МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

федеральное государственное бюджетное образовательное учреждение
высшего образования
«Сибирский государственный университет науки и технологий имени академика М.Ф. Решетнева»

Кафедра информатики и вычислительной техники

Рекомендовано
для использования в учебном процессе
методической комиссией ИИТК
протокол №___ от «___»_____2019 г.

ПРОГРАММИРОВАНИЕ

ЛЕКЦИИ ПО ООП

для студентов направления
09.03.01 Информатика и вычислительная техника
направленности (профиля) образовательной программы
Автоматизированные системы обработки информации и управления

Форма обучения
очная

ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ

В структурном программировании не реализовано естественное желание думать о данных и действиях над ними как о едином целом. Структурное программирование отделяет данные от функций их обработки.

Объектно-ориентированное программирование предоставляет технологию управления элементами любой сложности, создавая условия для многократного использования программных компонентов, и объединяет данные с функциями их обработки.

Язык C++ поддерживает основные принципы ООП, а именно:

- Абстракция данных;
- Инкапсуляция;
- Наследование;
- Полиморфизм.

Абстракция данных означает возможность создания новых типов данных на базе существующих.

В C++ инкапсуляция (сокрытие) поддерживается созданием нестандартных (пользовательских) типов данных, называемых классами. После создания хорошо определенный класс можно использовать в качестве целого программного модуля. Внутренняя работа класса должна быть скрыта (инкапсулирована). Пользователям хорошо определенного класса не нужно знать, как работает этот класс, им нужно знать только, как его использовать.

C++ поддерживает наследование. Это значит, что можно объявить новый класс, который является расширением существующего класса. Говорят, что новый класс унаследован от существующего класса и называют его производным классом.

C++ поддерживает возможность вносить изменения в выполнение одноименных функций для разных объектов благодаря полиморфизму (многообразию форм) функций и классов.

Основным понятием ООП является понятие класса.

КЛАССЫ В C++

Класс – это тип, определяемый пользователем, содержащий именованный набор компонентов: данные, функции и операции обработки данных.

Переменные пользовательского типа называют объектами.

Данные задают свойства класса, а функции – его поведение. Каждый класс воплощает только одну сущность.

Данные класса называют *полями*. Поля однозначно определяют описываемый классом объект.

Выполняемые над объектом операции и функции называются *методами*.

Поля и методы класса называются *компонентами класса.*

Класс можно определить и так: к*ласс – это пользовательский тип данных, содержащий поля и методы.*

Синтаксис объявления класса:

class НазваниеКласса
{
    Компоненты;
};

Пример 1. Объявить класс векторов, исходящих из начала координат на плоскости.

```
class Vector
{
    double x,y;
public:
    double Mod(){return sqrt(x*x+y*y);}
};
```

x и y – поля класса Vector, а Mod() – метод.

Управление доступом компонентами класса

Доступ к отдельным компонентам класса регулируется с помощью ключевых слов (спецификаторов): public – открытая часть класса, private – закрытая часть класса. По умолчанию доступ private, то есть компоненты доступны только внутри класса.

При написании ООП программ желательно придерживаться следующего правила: сокрытие данных и общедоступность методов для их вызова из любого места программы.

Методы, расположенные в открытой части, формируют интерфейс класса. Доступ к закрытой части класса, при необходимости, может быть осуществлен через открытые методы. В этом проявляется такое свойство ООП как *инкапсуляция.*

Объекты классов

После объявления класса (что эквивалентно созданию пользовательского типа) можно объявить переменные этого типа (класса). Конкретная переменная пользовательского типа называется *объектом класса*.

Программы, разрабатываемые по методологии ООП, реализуют алгоритмы, описывающие взаимодействие между объектами.

Как объявить объекты? Также как и любую переменную стандартного типа:

ИмяТипа ИмяПеременной;

Пример 2. Объявить простую переменную (объект) класса Vector, массив из трех объектов, указатель на объект и ссылку на объект.

```
Vector A, B[3], *p, &r=A;
p=B+1;
```

Ссылка при объявлении должна быть инициализирована (по определению), значением указателя становится адрес второго элемента массива.

Какие действия могут быть произведены над объектом?

- Объекты могут быть объявлены. Например: Vector A, D[4];
- Объекты могут быть присвоены. Например: A=D[0];

- Объекты могут быть переданы как аргумент функции.
  Например: func(A);
  Оптимально параметр при этом объявлять как *ссылку на объект* (например, Vector &V). Возможный модификатор const при этом (например, const Vector &V) информирует, что объект внутри функции не меняется.
- Объекты могут быть возвращены как результат функции. Если результатом является локальный объект (объявленный внутри функции), то передается его *значение* (тип функции при этом, например, Vector); если результатом является измененный параметр ссылка на объект, то передается *ссылка на объект* (тип функции при этом, например, Vector&).
- Над объектами можно производить операции, если в классе они переопределены (см. последующие лекции).

Для доступа к компонентам объекта (если это позволяет спецификатор доступа) используется следующая конструкция:

ИмяОбъекта.ИмяКомпонента или
АдресОбъекта->ИмяКомпонента

Пример 3.

```
Vector A;
A.x=4.34;
A.y=-6.12;
double Result=A.Mod();
Vector *p=new Vector;
p->x=1.;
p->y=6.56;
cout<<p->Mod();
```

Методы класса

Функции обработки объекта класса называются методами. Любое возможное действие над объектом класса обеспечивается соответствующим методом класса.

Особенности метода как функции. Метод *всегда* вызывается для конкретного объекта. Этот объект *автоматически* передается методу. Передаваемый объект внутри метода доступен по унифицированному **адресу this.**

Посмотрите на метод Mod(): чьими x и y она оперирует? x и y принадлежат тому объекту, для которого вызывается метод. Для доступа к этому объекту, как к единому целому, внутри метода существует унифицированный адрес this.

Учитывая это, метод Mod() внутри класса можно определить так:

```
double Mod()
{
     return sqrt(this->x*this->x+this->y*this->y);
}
```

Данный текст подчеркивает, что x и y не самостоятельные переменные, а поля обрабатываемого объекта.

Объявление метода. В классе метод должен быть обязательно *объявлен*.

Определение метода. Определение метода может быть размещено либо внутри класса, либо вне класса. Этот выбор необходимо делать осознанно, поскольку это не просто форма написания, это форма *реализации* метода.

Чаще *определение метода располагается вне класса*. При этом в заголовке метода идет ссылка на класс, указывая принадлежность метода к классу.

Синтаксис:

ТипФункции НазваниеКласса::НазваниеФункции(список параметров)
{
    тело функции
}

Пример 4. Метод Mod() определить вне класса.

```
double Vector::Mod()
{
     return sqrt(x*x+y*y);
}
```

Если *метод определен внутри класса* (Пример 1), то он является inline методом. При обращении к такой функции не происходит реального вызова, а исполняется код функции, который компилятор вставляет вместо вызова: код программы сильно не увеличивается, а вызова функции фактически нет, и это сокращает время выполнения программы. Следует помнить, что тело inline метода должно быть *кратким*, не может, например, содержать цикла.

Методы класса инкапсулированы в классе, независимы от остального текста, поэтому различные классы могут использовать одинаковые имена методов и компилятор их не путает.

Константные методы – это методы, внутри которых *не меняются поля объекта*, для которого метод вызывается. Попытка их изменения приведет к ошибке компиляции.

Пример 5. Определить метод Mod() как константный.

```
double Vector::Mod()const
{
     return sqrt(x*x+y*y);
}
```

Следует помнить, что *для константных данных можно вызывать только константные методы.*

Совет: пишите const везде, где только можно, или не пишите его нигде.

Распределение текста программы по файлам

Объявление класса принято помещать в заголовочный файл с тем же именем, что и класс. Определения методов класса помещают в файл исходного кода с тем же именем, что и класс. Пользовательскую функцию main помещают в отдельном файле исходного кода.

Задача. *Демонстрирует объявление и определение класса со статическими полями:* объявить и определить класс векторов на плоскости, исходящих из начала координат. Включить в состав класса следующие поля:

- Координаты x и y конца вектора;

и методы:

- Ввод вектора с клавиатуры;
- Вывод вектора не экран монитора;
- Вычисления длины вектора.

В пользовательской функции main протестировать методы.

Проект состоит из следующих файлов:

Заголовочные файлы:

- Vector.h – объявление класса
- Rus.h – объявление функции русификации

Файлы исходного кода:

- Vector.cpp – определение класса
- Rus.cpp – определение функции русификации
- Main1.cpp – пользовательская функция.

```
//Содержимое файла Vector.h
#ifndef VECTOR_H
#define VECTOR_H
class Vector
{
     double x,y;
public:
     void In();
     void Out()const;
     double Mod()const;
};
#endif
//Содержимое файла Vector.cpp
#include<iostream>
using namespace std;
#include<math.h>
#include"vector.h"
#include"Rus.h"

void Vector::In(void)
{
     cout<<Rus("\nВведите координаты конца вектора: ");
     cin>>x>>y;
     return;
}
void Vector::Out(void) const
{
     cout<<"("<<x<<"; "<<y<<")";
     return;
}
double Vector::Mod()const
{
     return sqrt(x*x+y*y);
}
//Содержимое файла main1.cpp
#include<iostream>
```

```
using namespace std;
#include <windows.h>     //прототип функций русификации
#include"vector.h"
int main(void)
{
     SetConsoleCP(1251); //вызов функций русификации
     SetConsoleOutputCP(1251);
     Vector A, *B=new Vector;
     cout<<"\nВведите вектор A: ";
     A.In();
     cout<<"\nВведите вектор B: ";
     B->In();
     cout<<"\nВектор A: ";
     A.Out();
     cout<<"\nЕго длина= "<<A.Mod()<<endl;
     cout<<"\nВектор B: ";
     B->Out();
     cout<<"\nЕго длина= "<<B->Mod()<<endl;
     return 0;
}
```

Контрольные вопросы и задания

1. Назовите основные принципы ООП.
2. Что означает каждый принцип?
3. Что такое класс?
4. Как называются данные и функции класса?
5. Определите компоненты класса.
6. Каков синтаксис объявления класса?
7. Что означает объявление класса в программе?
8. Как регулируется доступ к отдельным компонентам класса?
9. Что такое объект класса и как его объявить?
10. Какие действия можно произвести над объектом?
11. Как, зная имя объекта, обратиться к компонентам класса?
12. Как, зная адрес объекта, обратиться к компонентам класса?
13. Что такое методы? Чем они отличаются от обычных функций?
14. Является ли любой метод функцией?
15. Является ли любая функция методом?
16. Если функция не объявлена в классе, является ли она методом класса?
17. Если функция не определена в классе, является ли она методом класса?
18. Охарактеризуйте функцию, определенную в классе.
19. Каковы правила определения метода вне класса?
20. Что такое this?
21. Какой метод называется константным?
22. Может ли быть константной обычная функция (не метод)?
23. Какие методы могут быть вызваны для константных объектов?

