МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ УКРАИНЫ
Севастопольский национальный технический университет

Методические указания к лабораторным работам по дисциплине “Объектно-ориентированное программирование”

для студентов дневной и заочной формы обучения направления 050101 — “Компьютерные науки”

Часть 2

Севастополь
2012

УДК 004.42 (075.8)

Методические указания к лабораторным работам по дисциплине “Объектно-ориентированное программирование” для студентов дневной и заочной формы обучения направления 050101 — “Компьютерные науки”: в 3 ч. / СевНТУ ; сост. **Ю.В. Доронина**, **Т. И. Сметанина,** А**. К. Забаштанский**. — Севастополь: Изд-во СевНТУ, 2012. ч. 2. — 32 с.

Цель указаний: оказать помощь студентам направления “Компьютерные науки” в выполнении лабораторных работ по дисциплине “Объектно-ориентированное программирование”.

Методические указания составлены в соответствии с требованиями программы дисциплины “Объектно-ориентированное программирование” для студентов дневной и заочной формы обучения направления 050101 — “Компьютерные науки” и утверждены на заседании кафедры Информационных систем протоколом № 10 от 20 апреля 2011 г.

Допущено учебно-методическим центром СевНТУ в качестве методических указаний.

Рецензент: Кожаев Е.А., канд. техн. наук, доцент кафедры кибернетики и вычислительной техники.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Цели и задачи лабораторных работ

Основная цель выполнения лабораторных работ — приобретение практических навыков написания программ на объектно-ориентированном языке программирования — Си++. В результате выполнения лабораторных работ студенты углубляют знания основных теоретических положений дисциплины “Объектно-ориентированное программирование”, решая практические задачи на ЭВМ.

По окончанию курса студенты должны овладеть объектно-ориентированным подходом решения практических задач, овладеть инструментами, сопровождающими разработку программных систем с использованием этого подхода.

Выбор вариантов и график выполнения лабораторных работ

При изучении курса «Объектно-ориентированное программирование» выполняется 8 лабораторных работ. Каждая работа выполняется в течение 2 академических часов.

В лабораторных работах студент решает заданную индивидуальным вариантом задачу. Варианты заданий приведены в каждой лабораторной работе и уточняются с преподавателем. Номер варианта выбирается в соответствии с номером зачетной книжки студента. Вариант вычисляется как остаток от деления числа, соответствующего двум последним цифрам в номере зачетной книжки, на 15. Например, две последние цифры зачетки 42. Тогда остаток деления 42 на 15 будет равен 12. Следовательно, студент выбирает вариант 12.

Лабораторная работа выполняется в два этапа. На первом этапе, выполняемом самостоятельно дома, студенту нужно выполнить следующее:

- проработать по конспекту и рекомендованной литературе, приведенной в конце настоящих методических указаний, основные теоретические положения лабораторной работы и ответить на контрольные вопросы;
- разработать алгоритм решения задачи;
- написать программу на языке Си ++, реализующий разработанный алгоритм;
- оформить результаты первого этапа в виде заготовки отчета по выполнению лабораторной работы (студенты-заочники первый этап выполнения лабораторных работ оформляют в виде контрольной работы, которую сдают на кафедру до начала зачетно-экзаменационной сессии).

На втором этапе, выполняемом в лабораториях кафедры, студенту следует:

- отладить программу;
- разработать и выполнить тестовый пример;
- подготовить и защитить отчёт по лабораторной работе.

Студенты должны выполнять и защищать работы строго по графику.

График выполнения лабораторных работ:
- лабораторная работа №1 — 2-ая неделя весеннего семестра;
- лабораторная работа №2 — 4-ая неделя весеннего семестра;
- лабораторная работа №3 — 6-ая неделя весеннего семестра;
- лабораторная работа №4 — 8-ая неделя весеннего семестра;
- лабораторная работа №5 — 10-ая неделя весеннего семестра;
- лабораторная работа №6 — 11-ая неделя весеннего семестра;
- лабораторная работа №7 — 12-ая неделя весеннего семестра;
- лабораторная работа №8 — 14-ая неделя весеннего семестра.

Требования к оформлению отчета

Отчёты по лабораторной работе оформляются каждым студентом индивидуально. В общем случае отчёт должен включать: название и номер лабораторной работы; цель работы; вариант задания и постановку задачи; описание алгоритма решения задачи; описание программы; выводы по работе; приложения. Содержание отчета указано в методических указаниях к каждой лабораторной работе.

Постановка задачи представляет изложение содержания задачи, цели её решения. На этом этапе должны быть полностью определены исходные данные, перечислены задачи по преобразованию и обработке входных данных, описаны выходные данные, дана характеристика результатов.

Описание программы включает описание вычислительного процесса на языке Си++. Кроме текста программы, в отчёте представляется её пооператорное описание, даётся характеристика конструкций языка, обеспечивающих решение задачи конкретной лабораторной работы.

В приложении приводится текст программы, результат выполнения тестового примера.

ЛАБОРАТОРНАЯ РАБОТА №5
ПРОСТОЕ НАСЛЕДОВАНИЕ. ВИРТУАЛЬНЫЕ ФУНКЦИИ.

1. ЦЕЛЬ РАБОТЫ

Приобретение практических навыков при написании объектно-ориентированных программ с использованием механизма наследования и механизма виртуальных функций. Освоение особенностей отладки объектно-ориентированных программ.

2. ТЕОРЕТИЧЕСКИЙ РАЗДЕЛ

2.1. Простое наследование

Язык C++ позволяет классу наследовать данные-члены и функции-члены одного или нескольких других классов. При этом новый класс называют производным классом (или классом-потомком). Класс, элементы которого наследуются, называется базовым классом (родительским классом или классом-предком) для своего производного класса. Наследование дает возможность некоторые общие черты поведения классов абстрагировать в одном базовом классе. Производные классы, наследуя это общее поведение, могут его несколько видоизменять, переопределяя некоторые функции-члены базового класса, или дополнять, вводя новые данные-члены и функции-члены. Таким образом, определение производного класса значительно сокращается, поскольку нужно определить только отличающие его от производных классов черты поведения.

Если у производного класса имеется всего один базовый класс, то говорят о простом (или одиночном) наследовании. Синтаксис объявления производного класса имеет следующий вид:

```
class <имя класса>:[<спецификатор доступа>] <имя базового класса>
{
    <тело класса>
};
```

Спецификатор доступа — это одно из ключевых слов *private*, *protected* или *public*. Он не является обязательным и по умолчанию принимает значение *private* для классов и *public* для структур. Рассмотрим пример:

```
class Base  // Базовый класс
{  public:
   int SetX(int _x=0) { return x=_x;}
   int GetX( ) {return x;}
protected:                    // Обозначает, что данное поле доступно
   int x;                     // данному классу и всем его потомкам
};
class Derived : public Base  // Класс-наследник
{      };
```

```
int main(void)
{       int x;
        Derived ob;         // Создаем объект класса-наследника
        x = ob.GetX( );     // и обращаемся в нем к полям и методам базового класса
        return 0;
}
```

Спецификатор доступа при наследовании определяет уровень доступа к элементам базового класса, который получают элементы производного класса. В приведенной ниже таблице описаны возможные варианты наследуемого доступа.

