Федеральное государственное бюджетное образовательное учреждение
высшего образования
«КАЛИНИНГРАДСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»

И.В. Воробейкина

ПРОГРАММИРОВАНИЕ СРЕДСТВ ЗАЩИТЫ ИНФОРМАЦИИ

учебно-методическое пособие по выполнению лабораторных работ
для студентов специальности
10.05.03 Информационная безопасность автоматизированных систем

Калининград
Издательство ФГБОУ ВО «КГТУ»
2022

Рецензент
доцент кафедры информационной безопасности ФГБОУ ВО
«Калининградский государственный технический университет»
А. Г. Жестовский

Воробейкина, И. В.
Программирование средств защиты информации: учеб-методич. пособие по лабораторным работам для студентов специальности 10.05.03 Информационная безопасность автоматизированных систем / И. В. Воробейкина. - Калининград: Изд-во ФГБОУ ВО «КГТУ», 2022. – 89 с.

Учебно-методическое пособие является руководством по проведению цикла лабораторных работ по дисциплине «Программирование средств защиты информации» студентами, обучающимися по специальности 10.05.03 «Информационная безопасность автоматизированных систем». Лабораторные работы предназначены для закрепления теоретического материала.
Список лит. – 6 наименований

Учебно-методическое пособие рассмотрено и одобрено в качестве локального электронного методического материала кафедрой информационной безопасности 14 июня 2022 г., протокол № 09

Учебно-методическое пособие рекомендовано к использованию в качестве локального электронного методического материала в учебном процессе методической комиссией института цифровых технологий ФГБОУ ВО «Калининградский государственный технический университет» 28 июня 2022 г., протокол № 4

ОГЛАВЛЕНИЕ

1. ВВЕДЕНИЕ

Данное учебно-методическое пособие предназначено для студентов специальности 10.05.03 Информационная безопасность автоматизированных систем, изучающих дисциплину «Программирование средств защиты информации».

Цель лабораторного практикума по дисциплине: формирование у студентов умения составлять алгоритмы программ для средств защиты информации; умения использовать возможности языков программирования для решения задач защиты информации; тестирования, отладки и сопровождения программного обеспечения.

Лабораторный практикум содержит 14 лабораторных работ.

Лабораторные работы проводятся в лабораториях кафедры.

В результате выполнения лабораторных работ ожидается, что студенты сформируют навыки программирования криптографических шифров.

2. ЛАБОРАТОРНАЯ РАБОТА № 1. ПРОГРАММИРОВАНИЕ МЕТОДА ДИХОТОМИИ (МЕТОД ДЕЛЕНИЯ ОТРЕЗКА ПОПОЛАМ)

2.1. Общие сведения

Цель: Изучить метод дихотомии как один из численных методов, использующийся в классических алгоритмах.

Материалы, оборудование, программное обеспечение: online-компилятор C++ (необходима бесперебойная работа сети ИНТЕРНЕТ) или Visual Studio C++.

Условия допуска к выполнению: предварительно пройти инструктаж по ТБ. Показать конспект по теоретической подготовке и решить подготовительный пример.

Критерии положительной оценки: Показать выполненную работу: решенный пример и запрограммированный метод дихотомии. Ответить на вопросы преподавателя.

2.2. Теоретическое введение

В общем случае нелинейное уравнение с одним неизвестным можно записать как $f(x) = 0$, где $f(x)$ – некоторая непрерывная функция аргумента x.

Всякое число x_0, при котором $f(x_0) \equiv 0$, называется *корнем уравнения* $f(x) = 0$.

При численном подходе задача о решении нелинейных уравнений разбивается на два этапа: *локализация* (отделение) корней, т. е. нахождение таких отрезков на оси *OX*, в пределах которых содержится единственный корень, и *уточнение корней*.

Задача **уточнения корней** состоит в получении приближенного значения корня, принадлежащего отрезку $[a, b]$, с заданной точностью (погрешностью) ε. Это означает, что вычисленное значение корня $x^{\sim}$ должно отличаться от точного x_0 не более чем на величину ε: $|x_0 - x^{\sim}| \leq \varepsilon$

Процедура численного определения приближенных значений корней нелинейных уравнений состоит в выборе *начального приближения* x_0 на интервале $[a, b]$ и вычислении по некоторой формуле последующих приближений x_1, x_2 и т. д. Каждый такой шаг называется *итерацией*, а сами методы уточнения – *итерационными методами*. В результате итераций получается последовательность приближенных значений корня $x_0, x_1, \ldots, x_k, \ldots$, которая называется *итерационной последовательностью*.

Метод дихотомии (метод деления отрезка пополам)

Пусть мы нашли отрезок $[a; b]$, в котором расположено искомое значение корня $x = x^*$, т.е. $a < x^* < b$. Пусть для определенности $F(a) < 0$, $F(b) > 0$. В качестве начального приближения корня x_0 принимается середина этого от-

резка, т.е. $x_0 = (a + b)/ 2$. Далее исследуем значение функции $F(x)$ на концах отрезков $[a; x_0]$ и $[x_0; b]$. Тот из них, на концах которого $F(x)$ принимает значения разных знаков, содержит искомый корень. Поэтому его принимаем в качестве нового отрезка. Вторую половину отрезка $[a; b]$ отбрасываем. В качестве первой итерации корня принимаем середину нового отрезка и т. д.

Таким образом, после каждой итерации отрезок, на котором расположен корень, уменьшается вдвое, т. е. после n итераций он сокращается в 2^n раз. Если длина полученного отрезка становится меньше допустимой погрешности, т. е. $|b - a| < \varepsilon$, счет прекращается.

Литература:

Воробейкина, И.В. Программирование средств защиты информации: учебное пособие для студентов специальности 10.05.03 «Информационная безопасность автоматизированных систем» очной формы обучения / И.В. Воробейкина; БГАРФ ФГБОУ ВО «КГТУ». – Калининград: Изд-во БГАРФ, 2021. - 70 с.

Глава 2. Численные методы как основа алгоритмов.

2.3.Задание к лабораторной работе

1. Обоснуйте типы данных, которые вы выбрали для программирования метода дихотомии.

2. Как изменится количество итераций, если увеличить точность вычисления?

3. Начертить график выбранной вами нелинейной функции и показать на графике, как найти корень.

2.4.Методические указания и порядок выполнения работы

1. Решить пример:

Пример 1.1. Найти решение уравнения $x^3+x-1=0$ на отрезке $[0; 1]$ с точностью $\varepsilon=0,01$ методом деления отрезка пополам.

Решение.

Уравнение представим в виде $x^3=-x+1$. Корнем данного уравнения является x-координата точки пересечения графиков функций $y = x^3$ и $y = -x+1$. Искомый корень находится между точками $a = 0$ и $b = 1$. Функция $F(x) = x^3+x-1$ на концах отрезка $[0; 1]$ принимает значения разных знаков и $F(a)F(b) < 0$.

Начальное приближение: $a = 0, b = 1, x_0 = (a + b)/ 2 = 0.5$.
$F(a) = -1; F(x_0) = 0.5^3+0.5 - 1 = -0.375; F(b) = 1$.

1-е приближение: $a = 0.5, b = 1, x_1 = (a + b)/ 2 = 0.75$.

Погрешность $|b-a|=1-0.5=0.5>0.01$.
$F(a)=-0.375$; $F(x_1)=0.75^3+0.75-1=0.172$; $F(b)=1$.
Корень находится в интервале $[0.5; 0.75]$.

2-е приближение: $a=0.5$, $b=0.75$, $x_2=(a+b)/2=0.625$.
Погрешность $|b-a|=0.75-0.5=0,25>0.01$.
$F(a)=-0.375$; $F(x_2)=0.625^3+0.625-1=-0.132$; $F(b)=0.172$.
Корень находится в интервале $[0.625; 0.75]$.

3-е приближение: $a=0.625$, $b=0.75$, $x_3=(a+b)/2=0.69$
Погрешность $|b-a|=0.75-0.625=0.125>0.01$.
$F(a)=-0.132$; $F(x_3)=0.69^3+0.69-1=0.02$; $F(b)=0.172$.
Корень находится в интервале $[0.625; 0.69]$.

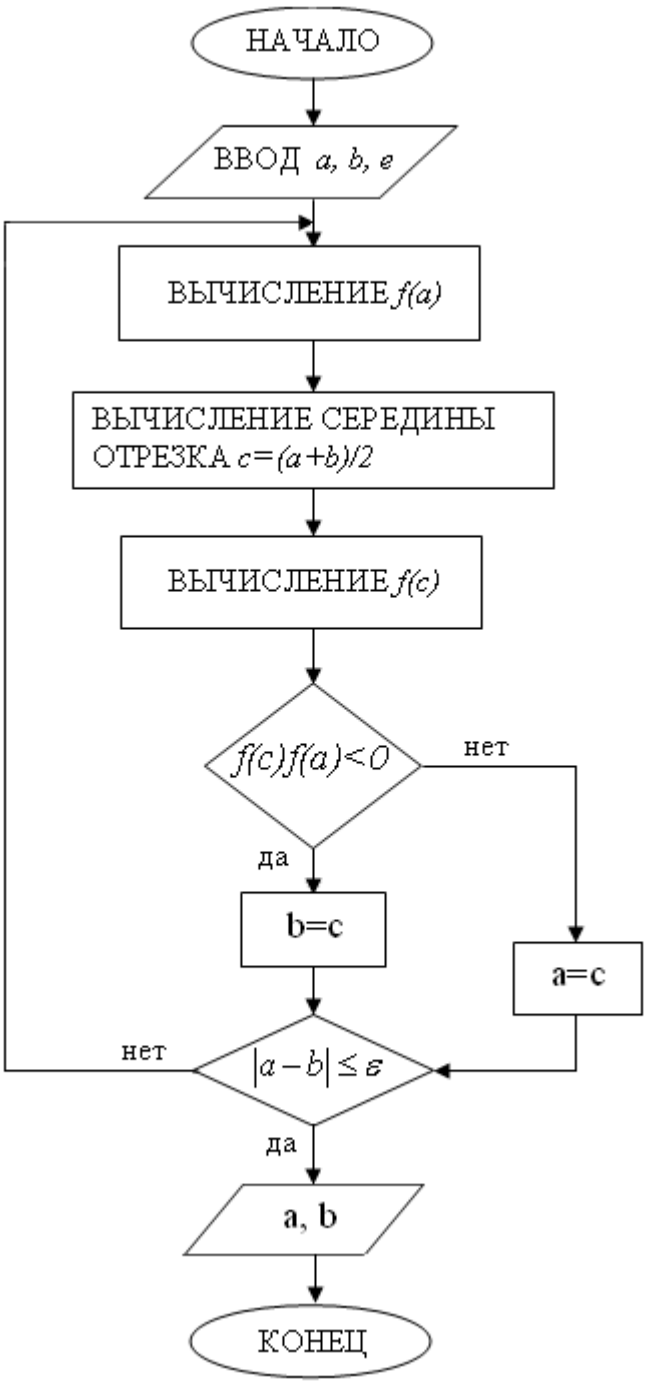
4-е приближение: $a=0.625$, $b=0.69$, $x_4=(a+b)/2=0.66$
Погрешность $|b-a|=0.69-0.625=0.065>0.01$.
$F(a)=-0.132$; $F(x_4)=0.66^3+0.66-1=-0.05$; $F(b)=0.02$.
Корень находится в интервале $[0.66; 0.69]$.

5-е приближение: $a=0.66$, $b=0.69$, $x_5=(a+b)/2=0.675$
Погрешность $|b-a|=0.69-0.66=0.03>0.01$.
$F(a)=-0.05$; $F(x_5)=0.675^3+0.675-1=-0.02$; $F(b)=0.02$.
Корень находится в интервале $[0.675; 0.69]$.

6-е приближение: $a=0.675$, $b=0.69$, $x_6=(a+b)/2=0.682$
Погрешность $|b-a|=0.69-0.675=0.015>0.01$.
$F(a)=-0.02$; $F(x_6)=0.682^3+0.682-1=-0.0008$; $F(b)=0.02$.
Корень находится в интервале $[0.682; 0.69]$.

7-е приближение: $a=0.682$, $b=0.69$, $x_7=(a+b)/2=0.686$.
Погрешность $|b-a|=0.69-0.682=0.008<0.01$.
Приближенным решением данного уравнения является $x=0.686$.

2. Используя блок-схему, написать программный код.



Литература:

Воробейкина, И.В. Программирование средств защиты информации: учебное пособие для студентов специальности 10.05.03 «Информационная безопасность автоматизированных систем» очной формы обучения / И.В. Воробейкина; БГАРФ ФГБОУ ВО «КГТУ». – Калининград: Изд-во БГАРФ, 2021. - 70 с.

Глава 2. Численные методы как основа алгоритмов.

2.5.Индивидуальное задание

Вариативность не предполагается. Освоить метод дихотомии, создать блок-схему и программный код.

Найти решение уравнения $x^3-12x-5=0$ на отрезке $[-1;\ 0]$ с точностью $e=0,01$ методом деления отрезка пополам. Создать блок-схему и программный код.

2.6.Требования к отчету и защите

Показать выполненную в тетради и на компьютере работу. Знать ответы на сформулированные выше вопросы. Защита работы проводится во время занятий. После защиты работа помещается в ЭИОС.

3. ЛАБОРАТОРНАЯ РАБОТА № 2. ПРОГРАММИРОВАНИЕ МЕТОДА НЬЮТОНА (МЕТОД КАСАТЕЛЬНЫХ)

3.1.Общие сведения

Цель: Изучить метод Ньютона, как один из численных методов, использующийся в классических алгоритмах.

Материалы, оборудование, программное обеспечение: online-компилятор C++ (необходима бесперебойная работа сети ИНТЕРНЕТ) или Visual Studio C++

Условия допуска к выполнению: предварительно пройти инструктаж по ТБ. Показать конспект по теоретической подготовке и решить подготовительный пример.

Критерии положительной оценки: Показать выполненную работу: решенный пример и запрограммированный метод Ньютона. Ответить на вопросы преподавателя.

3.2.Теоретическое введение

Метод Ньютона (метод касательных)

Метод состоит в том, что на k-й итерации в точке $(x_k;\ (F(x_k))$ строится касательная к кривой $y=F(x)$ и ищется точка пересечения касательной с осью абсцисс. Если задан интервал изоляции корня $[a;\ b]$, то за начальное приближение x_0 принимается тот конец отрезка, на котором $F(x_0)F''(x_0) > 0$. Уравнение касательной, проведенной к кривой $y=F(x)$ в точке M_0 с координатами $(x_0;\ (F(x_0))$, имеет вид: $y-F(x_0)=F'(x_0)(x-x_0)$.

За следующее приближение корня x_1 примем абсциссу точки пересечения касательной с осью OX. $x_1 = x_0 - \frac{F(x_0)}{F'(x_0)}$. При этом необходимо, чтобы $F'(x_0) \neq 0$.

Аналогично могут быть найдены и следующие приближения как точки пересечения с осью абсцисс касательных, проведенных в точках M_1, M_2 и т. д. Формула для $k+1$-го приближения имеет вид:

$$x_{k+1} = x_k - \frac{F(x_k)}{F\prime(x_k)}.$$

Для завершения итерационного процесса используется условие $|F(x_k)| < \varepsilon$ или $|x_{k+1} - x_k| < \varepsilon$. Объем вычислений в методе Ньютона больше, чем в других методах, поскольку приходится находить значение не только функции $F(x)$, но и ее производной. Однако скорость сходимости здесь значительно выше.

Литература:

Воробейкина, И.В. Программирование средств защиты информации: учебное пособие для студентов специальности 10.05.03 «Информационная безопасность автоматизированных систем» очной формы обучения / И.В. Воробейкина; БГАРФ ФГБОУ ВО «КГТУ». – Калининград: Изд-во БГАРФ, 2021. - 70 с.

Глава 2. Численные методы как основа алгоритмов.

3.3.Задание к лабораторной работе

1. Начертить график выбранной вами нелинейной функции и показать на графике, как найти корень.

2. Запишите правила нахождения производной.

3. Обоснуйте типы данных, которые вы выбрали для программирования метода дихотомии.

4. Как изменится количество итераций, если увеличить точность вычисления?

5. Решить уравнение $x^4 - x - 1 = 0$ на отрезке $[-1; 0]$ методом Ньютона с точностью $\varepsilon = 0.0001$.

6. Создать блок-схему и программный код.

3.4.Методические указания и порядок выполнения работы

Пример 2.1. Решить уравнение $x^3 + x - 1 = 0$ на отрезке $[0; 1]$ методом Ньютона с точностью $\varepsilon = 0{,}01$.

Решение.

Определим производные заданной функции $F(x) = x^3 + x - 1$: $F'(x) = 3x^2 + 1$; $F''(x) = 6x$. Проверим выполнение условия сходимости на концах заданного интервала: $F(0)F''(0) = 0$ – не выполняется, $F(1)F''(1) = 1 \cdot 6 > 0$ – выполняется. Значит, за начальное приближение корня принимаем $x_0 = 1$.

