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

И.В. Воробейкина

ТЕХНОЛОГИИ И МЕТОДЫ ПРОГРАММИРОВАНИЯ

Утверждено редакционно-издательским советом ФГБОУ ВО «КГТУ»
в качестве учебно-методического пособия по лабораторным работам
для студентов специальности
10.05.03 Информационная безопасность автоматизированных систем

Калининград
Издательство ФГБОУ ВО «КГТУ»
2022

УДК 004.9(075)

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

Воробейкина, И.В.
Технологии и методы программирования: учеб.-методич. пособие по лабораторным работам для студ. специальности 10.05.03 Информационная безопасность автоматизированных систем / И.В. Воробейкина. – Калининград: Изд-во ФГБОУ ВО «КГТУ», 2022. – 157 с.

Учебно-методическое пособие является руководством по проведению цикла лабораторных работ по дисциплине «Технологии и методы программирования» студентами, обучающимися по специальности 10.05.03 Информационная безопасность автоматизированных систем. Лабораторные работы предназначены для закрепления теоретического материала.
Список лит. – 7 наименований

Учебно-методическое пособие рассмотрено и одобрено в качестве локального электронного методического материала кафедрой информационной безопасности Института цифровых технологий ФГБОУ ВО «Калининградский государственный технический университет» 14 июня 2022 г., протокол № 9

Учебно-методическое пособие рекомендовано к использованию в качестве локального электронного методического материала в учебном процессе методической комиссией Института цифровых технологий 20 сентября 2022 г., протокол № 6

УДК 004.9(075)
© Федеральное государственное бюджетное образовательное учреждение высшего образования «Калининградский государственный технический университет», 2022 г.
© Воробейкина И.В., 2022 г.

ОГЛАВЛЕНИЕ

1. ВВЕДЕНИЕ

Данное учебно-методическое пособие предназначено для студентов направления 10.05.03 Информационная безопасность автоматизированных систем, изучающих дисциплину «Технологии и методы программирования».

Лабораторный практикум содержит 22 лабораторные работы.

Лабораторные работы проводятся в лабораториях кафедры.

Цель лабораторного практикума по дисциплине: в результате выполнения лабораторных работ ожидается, что студенты сформируют навыки программирования на языках HTML и C++.

2. ЛАБОРАТОРНАЯ РАБОТА № 1. ЯЗЫК HTML. ТЕГИ ФИЗИЧЕСКОГО ФОРМАТИРОВАНИЯ ТЕКСТА. ЦВЕТ В HTML-ДОКУМЕНТАХ

2.1. Общие сведения

Цель: изучить теги форматирования текста в html-документе.

Материалы, оборудование, программное обеспечение: браузер Google Chrome, приложение «Блокнот».

Условия допуска к выполнению: показать конспект по теоретической подготовке.

Критерии положительной оценки: показать выполненную работу; ответить на вопросы преподавателя.

2.2. Теоретическое введение

Структура HTML-документа. Теги

HTML-документ имеет простую, легко воспринимаемую структуру. Состоит он из двух частей – служебной, называемой «**HEAD**», или «головой» документа, и текстовой («**BODY**»). Служебная часть несет сведения о кодировке документа, его содержании, имени и некоторые другие дополнительные сведения. Текстовая часть содержит сам текст и команды браузеру по его отображению. Эти команды называются теги и заключаются в острые скобки. Большинство тегов являются парными. Закрывающий тег отличается от начального наличием специального знака – слеша (/).

< TITLE > наш первый файл < /TITLE > – это пример тегов-контейнеров, вводящих имя HTML-файла (располагается в голове документа).

< BR > – команда перевода строки. Этот тег не несет никакого содержания; он не форматирует текст. Поэтому теги такого типа – непарные – не требуют закрывающего тега.

Типичная структура HTML-документа:

<html>	Открываем документ
<head>	Открываем «голову»
<title> Наш первый файл</title>	Вводим заголовок документа
</head>	Закрываем «голову»
<body>	Открываем текстовую часть документа
*В этом месте содержится текст *	Текст может быть любым.
</body >	Закрываем текстовую часть.
</html>	Закрываем весь документ.

Теги заголовков

Для создания заголовков и подзаголовков разделов используется следующие конструкции:

<H цифра от 1 до 6> текст </H цифра от 1 до 6>

Главный заголовок вводится цифрой 1, подзаголовки меньших разделов – цифрами от 2 до 6, используется полужирное начертание шрифта.

Теги управления начертанием шрифта

Применяются следующие теги, управляющие начертанием текста:

Тег	Результат использования тега
<*b*>…некоторый текст…</*b*>	данный контейнер отображает заключенный в него текст как полужирный
<*strong*>...некоторый текст…</*strong*>	данный контейнер отображает заключенный в него текст как полужирный (аналог <*b*>…</*b*>)
<*i*>…некоторый текст…</*i*>	данный контейнер отображает заключенный в него текст как курсив
<*em*>…некоторый текст…</*em*>	данный контейнер отображает заключенный в него текст как курсив (аналог <*i*>…</*i*>)
<*s*>…некоторый текст…</*s*>	данный контейнер отображает заключенный в него текст как перечеркнутый
<*strike*>…некоторый текст…<*strike*>	данный контейнер отображает заключенный в него текст как перечеркнутый
<*u*>…некоторый текст…</*u*>	данный контейнер отображает заключенный в него текст как подчеркнутый
<*sub*>…некоторый текст…</*sub*>	данный контейнер отображает заключенный в него текст как подстрочный (как в формуле H_2O)
<*sup*>…некоторый текст…</*sup*>	данный контейнер отображает заключенный в него текст как надстрочный (как в формуле $A^2+B^2=C^2$)
<*code*>…некоторый текст…</*code*>	данный контейнер отображает заключенный в него текст как код или формула
<*address*>…некоторый текст…</*address*>	данный контейнер отображает заключенный в него текст как адрес (чаще всего курсивом)
<*var*>...некоторый текст…</*var*>	данный контейнер отображает заключенный в него текст как некоторую переменную (курсивом)
<*cite*>…некоторый текст…</*cite*>	данный контейнер отображает заключенный в него текст как цитату (курсив)
<*tt*>…некоторый текст…</*tt*>	данный контейнер отображает заключенный в него текст как моноширинный (аналог печатной машинки)

<kbd>...некоторый текст...*</kbd>*	данный контейнер отображает заключенный в него текст как полужирный моноширинный
<blockquote>...текст …*</blockquote>*	данный контейнер отображает заключенный в него текст как особую цитату – без изменения шрифта, но со сдвигом абзаца

Размер и тип шрифта

Для указания размера шрифта используются либо абсолютные, либо относительные размеры. Для абсолютных размеров используется следующая конструкция:

...текст...

Самый большой шрифт – цифра «7», самый маленький – «1». Относительные размеры даются от размера шрифта, используемого по умолчанию (обозначается как «+0»), и вводятся цифрами от «+4» до «-2»; т.е. «+4» относительного размера соответствует «7» абсолютного.

Для указания типа шрифта применяется следующий тег:

* ...текст... *

Рекомендуется использовать шрифты следующих типов, как наиболее распространенные: *Arial, Times; Times New Roman; Helvetica; Courier; Verdana; Tahoma; Comic Sans; Garamond.* Если будет указан шрифт, отсутствующий в операционной системе, то браузер будет подставлять любой подходящий, что может привести к искажению внешнего вида HTML-страницы. Размер и тип шрифта можно указывать одной строкой и в любой последовательности, например:

* текст * – абсолютный размер 7 и шрифт *Comic Sans*.Такую конструкцию можно применять для заголовков.

Цвет на HTML-странице

Цвет отдельно задается для фона и для текста HTML-странички. Для этого используются атрибуты тега *body*, например:

<body bgcolor="green" text="red" > – заливка будет зеленой, а текст – красным.

В указании цвета могут использоваться соответствующие слова английского языка – *yellow, green* или *silver*. Но для более правильного отображения цвета HTML-документа разными браузерами используется так называемый RGB-код. Он состоит из некоторого числа в шестнадцатеричной системе счисления, разбитого на три части, по две цифры в каждой. Первая часть – интенсивность красного цвета, следующие две цифры – зеленая составляющая, а последняя часть – синяя. Наименьшая интенсивность – черный цвет – указывается как "*00 00 00*", а наибольшая интенсивность – "*FF FF FF*" (белый цвет) Символ RGB-кода – «решетка» (#), которая означает, что вводится цвет в шестнадцатеричной форме:

<body text="#336699">

Любой другой элемент на HTML-странице тоже может быть раскрашен в тот или иной цвет. Для этого используется атрибут «color», которому присваивается некоторое значение этим же способом – либо при помощи английских слов, обозначающих цвета, либо при помощи RGB-кода, например: ** зеленый цвет шрифта**

<H2 color="#53FFCC"> текст заголовка </H2>

Выравнивание

Для выравнивания текста используется тег * …некоторый текст… ,* после закрывающего тега автоматически происходит перевод строки.

К тегу *<P>* для выравнивания содержимого параграфа по левой или правой стороне, по центру или с заполнением следует добавить атрибут *align* с соответствующим значением: *left, right, center, justify*, например:

* …некоторый текст… *

Контейнер * … * обязательно должен содержать внутри какой-либо текст; пустые параграфы браузерами могут опускаться и никак не отображаться на экране (даже переносом строки). Нельзя также вкладывать один тег параграфа в другой.

Литература:

Воробейкина, И.В. Технологии и методы программирования. Лабораторный практикум / И.В. Воробейкина. – Калининград: Изд-во БГАРФ, 2019. – 97 с.

Глава 1. Язык HTML.

2.3. Задание к лабораторной работе

– Что такое теги физического форматирования текста? В каких случаях они продолжают использоваться?

– Как изменить тип и размер шрифта?

– Как можно изменить фон HTML документа? Как можно менять цвет шрифта всего документа? Отдельного слова?

– Что такое выравнивание? Как выровнять текст по правому краю? По центру? По левому краю?

– Как изменить шрифт на подстрочный? Надстрочный? Зачеркнутый? Жирный курсив?

2.4. Методические указания и порядок выполнения работы

Запишите следующую программу и изучите работу тегов:

<html>

<head>

<title> Наш первый файл</title>
</head>
<body bgcolor="green" text="red" >
*Формула соляной кислоты
*
H₂SO₄
*Формула параболы
*
y=x²
</body >
</html>

2.5. Индивидуальное задание

– Создайте новый HTML-документ, в теле этого документа создайте главный заголовок: «Моя первая интернет-страница»

– Измените цвет фона этого документа на черный, а цвет шрифта – на золотой. Откройте его браузером, убедитесь, что цвета документа изменились.

– После заголовка введите свою фамилию, имя и отчество. Каждая буква в фамилии, имени и отчестве должна быть разного размера, разного шрифта и разного цвета. Расположите их по центру документа.

– Введите следующее предложение с сохранением предложенного форматирования:

H_2O – это формула *воды.*

До и после него – несколько тегов перевода строки. Выровнять по правому краю.

– Введите следующее предложение с сохранением предложенного форматирования:

Теорема Пифагора: **$A^2+B^2=C^2$**

До и после него – несколько тегов перевода строки. Выровнять по центру.