Таблица 3.1 — Распространение спецификаторов доступа в рамках иерархии наследования

Доступ в базовом классе	Спецификатор наследуемого доступа	Доступ в производном классе
Public *Protected* *Private*	*Public*	*Public* *Protected* Нет доступа
Public *Protected* *Private*	*Protected*	*Protected* *Protected* Нет доступа
Public *Protected* *Private*	*Private*	*Private* *Private* Нет доступа

Можно сделать вывод, что спецификатор наследуемого доступа устанавливает тот уровень доступа, до которого понижается уровень доступа к членам, установленный в базовом классе.

Следует отметить, что не все члены класса будут наследоваться. Не подлежат наследованию следующие элементы класса:

- конструкторы;
- конструкторы копирования;
- деструкторы;
- операторы присваивания, определенные программистом;
- друзья класса.

2.2. Переопределение функций

В производном классе обычно добавляются новые поля и методы, не существующие в базовом классе. Однако существует также возможность переопределения функций-членов базового класса. Чтобы переопределить функцию-член базового класса в производном классе, достаточно включить ее прототип в объявление этого класса и затем дать ее определение. Конечно, прототипы переопределяемой функции в базовом и производном классах должны совпадать. Рассмотрим пример:

```
 #include <iostream.h>
class MyInt
{ protected:
        int x;
 public:
      MyInt(int _x) {x = _x;}
      MyInt( ){x = 0;}
      int GetX( ){return x;}    // Определим метод вывода поля для базового класса
      void SetX(int _x) {x = _x;}
};
class IncMyInt: public MyInt
{ public:
      IncMyInt(int _x) : MyInt(_x) {}
      int GetX( ) {return ++x;} // Переопределим для наследника метод
                                // базового класса
      void ShowX( ) {cout << GetX()<<' ';}
};
main( )
{
      IncMyInt* ptr;
      ptr = new IncMyInt(10); // Создание объекта в куче
      ptr->ShowX( );
      delete ptr;             //Удаление объекта в куче
      return 0;
}
```

В этом случае функция-член *ShowX()* объекта *ptr* использует для получения значений данных-членов переопределенный вариант функции базового класса *GetX()*. В результате на экран программа выведет инкрементированное значение заданного числа.

Иногда возникает необходимость вызвать функцию-член базового класса, а не ее переопределенный вариант, это можно сделать с помощью операции ***расширения области видимости***, применяемой в форме:

< имя_класса> **::** <имя_функции>

Это дает возможность компилятору "видеть" за пределами текущей области видимости. Особенно часто это приходится делать при переопределении функций-членов. Переопределяемая функция-член может вызывать соответствующую функцию-член базового класса, а затем выполнять некоторую дополнительную работу (или наоборот). Например, в предыдущем примере можно было переопределить функцию *GetX()* следующим образом:

```
int IncMyInt:: GetX()
{
```

```
    int z ;
    z = MyInt::GetX{} ;  //Вызов переопределенной функции
    return ++z;
}
```

2.3. Конструкторы и деструкторы

Конструкторы не наследуются, поэтому производный класс либо должен объявить свой конструктор, либо предоставить возможность компилятору генерировать конструктор по умолчанию. Поскольку производный класс должен унаследовать все члены родительского, при построении объекта своего класса он должен обеспечить инициализацию унаследованных данных-членов, причем она должна быть выполнена до инициализации данных-членов производного класса, так как последние могут использовать значения первых. В связи с этим для построения конструктора производного класса применяется следующая конструкция:

<имя класса>([<список аргументов>]) **:** <имя класса предка>([<список аргументов>])

То есть используется список инициализации элементов, в котором указывается конструктор базового класса. Часть параметров, переданных конструктору производного класса, обычно используется в качестве аргументов конструктора базового класса. Затем в теле конструктора производного класса выполняется инициализация данных-членов, принадлежащих собственно этому классу.

В приведенном выше примере эта конструкция использована для создания конструктора класса *IncMyInt*. В данном случае конструктор имеет вид:

```
IncMyInt (int_x) : MyInt (_x) {}
```

Тело конструктора оставлено пустым, так как класс *IncMyInt* не имеет собственных данных-членов. Все параметры конструктора производного класса просто переданы конструктору базового класса для осуществления инициализации.

Если предоставляемый программистом конструктор не имеет параметров, то есть является конструктором по умолчанию, при создании экземпляра производного класса автоматически вызывается конструктор базового класса. После того как объект создан, конструктор базового класса становится недоступным.

В отношении деструкторов производных классов также действуют определенные правила. Деструктор производного класса должен выполняться раньше деструктора базового класса (иначе деструктор базового класса мог бы разрушить данные-члены, которые используются и в производном классе). Когда деструктор производного класса выполнит свою часть работы по уничтожению объекта, вызывается деструктор базового класса. Причем вся работа по организации соответствующего вызова возлагается на компилятор, программист не должен заботиться об этом.

2.4. Виртуальные методы

Работа с объектами чаще всего производится через указатели. Указателю на базовый класс можно присвоить значение адреса объекта любого производного класса, например:

*MyInt *p;* // Описывается указатель на базовый класс
p = new IncMyInt; // Указатель ссылается на объект производного класса

Вызов методов объекта происходит в соответствии с типом указателя, а не фактическим типом объекта, на который он ссылается, поэтому при выполнении оператора, например,

p->SetX(123);

будет вызван метод класса *MyInt*, а не класса *IncMyInt*, поскольку ссылки на методы разрешаются во время компоновки программы. Этот процесс называется ***ранним связыванием***. Чтобы вызвать метод класса daemon, можно использовать явное преобразование типа указателя:

*(IncMyInt * p)-> SetX (123);*

Это не всегда возможно, поскольку в разное время указатель может ссылаться на объекты разных классов иерархии, и во время компиляции программы конкретный класс может быть неизвестен. В качестве примера можно привести функцию, параметром которой является указатель на объект базового класса. На его место во время выполнения программы может быть передан указатель на любой производный класс. Другой пример — связный список указателей на различные объекты иерархии, с которым требуется работать единообразно.

В C++ реализован механизм ***позднего связывания***, когда разрешение ссылок на метод происходит на этапе выполнения программы в зависимости от конкретного типа объекта, вызвавшего метод. Этот механизм реализован с помощью виртуальных методов.

Функции, у которых известен интерфейс вызова (то есть прототип), но реализация не может быть задана в общем случае, а может быть определена только для конкретных случаев, называются виртуальными (термин, означающий, что функция может быть переопределена в производном классе).

Виртуальные функции — это функции, которые гарантируют, что будет вызнана правильная функция для объекта безотносительно к тому, какое выражение используется для осуществления вызова.

