ФЕДЕРАЛЬНОЕ АГЕНТСТВО ВОЗДУШНОГО ТРАНСПОРТА
(РОСАВИАЦИЯ)

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«МОСКОВСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ
УНИВЕРСИТЕТ ГРАЖДАНСКОЙ АВИАЦИИ» (МГТУ ГА)

Кафедра вычислительных машин, комплексов, систем и сетей

Л.А. Надейкина

ПРОГРАММИРОВАНИЕ

Учебно-методическое пособие
по выполнению лабораторных работ № 9, 10, 11

для студентов I курса
направления 09.03.01
очной формы обучения

Москва
ИД Академии Жуковского
2021

УДК 004.42
ББК 6Ф7.3
Н17

Рецензент:
Черкасова Н.И. – канд. физ.-мат. наук

Надейкина Л.А.

Н17 Программирование [Текст] : учебно-методическое пособие по выполнению лабораторных работ № 9, 10, 11 / Л.А. Надейкина. – М.: ИД Академии Жуковского, 2021. – 48 с.

Данное учебное пособие издается в соответствии с учебным планом для студентов I курса направления 09.03.01 «Информатика и вычислительная техника» (бакалавриат) очной формы обучения.

Рассмотрено и одобрено на заседаниях кафедры 26.10.2021 г. и методического совета 26.10.2021 г.

УДК 004.42
ББК 6Ф7.3

В авторской редакции

Подписано в печать 23.11.2021 г.
Формат 60x84/16 Печ. л. 3 Усл. печ. л. 2,79
Заказ № 872/1004-УМП31 Тираж 30 экз.

Московский государственный технический университет ГА
125993, Москва, Кронштадтский бульвар, д. 20

Издательский дом Академии имени Н. Е. Жуковского
125167, Москва, 8-го Марта 4-я ул., д. 6А
Тел.: (495) 973-45-68
E-mail: zakaz@itsbook.ru

© Московский государственный технический университет гражданской авиации, 2021

1. ЛАБОРАТОРНАЯ РАБОТА № 9

Наследование с использованием виртуальных функций

1.1. Цель лабораторной работы

Целью лабораторной работы является получение практических навыков применения основных методов наследования классов, создания иерархий классов и использования полиморфизма, основанного на механизме виртуальных функций при создании связанного списка полиморфных объектов класса.

1.2. Теоретические сведения

Включение и наследование классов

Семантика отношений между классами может быть реализована по схеме ***наследования*** и по схеме ***включения***.

Об отношении ***включения*** говорят, используя выражение “включает как часть” (*has a* – владеет, содержит в себе как часть).

При ***наследовании*** базовый класс – представляет объекты общего вида, производный класс описывает более конкретные объекты, которые являются разновидностью (частным случаем объектов базового класса).

Об отношении наследования можно сказать, используя выражение “является частным случаем” *(is a)*.

Например, самолет является частным случаем транспортного средства. Это отношение наследования классов.

Самолет имеет крылья, мотор – здесь отношение включения.

Наследование

Наследование – одна из наиболее фундаментальных концепций ООП.

Суть концепции в следующем. Одни классы можно трактовать как, классы для определения объектов общего вида, так называемые ***базовые классы***. Пользователь может создавать ***производные классы (порожденные, классы потомки, наследники),*** которые описывают более конкретные объекты, являющиеся разновидностью объектов базового класса. Классы потомки могут наследовать возможности родительских базовых классов (поля данных и методы), при этом производные классы могут пополняться собственными компонентами (данными и собственными методами).

Наследование - это механизм получения нового класса на основе уже существующего. Существующий класс может быть дополнен или изменен для создания нового класса.

Существующие классы называются ***базовыми***, а новые – ***производными***.

Производный класс получает в наследство поля данных и методы базового класса. При этом наследуемые компоненты не перемещаются в производный класс, а остаются в базовом классе. В каждый объект производного класса входит безымянный объект базового класса со всеми своими полями и методами.

Допускается ***множественное наследование*** - возможность для некоторого класса наследовать компоненты нескольких базовых классов, несвязанных между собой.

Простейший синтаксис определения (спецификации) производного класса:

ключ_класса имя_производного_класса:
список_спецификаторов_базовых_классов
{поля_данных_и_методы_производного_класса};

где ***ключ_класса*** – одно из служебных слов ***struct, class.*** Следует обратить внимание, что ни базовый, ни производный класс не могут быть объявлены с помощью ***union.*** Классы ***union*** не могут использоваться при наследовании**!**

Спецификаторы базовых классов в списке разделены запятыми и могут быть представлены одним из следующих конструкций:

1) *спецификатор_доступа имя_класса*
2) *virtual спецификатор_доступа имя_класса*
3) *спецификатор_доступа virtual имя_класса*

Производный класс, получая в наследство поля и методы базового класса, не перемещает к себе наследуемые компоненты, они остаются в базовом классе. Однако в каждый объект производного класса входит безымянный объект базового класса со всеми своими полями данных и методами.

При наследовании классов важную роль играет статус доступа компонентов базового класса и спецификатор доступа в определении производного класса.

При наследовании относительно доступности компонентов принято следующее соглашение:

1) ***private*** – член класса может использоваться только функциями – членами данного класса и функциями – “друзьями” своего класса. В производном классе он недоступен.
2) ***protected*** – то же, что и ***private,*** но дополнительно член класса с данным атрибутом доступа может использоваться функциями-членами и функциями – “друзьями” классов, производных от данного.
3) ***public*** – член класса может использоваться любой функцией, которая является членом данного или производного класса, а также к ***public*** - членам возможен доступ извне через имя объекта.

Из наследуемых компонентов базового класса для объектов производного класса доступны компоненты со статусом ***public*** *и* ***protected***.

Любой производный класс может в свою очередь быть базовым для других классов и таким образом формируется структура, называемая ***иерархией классов***, определяющая для каждого класса приложения родственные связи (“родитель - потомок”) его с другими классами приложения.

Класс является ***прямым базовым классом***, если он входит в список базовых классов при определении производного класса.

А если сам базовый класс является производным от некоторого родителя, причем этот родитель не входит в список базовых классов, то этот родитель является ***непрямым (косвенным) базовым классом.***

Иерархию производных классов принято отображать в виде ***направленного ациклического графа (НАГ)***, где стрелкой изображают связь “ производный от”.

Производные классы располагаются ниже базовых. В том же порядке они должны располагаться в программе и так их объявления рассматривает компилятор.

A (базовый класс – прямая база для **B**)

B (производный от **A** класс **–** прямая база для **C**)

C (производный класс – с прямой базой **B** и косвенной **A**)

На практике часто возникает необходимость создать производный класс, наследующий возможности нескольких классов.

При создании объекта производного класса сначала автоматически вызывается конструктор базового класса, который участвует в создании объекта базового класса, после этого вызывается конструктор производного класса, который проводит инициализацию полей данных производного класса.

Деструкторы автоматически вызываются в обратном порядке в соответствии с порядком уничтожения объекта. Сначала уничтожается то, что добавилось в производном классе, а затем и базовая часть.

Полиморфизм виртуальных функций

Термин ***полиморфизм*** дословно означает "множество форм".

Применительно к языкам программирования, говорят о полиморфизме, когда одни и те же функции или операторы в различных условиях проявляют себя по-разному.

Процедурный полиморфизм – это ***перегрузка функций***. В этом случае имя функции становится многозначным – ему соответствуют разные алгоритмы. Еще вид процедурного полиморфизма – это ***перегрузка операций.***

Виртуальные функции включены в ***C++*** для обеспечения еще одного вида ***полиморфизма.*** Этот вид полиморфизма связан с наследованием и реализуется с помощью механизма виртуальных функций.

К механизму виртуальных функций обращаются в тех случаях, когда в каждом производном классе требуется свой вариант некоторой компонентной функции. Классы, включающие такие функции, называются ***полиморфными*** и играют особую роль в ООП.

Виртуальные функции предоставляют механизм ***позднего (отложенного)*** или ***динамического связывания.*** Любая нестатическая

функция базового класса может быть сделана виртуальной, для чего используется ключевое слово ***virtual***.

Механизм виртуальных функций позволяет с помощью ***указателя с типом базового класса*** обращаться к ***переопределенным методам в производных классах***.

Рассматривается указатель на базовый класс, который может содержать как адреса объектов своего класса, так и объектов производных классов.

Конечно, ведь каждый объект производного класса содержит базовую часть (безымянный базовый объект), причем он располагается вначале, а затем данные производного класса.

Поэтому адрес объекта производного класса по значению совпадает с адресом ему принадлежащего базового объекта.

Вызов через указатель обычных компонентных функций (не виртуальных) зависит от ***типа указателя***. Если указатель на базовый класс, то вызывается базовая функция, если указатель на производный класс, то вызывается функция производного класса.

Проблема доступа к методам, переопределенным в производных классах, через указатель на базовый класс решается в C++ посредством использования ***виртуальных функций***.

Чтобы сделать некоторый нестатический метод виртуальным, надо в базовом классе предварить его заголовок спецификатором ***virtual***, метод становится ***виртуальной функцией.***

Если эту функцию переопределить в производном классе даже без спецификатора ***virtual,*** в производном классе также создается ***виртуальная функция.***

Виртуальность наследуется. После того как функция определена как виртуальная, ее повторное определение в производном классе (с тем же самым прототипом) создает в этом классе новую виртуальную функцию, причем спецификатор ***virtual*** может не использоваться.

Конструкторы не могут быть виртуальными, в отличие от деструкторов.

Сигнатуры функций должны различаться только их принадлежность разным классам, ***а алгоритмы могут быть различными!***

Все сказанное похоже на механизм замещения.

Однако виртуальность этих функций (или полиморфизм) проявляется в том, что выбор требуемой из множества определенных в иерархии классов виртуальных функций с одним именем осуществляется не во время компиляции программы, а динамически, ***по конкретному <u>значению указателя базового типа</u>***, ***с помощью которого и вызывается функция.***

Какая функция будет вызываться зависит от типа того объекта, адрес которого присвоен указателю базового типа.

Виртуальность функции проявляется только в том случае, если она вызывается <u>через указатель или ссылку на базовый класс</u>.

Указатель на базовый класс может принимать конкретные значения. Если значение указателя к моменту вызова функции есть адрес объекта базового класса, вызывается вариант функции из базового класса.

Если этот указатель имеет значение адреса объекта производного класса (фактически указывает на данные базового класса в объекте производного класса), то вызывается вариант функции из производного класса.

Рассмотрим вышесказанное на примере.

```
class A {
public:
virtual  void   F1 ( );
virtual  int    F2 ( char* );
 };
class B : public A {
public:
virtual  void   F1 ( );
virtual int    F2 ( char* );
};
class C : public A {
public:
void   F1 ( );
int    F2 ( char* );
};
int main ()  {
A * ap = new A;
B * bp = new B;
C * cp = new C;
ap->F1 ( );              // вызов  функции базового класса A
ap = bp ;
ap->F1 ( );            // вызов замещенной функции класса B
ap = cp;
ap ->F1 ( ) ;        // вызов замещенной функции класса C
return 0;
}
```

Если бы функции ***F1*** были бы обычными компонентными функциями, то при всех трех вызовах вызывалась бы функция базового класса.