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. СПИСКИ, ТЕКСТ ПРЕДВАРИТЕЛЬНО ЗАДАННОГО ФОРМАТА, РАЗДЕЛИТЕЛЬНЫЕ ПОЛОСЫ

3.1. Общие сведения

Цель: научиться использовать списки, разделительные полосы текст предварительно заданного формата в оформлении HTML-страниц.

Материалы, оборудование, программное обеспечение: браузер Google Chrome, приложение «Блокнот».

Условия допуска к выполнению: показать конспект по теоретической подготовке.

Критерии положительной оценки: показать выполненную работу; ответить на вопросы преподавателя.

3.2. Теоретическое введение

Для создания списка используют следующую конструкцию из парного тега-контейнера:

< UL > "ваша информация" < /UL > или. < OL > "ваша информация" < /OL >

Внутри контейнера используется непарный тег **<*li*>.**

<*ol***>** – тег упорядоченного списка, содержащий элементы **<*li*>**.

*Атрибуты**:***

compact – список делается по возможности более компактным;

*star**t**= n* – список будет пронумерован не с «*1*» (по умолчанию), а с *n;*

type =формат (установка формата нумерации данного списка: *A* – прописные буквы; *a* – строчные буквы; *I* – большие римские цифры; *i* – маленькие римские цифры; *1* – арабские цифры (по умолчанию)).

<*ul***> –** тег неупорядоченного списка, содержащий элементы **<*li*>**.

*Атрибуты**:***

compact – список делается по возможности более компактным;

type =формат (установка формата: *circle* – кружок, *disc* – диск (по умолчанию), *square* – квадрат).

Списки определений

Списки определений – это специальные списки, каждый элемент которого состоит из двух частей: ***термин*** и его ***определение***. Перед элементами списка не ставятся ни вводные маркеры, ни порядковые номера, поэтому они вводятся как часть текста.

<dl> – создание списка определений, содержащих теги *<dt>* и *<dd>*.

Атрибуты:

compact – список делается по возможности более компактным;

<dt> – задается описательно-условная часть (термин) для элемента списка определений;

<dd> – задается описательная часть (определение) для элемента списка определений.

<dl> < dt> Вторник </dt> <dd> Второй день недели.</dd> <dt> Четверг </dt> <dd> Четвертый день недели.</dd>

Разделительные полосы

<hr> – горизонтальная линия.

Атрибуты:

align= тип линии – задается способ выравнивания: по левому краю (*left*), по центру (*center*), по правому краю (*right*).

noshade – сплошная линия;

size=пиксели – установка толщины линии, равной числу пикселей;

width=значение – установка ширины линии равной целому значению пикселей или в процентах от ширины страницы.

Очень удобен парный тег *<pre>* – заключенный в теги ***<pre>…</pre>*** текст будет отображаться так, как он был предварительно отформатирован, без обработки, с точным соблюдением переносов строк и интервалов.

Литература:

Воробейкина, И.В. Технологии и методы программирования. Лабораторный практикум / И.В. Воробейкина. – Калининград: Изд-во БГАРФ, 2019. – 97 с.

Глава 1. Язык HTML

3.3. Задание к лабораторной работе

– Для чего используются списки?

– Какие виды списков вам известны?

3.4. Методические указания и порядок выполнения работы

Запишите следующую программу и изучите работу тегов:

<html>

<head>

<title> Наш первый файл</title>

</head>

<body >

*<ol start=2 type=a> ВТОРНИК< li >СРЕДА< li >ЧЕТВЕРГ
*

*<ul type=square> < li >январь< li >февраль< li >март *

<H1> Разделительные полосы</H1>

<H3> Для отдельных частей текста можно использовать разделительные полосы

Это – обычная полоса, без атрибутов<HR>

Это – полоса толщиной 5 <HR SIZE=5>
Это – полоса толщиной 10 и шириной 100 <HR SIZE= 10 WIDTH=100>
Это – полоса толщиной 5 с атрибутом NOSHADE
<HR SIZE=5 NOSHADE> </H3>

<pre>

	Январь	*Февраль*	*Март*
Товар 1	*1000*	*2000*	*3000*
Товар 2	*1500*	*2500*	*3500*

</pre>
</body >
</html>

3.5. Индивидуальное задание

Создайте документ, имеющий нумерацию с римскими цифрами.
Создайте маркированный список, в качестве маркера используйте кружок.

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. ГРАФИКА, ССЫЛКИ, БЕГУЩАЯ СТРОКА

4.1. Общие сведения

Цель: научиться использовать графическое оформление в HTML-документах, организовывать связи внутри и между в HTML-документами, применять бегущую строку в оформлении HTML-страниц.

Материалы, оборудование, программное обеспечение: браузер Google Chrome, приложение «Блокнот».

Условия допуска к выполнению: показать конспект по теоретической подготовке.

Критерии положительной оценки: показать выполненную работу; ответить на вопросы преподавателя.

4.2. Теоретическое введение

Для вставки графики используется непарный тег *<img>*. Он имеет следующие атрибуты:

align= тип выравнивания – задается расположение графического изображения относительно текста. Используются свойства: *center* – выравнивание центра изображения по нижней линии строки текста, *bottom* – выравнивание базовой линии строки текста по нижнему краю изображения, *left* – расположение текста в левой части текстового потока *right* – расположение текста в правой части текстового потока.

height= «значение» – задается высота изображения в пикселях или процентах;

width=«значение» – указывается ширина изображения в пикселях или процентах;

hspace=«значение» – задается размещение слева и справа от изображения областей свободного пространства шириной указанного количества пикселей;

vspace = «значение» – задается размещение над и под изображением областей свободного пространства размером указанного количества пикселей;

src = «URL» – источник изображения. Указывается URL – правильное местонахождение данного Интернет-ресурса (полный – если изображение находится вне данного сервера; относительный – если оно расположено на файловой подсистеме этого сервера). **Атрибут «src» является обязательным, пропускать его нельзя.**

Изображение может использоваться как фон HTML-страницы. Для этого применяют такой тег: <body background="fon.gif">.

Использование ссылок в гипертекстовых документах

Для создания ссылки используется следующий тег:

<A HREF = “адрес ссылки”> Текст ссылки.</A>

Атрибуты: *VLINK* – цвет прочитанной ссылки. *LINK* – цвет непрочитанной ссылки.

В качестве ссылки можно использовать изображение:

* <IMG SRC=имя графического файла> Поясняющий текст*

Локальные ссылки внутри документа

Если длина документа велика, имеет смысл организовать ссылки внутри этого документа на его отдельные самостоятельные логические части, расположив их, например, в начале документа. Такие ссылки называются локальными.

Якорь – именованная точка в документе, к которой происходит ссылка.

Формат якоря: *<A NAME = имя якоря> Текст на экране</A>*

Формат ссылки: *<A HREF = #имя якоря>Текст</A>*

Без указания имени якоря (например: *Назад*) мы попадаем на начало нашей страницы.

«Бегущая» строка

Бегущая строка создается парным тегом **<*MARQUEE*>.** Этот тег имеет большое количество атрибутов, которые мы приводим в следующей таблице:

Атрибут	Описание
WIDTH	Ширина поля строки в пикселях или % от ширины окна
HEIGHT	Высота поля строки в пикселях
HSPACE, VSPACE	Интервалы по горизонтали и вертикали между текстом строки и краями поля в пикселях
ALIGN	Положение текста строки в ее поле: *TOP* (вверху), *BOTTOM* (внизу), *MIDDLE* (посередине)
DIRECTION	Направление движения: *LEFT, RIGHT*
BEHAVIOR	Характер движения строки: *SCROLL* – текст появляется от одного края и скрывается за другим, *SLIDE* – вытягивается из одного края и останавливается у другого, *ALTERNATE* – переменное направление от одного края к другому, затем обратно.
SCROLLDELAY	Пауза
BGCOLOR	Цвет поля строки

Литература:

Воробейкина, И.В. Технологии и методы программирования. Лабораторный практикум / И.В. Воробейкина. – Калининград: Изд-во БГАРФ, 2019. – 97 с.

Глава 1. Язык HTML

4.3. Задание к лабораторной работе

– Какие ошибки HTML-программиста в оформлении страниц приводят к очень медленной загрузке документа?

– Какой элемент позволяет установить связи между отдельными HTML-документами? Между логическими элементами внутри документа?

4.4. Методические указания и порядок выполнения работы

Запишите следующие программы и изучите работу тегов:

<html>
<head>
<title> Наш первый файл</title>
</head>
<BODY BGCOLOR=lightblue><IMG SRC="pict.gif" ALIGN=LEFT>
<H1>Текст обтекает графику справа</H1> Это пример совместного использования текста и графики. <P>А это – новый абзац. </P>
</body >
</html>

<HTML> <HEAD title >Фоновая графика</ title ></HEAD>
*<body background = picture.gif text = YELLOW>

*
<H1>Текст поверх фоновой графики</H1> </BODY> </HTML>

<HTML>
<HEAD><title>Пример использования ссылок</ title ></HEAD>
<BODY BGCOLOR=lightblue>
<IMG SRC=pict.gif ALIGN=bottom> Это простая картинка
<A HREF= "text.htm"> Это – текстовая ссылка</A>
<A HREF= "Путь к вашему рисунку"> <IMG SRC= "Путь к вашему рисунку"> </A>
А это – графическая ссылка </BODY>
</HTML>

<HTML><HEAD><TITLE>Ссылки в сложном документе</TITLE></HEAD>
<BODY BGCOLOR=lightblue>
<H2>Содержание</H2>
*<P><A HREF=#GL1>Глава1</A>
 (повторите тег
 несколько раз)*
*<A HREF=#GL2>Глава2</A>
 (повторите тег
 несколько раз)*
<H2> <A NAME=GL1></A> Глава1 Основы языка HTML</H2>
В этой главе рассматриваются основные элементы языка HTML
<H2><A NAME=GL2></A> Глава2 Примеры</H2>
*В этой главе рассматриваются примеры программ
 (повторите тег
 несколько раз)*

На оглавление</BODY> </HTML>

<HTML><HEAD><TITLE>Пример бегущей строки</TITLE> </HEAD><BODY>

<MARQUEE HEIGHT=50 WIDTH=75% HSPACE=5 VSPACE=5 ALIGN=TOP BGCOLOR=#FFOOOO DIRECTION=LEFT BEHAVIOR= SCROLL SCROLLDE-LAY=100> ПРИМЕР бегущей строки</ MARQUEE > </BODY></HTML>

4.5. Индивидуальное задание

Создайте документ №1, имеющий простую ссылку на документ №2.

Создайте документ №2, имеющий графическую ссылку на документ №3.

Создайте документ №3, имеющий ссылку на документ №1 в виде бегущей строки.

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. ТАБЛИЦЫ. ФРЕЙМЫ КАК СПОСОБ СТРУКТУРИРОВАНИЯ HTML-ДОКУМЕНТОВ

5.1. Общие сведения

Цель: научиться использовать элемент «Таблицы» в структурировании и оформлении HTML-документов; научиться использовать фреймы в HTML-документах, организовывать сложную логическую структуру HTML-документа, состоящего из нескольких HTML-страниц, устанавливая необходимые связи внутри и между отдельными страницами.

Материалы, оборудование, программное обеспечение: браузер Google Chrome, приложение «Блокнот».

Условия допуска к выполнению: показать конспект по теоретической подготовке.

