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

МЕТОДИЧЕСКИЕ УКАЗАНИЯ

по выполнению лабораторных работ
по дисциплине «Объектно ориентированное программирование»
для студентов направлений 09.03.03 «Прикладная информатика».

Ставрополь
2023

ВВЕДЕНИЕ

Целью дисциплины является обучение студентов основам объектно-ориентированного проектирования и программирования в современных средах разработки ПО.

Задачами дисциплины являются:

- Основой задачей изучения курса является получение знаний и практических навыков в области проектирования и разработки объектно-ориентированных программ. В результате изучения курса студент должен иметь представление о предпосылках возникновения ООП и его месте в эволюции парадигм программирования, знать принципы объектно-ориентрованного проектирования и программирования, а также уметь разрабатывать объектно-ориентированные программы на языках

Лабораторная работа №1.
Классы.

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

Формируемые компетенции: ПК-7, ПК-8

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

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

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 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 сигнатурами[2]? 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

[2] *Сигнатура*, *прототип*, *заголовок* функции — синонимы.

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) внутрь класса. Поэтому в данной задаче более удачной является сигнатура метода void SetBi rthYear(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.h" 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 i = 0;
    while (fin.getline( buf, l_buf )) {
        if (i >= maxn_record) { cout << "Слишком длинный файл"; return 1; }
        man[i].SetName( buf );
         man[i].SetBirthYear( buf );
         man[i].SetPay( buf );
        i++;
    }
    int n_record = i, n_man = 0;     float mean_pay = 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 вовсе 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 Coo, а y — объект типа Coo. Тогда следующие операторы вызовут конструктор копирования класса Coo:

```
Coo 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 встроенного типа разницы в эффективности нет. Когда мы будем применять контейнерные классы стандартной библиотеки, параметр в заголовке цикла часто будет представлять объектитератор, 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. Вовторых, 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>
  #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.h"
    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 – опред-ь. отнош-ие 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:
  1. определения восьми точек p1, …, p8 сделаны произвольно, но так, чтобы из них можно было составить треугольники;
  1. определения четырех треугольников сделаны тоже произвольно, впоследствии на них будут демонстрироваться основные методы класса;
  1. 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 1[3]. 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.

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.

[3] После ввода числовой информации всегда подразумевается нажатие Enter.

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_2 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 ==========
1 - вывести все объекты 3 - найти максимальный
2 – 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: . . .
File: dbgdel.cpp
Line: 47

Чтобы понять, что происходит, давайте посмотрим на нашу отладочную печать. Перед тем как умереть, программа успела вывести на экран следующее:

5 Конец работы.
Destructor for: triaD
Destructor for: triaC
Destructor for: НННННННННЭээээЭЭЭЭЭЭЭЭЭЭQ

Пришло время для аналитической работы наших серых клеточек. Вспомните, как выглядела отладочная печать при завершении нормально работающей первой версии программы? Деструкторы вызывались в порядке, обратном вызову конструкторов. Значит, какая-то беда случилась после вызова деструктора для объекта triaB! Почему? Всмотримся в предыдущий вывод нашей программы. После того как функция FindMax выполнила основную работу и вывела результат на экран:

Максимальный треугольник: Треугольник 2

программа пригласила пользователя нажать клавишу Enter. Это приглашение выводится функцией ExitBack. А вот после нажатия клавиши Enter на экране появился текст:

Destructor for: triaB

после которого опять было выведено главное меню. Значит, деструктор для объекта triaB был вызван в момент возврата из функции FindMax. Но почему? Ведь объект triaB создается в основной функции main, а значит, там же должен и уничтожаться! Что-то нехорошее происходит, однако, в теле функции FindMax. Хотя внешне вроде все прилично... Стоп! Ведь внутри функции объявлен объект triaMax, и мы даже видели работу его конструктора:

Constructor_2 for: triaMax (Треугольник 5)

А где же вызов деструктора, который по идее должен произойти в момент возврата из функции FindMax? Кажется, мы нашли причину ненормативного поведения нашей программы. Объект triaMax после своего объявления неоднократно модифицируется с помощью операции присваивания. А теперь давайте вспомним, что, если мы не перегрузили операцию присваивания для некоторого класса, компилятор сделает это за нас, но в ней будут поэлементно копироваться все поля объекта. При наличии же полей типа указателей возможны неприятные проблемы.

В данном случае в поля objID и name объекта triaMax были скопированы значения одноименных полей объекта triaB. В момент выхода из функции FindMax деструктор объекта освободил память, на которую указывали эти поля. А при выходе из основной функции main деструктор объекта triaB попытался еще раз освободить эту же память. Результат вы уже видели…

Займемся починкой нашей программы. Нужно добавить в класс Triangle перегрузку операции присваивания, а заодно и конструктор копирования. Внесите следующие изменения в тексты модулей проекта:

4. **Triangle.h.** Добавьте объявления:

```
Triangle( const Triangle& );            // Конструктор копирования
Triangle& operator =( const Triangle& );  // Операция присваивания
```

5. **Triangle.cpp.** Добавьте реализацию:

```
// Конструктор копирования
Triangle::Triangle(const Triangle& tria): v1(tria.v1), v2(tria.v2), v3(tria.v3) {
   cout << "Copy constructor for: " << tria.objID << endl;    // Отладочный вывод
   objID = new char[strlen(tria.objID) + strlen("(копия)") + 1];
   strcpy(objID, tria.objID);
   strcat(objID, "(копия)");
   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;
}
// Присвоить значение объекта tria
Triangle& Triangle::operator =(const Triangle& tria) {
   // отладочный вывод:
```

```
    cout << "Assign operator: " << objID << " = " << tria.objID << endl;
    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 ==================
  1     - вывести все объекты   3 - найти максимальный
  2     – 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 AYB, 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;  // Площать 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 tria1, Triangle tria2) {
   Point v1 = tria1.Get_v1();
   Point v2 = tria1.Get_v2();
   Point v3 = tria1.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

При выполнении задания необходимо построить UML-диаграмма классов приложения

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; ‰ 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 (неравенство);
- 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); ‰ 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 выравнен 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:

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 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 встроенных 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. Язык программирования C++. Специальное издание / Б. Страуструп. – Издательство: Бином, 2008. – 1104 с.
2. Прата, С. Язык программирования C++. Лекции и упражнения / С. Прата. – М.: Издательство: ДиаСофтЮП, 2005. – 1104 с.
3. Архангельский, А.Я. Программирование в C++ Builder / А.Я. Архангельский. – М.: Бином, 2008.
4. Павловская, Т.А. C/ C++. Программирование на языке высокого уровня / Т.А. Павловская. – СПб.: Питер, 2011. – 461 с.
5. 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.fincher.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.