, выполняющую следующие действия:

- ввод с клавиатуры данных в массив, состоящий из десяти объектов типа STUDENT; записи должны быть упорядочены по возрастанию среднего балла;
- вывод на дисплей фамилий и номеров групп для всех студентов, имеющих оценки 4 и 5;
- если таких студентов нет, вывести соответствующее сообщение.

Вариант 3

1. Определить класс с именем STUDENT, содержащий следующие поля: фамилия и инициалы; номер группы; успеваемость (массив из пяти элементов).

Определить методы доступа к этим полям и перегруженные операции извлечения и вставки для объектов типа STUDENT.

2. Написать программу, выполняющую следующие действия:

- ввод с клавиатуры данных в массив, состоящий из десяти объектов типа STUDENT; записи должны быть упорядочены по алфавиту;
- вывод на дисплей фамилий и номеров групп для всех студентов, имеющих хотя бы одну оценку 2;
- если таких студентов нет, вывести соответствующее сообщение.

Вариант 4

1. Определить класс с именем AEROFLOT, содержащий следующие поля: название пункта назначения рейса; номер рейса; тип самолета.

Определить методы доступа к этим полям и перегруженные операции извлечения и вставки для объектов типа AEROFLOT.

2. Написать программу, выполняющую следующие действия:

- ввод с клавиатуры данных в массив, состоящий из семи объектов типа AEROFLOT; записи должны быть упорядочены по возрастанию номера рейса;

- вывод на экран номеров рейсов и типов самолетов, вылетающих в пункт назначения, название которого совпало с названием, введенным с клавиатуры; если таких рейсов нет, выдать на дисплей соответствующее сообщение.

Вариант 5

1. Определить класс с именем AEROFLOT, содержащий следующие поля: название пункта назначения рейса; номер рейса; тип самолета.

Определить методы доступа к этим полям и перегруженные операции извлечения и вставки для объектов типа AEROFLOT.

2. Написать программу, выполняющую следующие действия:

- ввод с клавиатуры данных в массив, состоящий из семи объектов типа AEROFLOT; записи должны быть размещены в алфавитном порядке по названиям пунктов назначения;
- вывод на экран пунктов назначения и номеров рейсов, обслуживаемых самолетом, тип которого введен с клавиатуры;
- если таких рейсов нет, выдать на дисплей соответствующее сообщение.

Вариант 6

1. Определить класс с именем WORKER, содержащий следующие поля:

- фамилия и инициалы работника; название занимаемой должности; год поступления на работу.

Определить методы доступа к этим полям и перегруженные операции извлечения и вставки для объектов типа WORKER.

2. Написать программу, выполняющую следующие действия:

- ввод с клавиатуры данных в массив, состоящий из десяти объектов типа WORKER; записи должны быть размещены по алфавиту;
- вывод на дисплей фамилий работников, чей стаж работы в организации превышает значение, введенное с клавиатуры;
- если таких работников нет, вывести на дисплей соответствующее сообщение.

Вариант 7

1. Определить класс с именем TRAIN, содержащий следующие поля: название пункта назначения; номер поезда; время отправления.

Определить методы доступа к этим полям и перегруженные операции извлечения и вставки для объектов типа TRAIN.

2. Написать программу, выполняющую следующие действия:

- ввод с клавиатуры данных в массив, состоящий из восьми объектов типа TRAIN; записи должны быть размещены в алфавитном порядке по названиям пунктов назначения;
- вывод на экран информации о поездах, отправляющихся после введенного с клавиатуры времени;
- если таких поездов нет, выдать на дисплей соответствующее сообщение.

Вариант 8

1. Определить класс с именем TRAIN, содержащий следующие поля: название пункта назначения; номер поезда; время отправления.

Определить методы доступа к этим полям и перегруженные операции извлечения и вставки для объектов типа TRAIN.

2. Написать программу, выполняющую следующие действия:

- ввод с клавиатуры данных в массив, состоящий из шести объектов типа TRAIN; записи должны быть упорядочены по времени отправления поезда;
- вывод на экран информации о поездах, направляющихся в пункт, название которого введено с клавиатуры;
- если таких поездов нет, выдать на дисплей соответствующее сообщение.

Вариант 9

1. Определить класс с именем TRAIN, содержащий следующие поля: название пункта назначения; номер поезда; время отправления.

Определить методы доступа к этим полям и перегруженные операции извлечения и вставки для объектов типа TRAIN.

2. Написать программу, выполняющую следующие действия:

- ввод с клавиатуры данных в массив, состоящий из восьми объектов типа TRAIN; записи должны быть упорядочены по номерам поездов;
- вывод на экран информации о поезде, номер которого введен с клавиатуры; если таких поездов нет, выдать на дисплей соответствующее сообщение.

Вариант 10

1. Определить класс с именем MARSH, содержащий следующие поля:

- название начального пункта маршрута; название конечного пункта маршрута; номер маршрута.

Определить методы доступа к этим полям и перегруженные операции извлечения и вставки для объектов типа MARSH.

2. Написать программу, выполняющую следующие действия:

- ввод с клавиатуры данных в массив, состоящий из восьми объектов типа MARSH; записи должны быть упорядочены по номерам маршрутов;

- вывод на экран информации о маршруте, номер которого введен с клавиатуры; если таких маршрутов нет, выдать на дисплей соответствующее сообщение.

Вариант 11

1. Определить класс с именем MARSH, содержащий следующие поля:

- название начального пункта маршрута; название конечного пункта маршрута; номер маршрута.

Определить методы доступа к этим полям и перегруженные операции извлечения и вставки для объектов типа MARSH.

2. Написать программу, выполняющую следующие действия:

- ввод с клавиатуры данных в массив, состоящий из восьми объектов типа MARSH; записи должны быть упорядочены по номерам маршрутов;
- вывод на экран информации о маршрутах, которые начинаются или кончаются в пункте, название которого введено с клавиатуры;
- если таких маршрутов нет, выдать на дисплей соответствующее сообщение.

Вариант 12

1. Определить класс с именем NOTE, содержащий следующие поля: фамилия, имя; номер телефона; день рождения (массив из трех чисел).

Определить методы доступа к этим полям и перегруженные операции извлечения и вставки для объектов типа NOTE.

2. Написать программу, выполняющую следующие действия:

- ввод с клавиатуры данных в массив, состоящий из восьми объектов типа NOTE; записи должны быть упорядочены по датам дней рождения;
- вывод на экран информации о человеке, номер телефона которого введен с клавиатуры;
- если такого нет, выдать на дисплей соответствующее сообщение.

Вариант 13

1. Определить класс с именем NOTE, содержащий следующие поля: фамилия, имя; номер телефона; день рождения (массив из трех чисел).

Определить методы доступа к этим полям и перегруженные операции извлечения и вставки для объектов типа NOTE.

2. Написать программу, выполняющую следующие действия:

- ввод с клавиатуры данных в массив, состоящий из восьми объектов типа NOTE; записи должны быть размещены по алфавиту;
- вывод на экран информации о людях, чьи дни рождения приходятся на месяц, значение которого введено с клавиатуры;

- если таких нет, выдать на дисплей соответствующее сообщение.

Вариант 14

1. Определить класс с именем NOTE, содержащий следующие поля: фамилия, имя; номер телефона; день рождения (массив из трех чисел).

Определить методы доступа к этим полям и перегруженные операции извлечения и вставки для объектов типа NOTE.

2. Написать программу, выполняющую следующие действия:

- ввод с клавиатуры данных в массив, состоящий из восьми объектов типа NOTE; записи должны быть упорядочены по трем первым цифрам номера телефона;
- вывод на экран информации о человеке, чья фамилия введена с клавиатуры; если такого нет, выдать на дисплей соответствующее сообщение.

Вариант 15

1. Определить класс с именем ZNAK, содержащий следующие поля: фамилия, имя; знак Зодиака; день рождения (массив из трех чисел).

Определить методы доступа к этим полям и перегруженные операции извлечения и вставки для объектов типа ZNAK.

2. Написать программу, выполняющую следующие действия:

- ввод с клавиатуры данных в массив, состоящий из восьми объектов типа ZNAK; записи должны быть упорядочены по датам дней рождения;
- вывод на экран информации о человеке, чья фамилия введена с клавиатуры;
- если такого нет, выдать на дисплей соответствующее сообщение.

Вариант 16

1. Определить класс с именем ZNAK, содержащий следующие поля: фамилия, имя; знак Зодиака; день рождения (массив из трех чисел).

Определить методы доступа к этим полям и перегруженные операции извлечения и вставки для объектов типа ZNAK.

2. Написать программу, выполняющую следующие действия:

- ввод с клавиатуры данных в массив, состоящий из восьми объектов типа ZNAK; записи должны быть упорядочены по датам дней рождения;
- вывод на экран информации о людях, родившихся под знаком, наименование которого введено с клавиатуры;
- если таких нет, выдать на дисплей соответствующее сообщение.

Вариант 17

1. Определить класс с именем ZNAK, содержащий следующие поля: фамилия, имя; знак Зодиака; день рождения (массив из трех чисел).

Определить методы доступа к этим полям и перегруженные операции извлечения и вставки для объектов типа ZNAK.

2. Написать программу, выполняющую следующие действия:

- ввод с клавиатуры данных в массив, состоящий из восьми объектов типа ZNAK; записи должны быть упорядочены по знакам Зодиака;
- вывод на экран информации о людях, родившихся в месяц, значение которого введено с клавиатуры;
- если таких нет, выдать на дисплей соответствующее сообщение.

Вариант 18

1. Определить класс с именем PRICE, содержащий следующие поля:

- название товара;
- название магазина, в котором продается товар; стоимость товара в руб.

Определить методы доступа к этим полям и перегруженные операции извлечения и вставки для объектов типа PRICE.

2. Написать программу, выполняющую следующие действия:

- ввод с клавиатуры данных в массив, состоящий из восьми объектов типа PRICE; записи должны быть размещены в алфавитном порядке по названиям товаров;
- вывод на экран информации о товаре, название которого введено с клавиатуры;
- если таких товаров нет, выдать на дисплей соответствующее сообщение.

Вариант 19

1. Определить класс с именем PRICE, содержащий следующие поля:

- название товара;
- название магазина, в котором продается товар; стоимость товара в руб.

Определить методы доступа к этим полям и перегруженные операции извлечения и вставки для объектов типа PRICE.

2. Написать программу, выполняющую следующие действия:

- ввод с клавиатуры данных в массив, состоящий из восьми объектов типа PRICE; записи должны быть размещены в алфавитном порядке по названиям магазинов;
- вывод на экран информации о товарах, продающихся в магазине, название которого введено с клавиатуры;
- если такого магазина нет, выдать на дисплей соответствующее сообщение.

Вариант 20

1. Определить класс с именем ORDER, содержащий следующие поля: расчетный счет плательщика; расчетный счет получателя; перечисляемая сумма.

Определить методы доступа к этим полям и перегруженные операции извлечения и вставки для объектов типа ORDER.

2. Написать программу, выполняющую следующие действия:

- ввод с клавиатуры данных в массив, состоящий из восьми объектов типа ORDER; записи должны быть размещены в алфавитном порядке по расчетным счетам плательщиков;
- вывод на экран информации о сумме, снятой с расчетного счета плательщика, введенного с клавиатуры;
- если такого расчетного счета нет, выдать на дисплей соответствующее сообщение.

Лабораторная работа № 5.

Файловые и строковые потоки. Строки класса string

Файловые потоки

Для поддержки файлового ввода и вывода стандартная библиотека C++ содержит классы, указанные в табл. 5.1.

Таблица 5.1. Классы файловых потоков

Класс	Инстанцирован шаблона	из Базовый шаблонный класс	Назначение
ifstream	basic_ifstream	basic_istream	Входной файловый поток
ofstream	basic_ofstream	basic_ostream	Выходной файловый поток
fstream	basic_fstream	basic_iostream	Двунаправленный файловый поток

Так как классы файловых потоков являются производными от классов `istream`, `ostream` и `iostream` соответственно, то они наследуют все методы указанных классов, перегруженные операции вставки и извлечения, манипуляторы, состояние потоков и т.д. Как и стандартные потоки, файловые потоки обеспечивают гораздо более надежный ввод-вывод, чем старые функции библиотеки C. Для использования файловых потоков необходимо подключить заголовок `<fstream>`.

Работа с файлом обычно предполагает следующие операции:

- создание потока (потокowego объекта);
- открытие потока и связывание его с файлом;
- обмен с потоком (ввод-вывод);
- закрытие файла.

Классы файловых потоков содержат несколько конструкторов, позволяющих варьировать способы создания потоковых объектов. *Конструкторы с параметрами* создают объект соответствующего класса, открывают файл с указанным именем и связывают файл с объектом:

```
ifstream( const char* name, int mode = ios::in );  
ofstream( const char* name, int mode = ios::out | ios::trunc );  
fstream ( const char* name, int mode = ios::in | ios::out );
```

Второй параметр конструктора задает режим открытия файла. Если значение по умолчанию вас не устраивает, можно указать другое, выбрав одно или несколько значений (объединенных операцией `|`) из указанных в табл. 5.2.

Таблица 5.2. Значения аргумента `mode`

Флаг	Назначение
------	------------

<code>ios::in</code>	Открыть файл для ввода
<code>ios::out</code>	Открыть файл для вывода
<code>ios::ate</code>	Установить указатель на конец файла
<code>ios::app</code>	Открыть в режиме добавления в конец файла
<code>ios::trunc</code>	Если файл существует, обрезать его до нулевой длины
<code>ios::binary</code>	Открыть в двоичном режиме (по умолчанию используется текстовый)

Конструкторы без параметров создают объект соответствующего класса, не связывая его с файлом. В этом случае связь потока с конкретным файлом осуществляется позже — вызовом метода `open`, который имеет параметры, аналогичные параметрам рассмотренных выше конструкторов. Приведем примеры создания потоковых объектов и связывания их с конкретными файлами:

```
ofstream flog( "flog.txt" );           // <---- Файлы для вывода
ofstream fout1, fout2;                // <----
fout1.open( "test1", ios::app );       // <----
fout2.open( "test2", ios::binary );    // <----
ifstream finpl( "data.txt" );          // Файл для ввода
fstream myfile;                        // Файл для ввода и вывода
myfile.open( "mf.dat" );
```

Если в качестве параметра `name` задано *краткое имя файла* (без указания полного пути), подразумевается, что файл открывается в текущем каталоге, в противном случае требуется задавать *полное имя файла*, например:

```
ifstream finpl( "D:\\VCwork\\Task1\\data.txt" );
```

ВНИМАНИЕ

Если файл не удастся открыть (например, входной файл в указанном каталоге не найден или нет свободного места на диске для выходного файла), то независимо от способа его открытия — конструктором или методом `open` — потоковый объект принимает значение, равное нулю. Поэтому рекомендуется всегда проверять, чем завершилась попытка открытия файла, например:

```
ifstream finpl( "data.txt" );
if ( !finpl ) { cerr << "Файл data.txt не найден." << endl;
               throw "Ошибка открытия файла "; }
```

После того как файловый поток открыт, работа с ним чрезвычайно проста: с входным потоком можно обращаться так же, как со стандартным объектом `cin`, а с выходным так же, как со стандартным объектом `cout`.