Критерии положительной оценки: показать выполненную работу; ответить на вопросы преподавателя.

5.2. Теоретическое введение

<table> парный тег, создает таблицу;

атрибуты:

align=позиция – задается способ выравнивания таблицы относительно текстового потока, в котором она находится: ***left*** – по левому краю; ***right*** – по правому краю, ***center*** – по центру;

background=URL – задается фоновое изображение для таблицы;

bgcolor=цвет – задается цвет фона таблицы;

border=n – будет создано обрамление толщиной *n* пикселей;

cellpadding=n – между границами ячейки и ее содержимым поместить область свободного пространства шириной *n* пикселей;

cellspacing=n – задается интервал между ячейками таблицы размером в *n* пикселей;

hspace=n – задается размещение справа и слева от таблицы областей свободного пространства заданной ширины (в пикселях);

vspace=n – задается размещение над таблицей и под ней областей свободного пространства заданной ширины (в пикселях);

width=n – установка ширины таблицы в пикселях или в процентах от ширины окна;

<caption> Задается заголовок таблицы*</caption>*

атрибуты:

align =позиция – установка размещения заголовка по вертикали (*top* или *bottom*). По умолчанию – размещение сверху (*top*), по центру.

<tr>…</tr> парный тег; создает строки в таблице;

атрибуты:

align= left/right/center – задается способ выравнивания содержимого строки;

background= URL – задается фоновое изображение строки;

bgcolor=цвет – задается цвет фона строки;

bordercolor= цвет – задается цвет обрамления строки;

valign=mun – задается размещение содержимого данной строки: *top* – вверху; *center* – по центру; *bottom* – внизу; *baseline* – по базовой линии строки.

<td>…</td> парный тег, создает ячейку таблицы;

атрибуты:

align= left/right/center – задается способ выравнивания содержимого ячейки;

background= URL – задается фоновое изображение ячейки;

bgcolor=цвет – задается цвет фона ячейки;

colspan=n – данная ячейка объединяет *n* соседних столбцов;

rowspan=n – данная ячейка объединяет *n* соседних строк.

nowrap – при указании этого атрибута в данной ячейке отключается режим автоматического распределения текста (будет отображаться лишь та часть текста, которая умещается в ячейке по длине);

valign=mun – задается размещение содержимого данной ячейки: *top* – вверху; *center* – по центру; *bottom* – внизу; *baseline* – по базовой линии ячейки.

width=n – установка ширины данной ячейки в пикселях или в процентах от ширины окна.

<th>…</th> – парный тег, создает ячейку таблицы. Почти полная аналогия тега *<td>…</td>*, за исключением того, что тег *<th>…</th>* по умолчанию центрирует текст и выводит его полужирным шрифтом.

Фреймы (буквальный перевод с английского на русский язык – «кадры») позволяют наиболее логично структурировать HTML-документ, установить необходимые взаимосвязи между его частями и внести дополнительные удобства для пользователя при просмотре и навигации по сайту.

Структура документа с фреймами несколько отличается от обычного документа. Отсутствует тэг *<BODY>*, который заменяется тегами *<FRAMESET>…</FRAMESET>*. Атрибуты тэга *<FRAMESET>* описывают набор фреймов в целом, задавая размеры, внешний вид, рамки и т.д.

Между тэгами *<FRAMESET>…</FRAMESET>* находятся тэги *<FRAME>*, которые определяют параметры отдельных фреймов.

Атрибуты тега <FRAMESET>:

border = «цифра» – установка размера (в пикселях) обрамления кадров в наборе кадров (по умолчанию толщина линий обрамления равна 5 пикселям);

bordercolor = «цвет» – установка цвета обрамления для набора кадров;

cols = список значений – указание количества и ширины столбцов;

rows = список значений – указывается количество и высота строк.

frameborder = [yes/no] – включение или выключение отображения трехмерного или обычного обрамления кадров. По умолчанию – yes (трехмерное обрамление).

«Список значений» параметров *rows* и *cols* представляет собой разделенный запятыми список значений, которые могут задаваться в пикселях, процентах или в относительных единицах. Число строк или столбцов определяется числом значений в соответствующем списке.

Атрибуты тега <FRAME>:

bordercolor = «цвет» – установка цвета обрамления для набора кадров, если обрамление включено *атрибутом frameborder=yes;*

frameborder = [yes/no] – включение или выключение отображения трехмерного или обычного обрамления кадров (по умолчанию – *yes* (трехмерное обрамление));

marginheigth = «цифра» – задается размещение над содержимым кадра и под ним областей свободного пространства высотой по *n* пикселей;

marginwidth = «цифра» – задается размещение слева и справа от содержимого кадра областей свободного пространства высотой по *n* пикселей;

name = строка – задается имя кадра;

noresize – пользователю запрещается изменять размеры кадра;

scrolling = mun – линейка прокрутки будет добавляться всегда (*yes*), не будет добавляться (*no*), будет добавляться, если необходимо (*auto*);

src = “адрес источника” – задается URL исходного документа для данного кадра.

Запись

<FRAMESET ROWS=”300, 240, 100”>

определяет набор из трех фреймов. Эти значения – абсолютные в пикселях (сумма пикселей должна соответствовать разрешению экрана). Абсолютные значения не очень удобны из-за различных разрешений дисплеев. Лучшим вариантом будет задание значений в процентах (сумма процентов должна равняться 100) или относительных единицах. Например,

<FRAMESET ROWS=”25%, 50%, 25%”>.

Значения в относительных единицах:

<FRAMESET COLS=”, 2*, 3*”>*.

Звездочка используется для пропорционального деления пространства. Каждая звездочка – одна часть целого. Складывая все значения коэффициентов, стоящих у звездочек, получим знаменатель дроби. В нашем примере первый столбец занимает 1/6 часть от общей ширины окна, второй столбец – 2/6 (1/3), последний – 3/6 (1/2).

Приведем пример, использующий все три варианта задания значений:

*<FRAMESET COLS=”100, 25%, *, 2*”>.*

Здесь первый столбец имеет ширину 100 пикселей, второй столбец займет 25% от всей ширины окна, третий – 1/3 оставшегося пространства, последний – 2/3. Абсолютные значения рекомендуется назначать первыми по порядку слева направо, за ними следуют процентные значения от общего размера пространства, в заключение записываются значения, определяющие пропорциональное разбиение оставшегося пространства. Если в теге *<FRAMESET>* используются и *COLS,* и *ROWS,* то будет создана сетка из фреймов:

<FRAMESET ROWS=”, 2*, *” COLS=”2*, *”>.*

при подготовке ссылок на фреймы в якоре (тег *<A>*) необходимо использовать параметр *TARGET*, указывающий имя окна, в которое будет выполнена загрузка документа. Так, например, если необходимо, чтобы документ был загружен в окно с именем "*main page*", ссылка должна выглядеть подобным образом:

<A HREF="main.htm" TARGET="main page">Добро пожаловать</A>

<A HREF=адрес файла TARGET =имя фрейма>Текст ссылки</A> – для текстовых ссылок.

Если атрибут TARGET ссылается на blank, то документ всегда будет появляться в новом пустом окне. Имя self указывает на то, что выбранная страница загружается в тот же фрейм, где была активирована ссылка. При указании ссылки top документы появляются в отдельном окне вне фрейма. Браузер откроет новое окно для просмотра.

Литература:

Воробейкина, И.В. Технологии и методы программирования. Лабораторный практикум / И.В. Воробейкина. – Калининград: Изд-во БГАРФ, 2019. – 97 с.

Глава 1. Язык HTML.

5.3. Задание к лабораторной работе

– Охарактеризуйте роль и место элемента «Таблица» в создании HTML-страниц.

– Перечислите теги таблиц и их атрибуты.

– Создайте HTML-файл, содержащий описание деления окна на два вертикальных кадра: ширина левого кадра *400* пикселей. Цвет фона HTML-документа, загружаемого в левый кадр: *#FFFFCC* Цвет фона HTML-документа, загружаемого в правый кадр: *#CCFFFF*.

– Создать HTML-файл, содержащий описание деления окна на два горизонтальных кадра. Высота верхнего кадра *50%*. Цвет фона HTML-документа, загружаемого в верхний кадр*: #FFCCFF* Цвет фона HTML-документа, загружаемого в нижний кадр*: #CAFFCA.*

– Создать HTML-файл, содержащий описание деления окна на три кадра: левый кадр – зеленый, верхний правый кадр – синий, нижний правый кадр – красный.

5.4. Методические указания и порядок выполнения работы

Запишите следующие программы и изучите работу тегов:

Простая таблица

<TABLE BORDER> (это тег начала таблицы)

<CAPTION ALIGN=TOP> Простая таблица </CAPTION > (это заголовок таблицы)

<TR> (это тег строки таблицы)

<TH>Имя</TH><TH>Фамилия</TH><TH>Должность</TH> (это выделенная жирным шрифтом первая строка таблицы)

</TR> (далее тег <TD>.Он повторяется столько раз, сколько нужно ячеек.)

<TR>

<TD>Андрей</TD><TD>Андреев</TD><TD>Директор</TD>

</TR> <TR>

<TD>Максим</TD><TD>Максимов</TD><TD>Бухгалтер</TD>

</TR>

<TR>

<TD>Тихон</TD><TD>Тихонов</TD><TD>Менеджер</TD>

</TR> </TABLE>

Таблица с объединенными ячейками

<TABLE BORDER WIDTH= “75 %”>

<CAPTION ALIGN =TOP>Таблица с объединенными ячейками</CAPTION >

<TR>

<TH COLSPAN=2 WIDTH= “75 %”>Имя и Фамилия</TH> <TH>Должность</TH>

</TR>

<TR><TD>Иван</TD><TD>Андреев</TD><TD>Директор</TD></TR>
<TR><TD>Максим</TD><TD>Максимов</TD><TD>Бухгалтер</TD></TR>
<TR><TD>Павел</TD><TD>Титов</TD><TD>Менеджер</TD></TR>
</TABLE>

Использования цвета в таблицах

<TABLE BGCOLOR=#F0F0F0 BORDER=10 BORDERCOLOR="#808080" BORDERCOLORLIGHT=#707070 BORDERCOLORDARK=#202020>
<TR BGCOLOR=blue>
<TH>Имя</TH> <TH>Фамилия</TH>
</TR>
<TR><TD BGCOLOR=YELLOW>Андрей</TD><TD>Андреев</TD></TR>
</TABLE>

Работа с фреймами

Создадим два фрейма:

Файл main.htm

<HTML> <HEAD><title>Пример фреймов</ title ></HEAD>
<FRAMESET COLS="40%," BORDER=10>*
<FRAME SRC= "1.htm" NAME="Левый" SCROLLING=NO BORDER=10>
<FRAME SRC="2.htm" NAME="Правый"> </FRAMESET> </HTML>

Комментарий: звездочку (*) не забудьте (это вся оставшаяся свободная площадь окна).

Файл 1.htm

<HTML> <HEAD>
<title>Первый</title></HEAD><BODY text="#336699" bgcolor=navyblue> <H1>Это левый фрейм, который имеет ширину 40%</H1> </BODY> </HTML>

Файл 2.htm

<HTML> <HEAD>
<T1TLE>Второй</T1TLE>
</HEAD> <BODY bgcolor= lightblue text = darkblue >
<h2>Это правый фрейм, который занимает оставшуюся часть окна </h2></BODY> </HTML>

Внимание: для двух фреймов надо создать три файла, для трех – 4, и так далее.