24. Как распределяется текст программы по файлам при ООП методологии?
25. Объявите класс Point (точка на плоскости).
26. Объявите простой объект класса Point статически и динамически.
27. Объявите массив объектов класса Point статически и динамически.
28. Пусть Metod() – метод класса Point, не возвращающий результата. Как вызвать этот метод для каждого объекта, объявленного в п.25-26?
29. Написать прототип и вызвать функцию, которой передается объект класса Point и не возвращается результат.
30. Написать прототип и вызвать функцию без параметров, из которой передается локальный объект класса Point.
31. Объявить и вызвать в классе Point inline метод.
32. Объявить и вызвать в классе Point константный метод.

КОНСТРУКТОРЫ И ДЕСТРУКТОР В КЛАССЕ

Любое возможное действие над объектом обеспечивается соответствующим методом класса.

Объявить объект можно следующими способами:

```
Vector A;          (1)
Vector B(2, -3);   (2)
Vector C(A);       (3)
```

Объекты создаются благодаря наличию в классе *конструкторов*, и уничтожаются благодаря наличию в классе *деструктора*.

Конструкторы

Конструкторы это методы, которые вызываются автоматически, если в программе объявляются объекты (см. выше):

(1) – вызывается конструктор по умолчанию;

(2) – вызывается конструктор с параметрами;

(3) – вызывается конструктор копий (копирования).

Назначение конструктора – конструктор выделяет память под поля объекта и придает им конкретные значения. Если поля класса – статические, память выделяется автоматически; если поля класса – динамические (то есть поля являются указателями), то память выделяется динамически в соответствующем конструкторе.

Конструктор должен быть объявлен в классе, а определен либо внутри, либо вне класса (как и любой другой метод).

Правила написания конструктора. Конструктор, как метод, имеет ряд особенностей:

- Конструктор имеет то же название, что и класс;
- Конструктор не использует оператор return;
- Конструктор не имеет возвращаемого результата, и тип void не пишется.

В зависимости от *способа передачи исходных данных* для объекта в классе могут быть определены *три вида конструкторов*:

- Конструктор по умолчанию;

- Конструктор с параметрами;
- Конструктор копирования.

Конструкторы в классе *перегружаются*, что является проявлением *полиморфизма*.

Форма написания конструктора. Каждый конструктор может быть написан *с инициализацией* или *без инициализации*.

Выполнение конструктора с инициализацией состоит из двух этапов: *инициализации* и *выполнения тела* конструктора. Часто поля создаваемого объекта могут получить значения или во время инициализации (что логически правильнее и эффективнее), или во время выполнения тела конструктора. Исключение составляют поля-константы, поля-ссылки, поля-объекты или при наследовании: они могут получить свои значения только во время инициализации. Синтаксис конструктора с инициализацией будет показан позже на конкретных примерах.

Конструктор без инициализации пишется как обычный метод.

Если *конструкторы в классе не объявлены и не определены*, то компилятор сам формирует *конструктор по умолчанию*, который является пустым и *конструктор копирования*, который формирует поля создаваемого объекта путем поэлементного копирования из эталонного объекта, что не всегда правильно.

Если в классе определен хотя бы один конструктор, то добавляется только конструктор копирования.

Конструктор по умолчанию.

Конструктор без параметров называется *конструктором по умолчанию*. Если данные в поля не поступают извне (через параметры), то они должны определяться каким-либо другим способом, например:

- Константами;
- Случайным образом;
- Вводом с клавиатуры и пр.

Пример 1. Добавить в класс Vector конструктор по умолчанию с *инициализацией* с использованием констант.

```
Vector():x(1),y(1){}
```

Пример 2. Добавить в класс Vector конструктор по умолчанию *без инициализации.*

```
Vector(){x=1;y=1;}
```

Конструктор с параметрами.

Данные в поля создаваемого объекта с помощью *конструктора с параметрами* поступают извне через параметры.

Пример 3. Добавить в класс Vector конструктор с параметрами *с инициализацией.*

```
Vector(double myX,double myY):x(myX),y(myY){}
```

Пример 4. Добавить в класс Vector конструктор с параметрами *без инициализации.*

```
Vector(double myX,double myY){x=myX;y=myY;}
```

Конструктор копирования.

С помощью *конструктора копирования* создается объект точно такой же, как уже существующий (эталонный). Эталонный объект передается конструктору через параметр ссылку, поскольку это оптимальный способ передачи объекта.

Пример 5. Добавить в класс Vector конструктор копирования *с инициализацией.*

```
Vector(const Vector&V):x(V.x),y(V.y){}
```

Пример 6. Добавить в класс Vector конструктор копирования *без инициализации.*

```
Vector(const Vector&V){x=V.x;y=V.y;}
```

Деструктор

Назначение деструктора – уничтожить объект. Если поля объекта статические – деструктор пустой; если поля – указатели – освобождается динамическая память, выделенная в конструкторах. Вызов деструктора осуществляется неявно, когда освобождается динамическая память, если объект создан динамически, или статический объект выходит из объявленной области видимости.

Деструктор должен быть объявлен в классе.

Деструктор определяется либо внутри, либо вне класса (как и любой другой метод).

Правила написания деструктора. Деструктор, как метод, имеет ряд особенностей:

- Деструктор имеет то же название, что и класс, перед которым стоит ~;
- Деструктор не использует оператор return;
- Деструктор не имеет возвращаемого результата, и тип void не пишется;
- Деструктор не имеет параметров.

Класс имеет один деструктор или не имеет его совсем, при этом компилятор генерирует его автоматически как пустой.

Пример 7. Добавить в класс Vector деструктор.

```
~Vector(){}
```

Как вызвать конструкторы и деструктор?

Конструкторы вызываются автоматически при создании (объявлении) объектов статически или динамически. Деструктор вызывается при удалении объектов.

Примеры:

```
Vector A;//создан объект конструктором по умолчанию
Vector B(3,5);//создан объект конструктором с параметрами
Vector C(A);//создан объект конструктором копирования
Vector  *D=new  Vector[3];//создан массив из трех объектов конструктором по умолчанию
delete []D;//освобождается память деструктором
```

Примечание. Массив объектов можно создать только конструктором по умолчанию.

Задача 1. *Демонстрирует использование конструкторов и деструктора для статических полей класса:* объявить и определить класс векторов на плоскости, исходящих из начала координат. **Добавить** в состав класса, разработанного ранее, следующие методы:

- Конструктор по умолчанию;
- Конструктор с параметрами;
- Конструктор копий.

В пользовательской функции main протестировать методы.

Проект состоит из следующих файлов:

Заголовочные файлы:

- Vector.h – объявление класса

Файлы исходного кода:

- Vector.cpp – определение класса
- Main2.cpp – пользовательская функция.

```
//Содержимое файла Vector.h (конструкторы используем с
//инициализацией)
#ifndef VECTOR_H
#define VECTOR_H
class Vector
{
      double x,y;
public:
      void In();
      void Out()const;
      double Mod()const;
      Vector():x(1),y(1){}
      Vector(double myX,double myY):x(myX),y(myY){}
      Vector(const Vector&V):x(V.x),y(V.y){}
      ~Vector(){}
};
      #endif
//Содержимое файла Vector.cpp (не изменился, т.к. конструкторы определены
//в классе)
#include<iostream>
using namespace std;
#include<math.h>
#include"vector.h"
#include <windows.h>     //прототип функций русификации

void Vector::In(void)
{
      cout<<"\nВведите координаты конца вектора: ";
      cin>>x>>y;
      return;
}
void Vector::Out(void) const
{
      cout<<"("<<x<<"; "<<y<<")"<<endl;
      return;
}
double Vector::Mod()const
{
      return sqrt(x*x+y*y);
{
//Содержимое файла main2.cpp
#include<iostream>
using namespace std;
#include <windows.h>     //прототип функций русификации
```

```
#include"vector.h"
int main(void)
{
	SetConsoleCP(1251);	//вызов функций русификации
	SetConsoleOutputCP(1251);

	Vector A;//создан объект конструктором по умолчанию
	Vector B(3,5);//создан объект конструктором с параметрами
	Vector C(A);//создан объект конструктором копий
	Vector *D=new Vector(-7,6);//создан объект динамически конструктором с
//параметрами
//вывод объектов
	cout<<"Vector A: ";
	A.Out();
	cout<<"Vector B: ";
	B.Out();
	cout<<"Vector C: ";
	C.Out();
	cout<<"Vector D: ";
	D->Out();
	return 0;
}
```

Задача 2. *Демонстрирует использование конструкторов и деструктора для динамических полей класса: о*бъявить и определить класс *именованных* векторов на плоскости, исходящих из начала координат. Включить в состав класса следующие поля:

- Координаты x и y конца вектора;
- Название вектора (динамическая строка).