Для определения ***виртуального метода*** используется спецификатор *virtual*, например:

virtual void draw(int x, int y, int scale, int position);

Рассмотрим правила описания и использования виртуальных методов:

1. Если в базовом классе метод определен как виртуальный, то метод, определенный в производном классе *с тем же именем и набором параметров*, автоматически становится виртуальным, а *с отличающимся набором параметров* — обычным.

2. Виртуальные методы *наследуются*, то есть, переопределять их в производном классе требуется только при необходимости задать отличающиеся действия. Права доступа при переопределении изменить нельзя.

3. Если виртуальный метод переопределен в производном классе, объекты этого класса могут получить доступ к методу базового класса с помощью операции расширения области видимости.

4. Виртуальный метод не может объявляться с модификатором static, но может быть объявлен как дружественный.

5. Если в классе вводится описание виртуального метода, он должен быть определен хотя бы как чисто виртуальный.

Чисто виртуальный метод содержит признак “= 0” вместо тела, например:

```
virtual void f(int) = 0;
```

Чисто виртуальный метод должен переопределяться в производном классе (возможно, опять как чисто виртуальный).

Итак, ***виртуальным называется метод, ссылка на который разрешается на этапе выполнения программы*** (перевод красивого английского слова *virtual* — в данном значении всего-навсего “фактический”, то есть ссылка разрешается по факту вызова).

2.5. Механизм позднего связывания

Для каждого класса (не объекта!), содержащего хотя бы один виртуальный метод, компилятор создает ***таблицу виртуальных методов*** (*vtbl*), в которой для каждого виртуального метода записан его адрес в памяти. Адреса методов содержатся в таблице в порядке их описания в классах. Адрес любого виртуального метода имеет в *vtbl* одно и то же смещение для каждого класса в пределах иерархии.

Каждый объект содержит скрытое дополнительное ***поле ссылки*** на *vtbl*, называемое *vptr*. Оно заполняется конструктором при создании объекта (для этого компилятор добавляет в начало тела конструктора соответствующие инструкции).

На этапе компиляции ссылки на виртуальные методы заменяются на обращения к *vtbl* через *vptr* объекта, а на этапе выполнения в момент обращения к методу его адрес выбирается из таблицы. Таким образом, вызов виртуального метода, в отличие от обычных методов и функций, выполняется через дополнительный этап получения адреса метода из таблицы. Это несколько замедляет выполнение программы.

Поскольку объекты с виртуальными функциями должны поддерживать и таблицу виртуальных функций, то их использование всегда ведет к некоторому повышению затрат памяти и снижению быстродействия программы. Если вы работаете с небольшим классом, который не собираетесь делать базовым для других классов, то в этом случае нет никакого смысла в использовании виртуальных функций.

2.6. Виртуальный деструктор

Рекомендуется делать виртуальными деструкторы для того, чтобы гарантировать правильное освобождение памяти из-под динамического объекта, поскольку в этом случае в любой момент времени будет выбран деструктор, соответствующий фактическому типу объекта. Деструктор передает операции delete размер объекта, имеющий тип *size_t*. Если удаляемый объект является производным и в нем не определен виртуальный деструктор, передаваемый размер объекта может оказаться неправильным.

При удалении объекта производного класса, на который ссылается указатель базового класса, если деструктор объявлен как виртуальный, то будет вызван деструктор соответствующего производного класса. Затем деструктор производного класса вызовет деструктор базового класса и объект будет правильно удален (удален целиком). В противном случае будет вызван только деструктор базового класса и произойдет утечка памяти. Рассмотрим пример:

```
class Base
{       int x;
`public:
        Base(_x){x = _x;};
        ~Base( )  { /*Освобождение памяти */ }
};
class Derived: public Base
{ public:
       ~Derived{ )  { /*Освобождение памяти */ }
};
int main ( )
{       Base* pBase = new Derived;
        //...
        delete pBase;
        return 0;
}
```

Так как объявленный в классе *Derived* деструктор не является виртуальным, инструкция “*delete pBase*;” вызовет удаление только памяти, принадлежащей классу *Base*, поскольку она будет квалифицирована как вызов *Base::~Base()*. Это может привести к утечке памяти. Если же сделать деструктор базового класса виртуальным, то и деструктор производного класса будет виртуальным. Значит использование предыдущей инструкции будет квалифицировано как вызов ”*Derived :: ~ Derived();*” и удаление объекта произойдет правильно.

В связи с этим можно сформулировать следующие правила:

- используйте виртуальный деструктор, если в базовом классе имеются виртуальные функции;

- используйте виртуальные функции только в том случае, если программа содержит и базовый, и производный классы;

- не пытайтесь создать виртуальный конструктор.

Виртуальный механизм работает ***только при использовании указателей или ссылок*** на объекты. Объект, определенный через указатель или ссылку и содержащий виртуальные методы, называется ***полиморфным***. В данном случае полиморфизм состоит в том, что с помощью одного и того же обращения к методу выполняются различные действия в зависимости от типа, на который ссылается указатель в каждый момент времени.

2.7. Абстрактные классы

Класс, содержащий хотя бы один чисто виртуальный метод, называется ***абстрактный***. Абстрактные классы предназначены для представления общих понятий, которые предполагается конкретизировать в производных классах. Абстрактный класс может использоваться ***только в качестве базового*** для других классов — объекты абстрактного класса создавать нельзя, поскольку прямой или косвенный вызов чисто виртуального метода приводит к ошибке при выполнении.

При определении абстрактного класса необходимо иметь в виду следующее:

a) абстрактный класс не может использоваться в качестве типа аргумента, передаваемого функции;
b) абстрактный класс не может использоваться в качестве типа возвращаемого значения функции;
c) нельзя осуществлять явное преобразование типа объекта к типу абстрактного класса;
d) нельзя объявить представитель абстрактного класса;
e) можно объявить указатель или ссылку на абстрактный класс.

Таким образом, можно создать функцию, параметром которой является указатель на абстрактный класс. На место этого параметра при выполнении программы может передаваться указатель на объект любого производного класса. Это позволяет создавать ***полиморфные функции***, работающие с объектом любого типа в пределах одной иерархии.

3. ПОРЯДОК ВЫПОЛНЕНИЯ ЛАБОРАТОРНОЙ РАБОТЫ

1. В ходе самостоятельной подготовки изучить основы работы с потоковым вводом/выводом.

2. Разработать программу на языке C++.

3. Разработать тестовые примеры и выполнить отладку программы.

4. Получить результаты работы программы и исследовать её свойства для различных режимов работы, сформулировать выводы.

5. Оформить отчет по проделанной работе.

4. ВАРИАНТЫ ЗАДАНИЙ

Для заданной по варианту иерархии описать классы, конструкторы и деструктор, функции ввода и вывода информации на экран. Базовый класс определить как абстрактный, а заданную функцию — как чисто виртуальную в базовом классе и переопределить ее в остальных классах иерархии. Проиллюстрировать корректную работу виртуальных функций и механизма наследования.