Находим первое приближение:

$$x_1 = x_0 - \frac{F(x_0)}{F'(x_0)} = x_0 - \frac{x_0^3+x_0-1}{3x_0^2+1} = 1 - \frac{1^3+1-1}{3\cdot 1^2+1} = 0.75.$$

Так как $|x_1 - x_0| = |0.75 - 1| = 0.25 > 0.01$, итерационный процесс продолжается.

Аналогично находится второе приближение:

$$x_2 = x_1 - \frac{F(x_1)}{F'(x_1)} = x_1 - \frac{x_1^3+x_1-1}{3x_1^2+1} = 0.75 - \frac{0.75^3+0.75-1}{3\cdot 0.75^2+1} = 0.686.$$

Так как $|x_2 - x_1| = |0.686 - 0.75| = 0.064 > 0.01$, итерационный процесс продолжается.

Третье приближение:

$$x_3 = x_2 - \frac{F(x_2)}{F'(x_2)} = x_2 - \frac{x_2^3+x_2-1}{3x_2^2+1} = 0.686 - \frac{0.686^3+0.686-1}{3\cdot 0.686^2+1} = 0.682.$$

Так как $|x_3 - x_2| = |0.682 - 0.686| = 0.004 < 0.01$, итерационный процесс завершается. Таким образом, приближенным решением данного уравнения является $x = 0.68$.

2. Используя блок-схему написать программный код.

3.5. Индивидуальное задание

Вариативность не предполагается. Освоить метод Ньютона, создать блок-схему и программный код.

Найти решение уравнения $x^4-x-1=0$ на отрезке $[-1; 0]$ с точностью $e=0,0001$ методом Ньютона. Создать блок-схему и программный код

3.6. Требования к отчету и защите

Показать выполненную в тетради и на компьютере работу. Знать ответы на сформулированные выше вопросы. Защита работы проводится во время занятий. После защиты работа помещается в ЭИОС.

4. ЛАБОРАТОРНАЯ РАБОТА № 3. АЛГОРИТМ АФФИННОГО ШИФРА

4.1. Общие сведения

Цель: Изучить аффинный шифр и методы его программирования.

Материалы, оборудование, программное обеспечение: online-компилятор C++ (необходима бесперебойная работа сети ИНТЕРНЕТ) или Visual Studio C++

Условия допуска к выполнению: предварительно пройти инструктаж по ТБ. Показать конспект по теоретической подготовке и решить подготовительный пример.

Критерии положительной оценки: Показать выполненную работу: решенный пример и запрограммированный аффинный шифр. Ответить на вопросы преподавателя.

4.2. Теоретическое введение

Аффинный шифр – комбинация аддитивного и мультипликативного шифров с парой ключей. Первый ключ используется мультипликативным шифром, второй – аддитивным шифром. Аффинный шифр – фактически два шифра, применяемые один за другим. При аффинном шифре отношение между исходным текстом P и шифрованным текстом C определяется, как это показано ниже.

$C = (P \times k_1 + k_2) \bmod 26$ $P = ((C - k_2) \times k_1^{-1}) \bmod 26$,

где k_1^{-1} мультипликативная инверсия k_1, а $(-k_2)$ – аддитивная инверсия k_2.

Аффинный шифр использует пару ключей, в которой первый ключ из Z_{26}^*, а второй – из Z_{26}. $X=Y=Z_{26}$, $K= Z_{26^*} \times Z_{26}$. $k=(\alpha,\beta) \in K$, $\alpha \neq 0$, $x=(x_1,\ldots,x_m)$, $y=(y_1,\ldots,y_m)$, полагаем

$y= E_k(x)=(\alpha \times x_1+\beta,\ldots,\alpha \times x_m+\beta)$, $x=D_k(y)=((y_1+(26-\beta)) \times \alpha^{-1},\ldots, (y_m+(26-\beta)) \times \alpha^{-1})$, где + и × являются операциями кольца Z_{26}, а α^{-1} элемент мультипликативной группы Z_{26^*}, обратный α.

Исходный текст состоит из строчных букв от *a* до *z*, а зашифрованный текст состоит из прописных букв от *A* до *Z*. Каждой букве присвоим числовое значение.

a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z
A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
00	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25

Числовое значение
Зашифрованный текст
Исходный текст

Здесь каждому символу сопоставлено целое число из Z_{26}.

Литература:

Воробейкина, И.В. Программирование средств защиты информации: учебное пособие для студентов специальности 10.05.03 «Информационная безопасность автоматизированных систем» очной формы обучения / И.В. Воробейкина; БГАРФ ФГБОУ ВО «КГТУ». – Калининград: Изд-во БГАРФ, 2021. - 70 с.

Глава 3. Аффинный шифр.

4.3.Задание к лабораторной работе

1. Назовите правило выбора мультипликативного и аддитивного ключей.

2. Что такое мультипликативная инверсия?

3. Если в аффинном шифре мультипликативный ключ равен 1, какой шифр получится?

4. Можно ли в системе вычетов Z_{26} в качестве мультипликативного ключа выбрать *4*? Ответ обосновать.

5. Напишите программу, вычисляющую мультипликативные ключи для любой системы вычетов.

6. Запрограммируйте аффинный шифр.

4.4.Методические указания и порядок выполнения работы

Пример 3.1. С помощью аффинного шифра зашифровать сообщение *"happy"* с ключевой парой *(3,2)* в Z_{26}.

Решение:

Мы используем *3* в качестве мультипликативного ключа и *2* в качестве аддитивного ключа. Получаем *"XCVVW"*.

h=07 шифруем *(07×03+2)mod26* зашифрованный текст *23=X*

a=00 шифруем *(00×03+2)mod26* зашифрованный текст *02=C*

p=15 шифруем *(15×03+2) mod 26* зашифрованный текст *21=V*

p=15 шифруем *(15×03+2) mod 26* зашифрованный текст *21=V*

y=24 шифруем *(24×03+2) mod 26* зашифрованный текст *22=W*

4.5.Индивидуальное задание

Освоить аффинный шифр, создать блок-схему и программный код.

С помощью аффинного шифра зашифровать свою фамилию в Z_{26}. Ключевую пару выбрать самостоятельно, создать блок-схему и программный код.

4.6.Требования к отчету и защите

Показать выполненную в тетради и на компьютере работу. Знать ответы на сформулированные выше вопросы. Защита работы проводится во время занятий. После защиты работа помещается в ЭИОС.

5. ЛАБОРАТОРНАЯ РАБОТА № 4. СТРУКТУРА КЛАССА. СПЕЦИФИКАТОРЫ ДОСТУПА *private, public, protected*. ПОЛЯ, МЕТОДЫ. ОБЪЯВЛЕНИЕ ОБЪЕКТОВ

5.1.Общие сведения

Цель: Изучить структуру класса, спецификаторы доступа, объявление объектов.

Материалы, оборудование, программное обеспечение: online-компилятор C++ (необходима бесперебойная работа сети ИНТЕРНЕТ) или Visual Studio C++

Условия допуска к выполнению: предварительно пройти инструктаж по ТБ. Показать конспект по теоретической подготовке и запрограммировать подготовительный пример.

Критерии положительной оценки: Показать выполненную на компьютере работу. Ответить на вопросы преподавателя.

5.2. Теоретическое введение

Введение в классы

Класс – это механизм для создания объекта. Класс объявляется с помощью ключевого слова *class*. Синтаксис объявления:

```
class имя_класса
{
спецификатор_ доступа:
поля и методы
……….
}список объектов;
```

В объявлении класса *список объектов* необязателен. Для компилятора *имя_класса* необязательно, но с точки зрения практики оно необходимо, потому что является именем нового типа данных. После объявления класса за закрывающейся фигурной скобкой ставится точка с запятой. Методы и поля, объявленные внутри класса, становятся *членами* или *элементами* этого класса. Методы можно называть *функциями-членами* класса, а поля – *переменными.*

Спецификаторы доступа могут принимать значения *private* (закрытые), *public* (открытые), *protected* (защищенные). Ключевое слово *private* означает, что члены класса доступны только для других членов того же класса; *public* означает, что члены класса доступны как для других членов класса, так и для любой части программы, в которой объявлен этот класс; *protected* эквивалентен спецификатору *private* и, кроме того, защищенные члены доступны для производных классов (классов-потомков). Если спецификатор доступа не указан, то по умолчанию принимается *private*. После спецификаторов доступа ставится двоеточие.

Ниже приводится простое объявление класса:

```
class myclass
{// закрытый элемент класса
int a;
public:
        void set_a(int num) int get_a();
};
void myclass::set_a(int num) // определение функции
```

{a=num;}
int myclass::get_a()
{return a;}

Этот класс имеет одну закрытую переменную *a*, и две открытые функции, *set_a()* и *get_a()*. Прототипы функций объявляются внутри класса. Функции, которые объявляются внутри класса, называются *функциями-членами (member functions)*. Поскольку *a* является закрытой переменной класса, она недоступна для любой функции вне *myclass*. Однако *set_a()* и *get_a()* являются членами *myclass*, следовательно, они имеют доступ к *a*. Кроме того, *set_a()* и *get_a()*, являясь открытыми членами *myclass*, могут вызываться из любой части программы, использующей *myclass*. Для определения функции-члена необходимо связать имя класса, частью которого является функция-член, с именем функции. Это достигается путем написания имени функции вслед за именем класса с двумя двоеточиями. Два двоеточия называются *оператором расширения области видимости (scope resolution operator)*.

При определении функции-члена пользуйтесь следующей основной формой:

Тип_возвр_значения имя_класса::имя_функции(список_параметров)
{тело функции}

Здесь *имя_класса* – это имя того класса, которому принадлежит определяемая функция. Чтобы создать объект, используйте имя класса, как спецификатор типа данных. Например, в этой строке объявляются два объекта типа *myclass*:

myclass ob1, ob2; // это объекты типа myclass

После того как объект класса создан, можно обращаться к открытым членам класса, используя оператор точка (.). Предположим, что ранее объекты были объявлены, тогда следующие инструкции вызывают *set_a()* для объектов *ob1* и *ob2*:

ob1.set_a(10); // установка версии a объекта ob1 равной 10
ob2.set_a(99); // установка версии a объекта ob2 равной 99

Каждый объект содержит собственную копию всех данных, объявленных в классе. Это значит, что *a* в *ob1* отлично от *a* в *ob2*.

В качестве примера рассмотрим программу:

Пример 4.1

```
#include <iostream>
using namespace std;
class myclass
{// закрытая часть myclass
int a;
public:
```

```
        void set_a(int num);
        int get_a();
};
void myclass::set_a(int num) {a=num;}
int myclass::get_a(){return a;}
int main()
{
myclass ob1, ob2;
obi.set_a(10);
ob2.set_a(99);
cout<<ob1.get_a() << "\n";
cout<<ob2.get_a()<<"\n";
return 0;
}
```

Программа выводит на экран величины *10* и *99*.

Точно так же, как открытые функции-члены, могут существовать и открытые переменные-члены. Например, если бы *a* была объявлена в открытой секции *myclass*, тогда к ней можно было бы обратиться из любой части программы. В этом примере, поскольку *a* объявлена открытым членом *myclass*, к ней имеется явный доступ из *main()*.

Пример 4.2

```
#include <iostream>
using namespace std;
class myclass
{
public:
        int a;// a открыта, и здесь не нужны функции set_a() и get__a ()
};
int main()
{
myclass ob1, ob2;
// здесь есть явный доступ к a
ob1.a = 10;
ob2.a = 99;
cout << ob1.a << "\n"<< ob2.a << "\n";
return 0;
}
```

Литература:

Воробейкина, И.В. Технологии и методы программирования, часть II: учебное пособие для студентов специальности 10.05.03 «Информационная безопасность автоматизированных систем» очной формы обучения / И.В. Воробейкина; БГАРФ ФГБОУ ВО «КГТУ». – Калининград: Изд-во БГАРФ, 2021.

Глава 1. Классы.

5.3.Задание к лабораторной работе

Задание к лабораторной работе

1. Для чего необходимы классы?

2. Объясните структуру класса.

3. Что такое спецификаторы доступа? Перечислите их и объясните их работу.

4. Создайте класс *card*, который поддерживает каталог библиотечных карточек. Этот класс должен хранить заглавие книги, имя автора и выданное на руки число экземпляров книги. Заглавие и имя автора храните в виде строки символов, а количество экземпляров – в виде целого числа. Используйте открытую функцию-член *store()* для запоминания информации о книгах и открытую функцию-член *show()* для вывода информации на экран. В функцию *main()* включите демонстрацию работы созданного класса.

5. Создать класс, содержащий фамилию, пол и год рождения в закрытой части класса. Включите в класс открытую функцию для ввода этих данных и открытую функцию для вывода данных на экран.

5.4.Методические указания и порядок выполнения работы

Создать классы, информацию для их заполнения получить в *main()*, обязательно проверить корректность вводимой информации.

Рассмотрим пример. В этой программе создается класс *stack*, реализующий стек, который можно использовать для хранения символов.

Пример 4.3

```
#include <iostream> using namespace std;
#define SIZE 10
// Объявление класса stack для символов
class stack
{
char stck[SIZE]; // содержит стек
int tos; // индекс вершины стека
public:
        void init(); // инициализация стека
        void push(char ch); // помещает в стек символ
        char pop(); // выталкивает из стека символ
```

```
};
// Инициализация стека
void stack::init(){tos=0;}
// Помещение символа в стек
void stack:: push(char ch)
{
if (tos==SIZE)
        {
        cout <<”Стек полон";
        return ;
        }
stck[tos] = ch;
tos++;
}
// Выталкивание символа из стека
char stack::pop()
{
if (tos==0)
        {
        cout << "Стек пуст";
        return 0; // возврат нуля при пустом стеке
        }
        tos--
        return stck[tos];
        }
        int main()
        {
        stack s1, s2; // создание двух стеков
        int i ;// инициализация стеков
        s1.init();
        s2.init();
        s1.push('a');
        s2.push('x');
        s1.push('b');
        s2.push('y');
        s1.push(‘c');
        s2.push('z');
        for(i=0;i<3;i++) cout << "символ из s1:" << s1.pop() <<"\n";
        for(i=0;i<3;i++) cout << "символ из s2:" << s2.pop()<<"\n";
        return 0;
```

```
}
```

Эта программа выводит на экран следующее:

символ из s1: c
символ из s1: b
символ из s1: a
символ из s2: z
символ из s2: y
символ из s2: x

Проанализируем программу. Класс *stack* содержит две закрытые переменные: *stck* и *tos*. Массив *stck* содержит символы, фактически помещаемые в стек, а *tos* содержит индекс вершины стека. Открытыми функциями стека являются *init(), push()* и *pop()*, которые, соответственно, инициализируют стек, помещают символ в стек и выталкивают его из стека. Внутри функции *main()* создаются два стека – *s1* и *s2* – , и в каждый из них помещаются по три символа. Важно понимать, что один объект (стек) не зависит от другого. Поэтому у символов в *s1* нет способа влиять на символы в *s2*. Каждый объект содержит свою собственную копию *stck* и *tos*. Это фундаментальная для понимания объектов концепция. Хотя все объекты класса имеют общие функции-члены, каждый объект создает и поддерживает свои собственные данные.

5.5. Индивидуальное задание

Вариативность не предполагается. Освоить создание классов, создать программный код.

Создать класс, содержащий фамилию, пол и год рождения в закрытой части класса. Включите в класс открытую функцию для ввода этих данных и открытую функцию для вывода данных на экран.

5.6. Требования к отчету и защите

Показать выполненную на компьютере работу. Знать ответы на сформулированные выше вопросы. Защита работы проводится во время занятий. После защиты работа помещается в ЭИОС.

6. ЛАБОРАТОРНАЯ РАБОТА № 5. ПРОГРАММЫ С КОНСТРУКТОРОМ И БЕЗ КОНСТРУКТОРА. ПРОГРАММИРОВАНИЕ КОНСТРУКТОРОВ С ПАРАМЕТРАМИ

6.1. Общие сведения

Цель: Познакомиться с конструкторами класса.

Материалы, оборудование, программное обеспечение: online-компилятор C++ (необходима бесперебойная работа сети ИНТЕРНЕТ) или Visual Studio C++

Условия допуска к выполнению: предварительно пройти инструктаж по ТБ. Показать конспект по теоретической подготовке и запрограммировать подготовительный пример.

Критерии положительной оценки: Показать выполненную на компьютере работу. Ответить на вопросы преподавателя.

Литература:

Воробейкина, И.В. Технологии и методы программирования, часть II: учебное пособие для студентов специальности 10.05.03 «Информационная безопасность автоматизированных систем» очной формы обучения / И.В. Воробейкина; БГАРФ ФГБОУ ВО «КГТУ». – Калининград: Изд-во БГАРФ, 2021.

Глава 1. Классы.

6.2. Теоретическое введение

Конструкторы

При работе с объектами часто возникает необходимость инициализации. Для этого существует *функция-конструктор*, включаемая в описание класса. **Конструктор класса вызывается при создании объекта этого класса.** Таким образом, любая необходимая объекту инициализация при наличии конструктора выполняется автоматически. Конструктор имеет то же имя, что и класс, частью которого он является, и не имеет возвращаемого значения.