Через указатель на базовый класс нельзя обращаться к не виртуальным компонентной функции производного класса.

Механизм виртуальных функций снимает этот запрет и позволяет с помощью указателя на базовый класс обращаться к виртуальным функциям производных классов (после присваивания этому указателю значения указателя на соответствующий объект производного класса).

Отметим важный момент.

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

Чистая виртуальные функции. Абстрактный класс.

Как правило, в конкретных задачах вызываются лишь функции производных классов. "Исходная" виртуальная функция базового класса часто нужна только для того, чтобы в производных классах было, что замещать.

Содержимое базовой функции в этом случае не имеет значения и может быть пустым:

```
class A  {
public :
virtual void Func ()   { }
};
```

Также можно объявить в базовом классе ***чистую виртуальную функцию***, которая вводится с помощью такого определения:

virtual тип имя (спецификация параметров) = 0;

Это некоторая абстракция и такую функцию обязательно надо замещать в производных класса, в которых она и наполнится разумным содержанием.

Объявление такой функции в базовом классе носит формальный характер для указания на виртуальность функций с данным именем.

Класс, в котором есть хотя бы одна чистая виртуальная функция, называется ***абстрактным классом*.**

Чистая виртуальная функция ничего не делает и недоступна для вызовов. Ее назначение – служить основой для подменяющих ее функций в производных классах. Абстрактный класс может использоваться только в качестве базового для производных классов.

Перечислим основные свойства ***абстрактного класса:***

- невозможно создать самостоятельных объектов абстрактного класса.
- ***абстрактный класс*** может использоваться только в качестве базового класса.
- если в производном классе от абстрактного базового класса происходит замещение чистой виртуальной функции, то производный класс не является абстрактным.
- если замещение не производится, то производный класс также является абстрактным.

Пример:

```
class A {  // абстрактный класс
```

```
public:
virtual int Func (char*)   = 0;
void F ( );
};
class B: public A {          //B - не абстрактный класс
int Func (char*);
};
class C: public A {          // C – также абстрактный класс
void F ( );
};
```

- ***абстрактные классы*** предназначены ***для представления общих понятий,*** которые предстоит конкретизировать. На базе общих понятий строятся частные производные классы для ***описания уже конкретных объектов***.

- ***абстрактный класс*** может иметь поля данных, а также методы, отличные от чисто виртуальных, и как всякий класс может иметь явно определенный конструктор.

- ***конструктор абстрактного класса не может использоваться для создания объектов, но может использоваться при наследовании.*** С его помощью инициализируются поля данных абстрактного базового класса, входящие в объект производного класса.

- ***указатель на абстрактный класс*** может использоваться в качестве формального параметра. Соответствующий фактический параметр должен иметь тип указателя на объекты производного (уже не абстрактного) класса.

- ***абстрактный класс*** может быть производным как от абстрактного, так и от обычного классов.

Статические члены класса

Такие компоненты должны быть определены в классе со спецификатором ***static***. Статические данные классов не дублируются при создании объектов, то есть каждый статический компонент существует в единственном экземпляре для всех объектов класса. Доступ к статическому компоненту возможен только после его инициализации. Для инициализации используется конструкция

тип имя_класса : : имя_данного инициализатор;

Например,

int complex : : count = 0;

Это предложение должно быть размещено в глобальной области после определения класса. Только при инициализации статический компонент класса получает память и становится доступным. Обращаться к статическому данному класса можно обычным образом через имя объекта:

имя_объекта.имя_компонента

К статическим компонентам после их инициализации можно обращаться и тогда, когда объекты класса еще не определены. Доступ к статическим компонентам возможен не только через имя объекта, но и через имя класса:

имя_класса : : имя_компонента

Однако так можно извне обращаться только к открытым, ***public*** компонентам.

Обращаться к ***private*** статической компоненте извне можно с помощью открытых методов класса. К моменту обращения к статическим данным класса, объекты класса могут быть еще не определены. Без имени объекта обычный метод класса вызвать нельзя в соответствии с требованиями синтаксиса. Хотелось бы иметь возможность обойтись без имени конкретного объекта при обращении к статическим данным. Такую возможность дают ***статические компонентные функции.*** Статические методы класса можно вызывать, используя квалифицированное имя функции:

имя_класса : : имя_статической_функции

Пример,

```
#include <iostream>
using namespace std;
class TPoint {
  double x, y;
  static int N;  // статический компонент: количество точек
   public:
  TPoint (double x1 = 0.0, double y1 = 0.0) { N++; x = x1; y = y1;}
  static int& count () {return N;} // статический компонент-функция
};
int TPoint :: N = 0; //инициализация статического компонента
int main () {
  TPoint A (1.0,2.0);
  TPoint B (4.0,5.0);
  TPoint C (7.0,8.0);
  cout<< \nОпределены" <<TPoint : : count () << "точки";
return 0;
}
```

Указатель this

Когда функция-член класса вызывается для обработки данных конкретного объекта, этой функции автоматически и неявно передается указатель на тот объект, для которого функция вызвана. Этот указатель имеет имя this и неявно определен в каждой функции класса следующим образом:

имя_класса *const this = адрес_объекта

Указатель ***this*** является дополнительным скрытым параметром каждой нестатической компонентной функции. При входе в тело принадлежащей классу функции ***this*** инициализируется значением адреса того объекта, для которого вызвана функция. В результате этого объект становится доступным внутри этой функции.

В большинстве случаев использование ***this*** является неявным. В частности, каждое обращение к нестатической функции-члену класса неявно использует ***this*** для доступа к члену соответствующего объекта.

Примером широко распространенного явного использования ***this*** являются операции со связанными списками.

1.3 Задание на выполнение лабораторной работы

Написать программу, в которой создается иерархия классов. Использовать механизм виртуальных функций при включении полиморфных объектов в связанный список. Применить статические компоненты класса.

1.4 Порядок выполнения

1) Определить иерархию классов (в соответствии с вариантом).
2) Определить в базовом классе статический компонент - указатель на начало (вершину) связанного списка объектов и статические функции для просмотра списка и очистки списка. В базовом классе объявить чистую виртуальную функцию для просмотра объекта.
3) Реализовать производные классы в соответствии с иерархией классов.
4) Написать демонстрационную программу, в которой создаются объекты различных производных классов и помещаются в список, после чего список просматривается и очищается.
5) Определить методы классов для просмотра объектов и деструкторы виртуальными.
6) Реализовать вариант, когда объект добавляется в список при его создании, то есть в конструкторе.

1.5 Методические указания

1) Для определения иерархии классов связать отношением наследования классы, приведенные для заданного варианта. Из перечисленных классов выбрать один, который будет стоять во главе иерархии. Это абстрактный класс.
2) Определить в классах все необходимые конструкторы и деструктор.
3) Компонентные данные класса специфицировать как ***protected***.
4) Пример определения статических компонентов:

static person* begin; // указатель на начало списка
static void print(void); **//** просмотр списка
static void clear(void); //освобождение списка

5) Статическую компоненту инициализировать вне определения класса, в глобальной области.
6) Добавление объекта в список можно осуществлять следующими описанными ниже способами.

- Во-первых, можно предусмотреть метод класса, то есть объект сам добавляет себя в список. Например, ***a.Add()*** - объект ***a*** добавляет себя в список.
- Во-вторых, включение объекта в список можно выполнять при создании объекта, то есть поместить операторы включения создаваемого объекта в конструктор класса. В случае иерархии классов, включение объекта в список должен выполнять только конструктор базового класса. Программа должна продемонстрировать оба этих способа.

7) Список будет состоять из объектов производных классов. Список просматривать путем вызова виртуального метода ***show*** для каждого объекта списка.

8) Статический метод просмотра списка вызывать не через объект, а через класс.

9) Если производные классы ресурсоемкие, то деструктор в базовом классе следует объявлять виртуальным. Чтобы не осталось не освобожденных ресурсов при уже удаленных объектах.

10) Определение классов, их реализацию, демонстрационную программу поместить в отдельные файлы.

1.6 Содержание отчета

1) Титульный лист.
2) Техническое задание
3) Иерархия классов в виде графа.
4) Определение пользовательских классов с комментариями.
5) Реализация конструкторов с параметрами и деструктора.
6) Реализация методов для добавления объектов в список.
7) Реализация методов для просмотра списка и очищения списка.
8) Листинг демонстрационной программы.
9) Дать обоснование использования виртуальных функций.

1.7 Контрольные вопросы

1) Применение статического элемента класса в связанных списках объектов класса.
2) Указатель ***this***. Применение указателя ***this*** в связанных списках объектов класса.
3) Различие между копированием и присваиванием. Блокировка копирования и присваивания.
4) Преобразование типов в классах пользователя, явные и неявные.
5) Отношения включения классов и наследования классов.
6) Наследование. Суть метода. Определение производного класса. Влияния формата определения производного классов и спецификаторов доступа на доступ наследуемых элементов.
7) Наследование. Передача параметров конструктора в базовый класс. Конструкторы с инициализацией по умолчанию в иерархии классов.
8) Множественное наследование. Порядок вызовов конструкторов и деструкторов базовых классов при множественном наследовании.
9) Множественное наследование. Прямое и косвенное наследование.
10) Иерархия производных классов в виде графа (НАГ).
11) Дублирование объектов базового класса, косвенно наследуемого при множественном наследовании.
12) Виртуальные базовые классы. Примеры иерархии классов (НАГ), с участием виртуальных базовых классов**.**

13) Полиморфизм. Понятие виртуальной функции. Режимы раннего и позднего связывания. Полиморфные классы.

14) Замещение функций в производных классах. Виртуальные функции.

15) Пустая и чистая виртуальные функции. Абстрактный класс, назначение, свойства.

16) Преобразование типов указателей в иерархии классов. Работа с виртуальными функциями.

1.8 Варианты задания

Перечень классов:

1) студент, преподаватель, персона, зав. кафедрой;
2) служащий, персона, рабочий, инженер;
3) рабочий, кадры, инженер, администрация;
4) деталь, механизм, изделие, узел;
5) организация, страховая компания, судостроительная компания, завод;
6) журнал, книга, печатное издание, учебник;
7) тест, экзамен, выпускной экзамен, испытание;
8) место, область, город, мегаполис;
9) игрушка, продукт, товар, молочный продукт;
10) квитанция, накладная, документ, чек;
11) автомобиль, поезд, транспортное средство, экспресс;
12) двигатель, двигатель внутреннего сгорания, дизель, турбореактивный двигатель;
13) республика, монархия, королевство, государство;
14) млекопитающие, парнокопытные, птицы, животное;
15) корабль, пароход, парусник, корвет.

2 ЛАБОРАТОРНАЯ РАБОТА № 10

Обработка типовых исключений. Исключения типа стандартных данных, исключения - классы.