и методы:

- Конструктор по умолчанию;
- Конструктор с параметрами;
- Конструктор копий;
- Деструктор;
- Вывод вектора на экран монитора.

В пользовательской функции main протестировать методы.

Проект состоит из следующих файлов:

Заголовочные файлы:

- Vector1.h – объявление класса

Файлы исходного кода:

- Vector1.cpp – определение класса
- Main3.cpp – пользовательская функция.

```
//Содержимое файла Vector1.h
#ifndef VECTOR1_H
#define VECTOR1_H
class Vector1
{
	double x,y;
	char *name;
public:
	Vector1();
	Vector1(double myX,double myY, char * myName);
	Vector1(const Vector1&V);
	~Vector1(){delete[]name;}
	void Out()const;
};
#endif
```

```
//Содержимое файла Vector1.cpp
#include<iostream>
using namespace std;
#define _CRT_SECURE_NO_WARNINGS
#include<math.h>
#include<string.h>
#include"vector1.h"
#include <windows.h>     //прототип функций русификации
Vector1::Vector1():x(1),y(1)
{
      name=new char[strlen("No name")+1];
      strcpy(name,"No name");
}
Vector1::Vector1(double myX,double myY, char * myName):x(myX),y(myY)
{
      name=new char[strlen(myName)+1];
      strcpy(name,myName);
}
Vector1::Vector1(const Vector1&V):x(V.x),y(V.y)
{
      name=new char[strlen(V.name)+strlen("{copy}")+1];
      strcpy(name,V.name);
      strcat(name,"{copy}");
}
void Vector1::Out(void)const
{
      cout<<name;
      cout<<"("<<x<<"; "<<y<<")"<<endl;
      return;
}
//Содержимое файла main3.cpp
#include<iostream>
using namespace std;
#include <windows.h>     //прототип функций русификации
#include"vector1.h"
int main(void)
{
      SetConsoleCP(1251);     //вызов функций русификации
      SetConsoleOutputCP(1251);

      Vector1 A;//создан объект конструктором по умолчанию
      Vector1 B(3,5,"Vector B");//создан объект конструктором с параметрами
      Vector1 C(A);//создан объект конструктором копий
      Vector1 *D=new Vector1(-7,6,"Vector D");//создан объект динамически
//конструктором с параметрами
//выводим объекты
      A.Out();
      B.Out();
      C.Out();
      D->Out();
      return 0;
}
```

ПЕРЕГРУЗКА ОПЕРАЦИЙ В С++

С++ позволяет переопределить действие большинства операций так, чтобы при использовании их с объектами класса они выполняли заданные действия. Это дает возможность *использовать пользовательские типы как стандартные.*

Перегрузка операций является проявлением *полиморфизма*.

Стоит отметить, что необязательно в классе перегружать каждую известную вам операцию. Программист сам решает, какие операции над объектами имеют смысл, и как они будут выполняться. Но! *Если операция в классе перегружена, она* ***всегда*** *должна выполняться над любыми объектами класса!*

Большинство операций могут быть перегружены за исключением: sizeof :: . .* # ## ?:

Правила перегрузки операций:

- Операции для встроенных типов не могут быть перегружены;
- Нельзя создать новую операцию;
- Не могут быть изменены семантика операций, количество операндов, приоритет операций, и правила ассоциативности;
- Перегружающие операцию функции не могут иметь параметры по умолчанию.

Перегрузка операций осуществляется с помощью *функций операторов*.

Синтаксис функции оператора:

```
ТипФункции operatorЗнакОперации(список параметров)
{
        Тело функции
}
```

Операции могут быть перегруженными с помощью методов класса или с помощью дружественных функций.

Перегрузка операций с помощью методов класса

Перегрузка операций с помощью методов класса может быть осуществлена, если *левый (или единственный) операнд* операции – *объект класса.*

Левый (или единственный) операнд передается методу автоматически, а правый (если он есть) – через параметры-ссылки.

Модификатор const в именах методов расставляем так, чтобы максимально поддержать семантику операций. Если вам это неважно – не ставьте const нигде.

Выполняя перегрузку операций, следует сначала проанализировать ее, ответив на вопросы:

1. Какую возможность в обработке объекта мы хотим получить?
2. Каков алгоритм выполнения операции над объектами?
3. Сколько операндов в операции и как они будут передаваться в метод?
4. Будет ли передаваться результат из метода (другими словами: можно ли применять операцию внутри выражения)?
5. Каким будет тип метода?

Замечание. В примерах перегрузки операций будем использовать в качестве операндов объекты класса Vector. Однако правым операндом может быть данное другого типа, например, double или int.

Перегрузка унарных операций

Перегрузку унарных операций покажем на примере операции ++ для объекта класса Vector. Операция может быть в двух формах – префиксной и постфиксной. Префиксная форма *более эффективная* для классов.

Анализ префиксной формы операции ++

1. Хотим получить возможность выполнять следующую операцию над объектом класса Vector A:
   ++A;
2. Объект-операнд будет увеличен на 1. Что это значит? Увеличим каждое поле на 1.
3. Единственный операнд передается методу автоматически.
4. Результат из метода передаваться будет, так как по семантике операция префиксный ++ может применяться в выражении.
5. Результат – увеличенный исходный объект, переданный по адресу. Значит, тип метода будет Vector &.

Анализ постфиксной формы операции ++

1. Хотим получить возможность выполнять следующую операцию над объектом класса Vector A:
   A++;
2. Создаем локальный объект – копию операнда. Зачем? По семантике результатом операции будет *неизмененный операнд*, который *впоследствии* увеличивается на 1. Что это значит? Увеличим каждое поле на 1.
3. Единственный операнд передается методу автоматически.
4. Результат из метода передаваться будет, так как по семантике операция постфиксный ++ может применяться в выражении.
5. Результат: *локальный объект* – копия исходного объекта. Значит, тип метода будет Vector.

Включим в состав класса Vector объявления унарных операций ++

```
const Vector & operator++();//префиксная форма
const Vector  operator++(int);//постфиксная форма
```

Добавим вне класса Vector определения унарных операций ++

```
const Vector & Vector::operator++()//префиксная форма
{
	++x;
	++y;
	return(*this);
}
const Vector  Vector::operator++(int)//постфиксная форма
//параметр-фиктивный, для отличия от префиксной формы
{
	Vector temp(*this);
	++x;
	++y;
	return temp;
}
```

Протестируем в функции main префиксную операцию ++

```
Vector A;
cout<<"Vector A: ";
A.Out();
++A;
cout<<"Vector ++A: ";
```

```
A.Out();
```

Протестируем в функции main постфиксную операцию ++

```
Vector A;
Vector B(A++);
cout<<"Vector B=A++: ";
B.Out();
cout<<"Vector A++: ";
A.Out();
```

Const на страже семантики операции

Если удалить модификатор const:

```
Vector & operator++();//префиксная форма
```

то будет возможен следующий вызов:

```
++++A;
```

а это противоречит семантике операции.

Если удалить модификатор const:

```
Vector  operator++(int);//постфиксная форма
```

то будет возможен следующий вызов:

```
Vector B(A++++);
```

а это противоречит семантике операции.

Перегрузка арифметических операций

Любая арифметическая операция имеет два операнда: *левый передается автоматически, правый – через параметр ссылку* на объект. *Результатом* операции является *локальный объект* метода, поля которого заполняются в соответствии со смыслом этой операции для класса.

Перегрузку арифметических операций покажем на примере операции сложения объектов класса Vector.

Анализ операции сложения

1. Хотим получить возможность выполнять следующую операцию над объектами класса Vector A и B:
   A+B
2. Сложение векторов означает получение вектора суммы, каждое поле которого получается сложением соответствующих полей слагаемых.
3. Два операнда-объекта: левый передается автоматически, правый – через параметр ссылку на объект.
4. Результат из метода передаваться будет, так как по семантике операция + может применяться в выражении.
5. Результат – локальный объект суммы. Значит, тип метода будет Vector.

Включим в состав класса Vector объявление операции +

```
const Vector operator+(const Vector&V)const;
```

Добавим вне класса Vector определение операции +

```
const Vector Vector::operator+(const Vector&V)const
{
      Vector Sum;
      Sum.x=x+V.x;
      Sum.y=y+V.y;
      return Sum;
}
```

Протестируем в функции main операцию + для двух объектов

```
Vector A(2,3), B(4.5,7);
Vector C=A+B;
cout<<"Vector A: ";
A.Out();
cout<<"Vector B: ";
B.Out();
cout<<"Vector A+B: ";
C.Out();
```

Const на страже семантики операции

Если убрать впередистоящий модификатор const:

```
Vector operator+(const Vector&V)const;
```

то будет возможен следующий вызов:

```
A+B=C;
```

а это противоречит семантике операции.

Если убрать завершающий модификатор const:

```
const Vector operator+(const Vector&V);
```

то будет невозможен следующий вызов:

```
Vector C=A+B+A;
```

а это противоречит семантике операции.

Перегрузка операции присваивания

Операция присваивания = имеет два операнда: *левый передается автоматически, правый – через параметр ссылку* на объект. *Результатом* операции является левый измененный операнд.

Анализ операции присваивания

1. Хотим получить возможность выполнять следующую операцию над объектами класса Vector A и B:
   A=B
2. Присваивание векторов означает, что полям левого вектора будет присвоены значения полей правого вектора.
3. Два операнда-объекта: левый передается автоматически, правый – через параметр ссылку на объект.
4. Результат из метода передаваться будет, так как по семантике операция = может применяться в выражении.
5. Результат – измененный левый операнд. Значит, результат можно передать по ссылке и тип метода будет Vector&.