Пример 5.1

```
#include <iostream>
using namespace std;
class myclass
{
int a;
public:
    myclass(); // конструктор
    void show();
    };
    myclass::myclass()
    {
    cout << "Рапботает конструктор\n";
    a= 10;
    }
    void myclass::show()
    {
    cout << a;
    }
```

```
int main()
{
myclass ob;
ob.show();
return 0;
}
```

В этом примере значение *a* инициализируется конструктором *myclass()*. Конструктор вызывается тогда, когда создается объект *ob*. Как уже говорилось, конструктор не имеет возвращаемого значения.

Для глобальных объектов конструктор объекта вызывается тогда, когда начинается выполнение программы. Для локальных объектов конструктор вызывается всякий раз при выполнении инструкции объявления переменной.

Фактически конструктор может выполнить любой тип операции. Тем не менее считается, что конструктор не должен делать ничего, не имеющего отношения к инициализации или возвращению объектов в исходное состояние. Применение конструктора для действий, прямо не связанных с инициализацией, является очень плохим стилем программирования, и его следует избегать.

Конструктору можно передавать аргументы. Для этого добавляются необходимые параметры в объявление и определение конструктора. Затем при объявлении объекта задаются параметры в качестве аргументов.

Пример 5.2

```
#include <iostream>
using namespace std;
class myclass
{
int a;
public:
myclass(int x); // конструктор
void show();
};
myclass::myclass(int x){cout << "Работает конструкторе\n"; a = x;}
void myclass::show(){cout << a << "\n";}
int main()
{
myclass ob(4);
ob.show();
return 0;
}
```

Здесь конструктор класса *myclass* имеет один параметр. Значение, передаваемое в *myclass(),* используется для инициализации переменной *a*. Обратите внимание на то, как в функции *main()* объявляется объект *ob*. Число *4*, записанное в круглых скобках, является аргументом, передаваемым параметру *x* конструктора *myclass()*, который используется для инициализации переменной *a*. В отличие от конструктора деструктор не может иметь параметров: отсутствует механизм передачи аргументов удаленному объекту.

В следующем примере конструктору *myclass()* передается два аргумента.

Пример 5.3

```
#include <iostream>
using namespace std;
class myclass
{
int a, b;
public:
        myclass(int x, int y); // конструктор
        void show();
};
        myclass::myclass(int x, int y)
        {
        cout << "В конструкторе\n";
        a = x;
        b = y;
        }
        void myclass::show()
        {
        cout << a << ' ' << b << "\n";
        }
        int main()
        {
        myclass ob(4, 7);
        ob.show();
        return 0;
        }
```

6.3.Задание к лабораторной работе

1. Как можно проинициализировать объект класса?

2. Чем конструктор отличается от обычных функций?

3. Можно ли при инициализации полей класса обойтись без конструктора?

4. Когда конструктор начинает работать?

5. Создайте класс *box,* конструктору которого передаются три значения типа *double*, представляющие собой длины сторон параллелепипеда. Класс *box* должен вычислять его объем и хранить результат также в виде значения типа *double*. Включите в класс функцию-член *voI()*, которая будет выводить на экран объем любого объекта типа *box.*

6. Создать класс, конструктору которого передаются три значения типа *float*, являющиеся длинами сторон треугольника. В классе создать: 1) функцию, которая вычисляет площадь треугольника по формуле Герона; 2) функцию, выводящую результат на экран.

6.4.Методические указания и порядок выполнения работы

Прежде чем приступить к выполнению задания по лабораторной работе, проанализируйте программы, указанные ниже:

В примере 6.4 в классе *stack* для установки переменной индекса стека требовалась функция инициализации. Это именно тот тип действия, для выполнения которого и придуман конструктор. В примере 6.5 представлена улучшенная версия класса *stack*, где для автоматической инициализации объекта стека при его создании используется конструктор.

Пример 5.4

```
#include <iostream>
using namespace std;
#define SIZE 10
// Объявление класса stack для символов
class stack
{
char stck[SIZE]; // содержит стек
int tos; // индекс вершины стека
public:
        stack(); // конструктор
        void push(char ch); // помещает в стек символ
        char pop(); // выталкивает из стека символ
        };
        // Инициализация стека
        stack::stack()
        {
        cout << "Работа конструктора стека \n";
        tos=0;
        }
        // Помещение символа в стек
        void stack:rpush(char ch)
```

```
        {
        if (tos==SIZE)
                {
                cout << "Стек полон";
                return;
                }
                stck[tos]=ch;
                tos++;
        }
// Выталкивание символа из стека
char stack::pop()
        {
        if (tos==0)
                {
                cout << "Стек пуст";
                return 0; // возврат нуля при пустом стеке
                }
        tos--;
        return stck[tos];
        }
int main()
{
// образование двух автоматически инициализируемых стеков
stack s1, s2;
int i;
s1.push('a');
s2.push('x');
s1.push('b');
s2.push('y');
s1.push('c');
s2.push('z');
for(i=0; i<3; i++) cout << "символ из s1:" << s1.popO << "\n";
for(i=0; i<3; i++) cout << "символ из s2:" << s2.pop() << "\n";
return 0;
}
```

Обратите внимание, что теперь вместо отдельной, специально вызываемой программой функции, задача инициализации выполняется конструктором автоматически. Это исключает возможность того, что по ошибке инициализация не будет выполнена.

Пример 5.5

Здесь представлена следующая версия класса *stack*, в котором конструктор с параметром используется для присвоения стеку имени. Это односимвольное имя необходимо для идентификации стека в случае возникновения ошибки.

```
#include <iostream>
using namespace std;
#define SIZE 10
// Объявление класса stack для символов
class stack
{
char stck[SIZE]; // содержит стек
int tos; // индекс вершины стека
char who; // идентифицирует стек
public:
        stack(char c); // конструктор
        void push(char ch); // помещает в стек символ
        char pop(); // выталкивает из стека символ
};
stack::stack(char c) // Инициализация стека
        {
        tos = 0;
        who = c;
        cout << "Работа конструктора стека " << who << "\n";
        }
void stack::push(char ch) // Помещение символа в стек
        {
        if (tos==SIZE)
                {
                cout << "Стек " << who << " полон \n";
                return;
                }
                stck[tos]=ch;
                tos++;
                }
                char stack::pop()// Выталкивание символа из стека
                {
                if (tos==0)
                        {
                        cout << "Стек " << who << " пуст ";
                        return 0; // возврат нуля при пустом стеке
```

```
                    }
                    tos--;
                    return stck[tos];
                    }
int main()
{
// образование двух автоматически инициализируемых стеков
stack s1('A'), s2('B');
int i;
s1.push('a');
s2.push(‘x');
s1.push('b');
s2.push('y');
s1.push('c');
s2.push('z');
// Это вызовет сообщения об ошибках
for(i=0; i<5; i++) cout<<"символ из стека s1:"<<s1.pop();
cout<< "\n";
for(i=0; i<5; i++) cout << "символ из стека s2: "<< s2.pop();
cout<<"\n";
return 0;
```

Присвоение имени объекту является особенно полезным при отладке, когда важно выяснить, какой из объектов вызывает ошибку.

6.5.Индивидуальное задание

Вариативность не предполагается. Освоить создание конструкторов в классе, создать программный код.

Создать класс, конструктору которого передаются три значения типа *float*, являющиеся длинами сторон треугольника. В классе создать: 1) функцию, которая вычисляет площадь треугольника по формуле Герона; 2) функцию, выводящую результат на экран.

6.6.Требования к отчету и защите

Показать выполненную на компьютере работу. Знать ответы на сформулированные выше вопросы. Защита работы проводится во время занятий. После защиты работа помещается в ЭИОС.

7. ЛАБОРАТОРНАЯ РАБОТА № 6. УПРАВЛЕНИЕ ДОСТУПОМ К БАЗОВОМУ КЛАССУ. КОНСТРУКТОРЫ И НАСЛЕДОВАНИЕ

7.1. Общие сведения

Цель: Познакомиться с наследованием, освоить методы доступа к базовым классам.

Материалы, оборудование, программное обеспечение: online-компилятор C++ (необходима бесперебойная работа сети ИНТЕРНЕТ) или Visual Studio C++

Условия допуска к выполнению: предварительно пройти инструктаж по ТБ. Показать конспект по теоретической подготовке и запрограммировать подготовительный пример.

Критерии положительной оценки: Показать выполненную на компьютере работу. Ответить на вопросы преподавателя.

7.2. Теоретическое введение

Введение в наследование

Наследование – это механизм, посредством которого один класс может наследовать свойства другого. Наследование позволяет строить иерархию классов. При наследовании одного класса другим, класс, который наследуется, называют *базовым классом (base class)*. Наследующий класс называют *производным классом (derived class).* Обычно процесс наследования начинается с задания базового класса. Базовый класс определяет все те качества, которые будут общими для всех производных от него классов, то есть, он представляет собой наиболее общее описание ряда характерных черт. Производный класс наследует эти общие черты и добавляет свойства, характерные только для него.

```
// Определение базового класса
class B
{
int i;
public:
        void set_i(int n);
        int get_i();
        };
        // Определение производного класса
        class D: public B
        {
         int j ;
        public:
        void set_j(int n);
        int mul();
```

} ;

Обратите внимание, что после имени класса *D* имеется двоеточие, за которым следуют ключевое слово *public* и имя класса *B*. Для компилятора это указание на то, что класс *D* будет наследовать все компоненты класса *B*. Само ключевое слово *public* информирует компилятор о том, что, поскольку класс *B* будет наследоваться, значит, все открытые элементы базового класса будут также открытыми элементами производного класса. Однако все закрытые элементы базового класса останутся закрытыми, и к ним не будет прямого доступа из производного класса.

Пример 6.1

```
#include <iostream>
using namespace std;
// Определение базового класса
class B
{
int i;
public:
        void set_i(int n);
        int get_i ();
        };
// Определение производного класса
class D: public B
{
int j;
public:
        void set_j(int n);
        int mul ();
};
        // Задание значения i в базовом классе
        void B::set_i(int n){i = n;}
        // Возвращение значения i в базовом классе
        int B::get_i(){return i;}
        // Задание значения j в производном классе
        void D::set_j(int n){j = n;}
        // Возвращение значения i базового класса и j производного
        int D::mul()
        {
        // производный класс может вызывать
        //функции-члены базового класса
        return j * get_i();
```

```
}
int main()
{
D ob;
ob.set_i(10); // загрузка i в базовый класс
ob.set_j(4); // загрузка j в производный класс
cout << ob.mul(); // вывод числа 40
return 0;
}
```

Обратите внимание на определение функции *mul()*. Функция *get_i()*, которая является членом базового класса *B*, а не производного *D*, вызывается внутри класса *D* без всякой связи с каким бы то ни было объектом. Это возможно потому, что открытые члены класса *B* становятся открытыми членами класса *D*. В функции *mul()* вместо прямого доступа к *i*, необходимо вызывать функцию *get_i()*, поскольку закрытые члены базового класса (в данном случае *i*) остаются закрытыми для нее и недоступными из любого производного класса. Причина, по которой закрытые члены класса становятся недоступными для производных классов, – поддержка инкапсуляции. Если бы закрытые члены класса становились открытыми просто посредством наследования этого класса, инкапсуляция была бы совершенно несостоятельна.

Форма наследования базового класса:

class имя_производного_класса: с_д имя_базового_класса {};

Здесь *с_д* (спецификатор доступа) – это одно из следующих трех ключевых слов: *public* (открытый), *private* (закрытый) или *protected* (защищенный).

Когда базовый класс наследуется производным классом как открытый (*public*), защищенный член базового класса становится защищенным членом производного класса. Когда базовый класс наследуется как закрытый (*private*), то защищенный член базового класса становится закрытым членом производного класса. Базовый класс может также наследоваться производным классом как защищенный (*protected*). В этом случае открытые и защищенные члены базового класса становятся защищенными членами производного класса. Естественно, что закрытые члены базового класса остаются закрытыми, и они не доступны для производного класса.

Литература:

Воробейкина, И.В. Технологии и методы программирования, часть II: учебное пособие для студентов специальности 10.05.03 «Информационная безопасность автоматизированных систем» очной формы обучения / И.В. Воробейкина; БГАРФ ФГБОУ ВО «КГТУ». – Калининград: Изд-во БГАРФ, 2021.

Глава 5. Наследование.

7.3.Задание к лабораторной работе

1. Создать базовый класс *area_cl*, **открытые** члены которого – основание и высота геометрической фигуры. Создать производные классы *rectangle* и *isosceles*. Каждый производный класс должен включать в себя функцию *area()*, которая возвращает площадь соответственно прямоугольника и равнобедренного треугольника и функцию *show()*, выводящую полученные значения на экран. Для инициализации высоты и длины основания используйте конструктор с параметрами.

2. Создать базовый класс *building* для хранения числа этажей и комнат в здании, а также общую площадь комнат. Создать производный класс *house*, который хранит число лекционных аудиторий и компьютерных классов и производный класс *office*, который хранит число штатных единиц заведующих компьютерным классом и число огнетушителей. В производных классах создать функцию *show()*, выводящую на экран поля базового и производного классов.

Вопросы, на которые необходимо ответить.

1. Перечислите спецификаторы доступа производного класса к базовому.
2. Может ли конструктор производного класса инициализировать поля базового класса?
3. Обязателен ли конструктор в базовом классе?
4. При каких спецификаторах доступа производный класс «не видит» закрытые поля базового класса?

7.4.Методические указания и порядок выполнения работы

Перед выполнением лабораторной работы разберите нижестоящие примеры, особое внимание обратите на работу спецификаторов доступа.

Здесь представлены базовый и наследующий его производный классы (наследование со спецификатором *public*):

Пример 6.2

```
#include <iostream>
using namespace std;
class base
{
        int x;
public:
        void setx(int n) { x = n; }
        void showx() { cout << x << ' \n'; }
};
// Класс наследуется как открытый
class derived: public base
```

```
{
        int y;
public:
        void sety(int n) { y = n; }
        void showy() { cout << y << ' \n'; }
        };
int main()
{
derived ob;
ob.setx(10); // доступ к члену базового класса
ob.sety(20); // доступ к члену производного класса
ob.showx(); // доступ к члену базового класса
ob.showy(); // доступ к члену производного класса
return 0;
}
```

Как показано в программе, поскольку класс *base* наследуется как открытый, открытые члены класса *base* – функции *setx()* и *showx()* – становятся открытыми производного класса *derived* и поэтому доступны из любой части программы. Следовательно, совершенно правильно вызывать эти функции из функции *main()*.

Наследование производным классом базового как открытого совсем не означает, что для производного класса станут доступными закрытые члены базового. Закрытые члены базового класса остаются закрытыми, **независимо от того, как он наследуется**.

Хотя открытые члены базового класса при наследовании с использованием спецификатора *private* в производном классе становятся закрытыми, **внутри** производного класса они остаются доступными.

Пример 6.3

```
#include <iostream>
using namespace std;
class base
{
        int x;
public:
        void setx(int n) { x = n; }
        void showx() { cout << x << ' \n'; }
};
// Класс base наследуется как закрытый
class derived: private base
{
```

```
        int y;
public:
// метод setx() доступен в классе derived
        void setxy(int n, int m) { setx(n); y = m; }
    // метод showx() доступен в классе derived
        void showxy() { showx(); cout << y << *\n *; }
};
int main()
{
derived ob;
ob.setxy(10, 20) ;
ob.showxy() ;
return 0;
}
```

В данном случае функции *showx()* и *setx()* доступны внутри производного класса, поскольку они являются закрытыми членами этого класса.

7.5. Индивидуальное задание

Вариативность не предполагается. Освоить наследование в классах, создать программный код.

Создать базовый класс *Zoo* для зверей и птиц. Защищенные члены класса *Zoo* – число ног и год рождения. В классе *Zoo* есть конструктор. Производные классы *Anim* и *Bird* содержат по одному полю – название зверя или птицы. Конструкторы *Anim()* и *Bird()* должны передавать необходимые аргументы классу *Zoo* и инициализировать собственные поля. Классы *Anim* и *Bird* должны содержать функцию, выводящую на экран всю информацию о своем объекте.

7.6. Требования к отчету и защите

Показать выполненную на компьютере работу. Знать ответы на сформулированные выше вопросы. Защита работы проводится во время занятий. После защиты работа помещается в ЭИОС.

8. ЛАБОРАТОРНАЯ РАБОТА № 7. ВСТРАИВАЕМЫЕ ФУНКЦИИ В КЛАССАХ. ДОСТУП К ЧЛЕНАМ КЛАССА С ПОМОЩЬЮ ОПЕРАТОРА ТОЧКА (.) И ЧЕРЕЗ УКАЗАТЕЛЬ

8.1. Общие сведения

Цель: Изучить методы доступа к членам класса.

Материалы, оборудование, программное обеспечение: online-компилятор C++ (необходима бесперебойная работа сети ИНТЕРНЕТ) или Visual Studio C++

Условия допуска к выполнению: предварительно пройти инструктаж по ТБ. Показать конспект по теоретической подготовке и запрограммировать подготовительный пример.

Критерии положительной оценки: Показать выполненную на компьютере работу. Ответить на вопросы преподавателя.