2.1. Цель лабораторной работы

Целью лабораторной работы является получение навыков программирования с использованием механизма обработки исключительных ситуаций.

2.2. Теоретические сведения

Общие сведения об исключениях

Исключительная ситуация, или исключение — это возникновение во время выполнения программы непредвиденного или аварийного события, которое делает дальнейшие выполнение программы в соответствии с базовым алгоритмом невозможными или бессмысленными.

Есть целый ряд встроенных ситуаций, таких как

- деление на ноль,
- достижение конца файла,

- переполнение в арифметических операциях,
- обращение по несуществующему адресу памяти

и т. п.

Обычно эти события приводят к завершению программы с системным сообщением об ошибке.

C++ дает программисту возможность восстанавливать программу и продолжать ее выполнение. Делается это с помощью исключений.

Исключительные ситуации, можно разделить на два основных типа: ***синхронные и асинхронные***.

Синхронные исключения могут возникнуть только в определённых, заранее известных точках программы. Так, например, ошибка чтения файла или нехватка памяти — синхронные исключения, так как возникают они только в операции чтения из файла или в операции выделения памяти соответственно.

Асинхронные исключения могут возникать в любой момент времени и не зависят от того, какую конкретно инструкцию программы выполняет система. Типичные примеры таких исключений: аварийный отказ питания.

Исключения C++ не поддерживают обработку асинхронных событий, таких, как обработку аппаратных прерываний, например, нажатие клавиш Ctrl+C.

Механизм исключений предназначен только для событий, которые происходят в результате работы самой программы.

Исключения позволяют логически разделить вычислительный процесс на две части — обнаружение аварийной ситуации и ее обработку.

Обработка исключений обеспечивает механизм, позволяющий отделить обработку ошибок или других исключительных обстоятельств от общего потока выполнения кода.

Это важно не только для лучшей структуризации программы.

Главной причиной является то, что функция, обнаружившая ошибку, может не знать, что предпринимать для ее исправления, а использующий эту функцию код может знать, что делать, но не уметь определить место возникновения.

Кроме того, используя механизм исключений, для передачи информации об ошибке в вызывающую функцию не требуется применять возвращаемое значение, параметры или глобальные переменные, поэтому интерфейс функций не раздувается.

Это важно, например, для ***конструкторов*** классов, которые не могут возвращать значение.

Механизм обработки исключений является общим средством управления программой. С его помощью можно обрабатывать не только аварийные, но и любые другие ситуации, возникшие в результате выполнения программы.

Но именно для выхода из тупиковых положений служит механизм исключений – средство, позволяющее отделить выявление особой ситуации от обработки информации о ней.

Механизм обработки исключений (МОИ)

Для реализации МОИ в язык C++ введены следующие ключевые слова: ***try*** (контролировать, пытаться, пробовать), ***catch*** (ловить, перехватывать), ***throw*** (бросать, генерировать, посылать).

Общая схема посылки и обработки исключений:

```
try {
операторы
throw выражение_1;
операторы
throw выражение_2
операторы
}
catch (спецификация_исключения_1)
{операторы_обработки_исключения_1}
catch (спецификация_исключения_2)
{операторы_обработки_исключения_2}
```

Место, в котором может произойти ошибка, должно входить в контролируемый блок — составной оператор, перед которым записано ключевое слово ***try***:

try {операторы}

Среди операторов, заключенных в фигурные скобки, могут быть любые операторы языка, описания объектов и функций, определения локальных переменных.

Кроме того, в блок контроля за исключениями помещаются специальные операторы, генерирующие исключения и имеющие формат:

throw выражение;

С помощью выражения формируется специальный объект, называемый ***исключением***.

Например:

```
throw -1; // генерация исключения типа int
throw ENUM_INVALID_INDEX; // генерация исключения типа enum
throw "Cannot take square root of negative number"; // генерация
                    //исключения типа const char* (строка C-style)
throw dX; // генерация исключения типа double (если dX - переменная типа
          // double, которая была определена ранее)
throw MyException ("Fatal Error"); // генерация исключения с
                                   //использованием объекта класса MyException
```

Каждая из этих строк сигнализирует о том, что возникла какая-то ошибка, которую нужно обработать.

Блок ***try*** действует как наблюдатель в поисках исключений, которые были выброшены каким-либо из операторов в этом же блоке ***try,*** например:

```
try {
```

// Здесь операторы, которые могут генерировать следующее исключение
throw -1;
}

Блок ***try*** не определяет, как будет обрабатываться исключение. Он просто сообщает компилятору, что внутри этого блока сгенерировано исключение и надо "поймать его и обработать".

Все исключения создаются как временные объекты, а тип и значение каждого исключения определяется формирующим его выражением.

Оператор ***throw*** выполняет посылку исключения, то есть передает управление и пересылает исключение непосредственно за блок контроля.

В этом месте обязательно должна располагается ловушка или обработчик исключения, ловушек может быть несколько (как правило, сколько исключений).

Обработчики исключений должны располагаться непосредственно за ***try-блоком.*** Они начинаются с ключевого слова ***catch***, за которым в скобках следует тип обрабатываемого исключения. Обработчик имеет формат:

catch (спецификация_исключения)
{операторы_обработки_исключения}

Можно записать один или несколько обработчиков в соответствии с типами обрабатываемых исключений.

Внешне и функционально обработчик исключений похож на определение функции с одним параметром – типом исключения, не возвращающей значения.

Когда за блоком контроля размещены несколько ловушек, то они должны отличаться друг от друга.

Существует три формы записи:

catch (тип исключения имя) {/* тело обработчика */}

catch(тип_исключения) {/* тело обработчика */}

catch(...) {/* тело обработчика */}

Первая форма применяется, когда имя параметра используется в теле обработчика для выполнения каких-либо действий — например, вывода информации об исключении.

Вторая форма не предполагает использования информации об исключении, играет роль только его тип.

Многоточие обозначает, что обработчик перехватывает все исключения (***catch-all***). Так как обработчики просматриваются в том порядке, в котором они записаны, обработчик третьего типа следует помещать после всех остальных.

Пример:

catch (int i){ *// Обработка исключений типа* ***int***
}
catch (const char *) {// Обработка исключений типа const char*
}
catch (Overflow) {// Обработка исключений класса Overflow

}

catch (...) {// Обработка всех необслуженных исключений

}

После обработки исключения управление передается первому оператору, находящемуся непосредственно за обработчиками исключений.

Туда же, минуя код всех обработчиков, передается управление, если исключение в ***try-блоке*** не было сгенерировано.

Перехват исключений

Когда с помощью выражения в операторе ***throw*** генерируется исключение в контролируемом блоке, функции исполнительной библиотеки C++ выполняют следующие действия:

1) создают копию параметра ***throw*** в виде статического объекта, который существует до тех пор, пока исключение не будет обработано;

2) осуществляют поиск подходящего обработчика; одновременно, вызываются деструкторы локальных объектов, выходящих из области действия;

3) передают объект исключения и управление обработчику, имеющему параметр, совместимый по типу с этим объектом.

Обработчик считается найденным, если тип выражения после throw:

1) тот же, что и указанный в параметре ***catch***, параметр может быть записан в форме ***T***, ***const T, T&*** или ***const T&,*** где ***T***— тип исключения;

2) является производным от указанного в параметре ***catch*** (если наследование производилось с ключом доступа ***public***);

3) является указателем, который может быть преобразован по стандартным правилам преобразования указателей к типу указателя в параметре ***catch***.

Из вышеизложенного следует, что

- обработчики производных классов следует размещать до обработчиков базовых, поскольку в противном случае им никогда не будет передано управление;
- обработчик указателя типа ***void*** автоматически скрывает указатель любого другого типа, поэтому его также следует размещать после обработчиков указателей конкретного типа.

Если исключение послано, но соответствующий обработчик не найден, то вызывается специальная функция ***terminate ()***, и тем самым завершается программа.

Оператор, формирующий исключение, может иметь две формы:

throw выражение;

throw;

В первом случае исключение формируется как статический объект, значение и тип которого определяются выражением.

Копия этого объекта передается за блок контроля и существует пока исключение не будет полностью обработано.

Во втором случае – оператор ***throw*** используется только в обработчике исключений, в том случае, когда существует вложение блоков контроля за

исключениями. Цель этого оператора – ретрансляция исключения во внешний блок контроля. Если обработчик не настроен на обработку исключения, он может оператором ***throw*** направить его дальше.

Рассмотрим пример применения механизма исключений.

```
#include <fstream>
using namespace std;
class Hello {// Класс, информирующий о создании и уничтожении объектов
public:
Hello () {cout << "Hello!" << endl;}
~Hello () {cout << "Bye!" << endl;}
};
void f1() {
ifstream ifs ("NAME"); // Открываем файл для чтения
if (!ifs) {cout << "Генерируем исключение" << endl;
throw "Ошибка при открытии файла.";
}
void f2() {
Hello H; // Создается локальный объект
f1();  // Вызывается функцию, генерирующую исключение
}
int main () {
try {
cout << "Входим в try-блок" << endl;
f2();
cout << "Выходим из try-блока" <<endl;
}
catch (int i) { cout << "Вызван обработчик int, исключение - " << i << endl;
return -1;
}
catch (const char * p) {cout <<"Вызван обработчик const char*, исключение - "
<< p << endl;
return -1;
}
catch(...) {cout << "Вызван обработчик всех исключений" << endl;
return -1;
}
return 0; // Все обошлось благополучно
}
```

Результаты выполнения программы, если файл не удалось открыть:

```
Входим в try-блок
Hello!
Генерируем исключение
```

Bye!

Вызван обработчик const char*, исключение - Ошибка при открытии файла.

Обратите внимание, что после порождения исключения был вызван деструктор локального объекта, хотя управление из функции ***f1*** было передано обработчику, находящемуся в функции ***main***. Сообщение «***Выходим из try-блока***» не было выведено.

Таким образом, механизм исключений позволяет корректно уничтожать объекты при возникновении ошибочных ситуаций.

Как уже упоминалось, исключение может быть, как стандартного, так и определенного пользователем типа.

При этом нет необходимости определять этот тип глобально — достаточно, чтобы он был известен в точке порождения исключения и в точке его обработки.

Класс для представления исключения можно описать внутри класса, при работе с которым оно может возникать.

Конструктор копирования этого класса должен быть объявлен как ***public***, поскольку иначе будет невозможно создать копию объекта при генерации исключения.

В качестве примера рассмотрим применения МОИ при определении наибольшего общего делителя (НОД) двух целых чисел (x, y).

Классический алгоритм Евклида:

- если x=y, то ответ: НОД =x;
- x<y, то y заменяется на y-x;
- x>y, то x заменяется на x-y.

Алгоритм применим при условиях:

- оба числа неотрицательны;
- оба числа отличны от нуля.