Включим в состав класса Vector объявление операции =

```
const Vector& operator=(const Vector&V);
```

Добавим вне класса Vector определение операции =

```
const Vector& Vector::operator=(const Vector&V)
{
	if(this==&V)
		return *this;
	x=V.x;
	y=V.y;
	return *this;
}
```

Протестируем в функции main операцию = для двух объектов

```
Vector A(2,3), B(4.5,7);
cout<<"Vector A: ";
A.Out();
cout<<"Vector B: ";
```

```
B.Out();
A=B;
cout<<"Vector A=B: ";
A.Out();
```

Защита семантики операции

Можно ли не возвращать результат?

```
void operator=(const Vector&V);
```

Да, но тогда будет невозможно написать цепочку операций:

```
A=B=C;
```

а это противоречит семантике операции.

Перегрузка операций с присваиванием

Известно несколько операций с присваиванием: += -= *= /= %= .

Необходимо различать логику выполнения операции с присваиванием, например, += от похожей операции +. При выполнении сложения в результате получается объект суммы, операнды же остаются неизменными. При выполнении операции += левый операнд будет увеличен на значение правого операнда.

Операции с присваиванием имеют два операнда: *левый передается автоматически, правый – через параметр ссылку* на объект. *Результатом* операции является левый измененный операнд.

Перегрузку операций покажем на примере операции += для двух объектов класса Vector.

Анализ операции с присваиванием

1. Хотим получить возможность выполнять следующую операцию над объектами класса Vector A и B:
   A+=B
2. Операция означает, что поля левого вектора будет увеличены на значение полей правого вектора.
3. Два операнда-объекта: левый передается автоматически, правый – через параметр ссылку на объект.
4. Результат из метода передаваться будет, так как по семантике операция += может применяться в выражении (хотя это делают редко).
5. Результат – измененный левый операнд. Значит, результат можно передать по ссылке и тип метода будет Vector&.

Включим в состав класса Vector объявление операции +=

```
const Vector& operator+=(const Vector&V);
```

Добавим вне класса Vector определение операции +=

```
const Vector& Vector::operator+=(const Vector&V)
{
	x+=V.x;
	y+=V.y;
	return *this;
}
```

Протестируем в функции main операцию += для двух объектов

```
Vector A(2,3), B(4.5,7);
cout<<"Vector A: ";
A.Out();
cout<<"Vector B: ";
B.Out();
```

```
A+=B;
cout<<"Vector A+=B: ";
A.Out();
```

Защита семантики операции

Можно ли не возвращать результат?

```
void operator+=(const Vector&V);
```

Да, но тогда будет невозможен следующий вызов:

```
A=(B+=C)+D;
```

а это противоречит семантике операции.

Перегрузка операций отношения

Известно несколько операций отношения: >, >=, <, <=, ==, != .

Операции отношения имеют два операнда: *левый передается автоматически, правый – через параметр ссылку* на объект. *Результатом* операции является величина типа bool с возможными значениями true, false.

Перегрузку операций покажем на примере операции > для двух объектов класса Vector.

Анализ операции отношения

1. Хотим получить возможность выполнять следующую операцию над объектами класса Vector A и B:
   A>B
2. Как можно сравнить два вектора? Будем сравнивать по длине.
3. Два операнда-объекта: левый передается автоматически, правый – через параметр ссылку на объект.
4. Результат из метода передаваться будет, так как по семантике операция > может применяться в выражении.
5. Значение результата – true или false. Значит, тип метода будет bool.

Включим в состав класса Vector объявление операции >

```
bool operator>(const Vector&V);
```

Добавим вне класса Vector определение операции >

```
bool Vector::operator>(const Vector&V)
{
	double L1,L2;
	L1=this->Mod();
	L2=V.Mod();
	if(L1>L2)
		return true;
	else
		return false;
}
```

Протестируем в функции main операцию > для двух объектов

```
cout<<Rus("тестирование операции >")<<endl;
Vector A(2,3), B(4.5,7);
cout<<"Vector A: ";
A.Out();
cout<<"Vector B: ";
B.Out();
if(B>A)
	cout<<"Vector B>A ";
else
	cout<<"No";
```

Перегрузка операций с помощью дружественных функций

Если левый операнд операции *не является объектом класса*, то перегрузка может быть осуществлена с помощью дружественных функций.

Дружественные функции *не являются методами* и применяются для доступа к скрытым полям класса.

Дружественные функции не описывают поведение объекта, но концептуально входят в его интерфейс, например, ввод и вывод объекта.

Не следует увлекаться вводом в класс дружественных функций, так как они нарушают принцип *инкапсуляции*.

Особенности дружественных функций.

- Объявляется внутри класса со спецификатором friend в начале;
- Все операнды передаются через параметры ссылки;
- Внутри функции нет адреса this;
- Внутри функции есть доступ к закрытым полям класса;
- На функцию не распространяются спецификаторы доступа;
- Если функция определяется вне класса, то без ссылки на класс и без спецификатора friend.

Перегрузка операций ввода-вывода

Перегрузка операций ввода-вывода должна осуществляться дружественными функциями, так как левый операнд – не объект класса Vector.

Операции >> и << имеют по два операнда: левый – объект класса istream (ostream), правый – объект класса Vector. Оба объекта передадим через параметры ссылки.

Анализ операции >>

1. Хотим получить возможность выполнять следующую операцию над объектом класса Vector A:
   cin>>A или
   fin>>A
2. Ввод объекта означает ввод полей объекта из указанного потока.
3. Два операнда-объекта передаются через параметры ссылки на объекты. Объекты разных классов.
4. Результат из метода передаваться будет, так как по семантике операция >> может применяться в выражении.
5. Значение результата – ссылка на объект класса istream.

Включим в состав класса Vector объявление операции >>

```
friend istream& operator>>(istream& stream,Vector&V);
```

Добавим вне класса Vector определение операции >>

```
istream& operator>>(istream& stream,Vector&V)
{
	cout<<Rus("\nВведите координаты конца вектора: ");
	stream>>V.x>>V.y;
	return stream;
}
```

Анализ операции <<

1. Хотим получить возможность выполнять следующую операцию над объектом класса Vector A:
   cout<<A или

fout<<A

2. Вывод объекта означает вывод полей объекта в указанный поток.
3. Два операнда-объекта передаются через параметры ссылки на объекты. Объекты разных классов.
4. Результат из метода передаваться будет, так как по семантике операция << может применяться в выражении.
5. Значение результата – ссылка на объект класса ostream.

Включим в состав класса Vector объявление операции <<

```
friend ostream& operator<<(ostream& stream, const Vector&V);
```

Добавим вне класса Vector определение операции <<

```
ostream& operator<<(ostream& stream, const Vector&V)
{
	stream<<"("<<V.x<<"; "<<V.y<<")";
	stream<<endl;
	return stream;
}
```

Протестируем в функции main операции << и >> для двух объектов

```
cout<<Rus("тестирование операции >> и <<")<<endl;
Vector A, B;
cin>>A>>B;
cout<<"Vector A: ";
cout<<A;
cout<<"Vector B: ";
cout<<B;
```

Защита семантики операции

Можно ли не возвращать результат? Например:

```
void operator>>(istream& stream,Vector&V);
```

Да, но тогда будет невозможно написать цепочку операций:

```
cin>>A>>B;
```

а это противоречит семантике операции.

Замечание. Поскольку иерархия классов построена на основе *наследования*, то функции ввода-вывода могут применяться не только к стандартным потокам ввода-вывода, но и к *файловым потокам*.

Дружественные классы

Если все методы какого-либо класса должны иметь доступ к скрытым полям другого класса, то весь класс объявляется дружественным.

```
class A
{
	void f1();
	void f2();
};
class B
{
	…
	friend class A;
};
```

Методы f1() и f2() класса A имеют доступ к закрытым полям класса B.

Не следует увлекаться вводом в класс дружественных классов, так как они нарушают принцип *инкапсуляции*.

СТАТИЧЕСКИЕ КОМПОНЕНТЫ КЛАССА

До сих пор мы считали, что поля в каждом объекте являются собственностью этого объекта.

Может возникнуть ситуация*, когда требуется данное, общее для всех объектов класса*. Например, в классе Vector требуется отслеживать, сколько объектов существует в определенный момент времени. В этом нам поможет статическое поле класса.

Статическое поле объявляется в классе, создается в одном экземпляре, доступно каждому объекту и даже когда объектов еще нет.

Синтаксис объявления в классе:

static ТипПеременной ИмяПеременной;

Статическое поле может быть открытым (public), или закрытым (private). Если поле в открытом доступе, то обращаться к нему можно через имя объекта, а если объектов еще нет, то через ссылку на класс. Если поле закрыто, то обращаться к нему можно через статические функции класса. Они также объявляются внутри класса со словом static.

Особенности статических функций:

- ✓ Функциям не передается объект по умолчанию;
- ✓ Функции не имеют указателя this;
- ✓ Функции не могут быть константными;
- ✓ Статические функции не могут обращаться к нестатическим полям класса.

Вызов статических функций может быть организован через объект класса или полное имя функции со ссылкой на класс (как показано в примере).

Память под статическую переменную не выделяется при создании объектов, ведь она одна для всех объектов! Поэтому память для нее выделяется, когда *статическая переменная объявляется со ссылкой на класс и явно инициализируется вне класса и вне функций*.

Пример 1. Добавить в класс Vector закрытое статическое данное, отслеживающее количество объектов в программе.

Анализ ситуации: При создании объекта количество объектов должно увеличиваться на 1; при уничтожении объекта количество должно уменьшаться на 1. Поскольку объект создается конструкторами, *то в каждом* из них количество объектов увеличиваем на 1, а в деструкторе – уменьшаем на 1.

Для доступа к закрытому статическому полю напишем статическую функцию.

В приведенном примере используется только *один* конструктор.