8.2. Теоретическое введение

Понятие встраиваемых функций

В *C++* можно задать функцию, которая на самом деле не вызывается, а ее тело встраивается в программу в месте ее вызова. Преимуществом встраиваемых (*inline*) функций является то, что они не связаны с механизмом вызова функций и возврата ими своего значения. Это значит, что встраиваемые функции могут выполняться гораздо быстрее обычных. Выполнение машинных команд, которые генерируют вызов функции и возвращение функцией своего значения, занимает определенное время. Если функция имеет параметры, то ее вызов занимает еще большее время. Недостатком встраиваемых функций является то, что если они слишком большие и вызываются слишком часто, объем программ сильно возрастает. Из-за этого применение встраиваемых функций обычно ограничивается короткими функциями. Для объявления встраиваемой функции просто впишите спецификатор *inline* перед определением функции. В этой программе показано, как объявить встраиваемую функцию:

Пример 7.1

```
// Пример встраиваемой функции
#include <iostream>
using namespace std;
inline int even(int x){return !(x%2);}
int main()
{
if (even(10)) cout <<"10 является четным\n";
if (even(11)) cout << "11 является нечетным\n";
return 0;
}
```

В этом примере функция *even()*, которая возвращает истину при четном аргументе, объявлена встраиваемой. Это означает, что строка

```
if (even(10)) cout << "10 является четным\n";
```

функционально идентична строке

```
if (!(10%2)) cout <<"10 является четным\n";
```

Указатели на объекты

До сих пор доступ к членам объекта осуществлялся с помощью оператора точка (.). Однако доступ к члену объекта можно получить также и через указатель на этот объект. В этом случае обычно применяется оператор стрелка (–>). Указатель на объект объявляется точно так же, как и указатель на переменную любого другого типа: задается имя класса этого объекта, а затем имя переменной со звездочкой перед ним. Для получения адреса объекта перед ним необходим оператор &, точно так же, как это делается для получения адреса переменной другого типа. Как и для любого другого указателя, если вы инкрементируете указатель на объект, он будет указывать на следующий объект такого же типа.

Пример 7.2

```
// Пример использования указателя на объект:
#include <iostream>
using namespace std;
class myclass
{
int a;
public:
        myclass(int x){a = x;};
        int get(){return a;};
};
int main()
{
myclass ob(120); // создание объекта
myclass *p; // создание указателя на объект
p = &ob; // передача адреса ob в p
cout << "Значение, получаемое через объект:" << ob.get() << "\n";
cout << "Значение, получаемое через указатель:" << p–>get();
return 0;
}
```

C++ содержит специальный указатель *this*. Это указатель, который автоматически передается любой функции-члену при ее вызове и указывает на объект, генерирующий вызов. Рассмотрим следующую инструкцию:

```
ob.f1(); // предположим, что ob – это объект
```

Функции *f1()* автоматически передается указатель на объект *ob*. Этот указатель и называется *this*. Указатель *this* передается только функциям-членам. Дружественным функциям указатель *this* не передается.

Итак, если вызывается функция-член, ей автоматически передается указатель *this* на объект, который является источником вызова.

Пример 7.3

```
// Демонстрация указателя this
#include <iostream>
#include <cstring>
using namespace std;
class inventory
{
char item[20];
double cost;
int on_hand;
public:
        inventory(char *i, double c, int o)
        {
        strcpy(this–>item, i); // доступ к члену
        this–>cost = c; // через
        this–>on_hand = o; // указатель this
        }
        void show()
        {
        cout << this–>item; // использование this для доступа к членам
        cout <<” $” << this–>cost;
        cout << " On hand: " << this–>on_hand << "\n";
        }
};
        int main()
        {
        inventory ob("wrench", 4.95, 4);
        ob.show();
        return 0;
        }
```

Здесь к переменным-членам объекта *ob* осуществляется прямой доступ через указатель *this*. Таким образом, внутри функции *show()* следующие две инструкции равнозначны:

```
cost = 123.23;
this–>cost = 123.23;
```

На самом деле первая форма – это сокращенная запись второй. Очевидно, сокращенная форма намного проще, но здесь важно понимать, что под этим сокращением подразумевается.

Литература:

Воробейкина, И.В. Технологии и методы программирования, часть II: учебное пособие для студентов специальности 10.05.03 «Информационная безопасность автоматизированных систем» очной формы обучения / И.В. Воробейкина; БГАРФ ФГБОУ ВО «КГТУ». – Калининград: Изд-во БГАРФ, 2021.

Глава 5. Наследование.

8.3.Задание к лабораторной работе

1. Используя класс *stack* из примера 5.4 (лабораторная работа №5), добавьте в программу функцию *showstack(),* которой в качестве аргумента передается объект типа *stack.* Эта функция должна выводить содержимое стека на экран.

Вопросы:

1. Когда удобно применять встраиваемые функции?
2. Инструкция *inline* – запрос компилятору или команда?
3. В каких случаях компилятор может отказать интерпретировать функцию как встраиваемую?
4. Какие преимущества предоставляет программе inline-функция по сравнению с обычными функциями?

8.4.Методические указания и порядок выполнения работы

Пример 7.4

```
// Демонстрация встраиваемой функции-члена
#include <iostream>
using namespace std;
class samp
{
int i, j;
public:
        samp(int a, int b);
        int divisible(); // встраивание происходит в этом определении
        };
        samp::samp(int a, int b){i = a; j = b;}
        /* Возврат 1, если i без остатка делится на j. Тело этой функции-члена встраивается в программу*/
        inline int samp::divisible(){return !(i%j);}
        int main()
        {
        samp obi(10,2), ob2(10,3);// это истина
```

```
if (obi.divisible () ) cout << "10 делится на 2_\n";// это ложь
if(ob2.divisible()) cout << "10 делится на 3\n";
return 0;
}
```

Пример 7.5

```
// Пример арифметики указателей на объекты:
#include <iostream>
using namespace std;
class samp
{
int a, b;
public:
    samp(int n, int m) { a = n; b = m; }
    int get_a() { return a; }
    int get_b() { return b; }
};
    int main()
    {
    samp ob[4] = { samp(1, 2), samp(3, 4), samp (5, 6) , samp(7, 8)};
    int i;
    samp *p;
    p = ob; // получение адреса начала массива
    for(i=0; i<4; i++)
    {
    cout << p–>get_a() << p–>get_b() << "\n";
    p++; // переход к следующему объекту
    }
    return 0;
    }
```

Эта программа выводит на экран следующее:

```
1 2
3 4
5 6
7 8
```

Как видно из результата, при каждом инкрементировании указателя *p* он указывает на следующий объект массива.

Пример 7.6

Дана программа, переделайте все соответствующие обращения к членам класса так, чтобы в них явно присутствовал указатель *this*.

```
#include <iostream>
```

```
using namespace std;
class myclass
{
int a, b;
public:
        myclass(int n, int m) { a = n; b = m; }
        int add() { return a + b; }
        void show()
                {
                int t;
                t = add(); // вызов функции-члена
                cout << t <<"\n";
                }
};
int main()
{
myclass ob(10, 14);
ob.show();
return 0;
}
```

Решение:

```
class myclass
{
int a, b;
public:
        myclass(int n, int m) { this–>a = n; this–>b = m; }
        int add() { return this–>a + this–>b; }
        void show();
{
int t;
t = this–>add(); // вызов функции-члена
cout << t <<”\n”;
}
        };
        int main ()
        {
        myclass ob(10, 14);
        ob.show();
        return 0;
        }
```

8.5.Индивидуальное задание

Вариативность не предполагается. Освоить доступы к членам класса, создать программный код.

8.6.Требования к отчету и защите

Показать выполненную на компьютере работу. Знать ответы на сформулированные выше вопросы. Защита работы проводится во время занятий. После защиты работа помещается в ЭИОС.

9. ЛАБОРАТОРНАЯ РАБОТА № 8. ДРУЖЕСТВЕННЫЕ ФУНКЦИИ

9.1.Общие сведения

Цель: Изучить дружественные функции.

Материалы, оборудование, программное обеспечение: online-компилятор C++ (необходима бесперебойная работа сети ИНТЕРНЕТ) или Visual Studio C++

Условия допуска к выполнению: предварительно пройти инструктаж по ТБ. Показать конспект по теоретической подготовке и запрограммировать подготовительный пример.

Критерии положительной оценки: Показать выполненную на компьютере работу. Ответить на вопросы преподавателя.

9.2.Теоретическое введение

Функция, не являющаяся членом класса, но имеющая доступ к его закрытым элементам, называется *дружественной.* Дружественная функция задается так же, как и обычная, не являющаяся элементом класса функция. В объявление класса, для которого функция будет дружественной, включается ее прототип, перед которым ставится ключевое слово *friend*. Поскольку дружественные функции не являются членами класса, в качестве аргументов им передаются объекты классов, для которых эти функции дружественны.

Пример 8.1

```
class myclass
{
        int n,d;
public:
        myclass(int i, int j) {n=i; d=j;}
        friend int factor(myclass ob); // дружественная функция
};
```

```
int factor(myclass ob)
{
if(!(ob.n % ob.d)) return 1;
else return 0;
}
int main()
{
myclass ob1(10, 2), ob2(13, 3);
if(factor(ob1)) cout<<"10 делится на 2 без остатка\n";
        else cout<<"10 не делится на 2 без остатка\n";
if(factor(ob2)) cout<<"13 делится на 3 без остатка\n";
        else cout<<"13 не делится на 3 без остатка\n";
return 0;
}
```

Дружественная функция - вызывается она не с помощью объекта, а как обычная функция. Хотя дружественная функция «знает» о закрытых элементах класса, для которого она дружественна, доступ к ним она может получить только через объект этого класса. Функция может быть дружественна более, чем к одному классу.

При работе с дружественными функциями используется предварительное объявление – способ сообщить компилятору имя класса без его фактического объявления. Это делается следующим образом: перед первым использованием имени класса пишется строка:

class имя_класса;

Функция может быть членом одного класса и быть дружественной другому.

Литература:

Воробейкина, И.В. Технологии и методы программирования, часть II: учебное пособие для студентов специальности 10.05.03 «Информационная безопасность автоматизированных систем» очной формы обучения / И.В. Воробейкина; БГАРФ ФГБОУ ВО «КГТУ». – Калининград: Изд-во БГАРФ, 2021.

Глава 4. Дружественные функции.

9.3.Задание к лабораторной работе

1. Создать классы *parall* и *rectang*; оба класса содержат закрытые переменные – длину и ширину геометрической фигуры. Оба класса имеют конструктор и функцию, которая вычисляет площадь параллелограмма в классе *parall* и площадь прямоугольника в классе *rectang*. Функция *pl_greater()* дружественна для обоих классов. Эта функция возвращает положительное число, если площадь параллелограмма больше площади прямоугольника; нуль, если их

площади одинаковы; отрицательное число, если площадь параллелограмма меньше площади прямоугольника.

2. Создать классы *tryan* и *rectan*; оба класса содержат закрытые переменные: *rectan* – длину и ширину прямоугольника, *tryan* – катеты прямоугольного треугольника. Оба класса имеют конструктор и функцию, которая вычисляет периметр прямоугольника в классе *rectan* и периметр треугольника в классе *tryan*. Функция *per_greater()* дружественна для обоих классов. Эта функция вычисляет кратность периметров друг другу. *Примечание:* оператор % работает с целочисленными переменными!

3. Выполнить задания 1, при условии, чтобы функция *pl_greater()* была членом одного класса и дружественной другому.

Вопросы:

1. Дайте определение дружественной функции.

2. Может ли функция быть одновременно членом одного класса и дружественной другому.

3. Какие аргументы передаются дружественной функции?

4. Зачем при работе с дружественными функциями используется предварительное описание класса?

9.4.Методические указания и порядок выполнения работы

Обычно дружественные функции полезны тогда, когда у двух (или более) разных классов имеется нечто общее, что необходимо сравнить.

Пример 8.2

Создать классы *car* и *truck*; оба класса содержат в закрытой переменной скорость; кроме того, у класса *car* есть закрытая переменная *количество пассажиров*, а у класса *truck* – закрытая переменная *грузоподъемность*. Оба класса имеют конструктор. Функция *sp_greater()* дружественна для классов *car* и *truck.* Эта функция возвращает положительное число, если объект *car* движется быстрее объекта *truck*; нуль, если их скорости одинаковы; отрицательное число, если скорость объекта *truck* больше, чем скорость объекта *car*.

```
class truck; //предварительно объявление
class car
{
        int pass; float speed;
public:
        car(int p, float s){pass=p; speed=s;}
        friend float sp_greater(car c, truck t);
};
```

```
class truck
{
        float weight, speed;
public:
        truck(float w, float s){weight=w; speed=s;}
        friend float sp_greater(car c, truck t);
};
/*Функция sp_greater() возвращает положительное число, если легковая машина быстрее грузовика, ноль – если скорости одинаковы, отрицательное число, если грузовик быстрее легковой машины.*/
float sp_greater(car c, truck t) {return c.speed – t.speed;}
int main()
{
float t;
car c1(6,55), c2(2,120);
truck t1(10000,55), t2(20000,72);
t= sp_greater(c1, t1);
if(t<0) cout<<"Грузовик быстрее\n";
        else if(t==0) cout<<"Скорости машин одинаковы\n";
                else cout<<"Легковая машина быстрее\n";
t= sp_greater(c2, t2);
if(t<0) cout<<"Грузовик быстрее\n";
        else if(t==0) cout<<"Скорости машин одинаковы\n";
                else cout<<"Легковая машина быстрее\n";
return 0;
}
```

Эта программа иллюстрирует важный момент синтаксиса *C++ – предварительное объявление* или *ссылка вперед.* Поскольку функция *sp_greater()* получает параметры от обоих классов *car* и *truck,* то логически невозможно объявить и тот, и другой класс перед включением функции *sp_greater()* в каждый из них. Поэтому используется предварительное объявление – инструкция *class truck;* после которой класс *truck* можно использовать в прототипе дружественной функции *sp_greater().*

Рассмотрим случай, когда функция может быть членом одного класса и быть дружественной другому.

Пример 8.3

```
class truck; //предварительно объявление
class car
{
        int pass; float speed;
```

```
public:
        car(int p, float s){pass=p; speed=s;}
        float sp_greater(truck t);
};
class truck
{
        float weight, speed;
public:
        truck(float w, float s){weight=w; speed=s;}
//новое использование оператора расширения области видимости ::
        friend float car::sp_greater(truck t);
};
/*Поскольку функция sp_greater() теперь член класса car, ей должен передаваться только объект truck.*/
float car::sp_greater(truck t) {return speed – t.speed;}
int main()
{
int t;
car c1(6,55), c2(2,120);
truck t1(10000,55), t2(20000,72);
t= c1.sp_greater(t1);
if(t<0) cout<<"Грузовик быстрее\n";
        else if(t==0) cout<<"Скорости машин одинаковы\n";
                else cout<<"Легковая машина быстрее\n";
t= c2.sp_greater(t2);
if(t<0) cout<<"Грузовик быстрее\n";
        else if(t==0) cout<<"Скорости машин одинаковы\n";
                else cout<<"Легковая машина быстрее\n";
return 0;
}
```

Обратите внимание на новое использование оператора расширения области видимости, который имеется в объявлении дружественной функции внутри класса *truck*. Здесь он информирует компилятор, что функция *sp_greater()* – член класса *car*.

9.5.Индивидуальное задание

Вариативность не предполагается. Освоить методы работы с дружественными функциями, создать программный код.

9.6. Требования к отчету и защите

Показать выполненную на компьютере работу. Знать ответы на сформулированные выше вопросы. Защита работы проводится во время занятий. После защиты работа помещается в ЭИОС.

10. ЛАБОРАТОРНАЯ РАБОТА № 9. МНОЖЕСТВЕННОЕ НАСЛЕДОВАНИЕ. ВИРТУАЛЬНЫЕ БАЗОВЫЕ КЛАССЫ

10.1. Общие сведения

Цель: Изучить множественное косвенное наследование, множественное прямое наследование, виртуальные базовые классы.

Материалы, оборудование, программное обеспечение: online-компилятор C++ (необходима бесперебойная работа сети ИНТЕРНЕТ) или Visual Studio C++

Условия допуска к выполнению: предварительно пройти инструктаж по ТБ. Показать конспект по теоретической подготовке и запрограммировать подготовительный пример.

Критерии положительной оценки: Показать выполненную на компьютере работу. Ответить на вопросы преподавателя.

10.2. Теоретическое введение

Если производный класс напрямую наследует несколько базовых классов, используется такое расширенное объявление:

class имя_производного_класса: сп_доступа имя_базового_класса1, сп_достула имя_базового_класса2, ..., сп_достула имя_базового-классаN

{// ... тело класса}

Здесь ***имя_базового_класса1 ... имя_базового_классаN*** – имена базовых классов, ***сп_доступа*** – спецификатор доступа, который может быть разным у разных базовых классов. Когда наследуется несколько базовых классов, конструкторы выполняются слева направо в том порядке, который задан в объявлении производного класса. Деструкторы выполняются в обратном порядке. Когда класс наследует несколько базовых классов, конструкторам которых необходимы аргументы, производный класс передает эти аргументы, используя расширенную форму объявления конструктора производного класса:

констр_произв_класса(список_арг): имя_базового_класса1(список_арг), имя_базового_класса2(список_арг),..., имя_базового_классаN(список_арг)

{// ... тело конструктора производного класса}

Здесь ***имя_базового_класса1 ... имя_базового_классаN*** – имена базовых классов. Если производный класс наследует иерархию классов, каждый произ-

водный класс должен передавать предшествующему в цепочке базовому классу все необходимые аргументы.