5.5. Индивидуальное задание

Вариативность не предполагается.

1. Создайте следующие таблицы:

Фамилия, Имя, Отчество			год рождения	Должность
Иванов	*Иван*	*Иванович*	1935	Пенсионер
Петров	*Петр*	*Петрович*	1936	
Сидоров	*Сидор*	*Сидорович*	1937	

Чемпионат Электротехнического завода по футболу				
Название команды	*Ротор*	*Статор*	*Генератор*	**Место**
Ротор		1:0	2:0	**1**
Статор	0:1		0:0	**2**
Генератор	0:2	0:0		**3**

2. Создайте мини-сайт группы.

– Создать главную страницу, состоящую из 3 фреймов. В заголовке HTML-документа указать номер группы и свою фамилию.

– Создать страницу с таблицей «Преподаватели».

– Создать страницу «Новости».

– Создать страницу «О нас».

– На странице «Расписание» вынести список дней недели вверх и организовать ссылки на расписания конкретных дней недели (ссылки внутри документа).

– Организовать переход на страницы «Расписание», «Преподаватели», «Новости», «О нас» с главной страницы.

Образец страницы:

	Институт **"Калининградская Школа №2" Группа №"Автоматизированные системы обработки информации и управления"**

○ ***О нас*** ○ ***Расписание*** ○ ***Преподаватели*** ○ ***Новости*** ***Примечание: это гиперссылки на отдельные страницы***	***Примечание: эта часть окна обозревателя должна быть предназначена для отображения отдельных страниц, запускаемых из ссылок левого фрейма***

5.6. 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.

6. ЛАБОРАТОРНАЯ РАБОТА № 5. ДИАЛОГОВЫЕ ФОРМЫ В HTML-ДОКУМЕНТАХ

6.1. Общие сведения

Цель: получить навыки создания простейших диалоговых форм в HTML-документах. Научиться использовать диалоговые формы в оформлении HTML-страниц.

Материалы, оборудование, программное обеспечение: браузер Google Chrome, приложение «Блокнот».

Условия допуска к выполнению: показать конспект по теоретической подготовке.

Критерии положительной оценки: показать выполненную работу; ответить на вопросы преподавателя.

6.2. Теоретическое введение

Форма вводится тегом *<FORM>… </FORM>*. HTML-документ может содержать в себе сразу несколько форм, однако одна форма не может находиться внутри другой.

Атрибуты:

ACTION	Обязательный атрибут. Определяет, куда в Интернете форма пересылает данные (где находится обработчик формы).
METHOD	Определяет, каким образом данные из формы будут переданы обработчику. Допустимые значения: *METHOD=POST* и *METHOD=GET*. Если значение атрибута не установлено, по умолчанию предполагается *METHOD=GET*.
ENCTYPE	Определяет, каким образом данные из формы будут зашифрованы для передачи обработчику.
Пример № 1 *<FORM METHOD="POST" ACTION ="mailto: ADMIN@SHKOLA.RU">* форма что-то посылает на почтовый ящик администратора сайта.	

Теги полей диалоговых форм

Основной тег диалоговой формы – элемент *<INPUT>*. Этот тег имеет следующие **атрибуты**: *NAME* – имя, определяющий идентификатор данного поля (имя пересылаемой обработчику данных переменной), *VALUE* – значение данной переменной и *TYPE* – вид данного поля. Стандартом HTML определены следующие виды полей, приведенные в нижеследующей таблице (с примерами):

TYPE=text	Определяет окно для ввода строки текста. Может содержать дополнительные атрибуты SIZE=число (ширина окна ввода в символах) и MAXLENGTH=число (максимально допустимая длина вводимой строки в символах)
<INPUT TYPE=text SIZE=20 NAME=User VALUE="ШКОЛА"> Определяет окно шириной 20 символов для ввода текста. По умолчанию в окне находится текст ***ШКОЛА***, который пользователь может изменить	
TYPE=password	Определяет окно для ввода пароля. Аналогичен типу *text,* только вместо символов вводимого текста показывает на экране звездочки (*).
<INPUT TYPE=password NAME=PW SIZE=20 MAXLENGTH=10> Определяет окно шириной 20 символов для ввода пароля. Максимально допустимая длина пароля – 10 символов.	
TYPE=radio	Определяет радиокнопку, т.е. кружок, куда можно поставить галочку/точку. Может содержать необязательный атрибут *CHECKED* (показывает, что галочка/точка установлена). В группе радиокнопок с одинаковыми именами может быть только одна помеченная радиокнопка:
<INPUT TYPE=radio NAME=Question VALUE="Yes" CHECKED> Да<INPUT TYPE=radio NAME=Question VALUE="No"> Нет<INPUT TYPE=radio NAME=Question VALUE="Possible"> Возможно	
TYPE=checkbox	Создает квадрат, в котором можно поставить галочку. Необязательный атрибут *CHECKED* может быть использован сразу для всех квадратов, т.е. все они могут быть предварительно выбраны, в отличие от радиокнопок.
<INPUT TYPE=checkbox NAME=year VALUE="1"> Первый<INPUT TYPE=checkbox NAME= year VALUE="2" CHECKED>Второй<INPUT TYPE=checkbox NAME= year VALUE="3"> Третий <INPUT TYPE=checkbox NAME= year VALUE="4" CHECKED>Четвертый	
TYPE=hidden	Определяет скрытый элемент данных, который не виден пользователю при заполнении формы и передается обработчику без изменений. Такой элемент иногда полезно иметь в форме, которая время от времени подвергается переработке, чтобы обработчик мог знать, с какой версией формы он имеет дело

<INPUT TYPE=hidden NAME=version VALUE="1.1"> Определяет скрытую переменную version, которая передается обработчику со значением *1.1*.	
TYPE=submit	Определяет кнопку, при нажатии на которую запускается процесс передачи данных из формы обработчику.
<INPUT TYPE=submit VALUE="Отправить"> – Надпись на кнопке: «Отправить»	
TYPE=reset	Определяет кнопку, при нажатии на которую очищаются поля формы. Поскольку при использовании этой кнопки данные обработчику не передаются, кнопка типа reset может и не иметь атрибута name
<INPUT TYPE=reset VALUE="Сброс "> – Надпись на кнопке: «Сброс»	
TYPE=file	Позволяет «прикреплять» к пересылаемым формой данным некоторый файл. Определяет поле, в котором содержится адрес прикрепляемого файла и кнопку, при нажатии на которую появляется диалог операционной системы «Выбор файла». Имеет атрибут *name.*

Формы также могут содержать поля для ввода большого текста, описываемые тегом *<TEXTAREA>…</TEXTAREA>*. Тег имеет следующие атрибуты: *NAME* **–** имя переменной, *ROWS* – устанавливает высоту окна в строках, *COLS* – устанавливает ширину окна в символах. Текст, размещенный между тегами *<TEXTAREA>…</TEXTAREA>,* представляет собой содержимое окна по умолчанию. Пользователь может его отредактировать или просто стереть.

<TEXTAREA rows=5 cols=30> Здесь расположен предварительно размещенный текст</TEXTAREA> **–** высота данного поля в примере будет в 5 строчек, а ширина – 30.

Следующий элемент формы – меню выбора, вводимое тегами *<SELECT>…</SELECT>* (естественно, с атрибутом *NAME***)**. Между ними находятся теги *<OPTION>,* определяющие элемент меню. Обязательный атрибут *VALUE* устанавливает значение, которое будет передано обработчику, если выбран этот элемент меню. Тег *<OPTION>* может включать атрибут *SELECTED***,** показывающий, что данный элемент *выбран* / *отмечен* по умолчанию.

<SELECT NAME="имя"> <OPTION VALUE="option_1" selected>текст 1 <OPTION VALUE="option_2">текст 2<OPTION VALUE=" option_n">текст n</SELECT>

Тег *<SELECT>* может также содержать атрибут *MULTIPLE,* присутствие которого показывает, что из меню можно выбрать несколько элементов. Боль-

шинство браузеров показывают меню *<SELECT MULTIPLE>* в виде окна, в котором находятся элементы меню. Высоту окна в строках можно задать атрибутом *SIZE,* определяющим количество строк.

<SELECT MULTIPLE SIZE=3 NAME="имя"><<OPTION VALUE="option_1" selected>текст 1 <OPTION VALUE="option_2">текст 2<OPTION VALUE=" option_n">текст n</SELECT>

Напоминаем, что все вышеприведенные теги примеров должны находиться внутри контейнера *<FORM>… </FORM>*.

Литература:

Воробейкина, И.В. Технологии и методы программирования. Лабораторный практикум / И.В. Воробейкина. – Калининград: Изд-во БГАРФ, 2019. – 97 с.

Глава 1. Язык HTML.

6.3. Задание к лабораторной работе

– Для чего используется диалоговая форма?

– Какой вид имеют данные, пересылаемые диалоговой формой?

– Где и как обрабатываются данные, пересылаемые диалоговой формой?

– Выполните примеры из п. 6.4.

6.4. Методические указания и порядок выполнения работы

Запишите следующие программы и изучите работу тегов:

Создать файлы по предложенным примерам:

№ 1. Форма с полями для ввода текста

```
< Html ><HEAD><title>Пример ввода текста</title></head>
<body text="#000000" bgcolor="#D6FDD5">
<form>
<center><table BORDER=0 WIDTH="96%" >
<tr><td><div align=right>Имя:</div></td>
<td><input type="text" maxlength=10 name="firstname" size=10></td></tr>
<tr><td><div align=right>Фамилия:</div></td>
<td><input type="text" maxlength=20 name="lastname" size=20></td></tr>
</table></center>
</form>
</body></html>
```

№ 2. Использование кнопки для очистки поля ввода текста

```
<html><head><title>Пример очистки формы для элемента ввода текста</title></head>
<body text="#000000" bgcolor="#D6FDD5">
<form>
<center><table BORDER=0 WIDTH="96%" >
<tr><td><div align=right>Cтрана : </div></td>
<td><input type="text" maxlength=20 name="firstname" size=20></td>
    <td><div align=right>Город : </div></td>
    <td><input type"text" maxlength=20 name="lastname" size=20></td></tr>
</table>
<INPUT type="reset" value="Очистить форму"></center>
</form></body></html>
```

№ 3. Чекбокс

```
<html><head><title> Пример чекбокса</title></head>
<body text="#000000" bgcolor="#D6FDD5">
<form>
<center><table WIDTH="96%" BORDER="0" >
<tr><td><div align=right>Какие книги Вам нужны:</div>
</td><td><input type="checkbox" name="ch1" value="ich1"> Книга 1
<br><input type="checkbox" name="ch2" value="ich2"> Книга 2
<br><input type="checkbox" name="ch3" value="ich3"> Книга 3</td>
</tr></table></center>
</form></body></html>
```

№ 4. Меню выбора из выпадающего списка

```
<html><head><title>Пример выбора из списка</title></head>
<body text="#000000" bgcolor="#D6FDD5">
<center><form><b>Шахматы</b>
<br>Какими фигурами предпочитаете играть? 
<select size=1 name="color"><option>белыми</option><option>черными</option></select>
</form></center></body></html>
```