Проект состоит из следующих файлов:

Заголовочные файлы:

- Vector.h – объявление класса

Файлы исходного кода:

- Main4.cpp – пользовательская функция.

```
//Содержимое файла Vector.h
#ifndef VECTOR_H
#define VECTOR_H
class Vector //в классе могут быть добавлены и другие методы
{
	double x,y;
	static int HowMany; //объявили статическую переменную
public:
	Vector(double myX,double myY):x(myX),y(myY){HowMany++;}
	~Vector(){HowMany--;}
	static int GetHowMany(){return HowMany;} //статическая функция

};
#endif
//Содержимое файла main4.cpp
#include<iostream>
using namespace std;
#include <windows.h>	//прототип функций русификации
#include"vector.h"

int Vector::HowMany=0; //сейчас статическая переменная существует
int main(void)
{
	const int Max=5;
	Vector*mas[Max];
	int i;
//вызов статической функции без объекта, со ссылкой на класс
	cout<<"How many objects? "<<Vector::GetHowMany()<<endl;
	for(i=0;i<Max;i++)
	{
		mas[i]=new Vector(i,i+2);
		cout<<"How many objects? "<<Vector::GetHowMany()<<endl;
	}
	for(i=0;i<Max;i++)
	{
		delete(mas[i]);
		cout<<"How many objects? "<<Vector::GetHowMany()<<endl;
	}
	return 0;
}
```

Замечание. В примере использован только один конструктор. Удобнее и нагляднее создавать и уничтожать динамические объекты.

Добавьте в класс другие конструкторы. Создавайте *статические* и динамические объекты разными конструкторами и уничтожайте их, каждый раз фиксируя увеличение или уменьшение общего количества объектов.

КЛАССЫ СО СТАТИЧЕСКИМИ ПОЛЯМИ: ВЫВОДЫ

Класс – пользовательский тип данных. Наполнение его таково, чтобы полностью определить его свойства (поля) и его поведение (методы).

Перечислим основные методы:

- ✓ Конструкторы
- ✓ Деструктор
- ✓ Операции над объектами
- ✓ Прочие методы.

В классе программист может не объявлять и не определять следующие методы:

- ✓ Конструктор по умолчанию
- ✓ Конструктор копий
- ✓ Деструктор
- ✓ Операцию присваивания.

Эти методы компилятор добавляет автоматически следующим образом:

- ✓ Конструктор по умолчанию как пустой метод
- ✓ Конструктор копий как поэлементное копирование полей копируемого объекта
- ✓ Деструктор как пустой метод
- ✓ Операцию присваивания как поэлементное копирование полей правого операнда.

И эти добавленные методы будут работать правильно.

КЛАССЫ С ДИНАМИЧЕСКИМИ ПОЛЯМИ

Классы с динамическими полями содержат поля-указатели. Прежде чем работать с такими полями, указатели необходимо инициализировать «правильным» адресом, выделяя память динамически.

Например, создаем класс MyString (строка). Для определения объекта-строки объявляем поле:

```
char *p;
```

Понимаем, что память под строку необходимо выделить динамически столько, сколько необходимо. Память должна выделяться при создании объекта, то есть в конструкторах.

Особенностью классов с динамическими полями является то, что все методы класса необходимо написать явно. Нельзя полагаться на методы, добавляемые компилятором, ибо работать они будут неправильно.

Например, операция присваивания добавляется автоматически как поэлементное копирование.

```
MyString A,B;
A=B;
```

Что при этом происходит?

```
A.p=B.p;
```

А это значит, что на одну и ту же область памяти (строку с адресом B.p) будут претендовать два объекта. А это неправильно, так как каждый объект должен обладать своей строкой.

Вывод: *при формировании класса с динамическими полями необходимо тщательно следить за выделенной под поле памятью. Все методы, особенно те, что могут быть добавлены компилятором, тщательно написать, даже если вам кажется, что вы не будете их использовать*.

При выполнении операций и других действий над объектами динамическая память может изменяться (увеличиваться, уменьшаться).

Например, рассмотрим следующую ситуацию с операцией присваивания:

MyString A(“Hello”), B(“Hello, my friend”);

A=B;

Понимаем, что строка А меньше В, но согласно смыслу операции содержимое строки В должно копироваться в А. Следовательно, размер строки А должен быть предварительно увеличен.

Продемонстрируем построение класса с динамическими полями на примере класса матрица.

Задача. Для двух матриц определить, перестановочно ли их умножение, то есть если A*B и B*A получатся ли две равные матрицы-результата.

Анализ решения задачи. При умножении матрицы могут быть прямоугольными, соответствующего размера. Но если мы хотим, чтобы при перестановке сомножителей получились две равные матрицы, то сомножители должны быть квадратными.

Функция main() будет выглядеть так:

```
int main(void)
{
	const int N=2;
	Matr A(N,N), B(N,N);
	cin>>A;
	cin>>B;
	if(A*B==B*A)
		cout<<endl<<”Yes\n”;
	else
		cout<<endl<<”No\n”;
	return 0;
}
```

Какие методы обеспечивают работу этой функции?

- ✓ Конструктор с параметрами
- ✓ Конструктор копий
- ✓ Деструктор
- ✓ Перегрузка операций с помощью методов (*, ==, =, операции вызова функции)
- ✓ Перегрузка операций с помощью дружественных функций (<<, >>).

ВАЖНО! Перегрузив операцию, мы должны помнить, что пользователь может применить ее к матрицам ЛЮБОГО размера. Это надо

учитывать при написании текста метода. Например, при операции = операнды-матрицы могут быть разного размера, при операции * операнды могут быть несоразмерны и так далее. *Хороший программист сделает все для того, чтобы выполнить задачу по максимуму*.

Найдите применение каждой операции в тексте программы, и увидите, что конструктор копий и операция присваивания явно не используется, но они совершенно не лишние! Конструктор копий используется при передаче результата из операции умножения (вспомните правила реализации умножения!). Операция присваивания будет использоваться при отладке умножения. Ведь нам надо видеть и анализировать результат, поэтому первоначально напишем:

```
Matr C(N,N), D(N,N);
C=A*B;
D=B*A;
cout<<endl<<C<<endl;
cout<<endl<<D<<endl;
```

Очень интересной и важной является перегрузка операции вызова функции (показывается впервые). Будем использовать ее для упрощенного доступа к элементу массива.

Работа с динамической памятью. Из ранее изученного материала (первый семестр) известно, что память под матрицу выделяется так (есть и другой вариант, наиболее сложный):

```
unsigned int n=2, m=2; //размер матрицы (строки, столбцы)
int *p=new int [n*m]; //выделение памяти
```

Тогда доступ к элементу массива осуществляется по формуле:

```
p[i*m+j]  //i и j – текущие индексы строки и столбца
```

Формула вроде и не сложная, но когда будем перегружать операцию умножения, придется не сладко! Придется оперировать размерностями и индексами трех! матриц.

С использованием перегрузки операции вызова функции к элементу матрицы будем обращаться:

```
A(i,j)
```

Почему операция вызова функции? Да просто ее вид очень напоминает доступ к элементу матрицы.

Поля класса:

Матрица однозначно определяется количеством строк, столбцов и адресом первого элемента матрицы (матрица пусть будет целой):

```
unsigned int n, m;
int *p;
```

Проанализируем алгоритмы тех методов, которые требуют работу с динамической памятью:

- ✓ Конструктор с параметрами: параметры – количество строк и столбцов; выделяя память динамически, инициализируем p. Нужно ли хотя бы обнулить элементы матрицы? Можно, но необязательно.

- ✓ Конструктор копий: присваиваем размеры существующей матрицы – создаваемой. Выделяем **свою память** создаваемой матрице. Копируем каждый элемент из существующей матрицы – в создаваемую матрицу.
- ✓ Деструктор: освобождаем динамическую память с адресом p.
- ✓ Перегрузка операции умножения: создаем матрицу-результат конструктором с параметрами (ее размер вычисляется на основе исходных матриц). Явно выделять память не нужно, это сделает конструктор. Определяем каждый элемент матрицы-результата и передаем ее *по значению*.
- ✓ Перегрузка операции присваивания: помним смысл операции – копировать элементы матрицы из правого операнда в левый. Но матрицы могут быть разного размера! Поэтому, сначала освобождаем память левого операнда; копируем размеры матрицы из правого операнда; выделяем память заново; копируем каждый элемент матрицы из правого операнда в левый.

Остальные методы не требуют работы с динамической памятью.

Напишем тексты методов, которые требуют работу с динамической памятью:

```
//конструктор с параметрами
Matr::Matr(unsigned int N, unsigned int M)
{
	n=N;
	m=M;
	p=new int[n*m];
}
//конструктор копий
Matr::Matr(Matr & B)
{
	n=B.n;
	m=B.m;
	p=new int[n*m];
	unsigned int i, j;
	for(i=0; i<n; i++)
		for(j=0; j<m; j++)
			(*this)(i, j)=B(i, j);
}
//деструктор
Matr::~Matr()
{
	delete[]p;
	p=0;
}
```

```
//перегрузка умножения
Matr Matr::operator*(Matr & B)
{
	Matr C(n, B.m);
	unsigned int K=(m<B.n)? m: B.n; //если матрицы несоразмерны
	unsigned int i, j, k;
	for(i=0; i<n; i++)
		for(j=0; j<B.m; j++)
		{
			C(i, j)=0;
			for(k=0; k<K; k++)
				C(i, j)+= (*this)(i, k)*B(k, j);
		}
	return C;
}
//перегрузка операции присваивания
Matr& Matr::operator=(Matr & B)
{
	if(this==&B)
		return *this;
	n=B.n;
	m=B.m;
	delete []p;
	p=new int[n*m];
	unsigned int i, j;
	for(i=0; i<n; i++)
		for(j=0; j<m; j++)
			(*this)(i, j)=B(i, j);
}
//перегрузка операции вызова функции (используйте по желанию!)
int & Matr::operator()(unsigned int i, unsigned int j)
{
	return p[i*m+j];
}
```

Задание. Соберите всю программу целиком, добавьте отсутствующие методы. Выполните и протестируйте программу.