Здесь показана иерархия классов при косвенном наследовании:

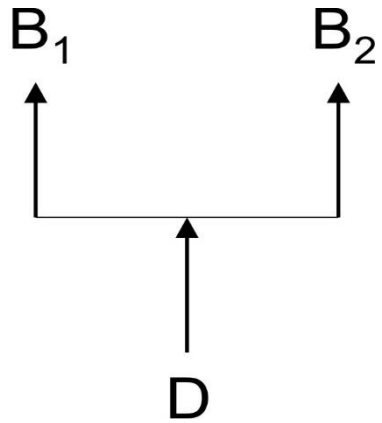
Пример 9.1

```
#include <iostream>
using namespace std;
class B1
{
        int a;
public:
        B1(int x) { a = x; }
        int geta() { return a; }
};
// Прямое наследование базового класса
class D1: public B1
{
        int b;
public:
        D1(int x, int y): B1(y) // передача переменной y классу B1
        {
        b = x;
        }
        int getb() { return b; }
};
// Прямое наследование производного класса
// и косвенное наследование базового класса
class D2: public D1
{
        int c;
public:
```

```
    D2(int x, int y, int z) : D1(y, z) // передача аргументов y и z классу D1
    {
    c = x;
    }
/* Поскольку базовые классы наследуются как открытые, класс D2 имеет доступ к открытым элементам классов B1 и D1 */
    void show()
    {
    cout << geta() << ' ' << getb() << ' ' << c << ' \n';
    }
};
int main()
{
D2 ob(1, 2, 3);
ob.show();
// функции geta() и getb () здесь тоже открыты
cout << ob.getaO << ' ' << ob.getb() << '\n';
return 0;
}
```

В этом примере класс *B1* является косвенным базовым классом для класса *D2*.

Иерархия классов при прямом наследовании выглядит таким образом:



В следующей программе производный класс прямо наследует два базовых класса.

Пример 9.2

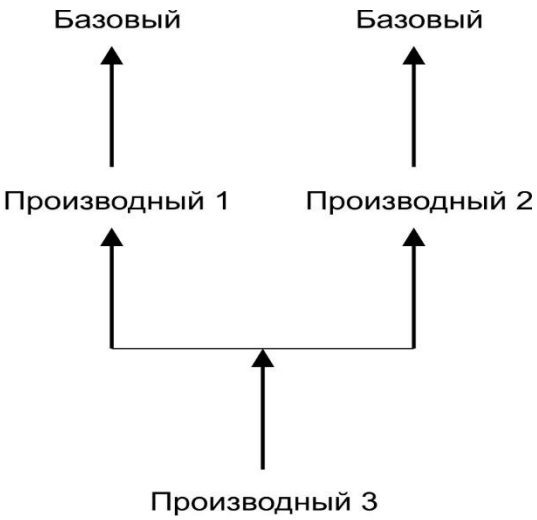
```
#include <iostream>
using namespace std;
// Создание первого базового класса
```

```
class B1
{
        int a;
public:
        B1(int x) { a = x; }
        int geta() { return a; }
};
// Создание второго базового класса
class B2
{
int b;
public:
B2(int x){b = x;}
int getb() { return b; }
};
// Прямое наследование двух базовых классов
class D: public B1, public B2
{
int c;
public:
// здесь переменные z и y напрямую передаются классам B1 и B2
D(int x, int y, int z): B1(z), B2(y){c = x;}
/* Поскольку базовые классы наследуются как открытые, класс D имеет доступ к открытым элементам классов B1 и B2 */
void show() {cout << geta() << ' ' << getb() << ' ' << c << '\n';}
};
int main()
{
D ob(1, 2, 3);
ob.show();
return 0;
}
```

В этой версии программы класс D передает аргументы по отдельности классам B1 и B2.

Виртуальные базовые классы

При многократном прямом наследовании производным классом одного и того же базового класса может возникнуть проблема. Рассмотрим следующую иерархию классов:



Здесь класс ***Базовый*** наследуется классами ***Производный1*** и ***Производный2***. Класс ***Производный3*** прямо наследует классы ***Производный1*** и ***Производный2***. Это значит, что класс ***Базовый*** наследуется классом ***Производный3*** дважды: первый раз – через класс ***Производный1***, а второй – через класс ***Производный2***. Это вызовет неоднозначность: поскольку в классе ***Производный3*** имеется две копии класса ***Базовый***, то будет ли ссылка на элемент класса ***Базовый*** относиться к классу ***Базовый***, наследуемому через класс ***Производный1***, или к классу ***Базовый***, наследуемому через класс ***Производный2***? Для преодоления этой неоднозначности в *C++* существует механизм, благодаря которому в класс ***Производный3*** будет включена только одна копия класса ***Базовый***. Класс, поддерживающий этот механизм, называется *виртуальным базовым классом.* В этом случае перед спецификатором доступа базового класса ставится ключевое слово *virtual.*

Пример 9.3

```
class base
{
public:
        int i;
};
//наследование класса base как виртуального
class derived1: virtual public base
{
public:
        int j;
};
//здесь класс base тоже наследуется как виртуальный
class derived2: virtual public base
{
```

```
public:
    int k;
};
    /*здесь класс derived3 наследует как класс derived1, так и класс derived2; однако в классе derived3 создается только одна копия класса base
*/
class derived3: public derived1, public derived2
{
public:
    int product() {return i*j*k;}
    };
    int main()
    {
    derived3 ob;
    ob.i=10;
    ob.j=3;
    ob.k=5;
    cout<<"Результат: "<<ob.product()<<endl;
    return 0;
    }
```

Если бы классы *derived1* и *derived2* наследовали класс *base* не как виртуальный, тогда инструкция *ob.i=10* вызывала бы неоднозначность, и при компиляции возникла бы ошибка.

Литература:

Воробейкина, И.В. Технологии и методы программирования, часть II: учебное пособие для студентов специальности 10.05.03 «Информационная безопасность автоматизированных систем» очной формы обучения / И.В. Воробейкина; БГАРФ ФГБОУ ВО «КГТУ». – Калининград: Изд-во БГАРФ, 2021.

Глава 5. Наследование.

10.3. Задание к лабораторной работе

1. Создать производный класс *familia,* содержащий закрытый член класса для хранения фамилии. Наследование косвенное: классу *familia* предшествует класс *shtat,* который содержит должность и зарплату, а классу *shtat* предшествует класс *sotrudnik,* содержащий год рождения и пол. Все три класса содержат конструкторы. Класс *familia* содержит метод, выводящий поля всех классов на экран.

2. Создать производный класс *bilet*, содержащий закрытый член класса для хранения стоимости билета. Базовые классы: класс *bus* содержит количество посадочных мест и количество колес; класс *gorod* – конечный пункт и

время нахождения в пути. Наследование обоих классов прямое! Класс *bilet* содержит метод, выводящий поля всех классов на экран. Конструктор есть только у производного класса.

3. Есть два земельных участка: один – в виде прямоугольника, другой – в виде треугольника. Длина и ширина задана в базовом классе *base*, который наследуют два класса – *rect* и *tryan*. В классе *rect* вычисляется периметр прямоугольника, в классе *tryan* – периметр треугольника. Классы *rect* и *tryan* наследует класс *perimeter*, который вычисляет, сколько метров забора надо купить, чтобы огородить оба участка. Конструктор имеется только у класса *perimeter*.

10.4. Методические указания и порядок выполнения работы

В следующей программе показан порядок, в котором вызываются конструкторы и деструкторы, когда производный класс прямо наследует несколько базовых классов:

Пример 9.4

```
#include <iostream>
using namespace std;
class B1
{
public:
B1 () { cout << "Работа конструктора класса Bl\n"; }
~B1() { cout << "Работа деструктора класса Bl\n"; }
};
class B2
{
int b;
public:
B2 () { cout << "Работа конструктора класса B2\n"; }
~B2() { cout << "Работа деструктора класса B2\n"; }
};
// Наследование двух базовых классов
class D: public B1, public B2
{
public:
D() { cout << "Работа конструктора класса D\n"; }
~D() { cout << "Работа деструктора класса D\n"; }
};
int main()
{
D ob;
```

```
return 0;
}
```

Эта программа выводит на экран следующее:

```
Работа конструктора класса B1
Работа конструктора класса B2
Работа конструктора класса D
Работа деструктора класса D
Работа деструктора класса B2
Работа деструктора класса B1
```

10.5. Индивидуальное задание

Вариативность не предполагается. Освоить наследование в классах, создать программный код.

10.6. Требования к отчету и защите

Показать выполненную на компьютере работу. Знать ответы на сформулированные выше вопросы. Защита работы проводится во время занятий. После защиты работа помещается в ЭИОС.

11. ЛАБОРАТОРНАЯ РАБОТА № 10. ВИРТУАЛЬНЫЕ ФУНКЦИИ. ЧИСТЫЕ ВИРТУАЛЬНЫЕ ФУНКЦИИ И АБСТРАКТНЫЕ КЛАССЫ

11.1. Общие сведения

Цель: Познакомиться с виртуальными функциями, получить представление об абстрактных классах.

Материалы, оборудование, программное обеспечение: online-компилятор C++ (необходима бесперебойная работа сети ИНТЕРНЕТ) или Visual Studio C++.

Условия допуска к выполнению: предварительно пройти инструктаж по ТБ. Показать конспект по теоретической подготовке и запрограммировать подготовительный пример.

Критерии положительной оценки: Показать выполненную на компьютере работу. Ответить на вопросы преподавателя.

11.2. Теоретическое введение

Перед объявлением виртуальной функции ставится ключевое слово *virtual*. Если класс, содержащий виртуальную функцию, наследуется, то в производном классе виртуальная функция переопределяется.

При переопределении виртуальной функции в производном классе ключевое слово *virtual* не требуется.

Пример 10.1

```
#include <iostream>
using namespace std;
class base
{
        public:
        int i;
        base(int x) {i = x; }
        virtual void func()
        {cout << "Выполнение функции func() базового класса: " << i <<
'\n';}
};
        class derived1: public base
        {
        public:
        derived1(int x): base(x) { }
        void func(){cout << "Выполнение функции func() класса derived1: "
<< i*i << '\n';}
};
        class derived2: public base
        {
        public:
        derived2(int x): base(x) { }
        void func() {cout << "Выполнение функции func() класса derived2: "
<< i + i << '\n';}
};
        int main( )
        {
        base *p;
        base ob(10);
        derived1 d_obl(10);
        derived2 d_ob2(10);
        p = &ob;
        p–>func(); // функция func ( ) класса base
        p = &d_ob1;
        p–>func(); // функция func() производного класса derived1
        p = &d_ob2;
        p–>func(); // функция func() производного класса derived2
```

```
return 0;
}
```

После выполнения программы на экране появится следующее:

Выполнение функции func() базового класса: 10
Выполнение функции func() класса derived1: 100
Выполнение функции func() класса derived2:

В рассмотренном примере создаются три класса. В классе *base* определяется виртуальная функция *func()*. Затем этот класс наследуется двумя производными классами: *derived1* и *derived2*. Каждый из этих классов переопределяет функцию *func()* по-своему. Внутри функции *main()* указатель базового класса *p* поочередно ссылается на объекты типа *base, derived1* и *derived2*.

Чистые виртуальные функции и абстрактные классы

Чистые виртуальные функции не определяются в базовом классе. Туда включаются только прототипы этих функций:

virtual тип имя_функции (список_параметров) = 0;

Ключевой частью этого объявления является приравнивание функции нулю. Это сообщает компилятору, что в базовом классе не существует тела функции. Если функция задается как чистая виртуальная, это предполагает, что она обязательно должна подменяться в каждом производном классе. Если этого нет, то при компиляции возникнет ошибка.

Если класс содержит хотя бы одну чистую виртуальную функцию, то о нем говорят как об *абстрактном классе*.

Если производный класс используется в качестве базового для другого производного класса, то виртуальная функция может подменяться в последнем производном классе (так же, как и в первом производном классе). Например, если базовый класс *B* содержит виртуальную функцию *f()* и класс *D1* наследует класс *B*, а класс *D2* наследует класс *D1,* тогда функция *f()* может подменяться как в классе *D1,* так и в классе *D2*.

Далее представлена усовершенствованная версия программы, показанной в примере 10.4. В этой версии программы в базовом классе *area* функция *getarea()* объявляется как чистая виртуальная функция.

Пример 10.2

```
// Создание абстрактного класса
#include <iostream>
using namespace std;
class area
{
double dim1, dim2; // размеры фигуры
public:
    void setarea(double d1, double d2){dim1 = d1; dim2 = d2;}
```

```
void getdim(double &d1, double &d2){d1 = dim1; d2 = dim2;}
virtual double getarea() =0; // чистая виртуальная функция
};
class rectangle: public area
{
public:
double getarea() {double d1, d2;getdim(d1, d2); return d1*d2;}
};
class triangle: public area
{
public:
double getarea()
{
double d1, d2;
getdim(d1, d2);
return 0.5*d1*d2;
}
};
int main()
{
area *p;
rectangle r;
triangle t;
r.setarea(3.3, 4.5);
t.setarea(4.0, 5.0);
p = &r;
cout << "Площадь прямоугольника: " << p–>getarea() << '\n';
p = &t;
cout << "Площадь треугольника: " << p–>getarea() << '\n';
return 0;
}
```

Теперь то, что функция *getarea()* является чистой виртуальной, гарантирует ее обязательную подмену в каждом производном классе.

Литература:

Воробейкина, И.В. Технологии и методы программирования, часть II: учебное пособие для студентов специальности 10.05.03 «Информационная безопасность автоматизированных систем» очной формы обучения / И.В. Воробейкина; БГАРФ ФГБОУ ВО «КГТУ». – Калининград: Изд-во БГАРФ, 2021.

Глава 6. Виртуальные функции.

11.3. Задание к лабораторной работе

1. Создать базовый класс *dimen*, в котором хранятся катеты прямоугольного треугольника. В классе *dimen* также объявляется виртуальная функция *pl_gip(),* которая, при ее подмене в производном классе *pl* возвращает площадь треугольника, а при ее подмене в производном классе *gip* возвращает его гипотенузу. Во всех трех классах создать конструкторы. К виртуальной функции обратиться через указатель.

2. Напишите программу, в которой базовый класс *dist* используется для хранения в переменной типа *double* расстояния между двумя точками. В классе *dist* создайте виртуальную функцию *trav_time(),* которая выводит на экран время, необходимое для прохождения этого расстояния с учетом того, что расстояние задано в милях, а скорость равна *60* миль в час. В производном классе *metric* переопределите функцию *trav_time()* так, чтобы она выводила на экран время, необходимое для прохождения этого расстояния, считая теперь, что расстояние задано в километрах, а скорость равна *100* километров в час.

3. Создать базовый класс *dimen,* в котором хранятся катеты прямоугольного треугольника. В классе *dimen* объявляется **чистая виртуальная функция** *pl_gip(),* которая, при ее подмене в производном классе *pl* возвращает площадь треугольника, а при ее подмене в производном классе *gip* возвращает его гипотенузу. **Во всех трех классах создать конструкторы! К виртуальной функции обратиться через указатель.**

11.4. Методические указания и порядок выполнения работы

Виртуальные функции имеют иерархический порядок наследования. Если виртуальная функция не подменяется в производном классе, то используется версия функции, определенная в базовом классе.

Ниже приведена измененная версия предыдущей программы:

Пример 10.3

```
// Иерархический порядок виртуальных функций
#include <iostream>
using namespace std;
class base
{
public:
        int i;
        base(int x) { i = x; }
        virtual void func(){cout << "Выполнение функции func() базового
класса: "<< i << '\n';}
};
```

```
class derived1:public base
{
public:
        derived1(int x): base(x) { }
        void func(){cout << "Выполнение функции func() класса derived1: "
<< i*i << '\n';}
        };
        class derived2: public base
        {
        public:
        derived2(int x): base(x) { }
        // в классе derived2 функция func() не подменяется
};
        int main()
        {
        base *p;
        base ob(10);
        derived1 d_ob1(10);
        derived2 d_ob2(10);
        p = &ob;
        p–>func(); // функция func() базового класса
        p = &d_ob1;
        p–>func(); // функция func() производного класса derived1
        p = &d__ob2;
        p–>func(); // функция func() базового класса
        return 0;
        }
```

После выполнения программы на экран выводится следующее:

Выполнение функции func() базового класса: 10
Выполнение функции func() класса derived1: 100
Выполнение функции func() базового класса: 10

В этой программе в классе *derived2* функция *func()* не подменяется. Когда указателю *p* присваивается адрес объекта *d_ob2* и вызывается функция *func(),* используется версия функции из класса *base,* поскольку она следующая в иерархии классов. Если виртуальная функция не переопределена в производном классе, используется ее версия из базового класса.