В программе будет использован класс ***string*** стандартной библиотеки, описание которого находится в заголовочном файле ***<string>***. Программа:

```
#include <iostream>
#include <string>
using namespace std;
struct data { // класс для определения типа исключения
 int n, m;
string s;
data (int x, int y, string str):       // конструктор с параметрами
n(x), m(y), s(str) { }              // и с инициализатором
};
//определение функции нахождения наибольшего общего делителя (НОД) двух
//целых чисел
int GCD (int x, int y) {
if (x==0 || y==0) throw data (x, y, "zero! ");  //нулевые параметры
if(x<0) throw data (x, y, "Negative parameter 1. "); //отрицательный первый
```

```
                                                    // параметр
if(y<0) throw data (x, y, "Negative parameter 2. ");    //отрицательный второй
                                                    // параметр
while (x != y) { //алгоритм нахождения наибольшего общего делителя
if(x>y) x= x-y;
else y= y-x;
}
return x;
}
int main () {
try {
cout<<" GCD (66, 44) = "<< GCD (66, 44) <<endl;
cout<<" GCD (0, 7) = "<< GCD (0, 7) <<endl;
cout<<" GCD (-12, 8) = "<< GCD (-12, 8) <<endl;
}
catch (data d) {
cout << d.s << "  x= " << d.n << ",  y= " << d.m;
}
return 0;
}
```

Результат:

GCD (66, 44) = 22
zero! x=0, y=7

Первый вызов выполнится без проблем. Во втором вызове первый параметр имеет нулевое значение. Соответственно будет выброшено исключение в виде объекта класса ***data,*** параметрами конструктора которого, являются значения чисел и строка ***zero!***

Обработчик, получив такой объект, выводит сначала строку, а потом значения параметров. После обработки исключения управление передается первому оператору, находящемуся непосредственно за обработчиками исключений.

Поэтому третий вызов ***GCD (-12, 8)*** не будет выполнен!

Исключения в программе определены как объекты специального класса ***data.*** Объекты класса ***data*** формируются внутри одной функции, но доступны внутри другой.

Это особое свойство исключений. Они создаются как временные статические объекты в одном блоке, но доступны в другом.

Что касается определения класса объектов исключений, то следует отметить, что оно не обязательно размещается в глобальном пространстве имен, как класс ***data***.

Следовательно, при спецификации параметра обработчика (ловушки) исключений для обозначения типа параметра должна использоваться форма:

имя_пространства_имен :: имя_класса.

Рассмотрим еще пример применения механизма исключений. Используем опять для типов исключений пользовательские классы и рассмотрим случай, когда в обработчике исключений не будет значения, а только тип и случай, когда параметр вообще отсутствует.

Ниже представлен листинг программы.

```
#include <iostream>
using namespace std;
class zero_divide { }; //класс объектов исключений
class overflow { }; // класс объектов исключений
//-------определение функции деления----------
double div (double n, double d) {
if (d= =0.0&&n= =0.0) throw 0.0;
if (d= =0) throw zero_divide ();
double b=n/d;
if (b>1e+30) throw overflow ();
return b;
}
double x=1e-20, z=1e+20, w=0.0;
void RR () {   //вызовы функции деления
try {
w=div (4, w);
z=div (z, x);
w=div (0.0, 0.0);
 } // конец контрольного блока try
catch (overflow) {
cout<<" overFlow "<<endl;
z=1e+30; x=1.0;
}
catch(zero_divide) {
cout <<" zeroDivide "<<endl;
w=1.0;
}
catch (…) {cout << " Indeterminacy "<<endl;
}
} // конец функции RR
int main () {
RR ();
RR ();
RR ();
return 0;
}
```

Результат:

zeroDivide
overflow
Indeterminacy

При первом обращении к функции ***RR ()*** функция ***div ()*** вызывается один раз. Сразу же возникает исключительная ситуация и посылается исключение типа ***zero_divide***, обработчик выводит первое сообщение.

При втором вызове ***RR ()*** функция ***div ()*** вызывается уже два раза первый вызов проходит, а при втором возникает исключительная ситуация и посылается исключение уже типа ***overflow***, и соответствующий обработчик выводит второе сообщение.

При третьем обращении к функции ***RR ()*** функция ***div ()*** вызывается три раза. Для последнего случая не был определен тип, но ситуация исключительная, поэтому обработку проводит обработчик без параметров для всех случаев исключений.

Исключения, функции и раскручивание стека

Рассмотрим, как взаимодействуют функции во время обработки исключений в языке C++.

Выше было показано, что операторы ***throw*** вовсе не обязаны находиться непосредственно в блоке ***try***, благодаря выполнению такой операции, как ***«раскручивание стека»***. Это предоставляет нам необходимую гибкость в разделении общего потока выполнения кода программы и обработки исключений.

Продемонстрируем это еще раз, вынеся генерацию исключения и вычисление квадратного корня в отдельную функцию.

```
#include <cmath>      // для sqrt ()
#include <iostream>
 // Пользовательская функция вычисления квадратного корня
double mySqrt (double a) {
// Если пользователь ввел отрицательное число, то выбрасываем
//исключение
if (a < 0.0) // выбрасывается исключение типа const char*
throw " Cannot take sqrt of negative number ";
return sqrt(a); //если не возникла исключительная ситуация, корень
                // вычисляется функция возвращает его значение
}
int main () {
std::cout << " Enter a number: ";
double a;
std::cin >> a;
try {         // ищем исключения, которые выбрасываются в блоке try, и
              // отправляем их для обработки в блок(и) catch
double d = mySqrt (a);
```

```
std::cout << "The sqrt of " << a << " is " << d << '\n';
}
catch (const char* exception) //обработка исключений типа const char*
{ std::cerr << "Error: " << exception << std::endl;
}
return 0;
}
```

Здесь поместили генерацию исключения и операцию вычисления квадратного корня в отдельную функцию ***mySqrt ()***. Затем эта функция вызывается в блоке ***try***.

Убедимся, что всё работает, как нужно:

Enter a number: -3
Error: Cannot take sqrt of negative number

Работает! Рассмотрим ход выполнения программы.

Во-первых, при генерации исключения компилятор определяет, можно ли сразу же (мгновенно) обработать это исключение (для этого нужно, чтобы исключение выбрасывалось внутри блока ***try***). Поскольку точка выполнения не находится внутри блока ***try***, то и обработать исключение немедленно не получится.

Таким образом, выполнение функции ***mySqrt ()*** приостанавливается, и компилятор определяет, может ли программный код, который вызывает ***mySqrt*** (так называемый, ***caller)***, обработать это исключение.

Если нет, то компилятор завершает выполнение ***caller-a*** и переходит на уровень выше — к ***caller-y***, который вызывает текущего ***caller-a***, чтобы проверить, сможет ли тот обработать исключение. И так последовательно до тех пор, пока не будет найден соответствующий обработчик исключения, или пока функция ***main ()*** не завершит свое выполнение без обработки исключения. Этот процесс называется ***раскручиванием стека***.

Рассмотрим детально, как это относится к нашей программе. Сначала компилятор проверяет, генерируется ли исключение внутри блока ***try***. В нашем случае — нет, поэтому стек начинает раскручиваться. При этом функция ***mySqrt ()*** завершает свою работу, и точка выполнения перемещается обратно в функцию ***main ()***.

Теперь компилятор проверяет снова, находимся ли мы внутри блока ***try***. Поскольку вызов функции ***mySqrt ()*** был выполнен из блока ***try***, то компилятор начинает искать соответствующий обработчик ***catch***. Он находит обработчик типа ***const char****, и исключение обрабатывается блоком ***catch*** внутри ***main ()***.

Другими словами, блок ***try*** ловит исключения не только внутри себя, но и внутри функций, которые вызываются в этом блоке ***try***.

Самое интересное здесь в том, что ***mySqrt ()*** выбрасывает исключение, но обработка этой проблему в ***mySqrt ()*** отсутствует. Это, по сути, делегирование ответственности за обработку исключения ***caller-y.***

В конечном счете, это позволяет отделить функционал ***mySqrt ()*** от кода обработки исключений, который можно разместить в других (менее важных) частях кода.

Когда исключение обработано, выполнение кода продолжается как обычно, начиная с конца блока ***catch*** (в котором это исключение было обработано). К тому времени, когда точка выполнения возвращается обратно в функцию ***main ()***, исключение уже было сгенерировано и обработано. Функция ***main ()*** выполняется так, как если бы этого исключения не было вообще!

Раскручивание стека является очень полезным механизмом, так как позволяет в функциях не обрабатывать исключения, если это нецелесообразно. Операция раскручивания стека выполняется до тех пор, пока не будет обнаружен соответствующий блок ***catch***! Таким образом, можно решать, где следует обрабатывать исключения.

Непойманные исключения и обработчики всех типов исключений catch-all

Непойманные исключения

Выше было рассмотрено делегирование обработку исключений ***caller-y*** (или другой функции, которая «находится выше» в стеке вызовов).

В следующем примере функция ***mySqrt ()*** выбрасывает исключение, полагая, что оно будет обработано в функции ***main ()***. Но если функция ***main()*** не содержит блока ***try,*** что произойдет?

Программа:

```
#include <iostream>
#include <cmath>
double mySqrt (double a) {
  // Если пользователь ввел отрицательное число,
    if (a < 0.0)     // то выбрасывается исключение типа const char*
        throw "Cannot take sqrt of negative number";
  return sqrt(a);
}
int main () {
    std::cout << "Enter a number: ";
    double a;
    std::cin >> a;
   // Здесь нет никакого обработчика исключений!
    std::cout << "The sqrt of " << a << " is " << mySqrt (a) << '\n';
   return 0;
}
```

Теперь предположим, что пользователь ввел **-5**, и ***mySqrt (-5)*** сгенерировало исключение. Функция ***mySqrt ()*** не обрабатывает свои исключения самостоятельно, поэтому стек начинает раскручиваться, и точка

выполнения возвращается обратно в функцию ***main ()***. Но, поскольку в ***main()*** также нет обработчика исключений, выполнение ***main ()*** и всей программы прекращается.

Когда ***main ()*** завершает свое выполнение с необработанным исключением, то операционная система обычно уведомляет нас о том, что произошла ошибка необработанного исключения. Как она это сделает — зависит от каждой операционной системы отдельно:

- либо выведет сообщение об ошибке;
- либо откроет диалоговое окно с ошибкой;
- либо просто сбой.

Это то, что нельзя допускать!

Обработчики всех типов исключений

А теперь вопрос: «Функции могут генерировать исключения любого типа данных, и, если исключение не поймано, это приведет к раскручиванию стека и потенциальному завершению выполнения всей программы. Поскольку пользователи могут вызывать функции, не зная их реализации (и, следовательно, какие исключения они могут генерировать), то, как можно это предотвратить?».

Язык C++ предоставляет нам механизм обнаружения/обработки всех типов исключений — обработчик ***catch-all***. Обработчик ***catch-all*** работает так же, как и обычный блок ***catch***, за исключением того, что вместо обработки исключений определенного типа данных, он обрабатывает исключения любого типа данных.