НАСЛЕДОВАНИЕ

Наследование является важнейшим принципом ООП. Оно означает такое отношение между классами, когда один класс использует компоненты одного или нескольких других классов.

Наследование позволяет *абстрагировать некоторое общее или схожее* поведение различных объектов в одном базовом классе. Таким образом, создается иерархия классов: имеющиеся классы называются *базовыми*, а новые – *производными*.

Синтаксис объявления производного класса.

```
class ИмяПроизводногоКласса:<режим доступа> ИмяБазовогоКласса
{
        Компоненты производного класса
}
```

Производные классы наследуют *большинство* компонентов базового класса (есть исключения) и могут иметь собственные компоненты. В производном классе *не нужно дублировать работу с полями базового класса*, необходимо обрабатывать их с помощью методов базового класса.

Режимы доступа при наследовании

В обычном классе известны два режима доступа к компонентам класса: public и private.

Если в базовом классе режим доступа public, то он открыт для всех пользователей; если – private, то закрыт для всех, включая и производные классы.

Для базовых классов возможен режим protected – защищенный, означающий, что компоненты базового класса доступны для любого производного класса и не доступны извне иерархии.

Доступность регулируется также и режимом доступа к базовому классу при объявлении производного и определяет вид наследования:

- ✓ Public – открытое наследование;
- ✓ Protected – защищенное наследование;
- ✓ Private – закрытое наследование.

В рамках нашей лекции мы исследуем открытое наследование.

Открытое наследование

Открытое наследование означает: **«класс (производный) есть разновидность класса (базового)»**.

Пусть P – производный класс, а B – базовый. Тогда каждый объект класса P также *является* объектом класса B, *но не наоборот*!

Класс B представляет собой более общую концепцию, а класс P – более конкретную концепцию. *Везде*, где может быть *использован* объект класса B, *подойдет* и объект класса P.

Доступ к компонентам базовых классов из производного класса

Компоненты каждого класса инкапсулированы. Каждый класс разрабатывается независимо от другого класса несмотря на то, что они связаны наследованием. Компоненты классов могут иметь одинаковые имена. Как компилятор различит их? По полному имени компонента, которое включает в себя ИмяКласса::ИмяКомпонента.

Пример 1. Демонстрирует использование полных имен компонентов классов.

В примере не ищите много смысла. Не обращайте внимания на открытые поля классов. В каждом классе *имена компонентов одинаковые*, посмотрите, как их различить.

```
class B1
{
  public:
    int a;
    void F(int i){a+=i;}
};

 class B2
{
  public:
    int a;
    void F(char s){a+=int(s);}
};
class P:public B1, public B2  //производный класс
{
  public:
    int a;
    void F(void)
    {
      ::F();   //вызов внешней функции
    }
};
void F(void)  //внешняя функция
{
  ...
  return;
}
int main(void)
{
  P obj;   //объект производного класса содержит три поля, каждое по
                //имени a
  obj.B1::a=obj.B2::a=obj.a=1;
  obj.B1::F(2);   //вызов методов базовых классов
  obj.B2::F('t');
```

```
  obj.F();   //вызов метода производного класса
  ::F();   //вызов внешней функции
  return 0;
}
```

Замещение методов базового класса

Иногда в производном классе требуется *иное определение метода* базового класса.

Замещение метода производится путем объявления в *производном классе метода с таким же прототипом*, как в базовом классе, но *с другим определением.* Для объекта базового класса будет работать базовый метод, а для объекта производного класса – производный метод. Если же нам придет фантазия вызвать для производного объекта базовый метод – и это возможно, используя полное имя метода.

Замечание. Не нужно путать два схожих понятия: перегрузка и замещение. При перегрузке у функций одинаковые имена, но разные прототипы (а именно: списки параметров).

Конструкторы и деструктор в производном классе

Конструкторы, деструктор и операция присваивания *не наследуются*. Объект производного класса состоит из полей *базового класса* и из полей *производного класса*. Каждая часть объекта создается *своим конструктором*. Поэтому в конструкторе производного класса, сначала вызывается конструктор базового класса (создается базовая часть объекта), а затем работает конструктор производного класса (создается производная часть объекта).

Вызов базового конструктора производится в *части инициализации* производного конструктора (вспомните *структуру* конструктора!).

Если конструктор базового класса *имеет параметры*, то аргументы для его вызова передаются через *параметры производного конструктора*.

Синтаксис конструктора производного класса:

```
ИмяПроизвКонструктора(список параметров):
ИмяБазовогоКонструктора(список аргументов) //вызов базового конструктора
{
        Тело производного конструктора
}
```

Деструктор уничтожает объект в обратном порядке: сначала – производная часть, затем – базовая. Поскольку у деструкторов нет параметров, базовый деструктор вызывается *автоматически* из производного деструктора.

Пример 2. Демонстрирует:

- написание конструкторов и деструктора производного класса;
- замещение метода вывода базового класса;
- вызов базового метода вывода из производного класса.

Задача. Дан базовый класс. Создать производный – именованный базовый класс. Компоненты производного класса:

- ✓ Поле – динамическая строка – наименование объекта;
- ✓ Методы – три конструктора, деструктор и метод вывода объекта производного класса.

В функции main() создать объекты производного класса тремя конструкторами и вывести их на экран монитора.

```
#include<oistream>
#include<string>
using namespace std;
class Base  //объявление базового класса
{
  protected: //разрешен доступ из производного класса
   int a;
  public:
   Base(){a=0;}
   Base(int A){a=A;}
   Base(const Base & B){a=B.a;}
   ~Base(){}
   void Out(){cout<<" a="<<a<<endl;}
};
class Pr: public Base
{
  char *p;
  public:
   Pr():Base()  //вызывается конструктор по умолчанию базового класса
    {
      p=new char[80]; p[0]=0;
    }
   Pr(int A, char *str):Base(A) //вызывается конструктор с параметрами
    {                                    //базового класса
      p=new char[strlen(str)+1];
      strcpy(p, str);
    }
   Pr(const Pr & obj):Base(obj) //вызывается конструктор копий
    {                                    //базового класса
      p=new char[strlen(obj.p)+1];
      strcpy(p, obj.p);
    }
   ~Pr(){ delete[]p;}
   void Out() //замещение метода базового класса
     {
       cout<<"\nstring: "<<p<<endl; //вывод производной части объекта
       Base::Out();  //вызов базового метода для вывода базовой части объекта
```

```
            return;
        }
    };
    int main(void)
    {
      Base obj;
      obj.Out(); //метод базового класса
      Pr obj1;
      obj1.Out();  //метод производного класса
      Pr obj2(5, "Hello!");
      obj2.Out();
      Pr obj3(obj2);
      obj3.Out();
      return 0;
}
```

ВИРТУАЛЬНЫЕ МЕТОДЫ

Согласно наследованию *объекты производного класса в то же время являются и объектами базового класса*. Подразумеваем под этим то, что объекты производного класса наследуют компоненты базового класса.

Механизм виртуальных методов используется, если *метод базового класса должен по-другому выполняться в производном классе*.

Вы скажете, что, то же самое выполняется при *замещении методов*, но механизм виртуальных методов имеет более широкие возможности.

В C++ допускается *указателю базового класса* присвоить *адрес производного объекта.* Это верно, так как объект производного класса *является* объектом базового класса!

Если в дальнейшем указатель использовать для вызова виртуального метода, то метод будет вызываться в соответствии *с адресом*, помещенным в указатель:

если адрес базового объекта – вызывается базовый метод;

если адрес производного объекта – вызывается производный метод.

Пример 1. Через указатель *базового класса* создадим производный объект. Сравним работу *замещенного* и *виртуального* методов для производного объекта.

```
//необходимые подключения
class Animals  //базовый класс
{
  public:
    void Move()const
        {cout<<"\n Animals move!";}
    virtual void Speak()const
        {cout<<"\n Animals speak!";}
};
class Dog: public Animals
{
```

```
  public:
    void Move()const  //замещенный метод базового класса
        {cout<<"\n Dog move!";}
    void Speak()const  //виртуальный метод базового класса
        {cout<<"\n Dog speak!";}
    void WagTail()  //собственный метод производного класса
        {cout<<"\n Wagging Tail!";}
};
int main(void)
{
   Animals *p=new Dog;  //объект производного класса через указатель базового
   p->Move();  //работает метод базового класса
   p->Speak();  //работает метод производного класса
   p->WagTail();  //ошибка компиляции: метода в базовом классе нет!
   return 0;
}
```

Выводы: производный объект можно создать динамически, используя *указатель базового класса*. Только виртуальный метод подстраивается под *адрес*, помещенный в указатель. Остальные методы ориентируются на тип *указателя.*

Пример 2. Демонстрирует построение более абстрактной программы при использовании указателя на базовый класс для создания производных объектов.

```
//необходимые подключения
class Animals  //базовый класс
{
  public:
     virtual void Speak()const
        {cout<<"\n Animals speak!";}
};
class Dog: public Animals
{
  public:
     void Speak()const  //виртуальная функция
        {cout<<"\n Woof!";}
};
class Cat: public Animals
{
  public:
     void Speak()const  //виртуальная функция
        {cout<<"\n Meow!";}
};
int main(void)
{
  const int N=5;
```

```
  Animals *Array[N];
  int i, choice;
  for(i=0; i<N; i++)
  {
   cout<<"\n your choice:";
   cout<<"\n 1 - Dog";
   cout<<"\n 1 - Cat";
   cin>>choice;
   switch(choice)
   {
     case 1:
       Array[i]=new Dog; //создаем объект производного класса Dog
     case 2:
       Array[i]=new Cat;  // создаем объект производного класса Cat

     default:
       Array[i]=new Animals;
   }
  }
  cout<<"\n morning in my house:";
  for(i=0; i<N; i++)
  {
    Array[i]->Speak();
  }
  return 0;
}
```

Выводы: на момент компиляции неизвестно, какие объекты будут созданы, а, следовательно, какие версии виртуального метода будут использованы.