№ 5. Радиокнопки

```
<html><head><title>Примеры радиокнопок</title></head>
<body text="#000000" bgcolor="#D6FDD5">
<form>
<H2>какой язык вы используете?
```

<UL>

*<input type="RADIO" name="LANG" value="Русский" checked>Русский
*

*<input type="RADIO" name="LANG" value="Английский">Английский
*

<input type="RADIO" name="LANG" value="Немецкий">Немецкий</tr>

</UL>

</H2>

</form></body></html>

№ 6. Форма, отправляющая некоторую информацию на сервер

<HTML><HEAD><TITLE>Форма</TITLE></HEAD>

<BODY>

<H1 align="center">Почтовая форма</H1>

<hr width=75%>

* <u>Пожалуйста сообщите нам, что вы думаете.....тра ля ля ла.....</u>*

<b>Если вы сообщите</B> нам свою контактную информацию,

*у нас будет возможность связаться с вами и ответить на ваши вопросы. *

<FORM METHOD="POST" action="mailto: ADMIN@KVSHU.RU">

<H3 align="center">Контактная информация</H3>

<TABLE BORDER="0" align="center">

<TR><TD ALIGN="right">Имя</td><TD><input type="text" size="50%" name="Name"> </td> </TR>

<TR><TD ALIGN="right">Тема</td><TD><input type="text" size="50%" name="Title"> </td> </TR>

<TR><TD ALIGN ="right">Адрес</td><TD><input type="text" size="50%" name="Address"> </td> </TR>

<TR><TD ALIGN ="right">Телефон</td><TD><input type="text" size="50%" name="Telephone"></td></TR>

</TABLE>

* <input type="submit" value="Послать сообщение"><input type="reset" value="Очистить форму"> *

</FORM> </BODY> </HTML>

6.5. Индивидуальное задание

Вариативность не предполагается.

Выполнить лабораторные работы 1-2, проверить работу тегов. На основе полученных знаний выполнить лабораторную работу 3.

Л.р. 1.

```
<html>
<head>
<title> ПРИМЕР ФОРМЫ (ИНТЕРНЕТ-МАГАЗИН)</title>
</head>
<body>
<h2>(ИНТЕРНЕТ-МАГАЗИН)</h2>
<br>
<form>
<pre>
РЕГИСТРАЦИОННОЕ ИМЯ <input type="text" name="regname">
ВВЕДИТЕ ПАРОЛЬ <input type="password" name="password" maxlength=8>
ПОДТВЕРДИТЕ ПАРОЛЬ <input type="password" maxlength=8>
</pre>
СКОЛЬКО КИЛОГРАММОВ КОНФЕТ ВЫ ХОТИТЕ ПОЛУЧИТЬ:
<input type="radio" name="ves" value="lt1" checked> ДО 1 КГ
<input type="radio" name="ves" value="10-15" checked> > 10-15
<input type="radio" name="ves" value="l5-30" checked> > 15-30
<input type="radio" name="ves" value="gt100" checked> > БОЛЬШЕ 100
<br>
СОРТ КОНФЕТ
<input type="checkbox" name="konfeta" value="shokolad"> ШОКОЛАД-
НЫЕ
<input type="checkbox" name="konfeta" value="zhele"> ЖЕЛАТИНКИ
<input type="checkbox" name="konfeta" value="karamel"> КАРАМЕЛЬ
<input type="checkbox" name="konfeta" value="iris"> ИРИС
<br>
УПАКОВКА
<select name="upakovka" size=2>
<option selected value="kulek"> КУЛЕК
<option value="paket> ПАКЕТ
<option value="meshok"> МЕШОК
<option value="yashik"> ЯЩИК
</select>
<br>
ВАША ЛЮБИМАЯ КОНДИТЕРСКАЯ ФАБРИКА
<textarea name="wish" cols=40 rows=3></textarea>
<br><br>
<input type="submit" value="OK">
```

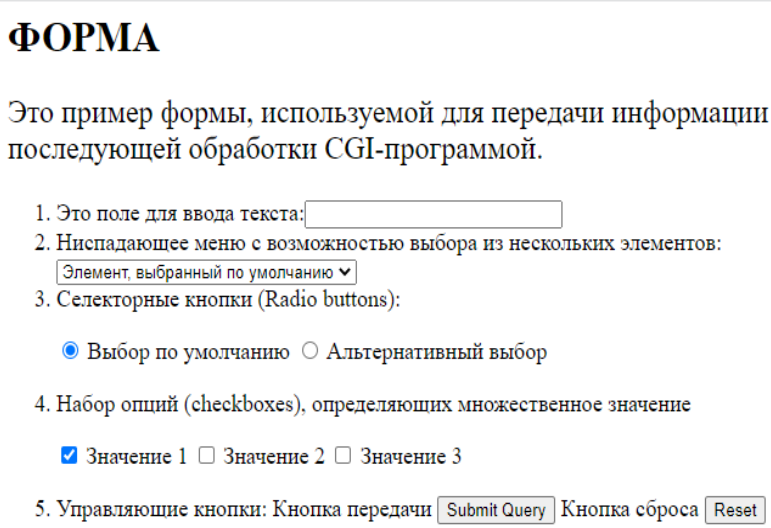
```
<input type="reset" value="OTMENIT">
</form>
</body>
</html>
```

Л.р. 2.

```
<html>
<head>
<title> форма </title>
</head>
<body>
<h1 align="center"> Почтовая форма </h1>
<hr width=75%>
<p align="left"><i><u> Пожалуйста сообщите нам, что вы думаете...</u>
<b> Если вы сообщите </b> нам свою контактную информацию,
у нас будет возможность связаться с вами и ответить на ваши вопросы.
</i></p>
<form method="post" action="mailto: alex@newmail.ru"> <!-- Посылаем что-то на
мой почтовый ящик.//-->
<h3 align="center"> Контактная информация </h3>
<table border="0" align="center">
<tr><td align="right"><i> Имя </i></td><td><input type="text" size="50%" name="name"></td></tr>
<tr><td align="right"><i> Тема </i></td><td><input type="text" size="50%" name="title"></td></tr>
<tr><td align="right"><i> Компания </i></td><td><input type="text" size="50%" name="company"></td></tr>
<tr><td align="right"><i> Адрес </i></td><td><input type="text" size="50%" name="address"></td></tr>
<tr><td align="right"><i> Телефон </i></td><td><input type="text" size="50%" name="telephone"></td></tr>
</table><!--Тег//-->
<p align="center">
<input type="submit" value="Послать сообщение">
<input type="reset" value="Очистить форму"></p>
</form>
</body>
</html>
```

Л.р. 3.

Восстановите программный код рисунка:



6.6. 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.

7. ЛАБОРАТОРНАЯ РАБОТА № 6. КАСКАДНЫЕ ТАБЛИЦЫ СТИЛЕЙ

7.1. Общие сведения

Цель: освоить использование каскадных таблиц стилей в оформлении HTML-страниц.

Материалы, оборудование, программное обеспечение: браузер Google Chrome, приложение «Блокнот».

Условия допуска к выполнению: показать конспект по теоретической подготовке.

Критерии положительной оценки: показать выполненную работу; ответить на вопросы преподавателя.

7.2. Теоретическое введение

Стиль – набор *правил* оформления и форматирования, который может быть применен к различным элементам страницы (цвет, шрифт, размер и прочее). Эти *правила* можно создавать и хранить:

1. Внутри HTML-тега; они применяются только для данного элемента документа и называются *«inline-правила».* Используется дополнительный атрибут – **style** для изменяемого тега, например: *<H1 style= "color: red" >…</ H1>* (будет применен красный цвет ко всему содержимому данного контейнера).

2. В заголовке документа, внутри тега *<HEAD>*, для применения по всей странице – это *«встроенные правила».* Конструкция:

<HEAD>… <STYLE type="text/css"> ***"здесь размещается описание нашего стиля"*** *</STYLE>…</HEAD>*.

Значение *"text/css"* является зарезервированным указанием браузеру о том, что будут применены каскадные таблицы стилей, оно является обязательным и пропускать его нежелательно.

3. Во внешнем файле, из которого эти правила – «*внешние»* и будут импортированы. Конструкция: *<LINK REL = STYLESHEET TYPE = "text/css" HREF= «**адрес внешнего файла с описанием стилей**»>;* обычно также размещается в заголовке документа, после тега *<HEAD>*. Адрес, или *URL*, может содержать как полный путь к файлу – особенно, если он находится на другом Интернет-ресурсе, или относительный, если он размещен на одном сервере с нашим документом.

Типы стилей

Существует несколько типов CSS-правил. Самый простой способ присвоения какому-либо элементу определенного стиля – ***HTML-селектор*** выглядит так:

НАЗВАНИЕ_ЭЛЕМЕНТА {свойство: значение;}

где *НАЗВАНИЕ_ЭЛЕМЕНТА* – так называемый ***«селектор»*** – имя HTML-тега (*H1, P, TD, A* и пр.), а параметры в фигурных скобках – ***«определение»,*** описывающее качество вводимого элемента (например, цвет) присвоенные ему значения (ключевое слово – *“yes”/ “no”*,число или проценты).

H1 {font-size: 30pt; color: blue;} – здесь всем заголовкам на странице, оформленным тегом *h1*, присваивается размер шрифта *30* пунктов и синий цвет.

Более сложный тип правил CSS – ***класс***. Он реализует возможность присваивать стили не всем одинаковым элементам страницы, а избирательно, лишь некоторым, несущим некоторую смысловую нагрузку. Для этого необходимо выполнить следующие действия, состоящие из двух шагов:

1. Создается описание стиля: ●имя_стиля {свойство: значение;} (*Обратите внимание на наличие точки перед описанием стиля*).

2. Как атрибут тега, к которому нужно применить данный стиль, используется следующая конструкция: *CLASS = “имя стиля”* (уже без точки, но в прямых кавычках).

Первый шаг:

.b-c {font-weight: bold; text-align: center} – описание стиля для класса *b-c.*

Все элементы класса *b-c* будут отображаться жирным шрифтом с выравниванием по центру страницы (или ячейки таблицы).

Второй шаг:

<P CLASS="b-c">содержание параграфа</P> – данному параграфу присвоен стиль класса *b-c.*

<td CLASS="b-c">текст ячейки</td> – ячейке присвоен стиль класса *b-c.*

Реже распространенный вид стилевого правила – *«идентификатор».* Идентификатор используется, когда требуется определить стиль только для одного элемента HTML-страницы; чаще всего для использования с языком JavaScript.

1. Первый шаг: создается описание стиля, например***:*** *#object1 {font-weight: bold; text-align: center}* (аналогично описанию класса)

2. Второй шаг: элементу присваивается уникальное имя – *.ID= «имя элемента»*, (У нас это будет “*object1*”).

Второй шаг – *<P id = "object1">* данному параграфу присвоен стиль *object1.*

Обратите внимание, что при написании названия классов и идентификаторов необходимо соблюдать регистр символов, согласно тому, как вы их назовете в описании стиля, т.е. *object* и *Object* **–** разные элементы.

Свойства элементов, управляемых с помощью CSS

Параметры шрифта:

font-weight: [bold | normal | number] – жирность шрифта;

font-style: [normal | italic | oblique] – наклон шрифта;

font-size: number – размер шрифта;

font-family: name – гарнитура шрифта(arial, times и прочие шрифты);

color: number – цвет шрифта.