Вариант 1

Создать абстрактный базовый класс Фигура с виртуальной функцией Площадь. Создать производные классы: прямоугольник, круг, прямоугольный треугольник со своими функциями площади.

Вариант 2

Создать абстрактный базовый класс с виртуальной функцией: норма. Создать производные классы: комплексные числа, вектор из 10 элементов, матрица (2x2). Определить функцию нормы: для комплексных чисел — модуль в квадрате, для вектора — корень квадратный из суммы элементов по модулю, для матрицы — максимальное значение по модулю.

Вариант 3

Создать абстрактный базовый класс Фигура и производные классы: круг, прямоугольник, четырехугольник. Определить виртуальную функцию вычисления периметра.

Вариант 4

Создать абстрактный базовый класс Вектор (одномерный массив) с виртуальной функцией Сумма, вычисляющей сумму элементов вектора. Создать производные классы: вектор целых чисел (*int**) и два вектора вещественных чисел (*float** и *double**).

Вариант 5

Создать абстрактный базовый класс Прогрессия с виртуальной функцией Сумма прогрессии. Создать производные классы: арифметическая прогрессия и геометрическая прогрессия. Каждый класс имеет два поля типа *double*. Первое — первый член прогрессии, второе — постоянная разность (для арифметической) и постоянное отношение (для геометрической). Определить функцию вычисления суммы, где параметром является количество элементов прогрессии.

Арифметическая прогрессия $a_j = a_0 + jd$, $j = 0, 1, 2, \ldots$

Сумма арифметической прогрессии: $s_n = (n+1)(a_0 + a_n) / 2$

Геометрическая прогрессия: $a_j = a_0 r^j$, $j = 0, 1, 2, \ldots$

Сумма геометрической прогрессии: $s_n = (a_0 - a_n r) / (1-r)$

Вариант 6

Создать абстрактный базовый класс Матрица с виртуальной функцией поиска максимального значения в массиве. Создать производные классы: матрица целых чисел (*int***), матрица символов (*char* **) и матрица вещественных чисел (*double***).

Вариант 7

Создать абстрактный базовый класс Уравнение с виртуальной функцией нахождения значения *y* по заданному значению *x*. Создать производные классы: линейное уравнение, квадратное уравнение и кубическая парабола.

Вариант 8

Создать абстрактный базовый класс Число с виртуальной функцией изменения знака числа. Создать производные классы: целое (*int*), вещественное (*double*) и комплексное (*float*, *float*).

Вариант 9

Создать абстрактный базовый класс Фигура с виртуальной функцией Объем. Создать производные классы: параллелепипед, тетраэдр, шар со своими функциями объема. Объем параллелепипеда вычисляется по формуле $V = xyz$ (x, y, z — стороны), тетраэдра: $V = \frac{\sqrt{2}}{12} a^3$, шара: $V = \frac{4}{3} \pi R^3$

Вариант 10

Создать абстрактный базовый класс Вектор (одномерный массив) с виртуальной функцией Сортировка, которая упорядочивает вектор по возрастанию значения элементов. Создать производные классы: вектор целых чисел (*int**) и два вектора вещественных чисел (*float** и *double**).

Вариант 11

Создать абстрактный базовый класс Поиск с виртуальной функцией поиска. Создать производные классы: символ (*char*), строка (*char **) и матрица (*char ***). Функция проверяет, существует ли введенная буква в строке/матрице/равна ли символу?

Вариант 12

Создать абстрактный базовый класс Студент с виртуальной функцией печати информации. Создать производные классы: студент-дипломник (тема диплома, специализация как бакалавра), студент-бакалавр (тема работы), студент младших курсов.

Вариант 13

Создать абстрактный базовый класс Матрица с виртуальной функцией поиска наиболее часто встречаемого элемента. Создать производные классы: матрица целых чисел (*int***), матрица символов (*char***) и матрица вещественных чисел (*double***).

Вариант 14

Создать абстрактный базовый класс Фигура, и производные классы: круг, прямоугольник, четырехугольник. Определить виртуальную функцию вычисления периметра.

Вариант 15

Создать абстрактный базовый класс Человек, у которого есть виртуальная функция печати личных данных. Создать производные классы: мужчина (поля: ФИО, воз-

раст, должность, статус, имя жены) и женщина (поля: ФИО, возраст, статус, имя мужа, дети, имена детей).

5. СОДЕРЖАНИЕ ОТЧЕТА О ВЫПОЛНЕНИИ ЛАБОРАТОРНОЙ РАБОТЫ

Титульный лист, цель работы, вариант задания, текст программы с комментариями, описание тестовых примеров и выводы по проделанной работе.

6. КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Что такое простое наследование, для чего оно применяется?
2. Что такое спецификатор доступа, для чего он нужен при наследовании?
3. Привести пример переопределения функции и объяснить суть механизма.
4. Перечислить особенности описания конструкторов и деструкторов при наследовании.
5. Что такое полиморфизм?
6. Что такое виртуальная функция? Когда она применяется? Привести пример.
7. Что такое чисто виртуальная функция? Абстрактный класс? Для чего они нужны?
8. Что такое механизм позднего связывания?

ЛАБОРАТОРНАЯ РАБОТА №6 МНОЖЕСТВЕННОЕ НАСЛЕДОВАНИЕ.

1. ЦЕЛЬ РАБОТЫ

Приобретение практических навыков при написании объектно-ориентированных программ. Освоение особенностей отладки объектно-ориентированных программ. Изучение механизма множественного наследования.

2. ТЕОРЕТИЧЕСКИЙ РАЗДЕЛ

2.1. Множественное наследование

Если у производного класса имеется несколько базовых классов, то говорят о множественном наследовании. ***Множественное наследование*** позволяет сочетать в одном производном классе свойства и поведение нескольких классов.

```
#include < iostream.h>
#include < conio.h>
#include < string.h>
class Coord
{  int x, y;
public:
        Coord(int _x, int _y) {x = _x; y = _y;}
        int GetX( ){return x;}
```

```
        int GetY( ) {return y;}
        void SetX(int _x){x = _x;}
        void SetY(int _y){y = _y;}
};
class SaveMsg
{   char Message[80];
public:
       SaveMsg(char* msg){SetMsg(msg);}
        void SetMsg(char* msg) {strcpy(Message, msg);}
        void ShowMsg() {cout << Message;}
};
class PrintMsg: public Coord, public SaveMsg
{public:
       PrintMsg (int _x, int _y, char* msg) : Coord(_x, _y), SaveMsg(msg){ };
       void Show();
};
void PrintMsg::Show()
{        gotoxy(GetX( ),GetY( ));
        Show*Msg ();
}
main( )
{       PrintMsg * ptr;
       ptr =new PrintMsg(10, 5, "Множественное");
       ptr->Show( ) ;
       ptr->SetX(25);
       ptr->SetY(5);
       ptr->SetMsg("наследование");
       ptr->Show() ;
       delete ptr;
       return 0;
}
```

В этом примере класс *Coord* отвечает за хранение и установку значений координат на экране, класс *SaveMsg* хранит и устанавливает сообщение, а класс *PrintMsg*, являющийся производным от них, выводит это сообщение на экран в заданных координатах. Выполнение функции-члена *Show()* приводит к выводу на экран в указанных координатах сообщения, заданного третьим аргументом. Следующие за этим вызовы функций-членов *SetX()*, *SetY()* и *SetMsg()* приводят к установке новых значений координат и нового сообщения, которое выводится повторным вызовом функции *Show()*.