Указатель привязывается к своему объекту в момент выполнения программы. Это называется динамическим связыванием имен.

Как происходит выбор метода? Объект сам отслеживает свои виртуальные методы. Для этого создается таблица виртуальных методов, и каждый объект хранит указатель на эту таблицу. При вызове конструктора производного класса указатель корректируется на свой виртуальный метод.

Пример 3. Демонстрирует, что магия виртуальных методов проявляется, если их вызывать через адрес или ссылку (но *не через имя* базового объекта!).

Добавим к проекту Примера 2 внешние функции и новую функцию main():

```
void Value(Animals A)
{
  A.Speak();  //вызов виртуального метода через имя базового объекта
}
void Ptr(Animals *p)
```

```
{
  p->Speak();  // вызов виртуального метода через указатель на базовый объект
}
void Ref(Animals &r)
{
  r.Speak();  // вызов виртуального метода через ссылку на базовый объект
}
int main(void)
{
  Animals *p=new Dog; //создали производный объект
  Value(*p);  //вызывается метод базового класса
  Ptr(p);  //вызывается метод производного класса
  Ref(*p);  //вызывается метод производного класса
  return 0;
}
```

Выводы: магия виртуальных методов проявляется, если их вызывать через адрес или ссылку (но не через имя базового объекта!).

Деструктор базового класса

При наследовании часто объект производного класса создается динамически через указатель базового класса:

Естественно, что в определенный момент времени выделенная память должна освобождаться:

Деструктор какого класса при этом вызывается? Базового, при этом уничтожается только базовая часть объекта, и это неверно. Необходимо *деструктор объявить виртуальным в базовом классе* и он будет подстраиваться *под адрес*, а не указатель.

Деструктор *всегда* должен быть виртуальным, если:

- ✓ Класс планируется как базовый;
- ✓ В классе есть хотя бы одна виртуальная функция;
- ✓ Планируем использовать указатель на базовый класс для доступа к производным объектам.

Абстрактные классы

Базовый класс часто создается лишь для организации общего интерфейса производных классов. Объекты такого класса никогда не создаются, а виртуальные методы базового класса не имеют определения (или оно формально). В таком случае виртуальный метод базового класса объявляется без тела, как чистый виртуальный метод:

virtual ТипМетода НазваниеМетода (список параметров)=0;

Например:

virtual void Speak()const=0;

Класс хотя бы с одним чистым виртуальным методом называется *абстрактным*. Нельзя создавать объекты абстрактного класса. Абстрактный класс используют только как базовый при построении иерархии классов для обозначения общего интерфейса.

ШАБЛОНЫ КЛАССОВ

Если тип объектов не влияет на поведение класса, то можно использовать шаблон класса.

Шаблоны классов позволяют создавать параметризованные классы. Параметризованный класс создает семейство родственных классов, которые можно применять к любому типу данных. Конкретный тип будет передаваться при создании объекта.

Преимущество шаблонов состоит в том, что как только алгоритм работы с данными определен и отлажен, он может применяться к любым типам данных без переписывания кода.

Синтаксис объявления шаблона класса:

template <список параметров шаблона>
class НазваниеКласса
{
    компоненты класса
};

Каждый параметр является либо абстрактным типом с ключевым словом class, либо именем встроенного типа с идентификатором.

Методы и поля шаблонного класса определяются обычным образом с использованием абстрактного типа в нужных местах класса.

Каждый метод, определяемый вне класса, пишется со следующим синтаксисом:

template <список параметров шаблона>
ТипМетода НазваниеКласса<список аргументов шаблона>::
НазваниеМетода(список параметров метода)
{
    тело метода
}

При создании объекта на основе шаблонного класса в пользовательской функции (ориентировочно в main**()) необходимо передать конкретный тип данных**, заменяющий абстрактный тип:

НазваниеШаблонаКласса <список аргументов> ИмяОбъекта;

Применим все эти правила при построении шаблона класса.

Пример 1. Превратить ранее рассмотренный класс Vector в шаблонный.

В самом деле, вектор задается двумя полями типа double. Представим, что в другом приложении требуется вектор в целочисленной системе координат. В языке C++ известны шесть целых типов и три вещественных. Поэтому вместо конкретного типа введем абстрактный тип T и определим шаблон класса:

```
//необходимые подключения
```

```
template<class T>
class Vector
{
     T x,y; //поля абстрактного типа
public:
     void In();
     void Out()const;
     double Mod()const;
};
template<class T>  //шаблон функции
void Vector<T>::In(void) //принадлежность к шаблонному классу
{
     cout<<"\nВведите координаты конца вектора: ";
     cin>>x>>y;
     return;
}
template<class T>
void Vector<T>::Out(void) const
{
     cout<<"("<<x<<"; "<<y<<")";
     return;
}
template<class T>
double Vector<T>::Mod()const
{
     return sqrt((double)(x*x+y*y));
}
int main(void)
{
     Vector<int> A;//абстрактный тип заменяем на int
     cout<<"\nВведите вектор A: ";
     A.In();
     cout<<"\nВектор A: ";
     A.Out();
     cout<<"\nЕго длина= "<<A.Mod()<<endl;
//абстрактный тип заменяем на double; объект динамический
     Vector<double>* B=new Vector<double>;
     cout<<"\nВведите вектор B: ";
     B->In();
     cout<<"\nВектор B: ";
     B->Out();
     cout<<"\nЕго длина= "<<B->Mod()<<endl;
     return 0;
}
```

Как это работает? Компилятор, «видя» объявление

Vector<int> A;

формирует экземпляр класса (со всеми методами), в котором абстрактный тип T будет заменен на int. На объявление

Vector<double>* B=new Vector<double>;

компилятор реагирует так же. Следовательно, в нашей программе, в конечном счете, будет два обычных класса, очень схожих, но все же разных.

Особенность построения проекта с шаблонами классов

- Объявление *и определение шаблона класса* помещаем в *заголовочный файл*;
- Заголовочный файл подключаем к вызывающей функции (обычно main()) директивой #include

Специализация шаблона класса

Каждый экземпляр шаблонного класса содержит одинаковый базовый код.

Если для какого-либо типа данных существует более эффективный код, можно, либо предусмотреть для этого типа специальную реализацию отдельных методов, либо полностью специализировать (переопределить) шаблон для этого типа.

Для специализации метода требуется определить вариант его кода, указав в заголовке конкретный тип данных.

Например, пусть шаблонный класс называется My

```
template<class T>
class My
{
     Компоненты
};
```

тогда его метод имеет вид:

```
template<class T>
void My<T>::print(){…}
```

специализированный метод:

```
void My<char>::print(){…}
```

При специализации класса после описания обобщенного варианта класса, помещается полное описание специализированного класса, при этом все методы требуется определить заново:

```
class My<char>
{
     Компоненты
};
```

Передача объектов шаблона в функции

Пусть объявлен шаблонный класс My. Если обычной функции требуется передать объект этого шаблона, то мы должны указать конкретный тип вместо параметризованного типа.

Например:

```
void Func(My<char> &Obj);
```

Чтобы использовать преимущества шаблонов, функцию следует также объявить шаблонной:

```
template<class T>
void Func(My<T> &Obj);
```

Шаблоны и дружественные функции

В шаблоне класса может присутствовать дружественная нешаблонная функция:

```
template<class T>
class My
{
      …
      friend void Func(My<char>&Obj);
      …
};
```

Применять эту функцию можно не ко всем экземплярам класса My, а только к экземплярам с заменой параметризованного типа на char.

Чтобы использовать преимущества шаблонов, функцию следует также объявить шаблонной:

```
template<class T>
class My
{
      …
      template<class T>
      friend void Func(My<T>&Obj);
      …
};
```

Пример 2.

Ранее (в первом семестре) изучалась тема "Структуры" и "Динамические структуры данных". Рассматривались однонаправленные и двунаправленные линейные списки.

Рассмотрим построение двунаправленного списка через классы. Поскольку организация списка не зависит от типа помещаемых в них данных, имеет смысл построить **шаблон класса "Двунаправленный список".**

В нашем примере будут фигурировать **два шаблонных класса:**

- **Node** - шаблон класса элемента списка;
- **List** - шаблон класса двунаправленный список.