P {font-family: Times New Roman, sans-serif;font-weight: bolder;font-size: 400pt;}

Параметры абзаца

text-align: [left | right | center | justify] – выравнивание абзаца;

text-indent: number – отступ красной строки;

line-height: number – абзацный отступ – расстояние между строками;

letter-spacing: number – отступ между буквами;

padding-left: number – отступ от текста слева;

padding-right: number – отступ от текста справа;

padding-top: number – отступ от текста сверху;

padding-bottom: number – отступ от текста снизу;

margin-left: number – отступ от границы слева, левое поле;

margin-right: number – отступ от границы справа, правое поле;

margin-top: number – отступ от границы сверху, верхнее поле;

margin-bottom: number – отступ от границы снизу, нижнее поле.

P {text-align: justify; text-indent: 50pt; line-height: 50 %; margin-bottom:2cm}

Совет: в Microsoft Word щелкните мышкой на меню *Формат→Абзац→Отступы и интервалы→ Положение на странице.*

Параметры внедренной картинки (обоев)

background-color: #number или соотв. словом (*red, blue*) – цвет заливки;

background-image: url («адрес рисунка фона») файл обоев, обязательно в скобках!

background-repeat: no-repeat – рисунок не повторять;

background-repeat: repeat-y, repeat-x – повторять по вертикали, горизонтали;

background-position: 50% 50% – пример размещения ровно посередине окна;

background-attachment: [fixed | scroll] – не перемещать | перемещать фон.

Выделение текста подчеркиванием

text-decoration:

underline – подчеркивание линией снизу;

overline – черта сверху;

line-through – перечеркивание.

Единицы измерения в CSS

В свойствах, которым требуется указание размеров, можно использовать несколько способов для их задания:

относительный размер ***в процентах*** *(%);*

относительный размер ***при помощи словесного описания*** *(larger, smaller, xx-small, x-small, small, medium, large, x-large, xx-large);*

абсолютный размер ***в типографских единицах*** – размер может задаваться в пунктах (*pt*), пикселах (*pc*), средней шириной буквы "*m*" (*em*), средней шириной буквы "*x*" (*ex*)

абсолютный размер ***в стандартных единицах длины*** – размер может задаваться в сантиметрах (*cm*), миллиметрах (*mm*), дюймах (*in*), а также в пикселях (*px*).

Браузеры могут менять по команде пользователя размер отображаемых шрифтов на Web-страничке – но только тогда, когда стили оформления используют относительные размеры. Если определены абсолютные размеры шрифтов и прочих элементов оформления HTML-страниц, то они будут именно такого размера, который указан, и не будут изменяться, например, при увеличении величины окна браузера.

Поэтому опытный дизайнер чаще всего использует для основного текста относительные величины, применяя абсолютные значения только для дополнительных элементов.

Задание цвета в CSS

Цвет для тех свойств, где это нужно, может быть определен одним из трех способов:

1. при помощи указания значения цвета английским словом – *color: yellow (red, green, grey, silver* и т.д.);
2. шестнадцатеричным заданием цвета в формате *#RRGGBB* – ***color: #FF00EA*** *(#883490, #0001EC* и пр);
3. десятичным заданием составляющих цвета в формате *rgb(red, green, blue)* **– *color:rgb(255,0,0)*** **(**или *rgb(100,23,78)***)**.

Комментарии в CSS

Существует также правило хорошего тона – желательно скрыть от старых браузеров, не умеющих отражать стили (даже если ими сейчас и мало кто пользуется), описания стилей. Для этого необходимо заключить описания стилей в тег комментариев. Делается это таким образом:

<HEAD><STYLE type="text/css"><!-- описание стилей --></STYLE></HEAD>

Устаревшие браузеры посчитают все заключенное между тегами комментариев информацией, не требующей отображения, а современные браузеры определят, что это – описание стилей, и задействуют их в оформлении документа.

Другой способ записать в теле документа некоторые неотображаемые браузером сведения – использовать следующую конструкцию: ***/* Этот текст является комментарием */***

Литература:

Воробейкина, И.В. Технологии и методы программирования. Лабораторный практикум / И.В. Воробейкина. – Калининград: Изд-во БГАРФ, 2019. – 97 с.

Глава 1. Язык HTML.

7.3. Задание к лабораторной работе

– Расскажите о трех способах задания стилей.
– Расскажите о типах стилей.
– Единицы измерения в CSS.
– Как можно создать комментарии к HTML-документу, не отображаемые браузером?
– Выполните примеры из п. 7.4.

7.4. Методические указания и порядок выполнения работы

Исходный код документа

```
<html>
 <head>
  <title>Стилевое оформление</title>
  <meta charset="utf-8">
  <link rel="stylesheet" href="style.css">
 </head>
 <body>
  <h1>Заголовок 1</h1>
  <p>Некоторый текст.</p>
 </body>
</html>
```

Обратите внимание на строку <link rel="stylesheet" href="style.css">. Она ссылается на внешний файл с описанием стилей под именем style.css. Содержимое этого файла показано в следующем примере.

```
body {
  font-family: Arial, Verdana,  sans-serif; /* Семейство шрифтов */
  font-size: 11pt; /* Размер основного шрифта в пунктах  */
```

```
  background-color: #f0f0f0; /* Цвет фона веб-страницы */
  color: #333; /* Цвет основного текста */
}
h1 {
  color: #a52a2a; /* Цвет заголовка */
  font-size: 24pt; /* Размер шрифта в пунктах */
  font-family: Georgia, Times, serif; /* Семейство шрифтов */
  font-weight: normal; /* Нормальное начертание текста  */
}
p {
  text-align: justify; /* Выравнивание по ширине */
  margin-left: 60px; /* Отступ слева в пикселах */
  margin-right: 10px; /* Отступ справа в пикселах */
  border-left: 1px solid #999; /* Параметры линии слева */
  border-bottom: 1px solid #999; /* Параметры линии снизу */
  padding-left: 10px; /* Отступ от линии слева до текста  */
  padding-bottom: 10px; /* Отступ от линии снизу до текста  */
}
```

В файле style.css описаны все параметры оформления таких тегов как **<body>**, **<h1>** и **<p>**. Сами теги в коде HTML пишутся как обычно.

Поскольку на файл со стилем можно ссылаться из любого веб-документа, это приводит в итоге к сокращению объема повторяющихся данных. А благодаря разделению кода и оформления повышается гибкость управления видом документа и скорость работы над сайтом.

7.5. Индивидуальное задание

Вариативность не предполагается.

Задание № 1

– Создайте файл *style.css*

– Подключите его к своим страницам (см. способ 3 задания стилей).

– Задайте по умолчанию следующие параметры для всех страниц (переопределив тег *<body>*):

цвет фона;

размер шрифта;

цвет шрифта;

семейство шрифта (например, *Arial*).

– В комментариях (в файле *style.css*) поясните эти параметры.

– Сохраните созданные файлы в своей папке на сервере аудитории, создав для них отдельный подкаталог «Задание1».

Примечание 1. Внешний файл может иметь любое имя, в том числе с использованием символов кириллицы. Расширение ***.css*** не является обязательным. Но практичнее всего сохранять такое расширение, а имена файлам стиля давать такие, какие позволят избежать путаницы и прямо укажут на свое содержание. Можно также создать несколько ссылок на нужные вам – и правильно названные – внешние файлы.

Примечание 2. В таком файле не должно быть ничего лишнего, особенно тега <*style*>. Иначе работать не будет!

Задание № 2

– Создайте описание нескольких классов в HTML-документе (см. способ 2 задания стилей).

– В описании классов переопределите цвет заливки ячейки и характер ее шрифтового оформления.

– Примените классы к нужным ячейкам.

– В комментариях к классам поясните эти параметры.

– Все сохранить в папке «Задание2».

Задание № 3

– Добавьте к заданию №2 единственный элемент со стилевым идентификатором.

– Опишите в комментариях отличие идентификатора от класса.

– Сохранить в папке «Задание3».

Задание № 4

– Добавьте к заданию №3 графическое изображение в виде обоев с троекратным повторением по вертикали и горизонтали

7.6. 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.

8. ЛАБОРАТОРНАЯ РАБОТА № 7. ФУНКЦИЯ И ОБРАБОТКА СОБЫТИЯ. ОРГАНИЗАЦИЯ ВЕТВЛЕНИЙ В ПРОГРАММАХ. ЦИКЛЫ

8.1. Общие сведения

Цель: познакомиться с основами языка JavaScript, организацией ветвления и циклов.

Материалы, оборудование, программное обеспечение: браузер Google Chrome, приложение «Блокнот».

Условия допуска к выполнению: показать конспект по теоретической подготовке.

Критерии положительной оценки: показать выполненную работу; ответить на вопросы преподавателя.

8.2. Теоретическое введение

Программа (сценарий) на языке JavaScript представляет собой последовательность операторов с «точкой с запятой» *(;)* между ними. Если каждый оператор размещается на одной строке, то разделитель можно не писать.

Простейшие данные, с которыми может оперировать программа, называются *литералами.*

Литералы не могут изменяться. Литералы целого типа могут быть заданы в десятичном (по основанию *10*), шестнадцатеричном (по основанию *16*) или восьмеричном (по основанию *8*) представлении. Шестнадцатеричные числа включают цифры *0-9* и буквы *a, b, c, d, e, f*. Шестнадцатеричные числа записываются с символами *0x* перед числом, например, *0x25*, *0xa1*,*0xff*. Запись вещественного литерала отличается от записи вещественного числа в математике тем, что вместо запятой, отделяющей целую часть от дробной, указывается точка, например, *123.34*, *-22.56*. Кроме того, для записи вещественных чисел можно использовать так называемую экспоненциальную форму.

Кроме целых и вещественных значений в языке JavaScript могут встречаться логические значения. В некоторых реализациях JavaScript может быть использована единица в качестве *true* и ноль в качестве *false*.

Строковый литерал представляется последовательностью символов, заключенной в одинарные или двойные кавычки. Примером строкового литерала может быть строка *"результат"* или *'результат'*.

Тип переменной зависит от хранимых в ней данных, при изменении типа данных меняется тип переменной. Определить переменную можно с помощью оператора *var*, например: *var test1*.

В данном случае тип переменной *test1* не определен и станет известен только после присвоения переменной некоторого значения. Оператор *var* можно использовать и для инициализации переменной, например, конструкцией *var test2=276* определяется переменная *test2* и ей присваивается значение *276*.

Пусть в сценарии описаны следующие переменные:

var n=3725
var x=2.75
var p=true
var s="Выполнение завершено"

n и *x* имеют тип *number,* тип переменной *p* – логический, переменная *s* имеет тип *string.*

Сценарии, написанные на языке JavaScript, могут располагаться непосредственно в HTML-документе между тегами <*script*> и </*script*>.

<*script language="JavaScript*>
…… </script>

Документ может содержать несколько тегов <*script*>. Все они последовательно обрабатываются интерпретатором JavaScript.