Этот пример демонстрирует также и то, как осуществляется передача параметров конструкторам базовых классов. При множественном наследовании, как и при про-

стом, конструкторы базовых классов вызываются компилятором до вызова конструктора производного класса. Единственная возможность передать им аргументы – использовать список инициализации элементов. Причем порядок объявления базовых классов при наследовании определяет и порядок вызова их конструкторов, которому должен соответствовать порядок следования конструкторов базовых классов в списке инициализации. В данном случае класс *PrintMsg* содержит такое объявление о наследовании:

```
class PrintMsg: public Coord, public SaveMsg
```

Этому объявлению соответствует следующий конструктор:

```
PrintMsg (int _x, int __y, char* msg) : Coord(_x, _y), SaveMsg(msg){}
```

В основной программе этому конструктору передаются аргументы при создании объекта класса *PrintMsg* с помощью оператора *new*:

```
ptr = new PrintMsg(10, 5, "Множественное ");
```

Деструкторы вызываются в обратном порядке.

2.2. Устранение неоднозначностей

Если в двух базовых классах будет объявлены одинаковые по сигнатуре методы, то, когда потребуется вызвать данный метод в классе-наследнике, возникнет вопрос: какой из унаследованных методов при этом будет использован? Ведь методы, объявленные в базовых классах, имеют одинаковые имена и сигнатуры. В результате при компилировании программы возникнет неоднозначность, которую необходимо устранить, иначе компилятор вернет сообщение об ошибке.

Неоднозначность можно устранить явным обращением к необходимой функции:

```
Ob.ClassName::Func();
```

В любом случае при возникновении подобной ситуации, когда необходимо сделать выбор между одноименными методами или переменными-членами различных базовых классов, следует явно указывать имя необходимого базового класса перед именем функции-члена или переменной.

2.3. Проблемы множественного наследования

Хотя множественное наследование имеет ряд преимуществ по сравнению с одиночным, программисты неохотно используют его. Основная проблема состоит в том, что многие компиляторы C++ не поддерживают множественное наследование; это затрудняет отладку программы, тем более что все возможности, реализуемые этим методом, можно получить и без него.

Рекомендуется использовать множественное наследование когда новый класс нуждается в функциях и возможностях из более чем одного базового класса.

2.4. Классы возможностей

Промежуточным решением между одиночным и множественным наследованием классов является использование ***классов возможностей***. Так, класс *Horse* может происходить от двух базовых классов — *Animal* и *Displayable*, причем последний добавляет лишь несколько методов отображения объектов на экране.

Методы класса возможностей передаются в производные классы с помощью обычного наследования. Единственное отличие классов возможностей от других классов состоит в том, что они практически не содержат никаких данных. Различие довольно субъективное и отражает лишь общую тенденцию программирования, сводящуюся к тому, что добавление функциональных возможностей классам не должно сопровождаться усложнением программы.

3. ПОРЯДОК ВЫПОЛНЕНИЯ ЛАБОРАТОРНОЙ РАБОТЫ

1. В ходе самостоятельной подготовки изучить основы работы с потоковым вводом/выводом.
2. Разработать программу на языке C++.
3. Разработать тестовые примеры и выполнить отладку программы.
4. Получить результаты работы программы и исследовать её свойства для различных режимов работы, сформулировать выводы.
5. Оформить отчет по проделанной работе.

4. ВАРИАНТЫ ЗАДАНИЙ

Описать иерархию классов, заданную по варианту. Для каждого класса описать конструкторы и деструктор (по необходимости), функции ввода и вывода значений полей. В каждом классе должны присутствовать минимум одно уникальное поле и один уникальный метод. Проиллюстрировать корректную работу механизма множественного наследования.

Вариант 1

Базовые классы: Юноша (ФИО, год рождения), Учащийся (№зачетки, курс, ср.бал). Класс-наследник: Студент.

Вариант 2

Базовые классы: Человек (ФИО, год рождения), Должность (название, оклад). Класс-наследник: Служащий.

Вариант 3

Базовые классы: Устройство (название, мощность, производитель), Звук (частота, громкость). Класс-наследник: Сирена.

Вариант 4

Базовые классы: Корабль (название, водоизмещение, скорость), Транспорт (тоннаж). Класс-наследник: Сухогруз.

Вариант 5

Базовые классы: Вода (объем), Вещество (вес, название, тип). Класс-наследник: Раствор.

Вариант 6

Базовые классы: Книга (название, издательство), Автор (ФИО, год рождения). Класс-наследник: Произведение.

Вариант 7

Базовые классы: Прибор (цена деления, погрешность вычислений), Спектр (количество составляющих). Класс-наследник: Спектрометр.

Вариант 8

Базовые классы: Машина (марка, мощность, скорость), Транспорт (тоннаж, тип). Класс-наследник: Грузовик.

Вариант 9

Базовые классы: Устройство (название, мощность, производитель), Свет (яркость, цвет). Класс-наследник: Фонарик.

Вариант 10

Базовые классы: Ткань (название, цвет, цена), Стиль (название). Класс-наследник: Костюм.

Вариант 11

Базовые классы: Бумага (цвет, фактура, производитель), Дата (день, месяц, год). Класс-наследник: Календарь.

Вариант 12

Базовые классы: Колесо (диаметр, количество), Обувь (размер, цвет). Класс-наследник: Ролики.

Вариант 13

Базовые классы: Топливо (тип, вес), Окислитель (тип, объем). Класс-наследник: Огонь.

Вариант 14

Базовые классы: Летательный аппарат (тип, скорость полета), Транспорт (тоннаж). Класс-наследник: Шаттл.

Вариант 15

Базовые классы: Птица (крылья, скорость), Лошадь (цвет, кличка). Класс-наследник: Пегас.

5. СОДЕРЖАНИЕ ОТЧЕТА О ВЫПОЛНЕНИИ ЛАБОРАТОРНОЙ РАБОТЫ

Титульный лист, цель работы, вариант задания, текст программы с комментариями, описание тестовых примеров и выводы по проделанной работе.

6. КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Что такое множественное наследование? Привести пример.

2. Пояснить особенности работы конструкторов и деструкторов при множественном наследовании.

3. Перечислите достоинства и недостатки данного механизма ООП. Когда он применяется?