Параметр шаблона - данные списка - назовем **Data** (не T).
В шаблоне класса **Node** все компоненты сделаем public (в том числе и поля), так как это вспомогательный класс и пользователь о нем даже знать не будет, а нам это даст определенные удобства. Пользователь будет оперировать только объектами шаблона **List.**

```
#include<iostream>
#include<fstream>
using namespace std;
#include<string.h>

//---------------------------------
#include <windows.h>
```

```
//-------------------------------------

////----------------------------------------
template <class Data>
class Node          //шаблон класса элемент списка
{
public:
     Data d;
     Node<Data> *next, *prev;
     Node (Data dat){d=dat; next=0; prev=0;}
     void print(){cout<<d<<endl;}

};
//----------------------------------------
template <class Data>
class List                    //шаблон класса двунаправленный
список
{

     Node<Data> *pBeg, *pEnd; //поля класса
public:
     List(char *file);
     List() {pBeg=0; pEnd=0;}      //конструкторы класса
     List(const List<Data>&l);
     ~List();                           //деструктор

     void add(Data d);
     void print();//шаблон метода вывод списка на экран

     int save(char *file);

};
//-----------------------------------------------


template <class Data>
List<Data>::List(const List<Data>&l)//шаблон конструктора копий
{
     pBeg=pEnd=0;

     Node<Data> *pv=l.pBeg;
     while(pv)
     {
          this->add(pv->d);
          pv=pv->next;
     }
}

//-----------------------------------------

template <class Data>
List<Data>::~List()      //шаблон деструктора
{
```

```
     cout<<("\nСписок уничтожен")<<endl;
               system("pause");
}
//-----------------------------------------

template <class Data>
void List<Data>::add(Data d)       //шаблон метода добавить
элемент в конец списка
{
     Node<Data> *pv=new Node<Data>(d);
     if(pBeg==0) pBeg=pEnd=pv;
     else
     {
          pv->prev=pEnd;
          pEnd->next=pv;
          pEnd=pv;
     }
}
//-----------------------------------------
template <class Data>
List<Data>::List(char *file)  //шаблон конструктора с параметром
{
     pBeg=pEnd=0;
     ifstream fin(file, ios::binary|ios::_Nocreate);
     if(fin)
     {
     Data d;
     while(1)
     {
          fin.read((char*)&d, sizeof(d));
          if(fin.eof()) break;
          //this->add(d);
          Node<Data> *pv=new Node<Data>(d);
          if(pBeg==0) pBeg=pEnd=pv;
          else
          {
          pv->prev=pEnd;
          pEnd->next=pv;
          pEnd=pv;
          }
     }

     fin.close();
     }

}
//-------------------------------------------------
template <class Data>
int List<Data>::save(char *file) //шаблон метода сохранить
список в файл
{
     if(pBeg==0)
```

```
          return 0;
     ofstream fout(file, ios::binary|ios::trunc);
     if(!fout)
          return -1;
     Node <Data>*pv=pBeg;
     while(pv)
     {
          fout.write((char*)&(pv->d),sizeof(Data));
          pv=pv->next;
     }
     fout.close();
     return 1;
}


//-----------------------------------------------
template <class Data>
void List<Data>::print()//шаблон метода вывод списка на экран
{
     Node <Data>*pv=pBeg;
     cout<<endl<<"**************************************";
     cout<<endl<<"list: ";
     while(pv)
     {
          cout<<pv->d<<' ';
          pv=pv->next;
     }
     cout<<endl<<"**************************************";
     cout<<endl;
}

////-----------------------------------------------------
int Menu(char*menu[], int N) //функция  меню
{
     int i,n=-1;
          for( i=0; i<N; i++)
               cout<</*Rus*/(menu[i])<<endl;
          while(n<1||n>N)
          {
               cout<<("\nВыберите пункт меню ");
               cin>>n;
          }
          return n;
}
//-----------------------------------------------------

int main(void)
{
     const int N=5;

          char*menu[]={

          "1. Создать  список-копию",
```

```
        "2. Добавить в конец",
        "3. Сохранить в файл",
        "4.Вывод списка с начала",
        "5.Завершение программы\n"
};

SetConsoleCP(1251);
SetConsoleOutputCP(1251);
int i=0, n;
List <int> L("f1.ttt"); //создать список на основе файла
int A;
char File[50]="f1.ttt"; //имя файла

while(1)
{
        system("cls");
        n=Menu(menu, N);
        switch(n)
        {

        case 1:   //Создать  список-копию
                {
                List<int> L2(L);
                L2.print();
                system("pause");
                break;
                }
        case 2:   //Добавить в конец
                cout<<("\nВведите данное\n");
                cin>>A;
                L.add(A);
                L.print();
                system("pause");
                break;
        case 3:   //Сохранить в файл

                L.save(File);

                system("pause");
                break;
        case 4:   //Вывод списка с начала
                L.print();
                system("pause");
                break;
        case 5://Завершение программы
                return 0;
        default:
                cout<<("\nНеверный номер пункта меню\n");
                system("pause");
                break;
        }
}
```

```
}
```

Достоинства и недостатки шаблонов

Шаблоны – мощное и эффективное средство разработки программ, которое можно назвать параметрическим полиморфизмом.

Программа с шаблонами содержит полный код каждого порожденного класса, что увеличивает размер исполняемого файла.

С некоторыми типами шаблоны могут работать менее эффективно, как с другими, но при этом можно использовать специализацию шаблона всего класса или его отдельных методов.

Стандартная библиотека C++ содержит большой набор шаблонов для различных способов хранения и обработки данных (контейнерные классы).

Проявление полиморфизма в C++

В C++ полиморфизм имеет две формы:

- ✓ Механизм статического связывания имен: перегрузка операций и функций, шаблоны функций и классов. Статическое связывание предполагает, что определение конкретного экземпляра операции, функции или класса выполняется *на этапе компиляции.*
- ✓ Механизм динамического связывания имен: виртуальные методы. Динамическое связывание означает, что определение конкретного экземпляра виртуального метода производится *во время выполнения программы.*

Шаблоны классов называют параметрическим полиморфизмом.

ПРОСТРАНСТВО ИМЕН

Конфликты имен стали причиной раздора разработчиков программ. Конфликт имен возникает при совпадении имен разных элементов в одной и той же области видимости.

Пространства имен используются для разделения глобального пространства имен.

Основным назначением пространства имен является группировка взаимосвязанных элементов в именованной области.

Синтаксически пространства имен напоминают классы:

- ✓ Элементы, объявленные внутри пространства, принадлежат ему;
- ✓ Все элементы имеют открытый доступ;
- ✓ Пространства имен могут быть вложены внутрь других пространств;
- ✓ Функции могут быть определены внутри пространства (как в классе), но лучше не загромождать пространство имен.

Пример объявления пространства имен:

```
namespace Window
{
        class List
        {
                …
        };
        int a;
}
```

Пространство имен можно пополнить в другой части программы:

```
namespace Window
{
        const int Size=10;
        void MyFunc(int);
}
```

Хорошим стилем является выносить определение функции за пределы пространства имен, но не забывать указывать принадлежность к пространству имен, например:

```
void Window:: MyFunc(int r)
{
        …
}
```

Можно использовать вложенные пространства имен, например:

```
namespace Window
{
        const int Size=10;
        namespace Other
        {
                int OtherFunc(int);
        }
```

….
}
Тогда определение функции будет:
int Window:: Other:: OtherFunc(int h)
{
…
}
Если название пространства имен длинное и неудобное в использовании мы можем создать псевдоним:
namespace TSC=the_software_company;

Ключевое слово using

Помним, что для доступа к компоненту пространства имен необходимо указывать его принадлежность к этому пространству (в примере – пространство std – пространство стандартной библиотеки C++). Например:
int main(void)
{
std::cout<<”Hello!”<<std::endl;
}

Для переноса в текущую область видимости всех компонентов пространства имен используется директива using. Ее можно писать на локальном или глобальном уровне. После этого в программе можно обратиться к любому компоненту пространства имен (в примере – cout) без указания принадлежности к пространству. Например:
using namespace std;
int main(void)
{
cout<<”Hello!”<<endl;
}

Но в программе редко требуются все имена из указанного пространства – создается избыток имен. Лучше применять директиву только к используемым именам, например:
using std::cout;
using std::endl;
int main(void)
{
cout<<”Hello!”<<endl;
}

Обработка исключительных ситуаций (исключений)

Механизм исключений предназначен только для событий, которые происходят в результате работы самой программы и указываются явным образом. Иные аварийные ситуации (некорректное использование аппаратуры, обработка прерываний и пр.) здесь не обрабатываются.

В процессе работы программы могут возникнуть различные ошибки: неудачная попытка открытия файла, деление на 0, обращение по несуществующему адресу памяти и т.д. Используя механизм исключений можно не допустить возникновение ошибки и выполнить альтернативные действия.

Другое достоинство исключений состоит в том, что для передачи информации об ошибке в функции в вызывающую функцию не требуется применять возвращаемое значение, параметры или глобальные переменные, поэтому интерфейс функции не раздувается. Это важно для конструкторов, которые по синтаксису не могут возвратить значение.

Цель исключений состоит в проектировании отказоустойчивых программ.

Механизм обработки исключений

Место ожидаемой ошибки должно входить в контролируемый блок **try**. При появлении ошибки генерируется исключение по ключевому слову **throw** (порождать, генерировать), которое имеет (или не имеет) параметр, определяющий тип исключения.

Выполнение текущего блока прекращается и отыскивается соответствующий обработчик исключения **catch** (ловить), которому передается тип исключения. Обработчиков может быть несколько. Не перехваченные исключения по умолчанию прерывают выполнение программы.

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

Если ошибка не генерируется, после выполнения контролируемого блока, управление передается первому оператору, находящемуся за обработчиками исключений.

Синтаксис исключений

Контролируемый блок пишется так:

```
try
{
        //код, в котором генерируется исключение
}
```

Код – несколько операторов или вся функция main() или несколько вызовов других функций. Необходимо помнить, что после генерации исключения, вся оставшаяся часть блока будет пропущена.

Генерация исключения происходит по ключевому слову
throw [выражение];

Обработчики исключений пишутся за try-блоком и начинаются с ключевого слова catch, за которым в скобках следует тип исключения.

Существуют три формы обработчиков:

```
catch(тип имя) {…}  //имя параметра используется в теле обработчика
catch(тип) {…}      //важен только тип
catch(…) {…}        //обработчик перехватывает все исключения
```

Не обязательно писать обработчики всех форм, но порядок должен быть таким, поскольку поиск обработчика происходит сверху вниз. Обработчик самого общего плана должен быть последним.

Пример. Демонстрирует предотвращение аварийного останова программы из-за деления на 0, путем генерации соответствующего исключения.

```
double divide (double a, double b)
{
        try
        {
          if (!b) throw b;
          return a/b;
        }
       catch(double b)
        {
          cout<<"\nОшибка! На 0 делить нельзя!";
        }
}
int main(void)
{
   double A,B;
   do
   {
        cout<<"\nВведите делимое (0 – для окончания работы): ";
        cin>>A;
        cout<<"\nВведите делитель: ";
        cin>>B;
        cout<<divide(A,B)<<endl;
    }
  while(A!=0);
  return 0;
}
```