В следующей программе создается исходный базовый класс *area*, в котором сохраняются две размерности фигуры. В нем также объявляется виртуальная функция *getarea(),* которая, при ее подмене в производном классе, возвращает площадь фигуры, вид которой задается в производном классе. В этом слу-

чае определение функции *getarea()* внутри базового класса задает интерфейс. Конкретная реализация остается тем классам, которые наследуют класс *area.* В этом примере рассчитывается площадь треугольника и прямоугольника.

Пример 10.4

```
// Использование виртуальной функции для определения интерфейса
#include <iostream>
using namespace std;
class area
{
double dim1, dim2; // размеры фигуры
public:
        void setarea(double d1, double d2){dim1 = d1; dim2 = d2;}
        void getdim(double &d1, double &d2){d1 = dim1; d2 = dim2;}
        virtual double getarea()
                {
                cout << "Вам необходимо подменить эту функцию\n";
                return 0.0;
                }
};
class rectangle: public area
{
public:
        double getarea()
        {
        double d1, d2;
        getdim(d1, d2);
        return d1*d2;
        }
        };
        class triangle: public area
        {
        public:
        double getarea()
        {
        double d1, d2;
        getdim(d1,d2);
        return 0.5*d1*d2;
        }
};
int main()
```

```
{
area *p;
rectangle r;
triangle t;
r.setarea(3.3, 4.5);
t.setarea(4.0, 5.0);
p = &r;
cout << "Площадь прямоугольника: " << p–>getarea() << '\n';
p = &t;
cout << "Площадь треугольника: " << p–>getarea() << '\n';
return 0;
}
```

Обратите внимание, что определение *getarea()* внутри класса *area* является только «заглушкой» и в действительности не выполняет никаких действий. Поскольку класс *area* не связан с фигурой конкретного типа, то нет значимого определения, которое можно дать функции *getarea()* внутри класса *area*. При этом, для того чтобы нести полезную нагрузку, функция *getarea()* должна быть переопределена в производном классе.

В следующей программе показано, как при наследовании сохраняется виртуальная природа функции.

Пример 10.5

```
/* Виртуальная функция при наследовании сохраняет свою виртуальную природу*/
#include <iostream>
using namespace std;
class base
{
public:
    virtual void func(){cout << "Выполнение функции func ( ) базового класса \n";}
};
class derived1: public base
{
public:
    void func(){cout << "Выполнение функции func() класса derived1\n";}
};
// Класс derived1 наследуется классом derived2
class derived2: public derived1
{
public:
```

```
    void func(){cout << " Выполнение функции func() класса de-
rived2\n";}
};
int main()
{
base *p;
base ob;
derived1 d_ob1;
derived2 d ob2;
p = &ob;
p–>func(); // функция func() базового класса
p = &d_ob1;
p–>func(); // функция func() производного класса derived1
p = &d_ob2;
p–>func(); // функция func() производного класса derived2
return 0;
}
```

В этой программе виртуальная функция *func()* сначала наследуется классом *derived1*, в котором она подменяется. Далее класс *derived1* наследуется классом *derived2*. В классе *derived2* функция *func()* снова подменяется. Поскольку виртуальные функции являются иерархическими, то, если бы в классе *derived2* функция *func()* не подменялась, при доступе к объекту *d_ob2* использовалась бы переопределенная в классе *derived1* версия функции *func()*. Если бы функция *func()* не подменялась ни в классе *derived1*, ни в классе *derived2*, то все ссылки на функцию *func()* относились бы к ее определению в классе *base*.

11.5. Индивидуальное задание

Вариативность не предполагается. Освоить виртуальные функции, абстрактные классы, создать программный код.

11.6. Требования к отчету и защите

Показать выполненную на компьютере работу. Знать ответы на сформулированные выше вопросы. Защита работы проводится во время занятий. После защиты работа помещается в ЭИОС.

12. ЛАБОРАТОРНАЯ РАБОТА № 11. МЕХАНИЗМ РАБОТЫ ФУНКЦИЙ-ШАБЛОНОВ. РОДОВЫЕ ФУНКЦИИ И ПЕРЕГРУЗКА ФУНКЦИЙ

12.1. Общие сведения

Цель: Изучить механизм работы функций-шаблонов.

Материалы, оборудование, программное обеспечение: online-компилятор C++ (необходима бесперебойная работа сети ИНТЕРНЕТ) или Visual Studio C++

Условия допуска к выполнению: предварительно пройти инструктаж по ТБ. Показать конспект по теоретической подготовке и запрограммировать подготовительный пример.

Критерии положительной оценки: Показать выполненную на компьютере работу. Ответить на вопросы преподавателя.

12.2. Теоретическое введение

Родовые функции

Форма определения функции-шаблона:

```
template <class Фтип> возвр_значение имя_функции(сп_параметров)
{
тело функции
}
```

Здесь вместо *Фтип* указывается тип используемых функцией данных.

Вместо ключевого слова class можно указывать ключевое слово typename.

Пример 11.1

В следующей программе создается родовая функция, которая меняет местами значения двух переменных, передаваемых ей в качестве параметров. Поскольку в своей основе процесс обмена двух значений не зависит от типа переменных, этот процесс реализуется с помощью родовой функции.

```
// Пример родовой функции или шаблона
#include <iostream>
using namespace std;
//Это функция-шаблон
template <class X> void swapargs(X &a, X &b)
{
X temp;
temp=a;
a=b;
b=temp;
}
int main()
{
int i = 10, j = 20;
float x = 10.1, y = 23.3;
cout << "Исходные значения i, j равны: " << i << ' ' << j << endl;
cout << "Исходные значения x, y равны: " << x << ' ' << y << endl;
```

```
swapargs(i, j); // обмен целых
swapargs(x, y); // обмен действительных
cout << "Новые значения i, j равны: " << i << ‘ ‘ << j << endl;
cout << "Новые значения x, y равны: " << x << ‘ ‘ << y << endl;
return 0;
}
```

Ключевое слово *template* используется для определения родовой функции. Строка

```
template <class X> void swapargs(X &a, X &b)
```

сообщает компилятору две вещи: во-первых, создается шаблон, и, во-вторых, начинается определение родовой функции. Здесь *X* – это родовой тип данных, используемый в качестве фиктивного имени. После строки с ключевым словом *template* функция *swapargs()* объявляется с именем *X* в качестве типа данных обмениваемых значений. В функции *main()* функция *swapargs()* вызывается с двумя разными типами данных: целыми и действительными. Поскольку функция *swapargs()* – это родовая функция, компилятор автоматически создает две ее версии: одну – для обмена целых значений, другую – для обмена действительных значений.

Другое название родовой функции – функция-шаблон (template function). Когда компилятор создает конкретную версию этой функции, говорят, что он создал порожденную функцию (generated function). Процесс генерации порожденной функции называют созданием экземпляра (instantiating) функции. Другими словами, порожденная функция – это конкретный экземпляр функции-шаблона.

Ключевое слово *template* в определении родовой функции не обязательно должно быть в той же строке, что и имя функции. Например, ниже приведен еще один обычный формат определения функции *swapargs():*

```
template <class X>
void swapargs(X &a, X &b)
{
X temp;
temp = a;
a = b;
b = temp;
}
```

Никаких других инструкций между инструкцией template и началом определения родовой функции быть не может.

С помощью инструкции *template* можно определить более одного родового типа данных, отделяя их друг от друга запятыми. В следующей программе создается родовая функция, в которой имеются два родовых типа данных:

```
#include <iostream>
using namespace std;
template <class type1, class type2>
void myfunc(type1 x, type2 y)
{
cout << x << ' ' << y << endl;
}
int main()
{
myfunc(10, "hi");
myfunc(0.23, 10L);
return 0;
}
```

В данном примере при генерации конкретных экземпляров функции *myfunc()*, фиктивные имена типов *type1* и *type2* заменяются компилятором на типы данных *int* и *string* или *double* и *long* соответственно.

Литература:

Воробейкина, И.В. Технологии и методы программирования, часть II: учебное пособие для студентов специальности 10.05.03 «Информационная безопасность автоматизированных систем» очной формы обучения / И.В. Воробейкина; БГАРФ ФГБОУ ВО «КГТУ». – Калининград: Изд-во БГАРФ, 2021.

Глава 7. Шаблоны и обработка исключительных ситуаций

12.3. Задание к лабораторной работе

1. Напишите родовую функцию *min()*, возвращающую меньший из двух своих аргументов. Например, версия функции *min(3, 4)* должна возвратить *3*, а версия *min('c', 'a') – a*.

2. Создать родовую функцию *sum*, возвращающую сумму элементов массива. Создать родовую функцию *mas_otr*, выводящую на экран только отрицательные элементы массива.

3. Создайте родовую функцию, возвращающую значение элемента, который чаще всего встречается в массиве.

12.4. Методические указания и порядок выполнения работы

Родовые функции похожи на перегруженные функции за исключением того, что они более ограничены по своим возможностям. При перегрузке функции внутри ее тела можно выполнять разные действия, а родовая функция должна выполнять одни и те же базовые действия для всех своих версий. Следующие перегруженные функции нельзя заменить на родовую функцию, поскольку они делают не одно и то же.

```
void outdata(int i){cout << i;}
void outdata(double d)
        {
        cout << setprecision(10) << setfill ('#') ;
        cout << d;
        cout << setprecision(6) << setfill (' ');
        }
```

Несмотря на то, что функция-шаблон при необходимости перегружается сама, ее также можно перегрузить явно. Если вы сами перегружаете родовую функцию, то перегруженная функция подменяет (или «скрывает») родовую функцию, которую бы создал компилятор для этой конкретной версии. Рассмотрим вариант примера 11.1:

Пример 11.2

```
// Подмена функции-шаблона
#include <iostream>
using namespace std;
template <class X> void swapargs(X &a, X &b)
{
X temp;
temp = a;
a = b;
b = temp;
}
// Здесь переопределяется родовая версия функции swapargs()
void swapargs(int a, int b)
{
cout << "это печатается внутри функции swapargs(int, int)\n";
}
int main()
{
int i = 10, j = 20;
float x = 10.1, y = 23.3;
cout << "Исходные значения i, j = " << i << ' ' << j << endl;
cout << "Исходные значения x, y = " << x << ' ' << y << endl;
swapargs(i, j); // вызов явно перегруженной функции swapargs(j
swapargs(x, y); // обмен действительными числами
cout << "Новые значения i, j равны: " << i << ' ' << j << endl;
cout << "Новые значения x, y равны: " << x << ' ' << y << endl;
return 0;
}
```

Здесь при вызове функции *swapargs(i, j)* вызывается определенная в программе явно перегруженная версия функции *swapargs().* Таким образом, компилятор не генерирует этой версии родовой функции *swapargs(),* поскольку родовая функция подменяется явно перегруженной функцией.

12.5. Индивидуальное задание

Вариативность не предполагается. Освоить механизм работы функций-шаблонов.

12.6. Требования к отчету и защите

Показать выполненную на компьютере работу. Знать ответы на сформулированные выше вопросы. Защита работы проводится во время занятий. После защиты работа помещается в ЭИОС.

13. ЛАБОРАТОРНАЯ РАБОТА № 12. ВОЗБУЖДЕНИЕ И ОБРАБОТКА ИСКЛЮЧИТЕЛЬНЫХ СИТУАЦИЙ. БЛОКИ *try – catch* И ОПЕРАТОР *throw*

13.1. Общие сведения

Цель: Научиться обрабатывать исключительные ситуации.

Материалы, оборудование, программное обеспечение: online-компилятор C++ (необходима бесперебойная работа сети ИНТЕРНЕТ) или Visual Studio C++

Условия допуска к выполнению: предварительно пройти инструктаж по ТБ. Показать конспект по теоретической подготовке и запрограммировать подготовительный пример.

Критерии положительной оценки: Показать выполненную на компьютере работу. Ответить на вопросы преподавателя.

13.2. Теоретическое введение

Обработка исключительных ситуаций

Обработка исключительных ситуаций в *C++* организуется с помощью трех ключевых слов: *try, catch* и *throw*. Инструкции программы, во время выполнения которых обеспечивается обработка исключительных ситуаций, располагаются в блоке *try*. Если исключительная ситуация (т. е. ошибка) имеет место внутри блока *try,* она возбуждается (ключевое слово *throw*), перехватывается (ключевое слово *catch*) и обрабатывается. Функции, которые вызываются из блока *try,* также могут возбуждать исключительную ситуацию. Любая исключительная ситуация должна перехватываться инструкцией *catch*, которая располагается непосредственно за блоком *try*, возбуждающим исключительную ситуацию. Форма инструкций *try* и *catch*:

```
try
        {
        // блок возбуждения исключительной ситуации
        }
catch (type1 arg)
        {
        // блок перехвата исключительной ситуации
        }
catch (type2 arg)
        {
        // блок перехвата исключительной ситуации
        }
.
.
.
catch (typeN arg)
        {
        // блок перехвата исключительной ситуации
        }
```

Блок *try* содержит ту часть программы, в которой отслеживаются ошибки.

После того, как исключительная ситуация возбуждена, она перехватывается и обрабатывается соответствующей этой конкретной исключительной ситуации инструкцией *catch*. С блоком *try* может быть связано более одной инструкции *catch*. То, какая именно инструкция *catch* используется, зависит от типа исключительной ситуации. То есть, если тип данных, указанный в инструкции *catch*, соответствует типу исключительной ситуации, выполняется данная инструкция *catch,* а все оставшиеся инструкции блока *try* игнорируются. Если исключительная ситуация перехвачена, аргумент *arg* получает ее значение. Если не нужен доступ к самой исключительной ситуации, можно в инструкции *catch* указать только ее тип *type*, а аргумент *arg* указывать не обязательно.

Форма инструкции *throw*:

throw исключительная_ситуация;

Инструкция *throw* должна выполняться либо внутри блока *try*, либо в любой функции, которую этот блок вызывает (прямо или косвенно).

В следующем примере показано, как в *C++* функционирует система обработки исключительных ситуаций.

Пример 12.1

```
// Пример обработки исключительной ситуации
#include <iostream>
```

```
using namespace std;
int main()
{
cout << "начало\n";
try
        { // начало блока try
        cout << "Внутри блока try\n";
        throw 10; // возбуждение ошибки
        cout << "Эта инструкция выполнена не будет";
        }
catch (int i)
        { // перехват ошибки
        cout << "перехвачена ошибка номер: " << i << "\n";
        }
cout << "конец";
return 0;
}
```

Результат работы программы:

```
начало
Внутри блока try
Перехвачена ошибка номер: 10
конец
```

В этой программе имеется блок *try*, содержащий три инструкции, и инструкция *catch (int i)*, обрабатывающая исключительную ситуацию целого типа. Внутри блока *try* будут выполнены только две из трех инструкций – инструкции *cout* и *throw*. После того как исключительная ситуация возбуждена, управление передается выражению *catch*, и выполнение блока *try* завершается. Таким образом, инструкция *catch* вызывается не явно, управление выполнением программы просто передается этой инструкции. Следовательно, следующая за инструкцией *throw* инструкция *cout* не будет выполнена никогда.

Литература:

Воробейкина, И.В. Технологии и методы программирования, часть II: учебное пособие для студентов специальности 10.05.03 «Информационная безопасность автоматизированных систем» очной формы обучения / И.В. Воробейкина; БГАРФ ФГБОУ ВО «КГТУ». – Калининград: Изд-во БГАРФ, 2021.

Глава 7. Шаблоны и обработка исключительных ситуаций

13.3. Задание к лабораторной работе

1. Написать программу *Калькулятор*, обработать исключительные ситуации ввода неправильной арифметической операции, деления на *0* и извлечения квадратного корня из отрицательного числа.

2. Ввести фамилию, год рождения и пол студента. Проверить правильность ввода пола и года рождения. Обработать исключительные ситуации.

13.4. Методические указания и порядок выполнения работы

Тип исключительной ситуации должен соответствовать типу, заданному в инструкции *catch*. В предыдущем примере, если изменить тип данных в инструкции *catch* на *double*, то исключительная ситуация не будет перехвачена и будет иметь место ненормальное завершение программы. Это продемонстрировано в следующем фрагменте:

Пример 12.2

```
// Этот пример работать не будет
#include <iostream>
using namespace std;
int main ()
{
cout << "начало\n";
try
        {
        cout << "Внутри блока try\n";
        throw 10; // возбуждение ошибки
        }
        catch (double i) // Эта инструкция не будет работать
                // с исключительной ситуацией целого типа
        {
        cout << "перехвачена ошибка номер: " << i << "\n";
        }
        cout << "конец";
        return 0;
        }
```

Так как исключительная ситуация целого типа не будет перехвачена инструкцией *catch* типа *double*, на экран программа выведет следующее:

```
начало
Внутри блока try
Abnormal program termination
```

Исключительная ситуация может быть возбуждена не входящей в блок *try* инструкцией, если сама эта инструкция входит в функцию, которая вызывается из блока *try*. Ниже представлена совершенно правильная программа.

Пример 12.3

```
/* Возбуждение исключительной ситуации из функции, находящейся вне блока try */
#include <iostream>
using namespace std;
void Xtest(int test)
{
cout <<"Внутри функции Xtest, test = " << test << "\n";
if(test) throw test;
}
int main()
{
cout << "начало\n";
try
{
cout << "Внутри блока try\n";
Xtest (0);
Xtest (1);
Xtest (2);
}
catch (int i)
{ // перехват ошибки
cout << "перехвачена ошибка номер: " << i << "\n";
}
cout << "конец";
return 0;
}
```

На экран программа выводит следующее:

начало
Внутри блока try
Внутри функции Xtest, test равно: 0
Внутри функции Xtest, test равно: 1
Перехвачена ошибка номер: 1
конец

В следующей программе перехватываются две исключительных ситуации – одна для целых чисел и одна для строки.