Вот простой пример:

```
#include <iostream>
int main (){
try {
   throw 7; // выбрасывается исключение типа int
}
catch (double a) {
   std::cout << "We caught an exception of type double: " << a << '\n';
}
catch (...) {  // обработчик catch-all
    std::cout << "We caught an exception of an undetermined type!\n";
  }
}
```

Поскольку для типа ***int*** не существует специального обработчика ***catch***, то обработчик ***catch-all*** ловит это исключение.

Следовательно, результат:

We caught an exception of an undetermined type!

Обработчик ***catch-all*** должен находиться последним в цепочке блоков ***catch.*** Это делается для того, чтобы исключения сначала могли быть

пойманы обработчиками ***catch***, адаптированными к конкретным типам данных (если они вообще существуют).

Часто блок обработчика ***catch-all*** оставляют пустым:

catch(...) {} // игнорируются любые непредвиденные исключения

Этот обработчик ловит любые непредвиденные исключения и предотвращает раскручивание стека (и, следовательно, потенциальное завершение выполнения всей программы), но здесь он не выполняет никакой обработки исключений.

Использование обработчика ***catch-all*** в функции ***main ()***

Рассмотрим следующую программу:

```
#include <iostream>
int main () {
try {
runGame();
}
catch(...) {
std::cerr << "Abnormal termination\n";//аварийное завершение
}
saveState(); // сохраняем текущее состояние игрока
return 1;
}
```

В этом случае, если функция ***runGame ()*** или любая другая из функций, которые вызываются в ***runGame ()***, выбросит исключение, которое не будет поймано функциями в стеке выше, то, в конечном итоге, оно попадет в обработчик ***catch-all***. Это предотвратит завершение выполнения функции ***main ()*** и даст нам возможность вывести сообщение с указанием ошибки на наше усмотрение, а затем сохранить состояние пользователя до выхода из программы. Это может быть полезно для обнаружения и устранения непредвиденных проблем.

Список исключений функции

В заголовке функции можно задать список исключений, которые она может прямо или косвенно порождать.

Механизм объявления функций с указанием того, будет ли функция генерировать исключения (и какие именно) или нет может быть полезен при определении необходимости помещения вызова функции в блок ***try***.

Существуют три типа спецификации исключений.

Во-первых, можно использовать пустой оператор ***throw*** для обозначения того, что функция не генерирует никакие исключения, которые выходят за её пределы:

int doSomething () throw () // не выбрасываются исключения

{/* Тело функции */}

Обратите внимание, функция ***doSomething ()*** все еще может генерировать исключения, только обрабатывать она должна их самостоятельно. Любая

функция, объявленная с использованием ***throw ()*** (как в вышеприведенном примере), должна немедленно прекратить выполнение программы, если она попытается сгенерировать исключение, которое приведет к раскручиванию стека. Другими словами, мы сообщаем, что все исключения функции ***doSomething (),*** функция ***doSomething ()*** будет обрабатывать самостоятельно.

Во-вторых, мы можем использовать оператор ***throw*** с указанием списка типов исключений, которые может генерировать эта функция:

int doSomething () throw (double, const char*)

*// могут генерироваться исключения типа double и const char**

{/* Тело функции */}

Если функция создает исключение отличное от указанных в ее спецификации исключений, то управление передается специальной функции ***unexpected ()***, которая вызывает функцию ***terminate()***, и программа завершается.

Но допускается форма, обозначающая, что функция может генерировать разные типы исключений:

int doSomething () throw(...) *// могут генерироваться любые исключения*

{/* Тело функции */}

До этого момента было рассмотрено использование исключений только в обычных функциях, которые не являются методами класса. Тем не менее, исключения одинаково полезны и в методах, включая и конструкторы и в перегрузке операций.

Исключения в конструкторах и деструкторах

Язык C++ не позволяет возвращать значение из конструктора и деструктора. Механизм исключений дает возможность сообщить об ошибке, возникшей в конструкторе или деструкторе объекта.

Для иллюстрации создадим класс ***Vector***, в котором ограничивается количество запрашиваемой памяти:

```
class Vector {
public:
class Size { }; // Класс исключения
enum {max = 3200}; // Максимальная длина вектора
Vector (int n) { //конструктор
if (n<0 || n>max) throw Size ();
p = new Type [n];
}
~ Vector () {//деструктор
delete [] p;
}
…

protected:
Type * p; //указатель на первый элемент вектора
};
```

При использовании класса ***Vector*** можно предусмотреть перехват исключений типа ***Size:***

```
…
try {
Vector v (i);
}
catch (Vector::Size) {
// Обработка ошибки размера вектора
}…
```

Обрабатывать исключения, возникающие в конструкторах, можно двумя способами.

Во-первых, можно создавать блок контроля за исключениями и набор обработчиков в том месте, где вызывается конструктор, то есть создаются объекты класса.

Во-вторых, введена специальная форма генерации и обработки исключений непосредственно в конструкторах.

Внешнее определение конструктора может выглядеть так:

```
имя_ класса::имя_класса (спецификация параметров)
try: список _инициализаторов
{операторы тела конструктора}
последовательность_ обработчиков_ исключений
```

Обработчики перехватывают исключения, возникшие при инициализации и при выполнении операторов тела конструктора.

В следующей программе рассматриваются оба способа.

Определим класс точек на плоскости с счетчиком их количества.

```
#include <iostream>
#include <string>
using namespace std;
#define print(X) cout<<#X<<"= "<<X<<endl;
class point {
double x, y;
static int N;
public:
point (double xn=0.0, double yn=0.0;);
static int& count () {
return N;
}
};
int point::N=0;
point::point (double xn=0.0, double yn=0.0)
try: x(xn), y(yn)
{ //тело конструктора
```

```
N++;
if(N==1) throw "The begin! ";
if(N>2) throw string "The end! ";
}
catch (const char*ch) {
cout<<ch<<endl;
}
int main () {
try {
print (point::count ());
point A (0.0,1.0);
print (A.count ());
point B;
print (point::count ());
point C;
print (point::count ());
point D (1.0,2.0);
print (D.count ());
}
catch (const string ch) {
cout<<ch<<endl;
}
return 0;
}
```

Результат:

```
point::count () =0
The begin
A.count () =1
point::count () =2
The end
```

Исключение типа ***const char**** послано при первом обращении к конструктору при создании первого объекта. Оно обработано в конструкторе, выводится сообщение ***"The begin!".***

Исключение типа ***string*** посылается, когда число объектов превысит **2**. Оно перехватывается в основной программе ***("The end!").***

Исключения в перегрузке операций

Рассмотрим следующую перегрузку операции индексации *[]* для простого целочисленного класса-массива:

```
int& ArrayInt :: operator [] (const int index){
   return m_data [index];
}
```

Хотя эта функция отлично работает, но это только до тех пор, пока значением переменной ***index*** является корректный индекс массива. Здесь явно не хватает механизма обработки ошибок. Добавим ***assert*** для проверки ***index***:

```
int& ArrayInt :: operator [](const int index) {
   assert (index >= 0 && index < getLength ());
   return m_data [index];
}
```

Функция ***assert ()*** — это функция сообщения об ошибке в C++ (и Си - тоже). Заголовочный файл: ***cassert,*** прототип функции:

void assert (int expression);

Функция ***assert()*** оценивает выражение, которое передается ей в качестве аргумента, через параметр ***expression***. Если аргумент-выражение равно нулю (т.е. выражение ложно), сообщение записывается на стандартное устройство вывода ошибок и вызывается функция ***abort***, работа программы прекращается. Для использования функции, надо включит следующие директивы:

```
#define NDEBUG
#include <cassert>
```

Содержание сообщения об ошибке зависит от конкретной реализации компилятора, но любое сообщение должно состоять из: выражения, которое ***assert*** оценивает, имя файла с ошибкой и номер строки, где произошла ошибка. Обычный формат сообщения об ошибке:

filename: line number: expression: assertion failed:

Теперь, если пользователь передаст недопустимый ***index***, то программа выдаст ошибку. Хотя это сообщит пользователю, что что-то пошло не так, лучшим вариантом было бы сообщить ***caller-y***, что что-то пошло не так и пусть он с этим разберется соответствующим образом (как именно — мы пропишем позднее).

К сожалению, поскольку перегрузка операторов имеет особые требования к количеству и типу параметров, которые они могут принимать и возвращать, нет никакой гибкости для передачи кодов ошибок или логических значений обратно в ***caller***. Однако, мы можем использовать исключения, которые не изменяют сигнатуру функции, например:

```
int& ArrayInt :: operator [] (const int index) {
 if (index < 0 || index >= getLength ())
      throw index;
   return m_data[index];
}
```

Теперь, если пользователь передаст недопустимый ***index***, ***operator []*** сгенерирует исключение типа ***int***.

Классы-Исключения

Одной из основных проблем использования фундаментальных типов данных (например, типа ***int***) в качестве типов исключений является то, что они, по своей сути, являются неопределенными. Еще более серьезной проблемой

является неоднозначность того, что означает исключение, когда в блоке ***try*** имеется несколько вызовов функций:

// Используем перегрузку ***operator []*** для класса ***ArrayInt***

```
try {
int *value = new int(array[index1] + array[index2]);
}
catch (int value) {
   // Какие исключения мы здесь ловим?
}
```

В этом примере, если мы поймаем исключение типа ***int***, что оно нам сообщит? Был ли передаваемый ***index*** недопустим? Может операция + вызвала целочисленное переполнение или может оператор ***new*** не сработал из-за нехватки памяти? Хотя мы можем генерировать исключения типа ***const char*,*** которые будут указывать ПРИЧИНУ сбоя, это все еще не даст нам возможности обрабатывать исключения из разных источников по-разному.

Одним из способов решения этой проблемы является использование классов-исключений. Класс-исключение — это обычный класс, который выбрасывается в качестве исключения. Создадим простой класс-исключение, который будет использоваться с нашим ***ArrayInt***:

```
#include <string>
class ArrayException {
private:
   std :: string m_error;
 public:
   ArrayException (std :: string error)
      : m_error(error)
   { }
    const char* getError () {return m_error.c_str ();}
};
```

Вот полная программа:

```
#include <iostream>
#include <string>
class ArrayException {
private:
        std::string m_error;
 public:
        ArrayException (std::string error): m_error(error)
        { }
         const char* getError () {return m_error.c_str ();}
};
class ArrayInt {
private:
```

```
int m_data [4]; // укажем значение 4 в качестве длины массива
public:
        ArrayInt () {}
        int getLength () {return 4;}
        int& operator [] (const int index) {// перегрузка
        if (index < 0 || index >= getLength ())
        throw ArrayException ("Invalid index");
         return m_data[index];
         }
};
int main () {
        ArrayInt array;
        try {
                int value = array [7];
        }
        catch (ArrayException &exception) {

        std::cerr << "An array exception occurred (" <<
                        exception.getError () << ") \n";
        }
}
```

Используя такой класс, мы можем генерировать исключение, возвращающее описание возникшей проблемы, это даст нам точно понять, что именно пошло не так. И, поскольку исключение ***ArrayException*** имеет уникальный тип, мы можем обрабатывать его соответствующим образом (не так как другие исключения).