Если при чтении данных требуется контролировать, был ли достигнут *конец файла* после очередной операции ввода, используется метод `eof`, возвращающий нулевое значение, если конец файла еще не достигнут, и ненулевое значение — если уже достигнут. Учтите, что в C++ после чтения из

файла *последнего элемента* условие конца файла *не возникает*! Оно возникает при следующем чтении, когда программа пытается считать данные за последним элементом в файле.

Если при выполнении операций ввода-вывода фиксируется некоторая ошибочная ситуация, потоковый объект также принимает значение, равное нулю. Рекомендуется особо следить за состоянием потокового объекта во время выполнения операций вывода, так как диски «не резиновые» и имеют тенденцию переполняться.

Когда программа покидает область видимости потокового объекта, он уничтожается. При этом перестает существовать связь между потоковым объектом и физическим файлом, а физический файл закрывается. Если алгоритм требует более раннего закрытия файла, можно воспользоваться методом `close`.

Для примера работы с файловыми потоками приведем программу копирования одного файла в другой (листинг 5.1) и программу вывода на экран содержимого текстового файла (листинг 5.2).

Листинг 5.1. Копирование файлов

```
#include <iostream>
#include <fstream>
using namespace std;
void error(const char* text1, const char* text2 = "") {
    cerr << text1 << ' ' << text2 << endl;
    cin.get(); exit(1);
}
int main(int argc, char* argv[]) { // Имена файлов берутся из командной
строки
    //if (argc != 3) error("Неверное число аргументов");
    ifstream from("data.txt"); // открываем входной
файл
    if (!from) error("Входной файл не найден:", "data.txt");
    ofstream to("data2.txt"); // открываем выходной
файл
    if (!to) error("Выходной файл не открыт:", "data2.txt");
    char ch;
    while ( from.get(ch) ) {
        to.put(ch);
        if (!to) error("Ошибка записи (диск переполнен).");
    }
    cout << "Копирование из " << "data.txt" << " в " << "data2.txt"
<< " завершено.\n";
    cin.get();
}
```

Листинг 5.2. Вывод на экран содержимого текстового файла

```
#include <iostream>
#include <fstream>
using namespace std;

// ... <----- Здесь определение функции error() — из листинга 5.1
```

```

int main(int argc, char* argv[]) { // имя файла задается в командной
строке
    //if (argc != 2) error("Неверное число аргументов");
    ifstream tfile("data.txt");           // открываем входной
файл
    if (!tfile) error("Входной файл не найден:", "data.txt");
    char buf[1024];
    while (!tfile.eof()) {
        tfile.getline(buf, sizeof(buf));
        cout << buf << endl;
    }
    cin.get();
}

```

В программе предполагается, что длина строки в файле не превышает 1024 символов.

При необходимости можно увеличить размер буфера buf до требуемой величины.

Строковые потоки

Работу со строковыми потоками обеспечивают классы `istringstream`, `ostringstream` и `stringstream`, которые являются производными от классов `istream`, `ostream` и `iostream` соответственно. Для использования строковых потоков необходимо подключить к программе заголовочный файл `<sstream>`.

Применение строковых потоков аналогично применению файловых потоков, но информация потока физически размещается в оперативной памяти, а не на диске. Кроме того, классы строковых потоков содержат метод `str`, возвращающий копию строки типа `string` или присваивающий потоку значение такой строки:

```

string str() const;
void str( const string& s );

```

Строковые потоки являются некоторыми аналогами функций `sscanf` и `sprintf` библиотеки C, которые также работают со строками в памяти, имитируя консольный ввод-вывод. Например, с помощью `sprintf` можно сформировать в памяти некоторую символьную строку, которую затем отобразить на экране. Эта же проблема легко решается с помощью объекта типа `ostringstream`.

В качестве примера приведем модифицированную версию листинга 5.2, которая выводит содержимое текстового файла на экран, предваряя каждую строку текстовой меткой «Line N:», где N — номер строки (листинг 5.3).

Листинг 5.3. Вывод на экран пронумерованного содержимого текстового файла

```

#include <iostream>
#include <iomanip>

```

```

#include <fstream>
#include <sstream>
using namespace std;

//<Здесь определение функции error() – из листинга 5.1

int main(int argc, char* argv[]) {
    //if (argc != 2) error("Неверное число аргументов.");
    ifstream tfile( "data.txt" );
    if (!tfile) error("Входной файл не найден:", "data.txt");
    int n = 0;
    char buf[1024];
    while ( !tfile.eof() ) {
        n++;
        tfile.getline(buf, sizeof(buf));
        ostringstream line;
        line << "Line " << setw(3) << n << ": " << buf << endl;
        cout << line.str();
    }
    cin.get();
}

```

Строки класса string

Мы уже пользовались объектами класса `string`, начиная со второго семинара, и успели оценить удобства, обеспечиваемые этим классом в сравнении с традиционными С-строками (то есть массивами символов типа `char`, завершаемыми нулевым байтом). Сейчас мы рассмотрим строки типа `string` более подробно.

Для использования строк типа `string` необходимо подключить к программе заголовочный файл `<string>`. Важнейшей особенностью класса `string` является управление памятью как при размещении строки, так и при ее модификациях, изменяющих длину строки. Поэтому вы можете «забыть» об операциях `new` и `delete`, неаккуратное обращение с которыми является источником труднодиагностируемых ошибок.

Кроме этого, строки типа `string` защищены от ошибочных обращений к памяти, связанных с выходом за их границы. Но за все надо платить: строки типа `string` значительно проигрывают С-строкам в эффективности. Поэтому, если от программы требуется максимальное быстродействие, иногда лучше воспользоваться С-строками. В большинстве же программ на С++ строки типа `string` обеспечивают необходимую скорость обработки, поэтому их применение предпочтительней.

Для понимания определений методов класса `string` необходимо знать назначение некоторых имен. Так, в пространстве `std` определен идентификатор `size_type`, являющийся синонимом типа `unsigned int`. В классе `string` определена константа `pros`, задающая максимально возможное число, которое в

зависимости от контекста означает либо «все элементы» строки, либо отрицательный результат поиска. Так как максимально возможное число имеет вид 0xFFFF...FFFF, то в случае присваивания его переменной типа `int` получится значение `-1`.

В классе `string` имеется несколько конструкторов, вот самые простые:

```
string(); // создает пустой объект класса string
string( const char* ); //создает объект, инициализируя его
                        значением C-строки
```

Класс содержит три операции присваивания:

```
string& operator=( const string& str );// присвоить значение другой
                                     строки string
string& operator=( const char* s );// присвоить значение C-строки
string& operator=( char c );// присвоить значение символа
```

В табл. 5.3 приведены допустимые для объектов класса `string` операции.

Таблица 5.3. Операции класса `string`

Операция	Действие	Операция	Действие
=	Присваивание	>	Больше
+	Конкатенация	>=	Больше или равно
==	Равенство	[]	Индексация
!=	Неравенство	<<	Вывод
<	Меньше	>>	Ввод
<=	Меньше или равно	+=	Добавление

Использование операций очевидно. Размеры строк устанавливаются автоматически так, чтобы объект мог содержать присваиваемое ему значение. Нумерация элементов строки начинается с нуля. Кроме операции индексации, для доступа к элементу строки определен метод `at(size_type n)`, который можно использовать как для чтения, так и для записи `n`-го элемента строки:

```
cout << s.at(2); // Будет выведен 2-й символ строки s
s.at(5) = 'W'; // 5-й символ заменяется символом W
```

Заметим, что в операции индексации не проверяется выход за диапазон строки. Метод `at`, напротив, такую проверку содержит, и, если индекс превышает длину строки, порождается исключение `out_of_range`.

В табл. 5.4 приведены некоторые употребительные методы класса `string`.

Таблица 5.4. Методы класса `string`

Метод	Назначение
<code>size_type size() const;</code>	Возвращает размер строки
<code>size_type length() const;</code>	То же, что и <code>size</code>

<code>insert(size_type pos1, const string& str);</code>	Вставляет строку <code>str</code> в вызывающую строку, начиная с позиции <code>pos1</code>
<code>replace(size_type pos1, size_type n1, const string& str);</code>	Заменяет <code>n1</code> элементов, начиная с позиции <code>pos1</code> вызывающей строки, элементами строки <code>str</code>
<code>string substr(size_type pos=0, size_type n=npos) const;</code>	Возвращает подстроку длиной <code>n</code> , начиная с позиции <code>pos</code>
<code>size_type find(const string& str, size_type pos=0) const;</code>	Ищет самое левое вхождение строки <code>str</code> в вызывающую строку, начиная с позиции <code>pos</code> . Возвращает позицию вхождения, или <code>npos</code> , если вхождение не найдено
<code>size_type find(char c, size_type pos=0) const;</code>	Ищет самое левое вхождение символа <code>c</code> , начиная с позиции <code>pos</code> . Возвращает позицию вхождения, или <code>npos</code> , если вхождение не найдено
Метод	Назначение
<code>size_type rfind(const string& str, size_type pos=0) const;</code>	Ищет самое правое вхождение строки <code>str</code> , начиная с позиции <code>pos</code> ¹
<code>size_type rfind(char c, size_type pos=0) const;</code>	Ищет самое правое вхождение символа <code>c</code> , начиная с позиции <code>pos</code>
<code>size_type find_first_of(const string& str, size_type pos=0) const;</code>	Ищет самое левое вхождение любого символа строки <code>str</code> , начиная с позиции <code>pos</code>
<code>size_type find_last_of(const string& str, size_type pos=0) const;</code>	Ищет самое правое вхождение любого символа строки <code>str</code> , начиная с позиции <code>pos</code>
<code>swap(string& str);</code>	Обменивает содержимого вызывающей строки и строки <code>str</code>
<code>erase(size_type pos=0, size_type n=npos);</code>	Удаляет <code>n</code> элементов, начиная с позиции <code>pos</code>
<code>clear();</code>	Очищает всю строку
<code>const char* c_str() const;</code>	Возвращает указатель на С-строку, содержащую копию вызывающей строки. Полученную С-строку нельзя пытаться изменить

¹ Возвращаемые значения такие же, как и у метода `find`.

Size_type copy(char* s, size_type n, size_type pos=0) const;	Копирует в массив s n элементов вызывающей строки, начиная с позиции pos. Нуль-символ в результирующий массив не заносятся. Метод возвращает количество скопированных элементов
--	---

Поясним применение метода find. Допустим, что вы работаете над программой для игры в шахматы с компьютером, а в данный момент пишете функцию для ввода обозначения колонки шахматной доски. Эти колонки, как известно, обозначаются символами латинского алфавита от А до Н. Желательно,

чтобы ваша функция не допускала ввод некорректных символов. Приведем одно из возможных решений:

```
char GetColumn() {
    string goodChar = "ABCDEFGH";
    char symb;
    cout << "Введите обозначение колонки: ";
    while (1) {
        cin >> symb;
        if ( -1 == goodChar.find( symb ) ) {
            cout << "Ошибка. Введите корректный символ:\n"; continue;
        }
        return symb;
    }
}
```

Метод find здесь используется для проверки, принадлежит ли введенный с клавиатуры символ множеству символов, заданному с помощью строки goodChar.

Задача 5.1. Подсчет количества вхождений слова в текст

Написать программу, которая определяет, сколько раз встретилось заданное слово в текстовом файле. Текст не содержит переносов слов. Максимальная длина строки в файле неизвестна².

Определим слово как последовательность алфавитно-цифровых символов, после которых следует либо знак пунктуации (., ,, ?, !), либо разделитель. В качестве разделителей могут выступать один или несколько пробелов, один или несколько символов табуляции '\t' и символ конца строки '\n'. Для хранения заданного слова (оно вводится с клавиатуры) определим переменную word типа string.

Поскольку максимальная длина строки в файле неизвестна, будем читать файл не построчно, а пословно, размещая очередное прочитанное слово в переменной curword типа string. Это можно реализовать с помощью операции

² Аналогичная задача решалась в первой части практикума (задача 5.2), но с упрощающим ограничением: длина строки в файле не более 80 символов.

>>, которая в случае операнда типа `string` игнорирует все разделители, предваряющие текущее слово, и считывает символы текущего слова в переменную `curword`, пока не встретится очередной разделитель.

Очевидно, что «опознание» текущего слова должно осуществляться с учетом возможного наличия после него одного из знаков пунктуации. Для решения этой задачи определим глобальную функцию

```
bool equal( const string& cw, const string& w );
```

которая возвращает значение `true`, если текущее слово `cw` совпадает с заданным словом `w` с точностью до знака пунктуации, или `false` — в противном случае. Имея такую функцию, очень просто составить алгоритм основного цикла:

- прочитать очередное слово;
- если оно совпадает с заданным словом `w` (с точностью до знака пунктуации), то увеличить на единицу значение счетчика `count`.

Текст решения приведен в листинге 5.4.

Листинг 5.4. Подсчет количества вхождений слова в текст

```
#include <windows.h> // здесь прототип функции OemToChar
#include <iostream>
#include <fstream>
#include <string>
#include <clocale> // здесь прототип функции setlocale
using namespace std;
bool equal( const string& cw, const string& w ) {
    char punct[] = { '.', ',', '?', '!' };
    if ( cw == w ) return true;
    for ( int i = 0; i < sizeof( punct ); ++i )
        if ( cw == w + punct[i] ) return true;
    return false;
}
int main() {
    string word, curword;
    setlocale( LC_ALL, "Russian" ); // Для работы с русскими символами
    cout << " Введите слово для поиска: ";
    cin >> word;
    char pattern[30];
    OemToChar( word.c_str(), pattern ); // Для работы с русскими
СИМВОЛАМИ
    ifstream fin("infile.txt");
    if ( !fin ) { cout << "Ошибка открытия файла." << endl; cin.get();
return 1; }
    int count = 0;
    while ( !fin.eof() ) {
        fin >> curword;
        if ( equal( curword, string( pattern ) ) ) count++;
    }
    cout << "Количество вхождений слова: " << count << endl;
    cin.get(); cin.get();
```

}

Обратите внимание на реализацию функции `equal` и, в частности, на использование операции сложения для добавления в конец строки `w` одного из знаков пунктуации.