ЛАБОРАТОРНАЯ РАБОТА №7
ШАБЛОНЫ.

1. ЦЕЛЬ РАБОТЫ

Приобретение практических навыков при написании объектно-ориентированных программ с использованием шаблонов функций и классов. Освоение особенностей отладки объектно-ориентированных программ.

2. ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

2.1. Шаблоны и их применение

Шаблоны позволяют давать обобщенные, в смысле произвольности используемых типов данных, определения функций и классов. Поэтому их часто называют параметризованными функциями и параметризованными классами. Часто также используются термины “шаблонные функции” и “шаблонные классы”.

Шаблон функции представляет собой обобщенное определение функции, из которого компилятор автоматически создает представитель функции для заданного пользователем типа (или типов) данных. Когда компилятор создает по шаблону функции конкретную ее реализацию, то говорят, что он создал порожденную функцию. Синтаксис объявления шаблона функции имеет следующий вид:

```
template < class T_1|T_1 идент_1>, . . . , <class Tn |Tn идент_n >
возвр_тип имя_функции (список параметров)   // Заголовок функции
{       /* Тело функции */
}
```

За ключевым словом *template* следуют один или несколько параметров, заключенных в угловые скобки, разделенные между собой запятыми. Каждый параметр является либо ключевым словом *class*, за которым следует имя типа, либо именем типа, за которым следует идентификатор.

Для задания параметризованных типов данных вместо ключевого слова *class* может также использоваться ключевое слово *typename*. Параметры шаблона, следующие за ключевыми словами *class* или *typename*, называют ***параметризованными типами***. Они информируют компилятор о том, что некоторый (пока что неизвестный) тип данных используется в шаблоне в качестве параметра (в момент вызова шаблона на место такого параметризованного типа станет тип, заданный программистом). Па-

раметры шаблона, состоящие из имени типа и следующего за ним имени идентификатора, информируют компилятор о том, что параметром шаблона является константа указанного типа. Шаблонную функцию можно вызывать как обычную, никакого специального синтаксиса не требуется.

Рассмотрим пример определения и вызова шаблонных функций:

```
#include < iostream>
template < class T>          // Шаблон функции, вычисляющей квадрат
                             // введенного значения
T Sqr(T x)
{
   return x*x;
}

template < class T>          // Шаблон функции обмена двух элементов
                             // в массиве по значению их индексов
T* Swap(T* t, int ind1, int ind2)
{
 T tmp = t[ind1];            // Собственно обмен
 t[ind1] = t[ind2];
 t[ind2] = tmp;
 return t;
}
int main ( )
{  int n=10,                 // Для возведения в квадрат
   i = 2, j = 5;             // Индексы в массиве
   double d = 10.21;         // Для возведения в квадрат
 char* str = "Шаблон";       // Массив букв
 cout << "Значение n = " << n << endl
 << "Его квадрат = " << Sqr (n) << endl      // Вызов шаблона для возведения
 << "Значение d = " << d << endl             // в квадрат целого числа
 << Его квадрат = " << Sqr(d) << endl        // Вызов шаблона для возведения
 << "Исходная строка = " << str << endl      // в квадрат вещественного числа
                                             // Преобразование массива
 << "Преобразованная строка = " << Swap(str, i, j) << endl;
return 0;
}
```

При выполнении программа выводит на экран следующее:

```
Значение n = 10
Его квадрат = 100
Значение d = 10.21
Его квадрат = 104.244
```

Исходная строка = Шаблон
Преобразованная строка = Шанлоб

Обратите внимание на вызовы шаблонных функций, сделанные в предыдущем примере: *Sqr(n)* и *Sqr(d)*. Каждый такой вызов приводит к построению конкретной ***порожденной функции***. В данном случае первый вызов приводит к построению компилятором функции *Sqr()* для целого типа данных, во втором — для типа *double*. В связи с этим процесс построения порожденной функции компилятором называют ***конкретизацией шаблонной функции***.

Как и для обычных функций, можно создать прототип шаблона функции в виде его предварительного объявления. Например:

```
template < class T> T* Swap(T* t, int ind1, int ind2);
```

Каждый типовой параметр шаблона должен появиться в списке формальных параметров функции. Например, следующее объявление приведет к ошибке во время компиляции:

```
template < class T, class U>
T FuncName(U);
```

Шаблонная функция может быть также объявлена внешней, статической или встраиваемой (*inline*). В этом случае соответствующий спецификатор располагается вслед за списком параметризированных типов шаблона, а не перед ключевым словом *template*:

```
template extern          //Объявление внешней шаблонной функции
T* Swap(T* t, int ind1, int ind2);

template static          //Объявление статической шаблонной функции
T* Swap(T* t, int ind1, int ind2);

template inline          //Объявление встроенной шаблонной функции
T *Swap(T* t, int ind1, int ind2);
```

2.2. Недостатки шаблонов

Шаблоны предоставляют определенные выгоды при программировании, связанные с широкой применимостью кода и легким его сопровождением. В общем, этот механизм позволяет решать те же задачи, для которых используется полиморфизм. С другой стороны, в отличие от макросов, они позволяют обеспечить безопасное использование типов данных. Однако с их использованием связаны и некоторые недостатки:

- программа содержит полный код для всех порожденных представителей шаблонного класса или функций;
- не для всех типов данных предусмотренная реализация класса или функции оптимальна.

Преодоление второго недостатка возможно с помощью специализации шаблонов.

3. ПОРЯДОК ВЫПОЛНЕНИЯ ЛАБОРАТОРНОЙ РАБОТЫ

1. В ходе самостоятельной подготовки изучить основы работы с потоковым вводом/выводом.

2. Разработать программу на языке C++.

3. Разработать тестовые примеры и выполнить отладку программы.

4. Получить результаты работы программы и исследовать её свойства для различных режимов работы, сформулировать выводы.

5. Оформить отчет по проделанной работе.

4. ВАРИАНТЫ ЗАДАНИЙ

Написать функцию-шаблон, заданную по варианту. Проиллюстрировать ее корректную работу на различных по типу наборах данных (*int*, *char* и др.).

Вариант 1

Написать функцию-шаблон, вычисляющую транспонирование матрицы.

Вариант 2

Написать функцию-шаблон, меняющую местами *i*-ю строку и *j*-й столбец в матрице.

Вариант 3

Написать функцию-шаблон сортировки массива по возрастанию методом пузырька.

Вариант 4

Написать функцию-шаблон, определяющую, существуют ли в матрице повторяющиеся строки.

Вариант 5

Написать функцию-шаблон, определяющую элемент, который встречается в массиве максимальное число раз.

Вариант 6

Написать функцию-шаблон, упорядочивающую строки матрицы по возрастанию значения первого элемента в строке.

Вариант 7

Написать функцию-шаблон, вычисляющую максимальное значение элемента в массиве.

Вариант 8

Написать функцию-шаблон, переставляющую *i*-ю и *j*-ю строки в матрице.