Обратите внимание, в обработчиках исключений объекты класса-исключения принимать нужно по ссылке, а не по значению. Это предотвратит создание копии исключения компилятором, что является затратной операцией (особенно в случае, когда исключение является объектом класса), и предотвратит обрезку объектов при работе с дочерними классами-исключениями. Передачу по адресу лучше не использовать, если у вас нет на это веских причин.

В случае иерархии классов.

В зависимости от обстоятельств можно использовать либо обработчик исключений базового класса, который будет перехватывать и производные исключения, либо собственные обработчики производных классов.

Интерфейсный класс std :: exception

Многие классы и операторы из Стандартной библиотеки C++ выбрасывают классы-исключения при сбое. Например, оператор ***new*** и ***std::string*** могут выбрасывать ***std::bad_alloc*** при нехватке памяти.

Неудачное динамическое приведение типов с помощью оператора ***dynamic_cast*** выбрасывает исключение ***std :: bad_cast*** и т.д.

Все эти классы-исключения являются дочерними классу ***std::exception***. ***std::exception*** — это небольшой интерфейсный класс, который используется в качестве родительского класса для любого исключения, которое выбрасывается в Стандартной библиотеке C++.

```
#include <iostream>
#include <exception> // для std::exception
#include <string> // для этого примера
int main () {
try {
// Здесь находится код, использующий Стандартную библиотеку C++.
// Сейчас намеренно спровоцируем генерацию одного из исключений
        std::string s;
       s.resize (-1); // генерируется исключение std::bad_alloc
}
// Обработчик ловит std::exception и все дочерние ему классы-исключения
catch (std::exception &exception) {
      std::cerr << "Standard exception: " << exception.what () << '\n';
}
return 0;
}
```

Результат выполнения программы:

Standard exception: string too long

В ***std::exception*** есть виртуальный метод ***what (),*** который возвращает строку ***C-style*** с описанием исключения. Большинство дочерних классов переопределяют функцию ***what (),*** изменяя это сообщение.

Иногда нам нужно будет обрабатывать определенный тип исключений несколько иначе, нежели остальные типы исключений. В таком случае можно добавить обработчик исключений для этого конкретного типа, а все остальные исключения «перенаправить» в родительский обработчик.

Например:

```
try {
// код, использующий Стандартную библиотеку C++
}
// Обработчик ловит std ::bad_alloc и все дочерние ему классы-исключения
catch (std ::bad_alloc &exception) {
std::cerr << "You ran out of memory!" <<'\n';
}
// Обработчик ловит std ::exception и все дочерние ему классы-исключения
catch (std::exception &exception) {
std::cerr << "Standard exception: " << exception.what() << '\n';}
```

В этом примере исключения типа ***std::bad_alloc*** перехватываются и обрабатываются первым обработчиком. Исключения типа ***std::exception*** и всех других дочерних ему классов-исключений обрабатываются вторым обработчиком. Такие иерархии наследования позволяют использовать определенные обработчики для перехвата определенного типа исключений или для перехвата одним (родительским) обработчиком всей иерархии исключений.

2.3 Задание на выполнение лабораторной работы

Провести анализ работы программного кода и обработку возможных ошибок (исключительных ситуаций) с использованием механизма исключений в соответствии с вариантом задания. Контролировать вводимую информацию. Вывести в стандартный поток результат обработки исключительных ситуаций. Предусмотреть также вывод имени функции, в которой произошла ошибка.

2.4 Порядок выполнения работы

1) Написать программу, в соответствии с вариантом задания.
2) Определить в программе возможные исключительные ситуации.
3) Определить типы исключения как стандартных типов, так и как типы пользовательских классов. Показать преимущества последних.
4) Провести отладку и тестирование программы.
5) Создать и защитить отчет.

2.5. Контрольные вопросы

1) Определение исключений. Типы исключений. Основное назначение и свойство механизма обработки исключений.
2) Реализация механизма обработки исключений в язык С++. Используемые операторы.
3) Обработчики исключений. Расположение. Параметры. Алгоритм перехвата исключения. Когда обработчик считается найденным?
4) Формы оператора ***throw***, формы оператора ***catch***, что такое ***try*** и его строение?
5) Непойманные исключения и обработчики всех типов исключений**.**
6) Исключения функции и раскручивание стека.
7) Список исключений функции.
8) Исключения в конструкторах.
9) Исключения в перегрузке операций.
10) Классы-Исключения. Достоинства и недостатки.
11) Исключения в иерархии классов. Интерфейсный класс ***std::exception***.

2.6. Варианты заданий лабораторной работы

1) Определить функцию, вычисляющую значение выражения в зависимости от вводимых параметров. На параметры накладываются ограничения, которые должны быть обнаружены функцией, функция должна выбрасывать соответствующие исключения, которые в программе должны быть обработаны. В функции не предусмотрены обработчики исключений. Функция вызывается в ***try – блоке***, расположенном в главной функции, за которым следуют

обработчики исключений. Реализовать тип исключения в виде класса. Протестировать программу и объяснить механизм обработки исключений.

2) Изменить программу лабораторной работы №9. Пусть создающийся список производных объектов иерархии классов выводится не на экран, а в текстовой файл. Обработать ошибку открытия файла. И вторая ситуация, которую надо обработать, - создаваемый список объектов должен ограничится 10-ю объектами. Протестировать программу и объяснить механизм обработки исключительных ситуаций.

3) Определить в программе класс ***Vector*** – динамический массив, в котором ограничивается количество запрашиваемой памяти. В конструкторе класса должно выбрасываться исключение, если запрашиваемый объем памяти больше максимально возможного. Обработку исключения провести в конструкторе.

4) Определить в программе класс – целочисленный массив фиксированной длины. Перегрузить в классе операцию квадратные скобки – обращение по индексу к элементу массива. В теле перегрузки должно выбрасываться исключение, если индекс не попадает в диапазон элементов массива. Исключение типа ***const char**** - строка сообщения ***"Недопустимый индекс элемента"***. Исключение обрабатывается в обработчике главной функции, где и должен находиться ***try-блок.***

5) В программе лабораторной работы №6 при работе с рекурсивной функцией нахождения суммы членов бесконечного ряды с заданной точность обработать исключительную ситуацию – количество рекурсивных вызовов больше 100. В теле функции должно выбрасываться исключение о переполнении. В обработчике этого исключения должно быть выведено сообщение о событии и точность вычисления суммы должна быть снижена.

6) Выбрать любую выполненную лабораторную работу. Проанализировать программу на предмет возможных исключительных ситуаций. Включить в программу возможные генерации исключений, а также операторы обработки этих исключений.

7) Существует ряд стандартных исключений, которые генерируются операциями или функциями C++. Все они являются производными от библиотечного класса ***exception***, описанного в заголовочном файле ***<exception>***. Например, операция ***new*** при неудачном выделении памяти генерирует исключение типа ***bad_alloc***. При обработке исключений может определить собственные исключения, производные от стандартных.

3. ЛАБОРАТОРНАЯ РАБОТА № 11

Разработка программ обработки символьной информации.

3.1. Цель лабораторной работы

Целью лабораторной работы является получение навыков программирования с использованием методов стандартного класса ***string*** для обработки символьной информации, хранящейся в текстовых файлах.

3.2. Теоретические сведения

В стандарте ***ISO /ANSI C++98*** библиотека C++ была расширена за счет добавления класса ***string***. С этого времени вместо использования символьных массивов для хранения символьных строк можно применять переменные типа ***string*** или, пользуясь терминологией C++, объекты.

Класс ***string*** проще в использовании, чем массив символов, и к тому же предлагает более естественное представление строки как типа.

Для работы с классом ***string*** в программе должен быть включен заголовочный файл ***<string>***. Класс ***string*** является частью пространства имен ***std,*** поэтому нужно либо указать в программе директиву ***using namespace std***, либо же ссылаться на класс как ***std :: string.***

Определение класса скрывает природу строки ***string*** как массива символов и позволяет трактовать ее как обычную переменную.

Во многих отношениях объект ***string*** можно использовать так же, как символьный массив.

- Объект ***string*** можно инициализировать строкой в стиле ***C-style***.
- Чтобы сохранить клавиатурный ввод в объекте ***string***, можно использовать ***cin*** и операцию ввода >>.
- Для отображения объекта ***string*** можно применять ***cout*** и операцию вывода <<.

В программе проиллюстрированы некоторые сходства и различия между объектами ***string*** и символьными массивами.

```
#include <iostream>
#include <string> // обеспечение доступа к классу string
int main () {
using namespace std;
char r1[20]; //создание пустого массива
char r2[20] = "jaguar"; // создание инициализированного массива
string str1; // создание пустой строки
string str2 = "panther"; // создание инициализированной строки
cout << "Enter a kind of feline: ";
// Введите животное из семейства кошачьих
cin >> r1;
cout << "Enter another kind of feline: ";
// Введите другое животное из семейства кошачьих
cin >> str1; // использование cin>> для ввода
cout << "Here are some felines:\n";
cout << r1 << " " << r2 << " "
<< str1 << " " << str2 << endl;  // использование cout << для вывода
cout << "The third letter in " << r2 << " is " << r2[2] << endl;
cout << "The third letter in " << str2 << " is " << str2[2] << endl;
// использование нотации массивов
return 0;
}
```

- Из этого примера следует, что во многих отношениях объект ***string*** можно использовать так же, как символьный массив.
- Объект string можно инициализировать строкой в стиле ***C***.
- Чтобы сохранить клавиатурный ввод в объекте ***string,*** можно использовать ***cin -*** стандартный входной поток и операцию ввода.
- Для отображения объекта ***string*** применяем ***cout*** и операцию вывода.
- Можно использовать нотацию массивов для доступа к индивидуальным символам, хранящимся в объекте ***string***.

Главное отличие между объектами ***string*** *и* ***символьными массивами****, продемонстрированное в листинге, заключается в том, что объект* ***string*** *объявляется как обычная переменная, а не массив:*

string str1; *// создание пустого объекта строки*
string str2 = "panther"; *// создание инициализированного объекта строки*

Проектное решение, положенное в основу класса, позволяет программе автоматически обрабатывать изменение размера строк.

Например, объявление ***str1*** создает объект ***string*** нулевой длины, но при вводе данных в объект ***str1***, программа автоматически его увеличивает:

cin >> str1; // *str1 увеличен для того, чтобы вместить ввод*.

Это делает использование объекта ***string*** более удобным и безопасным по сравнению с массивом.

C++ не содержит стандартного типа данных «строка». Вместо этого он поддерживает массивы символов, завершаемые нуль-символом. Стандартная библиотека C++ содержит функции для работы с такими массивами, унаследованные от C и описанные в заголовочном файле ***<string.h> (<cstring>).***

Они позволяют достичь высокой эффективности, но весьма неудобны и небезопасны в использовании, поскольку выход за границы массива не проверяется.