Задача 5.2. Вывод вопросительных предложений

*Написать программу, которая считывает текст из файла и выводит на экран только вопросительные предложения из этого текста*³.

Итак, имеется текстовый файл неизвестного размера, состоящий из неизвестного количества предложений. Предложение может занимать несколько строк, поэтому читать файл построчно неудобно. При решении аналогичной задачи в первой части практикума было принято решение выделить буфер, в который поместится *весь* файл. Такое решение тоже нельзя признать идеальным — ведь файл может иметь сколь угодно большие размеры, и тогда программа окажется неработоспособной.

Поищем более удачное решение, используя новые средства языка C++, с которыми мы познакомились на этом семинаре. Попробуем читать файл пословно, как и в предыдущей программе, в переменную `word` типа `string`, и отправлять каждое прочитанное слово в строковый поток `sentence` типа `ostringstream`, который, как вы уже догадались, будет хранилищем очередного предложения.

При таком подходе, однако, есть проблема, связанная с потерей разделителей при чтении файла операцией `fin >> word`. Чтобы ее решить, будем «заглядывать» в следующую позицию файлового потока `fin` с помощью метода `peek`. При обнаружении символа-разделителя его нужно отправить в поток `sentence` и переместиться на следующую позицию в потоке `fin`, используя метод `seekg`. Подробности обнаружения символа-разделителя инкапсулируем в глобальную функцию `isLimit`.

Осталось решить подзадачи:

- обнаружить конец предложения, то есть один из символов `\. ' , \! ' , ' ? ' ;`
- если это вопросительное предложение, вывести его в поток `cout`, в противном случае очистить поток `sentence` для накопления следующего предложения.

Рассмотренный алгоритм реализуется в листинге 5.5.

Листинг 5.5. Вывод вопросительных предложений

```
#include <iostream>
#include <fstream>
#include <sstream>
```

³ Аналогичная задача рассмотрена в первой части практикума (задача 5.3).

```

#include <string>
using namespace std;
bool isLimit( char c ) {
    char lim[] = { ' ', '\t', '\n' };
    for ( int i = 0; i < sizeof( lim ); ++i ) if ( c == lim[i] )
return true;
    return false;
}
int main() {
    ifstream fin( "infile.txt" );
    if ( !fin ) { cout << "Ошибка открытия файла." << endl; cin.get();
return 1; }
    int count = 0;
    string word;
    ostringstream sentence;
    while( !fin.eof() ) {
        char symb;
        while( isLimit( symb = fin.peek() ) ) {
            sentence << symb;
            if ( symb == '\n' ) break;
            fin.seekg( 1, ios::cur );
        }
        fin >> word;
        sentence << word;
        char last = word[word.size() - 1];
        if ( ( last == '.' ) || ( last == '!' ) ) {
            sentence.str(""); // очистка потока
            continue;
        }
        if ( last == '?' ) { cout << sentence.str(); sentence.str( ""
); count++; }
    }
    if ( !count ) cout << "Вопросительных предложений нет." << endl;
    cin.get();
}

```

Протестируйте приведенные программы. Не забудьте поместить в один каталог с программой текстовый файл `infile.txt`. Если программа запускается из среды Visual Studio, файл должен находиться в каталоге с исходными текстами проекта, если же из папки Debug (например, в режиме командной строки), то в этой папке.

Итоги

1. Для поддержки файлового ввода и вывода стандартная библиотека C++ содержит классы `ifstream`, `ofstream`, `fstream`.
2. Работа с файлом предполагает следующие операции: создание потока, открытие потока и связывание его с файлом, обмен с потоком (ввод-вывод), закрытие файла. Рекомендуется всегда проверять, чем завершилось открытие файла.

3. Если в процессе ввода-вывода фиксируется ошибочная ситуация, потоковый объект принимает значение, равное нулю.

4. Следите за состоянием выходного потока после каждой операции вывода, так как на диске может не оказаться свободного места.

5. Работу со строковыми потоками обеспечивают классы `istream`, `ostream`, `stringstream`. Использование строковых потоков аналогично применению файловых потоков. Различие в том, что физически информация потока размещается в оперативной памяти, а не в файле на диске.

6. Класс `string` стандартной библиотеки C++ предоставляет программисту очень удобные средства работы со строками. Класс берет на себя управление памятью, как при размещении строки, так и при всех ее модификациях.

Задания

Выполнить задания семинара 5, используя потоковые классы.

1 Написать программу, которая считывает из текстового файла три предложения и выводит их в обратном порядке.

2 Написать программу, которая считывает текст из файла и выводит на экран только предложения, содержащие введенное с клавиатуры слово.

3 Написать программу, которая считывает текст из файла и выводит на экран только строки, содержащие двузначные числа.

4 Написать программу, которая считывает английский текст из файла и выводит на экран слова, начинающиеся с гласных букв.

5 Написать программу, которая считывает текст из файла и выводит его на экран, меняя местами каждые два соседних слова.

6 Написать программу, которая считывает текст из файла и выводит на экран только предложения, не содержащие запятых.

7 Написать программу, которая считывает текст из файла и определяет, сколько в нем слов, состоящих из не более чем четырех буквами.

8 Написать программу, которая считывает текст из файла и выводит на экран только цитаты, то есть предложения, заключенные в кавычки.

9 Написать программу, которая считывает текст из файла и выводит на экран только предложения, состоящие из заданного количества слов.

10 . Написать программу, которая считывает английский текст из файла и выводит на экран слова текста, начинающиеся и оканчивающиеся гласными букв.

11 . Написать программу, которая считывает текст из файла и выводит на экран только строки, не содержащие двузначных чисел.

12 . Написать программу, которая считывает текст из файла и выводит на экран только предложения, начинающиеся с тире, перед которым могут находиться только пробельные символы.

13 . Написать программу, которая считывает английский текст из файла и выводит его на экран, заменив каждую первую букву слов, начинающихся с гласной буквы, на прописную.

14 . Написать программу, которая считывает текст из файла и выводит его на экран, заменив цифры от 0 до 9 на слова «ноль», «один», ..., «девять», начиная каждое предложение с новой строки.

15 . Написать программу, которая считывает текст из файла, находит самое длинное слово и определяет, сколько раз оно встретилось в тексте.

16 . Написать программу, которая считывает текст из файла и выводит на экран сначала вопросительные, а затем восклицательные предложения.

17 . Написать программу, которая считывает текст из файла и выводит его на экран, добавляя после каждого предложения, сколько раз встретилось в нем введенное с клавиатуры слово.

18 . Написать программу, которая считывает текст из файла и выводит на экран все его предложения в обратном порядке.

19 . Написать программу, которая считывает текст из файла и выводит на экран сначала предложения, начинающиеся с однобуквенных слов, а затем все остальные.

20 . Написать программу, которая считывает текст из файла и выводит на экран предложения, содержащие максимальное количество знаков пунктуации.

Основные концепции стандартной библиотеки

Стандартная библиотека шаблонов (Standard Template Library, STL) состоит из двух основных частей: набора контейнерных классов и набора обобщенных алгоритмов.

Контейнеры — это объекты, содержащие другие однотипные объекты. Контейнерные классы являются шаблонными, поэтому хранимые в них объекты могут быть как встроенных, так и пользовательских типов. Эти объекты должны допускать копирование и присваивание. Встроенные типы этим требованиям удовлетворяют; то же самое относится к классам, если конструктор копирования или операция присваивания не объявлены в них закрытыми или защищенными. В контейнерных классах реализованы такие типовые структуры данных, как стек, список, очередь и т. д.

Обобщенные алгоритмы реализуют большое количество процедур, применимых к контейнерам — например, поиск, сортировку, слияние и т. п., но они не являются методами контейнерных классов. Наоборот, алгоритмы представлены в STL в форме глобальных шаблонных функций. Благодаря этому достигается их универсальность: эти функции можно применять не только к объектам контейнерных классов, но и к массивам. Независимость от типов контейнеров достигается за счет косвенной связи функции с контейнером: в функцию передается не сам контейнер, а пара адресов, *first*, *last*, задающая диапазон обрабатываемых элементов.

Реализация указанного механизма взаимодействия базируется на использовании так называемых *итераторов*. *Итераторы* — это обобщение концепции указателей: они ссылаются на элементы контейнера. Их можно инкрементировать для последовательного продвижения по контейнеру, как обычные указатели, а также разыменовывать для получения или изменения значения элемента.

Контейнеры

Контейнеры STL можно разделить на два типа: последовательные и ассоциативные.

Последовательные контейнеры обеспечивают хранение конечного количества однотипных объектов в виде непрерывной последовательности. К базовым последовательным контейнерам относятся *векторы* (*vector*), *списки* (*list*) и *двусторонние очереди* (*deque*). Есть еще специализированные контейнеры (или *адаптеры* контейнеров), реализованные на основе базовых, — *стеки* (*stack*), *очереди* (*queue*) и *очереди с приоритетами* (*priority_queue*).

Кстати, обычный встроенный массив C++ также может рассматриваться как последовательный контейнер. Проблема с массивами заключается в том, что их размеры нужно указывать в исходном коде, а это часто бывает неизвестно заранее. Если же выделять память для массива динамически (операцией *new*), алгоритм усложняется из-за необходимости отслеживать время жизни массива и вовремя освобождать память. Использование контейнера *вектор* вместо динамического массива упрощает жизнь программиста, в чем вы могли убедиться на семинаре 11.

Для использования контейнера в программе необходимо включить в нее соответствующий заголовочный файл. Тип объектов, сохраняемых в контейнере, задается с помощью аргумента шаблона, например:

```
vector<int> aVect;           // создать вектор aVect целых чисел (типа int)
list<Man> department;       // создать список department из элементов типа Man
```

Ассоциативные контейнеры обеспечивают быстрый доступ к данным по ключу. Они построены на основе сбалансированных деревьев. Есть пять типов ассоциативных контейнеров: словари (*map*), словари с дубликатами (*multimap*), множества (*set*), множества с дубликатами (*multiset*) и битовые множества (*bitset*).

Итераторы

Чтобы понять, зачем нужны итераторы, давайте посмотрим, как можно реализовать шаблонную функцию для поиска значения *value* в обычном массиве, хранящем объекты типа *T*. Например, возможно следующее решение:

```
template <class T> T* Find( T* ar, int n, const T& value ) {
    for ( int i = 0; i < n; ++i ) if ( ar[i] == value ) return &ar[i];
    return 0;
}
```

Функция возвращает адрес найденного элемента или 0, если элемент с заданным значением не найден. Цикл *for* может быть записан и в несколько иной форме:

```
for ( int i = 0; i < n; ++i ) if ( *( ar + i ) == value ) return ar + i;
```

Работа функции при этом останется прежней. При продвижении по массиву адрес следующего элемента вычисляется и в первом, и во втором случаях с использованием арифметики указателей, то есть адреса соседних элементов различаются на количество байтов, требуемое для хранения одного элемента.

Попытаемся теперь расширить сферу применения нашей функции — хорошо бы, чтобы она решала задачу поиска заданного значения среди объектов, хранящихся в виде линейного списка! Однако, к сожалению, ничего не выйдет: адрес следующего элемента в списке нельзя вычислить, пользуясь арифметикой указателей.

Элементы списка могут размещаться в памяти самым причудливым образом, а информация об адресе следующего объекта хранится в одном из полей текущего объекта.

Авторы STL решили эту проблему, введя понятие *итератора* как более абстрактной сущности, чем указатель, но обладающей похожим поведением. Для всех контейнерных классов STL определен тип `iterator`, однако реализация его в разных классах разная. Например, в классе `vector`, где объекты размещаются один за другим, как в массиве, тип итератора определяется посредством `typedef T* iterator`. А вот в классе `list` тип итератора реализован как встроенный класс `iterator`, поддерживающий основные операции с итераторами.

К основным операциям, выполняемым с любыми итераторами, относятся:

- Разыменование итератора: если `p` — итератор, то `*p` — значение объекта, на который он ссылается.
- Присваивание одного итератора другому.
- Сравнение итераторов на равенство и неравенство (`==` и `!=`).
- Перемещение его по всем элементам контейнера с помощью префиксного (`++p`) или постфиксного инкремента (`p++`).

Так как реализация итератора специфична для каждого класса, при объявлении объектов типа «итератор» всегда указывается область видимости в форме `имя_шаблона::`, например:

```
vector<int>::iterator iter1;  
list<Man>::iterator iter2;
```

Организация циклов просмотра элементов контейнеров тоже имеет некоторую специфику. Так, если `i` — некоторый итератор, то вместо привычной формы

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

используется следующая:

```
for ( i = first; i != last; ++i )
```

где `first` — значение итератора, указывающее на первый элемент в контейнере, а `last` — значение итератора, указывающее на воображаемый элемент, который следует за последним элементом контейнера. Операция `<` заменена на операцию `!=`, поскольку операции `<` и `>` для итераторов в общем случае не поддерживаются.

Для всех контейнерных классов определены унифицированные методы `begin` и `end`, возвращающие адреса `first` и `last` соответственно.

Вообще, все типы итераторов в STL принадлежат одной из пяти категорий: входные, выходные, прямые, двунаправленные итераторы и итераторы произвольного доступа.

Входные итераторы (*InputIterator*) используются алгоритмами STL для чтения значений из контейнера, аналогично тому, как вводятся данные из потока `cin`. *Выходные итераторы* (*OutputIterator*) используются алгоритмами для записи значений в контейнер, аналогично тому, как выводятся данные в поток `cout`.

Прямые итераторы (*ForwardIterator*) используются алгоритмами для навигации по контейнеру только в прямом направлении, причем они позволяют и читать, и изменять данные в контейнере.

Двунаправленные итераторы (*BidirectionalIterator*) имеют все свойства прямых итераторов, но позволяют осуществлять навигацию по контейнеру и в прямом, и в обратном направлениях (для них дополнительно реализованы операции префиксного и постфиксного декремента).

Итераторы произвольного доступа (*RandomAccessIterator*) имеют все свойства двунаправленных итераторов плюс операции (наподобие сложения указателей) для доступа к произвольному элементу контейнера.

Значения прямых, двунаправленных и итераторов произвольного доступа могут быть сохранены, а вот значения входных и выходных итераторов сохраняться не могут (аналогично тому, как не может

быть гарантирован ввод тех же самых значений при вторичном обращении к потоку `cin`). Как следствие, любые алгоритмы, основанные на входных или выходных итераторах, должны быть однопроходными.