Основным элементом языка JavaScript является функция. Описание функции имеет вид

function F (V) {S},

где *F* – идентификатор функции, задающий имя, по которому можно обращаться к функции; *V* – список параметров функции, разделяемых запятыми; *S* – тело функции, в нем задаются действия, которые нужно выполнить, чтобы получить результат. Необязательный оператор *return* определяет возвращаемое функцией значение. Обычно все определения и функции задаются в разделе <*head*> документа. Это обеспечивает интерпретацию и сохранение в памяти всех функций при загрузке документа в браузер.

При интерпретации HTML-страницы браузером создаются объекты JavaScript. Взаимосвязь объектов между собой представляет иерархическую структуру. На самом верхнем уровне иерархии находится объект *windows*, представляющий окно браузера. Объект *windows* является «предком» или «родителем» всех остальных объектов. Каждая страница кроме объекта *windows* имеет объект *document*. Свойства объекта *document* определяются содержимым самого документа: цвет фона, цвет шрифта и т. д. Для получения значения основания треугольника, введенного в первом поле формы, должна быть выполнена конструкция

document.forml.stl.value

т.е. используем данные *value* из поля ввода с именем *st1* находящегося на форме *form1* объекта *document.*

Событие *Focus* возникает в момент, когда пользователь переходит к элементу формы с помощью клавиши <*Tab*> или щелчка мыши. Событие *«потеря фокуса» (Blur)* происходит в тот момент, когда элемент формы теряет фокус. Событие *select* вызывается выбором части или всего текста в текстовом поле.

Например, щелкнув дважды мышью по полю, мы выделим поле, наступит событие *select,* обработка которого приведет к вычислению требуемого значения.

Для организации ветвлений можно воспользоваться условным оператором, который имеет вид:

if B {S1} else {S2}

где *B* – выражение логического типа; *S1* и *S2* – операторы.

Для успешного решения широкого круга задач требуется многократно повторить некоторую последовательность действий, записанную в программе один раз. В том случае, когда число повторений последовательности действий нам неизвестно, либо число повторений зависит от некоторых условий, можно воспользоваться оператором цикла вида:

while (B) {s}

где *B* – выражение логического типа; *s* – операторы, называемые телом цикла. Операторы *s* в фигурных скобках выполняются до тех пор, пока условие *B* не станет ложным.

Если число повторений заранее известно, то можно воспользоваться циклом *for*, который часто называют оператором цикла арифметического типа. Синтаксис этого оператора таков:

for (A; B; I){S}

Выражение *A* служит для инициализации параметра цикла и вычисляется один раз в начале выполнения цикла. Выражение *B* (условие продолжения) управляет работой цикла. Если значение выражения ложно, то выполнение цикла завершается, если истинно, то выполняется оператор *S,* составляющий тело цикла. Выражение *I* служит для изменения значения параметра цикла. После выполнения тела цикла *S* вычисляется значение выражения *I,* затем опять вычисляется значение выражения *B* и т.д. Цикл может прекратить свою работу в результате выполнения оператора *break* в теле цикла.

Литература:

Воробейкина, И.В. Технологии и методы программирования. Лабораторный практикум / И.В. Воробейкина. – Калининград: Изд-во БГАРФ, 2019. – 97 с.

Глава 2. Язык JavaScript.

8.3. Задание к лабораторной работе

– Составить сценарий, в котором вычисляется площадь круга по заданному радиусу.

– Составить сценарий, вычисляющий гипотенузу по заданным катетам.

– Выполните примеры из п. 8.4.

8.4. Методические указания и порядок выполнения работы

Вычисление площади треугольника. Необходимо написать сценарий, определяющий площадь прямоугольного треугольника по заданным катетам. Сценарий разместим в разделе <*body*> HTML-документа.

Листинг *Первый сценарий в документе:*

```
 <HTML>
<HEAD>
<title>Первый сценарий в документе</title>
</HEAD>
<BODY>
<P>Страница, содержащая сценарий.</P>
<script>
<!--
var a=8; h=10 /*Инициализируются две переменные*/
document.write ("Площадь прямоугольного треугольника равна ", a*h/2,".") /*Для формирования вывода используется метод write объекта document*/
//-->
</script>
<P>Конец формирования страницы, содержащей сценарий</P>
</BODY>
</HTML>
```

В предыдущем примере пользователю не предоставлялась возможность вводить значения, и в зависимости от них получать результат. Интерактивные документы можно создавать, используя формы. Предположим, что мы хотим создать форму, в которой поля *Основание* и *Высота* служат для ввода соответствующих значений. Кроме того, в форме создадим кнопку *Вычислить*. При щелчке мышью по этой кнопке мы хотим получить значение площади треугольника. Действие пользователя (например, щелчок кнопкой мыши) вызывает событие. События в основном связаны с действиями, производимыми пользователем с элементами форм HTML. Обычно перехват и обработка события задается в параметрах элементов форм. Имя параметра обработки события начинается с приставки *on*, за которой следует имя самого события. Например, параметр обработки события *click* будет выглядеть как *onclick.*

Листинг *Реакция на событие Click.*

```
<HTML>
<HEAD>
<title>Обработка значений из формы</title>
```

```
<script language="JavaScript">
<!--//
function care (a, h)
{
var s=(a*h)/2;
document.write ("Площадь прямоугольного треугольника равна ",s);
return s
}
//-->
</script>
</HEAD>
<BODY>
<P>Пример сценария со значениями из формы</P>
<FORM name="form1">
Основание: <input type="text" size=5 name="st1"><hr>
Высота: <input type="text" size=5 name="st2"><hr>
<input type="button" value=Вычислить
onClick="care(document.form1.st1.value,  document.form1.st2.value)">  /*По клику мыши на кнопке в функцию care передаются два параметра - содержимое полей ввода*/
</FORM>
</BODY>
</HTML>
```

Нахождение максимального значения. Для трех заданных значений *a, b, c* необходимо написать сценарий, определяющий максимальное значение. Сначала максимальным значением *m* будем считать *a*, далее значение *b* сравним *c* максимальным. Если окажется, что *b* больше *m*, то максимальным становится *b*. И, наконец, значение *c* сравнивается с максимальным значением из *a* и *b*. Если *c* больше *m*, то максимальным становится *c*. Оператор присваивания *obj.res.value=m* обеспечивает запись вычисленного максимального значения в соответствующее поле формы. Функция *Number(s)* преобразует объект *s*, заданный в качестве параметра, в число.

Листинг *Вычисление максимального значения из трех заданных.*

```
<HTML>
 <HEAD>
 <TITLE>Вычисление максимального значения</TITLE>
 <script language="JavaScript">
 <!-- //
```

```
function maxval (obj )
{
var a = Number(obj.num1.value);
var b = Number(obj.num2.value);
var c = Number(obj.num3.value);
var m=a
if (b > m) m=b
if (c > m) m=c
obj.res.value=m }
//-->
</script>
</HEAD>
<BODY>
<H4>Вычисление максимального значения</H4>
<FORM name="form1">
Число 1: <input type="text" size=8 name="num1"><hr>
Число 2: <input type="text" size=8 name="num2"><hr>
Число 3: <input type="text" size=8 name="num3"><hr>
Максимальное значение равно
<input type="button" value=Определить onClick="maxval(form1)">
<input type="text" size=8 name="res"><hr>
<input type="reset">
</FORM>
</BODY>
</HTML>
```

Нахождение общего делителя. Напишем программу, которая для двух заданных чисел определяет наибольший общий делитель.

При решении задачи воспользуемся алгоритмом Евклида. Если значение *m* равно нулю, то наибольший общий делитель чисел *n* и *m* равен *n*: *НОД(n, 0) = n.* В остальных случаях верно следующее соотношение: *НОД(n, m) = НОД(m, n%m).*

В функции *nod* переменная *p* используется для получения остатка от деления чисел *n* и *m*. Выполнение цикла продолжается до тех пор, пока значение *p* не станет равным нулю. Последнее вычисленное значение *m* равно наибольшему общему делителю.

Листинг *Наибольший общий делитель двух чисел.*

```
<HTML>
<HEAD>
```

```
<TITLE>Наибольший общий делитель двух чисел</TITLE>
<script language="JavaScript">
<!-- //
function nod(obj)
{ var n=obj.num1.value
var m=obj.num2.value
var p = n%m
while (p!=0)
{ n=m
m=p
p=n%m
}
obj.res.value=m
}
//-->
</script>
</HEAD>
<BODY>
Наибольший общий делитель двух заданных чисел
<FORM name="form1">
Введите число <input type="text" name="num1" size="8"><br>
Введите число <input type="text" name="num2" size="8"><br>
<input type="button" value="Вычислить" onClick="nod(form1)"><br>
Наибольший общий делитель <input type="text" name="res"
size="8"><hr>
<input type="reset" value="Отменить">
</FORM>
</BODY>
</HTML>
```

Совершенные числа. Напишем программу, определяющую, является ли заданное число *n* совершенным.

Совершенным называется число *n*, равное сумме своих делителей, не считая самого числа. Например, число *6* является совершенным, т. к. верно *6 = 1 + 2 + 3*, где *1*, *2*, *3* – делители числа 6. Число 28 также является совершенным, справедливо *равенство 28 = 1 + 2 + 4 + 7 + 14*. При решении задачи воспользуемся только функцией *sumdel*.

Листинг *Итерационные методы. Совершенные числа.*

```
<HTML>
<HEAD>
```

```
<TITLE>Итерационные методы. Совершенные числа</TITLE>
<script language="JavaScript">
<!-- //
function sumdel(n)
{ var s=1;
for (var i=2; i<=n/2; i++)
{ if (n % i == 0) s += i }
 return s
}
function sov(obj)
{ var n=obj.numb.value;
var s=""
if (n==sumdel(n)) s="совершенное"
else s="не является совершенным"
return s
}
//-->
</script>
</HEAD>
<BODY>
<P> Итерационные методы. Совершенные числа</P>
<FORM name="form0">
Введите натуральное число: <input type="text" size=8 name="numb">
<input type="button" value=Выполнить on-
Click="this.form.res.value=sov(form0)"><hr>
Данное число: <input type="text" size=24 name="res"><hr>
<input type="reset" value=Отменить>
</FORM>
</BODY>
</HTML>
```

Обратите внимание на значение параметра обработки события. В данном случае это оператор присваивания, в правой части которого вызов функции *sov*.

Оператор *for...in* используется для анализа свойств объекта. Синтаксис оператора:

for (i in t) {s}

где *i* – переменная цикла; *t* – объект; *s* – последовательность операторов.

В результате выполнения оператора цикла производится перебор свойств объекта. Переменная цикла при каждом повторении содержит значение свойства объекта. Количество повторений тела цикла *s* равно числу свойств, определенных для объекта *t*.

8.5. Индивидуальное задание

Вариативность не предполагается.

1. Напишите сценарий, определяющий площадь квадрата по заданной стороне. Пусть форма содержит два текстовых поля: одно для длины стороны квадрата, другое для вычисленной площади. Кнопка *Обновить* очищает поля формы. Площадь квадрата вычисляется при возникновении события *change*, которое происходит в тот момент, когда значение элемента формы с именем *num1* изменилось, и элемент потерял фокус.

2. Напишите программу, определяющую все делители заданного натурального числа.

8.6. 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.

9. ЛАБОРАТОРНАЯ РАБОТА № 8. ОБРАБОТКА И ПРЕДСТАВЛЕНИЕ ДАТ. РАБОТА СО СТРОКАМИ

9.1. Общие сведения

Цель: познакомиться с предопределенными типами данных (дата).