Пример 12.4

```
#include <iostream>
using namespace std;
// Можно перехватывать разные типы исключительных ситуаций
void Xhandler(int test)
	{
	try
		{
		if(test) throw test;
		else throw "Значение равно нулю";
		}
		catch(int i)
		{
		cout << "Перехвачена ошибка номер: " << i << '\n';
		}
		catch(char *str)
		{
		cout << "Перехвачена строка: " << str << '\n';
		}
	}
int main()
{
cout << "начало\n";
Xhandler(1);
Xhandler(2);
Xhandler(0);
Xhandler(3) ;
cout << "конец";
return 0;
}
```

На экран программа выводит следующее:

начало
Перехвачена ошибка номер: 1
Перехвачена ошибка номер: 2
Перехвачена строка: Значение равно нулю
Перехвачена ошибка номер: 3
конец

Каждая инструкция *catch* перехватывает только исключительные ситуации соответствующего ей типа. Обычно выражения инструкций *catch* проверяются в том порядке, в котором они появляются в программе. Выполняется

только та инструкция, которая совпадает по типу данных с исключительной ситуацией. Все остальные блоки *catch* игнорируются.

В некоторых случаях необходимо настроить систему так, чтобы перехватывать все исключительные ситуации, независимо от их типа. Для этого используется следующая форма инструкции *catch:*

```
catch (...)
{
// обработка всех исключительных ситуаций
}
```

Здесь многоточие соответствует любому типу данных.

В следующей программе иллюстрируется использование инструкции *catch(...):*

Пример 12.5

```
#include <iostream>
using namespace std;
void Xhandler(int test)
{
try
        {
        // возбуждение исключительной ситуации типа int
        if(test==0) throw test;
        // возбуждение исключительной ситуации типа char
        if(test==1) throw 'a';
        // возбуждение исключительной ситуации типа double
        if(test==2) throw 123.23;
        }
        catch(...)
        {
        // перехват исключительных ситуаций всех типов
        cout << "Перехвачена ошибка!\n";
        }
        }
        int main()
        {
        cout << "начало\n";
        Xhandler(0);
        Xhandler(1);
        Xhandler(2);
        cout << "конец";
        return 0;
```

}

На экран программа выводит следующее:

начало

Перехвачена ошибка!

Перехвачена ошибка!

Перехвачена ошибка!

конец

Во всех трех случаях возбуждения исключительной ситуации в инструкции *throw*, она перехватывается с помощью единственной инструкции *catch*.

Инструкцию *catch(...)* удобно использовать в качестве последней в группе инструкций *catch*. В этом случае *catch(...)* по умолчанию становится инструкцией, которая *перехватывает все*.

13.5. Индивидуальное задание

Вариативность не предполагается. Изучить обработку исключительных ситуаций.

13.6. Требования к отчету и защите

Показать выполненную на компьютере работу. Знать ответы на сформулированные выше вопросы. Защита работы проводится во время занятий. После защиты работа помещается в ЭИОС.

14. ЛАБОРАТОРНАЯ РАБОТА № 13. ВЫЧИСЛЕНИЕ И ПРОГРАММИРОВАНИЕ ПОРЯДКА ТОЧКИ НА ЭЛЛИПТИЧЕСКОЙ КРИВОЙ

14.1. Общие сведения

Цель: Изучить шифрование ECDH, познакомиться с правилами вычисления порядка точки на эллиптической кривой.

Материалы, оборудование, программное обеспечение: online-компилятор C++ (необходима бесперебойная работа сети ИНТЕРНЕТ) или Visual Studio C++

Условия допуска к выполнению: предварительно пройти инструктаж по ТБ. Показать конспект по теоретической подготовке и запрограммировать подготовительный пример.

Критерии положительной оценки: Показать выполненную на компьютере работу. Ответить на вопросы преподавателя.

14.2. Теоретическое введение

Законы сложения точек эллиптической кривой

Суммой двух точек *P* и *Q* называется точка *R=P+Q,* обратная третьей точке пересечения эллиптической кривой и прямой, проходящей через точки *P* и *Q*.

Если суммируемые точки *P* и *Q* совпадают, то *P+Q=P+P=R*, что равносильно удвоению точки *2P=R.*

Формулы сложения и удвоения точек эллиптической кривой справедливы для всех полей, кроме полей характеристик *2* и *3*.

Таблица формул для операций с точками эллиптической кривой:

Операция	*Поле характеристики p (p≠2 и p≠3)*
Сложение точек $P \neq \pm Q$ $P(x_1,y_1)+Q(x_2,y_2)=R(x_3,y_3)$	$\lambda = \frac{y_2 - y_1}{x_2 - x_1}(mod\ p)$ $x_3 = \lambda^2 - x_1 - x_2\ (mod\ p)$ $y_3 = \lambda(x_1 - x_3) - y_1(mod\ p)$
Удвоение точки $R(x_3,y_3)=2\ P(x_1,y_1)$	$\lambda = \frac{3{x_1}^2 + a}{2y_1}(mod\ p)$ $x_3 = \lambda^2 - 2x_1\ (mod\ p)$ $y_3 = \lambda(x_1 - x_3) - y_1(mod\ p)$
$O+O=O$ $P(x,y)+O= P(x,y)$ $P(x,y)+ P(x,-y)=O$	

Пример 13.1. Найти все точки эллиптической кривой $E_7(2,6)$.

Решение.

$E_7(2,6)$ – это кривая $y^2=x^3+2x+6(mod\ 7)$. Вычислим $x^3+2x+6(mod\ 7)$ и $y^2\ (mod\ 7)$ для $x, y=1,2,\ldots,6$.

x	1	2	3	4	5	6
$x^3+2x+6(mod\ 7)$	2	4	4	1	1	3
y	1	2	3	4	5	6
$y^2\ (mod\ 7)$	1	4	2	2	4	1

Группа $E_7(2,6)$ состоит из точек (x,y), при которых $y^2(mod7)=x^3+2x+6(mod7)$. Это точки $(1,3)$, $(1,4)$, $(2,2)$, $(2,5)$, $(3,2)$, $(3,5)$, $(4,1)$, $(4,6)$, $(5,1)$, $(5,6)$ и O. На графике видна симметрия относительно прямой $y=p/2=3,5$.

Пример 13.2. Вычислить в группе $E_{11}(1,6)$: *а) (8,3)+(3,6); б) 2(1,8).*

Решение.

а) $(x_1,y_1)=(8,3)$; $(x_2,y_2)=(3,6)$.

$$\lambda = \frac{y_2 - y_1}{x_2 - x_1} mod\ p = \frac{6-3}{3-8} mod 11 = \frac{3}{-5} mod 11 = -3 \cdot 5^{-1} mod 11$$

$$= -3 \cdot 9 mod 11 = 6 mod 11 = 6$$

$$x_3 = \lambda^2 - x_1 - x_2 mod\ p = 36 - 8 - 3 mod 11 = 3$$

$$y_3 = \lambda(x_1 - x_3) - y_1 mod\ p = 6(8-3) - 3 mod 11 = 5$$

Ответ: (8,3)+(3,6)=(3,5).

б) $(x_1,y_1)=(1,8)$.

$$\lambda = \frac{3x_1^2 + a}{2y_1} mod\ p = \frac{3 \cdot 1^2 + 1}{2 \cdot 8} mod 11 = \frac{4}{16} mod 11 = 4^{-1} mod 11 = 3 mod 11$$

$$= 3$$

$$x_3 = \lambda^2 - 2x_1 mod\ p = 9 - 2 \cdot 1 mod 11 = 7$$

$$y_3 = \lambda(x_1 - x_3) - y_1 mod\ p = 3(1-7) - 8 mod 11 = 7$$

Ответ: 2(1,8)=(7,7).

Число элементов группы эллиптической кривой $E_p(a,b)$ называется *порядком* этой группы.

Литература:

Воробейкина, И.В. Программирование средств защиты информации: учебное пособие для студентов специальности 10.05.03 «Информационная безопасность автоматизированных систем» очной формы обучения / И.В. Воробейкина; БГАРФ ФГБОУ ВО «КГТУ». – Калининград: Изд-во БГАРФ, 2021. - 70 с.

Глава 4. Алгоритм Диффи-Хеллмана.

14.3. Задание к лабораторной работе

Написать программу поиска порядка точки *P(0,1)* в группе эллиптической кривой $y^2=x^3+1$ над полем *GF(5).*

14.4. Методические указания и порядок выполнения работы

Арифметика эллиптических кривых не содержит прямых формул для вычисления кратного *mP* для заданной точки *P(x,y).* Эту операцию выполняют с помощью операций сложения и удвоения точки. Для этого надо представить число *m* в двоичной форме $m=b_t b_{t-1} \ldots b_0$, потом вычислить все точки *2P, 4P,…,* 2^tP и вычислить сумму тех точек 2^iP, для которых $b_i=1$. Например, $13_{10}=1101_2=1\times2^0+0\times2^1+1\times2^2+1\times2^3$, значит, *13P=P+4P+8P.*

Порядок точки P – это наименьшее натуральное число *n,* при котором *nP=O.*

Пример 13.3. Найти порядок точки *P(9,4)* в группе эллиптической кривой $E_{11}(6,3)$.

Решение.

1) *2P=P+P=(9,4)+(9,4):*

$$\lambda = \frac{3 \cdot 9^2 + 6}{2 \cdot 4} mod 11 = 249 \cdot 8^{-1} mod 11 = 249 \cdot 7 mod 11 = 5$$

$$x_3 = 5^2 - 2 \cdot 9 mod 11 = 7, \;\; y_3 = 5(9-7) - 4 mod 11 = 6,$$

$$2P(9,4) = (7,6)$$

2) *3P=2P+P=(7,6)+(9,4):*

$$\lambda = \frac{4-6}{9-7} mod 11 = \frac{-2}{2} mod 11 = -1 mod 11 = 10$$

$$x_3 = 10^2 - 2 - 9 mod 11 = 7, \;\; y_3 = 10(7-7) - 6 mod 11 = 5,$$

$$3P(9,4) = (7,5)$$

3) *4P=3P+P=(7,5)+(9,4):*

$$\lambda = \frac{4-5}{9-7} mod 11 = \frac{-1}{2} mod 11 = -2^{-1} mod 11 - 6 mod 11 = 5$$

$$x_3 = 5^2 - 7 - 9 mod 11 = 9, \;\; y_3 = 5(7-9) - 5 mod 11 = 7,$$

$$4P(9,4) = (9,7)$$

4) *5P=4P+P=(9,7)+(9,4):*

$$\lambda = \frac{4-7}{9-9} mod 11 \to \infty$$

5P=O, значит, порядок точки *P(9,4)* равен *5*.

14.5. Индивидуальное задание

Вариативность не предполагается. Освоить суммирование точек на эллиптической кривой, нахождение порядка точки, создать программный код.

14.6. Требования к отчету и защите

Показать выполненную на компьютере работу. Знать ответы на сформулированные выше вопросы. Защита работы проводится во время занятий. После защиты работа помещается в ЭИОС.

15. ЛАБОРАТОРНАЯ РАБОТА № 14. ВЫЧИСЛЕНИЕ СЕКРЕТНЫХ КЛЮЧЕЙ ПО АЛГОРИТМУ ECDH. ПРОГРАММИРОВАНИЕ СЕКРЕТНЫХ КЛЮЧЕЙ ПО АЛГОРИТМУ ECDH

15.1. Общие сведения

Цель: Научиться программировать вычисление секретных ключей по алгоритму ECDH.

Материалы, оборудование, программное обеспечение: online-компилятор C++ (необходима бесперебойная работа сети ИНТЕРНЕТ) или Visual Studio C++

Условия допуска к выполнению: предварительно пройти инструктаж по ТБ. Показать конспект по теоретической подготовке и запрограммировать подготовительный пример.

Критерии положительной оценки: Показать выполненную на компьютере работу. Ответить на вопросы преподавателя.

15.2. Теоретическое введение

Приведем эллиптический аналог открытого распределения ключей Диффи-Хеллмана (ECDH).

Пользователи *A* и *B* выбирают общие параметры:

- эллиптическую кривую над конечным полем;
- точку *P* на этой кривой, имеющую большой порядок *n* (она не обязательно должна быть порождающим элементом группы точек кривой, но порожденная ею подгруппа должна быть большой, предпочтительно того же порядка, что и сама группа). Точка *P* называется ***базовой.***

Общие параметры передаются открытым каналом связи.

1. Пользователь *A* случайно выбирает число K_A – свой секретный ключ, а пользователь *B* случайно выбирает число K_B – свой секретный ключ (числа близки по порядку к общему числу N_E точек кривой). Далее пользователи находят свои точки $Q= K_AP$ и $R= K_BP$ соответственно.

2. Пользователи обмениваются точками *Q* и *R* по открытому каналу.

3. Пользователь *A*, получив точку *R*, вычисляет точку $S= K_AR$.

4. Пользователь *B*, получив точку *Q*, вычисляет точку $S= K_BQ$.

Так как $K_AR= K_A(K_BP)= K_B(K_AP)= K_BQ$, то *S* – общий секретный ключ пользователей.

Вычисление секретного ключа на эллиптической кривой

Пример 14.1. Найти общий ключ для шифрования *K=(x,y),* используя алгоритм Диффи-Хеллмана на основе эллиптических кривых, если кривая имеет вид $y^2 = x^3 + 2x + 2\ mod 17$ и задает циклическую группу порядка *#C=19.*

Примитивный элемент равен *P=(5,1).* Секретный ключ Алисы *c=3*, секретный ключ Боба *d=10.*

Решение.

Алиса шифрует точку *P* своим секретным ключом: $Q=c\times P$, результат *Q* пересылает Бобу. Боб шифрует точку *P* своим секретным ключом: $R=d\times P$, результат *R* пересылает Алисе. Передача осуществляется по открытому каналу. Далее, Алиса шифрует полученный результат: $S=c\times R=c\times d\times P$, Боб шифрует полученный результат: $S=d\times Q=d\times c\times P$. Получается общий секретный ключ *S.*

Шифрует Алиса:

3P(5,1)=11_2(5,1)=1×2^0P(5,1)+1×2^1P(5,1)= P(5,1)+2P(5,1)=P(6,3)+P(5,1)= P(10,6) (вычисления см. ниже).

Вычисляем 2P(5,1):

$$\lambda=\frac{3\cdot 5^2+2}{2\cdot 1}mod17=\frac{77}{2}mod17=77\cdot 2^{-1}mod17=77\cdot 9mod17=13$$

$$x_3=13^2-2\cdot 5mod17=6,\ \ y_3=13\cdot(5-6)-1mod17=3$$

Следовательно, 2P(5,1)=P(6,3).

Вычисляем 2P(5,1)+ P(5,1)=P(6,3)+P(5,1):

$$\lambda=\frac{3-1}{6-5}mod17=2mod17=2$$

$$x_3=2^2-5-6mod17=10,\ \ y_3=2\cdot(5-10)-1mod17=6$$

Результат шифрования Алисы: 3P(5,1)=P(10,6).

Шифрует Боб:

10P(5,1)=1010_2P(5,1)= 1×2^3P(5,1)+1×2^1P(5,1)=8 P(5,1)+2P(5,1)= 2P(5,1)+2P(5,1)+2P(5,1)+2P(5,1)+2P(5,1)=P(6,3)+P(6,3)+P(6,3)+ P(6,3)+ P(6,3)=2P(6,3)+2P(6,3)+ P(6,3)= P(3,1)+P(3,1)+P(6,3)=2P(3,1)+P(6,3)= P(13,7)+P(6,3)=P(7,11) (вычисления см. ниже).

Вычисляем 2P(6,3):

$$\lambda=\frac{3\cdot 6^2+2}{2\cdot 3}mod17=\frac{110mod17}{6mod17}=8\cdot 6^{-1}mod17=8\cdot 3mod17=7$$

$$x_3=7^2-2\cdot 6mod17=3,\ \ y_3=7\cdot(6-3)-3mod17=1$$

Следовательно, 2P(6,3)=P(3,1).

Вычисляем 2P(3,1):

$$\lambda=\frac{3\cdot 3^2+2}{2\cdot 1}mod17=\frac{29mod17}{2mod17}=\frac{12}{2}mod17=6$$

$$x_3=6^2-2\cdot 3mod17=13,\ \ y_3=6\cdot(3-13)-1mod17=7$$

Следовательно, 2P(3,1)=P(13,7).

Вычисляем P(13,7)+P(6,3):

$$\lambda=\frac{3-7}{6-13}mod17=\frac{4}{7}mod17=4\cdot 7^{-1}mod17=4\cdot 5mod17=3$$

$$x_3=3^2-13-16mod17=7,\ \ y_3=3\cdot(13-7)-7mod17=11$$

Следовательно, P(13,7)+P(6,3)=P(7,11).