Вернемся к функции `Find`, которую мы безуспешно пытались обобщить для любых типов контейнеров. В STL аналогичный алгоритм имеет следующий прототип:

```
template <class InputIterator, class T>
InputIterator find( InputIterator first, InputIterator last, const T& value );
```

Для двунаправленных итераторов и итераторов произвольного доступа определены разновидности, называемые *адаптерами итераторов*. Адаптер, просматривающий последовательность в обратном направлении, называется `reverse_iterator`. Другие специализированные итераторы-адаптеры мы рассмотрим ниже.

Общие свойства контейнеров

В табл. 6.1 приведены имена типов, определенные с помощью `typedef` в большинстве контейнерных классов, а в табл. 6.2 — некоторые общие для всех контейнеров операции.

Таблица 6.1. Унифицированные типы, определенные в STL

Поле	Пояснение
<code>value_type</code>	Тип элемента контейнера
<code>size_type</code> ¹	Тип индексов, счетчиков элементов и т. д.
<code>iterator</code>	Итератор
<code>const_iterator</code>	Константный итератор (значения элементов изменять запрещено)
<code>reference</code>	Ссылка на элемент
<code>const_reference</code>	Константная ссылка на элемент (значение элемента изменять запрещено)
<code>key_type</code>	Тип ключа (для ассоциативных контейнеров)
<code>key_compare</code>	Тип критерия сравнения (для ассоциативных контейнеров)

Таблица 6.2. Операции и методы, общие для всех контейнеров

Операция или метод	Пояснение
Операции равенства (<code>==</code>) и неравенства (<code>!=</code>)	Возвращают значение <code>true</code> или <code>false</code>
Операция присваивания (<code>=</code>)	Копирует один контейнер в другой
<code>clear</code>	Удаляет все элементы
<code>insert</code>	Добавляет один элемент или диапазон элементов
<code>erase</code>	Удаляет один элемент или диапазон элементов
<code>size_type size()</code> <code>const</code>	Возвращает число элементов
<code>size_type max_size()</code> <code>const</code>	Возвращает максимально допустимый размер контейнера
<code>bool empty()</code> <code>const</code>	Возвращает <code>true</code> , если контейнер пуст
<code>iterator begin()</code>	Возвращают итератор на начало контейнера (итерации будут производиться в прямом направлении)
<code>iterator end()</code>	Возвращают итератор на конец контейнера (итерации в прямом направлении будут закончены)
<code>reverse_iterator begin()</code>	Возвращают реверсивный итератор на конец контейнера (итерации будут производиться в обратном направлении)

¹ Эквивалентен `unsigned int`.

<code>reverse_iterator</code> <code>end()</code>	Возвращают реверсивный итератор на начало контейнера (итерации в обратном направлении будут закончены)
---	--

Алгоритмы

Алгоритм — это функция, которая производит некоторые действия над элементами контейнера (контейнеров). Чтобы использовать обобщенные алгоритмы, нужно подключить к программе заголовочный файл `<algorithm>`. В табл. 6.3 приведены наиболее популярные алгоритмы STL.

Таблица 6.3. Некоторые типичные алгоритмы STL

Алгоритм	Назначение
<code>accumulate</code>	Вычисление суммы элементов в заданном диапазоне
<code>copy</code>	Копирование последовательности, начиная с первого элемента
Алгоритм	Назначение
<code>count</code>	Подсчет количества вхождений значения в последовательность
<code>count_if</code>	Подсчет количества выполнений условия в последовательности
<code>equal</code>	Попарное равенство элементов двух последовательностей
<code>fill</code>	Замена всех элементов заданным значением
<code>find</code>	Нахождение первого вхождения значения в последовательность
<code>find_first_of</code>	Нахождение первого значения из одной последовательности в другой
<code>find_if</code>	Нахождение первого соответствия условию в последовательности
<code>for_each</code>	Вызов функции для каждого элемента последовательности
<code>merge</code>	Слияние отсортированных последовательностей
<code>remove</code>	Перемещение элементов с заданным значением
<code>replace</code>	Замена элементов с заданным значением
<code>search</code>	Нахождение первого вхождения в первую последовательность второй последовательности
<code>sort</code>	Сортировка
<code>swap</code>	Обмен местами двух элементов
<code>transform</code>	Выполнение заданной операции над каждым элементом последовательности

В списках параметров всех алгоритмов первые два параметра задают диапазон обрабатываемых элементов в виде полуинтервала $[first, last)^2$, где `first` — итератор, указывающий на начало диапазона, а `last` — итератор, указывающий на выход за границы диапазона. Пример сортировки массива с помощью алгоритма `sort` (имя массива и `arr` имеет тип указателя `int*` и используется как итератор):

```
int arr[7] = { 15, 2, 19, -3, 28, 6, 8 };
sort( arr, arr + 7 );
```

Примеры использования некоторых алгоритмов будут даны ниже.

² Полуинтервал $[a, b)$ — это промежуток, включающий a , но не включающий b , — последний элемент полуинтервала предшествует элементу b .

Использование последовательных контейнеров

К основным последовательным контейнерам относятся *вектор* (vector), *список* (list) и *двусторонняя очередь* (deque). Чтобы использовать последовательный контейнер, нужно включить в программу соответствующий заголовочный файл: <vector>, <list> или <deque>.

Контейнер *вектор* является аналогом обычного массива, за исключением того, что он автоматически выделяет и освобождает память по мере необходимости. Контейнер эффективно обрабатывает произвольную выборку элементов с помощью операции индексации [] или метода at³. Однако вставка элемента в любую позицию, кроме конца вектора, неэффективна. Для этого потребуется сдвинуть все последующие элементы путем копирования их значений. По этой же причине неэффективным является удаление любого элемента, кроме последнего.

Контейнер *список* организует хранение объектов в виде двусвязного списка. Каждый элемент списка содержит три поля: значение элемента, указатель на предшествующий и указатель на последующий элементы списка. Вставка и удаление работают эффективно для любой позиции элемента в списке. Однако список не поддерживает произвольного доступа к своим элементам: например, для выборки n-го элемента нужно последовательно выбрать предыдущие n-1 элементов. Контейнер *двусторонняя очередь* (дек) во многом аналогичен вектору, элементы хранятся в непрерывной области памяти. Но в отличие от вектора дек эффективно поддерживает вставку и удаление первого элемента (так же, как и последнего).

Существует пять способов определить объект для последовательного контейнера.

1. Создать пустой контейнер:

```
vector<int> vec1;  
list<string> list1;
```

2. Создать контейнер заданного размера и инициализировать его элементы значениями по умолчанию:

```
vector<string> vec1( 100 );  
list<double> list1( 20 );
```

3. Создать контейнер заданного размера и инициализировать его элементы указанным значением:

```
vector<string> vec1( 100, "Hello!" );  
deque<int> dec1( 300, -1 );
```

4. Создать контейнер и инициализировать его элементы значениями диапазона [first, last) элементов другого контейнера:

```
int arr[7] = { 15, 2, 19, -3, 28, 6, 8 };  
vector<int> v1( arr, arr + 7 );  
list<int> lst(v1.begin() + 2, v1.end() );
```

5. Создать контейнер и инициализировать его элементы значениями элементов другого *однотипного* контейнера:

```
vector<int> v1;  
// ... здесь добавление элементов в v1  
vector<int> v2( v1 );
```

Для вставки и удаления последнего элемента контейнера любого из трех рассматриваемых классов предназначены методы push_back и pop_back. Кроме того, список и очередь (но не вектор) поддерживают операции вставки и удаления первого элемента контейнера push_front и pop_front. Учтите, что методы pop_back и pop_front *не возвращают* удаленное значение. Чтобы считать первый элемент, используется метод front, а для считывания последнего элемента — метод back. Кроме этого, все типы контейнеров имеют более общие операции вставки и удаления, перечисленные в табл. 6.4.

Таблица 6.4. Методы вставки в контейнер и удаления из контейнера

Метод	Пояснение
insert(iterator position, const T& value)	Вставка элемента со значением value в позицию, заданную итератором position

³ Метод at аналогичен операции индексации, но в отличие от нее проверяет выход за границу вектора и в случае ошибки генерирует исключение out_of_range.

<code>insert(iterator position, size_type n, const T& value)</code>	Вставка <code>n</code> элементов со значением <code>value</code> , начиная с позиции <code>position</code>
<code>template <class InputIter> void insert(iterator position, InputIter first, InputIter last)</code>	Вставка диапазона элементов, заданного итераторами <code>first</code> и <code>last</code> , начиная с позиции <code>position</code>
<code>erase(iterator position)</code>	Удаление элемента, на который указывает итератор <code>position</code>
<code>erase(iterator first, iterator last)</code>	Удаление диапазона элементов, заданного позициями <code>first</code> и <code>last</code>

Задача 6.1. Сортировка вектора

В файле находится произвольное количество целых чисел. Вывести их на экран в порядке возрастания.

Решение этой задачи приводится в листинге 6.1. Вместо вектора можно использовать любой последовательный контейнер, заменив слово `vector` на `deque` или `list`. При этом изменится внутреннее представление данных, а результат работы программы останется таким же.

Листинг 6.1. Сортировка вектора

```
#include <iostream>
#include <fstream>
#include <vector>
#include <algorithm>
using namespace std;
int main() {
    ifstream in ( "inpnum.txt" );
    if ( !in ) { cerr << "File not found\n"; exit( 1 ); }
    vector<int> v;
    int x;
    while ( in >> x ) v.push_back( x );
    sort( v.begin(), v.end() );
    vector<int>::const_iterator i;
    for ( i = v.begin(); i != v.end(); ++i ) cout << *i << " ";
}
```

Приведем еще один пример работы с векторами, демонстрирующий использование методов `swap`, `empty`, `back`, `pop_back`:

```
#include <iostream>
#include <vector>
using namespace std;
int main() {
    double arr[] = { 1.1, 2.2, 3.3, 4.4 };
    int n = sizeof( arr )/sizeof( double );
    vector<double> v1( arr, arr + n ); // Инициализация вектора массивом
    vector<double> v2;                // пустой вектор
    v1.swap( v2 );                    // обменять содержимое v1 и v2
    while ( !v2.empty() ) {
        cout << v2.back() << ' ';    // вывести последний элемент
        v2.pop_back();               // и удалить его
    }
    // Результат выполнения программы: 4.4 3.3 2.2 1.1
}
```

Шаблонная функция `print` для вывода содержимого контейнера

В процессе работы над программами, использующими контейнеры, часто приходится выводить на экран их текущее содержимое. Приведем шаблон функции, решающей эту задачу для любого типа контейнера (листинг 6.2).

Листинг 6.2. Шаблонная функция `print` для вывода содержимого контейнера

```
template <class T> void print( T& cont ) {
    typename T::const_iterator p = cont.begin();
    if ( cont.empty() ) cout << "Container is empty.";
    for ( p; p != cont.end(); ++p ) cout << *p << ' ';
    cout << endl;
}
```

Теперь можно пользоваться функцией `print`, включая ее определение в исходный файл с программой, как, например, в следующем эксперименте с очередью:

```

#include <iostream>
#include <deque>
using namespace std;
/* ... определение функции print ... */
int main() {
    deque<int> dec;      print( dec ); // Пустой контейнер
    dec.push_back( 4 );  print( dec ); // 4
    dec.push_front( 3 ); print( dec ); // 3 4
    dec.push_back( 5 );  print( dec ); // 3 4 5
    dec.push_front( 2 ); print( dec ); // 2 3 4 5
    dec.push_back( 6 );  print( dec ); // 2 3 4 5 6
    dec.push_front( 1 ); print( dec ); // 1 2 3 4 5 6
}

```

Адаптеры контейнеров

Специализированные последовательные контейнеры — *стек, очередь и очередь с приоритетами* — не являются самостоятельными контейнерными классами, а реализованы на основе рассмотренных выше классов, поэтому они называются *адаптерами контейнеров*.

Стек

Шаблонный класс `stack` (заголовочный файл `<stack>`) определен как

```

template <class T, class Container = deque<T> >
class stack { /* ... */ };

```

где параметр `Container` задает класс-прототип. По умолчанию для стека прототипом является класс `deque`. Смысл такой реализации заключается в том, что специализированный класс переопределяет интерфейс класса-прототипа, ограничивая его только методами, необходимыми новому классу. В табл. 6.5 показано, как сформирован интерфейс класса `stack` из методов класса-прототипа.

Таблица 6.5. Интерфейс класса `stack`

Методы класса <code>stack</code>	Методы класса-прототипа
<code>push</code>	<code>push_back</code>
<code>pop</code>	<code>pop_back</code>
<code>top</code>	<code>back</code>
<code>empty</code>	<code>empty</code>
<code>size</code>	<code>size</code>

В соответствии со своим назначением стек не только не позволяет выполнить произвольный доступ к своим элементам, но даже не дает возможности пошагового перемещения, в связи с чем концепция итераторов в стеке не поддерживается. Напоминаем, что метод `pop` не возвращает удаленное значение. Чтобы прочитать значение на вершине стека, используется метод `top`. Ниже приведен пример работы со стеком. Программа вводит из файла числа и выводит их в обратном порядке⁴:

```

int main() {
    ifstream in ( "inpnun.txt" );
    stack<int> s;
    int x;
    while ( in >> x ) s.push( x );
    while ( !s.empty() ) {
        cout << s.top() << ' ';
        s.pop();
    }
}

```

Объявление `stack<int> s` создает стек на базе двусторонней очереди (по умолчанию). Если это нас по каким-то причинам не устраивает и мы хотим создать стек на базе списка, то объявление будет выглядеть следующим образом: `stack<int, list<int> > s;` // Не забывайте о пробеле между угловыми скобками > >

Очередь

Шаблонный класс `queue` (заголовочный файл `<queue>`) является адаптером, который может быть реализован на основе двусторонней очереди (по умолчанию) или списка. Класс `vector` в качестве

⁴ В дальнейших примерах мы будем опускать директивы `#include` и объявление `using namespace std` для экономии места, то есть стоимости книги.

прототипа не подходит, так как в нем нет выборки из начала контейнера. Очередь использует для проталкивания данных один конец, а для выталкивания — другой. В соответствии с этим ее интерфейс образуют методы из табл. 6.6.

Таблица 6.6. Интерфейс класса queue

Методы класса queue	Методы класса-прототипа
push	push_back
pop	pop_front
front	front
back	back
empty	empty
size	size

Очередь с приоритетами

Шаблонный класс `priority_queue` (заголовочный файл `<queue>`) поддерживает такие же операции, что и класс `queue`, но реализация класса возможна либо на основе вектора (реализация по умолчанию), либо на основе списка. *Очередь с приоритетами* отличается от обычной тем, что для извлечения выбирается максимальный элемент из хранимых в контейнере. Поэтому после каждого изменения состояния очереди максимальный элемент из оставшихся сдвигается в начало контейнера. Если очередь с приоритетами организуется для объектов класса, определенного программистом, в этом классе следует определить операцию `<`. Пример работы с очередью с приоритетами приведен в листинге 6.3.