Вариант 9

Написать функцию-шаблон, записывающую массив в обратном порядке.

Вариант 10

Написать функцию-шаблон, меняющую диагонали матрицы местами.

Вариант 11

Написать функцию-шаблон последовательного поиска в массиве по ключу. Функция возвращает индекс первого найденного элемента в массиве, равного ключу.

Вариант 12

Написать функцию-шаблон, заменяющую в матрице все значения элементов, меньшие введенного значения, на него.

Вариант 13

Написать функцию-шаблон, циклически сдвигающую одномерный массив элементов n раз (1й элемент становится 2м, 2й – 3м …*n*-й элемент становится первым).

Вариант 14

Написать функцию-шаблон, упорядочивающую строки в матрице по возрастанию элементов характеристического столбца. Элементы характеристического столбца представляют собой максимальный элемент в строке матрицы.

Вариант 15

Написать функцию-шаблон, проверяющую матрицу на симметричность.

5. СОДЕРЖАНИЕ ОТЧЕТА О ВЫПОЛНЕНИИ ЛАБОРАТОРНОЙ РАБОТЫ

Титульный лист, цель работы, вариант задания, текст программы с комментариями, описание тестовых примеров и выводы по проделанной работе.

6. КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Что такое шаблон? Привести пример.
2. Для чего применяются шаблоны?
3. Перечислить особенности работы шаблонов. Что такое порожденная функция?
4. Перечислить достоинства и недостатки шаблонов.

ЛАБОРАТОРНАЯ РАБОТА №8 ОБРАБОТКА ИСКЛЮЧИТЕЛЬНЫХ СИТУАЦИЙ.

1. ЦЕЛЬ РАБОТЫ

Приобретение практических навыков при написании объектно-ориентированных программ с использованием обработки исключительных ситуаций. Освоение особенностей отладки объектно-ориентированных программ.

2. ТЕОРЕТИЧЕСКИЙ РАЗДЕЛ

2.1. Обработка исключительных ситуаций

Исключение — это некоторое событие, которое является неожиданным и зачастую прерывает нормальный процесс выполнения программы.

Стандарт языка *C++* предусматривает встроенный механизм обработки исключений. Кроме того, компиляторы *Visual C++ 6.0* и *Borland C++ Builder* предусматривают еще один механизм обработки исключений, который реализуется в тесном взаимодействии этих компиляторов с операционной системой и называется структурированной обработкой исключений (*SHE* — от англ. *Structured Exception Handling*). Этот последний механизм был реализован еще в компиляторах языка С, в связи с чем часто называется С-исключениями. Хотя структурированная обработка исключений может быть использована в *C++*, для программ, написанных на этом языке, рекомендуется использовать более новый механизм обработки исключений *C++* (*C++ EH* от англ. *C++ Exception Handling*). Для управления исключениями в C++ используются три ключевых слова: *try, catch u throw*.

Ключевое слово *try* служит для обозначения блока кода, который может генерировать исключение. При этом соответствующий блок заключается в фигурные скобки и называется ***защищенным*** или ***try-блоком***:

```
try
{
        // Защищенный блок кода
}
```

Тело всякой функции, вызываемой из *try*-блока, также принадлежит try-блоку. Предполагается, что одна или несколько функций или инструкций из защищенного блока могут выбрасывать (или генерировать) исключение. Если выброшено исключение, выполнение соответствующей функции или инструкции приостанавливается, все оставшиеся инструкции *try*-блока игнорируются, а управление передается вовне блока try.

Ключевое слово *catch* следует непосредственно за *try*-блоком и обозначает секцию кода, в которую передается управление, если произойдет исключение. За ключевым словом *catch* следует описание исключения, состоящее из имени типа исключения и необязательной переменной, заключенное в круглые скобки:

```
catch (<имя типа>[<перменная>])
{
        //Обработчик исключения
}
```

Имя типа исключения идентифицирует обслуживаемый тип исключений. Блок кода, обрабатывающего исключение, заключается в фигурные скобки и называется ***catch-блоком*** или ***обработчиком*** исключения. При этом говорят, что данный *catch*-блок перехватывает исключения описанного в нем типа. Если исключение перехваче-

но, переменная получает его значение. Если вам не нужен доступ к самому исключению, указывать эту переменную не обязательно. Переменная исключения может иметь любой тип данных, включая созданные пользователем типы классов.

За одним *try*-блоком могут следовать несколько *catch*-блоков. Оператор *catch*, с указанным вместо типа исключения многоточием, перехватывает исключения любого типа и должен быть последним из операторов *catch*, следующих за *try*-блоком. Рассмотрим простой пример обработки исключения:

```
#include < exception.h>
#include < iostream.h>
void func()
{ /*Функция генерирует исключение */ }
//--------------------------------------------------------------------------------

int main()

{
 try            //Пример использования механизма обработки исключений C++
        {
         func();       //Защищенный участок кода
         return 0;
        }
 catch (const char*)     //Обработчик исключения типа const char*
        {
         cout << "Было выброшено исключение типа const char*\n";
         return 1;
        }
 catch (...)            //Обработчик всех необработанных исключений
        {
         cout << "Было выброшено исключение другого типа\n";
         return 1;
        }
}
```

В данном примере функция *func()* может генерировать исключения. В связи с этим она включена в защищенный блок. Перехват исключений осуществляется в двух *catch*-блоках. Первый из них перехватывает исключения, имеющие тип *const char**, второй — любого другого типа.

2.2. Генерирование исключений

Для обозначения в программе места и типа выбрасываемого исключения служит ключевое слово *throw*. Местоположение инструкции *throw* определяет точку выброса исключения. Синтаксис его использования следующий:

***throw* < объект >.**

Операнд в инструкции *throw* не является обязательным. Инструкция *throw* без операнда используется для повторного выбрасывания исключения того же типа, которое обрабатывается в данный момент, следовательно, она может использоваться только в catch-блоках. Рассмотрим пример, демонстрирующий выбрасывание исключений:

```
#include < stdio.h>
bool test;
class SomeException{};        //Класс исключений
void Func (bool bvar)  //Функция, создающая исключительную ситуацию
{
   if (bvar) throw SomeException();        // Вбрасывание исключения
}
int main()
{
        try             // Блок защищенного кода
            {
               test = true;
               Func(true) ;
            }
        catch(SomeException& e)    // Блок обработчика исключения
              {
                 test = false;
              }
        return test ? (puts("test = true.\n"),1) : (puts("test = false.\n"),0);
}
```

При выполнении программа выводит на экран:

test = false.

2.3. Перехватывание исключений

Если в некоторой точке выбрасывается исключение, поток выполнения программы прерывается и происходит следующее:

1. Если выброшена переменная встроенного типа или объект класса по значению, создается копия выброшенной переменной (причем в последнем случае используется конструктор копирования); если выброшена переменная по ссылке, копирование не производится.