Тип данных ***string*** Стандартной библиотеки лишен этих недостатков, но может проигрывать массивам символов в эффективности.

Рассмотрим еще пример:

```
#include <cstring>
#include <string>
#include <iostream>
using namespace std;
int main (){
char c1[80], c2[80], c3[80]; // Строки с завершающим нулем
string s1, s2, s3;
// Присваивание строк
strcpy (c1, "old string one");
strcpy (c2, c1);
s1 = "new string one";
s2 = s1;
// Конкатенация строк
```

```
strcpy (c3, c1);
strcat (c3, c2);
s3 = s1 + s2;
// Сравнение строк
if (strcmp (c2, c3) < 0) cout << c2;
else cout << c3;
if (s2 < s3) cout << s2;
else cout << s3;
}
```

Как видно из примера, выполнение любых действий со строками старого стиля требует использования функций и менее наглядно.

Кроме того, необходимо проверять, достаточно ли места в строке-приемнике при копировании, то есть фактически код работы со строками старого стиля должен быть еще более длинным.

Строки типа ***string*** защищены от выхода информации за их границы, и с ними можно работать так же, как с любым встроенным типом данных, то есть с помощью операций.

Рассмотрим основные особенности и приемы работы со строками.

- В классе ***string*** предполагается, что отдельные символы имеют тип ***char*** и длина строки в объекте изменяется автоматически в соответствии с количеством символов, хранящихся в объекте.
- В классе ***string*** введен новый тип ***string::size_type***,

это беззнаковый целочисленный тип, предназначенный для обозначения номера символа в строке, количества символов.

- Тип определен для независимости объектов и методов ***string*** от конкретной реализации библиотеки.
- Тип ***size_type***

представляет собой беззнаковый целый тип, достаточный для хранения размера самого большого объекта для данной системы.

- В классе введено данное такого типа ***size_type - npos***.
- Величина ***npos*** является статическим членом класса ***string*** и представляет собой самое большое положительное число типа ***size_type***.

Конструкторы и присваивание строк

Для создания объектов в классе ***string*** определено несколько конструкторов. Ниже в упрощенном виде приведены заголовки наиболее употребительных:

1) ***string (const char * s) -*** Инициализирует объект ***string*** строкой, завершающейся нулевым байтом, которая указана в ***s.***
2) ***string (size_type n, char c) -*** Создает объект типа ***string*** *из* ***n*** элементов, каждый из которых инициализируется символом ***c.***
3) ***string (const string & str) -*** Инициализирует объект string объектом str типа string (конструктор копирования).
4) ***string () -*** Создает объект типа string нулевого размера (конструктор по умолчанию).

5) ***string (const char * s, size_type n)*** - Инициализирует объект типа ***string*** строкой, завершающейся нулевым байтом, которая указана в ***s*** и содержит ***n*** символов, даже если ***n*** превышает длину ***s***

6) ***template <class Iter> string (Iter begin, Iter end)*** - Инициализирует объект типа ***string*** значениями в диапазоне ***[begin, end),*** причем ***begin*** и ***end*** служат указателями начала и конца диапазона. Диапазон начинается с позиции ***begin*** включительно и заканчивается позицией ***end***, не включая ее.

7) ***string (const string & str, size_type pos, size_type n = npos)*** - Инициализирует объект типа ***string*** объектом ***str***, начиная с позиции ***pos*** и оканчивая концом ***str***, либо ограничиваясь ***n*** символами, в зависимости от того, какое условие будет удовлетворено раньше.

8) ***string (string && str) noexcept (C++11)*** - Инициализирует объект ***string*** объектом ***str*** типа ***string***. Объект str может быть изменен (конструктор переноса).

9) ***string (initializer_list <char> i1) (C++11)*** - Инициализирует объект типа ***string*** символами, указанными в списке инициализации ***i1***.

Программа, демонстрирующая использование различных конструкторов при создании объектов класса ***string***

```
#include <iostream>
#include <string>
using namespace std;
string one ("Lottery Winner!"); // конструктор #1
cout << one << endl;  // перегруженная <<
string two (20, '$'); // конструктор #2
cout << two << endl;
string three (one); // конструктор #3
cout << three << endl;
one += " Oops! "; // перегруженная +=
cout << one << endl;
two = "Sorry! That was ";
three [0] = ' P';
string four; // конструктор #4
four = two + three; // перегруженная +, =
cout << four << endl;
char alls [] = "All's well that ends well";
string five (alls,20); // конструктор #5
cout<< five << "!\n";
string six (alls+6, alls + 10); // конструктор #6
cout << six << ", ";
string seven (&five [6], &five [10]); // снова конструктор #6
cout << seven << "... \n";
string eight (four, 7, 16); // конструктор #7
cout << eight << " in motion!" << endl;
```

return 0;
}

В программе также используется перегруженная операция **+=** для добавления строк, перегруженная операция **=** для присваивания одной переменной типа ***string*** другой, перегруженная операция **<<** для отображения объекта ***string*** и перегруженная операция ***[ ]*** для доступа к отдельным символам в строке.

Вывод этой программы имеет следующий вид:

Lottery Winner!
$$$$$$$$$$$$$$$$$$$$
Lottery Winner!
Lottery Winner! Oops!
Sorry! That was Pottery Winner!
All's well that ends!
well, well...
That was Pottery in motion!

Начало программы иллюстрирует возможность инициализации объекта ***string*** обычной строкой в стиле ***C*** и ее вывод на экран с помощью перегруженной операции **<<**:

string one ("Lottery Winner!"); // #1
cout << one << endl; *// перегруженная* ***<<***

Следующий конструктор инициализирует объект ***two*** типа ***string*** строкой, состоящей из 20 символов ***$:***

string two (20, '$'); // #2

Конструктор копирования инициализирует объект ***three*** типа ***string*** объектом ***one*** этого же типа:

string three (one); // #3

Перегруженная операция **+=** дописывает строку ***" Oops! "*** к строке ***one:***

one += " Oops!"; *// перегруженная операция* +=

В этом примере строка в стиле языка ***C*** добавляется к объекту типа ***string.***

Однако операция **+=** имеет несколько перегрузок и с ее помощью можно добавлять как объекты ***string,*** так и отдельные символы.

Аналогично, операция = тоже является перегружаемой, что позволяет присвоить объекту типа ***string*** другой объект этого же типа, строку в стиле *C* или простое значение типа ***char****:*

Перегрузка операции ***[]***, как показано в примере класса ***string***, позволяет обращаться к отдельным символам объекта типа ***string***, используя нотацию массива: ***three [0] = ' P';***

Конструктор по умолчанию создает пустую строку, которой впоследствии может быть присвоено значение:

string four; // #4
four = two + three; *// перегруженные операции* **+** *и* **=**

Эта операция имеет несколько перегрузок, поэтому второй операнд может быть объектом типа ***string,*** строкой в стиле ***C*** или значением типа ***char.***

Пятый конструктор принимает в качестве аргументов строку в стиле ***C*** и целочисленное значение, которое указывает количество копируемых символов:

char alls [] = "All's well that ends well";

string five (alls,20); // #5

Шестой конструктор использует шаблон в качестве аргумента:

template<class Iter> string (Iter begin, Iter end);

begin и ***end*** выступают в роли указателей на начало и конец диапазона памяти. Конструктор применяет значения элементов памяти, хранящиеся между позициями, указанными аргументами ***begin*** и ***end.***

Рассмотрим следующий оператор:

string six (alls + 6, alls + 10); // #6

Поскольку имя массива является указателем, значения ***alls +6*** и ***alls + 10*** будут иметь тип ***char ****, и поэтому тип ***Iter*** заменяется типом ***char *.***

В результате объект ***six*** инициализируется строкой ***"well".***

Предположим, что нужно инициализировать объект частью другого объекта типа ***string,*** например, объекта ***five.***

Следующий код работать не будет**:**

string seven (five + 6, five + 10);

Причина в том, что имя объекта, в отличие от имени массива, не является адресом объекта. Следовательно, ***five*** — не указатель, и выражение ***five + 6*** не имеет смысла. Однако ***five [6]*** является значением типа ***char,*** поэтому выражение ***&five [6]*** — это адрес, который может использоваться в качестве аргумента конструктора:

string seven (&five [6], &five [10]); // снова #6

Седьмой конструктор копирует часть объекта типа ***string*** в созданный объект:

string eight (four, 7, 16); // #7

Этот оператор копирует 16 символов из объекта ***four*** в объект ***eight,*** начиная с седьмой позиции (восьмого символа) объекта ***four.***

Конструкторы C++11

Конструктор ***string (string && str) noexcept*** подобен конструктору копирования в том смысле, что новый объект ***string*** является копией объекта ***str***. Однако, в отличие от конструктора копии, он не гарантирует, что объект ***str*** будет трактоваться как ***const.*** Эту форму конструктора называют конструктором переноса.

Конструктор ***string (initializer_list<char> il)*** обеспечивает возможность списковой инициализации класса ***string.***

Он делает возможными объявления наподобие следующего:

string man = {'L', 'i1', 's', 'z', 't'};

Допустимые для объектов класса string операции:

Операция	***Действие***	***Операция***	***Действие***
=	*присваивание*	>	*больше*
+	*конкатенация*	>=	*больше или равно*

= =	*равенство*	[ ]	*индексация*
!=	*неравенство*	<<	*вывод*
<	*меньше*	>>	*ввод*
<=	*меньше или равно*	+=	*добавление*

Кроме операции индексации, для доступа к элементу строки определена функция ***at ()***:

string s("Вася");

cout <<s.at (1); // Будет выведен символ a

Если индекс превышает длину строки, порождается системное исключение ***out_of_range.***

Операция индексирования ***[]*** не обеспечивает проверку правильности задания номера символа.

Для работы со строками целиком этих операций достаточно, а для обработки частей строк (например, поиска подстроки, вставки в строку, удаления символов) в классе ***string*** определено множество разнообразных методов (функций). Функции класса ***string*** для удобства рассмотрения можно разбить на несколько категорий: *присваивание и добавление частей строк, преобразования строк, поиск подстрок, сравнение и получение характеристик строк.* Этот материал подробно рассмотрен в курсе «Программирование» (лекция - «Основные особенности и приемы работы со строками класса ***string***»).

3.3 Задание на выполнение лабораторной работы

Разработать алгоритм и программу обработки символьной информации, хранящейся в текстовом файле в соответствии с вариантом задания. В каждом варианте задания исходным является файл с текстом, состоящий из нескольких предложений.

3.4 Порядок выполнения работы

1) Посимвольно считать исходные данные и вывести их посимвольно в текстовой файл – протокол.
2) Выполнить над текстом заданный вариант обработки и вывести результаты обработки в рабочий текстовой файл.
3) Ввести данные построчно из рабочего файла в файл - протокол (также текстовый файл).
4) Файл – протокол должен включать:
   - заголовки каждого этапа обработки;
   - исходный текст, предназначенный для обработки;
   - результаты обработки, выводимые в рабочий файл;
   - исходный текст после обработки.
5) По завершению программы вывести на печать файлы: файл с исходными данными, рабочий файл и файл-протокол.
6) Провести отладку и тестирование программы.