Листинг 6.3. Пример работы с очередью с приоритетами

```
#include <queue>
#include <iostream>
using namespace std;
int main() {
    priority_queue<int> P;
    P.push( 17 ); P.push( 5 ); P.push( 400 ); P.push( 2500 );
    P.push( 1 );
    while ( !P.empty() ) {
        cout << P.top() << ' ';
        P.pop();
    }
    // Результат выполнения программы: 2500 400 17 5 1
}
```

Использование алгоритмов

Вернемся к изучению алгоритмов. Не забывайте включать заголовочный файл `<algorithm>` и добавлять определение функции `print`, если она используется.

Алгоритмы `count` и `find`

Алгоритм `count` подсчитывает количество вхождений в контейнер (или его часть) значения, заданного его третьим аргументом. Алгоритм `find` выполняет поиск заданного значения и возвращает итератор на самое первое вхождение этого значения. Если значение не найдено, возвращается итератор, соответствующий возврату метода `end`. В листинге 6.4 показано использование этих алгоритмов.

Листинг 6.4. Пример использования алгоритмов `count` и `find` int

```
#include <string>
#include <iostream>
#include <list>
#include <bits/stdc++.h>
using namespace std;
template <class T> void print( T& cont ) {
    typename T::const_iterator p = cont.begin();
    if ( cont.empty() ) cout << "Container is empty.";
    for ( p; p != cont.end(); ++p ) cout << *p << ' ';
    cout << endl;
}
int main() {
    int arr[] = { 1, 2, 3, 4, 5, 2, 6, 2, 7 };
    int n = sizeof( arr ) / sizeof( int );
    vector<int> v1( arr, arr + n );
}
```

```

int value = 2; // искомая величина
int how_much = count( v1.begin(), v1.end(), value ); // 1
cout << how_much << endl; // вывод: 3
list<int> loc_list; // список позиций искомой величины
vector<int>::iterator location = v1.begin();
while ( 1 ) {
    location = find( location, v1.end(), value ); // 2
    if ( location == v1.end() ) break;
    loc_list.push_back( location - v1.begin() );
    location++;
}
print( loc_list ); // функция из листинга 15.2 // вывод: 1 5 7
}

```

Вектор `v1` создается, инициализируясь значениями массива `arr`. Затем с помощью алгоритма `count` подсчитывается количество вхождений в вектор значения `value`, равного двум. В цикле `while` выясняется, в каких позициях вектора размещена эта величина. Первый аргумент алгоритма `find` (переменная `location`) первоначально, перед входом в цикл, принимает значение итератора, указывающего на нулевой элемент контейнера (элементы вектора нумеруются с нуля). Затем `location` получает значение итератора, указывающего на найденный элемент.

Если поиск завершился успешно, то, во-первых, вычисляется позиция найденного элемента как разность значений `location` и адреса нулевого элемента и полученное значение заносится в список `loc_list`. Во-вторых, итератор `location` сдвигается операцией инкремента на следующую позицию в контейнере, чтобы обеспечить на следующей итерации цикла продолжение поиска в оставшейся части контейнера. Если поиск завершился неудачей, то `break` приведет к выходу из цикла.

Алгоритмы `count_if` и `find_if`

Алгоритмы `count_if` и `find_if` отличаются от алгоритмов `count` и `find` тем, что их третьим аргументом является некоторый *предикат*. *Предикат* — это функция или функциональный объект, возвращающие значение типа `bool`. Например, если в листинг 6.4 добавить определение глобальной функции `bool isMyValue( int x ) { return ( ( x > 2 ) && ( x < 5 ) ); }` и заменить инструкцию с вызовом `count` (оператор, помеченный цифрой 1) на:

```
int how_much = count_if( v1.begin(), v1.end(), isMyValue );
```

то программа определит, что контейнер содержит два числа, значение которых больше двух, но меньше пяти.

Аналогично, замена инструкции с вызовом `find` (оператор 2) на:

```
location = find_if( location, v1.end(), isMyValue );
```

приведет к наполнению списка `loc_list` значениями 2 и 3 (номера позиций вектора `v1`, в которых находятся числа, удовлетворяющие предикату `isMyValue`).

Алгоритм `for_each`

Этот алгоритм позволяет выполнить некоторое действие над каждым элементом диапазона `[first, last)`. Чтобы определить, какое именно действие должно быть выполнено, нужно написать соответствующую функцию с одним аргументом типа `T` (`T` — тип данных, содержащихся в контейнере). Функция не имеет права модифицировать данные в контейнере, но может их использовать в своей работе. Имя этой функции передается алгоритму третьим аргументом. Например, в следующей программе `for_each` используется для перевода всех значений массива из дюймов в сантиметры и вывода их на экран:

```

void InchToCm( double inch ) { cout << ( inch * 2.54 ) << ' '; }
int main() {
    double inches[] = {0.5, 1.0, 1.5, 2.0, 2.5};
    for_each( inches, inches + 5, InchToCm );
}

```

Алгоритм `search`

Некоторые алгоритмы оперируют одновременно двумя контейнерами. Таков и алгоритм `search`, находящий первое вхождение в первую последовательность `[first1, last1)` второй последовательности `[first2, last2)`. Пример приведен в листинге 6.5.

Листинг 6.5. Пример использования алгоритма `search`

```

#include <string>
#include <iostream>

```

```

#include <list>
#include <bits/stdc++.h>
using namespace std;
int main() {
    int arr[] = { 11, 77, 33, 11, 22, 33, 11, 22, 55 };
    int pattern[] = { 11, 22, 33 };
    int* ptr = search( arr, arr + 9, pattern, pattern + 3 );
    if ( ptr == arr + 9 ) cout << "Pattern not found" << endl;
    else cout << "Found at position " << ( ptr - arr ) << endl;
    list<int> lst( arr, arr + 9 );
    list<int>::iterator ifound;
    ifound = search( lst.begin(), lst.end(), pattern, pattern + 3 );
    if ( ifound == lst.end() ) cout << "Pattern not found" << endl;
    else cout << "Found." << endl;
}

```

Результат выполнения программы:

Found at position 3

Found.

Отметим, что список не поддерживает произвольного доступа к своим элементам, а значит, не допускает операций "+" и "-" с итераторами. Поэтому мы можем только зафиксировать факт вхождения последовательности *pattern* в контейнер *lst*.

Алгоритм *sort*

Алгоритм сортировки можно применять только для тех контейнеров, которые обеспечивают произвольный доступ к элементам, — этому требованию удовлетворяют *массив*, *вектор* и *двусторонняя очередь*, но не удовлетворяет *список*. В связи с этим класс *list* содержит метод *sort*, решающий задачу сортировки.

Алгоритм *sort* имеет две сигнатуры:

```

template<class RandomAccessIt>
void sort( RandomAccessIt first, RandomAccessIt last );
template<class RandomAccessIt>
void sort( RandomAccessIt first, RandomAccessIt last, Compare comp );

```

Первая форма алгоритма обеспечивает сортировку элементов из диапазона $[first, last)$, причем для упорядочения по умолчанию используется операция $<$, которая должна быть определена для типа *T* (тип данных, содержащихся в контейнере). То есть сортировка по умолчанию выполняется по возрастанию значений, например:

```

int main() {
    double arr[6] = { 2.2, 0.0, 4.4, 1.1, 3.3, 1.1 };
    vector<double> v1( arr, arr + 6 );
    sort( v1.begin(), v1.end() );
    print( v1 ); // Функция из листинга 15.2
} // Результат выполнения программы: 0 1.1 1.1 2.2 3.3 4.4

```

Вторая форма алгоритма позволяет задать произвольный критерий упорядочения. Для этого нужно передать через третий аргумент соответствующий предикат, то есть функцию или функциональный объект, возвращающие значение типа *bool*. Использование функции в качестве предиката было показано при описании алгоритмов *count_if* и *find_if*. Использованию функциональных объектов посвящен следующий раздел.

Функциональные объекты

На семинаре 12 было показано, как можно использовать функциональные объекты для настройки шаблонных классов, поэтому рекомендуем вам еще раз просмотреть этот материал. *Функциональным объектом* называется объект некоторого класса, для которого определена единственная операция вызова функции *operator()*. В стандартной библиотеке определены шаблоны функциональных объектов для операций сравнения, встроенных в язык C++. Они возвращают значение типа *bool*, то есть являются предикатами (табл. 6.7).

Таблица 6.7. Предикаты стандартной библиотеки

Операция	Эквивалентный предикат (функциональный объект)	Операция	Эквивалентный предикат (функциональный объект)
==	<code>equal_to</code>	<	<code>less</code>

Очевидно, что при подстановке в качестве аргумента алгоритма требуется инстанцирование этих шаблонов, например: `equal_to<int>()`.

```
sort( v1.begin(), v1.end(), greater<double>() );
```

В листинге 6.6 показаны варианты вызова алгоритма `sort` для вектора `men`, содержащего объекты класса `Man`. В классе `Man` определен предикат (операция `operator<()`), благодаря которому сортировка по умолчанию будет происходить по возрастанию значений поля `name`. Кроме этого, определен функциональный класс `LessAge`, использование которого позволяет выполнить сортировку по возрастанию значений поля `age`.

[illegible]

Обратные итераторы

Эта разновидность итераторов (`reverse_iterator`) удобна для прохода по контейнеру от конца к началу. Например, для вывода содержимого контейнера `vector<double> v1` в обратном порядке можно написать

```
vector<double>::reverse_iterator ri;
ri = v1.rbegin();
while ( ri != v1.rend() ) cout << *ri++ << ' ';
```

Операция инкремента для такого итератора перемещает указатель на предыдущий элемент контейнера.

Итераторы вставки и алгоритм `copy`

Мы можем использовать алгоритм `copy` для копирования элементов одного контейнера в другой, причем источником может быть, например, вектор, а приемником — список, как показывает следующий фрагмент:

```
int main() {
    int a[4] = { 10, 20, 30, 40 };
    vector<int> v( a, a + 4 );
    list<int> L( 4 ); // список из 4 элементов
    copy( v.begin(), v.end(), L.begin() );
    print( L ); // функция из листинга 15.2
}
```

Алгоритм `copy` при таком использовании, как в этом примере, работает в *режиме замещения*. Это означает, что *i*-й элемент контейнера-источника замещает *i*-й элемент контейнера-приемника (это напоминает режим замены при вводе текста в текстовом редакторе). Этот же алгоритм может работать и в *режиме вставки*, если третьим аргументом использовать так называемый *итератор вставки*.

Итераторы вставки `front_inserter`, `back_inserter`, `inserter` предназначены для добавления новых элементов в начало, конец или произвольное место контейнера. Покажем использование этих итераторов на следующем примере.

```
int main() {
    int a[4] = { 40, 30, 20, 10 }; vector<int> va( a, a + 4 );
    int b[3] = { 80, 90, 100 }; vector<int> vb( b, b + 3 );
    int c[3] = { 50, 60, 70 }; vector<int> vc( c, c + 3 );
    list<int> L; // пустой список
    copy( va.begin(), va.end(), front_inserter(L) ); // 1
    print( L ); // функция из листинга 15.2
    copy( vb.begin(),
vb.end(), back_inserter(L) ); // 2
    print( L );
    list<int>::iterator from = L.begin();
    advance( from, 4 );
    copy( vc.begin(), vc.end(), inserter( L, from) ); // 3
    print( L );
}
```

Результат выполнения программы:

```
10 20 30 40
10 20 30 40 80 90 100
10 20 30 40 50 60 70 80 90 100
```

В *операторе 1* выполняется копирование (вставка) вектора `va` в список `L`. Итератор вставки `front_inserter` обеспечивает вставку очередного элемента вектора `va` в начало списка, поэтому порядок элементов в списке изменяется на обратный.

В *операторе 2* элементы вектора `vb` пересылаются в конец списка `L` благодаря итератору вставки `back_inserter` (порядок копируемых элементов не меняется). В *операторе 3* вектор `vc` копируется в заданное итератором `from` место списка `L`, а именно, после четвертого элемента списка. Чтобы определить нужное значение итератора `from`, мы предварительно устанавливаем его в начало списка, а затем обеспечиваем приращение на 4 вызовом функции `advance`.

Алгоритм `merge`

Алгоритм `merge` выполняет слияние отсортированных последовательностей для любого типа последовательного контейнера, более того — все три участника алгоритма могут представлять различные контейнерные типы. Например, вектор `a` и массив `b` могут быть слиты в список `c`:

```
int main() {
    int arr[5] = { 2, 3, 8, 20, 25 };
```

```

vector<int> a( arr, arr + 5 );
int b[6] = { 7, 9, 23, 28, 30, 33 };
list<int> c; // Список c сначала пуст
merge( a.begin(), a.end(), b, b + 6, back_inserter(c) );
print( c ); // функция из листинга 6.2
} // Результат выполнения программы: 2 3 7 8 9 20 23 25 28 30 33

```

Использование ассоциативных контейнеров

В ассоциативных контейнерах элементы не выстроены в линейную последовательность. Они организованы в более сложные структуры, что дает большой выигрыш в скорости поиска. Поиск выполняется с помощью *ключа*, обычно представляющего собой одно число или строку. Рассмотрим две основные категории ассоциативных контейнеров в STL: множества и словари (отображения).

В *множестве* (set) хранятся объекты, упорядоченные по некоторому ключу, являющемуся атрибутом самого объекта. Например, множество может хранить объекты класса Man, упорядоченные в алфавитном порядке по значению ключевого поля name. Если в множестве хранятся значения одного из встроенных типов, например int, то ключом является сам элемент.

Словарь (map) можно представить себе как своего рода таблицу из двух столбцов, в первом из которых хранятся объекты, содержащие ключи, а во втором — объекты, содержащие значения. Похожая организация данных рассматривалась нами в задаче 3.1 (шаблонный класс для разреженных массивов).

И в множествах, и в словарях все ключи являются уникальными (только одно значение соответствует ключу). *Мультимножества* (multiset) и *мультисловари* (multimap) аналогичны своим родственным контейнерам, но в них одному ключу может соответствовать несколько значений.

Ассоциативные контейнеры имеют много общих методов с последовательными контейнерами, но некоторые методы и алгоритмы характерны только для них.

Множества

Шаблон множества имеет два параметра: тип ключа и тип функционального объекта, определяющего отношение «меньше»:

```

template <class Key, class Compare = less<Key> >
class set{ /* ... */ };

```

Таким образом, если объявить некоторое множество set<int> s1 с опущенным вторым параметром шаблона, то по умолчанию для упорядочения членов множества будет использован предикат less<int>. Точно так же можно опустить второй параметр при объявлении множества set<MyClass> s2, если в классе MyClass определена операция operator<()>.