2. Осуществляется поиск ближайшего обработчика, принимающего параметр, совместимый по типу с выброшенной переменной;

3. Если обработчик найден, управление программой передается найденному обработчику.

4. Если обработчик не найден, можно вызвать *set_terminate()*, предоставив обработчик завершения; в противном случае программа вызывает функцию завершения.

Если исключение не выброшено, программа выполняется обычным образом.

При поиске соответствующего типа исключения компилятор пользуется следующими правилами. Обработчик считается найденным, если:

a) тип выброшенной переменной совпадает с типом, ожидаемым обработчиком, либо может быть приведен к нему (другими словами, если тип выброшенной переменной есть *T*, соответствующими ему признаются обработчики, перехватывающие исключение типа *T*, *const T, T&* и *const T&*);

b) тип выброшенной переменной представляет собой указатель, тип которого может быть преобразован к типу указателя, перехватываемого обработчиком;

c) если выброшенная переменная представляет собой объект некоторого класса, а тип перехватываемого исключения является базовым классом, наследуемым этим объектом как *public*.

Следует обратить внимание на то, что компилятор ищет ближайший обработчик, принимающий параметр, совместимый по типу с выброшенной переменной. Этот момент не следует забывать и внимательно следить за порядком, в котором располагаются в программе обработчики исключений для данного *try*-блока. Обработчик, ожидающий исключение базового класса, скрывает обработчик производного класса. Точно так же обработчик для указателя типа *void** скрывает обработчик для указателя любого типа.

2.4. Использование вложенных блоков *try/catch*

Иногда возникает необходимость использования вложенных блоков *try/catch*. Для этого в языке C++ нет никаких препятствий, однако должно соблюдаться единственное требование: за каждым *try*-блоком обязательно должен стоять *catch*-блок.

Часто вложенные *try/catch* блоки возникают неявно, в результате создания защищенного блока в функции, которая сама находится в защищенном блоке.

К вложенным *try/catch*-блокам приходится прибегать потому, что какая-то функция может непредвиденно привести к исключению. В этом случае простейший выход — заключить весь код в функции *main()* в такой блок, причем для перехвата исключения использовать обработчик вида *catch(...)*. Затем можно использовать средства, предоставляемые компилятором для определения места в программе, вызвавшего появление исключения.

3. ПОРЯДОК ВЫПОЛНЕНИЯ ЛАБОРАТОРНОЙ РАБОТЫ

1. В ходе самостоятельной подготовки изучить основы работы с потоковым вводом/выводом.

2. Разработать программу на языке C++.

3. Разработать тестовые примеры и выполнить отладку программы.

4. Получить результаты работы программы и исследовать её свойства для различных режимов работы, сформулировать выводы.

5. Оформить отчет по проделанной работе.

4. ВАРИАНТЫ ЗАДАНИЙ

Вариант 1

Создать класс массив букв. Описать перехват ошибок, связанных с недопустимым индексом массива. Пример: *MyClass Mass(13); cout<<Mass.GetValue(27).*

Вариант 2

Найти значение математического выражения, описав перехват ошибок вычислений: $y = 14(\ln(x))^2 + x^2$.

Вариант 3

Найти значение математического выражения, описав перехват ошибок вычислений: $y = \dfrac{3.8\sinh(x)}{36x^3}$.

Вариант 4

Найти значение математического выражения, описав перехват ошибок вычислений: $y = 99x^2 + \sinh(x)$.

Вариант 5

Создать класс Вектор (*float **). Описать перехват ошибок, связанных с неверным вводом значений. Пример: ввели букву вместо цифры.

Вариант 6

Найти значение математического выражения, описав перехват ошибок вычислений: $y = \dfrac{1}{x^2 + 4x + 4}$.

Вариант 7

Найти значение математического выражения, описав перехват ошибок вычислений: $y = ctg(x) * x^2$.

Вариант 8

Найти значение математического выражения, описав перехват ошибок вычислений: $y = 10\arcsin(10x + 2.2)$.

Вариант 9

Найти значение математического выражения, описав перехват ошибок вычислений: $y = \dfrac{10x}{\arccos(x^2)}$.

Вариант 10

Создать класс МойФайл, содержащий строку — путь к файлу. Описать перехват ошибок, связанных с некорректной работой с файлом. Пример: попытка открыть файл, не существующий по данному пути.

Вариант 11

Найти значение математического выражения, описав перехват ошибок вычислений: $y = 12x + \sqrt{x-8}$.

Вариант 12

Найти значение математического выражения, описав перехват ошибок вычислений: $y = \dfrac{\sqrt{x^3}}{tg(x)}$.

Вариант 13

Найти значение математического выражения, описав перехват ошибок вычислений: $y = \ln(\cos(1/x))$.

Вариант 14

Найти значение математического выражения, описав перехват ошибок вычислений: $y = \ln(1/\arctan(x))$.

Вариант 15

Перехватить исключение типа переполнение значения переменной, например, при вычислении уравнения для целочисленного типа результата:

$y = 10x^{10}$.

5. СОДЕРЖАНИЕ ОТЧЕТА О ВЫПОЛНЕНИИ ЛАБОРАТОРНОЙ РАБОТЫ

Титульный лист, цель работы, вариант задания, текст программы с комментариями, описание тестовых примеров и выводы по проделанной работе.

6. КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Что такое исключение? Когда оно возникает?
2. Пояснить механизм обработки исключений.
3. Как происходит перехват исключения?
4. Можно ли использовать вложенные конструкции *try/catch*? Когда и при каких условиях?

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. Буч Г. Объектно-ориентированное проектирование с примерами применения / Г.Буч. — М.: Конкорд, 1992. — 519 с.

2. Дейтел Х. М. Как программировать на C++ (полное издание) / Х. М. Дейтел, П. Дж. Дейтел.; под ред. В.В.Тимофеева. — М.: Бином, 2008. — 1454 с.

3. Айра Пол. Объектно-ориентированное программирование на C++ /Айра Пол. — СПб: “Невский диалект”, 2001. — 464 с.

4. Павловская Т.А. Программирование на языке высокого уровня C/C++ / Т.А. Павловская. — Питер, 2003. — 461с.

5. Лаптев В.В. C++. Объектно-ориентированное программирование. Задачи и упражнения. / В.В. Лаптев, А.В. Морозов, А.В. Бокова.; под ред. В.В. Лаптева. — СПб.: Питер, 2007. — 288с.

6. Иванова Г.С Объектно-ориентированное программирование: учеб для студ. вузов/ Г.С. Иванова, Т.Н. Ничушкина, Е.К. Пугачев; под ред. Г.С.Ивановой. — М.: Изд-во МГТУ им. Н.Э.Баумана, 2001. — 320 с.

7. Павловская Т.А. C/C++. Программирование на языке высокого уровня/ Т.А. Павловская. — СПб: Питер, 2003. — 461с.

Заказ № _____ от «_____» _____2012г. Тираж _____экз.

Изд-во СевНТУ