3.5. Методические указания

Для ввода строки удобно использовать функцию ***getline()***:

string str;

```
cout << "Введите строку:";
getline (cin, Str);
```

Функция ***getline ()*** извлекает символы из входного потока и добавляет его к строковому объекту, пока не встретится символ-разделитель.

Функция определена в заголовке ***<string>,*** глобальна и предназначена для работы со строковыми объектами.

Синтаксис:

istream& getline (istream& is, string& str, char delim);

Параметры:

1) ***is*** - объект класса ***istream,*** из которого читаются данные,
2) ***str*** - строковый объект, входные данные сохраняются в этом объекте.
3) ***delim*** - это разделитель, при достижении которого чтение прекращается.

И вторая форма:

istream& getline (istream& is, string& str);

Второе объявление почти такое же, как и первое. Единственное отличие состоит в том, что последний не принимает никаких символов-разделителей. Эта функция считает символ новой строки или ('\ n') символом-разделителем.

Пример 1.

```
#include <iostream>
#include <string>
using namespace std;
int main () {
string str;
cout << "Please enter your name: \n";
getline (cin, str);
cout << "Hello, " << str << ", welcome!\n";
return 0;}
```

Входные данные:

Anny Agarwal

Результат:

Hello, Anny Agarwal, welcome!

Пример 2.

Можно использовать функцию ***getline ()***, чтобы разделить предложение на основе символа разделителя. Рассмотрим пример, чтобы понять, как это можно сделать.

```
#include <string>
#include <sstream>
using namespace std;
int main () {
string S, T;
getline (cin, S);
stringstream X(S);
while (getline (X, T, ' '))
```

```
cout << T << endl;
return 0;}
```

Входные данные:

```
Hello, Lucy Stern. Welcome!
```

Результат:

```
Hello,
Lucy
Stern.
Welcome!
```

3.6 Контрольные вопросы

1) Основные особенности и приемы работы со строками типа ***string***.
2) Конструкторы класса ***string***.
3) Операции над строками. Перегрузки операции присваивания.
4) Функция getline () типа ***string.***
5) Различия объектов ***string*** от символьных массивов.
6) Категории функций (методов) класса ***string.*** Присваивание и добавление частей строк.
7) Преобразования строк. Поиск подстрок.
8) Сравнение частей строк. Получение характеристик строк.
9) Изменение размеров. Ввод/вывод для класса ***string.***

3.7 Варианты заданий лабораторной работы

1) - Считать из текстового файла три предложения и вывести их в обратном порядке.

- Даны два текста (в двух разных файлах). Найти во втором тексте все “перевертыши” слов первого текста.

2) - Считать текст из файла и вывести на экран только предложения, содержащие заданное с клавиатуры слово.

- Дан список фамилий с инициалами. Определить, есть ли в списке однофамильцы (инициалы не учитывать). Вывести список однофамильцев.

3) - Считывает текст из файла и вывести на экран только строки, содержащие двузначные числа.

- Определить различные слова текста и их количество. Сформировать таблицу кодировки слов случайными целыми числами. Зашифровать заданный текст пословно. Вывести зашифрованный текст в рабочий файл. Расшифровать зашифрованный текст и вывести его в протокол.

4) - Считать английский текст из файла и вывести на экран слова, начинающиеся с гласных букв.

- Определить и вывести символы, с которых начинаются слова в порядке убывания количества таких слов.

5) - Считать текст из файла и вывести его на экран, меняя местами каждые два соседних слова.

- Считать текст из файла и вывести на экран только вопросительные предложения из этого текста.

6) - Считать текст файла и вывести на экран только предложения, не содержащие запятых.

- Определяет, сколько раз встретилось заданное слово в текстовом файле. Текст не содержит переносов слов. Максимальная длина строки в файле неизвестна.

7) - Считать текст из файла и определить, сколько в нем слов, состоящих не более чем из четырех букв.

- Определить и вывести символы, с которых начинаются слова в порядке убывания количества таких слов.

8) - Написать программу, которая считывает текст из файла и выводит на экран только цитаты, то есть предложения, заключенные в кавычки.

- Найти в тексте слова с длиною не менее 5 символов, которые встречаются в тексте более 3-х раз, и заменить их кодом (специальным символом). У каждого слова должен быть свой код. Расшифровать закодированный текст.

9) - Считать текст из файла и вывести на экран только предложения, состоящие из заданного количества слов.

- Определить сложность каждого предложения и всего текста. Сложность предложения оценить по количеству его слов и количеству знаков препинания. Каждое предложение вывести с новой строки; после него отпечатать характеристики его сложности. Сложность всего текста определить по количеству его слов и по количеству его предложений.

10) - Считать английский текст из файла и вывести на экран слова текста, начинающиеся с гласных букв и оканчивающиеся гласными буквами.

- Даны N и I - номер строки и позиция в строке. Дана подстрока для вставки ее в текст. Вставить заданную строку в текст после I -го символа N - ой строки исходного текста. Оставшийся текст вывести с новой строки.

11) - Считать текст из файла и вывести на экран только строки, не содержащие двузначные числа.

- Дана подстрока. Определить и вывести слова, в которых встречается эта подстрока, и номера позиций слов текста, начиная с которых располагается эта подстрока.

12) - Считать текст из файла и вывести на экран только предложения, начинающиеся с тире, перед которым могут следовать только пробельные символы.

- Определить, сколько слов текста имеют длину 1, 2, 3, …10 и более 10 символов. Вывести эти слова в последовательности возрастания их длины. Слова очередной длины вывести с новой строчки и общее их количество.

13) - Считать английский текст из файла и вывести его на экран, заменив каждую первую букву слов, начинающихся с гласной буквы, на прописную.

- Дан символ. Найти и удалить его из всех слов текста. Слова, в которых найден символ, вывести. Определить количество удаленных символов и количество слов, содержащих символ.

14) - Считать текст из файла и вывести его на экран, заменив цифры от 0 до 9 на слова «ноль», «один», ..., «девять», начиная каждое предложение с новой строки.

- Считать текст из файла и вывести на экран только строки, не содержащие числа.

15) - Считывает текст из файла, найти самое длинное слово и определить, сколько раз оно встретилось в тексте.

- Считывает текст из файла и вывести на экран только восклицательные предложения из этого текста.

16) - Считывает текст из файла и вывести на экран сначала вопросительные, а затем восклицательные предложения.

- Даны две подстроки. Найти в тексте слова, имеющие подстроку, совпадающую с первой заданною подстрокою и заменить ее второю заданною подстрокою. Определить количество замен и вывести слова, в которых проводились замены.

17) - Считать текст из файла и вывести его на экран, после каждого предложения добавляя, сколько раз встретилось в нем заданное с клавиатуры слово.

- Закодировать текст файла путем инвертирования его слов. Раскодировать текст и вывести в файл протокола.

18) - Считать текст из файла и вывести на экран все его предложения в обратном порядке.

- Даны два символа. Найти в тексте первый символ и заменить его вторым заданным символом. Определить количество замен и вывести слова, в которых были замены.

19) - Считать текст из файла и вывести на экран сначала предложения, начинающиеся с однобуквенных слов, а затем все остальные.

- Дан символ. Найти и удалить его из всех слов текста. Слова, в которых найден символ, вывести. Определить количество удаленных символов и количество слов, содержащих символ.

20) - Считать текст из файла и вывести на экран предложения, содержащие максимальное количество знаков пунктуации.

- Дана подстрока из 2-х символов. Определить и вывести слова, в которых встречается эта подстрока, и номера позиций слов текста, начиная с которых располагается эта подстрока.

21) - Определить и вывести:

а) количество предложений и слов каждого предложения;

б) текст каждого предложения с новой строки.

- Выделить из текста целые числа и определить их количество.

22) - Дана буква. Определить сколько слов в тексте начинается на эту букву. Вывести эти слова в столбик в порядке возрастания количества букв в слове.

- Определить, сколько слов текста имеют длину 1, 2, 3, …10 и более 10 символов. Вывести эти слова в последовательности возрастания их длины. Слова очередной длины вывести с новой строчки и общее их количество.

23) - Ввести подстроку. Подсчитать сколько раз встречается эта подстрока в тексте. Вывести эти слова в порядке возрастания количества вхождения подстроки в слово. Выводить в виде двух столбиков: слово – количество вхождений подстроки.

- Определить и вывести символы, с которых начинаются слова в порядке убывания количества таких слов.

24) - Ввести подстроку. Посчитать сколько раз встречается в тексте эта подстрока. Вывести эти слова в порядке возрастания номера символа в слове, начиная с которого входит подстрока.

- Определить различные слова текста и сколько раз встречается каждое слово. Упорядочить эти слова в порядке убывания частоты использования слов в тексте.

25) - Ввести подстроку. Найти в тексте заданную подстроку и удалить ее из всех слов текста. Оставшийся текст “сжать”. Определить количество удаленных подстрок и вывести слова, в которые они входили.

- Определить, сколько слов в тексте начинается и заканчивается на одну и туже букву. Вывести эти слова в алфавитном порядке и их количество.

26) - Определить, сколько слов в тексте начинается и заканчивается на одну и туже букву. Вывести эти слова в алфавитном порядке и их количество.

- Определить количество предложений в тексте, учитывая, что предложение заканчивается точкой, вопросительным или восклицательным знаком.

27) - Ввести две подстроки. Найти в тексте слова, имеющие подстроку, совпадающую с первой заданною подстрокою и заменить ее второю заданною подстрокою. Определить количество замен и вывести слова, в которых проводились замены.

- Променять местами первую и последнюю строки текста. Определить, есть ли в тексте пустые строки.

28) - Выделить из исходного текста части текста в круглых скобках вместе со скобками, вложенных скобок нет. Определить количество таких частей текста и в каждом из них количества: русских букв, латинских букв и цифр.

- Добавить в начало каждой строки текста её номер и пробел. Найти самую короткую строку текста и заменить её фразой "С новым годом!".

4. СПИСОК ЛИТЕРАТУРЫ

1) Надейкина Л.А. Программирование на языке высокого уровня. Часть 1. Учебное пособие. - М: МГТУ ГА, 2012, 84 с.

2) Надейкина Л.А. Программирование. Часть 2. Учебное пособие. - М: МГТУ ГА, 2017, 84 с.

3) Надейкина Л.А. Программирование. Обобщенное программирование. Учебное пособие. – Воронеж ООО «МИР», 2019, 80 с.

4) Подбельский В.В. Стандартный Си++. М.: Финансы и статистика, 2008, 688с.

5) Павловская Т.А. С/ С++. Программирование на языке высокого уровня - СПб: Питер, 2011. – 461 с.

СОДЕРЖАНИЕ