Для использования контейнеров типа set необходимо подключить заголовочный файл <set>. Имеется три способа определить объект типа set:

```

set<int> set1; // создается пустое множество
int a[5] = { 1, 2, 3, 4, 5 };
set<int> set2( a, a + 5 ); // инициализация копированием массива set<int> set3( set2 );
// инициализация другим множеством

```

Для вставки элементов в множество можно использовать метод insert, для удаления — метод erase. Также к множествам применимы общие для всех контейнеров методы, указанные в табл. 6.2.

Во всех ассоциативных контейнерах есть метод count, возвращающий количество объектов с заданным ключом. Так как и в множествах, и в словарях все ключи уникальны, то метод count возвращает либо 0, если элемент не обнаружен, либо 1. Для множеств библиотека содержит некоторые специальные алгоритмы, в частности, реализующие традиционные теоретико-множественные операции. Эти алгоритмы перечислены ниже.

Алгоритм includes выполняет проверку включения одной последовательности в другую. Результат равен true в случае, когда каждый элемент последовательности [first2, last2) содержится в последовательности [first1, last1).

Алгоритм set_intersection создает отсортированное *пересечение множеств*, то есть множество, содержащее только те элементы, которые одновременно входят и в первое, и во второе множество.

Алгоритм set_union создает отсортированное *объединение множеств*, то есть множество, содержащее элементы первого и второго множеств без повторяющихся элементов. Использование этих алгоритмов показано в листинге 6.7.

Листинг 6.7. Пример работы с множествами

```

#include <string>
#include <iostream>
#include <list>
#include <bits/stdc++.h>
using namespace std;
template <class T> void print( T& cont ) {
    typename T::const_iterator p = cont.begin();
    if ( cont.empty() ) cout << "Container is empty.";
    for ( p; p != cont.end(); ++p ) cout << *p << ' ';
    cout << endl;
}
int main() {
    const int N = 5;
    string s1[N] = { "Bill", "Jessica", "Ben", "Mary", "Monica"};
    string s2[N] = { "Sju", "Monica", "John", "Bill", "Sju" };
    typedef set<string> SetS;
    SetS A( s1, s1 + N );
    SetS B( s2, s2 + N );
    print( A ); print( B );
    SetS prod, sum;
    set_intersection( A.begin(), A.end(), B.begin(), B.end(),
                     inserter( prod, prod.begin() ) );
    print( prod ); // функция из листинга 6.2
    set_union( A.begin(), A.end(), B.begin(), B.end(),
              inserter( sum, sum.begin() ) );
    print( sum );
    if ( includes( A.begin(), A.end(), prod.begin(), prod.end() ) )
        cout << "Yes" << endl;
    else cout << "No" << endl;
}

```

Результат выполнения программы:

```

Ben Bill Jessica Mary Monica
Bill John Monica Sju
Bill Monica
Ben Bill Jessica John Mary Monica Sju
Yes

```

Словари

В определении класса map используется тип pair, который описан в заголовочном файле <utility> следующим образом:

```

template <class T1, class T2> struct pair{
    T1 first;
    T2 second;
    pair( const T1& x, const T2& y );
    . . .
};

```

Шаблон pair имеет два параметра, представляющих собой типы элементов пары. Первый элемент пары имеет имя first, второй — second. В этом же файле определены шаблонные операции ==, !=, <, >, <=, >= для двух объектов типа pair.

Шаблон словаря имеет три параметра: тип ключа, тип элемента и тип функционального объекта, определяющего отношение «меньше»:

```

template <class Key, class T, class Compare = less<Key> >
class map {
public:
    typedef pair <const Key, T> value_type;
    explicit map ( const Compare& comp = Compare() );
    map( const value_type* first, const value_type* last,
         const Compare& comp = Compare() );
    map( const map <Key, T, Compare>& x );
    . . .
};

```

Тип элементов словаря value_type определяется как пара элементов типа Key и T. Первый конструктор класса map создает пустой словарь. Второй — создает словарь и записывает в него

элементы, определяемые диапазоном `[first, last)`. Третий конструктор является конструктором копирования.

Для доступа к элементам по ключу определена операция `[ ]`:

```
T& operator[]( const Key & x );
```

С помощью нее можно не только получать значения элементов, но и добавлять в словарь новые.

Для использования контейнеров типа `map` необходимо подключить заголовочный файл `<map>`.

Задача 6.2. Формирование частотного словаря

Написать программу формирования частотного словаря появления отдельных слов в некотором тексте. Исходный текст читается из файла `prose.txt`, результат (частотный словарь) записывается в файл `freq_map.txt`.

Текст программы приведен в листинге 6.8.

Листинг 6.8. Формирование частотного словаря

```
#include <iostream>
#include <fstream>
#include <iomanip>
#include <map>
#include <set>
#include <string>
using namespace std;
int main() {
    char punct[6] = { '.', ',', '?', '!', ':', ';' };
    set<char> punctuation( punct, punct + 6 );
    ifstream in( "prose.txt" );
    if ( !in ) { cerr << "File not found\n"; exit( 1 ); }
    map<string, int> wordCount;
    string s;
    while ( in >> s ) {
        int n = s.size();
        if ( punctuation.count( s[n - 1] ) ) s.erase( n - 1, n );
        ++wordCount[s];
    }
    ofstream out( "freq_map.txt" );
    map<string, int>::const_iterator it = wordCount.begin();
    for ( it; it != wordCount.end(); ++it )
        out << setw( 20 ) << left << it->first
            << setw( 4 ) << right << it->second << endl;
}
```

Определяя объект `wordCount` как словарь типа `map<string, int>`, мы показываем намерение связать каждое прочитанное слово с целочисленным счетчиком. В цикле `while` разворачиваются следующие события.

- В строку `s` пословно считываются данные из входного файла.
- Определяется длина `n` строки `s`.
- С помощью метода `count` проверяется, принадлежит ли последний символ строки `s` множеству `punctuation`, содержащему знаки препинания, которыми может завершаться слово. Если да, то последний символ удаляется из строки (метод `erase`).
- Заслуживает особого внимания лаконичная инструкция `++wordCount[s]`. Здесь мы как бы «заглядываем» в объект `wordCount`, используя только что считанное слово в качестве ключа. Результат выражения `wordCount[s]` представляет собой некоторое целочисленное значение, обозначающее, сколько раз слово `s` уже встречалось ранее. Затем операция инкремента увеличивает это целое значение на единицу. А что будет, если мы встречаем некоторое слово в первый раз? Если в словаре нет элемента с таким ключом, он будет создан с инициализацией поля типа `int` значением по умолчанию, то есть нулем. Следовательно, после операции инкремента это значение будет равно единице.

Завершив считывание входных данных и формирование словаря `wordCount`, мы должны вывести в выходной файл `freq_map.txt` значения обнаруженных слов и соответствующих им счетчиков. Вывод результатов реализуется здесь практически так же, как и для последовательных контейнеров — с помощью соответствующего итератора. Однако есть тонкость, связанная с тем, что при разыменовании итератора `map`-объекта мы получаем значение, которое имеет тип `pair`, соответствующий данному `map`-

объекту. Так как `pair` — это структура, доступ к ее полям через «указатель» `it` выполняется посредством выражений `it->first`, `it->second`.

Рассмотрим теперь более сложную задачу, чтобы продемонстрировать применение принципов ООП на практике и удобства, которые дает использование STL.

Задача 6.3. Морской бой

Написать программу, реализующую упрощенную версию игры «Морской бой» между двумя игроками: пользователь и компьютер. Упрощения: все корабли размещаются только вертикально; размещение кораблей — случайное у обоих игроков.

Тем, у кого было тяжелое детство, израненное компьютером, напомним правила:

- Имеется два игровых поля: «свое» и «противника», каждое 10×10 клеток. У каждого игрока по 10 кораблей: один четырехпалубный (состоящий из четырех клеток), два трехпалубных (из трех клеток), три двухпалубных (из двух), четыре однопалубных (из одной клетки). При расстановке кораблей они не должны касаться друг друга (находиться в соседних клетках).
- Каждый игрок видит размещение кораблей на своем игровом поле, но не имеет информации о размещении кораблей на поле противника.
- После расстановки кораблей игроки начинают «стрелять» друг в друга. Для этого стреляющий выбирает клетку на поле противника и объявляет ему ее координаты (A1, E5 и т. д.). Противник смотрит на своем поле, находится ли по указанным координатам его корабль, и сообщает результат выстрела:
 - *промах* — на данной клетке нет корабля противника;
 - *ранен (поврежден)* — на данной клетке есть корабль противника с хотя бы еще одной непораженной клеткой (палубой);
 - *убит* — на данной клетке есть корабль противника, и все его клетки (палубы) уже поражены.
- В случае попадания в корабль противника игроку дается право на внеочередной выстрел, в противном случае ход переходит к противнику.
- Стрельба ведется до тех пор, пока у одного из игроков не окажутся «убитыми» все корабли (он признается проигравшим, а его противник — победителем). Так как мы пишем консольное приложение, доступные нам графические средства сильно ограничены — это текстовые символы и символы псевдографики. Примем решение, что после некоторого хода играющих картинка в консольном окне будет иметь примерно такой вид, какой показан на рис.6.1.

Изображенные на рисунке игровые поля «Мой флот» и «Флот неприятеля» отображают текущее состояние игры со стороны пользователя. Изначальное размещение кораблей на поле пользователя — в клетках, помеченных символом «заштрихованный прямоугольник» (символ 176 в кодовой таблице `sr866/MS DOS`). Символом «.» (точка) обозначены свободные клетки, по которым еще ни разу не стреляли, символом «o» — промахи стреляющих, символом «X» — пораженные клетки (палубы) кораблей. Пробелами обозначены клетки, в которых по правилам уже не могут находиться корабли противника. Эти «мертвые зоны» выявляются после гибели очередного корабля.



Рис. 6.1. Возможный вид консольного окна после i -го хода ($i = 31$)

После сделанных разъяснений мы можем приступить к решению задачи, и начнем, как всегда, с выявления понятий (классов) и их взаимосвязей.

Итак, имеется два игрока: пользователь (User) и компьютер, выступающий в роли робота (Robot). Каждый игрок «управляет» своим собственным флотом, поэтому логично создать два класса: UserNavy (флот пользователя) и RobotNavy (флот робота). Очевидно, что они обладают различным поведением — например, метод FireOff (выстрел по неприятелю) в первом классе должен пригласить пользователя ввести координаты выстрела, а во втором классе — автоматически сформировать координаты выстрела, сообразуясь с искусственным интеллектом робота. В то же время в этих классах есть и общие атрибуты, например, игровые поля (свое и неприятеля), корабли своего флота и т. д. Поэтому выделим все общие поля и методы в базовый класс Navy, который будут наследовать классы UserNavy и RobotNavy.

Каждый флот состоит из кораблей, отсюда вытекает потребность в классе Ship, объекты которого инкапсулируют такую информацию, как координаты размещения корабля, имя корабля, общее количество палуб, количество неповрежденных палуб.

Для описания размещения кораблей воспользуемся классом Rect, который позволяет задать любой прямоугольник в двумерном дискретном пространстве. Конечно, наши прямоугольники в формализме двумерного дискретного пространства вырождаются в линию, но такое описание удобно для единообразного представления как вертикально, так и горизонтально размещенных кораблей⁵.

Игровое поле (двумерное дискретное пространство) состоит из клеток (точек двумерного пространства), для представления которых используем класс Cell. Наконец, игроки должны обмениваться информацией (координаты и результаты очередного выстрела). Для моделирования процесса обмена информацией создадим класс Space, поля которого будут использоваться как глобальные переменные, поэтому они должны быть статическими, а сам класс — базовым для класса Navy.

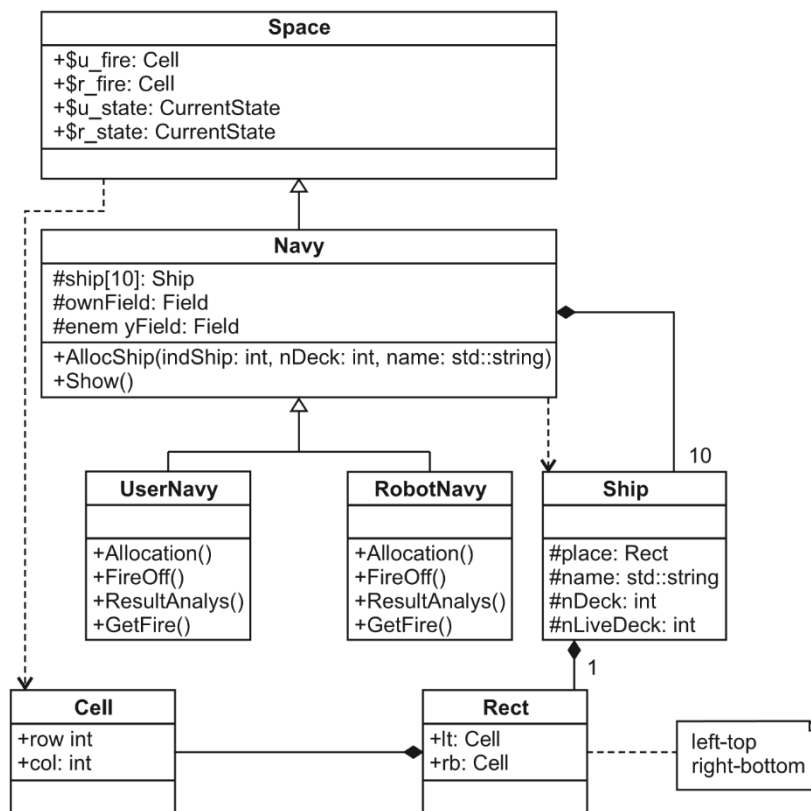


Рис. 6.2. Диаграмма классов для задачи 6.3

На рис. 6.2 показана диаграмма классов, обобщающая наши рассуждения по составу и взаимоотношениям классов для решения рассматриваемой задачи. Текст программы приводится в листинге 6.9.