Алиса передает Бобу результат *P(10,6),* Боб передает Алисе результат *P(7,11).*

Алиса шифрует:

*3P(7,11)=11$_2$(7,11)=1×2^{0}P(7,11)+1×2^{1}P(7,11)=P(7,11)+2P(7,11)=P(7,11)+P(5,1)=**P(13,10)*** (вычисления см. ниже).

Вычисляем 2P(7,11):

$$\lambda = \frac{3 \cdot 7^2 + 2}{2 \cdot 11} mod 17 = \frac{149 mod 17}{22 mod 17} = 13 \cdot 5^{-1} mod 17 = 6$$

$$x_3 = 6^2 - 2 \cdot 7 mod 17 = 5, \;\; y_3 = 6 \cdot (7 - 5) - 11 mod 17 = 1$$

Следовательно, 2P(7,11)=P(5,1).

Вычисляем P(7,11)+P(5,1):

$$\lambda = \frac{1 - 11}{5 - 7} mod 17 = \frac{10}{2} mod 17 = 10 \cdot 2^{-1} mod 17 = 5$$

$$x_3 = 5^2 - 7 - 5 mod 17 = 13, \;\; y_3 = 5 \cdot (7 - 13) - 11 mod 17 = 10$$

*Следовательно, P(7,11)+P(5,1)=P(13,10) – **общий секретный ключ**.*

Боб шифрует:

*10P(10,6)=1010$_2$P(10,6)=1×2^{3}P(10,6)+1×2^{1}P(10,6)=8P(10,6)+2P(10,6)=2P(10,6)+2P(10,6)+2P(10,6)+2P(10,6)+2P(10,6)=P(16,13)+P(16,13)+P(16,13)+P(16,13)+P(16,13)=2P(16,13)+2P(16,13)+P(16,13)=P(0,11)+P(0,11)+P(16,13)=2P(0,11)+P(16,13)=P(9,16)+P(16,13)=**P(13,10)*** (вычисления см. ниже).

Общий секретный ключ – ***P(13,10)*** – у Алисы и Боба совпадает.

Вычисляем 2P(10,6):

$$\lambda = \frac{3 \cdot 10^2 + 2}{2 \cdot 6} mod 17 = \frac{302 mod 17}{12 mod 17} = 13 \cdot 12^{-1} mod 17 = 11$$

$$x_3 = 11^2 - 2 \cdot 10 mod 17 = 16, \;\; y_3 = 11 \cdot (10 - 16) - 6 mod 17 = 13$$

Следовательно, 2P(10,6)=P(16,13).

Вычисляем 2P(16,13):

$$\lambda = \frac{3 \cdot 16^2 + 2}{2 \cdot 13} mod 17 = \frac{770 mod 17}{26 mod 17} = 5 \cdot 9^{-1} mod 17 = 10$$

$$x_3 = 10^2 - 2 \cdot 16 mod 17 = 0, \;\; y_3 = 10 \cdot (16 - 0) - 13 mod 17 = 11$$

Следовательно, 2P(16,13)=P(0,11).

Вычисляем 2P(0,11):

$$\lambda = \frac{3 \cdot 0^2 + 2}{2 \cdot 11} mod 17 = \frac{2}{22 mod 17} = 1 \cdot 11^{-1} mod 17 = 14$$

$$x_3 = 14^2 - 2 \cdot 0 mod 17 = 9, \;\; y_3 = 14 \cdot (0 - 9) - 11 mod 17 = 16$$

Следовательно, 2P(0,11)=P(9,16).

Вычисляем P(9,16)+P(16,13):

$$\lambda = \frac{13 - 16}{16 - 9} mod 17 = -\frac{3}{7} mod 17 = 14 \cdot 7^{-1} mod 17 = 2$$

$$x_3 = 2^2 - 9 - 16 mod 17 = 13, \;\; y_3 = 2 \cdot (9 - 13) - 16 mod 17 = 10$$

Следовательно, P(9,16)+P(16,13)=P(13,10) – ***общий секретный ключ.***

В процессе вычислений можно проверять результаты, подставляя значения *x* и *y* в уравнение эллиптической кривой. Если окажется, что точка *P(x, y)* принадлежит кривой (т.е. значения левой и правой частей уравнения совпадают), то координаты точки вычислены верно.

Литература:

Воробейкина, И.В. Программирование средств защиты информации: учебное пособие для студентов специальности 10.05.03 «Информационная безопасность автоматизированных систем» очной формы обучения / И.В. Воробейкина; БГАРФ ФГБОУ ВО «КГТУ». – Калининград: Изд-во БГАРФ, 2021. - 70 с.

Глава 4. Алгоритм Диффи-Хеллмана.

15.3. Задание к лабораторной работе

Сгенерировать общий секретный ключ для двух пользователей по схеме Диффи-Хеллмана, если выбрана эллиптическая кривая $E_{211}(0, -4)$ и точка $P(2,2)$. Пусть секретный ключ пользователя *A* будет $K_A=121$, а пользователя *B* – $K_B=203$.. Запрограммировать задачу.

Вопросы и задания.

1. Вычислить секретные ключи и общий ключ для системы Диффи-Хеллмана $p=23$ $g=5$ $x_a=5$ $x_b=7$.

Найти порядок точки $P(3,6)$ в группе эллиптической кривой $y^2=x^3+2x+3$ над полем $GF(97)$.

15.4. Методические указания и порядок выполнения работы

Программная реализация алгоритма суммирования координат

Разработанный алгоритм реализован на языке *C++* и позволяет в автоматизированном режиме выполнять один из этапов вычисления ключей по алгоритму Диффи-Хеллмана, а именно – суммирование координат точек на задаваемой пользователем эллиптической кривой.

Пользователь задает следующие параметры:

– *P* – размерность системы вычетов, используемой при определении поля эллиптической кривой;

– *x1*, *y1* – координаты точки *P*;

– *x2*, *y2* – координаты точки *Q*;

– *a* – аргумент переменной *x* в уравнении эллиптической кривой;

– *b* – свободный член уравнения эллиптической кривой.

Способы реализация алгоритма различны для ситуаций, когда координаты точек P и Q равны, и когда они различны.

Для $P \neq Q$:

1. определяется угловой коэффициент эллиптической кривой:

$$\lambda = \frac{y_2 - y_1}{x_2 - x_1} \; mod \; P$$

2. Вычисляются координаты результирующей точки:

$$x_3 = \lambda^2 - x_1 - x_2 \; mod \; P$$
$$y_3 = \lambda(x_1 - x_3) - y_1 \; mod \; P$$

Для $P = Q$:

1. Определяется угловой коэффициент эллиптической кривой:

$$\lambda = \frac{3x_1^2 + a}{2y_2} \; mod \; P$$

2. Вычисляются координаты результирующей точки:

$$x_3 = \lambda^2 - 2x \; mod \; P$$
$$y_3 = \lambda(x_1 - x_3) - y_1 \; mod \; P$$

При программировании следует учесть, что вычисления координат точек на эллиптических кривых выполняются по правилам модульной арифметики, что накладывает ряд особенностей на привычные арифметические алгоритмы. Так, операции сложения и умножения, как и обратные им, дополняются вычислением остатка от деления конечного результата на основании используемой системы вычетов, что делает невозможными прямые действия с дробными и отрицательными значениями. Таким образом, необходимо избежать наличия в конечном выражении отрицательного либо дробного делимого.

Для этих целей применяется ряд приемов:

1. дробное число записывается в виде произведения числителя на число, обратное знаменателю, причем для вычисления обратного числа в связи с использованием определенной системы вычетов применяется отдельный алгоритм поиска обратных значений;

2. остаток от деления отрицательного числа равняется разности основания системы вычетов и остатка от деления этого же числа без знака.

Алгоритм поиска обратного значения представляет собой простой процесс перебора для поиска всех решений уравнения $(a \times b) \; mod \; P = 1$, где $0 < a < P$ и $0 < b < P$. Таким образом, обратным для данного числа будет являться то, которое образует с ним решение вышеприведенного уравнения.

Программа автоматизирует сложение координат двух точек на эллиптической кривой. Параметры кривой и слагаемые координаты указываются пользователем и могут быть произвольны в рамках диапазона 32-битных значений со знаком.

Программа разработана на языке *C++*, и ее исходный код оптимизирован для компиляции с использованием среды разработки *Borland Turbo C++* для *32*-битных систем семейства *Windows*.

В процессе работы программа последовательно выполняет следующие шаги:

1. запуск цикла выполнения программы;
2. последовательный вывод на экран запросов на ввод с клавиатуры значений *P, x1, y1, x2, y2, a и b*.
3. проверка введенных координат на предмет совпадения;
4. выбор режима работы в зависимости от результата проверки;
5. вычисление результирующих координат в выбранном режиме;
6. вывод на консоль результатов вычисления;
7. проверка корректности результата вычисления путем подстановки полученных координат в уравнение эллиптической кривой;
8. вывод на консоль сообщения о результате проверки;
9. вывод сообщения об окончании работы программы и необходимости перезапуска либо выхода из нее;
10. начало следующей итерации основного цикла в случае, если пользователь выбрал вариант продолжения работы программы, в противном случае – завершение.

На этапах, связанных с вычислениями, включен ряд необходимых проверок используемых переменных, которые проходят преобразования, связанные с вышеописанными приемами устранения отрицательных и дробных значений.

Для вычисления обратного значения используется отдельная функция, вызываемая по необходимости и осуществляющая поиск нужной пары взаимно простых чисел путем последовательного перебора всех возможных значений второго аргумента уравнения.

Конечным результатом работы программы является пара чисел, соответствующих координатам точки, полученной в результате суммирования.

В программном коде используются следующие переменные:

– *p* – размерность системы вычетов;
– *a* – аргумент уравнения эллиптической кривой;
– *b* – свободный член уравнения эллиптической кривой;
– *x1* – координата первой точки по оси абсцисс;
– *y1* – координата первой точки по оси ординат;
– *x2* – координата второй точки по оси абсцисс;
– *y2* – координата второй точки по оси ординат;
– *x3* – координата результирующей точки по оси абсцисс;
– *y3* – координата результирующей точки по оси ординат;
– *lam* – значение углового коэффициента;

– *lu* – значение числителя при вычислении углового коэффициента;

– *ll* – значение знаменателя при вычислении углового коэффициента;

– *keycode* – вспомогательная переменная, используемая при идентификации нажатой клавиши;

– *lt* – результат вычисления левой части уравнения в ходе проверки;

– *rt* – результат вычисления правой части уравнения в ходе проверки.

obr() – функция, осуществляющая поиск обратного значения числа в заданной системе вычетов. Функция объявлена и описана в коде программы непосредственно перед главной функцией. В функции *obr()* применяются следующие переменные:

– *p* – аналогично одноименной переменной в главной функции – размерность системы вычетов;

– *n* – исходное число для поиска обратного значения;

– *i* – вспомогательная переменная, выполняющая роль счетчика цикла и также соответствующая в каждой итерации второму аргументу решаемого уравнения;

Листинг:

```
#include <stdlib.h>
#include <stdio.h>
#include <conio.h>
long obr(long p, long n)
{
long r;
for (long i = 1; i < p; i++)
        {
        if ((n*i)%p == 1)
                {
                r = i;
                break;
                }
        else
                        r = -1;
        }
return r;
}
int main()
{
long p, x1, x2, y1, y2, x3, y3, a, b, lu, ll, lam;
int keycode;
while (keycode != 27)
```

```
{
printf("P = ");
scanf("%ld", &p);
printf("x1 = ");
scanf("%ld", &x1);
printf("y1 = ");
scanf("%ld", &y1);
printf("x2 = ");
scanf("%ld", &x2);
printf("y2 = ");
scanf("%ld", &y2);
printf("a = ");
scanf("%ld", &a);
printf("b = ");
scanf("%ld", &b);
if (x1==x2 && y1==y2)
        {
        lu = 3*x1*x1+a;
        if (lu < 0)
                lu= p-(0-lu)%p;
        ll = 2*y1;
        if (ll < 0)
                ll = p-(0-ll)%p;
        if (lu%ll != 0)
                lam = (lu * obr(p,ll))%p;
        else
                lam = (lu/ll)%p;
        if (lam < 0)
                lam = p-(0-lam)%p;
        printf("\nLam = %ld / %ld mod %ld = %ld", lu, ll, p, lam);
        if ( lam*lam-2*x1 >= 0 )
                x3 = (lam*lam-2*x1)%p;
        else
                x3 = p-(0-(lam*lam-2*x1))%p;
        printf("\nx3 = %ld", x3);
        if ( lam*(x1-x3)-y1 >= 0 )
                y3 = (lam*(x1-x3)-y1)%p;
        else
                y3 = p-(0-(lam*(x1-x3)-y1))%p;
        printf("\ny3 = %ld", y3);
```

```
            }
        else
            {
            lu = y2-y1;
            if (lu < 0)
                lu= p-(0-lu)%p;
            ll = x2-x1;
            if (ll != 0) {
                if (ll < 0)
                    ll = p-(0-ll)%p;
                if (lu%ll != 0)
                    lam = (lu * obr(p,ll)) % p;
                else
                    lam = (lu/ll)%p;
                if (lam < 0)
                    lam = p-(0-lam)%p;
                printf("\nLam=%ld/%ld mod %ld =%ld", lu, ll, p, lam);
                if ( lam*lam-2*x1 >= 0 )
                    x3 = (lam*lam-x1-x2)%p;
                else
                    x3 = p-(0-(lam*lam-x2-x1))%p;
                printf("\nx3 = %ld", x3);
                if ( lam*(x1-x3)-y1 >= 0 )
                    y3 = (lam*(x1-x3)-y1)%p;
                else
                    y3 = p-(0-(lam*(x1-x3)-y1))%p;
                printf("\ny3 = %ld", y3);
            }
            else {
                printf("\nТочка бесконечно удалена\n");
            }
        }
long lt, rt;
lt = (y3*y3)%p;
if (lt < 0)
        lt = p-(0-lt)%p;
rt = (x3*x3*x3+a*x3+b)%p;
if (rt < 0)
        rt = p-(0-rt)%p;
printf("\nlt = %ld; rt = %ld",lt,rt);
```

```
if (lt == rt)
        printf("\n\nВерно");
else
        printf("\n\nНеверно");
printf("\n\nEsc - выход, Enter - продолжить\n");
keycode = getch();
}
return 0;
}
```

15.5. Индивидуальное задание

Вариативность не предполагается. Освоить генерирование секретных ключей в ECDH, создать программный код.

15.6. Требования к отчету и защите

Показать выполненную на компьютере работу. Знать ответы на сформулированные выше вопросы. Защита работы проводится во время занятий. После защиты работа помещается в ЭИОС.

16. ЗАКЛЮЧЕНИЕ

В данном пособии на примерах рассматриваются различные методы и приемы написания программ, которые можно применять в языке C++. Каждое занятие состоит из теоретических положений, упражнений. Пособие рассчитано как для начинающих программистов, так и для тех, кто хочет усовершенствовать свои знания.

Предполагается, что студенты пользуются лекционным материалом и рекомендованной литературой, поэтому теоретический материал в полном объеме не приводится.

В основу пособия положены лабораторные занятия, проводимые автором по дисциплине «Программирование средств защиты информации» для студентов специальности 10.05.03 «ИБАС».

17. ЛИТЕРАТУРА

1. Шилдт, Герберт. Полный справочник по C++, 4-е издание. Пер. с англ. / , Герберт Шилдт. – Мсква: Издательский дом «Вильямс», 2006. –800 с.

2. Страуструп, Бьярн. Программирование: принципы и практика использования C++: Пер. с англ. / Бьярн Страуструп. – Москва: ООО «И.Д. Вильямс», 2011. –1248 с.

3. https://www.turboreferat.ru/programming-computer/bezuslovnaya-odnomernaya-optimizaciya/244283-1279981-page1.html

4. Воробейкина, И. В. Методы и средства криптографической защиты информации: учебное пособие для студентов специальности 10.05.03 «Информационная безопасность автоматизированных систем» очной формы обучения / И. В. Воробейкина; БГАРФ ФГБОУ ВО «КГТУ». – Калининград: Изд-во БГАРФ, 2022. - 114 с.

5. Воробейкина, И. В. Программирование средств защиты информации: учебное пособие для студентов специальности 10.05.03 «Информационная безопасность автоматизированных систем» очной формы обучения / И. В. Воробейкина; БГАРФ ФГБОУ ВО «КГТУ». – Калининград: Изд-во БГАРФ, 2021. - *70* с.

6. Воробейкина, И. В. Технологии и методы программирования, часть II: учебное пособие для студентов специальности 10.05.03 «Информационная безопасность автоматизированных систем» очной формы обучения / И. В. Воробейкина; БГАРФ ФГБОУ ВО «КГТУ». – Калининград: Изд-во БГАРФ, 2021.

Локальный электронный методический материал

Ирина Владимировна Воробейкина

ПРОГРАММИРОВАНИЕ СРЕДСТВ ЗАЩИТЫ ИНФОРМАЦИИ

Редактор Г. А. Смирнова

Уч.-изд. л. 5,6 Печ. л. 5,6

Издательство федерального государственного бюджетного образовательного учреждения высшего образования
«Калининградский государственный технический университет».
236022, Калининград, Советский проспект, 1