Листинг 6.9. «Морской бой»

```

//////////////////////////////////// Ship.h
#ifndef SHIP_H

```

⁵ Хотя условие имеет упрощающее ограничение «все корабли размещаются вертикально», при проектировании структур данных рекомендуется учитывать возможность последующей модификации программы, снимающей это ограничение.

```

#define SHIP_H
#include <set>
#include <map>
#include <string>
#define N 10 // размер поля для размещения флота (N * N клеток)
struct Cell;
typedef std::set<Cell> CellSet; // множество клеток
struct Cell { // ----- Клетка (ячейка) на игровом поле
    Cell( int _r = 0, int _c = 0 ) : row( _r ), col( _c ) {}
    bool InSet( const CellSet& ) const; // определяет
        // принадлежность клетки множеству типа CellSet
    bool operator<( const Cell& ) const;
    int row; // ряд
    int col; // колонка
};
struct Rect { // --- Прямоугольная область (размещение кораблей и их "оболочек")
    Rect() {}
    Rect( Cell _lt, Cell _rb ) : lt( _lt ), rb( _rb ) { FillCset(); }
    void FillCset(); // заполнить cset клетками данной области
    bool Intersect( const CellSet& cs ) const; // определить наличие
        // непустого пересечения прямоугольника с множеством cs
    Cell lt; // left-top клетка
    Cell rb; // right-bottom клетка
    CellSet cset; // множество клеток, принадлежащих прямоугольнику
};
class Ship { // ----- Класс Ship (для представления корабля)
    friend class UserNavy;
    friend class RobotNavy;
public:
    Ship() : nDeck( 0 ), nLiveDeck( 0 ) {}
    Ship( int, std::string, Rect );
protected:
    Rect place; // координаты размещения
    std::string name; // имя корабля
    int nDeck; // количество палуб
    int nLiveDeck; // количество неповрежденных палуб
};
#endif /* SHIP_H */
////////////////////////////////////////////////////////////// Ship.cpp
#include <string>
#include <algorithm>
#include "Ship.h"
using namespace std;
bool Cell::InSet( const CellSet& cs ) const { //----- Класс Cell
    return ( cs.count( Cell( row, col ) ) > 0 );
}
bool Cell::operator<( const Cell& c ) const {
    return ( ( row < c.row ) || ( ( row == c.row ) && ( col < c.col ) ) );
}
void Rect::FillCset() { // ----- Класс Rect
    for ( int i = lt.row; i <= rb.row; i++ )
        for ( int j = lt.col; j <= rb.col; j++ )
            cset.insert( Cell( i, j ) );
}
bool Rect::Intersect( const CellSet& cs ) const {
    CellSet common_cell;
    set_intersection( cset.begin(), cset.end(), cs.begin(), cs.end(),
        inserter( common_cell, common_cell.begin() ) );
    return ( common_cell.size() > 0 );
}
Ship::Ship( int _nDeck, string _name, Rect _place ) : // Класс Ship
    place( _place ), name( _name ), nDeck( _nDeck ), nLiveDeck( _nDeck ) {}
////////////////////////////////////////////////////////////// Navy.h
#include "Ship.h"
#define DECK 176 // исправная клетка-палуба
#define DAMAGE 'X' // разрушенная клетка-палуба
#define MISS 'o' // пустая клетка, в которую упал снаряд

```

```

typedef unsigned char Field[N][N];          // игровое поле
typedef std::map<Cell, int> ShipMap;         // словарь
// ассоциаций "клетка - индекс корабля"
enum CurrentState { Miss, Damage, Kill }; // результат попадания в цель
struct Space { // -- Класс Space - информационное пространство для обмена
public:
    // информацией между игроками
    static Cell u_fire;          // огонь от пользователя
    static Cell r_fire;          // огонь от робота (компьютера)
    static CurrentState u_state; // состояние пользователя
    static CurrentState r_state; // состояние робота
    static int step;
};
class Navy : public Space { //----- Базовый класс Navy
public:
    Navy();
    void AllocShip( int, int, std::string ); // разместить корабль
    void Show() const;          // показать поля ownField и enemyField
    int GetInt();                // ввод целого числа
    bool IsLive() { return ( nLiveShip > 0 ); } // мы еще живы?
    Rect Shell( Rect ) const; /* вернуть "оболочку" для заданного
                               прямоугольника
    (сам прямоугольник плюс пограничные клетки) */
    void AddToVetoSet( Rect ); // добавить клетки прямоугольника
                               // в множество vetoSet.
protected:
    Ship ship[10];              // корабли флота
    Field ownField;              // мое игровое поле
    Field enemyField;           // игровое поле неприятеля
    ShipMap shipMap;            // словарь ассоциаций "клетка - индекс корабля"
    CellSet vetoSet;            // множество "запрещенных" клеток
    CellSet crushSet;           // множество "уничтоженных" клеток
    int nLiveShip;              // количество боеспособных кораблей
};
class UserNavy : public Navy { //----- Класс UserNavy
public:
    UserNavy() { Allocation(); }
    void Allocation();
    void FireOff();              // выстрел по неприятелю
    void ResultAnalys();         // анализ результатов выстрела
    void GetFire();              // "прием" огня противника
    void FillDeadZone( Rect r, Field& ); // заполнить
                                     // пробелами пограничные клетки для r
};
class RobotNavy : public Navy { //----- Класс RobotNavy
public:
    RobotNavy();
    void Allocation();
    void FireOff();              // выстрел по неприятелю
    void ResultAnalys();         // анализ результатов выстрела
    void GetFire();              // "прием" огня противника
private:
    bool isCrushContinue; // предыдущий выстрел был успешным
    bool upEmpty;         // у поврежденного корабля противника
                          // нет "живых" клеток в верхнем направлении
};
//////////////////////////////////// Navy.cpp
#include <iostream>
#include <cstdlib>
#include <time.h>
#include <algorithm>
#include "Navy.h"
using namespace std;
Cell Space::u_fire;
Cell Space::r_fire;
CurrentState Space::u_state = Miss;
CurrentState Space::r_state = Miss;
int Space::step = 1;

```

```

// Функция gap(n) возвращает строку из n пробелов
string gap( int n ) { return string( n, ' ' ); }
Navy::Navy() : nLiveShip( 10 ) { // ----- Класс Navy
    for ( int i = 0; i < N; i++ ) // Заполняем игровые поля символом "точка"
        for ( int j = 0; j < N; j++ ) {
            ownField[i][j] = '.';
            enemyField[i][j] = '.'; }
}
Rect Navy::Shell( Rect r ) const {
    Rect sh( r );
    sh.lt.row = ( --sh.lt.row < 0 ) ? 0 : sh.lt.row;
    sh.lt.col = ( --sh.lt.col < 0 ) ? 0 : sh.lt.col;
    sh.rb.row = ( ++sh.rb.row > ( N - 1 ) ) ? ( N - 1 ) : sh.rb.row;
    sh.rb.col = ( ++sh.rb.col > ( N - 1 ) ) ? ( N - 1 ) : sh.rb.col;
    return sh;
}
void Navy::AddToVetoSet( Rect r ) {
    for ( int i = r.lt.row; i <= r.rb.row; i++ )
        for ( int j = r.lt.col; j <= r.rb.col; j++ )
            vetoSet.insert( Cell( i, j ) );
}
void Navy::AllocShip( int indShip, int nDeck, string name ) {
    int i, j;
    Cell lt, rb;
    // Генерация случайно размещенной начальной клетки корабля учетом
    // недопустимости "пересечения" нового корабля с множеством клеток vetoSet
    while( 1 ) {
        lt.row = rand( ) % ( N + 1 - nDeck );
        lt.col = rb.col = rand( ) % N;
        rb.row = lt.row + nDeck - 1;
        if ( !Rect( lt, rb ).Intersect( vetoSet ) ) break;
    }
    // Сохраняем данные о новом корабле
    ship[indShip] = Ship( nDeck, name, Rect( lt, rb ) );
    // Заносим новый корабль на игровое поле (символ DECK).
    // Добавляем соответствующие элементы в словарь ассоциаций
    for ( i = lt.row; i <= rb.row; i++ )
        for ( j = lt.col; j <= rb.col; j++ ) {
            ownField[i][j] = DECK;
            shipMap[Cell( i, j )] = indShip;
        }
    // Добавляем в множество vetoSet клетки нового корабля
    // вместе с пограничными клетками
    AddToVetoSet( Shell( Rect( lt, rb ) ) );
}
void Navy::Show() const {
    char rowName[10] = { 'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J' };
    string colName( "1 2 3 4 5 6 7 8 9 10" );
    int i, j;
    cout << "-----\n";
    cout << gap( 3 ) << "Мой флот" << gap( 18 ) << "Флот неприятеля" << endl;
    cout << gap( 3 ) << colName << gap( 6 ) << colName << endl;
    for ( i = 0; i < N; i++ ) {
        string line = gap( 1 ) + rowName[i]; // Own
        for ( j = 0; j < N; j++ )
            line += gap( 1 ) + ( char )ownField[i][j];
        line += gap( 5 ) + rowName[i]; // Enemy
        for ( j = 0; j < N; j++ )
            line += gap( 1 ) + ( char )enemyField[i][j];
        cout << line << endl;
    }
    cout << endl << "=====\n";
    cout << step << ". " << "Мой выстрел: ";
    step++;
}
int Navy::GetInt() {
    int value;

```

```

while ( true ) {
cin >> value;
    if ( '\n' == cin.peek() ) { cin.get(); break; }
    else {
        cout << "Повторите ввод колонки (ожидается целое число):" << endl;
        cin.clear();
        while ( cin.get() != '\n' ) {};
    }
}
return value;
}

void UserNavy::Allocation() { // -----Класс UserNavy
    srand( ( unsigned )time( NULL ) );
    AllocShip(0, 4, "Авианосец 'Варяг'"); AllocShip(1, 3, "Линкор 'Муромец'");
    AllocShip(2, 3, "Линкор 'Никитич'"); AllocShip(3, 2, "Крейсер 'Чудный'");
    AllocShip(4, 2, "Крейсер 'Добрый'"); AllocShip(5, 2, "Крейсер 'Справедливый'");
    AllocShip(6, 1, "Миноносец 'Храбрый'"); AllocShip(7, 1, "Миноносец 'Ушлый'");
    AllocShip(8, 1, "Миноносец 'Проворный'"); AllocShip(9,1, "Миноносец 'Смелый'");
    vetoSet.clear();
}

void UserNavy::FillDeadZone( Rect r, Field& field ) {
    int i, j;
    Rect sh = Shell( r );
    for ( i = sh.lt.row, j = sh.lt.col; j <= sh.rb.col; j++ )
        if ( sh.lt.row < r.lt.row ) field[i][j] = ' ';
    for ( i = sh.rb.row, j = sh.lt.col; j <= sh.rb.col; j++ )
        if ( sh.rb.row > r.rb.row ) field[i][j] = ' ';
    for ( j = sh.lt.col, i = sh.lt.row; i <= sh.rb.row; i++ )
        if ( sh.lt.col < r.lt.col ) field[i][j] = ' ';
    for ( j = sh.rb.col, i = sh.lt.row; i <= sh.rb.row; i++ )
        if ( sh.rb.col > r.rb.col ) field[i][j] = ' ';
}

void UserNavy::FireOff() {
    string capital_letter = "ABCDEFGH IJ";
    string small_letter = "abcdefghij";
    unsigned char rowName; // обозначение ряда (A, B, ... , J)
    int colName;           // обозначение колонки (1, 2, ..., 10)
    int row;               // индекс ряда (0, 1, ... , 9)
    int col;               // индекс колонки (0, 1, ... , 9)
    bool success = false;
    while ( !success ) {
        cin >> rowName;
        row = capital_letter.find( rowName );
        if ( -1 == row ) row = small_letter.find( rowName );
        if ( -1 == row ) { cout << "Ошибка. Повторите ввод.\n"; continue; }
        colName = GetInt();
        col = colName - 1;
        if ( ( col < 0 ) || ( col > 9 ) ) {
            cout << "Ошибка. Повторите ввод.\n"; continue;
        }
        success = true;
    }
    u_fire = Cell( row, col );
}

void UserNavy::ResultAnalys() {
    // r_state - сообщение робота о результате выстрела
    // пользователя по клетке u_fire
    switch( r_state ) {
    case Miss:
        enemyField[u_fire.row][u_fire.col] = MISS; break;
    case Damage:
        enemyField[u_fire.row][u_fire.col] = DAMAGE;
        crushSet.insert( u_fire ); break;
    case Kill:
        enemyField[u_fire.row][u_fire.col] = DAMAGE;
        crushSet.insert( u_fire );
        Rect kill;

```

```

        kill.lt = *crushSet.begin();
        kill.rb = *( --crushSet.end() );
        FillDeadZone( kill, enemyField ); // Заполняем "обрамление" пробелами
        crushSet.clear();
    }
}

void UserNavy::GetFire() {
    // выстрел робота - по клетке r_fire
    string capital_letter = "ABCDEFGH IJ";
    char rowName = capital_letter[r_fire.row];
    int colName = r_fire.col + 1;
    cout << "\nВыстрел неприятеля: " << rowName << colName << endl;
    if ( DECK == ownField[r_fire.row][r_fire.col] ) {
        cout << "*** Есть попадание! ***";
        ownField[r_fire.row][r_fire.col] = DAMAGE;
        u_state = Damage;
        int ind = shipMap[r_fire]; // индекс корабля, занимающего клетку r_fire
        ship[ind].nLiveDeck--;
        if ( !ship[ind].nLiveDeck ) {
            u_state = Kill;
            cout << gap( 6 ) << "О ужас! Погиб " << ship[ind].name << " !!!";
            nLiveShip--;
            Rect kill = ship[ind].place;
            FillDeadZone( kill, ownField );
        }
    }
    else {
        u_state = Miss;
        cout << "*** Мимо! ***";
        ownField[r_fire.row][r_fire.col] = MISS;
    }
    cout << endl;
}

RobotNavy::RobotNavy() { // ----- Класс RobotNavy
    Allocation();
    isCrushContinue = false;
    upEmpty = false;
}

void RobotNavy::Allocation() {
    AllocShip(0, 4, "Авианосец 'Алькайда'"); AllocShip( 1, 3, "Линкор 'БенЛаден'");
    AllocShip(2, 3, "Линкор 'Хусейн'");      AllocShip(3, 2, "Крейсер 'Подлый'");
    AllocShip(4, 2, "Крейсер 'Коварный'");   AllocShip(5, 2, "Крейсер 'Злой'");
    AllocShip(6, 1, "Миноносец 'Гадкий'");   AllocShip(7, 1, "Миноносец 'Мерзкий'");
    AllocShip(8, 1, "Миноносец 'Пакостный'"); AllocShip(9, 1, "Миноносец 'Душный'");
    vetoSet.clear();
}

void RobotNavy::FireOff() {
    Cell c, cUp;
    if ( !isCrushContinue ) {
        while( 1 ) { // случайный выбор координат выстрела
            c.row = rand() % N;
            c.col = rand() % N;
            if ( !c.InSet( vetoSet ) ) break;
        }
    }
    else {
        c = cUp = r_fire; // "пляшем" от предыдущего попадания
        cUp.row--;
        if ( ( !upEmpty ) && c.row && ( !cUp.InSet( vetoSet ) ) ) c.row--;
        else { c = *( --crushSet.end() ); c.row++; }
    }
    r_fire = c;
    vetoSet.insert( r_fire );
}

void RobotNavy::ResultAnalys() {
    // u_state - сообщение пользователя о результате
    // выстрела робота по клетке r_fire
    switch( u_state ) {
    case Miss:

```

```

        if ( isCrushContinue ) upEmpty = true; break;
case Damage:
    isCrushContinue = true;
    crushSet.insert( r_fire );
                                break;
case Kill:
    isCrushContinue = false;
    upEmpty = false;
    crushSet.insert( r_fire );
    Rect kill;
    kill.lt = *crushSet.begin();
    kill.rb = *( --crushSet.end() );
    AddToVetoSet( Shell( kill ) );
    crushSet.clear();
}
}
void RobotNavy::GetFire() {
    // выстрел пользователя - по клетке u_fire
    if ( DECK == ownField[u_fire.row][u_fire.col] ) {
        cout << "**** Есть попадание! ****";
        r_state = Damage;
        int ind = shipMap[u_fire];    // индекс корабля, занимающего клетку u_fire
        ship[ind].nLiveDeck--;
        if ( !ship[ind].nLiveDeck ) {
            r_state = Kill;
            cout << gap( 6 ) << "Уничтожен " << ship[ind].name << " !!!";
            nLiveShip--; }
        }
    else { r_state = Miss; cout << "**** Мимо! ****"; }
    cout << endl;
}
//////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////// Main.cpp
#include <iostream>
#include "Navy.cpp"
#include "Ship.cpp"
using namespace std;
int main() {
    UserNavy userNavy;
    RobotNavy robotNavy;
    userNavy.Show();
    while ( userNavy.IsLive() && robotNavy.IsLive() ) {
        if ( Space::u_state != Miss ) {    // -- Выстрел пользователя
            cout << "пропускается...: <Enter>" << endl;
            cin.get(); }
        else {
            userNavy.FireOff();
            robotNavy.GetFire();
            userNavy.ResultAnalys();
            if ( !robotNavy.IsLive() ) { userNavy.Show(); break; }
        }
        if ( Space::r_state != Miss )    // --- Выстрел робота
            cout << "\nВыстрел неприятеля: пропускается..." << endl;
        else {
            robotNavy.FireOff();
            userNavy.GetFire();
            robotNavy.ResultAnalys();
        }
        userNavy.Show();
    }
    if ( userNavy.IsLive() ) cout << "\n:-))) Ура! Победа!!! :-)))" << endl;
    else { cout << "\n:-((( Увы. Непрятель оказался сильнее." << endl;
        cout << ":-((( Но ничего, в следующий раз мы ему покажем!!!" << endl;
    }
    cin.get();
}

```

Обратите внимание на использование объектов контейнерных классов: `cset` типа `set<Cell>` в классе `Rect`; `vetoSet` и `crushSet` типа `set<Cell>` в классе `Navy`; `shipMap` типа `map<Cell, int>` в классе `Navy`.

В качестве упражнения рекомендуем вам доработать эту программу, сняв ограничение «только вертикальное расположение кораблей» (решение об ориентации размещаемого корабля принимать случайным образом).

Итоги

1. Стандартная библиотека шаблонов содержит общецелевые классы и функции, которые реализуют широко используемые алгоритмы и структуры данных.
2. STL построена на основе шаблонных классов, поэтому входящие в нее алгоритмы и структуры могут настраиваться на различные типы данных.
3. Использование STL позволяет значительно повысить надежность программ, их переносимость и универсальность, а также уменьшить сроки их разработки.
4. Везде, где это возможно, используйте классы и алгоритмы STL!

Задания

Вариант 1

Написать программу для моделирования T-образного сортировочного узла на железной дороге с использованием контейнерного класса `stack` из STL.

Программа должна разделять на два направления состав, состоящий из вагонов двух типов (на каждое направление формируется состав из вагонов одного типа). Предусмотреть возможность ввода исходных данных с клавиатуры и из файла.

Вариант 2

Написать программу, отыскивающую проход по лабиринту, с использованием контейнерного класса `stack` из STL.

Лабиринт представляется в виде матрицы, состоящей из квадратов. Каждый квадрат либо открыт, либо закрыт. Вход в закрытый квадрат запрещен. Если квадрат открыт, то вход в него возможен со стороны, но не с угла. Программа находит проход через лабиринт, двигаясь от заданного входа. После отыскания прохода программа выводит найденный путь в виде координат квадратов.

Вариант 3

Написать программу, моделирующую управление каталогом в файловой системе. Для каждого файла в каталоге содержатся следующие сведения: имя файла, дата создания, количество обращений к файлу. Программа должна обеспечивать:

- начальное формирование каталога файлов;
- вывод каталога файлов;
- удаление файлов, дата создания которых раньше заданной;
- выборку файла с наибольшим количеством обращений.

Выбор моделируемой функции должен осуществляться с помощью меню. Для представления каталога использовать контейнерный класс `list` из STL.

Вариант 4

Написать программу моделирования работы автобусного парка. Сведения о каждом автобусе содержат: номер автобуса, фамилию и инициалы водителя, номер маршрута. Следует обеспечить выбор с помощью меню и выполнение следующих функций:

- начальное формирование данных о всех автобусах в парке в виде списка (ввод с клавиатуры или из файла);
- имитация выезда автобуса из парка: вводится номер автобуса; программа удаляет данные об этом автобусе из списка автобусов, находящихся в парке, и записывает эти данные в список автобусов, находящихся на маршруте;
- имитация въезда автобуса в парк: вводится номер автобуса; программа удаляет данные об этом автобусе из списка автобусов, находящихся на маршруте, и записывает эти данные в список автобусов, находящихся в парке;

- вывод сведений об автобусах, находящихся в парке, и об автобусах, находящихся на маршруте.
- Для представления необходимых списков использовать контейнерный класс `list`.

Вариант 5

Написать программу учета заявок на авиабилеты. Каждая заявка содержит: пункт назначения, номер рейса, фамилию и инициалы пассажира, желаемую дату вылета. Следует обеспечить выбор с помощью меню и выполнение следующих функций:

- добавление заявок в список;
- удаление заявок;
- вывод заявок по заданному номеру рейса и дате вылета;
- вывод всех заявок.

Для хранения данных использовать контейнерный класс `list`.

Вариант 6

Написать программу учета книг в библиотеке. Сведения о книгах содержат: фамилию и инициалы автора, название, год издания, количество экземпляров данной книги в библиотеке. Следует обеспечить выбор с помощью меню и выполнение следующих функций:

- добавление данных о книгах, вновь поступающих в библиотеку;
- удаление данных о списываемых книгах;
- выдача сведений о всех книгах, упорядоченных по авторам;
- выдача сведений о всех книгах, упорядоченных по годам издания.

Хранение данных организовать с применением контейнерного класса `multimap`, в качестве ключа использовать «фамилию и инициалы автора».

Вариант 7

Написать программу «Моя записная книжка». Предусмотреть возможность работы с произвольным числом записей, поиска записи по какому-либо признаку (например, по фамилии, дате рождения или номеру телефона), добавления и удаления записей, сортировки по разным полям.

Хранение данных организовать с помощью класса `map` или `multimap`.

Вариант 8

Написать программу учета заявок на обмен квартир и поиска вариантов обмена. Каждая заявка содержит сведения о двух квартирах: требуемой (искомой) и имеющейся. Сведения о каждой квартире содержат: количество комнат, площадь, этаж, район. Следует обеспечить выбор с помощью меню и выполнение следующих функций:

- ввод заявки на обмен;
- поиск в картотеке подходящего варианта: при совпадении требований и предложений по количеству комнат и этажности и различии по показателю «площадь» в пределах 10% выводится соответствующая карточка и удаляется из списка, в противном случае поступившая заявка включается в картотеку;
- вывод всей картотеки.

Для хранения данных картотеки использовать контейнерный класс `list`.

Вариант 9

Написать программу «Автоматизированная информационная система на железнодорожном вокзале». Информационная система содержит сведения об отправлении поездов дальнего следования. Для каждого поезда указывается: номер поезда, станция назначения, время отправления. Следует обеспечить выбор с помощью меню и выполнение следующих функций:

- первоначальный ввод данных в систему (с клавиатуры или из файла);
- вывод сведений по всем поездам;
- вывод сведений по поезду с запрошенным номером;
- вывод сведений по поездам, следующим до запрошенной станции назначения.

Хранение данных организовать с применением контейнерного класса `vector`.

Вариант 10

Написать программу «Англо-русский и русско-английский словарь». «База данных» словаря должна содержать синонимичные варианты перевода слов. Следует обеспечить выбор с помощью меню и выполнение следующих функций:

- загрузка «базы данных» словаря (из файла);
- выбор режима работы: англо-русский или русско-английский;
- вывод вариантов перевода заданного английского слова;
- вывод вариантов перевода заданного русского слова.

Базу данных словаря реализовать в виде двух контейнеров типа `map`.

Вариант 11

Написать программу, реализующую игру «Крестики-нолики» между двумя игроками: пользователь и компьютер (робот). В программе использовать контейнеры STL.

Вариант 12

Написать программу, решающую игру-головоломку «Игра в 15». Начальное размещение номеров — случайное. Предусмотреть два режима демонстрации решения: непрерывный (с некоторой задержкой визуализации) и пошаговый (по нажатию любой клавиши). В программе использовать контейнерные классы STL.

Вариант 13

Составить программу формирования списка кандидатов, участвующих в выборах губернатора. Каждая заявка от кандидата содержит: фамилию и инициалы, дату рождения, место рождения, индекс популярности. Следует обеспечить выбор с помощью меню и выполнение следующих функций:

- добавление заявки в список кандидатов. Для ввода индекса популярности (значение указано в скобках) предусмотреть выбор с помощью подменю одного из следующих вариантов: поддержан президентом (70);
- поддержан оппозиционной партией (15);
- оппозиционный кандидат, который снимет свою кандидатуру в пользу кандидата № 1 (10);
- прочие (5).
- удаление заявки по заявлению кандидата;
- формирование и вывод списка для голосования.

Хранение данных организовать с применением контейнерного класса `priority_queue` из STL. Для надлежащего функционирования очереди с приоритетами побеспокоиться о надлежащем определении операции `<` (меньше) в классе, описывающем заявку кандидата. Формирование и вывод списка для голосования реализовать посредством выборки заявок из очереди.

Вариант 14

Составить программу моделирования работы автобусного парка. Сведения о каждом автобусе содержат: номер автобуса, фамилию и инициалы водителя, номер маршрута. Следует обеспечить выбор с помощью меню и выполнение функций:

- начальное формирование данных о всех автобусах в парке в виде списка (ввод с клавиатуры или из файла);
- имитация выезда автобуса из парка: вводится номер автобуса; программа удаляет данные об этом автобусе из списка автобусов, находящихся в парке, и записывает эти данные в список автобусов, находящихся на маршруте;
- имитация въезда автобуса в парк: вводится номер автобуса; программа удаляет данные об этом автобусе из списка автобусов, находящихся на маршруте, и записывает эти данные в список автобусов, находящихся в парке;
- вывод сведений об автобусах, находящихся в парке, и об автобусах, находящихся на маршруте, упорядоченных по номерам автобусов;

- вывод сведений об автобусах, находящихся в парке, и об автобусах, находящихся на маршруте, упорядоченных по номерам маршрутов.

Хранение всех необходимых списков организовать с применением контейнерного класса `map`, в качестве ключа использовать «номер автобуса».

Вариант 15

Составить программу учета заявок на авиабилеты. Каждая заявка содержит: пункт назначения, номер рейса, фамилию и инициалы пассажира, желаемую дату вылета. Следует обеспечить выбор с помощью меню и выполнение следующих функций:

- добавление заявок в список;
- удаление заявок;
- вывод заявок по заданному номеру рейса и дате вылета;
- вывод всех заявок, упорядоченных по пунктам назначения;
- вывод всех заявок, упорядоченных по датам вылета.

Хранение данных организовать с применением контейнерного класса `multimap`, в качестве ключа использовать «пункт назначения».

Вариант 16

Написать программу учета книг в библиотеке. Сведения о книгах содержат: фамилию и инициалы автора, название, год издания, количество экземпляров данной книги в библиотеке. Следует обеспечить выбор с помощью меню и выполнение функций:

- добавление данных о книгах, вновь поступающих в библиотеку;
- удаление данных о списываемых книгах;
- выдача сведений о всех книгах, упорядоченных по авторам;
- выдача сведений о всех книгах, упорядоченных по годам издания.

Хранение данных организовать с применением контейнерного класса `vector`.

Вариант 17

Написать программу «Моя записная книжка». Предусмотреть возможность работы с произвольным числом записей, поиска записи по какому-либо признаку (например, по фамилии, дате рождения или номеру телефона), добавления и удаления записей, сортировки по разным полям.

Хранение данных организовать с применением контейнерного класса `list`.

Вариант 18

Написать программу учета заявок на обмен квартир и поиска вариантов обмена. Каждая заявка содержит фамилию и инициалы заявителя, а также сведения о двух квартирах: требуемой (искомой) и имеющейся. Сведения о каждой квартире содержат: количество комнат, площадь, этаж, район. Следует обеспечить выбор с помощью меню и выполнение следующих функций:

- ввод заявки на обмен;
- поиск в картотеке подходящего варианта: при совпадении требований и предложений по количеству комнат и этажности и при различии по показателю «площадь» в пределах 10% выводится соответствующая карточка и удаляется из списка, в противном случае поступившая заявка включается в картотеку;
- вывод всей картотеки.

Хранение данных организовать с применением контейнерного класса `set`.

Вариант 19

Написать программу «Автоматизированная информационная система на железнодорожном вокзале». Информационная система содержит сведения об отправлении поездов дальнего следования. Для каждого поезда указывается: номер поезда, станция назначения, время отправления. Следует обеспечить выбор с помощью меню и выполнение следующих функций:

- первоначальный ввод данных в информационную систему (с клавиатуры или из файла);
- вывод сведений по всем поездам;

- вывод сведений по поезду с запрошенным номером;
- вывод сведений по поездам, следующим до запрошенной станции назначения. Хранение данных организовать с применением контейнерного класса `set`.

Вариант 20

Написать программу «Англо-русский и русско-английский словарь». «База данных» словаря должна содержать синонимичные варианты перевода слов. Следует обеспечить выбор с помощью меню и выполнение следующих функций:

- загрузка «базы данных» словаря (из файла);
- выбор режима работы: англо-русский или русско-английский;
- вывод вариантов перевода заданного английского слова;
- вывод вариантов перевода заданного русского слова.

Базу данных словаря реализовать в виде двух контейнеров типа `set`.