Материалы, оборудование, программное обеспечение: браузер Google Chrome, приложение «Блокнот».

Условия допуска к выполнению: показать конспект по теоретической подготовке.

Критерии положительной оценки: показать выполненную работу; ответить на вопросы преподавателя.

Планируемое время выполнения: 3 ч.

Аудиторное время выполнения (под руководством преподавателя): 2 ч.

Время самостоятельной подготовки: 1 ч.

9.2. Теоретическое введение

Встроенный объект *Data* применяется для представления и обработки даты и времени. Он не имеет свойств, но обладает несколькими методами, позволяющими устанавливать и изменять дату и время. В языке JavaScript дата определяется числом миллисекунд, прошедших с 1 января 1970 года.

Объект *Data* создается оператором *new* с помощью конструктора *Data*. Если в конструкторе отсутствуют параметры, то значением *new Data()* будет текущая дата и время. Значением переменной *my_data1*, определенной следующим образом:

var my_data1 = new Data()

будет объект, соответствующий текущей дате и времени.

Параметром конструктора *new Data* может быть строка формата *«месяц, день, год часы: минуты: секунды».* Опишем переменную *my_data2* и присвоим ей начальное значение:

var my_data2 = new DataC'Fv, 12, 1978 16:45:10")

Переменная *my_data2* определяет *дату 12 февраля 1978 года и время 16 часов 45 минут и 10 секунд*. Значения часов, минут, секунд можно опустить, в этом случае они будут равны нулю:

var my_data3 = new Data("Feb, 12, 1978")

Параметры конструктора *new Data* могут определять год, месяц, число, время, минуты, секунды с помощью чисел. Дату *12 февраля 1978 года и время 16 часов 45 минут и 10 секунд* можно задать так:

var my_data4 = new Data(78, 1, 12, 16, 45, 10)

Если время опустить, то описание будет следующим:

var my_data5 = new Data(78, 1, 12)

Все числовые представления даты нумеруются с нуля, кроме номера дня в месяце. Месяцы представляются числами от *0* (январь) до *11* (декабрь*)*, поэтому второй параметр при задании переменных *my_data4 и my_data5* равен *1*.

Методами объекта *Date* можно получать и устанавливать отдельно значения месяца, дня недели, часов, минут и др.

Метод *getDate* возвращает число в диапазоне от *1* до *31*, представляющее число месяца.

Метод *getHours* возвращает час суток. Значение возвращается в *24*-часовом формате от *0* (полночь) до *23*.

Метод *getMinutes* возвращает минуты как целое от *0* до *59*.

Метод *getseconds* возвращает число секунд как целое от *0* до *59*.

Метод *getDay* возвращает день недели как целое число от *0* (воскресенье) до *6* (суббота).

Метод *getMonth* возвращает номер месяца в году как целое число в интервале между *0* (январь) и *11* (декабрь). Обратите внимание, что номер месяца не соответствует стандартному способу нумерации месяцев.

Метод *getYear* выдает год объекта.

Метод *setYear* устанавливает значение года для объекта *Date.*

Метод *setDate* устанавливает день месяца. Параметр должен быть числом в диапазоне от *1* до *31*.

Метод *setMonth* устанавливает значение месяца. Параметр должен быть числом в диапазоне от *0* (январь) до *11* (декабрь).

Метод *setHours* устанавливает час для текущего времени, использует целое число от *0 (полночь)* до *23* для установки даты по *24*-часовой шкале.

Метод *setMinuts* устанавливает минуты для текущего времени, использует целое число от *0* до *59*.

Метод *setSeconds* устанавливает секунды для текущего времени, использует целое число от *0* до *59.*

Метод *setTime* устанавливает значение объекта *Date* и возвращает количество миллисекунд, прошедших с 1 января 1970 года.

Строковые литералы или строковые переменные являются в языке *JavaScript* объектом типа *string,* к которому могут быть применены методы, определенные в языке. Создание нового объекта требует вызова функции-конструктора объекта. Для того чтобы создать строковый объект, надо применить конструктор *newstring*, например:

s=newString("результат=")

Объект *string* имеет единственное свойство *length* (длина_строки). Выражение *s.length* выдает значение *10*, равное длине строки, содержащейся в стро-

ковом объекте *s*. Объект *string* имеет два типа методов. С методами, непосредственно влияющими на саму строку, мы сейчас и познакомимся, рассматривая примеры обработки текстовой информации.

Одним из часто используемых методов является метод выделения из строки отдельного символа. Метод *charAt(ni)* возвращает символ, позицию которого определяет параметр *ni*. Символы в строке перенумерованы, начиная с *0*.

Литература:

Воробейкина, И.В. Технологии и методы программирования. Лабораторный практикум / И.В. Воробейкина. – Калининград: Изд-во БГАРФ, 2019. – 97 с.

Глава 2. Язык JavaScript.

9.3. Задание к лабораторной работе

Выполните примеры из п. 9.4.

9.4. Методические указания и порядок выполнения работы

Определение текущего времени. Напишем сценарий, который определяет текущее время и выводит его в текстовое поле в формате *"чч:мм:сс"*.

В переменной *res* формируется строка, которая затем будет отображена в поле *rest* формы с именем *form1*. Для того чтобы уточнить время, следует еще раз нажать кнопку *Время* и т. д.

Листинг *Определение времени.*

```
<HTML>
<HEAD>
<TITLE>Определение времени</TITLE>
<script language="JavaScript">
<!-- //
function c1()
{ var d=document
var t=new Date()
var h=t.getHours()
var m=t.getMinutes()
var s=t.getseconds()
var res=""
if (h < 10)
res += "0" + h
else res += h
if (m < 10) res += ":0"+m
else res += ":"+m
if (s < 10) res += ":0"+s
```

```
else res += ":"+s
d.form1.rest.value = res
}
//-->
</script>
</HEAD>
<BODY bgcolor="#FFFFCC">
<CENTER>
<IMG src=alarmWHT.gif><br>
При нажатии кнопки <B>Время</B>, Вы узнаете, который час
<FORM name="form1">
<input type="button" value=Время onClick="c1()">
<input type="text" size=10 narae="rest"><br>
</FORM>
</BODY>
</HTML>
```

Можно сделать так, что через некоторый заданный период значение времени будет обновляться. Для этого можно использовать функцию *setTimeout ("c1()", 3000).* Функция *setTimeout* выполняет указанные в первом параметре действия по истечении интервала времени, задаваемого вторым параметром. В приведенном примере через три секунды будет снова осуществлен вызов функции *c1*.

Вывод символов строки в столбик. Напишем сценарий, при выполнении которого заданный текст выводится в столбик, т.е. на каждой строке размещается по одному символу.

При решении задачи из заданной строки последовательно выбираются символы. Формируется новая строка, в которой за каждым символом ставится последовательность символов, обеспечивающая переход на новую строку. Когда строка результата сформирована, то она размещается в текстовом поле формы, тем самым для исходной строки осуществляется вывод в столбик.

Листинг *Вывод символов строки в столбик.*

```
<HTML>
<HEAD>
<TITLE>Вывод символов строки в "столбик"</TITLE>
<script language="JavaScript">
<!-- //
function ttest(s)
{ var sres="Прочитанный текст:"+" \r\n"+s+"\r\n"+
'Текст в "столбик":'+"\r\n"
```

```
var cur=""
for ( var i=0; i <= s.length-1; i += 1)
{c=s.charAt(i); cur +=c+"\r\n" }
sres+=cur
return sres
}
//-->
</script>
</HEAD>
<BODY bgcolor="#FFFFCC">
<H4>Символы текущей строки в столбик</H4>
<FORM name="form1">
Введите строку: <input type="text" size=20 name="st1"><hr>
<input type="button" value=Выполнить on-
Click="form1.res.value=ttest(form1.st1.value)">
<input type="reset" value=Очистить><hr>
<textarea cols=20 rows=7 name= res></textarea>
</FORM>
</BODY>
</HTML>
```

Метод *substr (n1,n2)* позволяет выделять из строки подстроку. Параметр *n1* задает позицию первого символа подстроки; параметр *n2* определяет количество символов в подстроке. Например, если строка *s="сборник",* то в результате выполнения *substr (0,4)* будет выделена подстрока *"сбор".*

9.5. Индивидуальное задание

Вариативность не предполагается.

1. Напишите сценарий, с помощью которого определяются все даты в указанном году, приходящиеся на пятницу, *13* число.

Указание: перебирать с помощью цикла месяцы и в каждом месяце устанавливать номер дня *13*. Установка требуемой даты выполняется использованием методов работы с датой:

```
t.setYear (y)
 t.setMonth (i)
 t.setDate (13)
```

Далее следует проверить, какой номер дня соответствует этой дате. Если номер равен пяти *((t.getday())==5),* то день недели – пятница, найденный месяц следует запомнить. Для формирования ответа используется строковая переменная с, после запоминания названия месяца добавляется символ перевода строки.

2. Напишите программу, которая определяет, сколько раз заданное слово встречается в определенном тексте.

Указание: в тексте слова разделяются пробелами. После того как очередное слово найдено, просмотр продолжается с символа, следующего за найденным словом.

9.6. 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.

10. ЛАБОРАТОРНАЯ РАБОТА № 9. МАССИВЫ

10.1. Общие сведения

Цель: изучить работу с массивами в JavaScript.

Материалы, оборудование, программное обеспечение: браузер Google Chrome, приложение «Блокнот».

Условия допуска к выполнению: показать конспект по теоретической подготовке.

Критерии положительной оценки: показать выполненную работу; ответить на вопросы преподавателя.

10.2. Теоретическое введение

Массивы

Тип *Array* введен в *JavaScript* для возможности манипулирования самыми разными объектами, которые может отображать *Navigator*. Это список всех гипертекстовых ссылок данной страницы, список всех картинок на данной странице, список всех элементов формы и т.п. Пользователь может создать и свой собственный массив, используя, конструктор *Array().* Делается это следующим образом:

```
new_array = new Array()
new_array5 = new Array(5)
colors = new Array("red", "white", "blue")
```

Размерность массива может изменяться. Можно сначала определить массив, а потом присвоить одному из его элементов значение. Как только это произойдет, изменится и размерность массива:

```
colors = new Array()
colors[5] = "red".
```

В данном случае массив будет состоять из *6* элементов, так как первым элементом массива считается элемент с индексом *0*.

Для массивов определены четыре метода: *join, reverse, sort, concat. Join* объединяет элементы массива в строку символов, в качестве аргумента в этом методе задается разделитель:

```
colors = new Array("red", "white", "blue")
string = acolors.join(" + ")
```

В результате выполнения присваивания значения строке символов *string* мы получим следующую строку: *string = "red + white + blue"*. Другой метод, *reverse*, изменяет порядок элементов массива на обратный, метод *sort* отсортировывает их в лексикографическом порядке, а метод *concat* объединяет два массива.

У массивов есть два свойства: *length* и *prototype. Length* определяет число элементов массива. Если нужно выполнить некоторую рутинную операцию над всеми элементами массива, то можно воспользоваться циклом типа:

```
color = new Array("red", "white", "blue")
n = 0 while(n != colors.length)
{... операторы тела цикла ...}
```

Свойство *prototype* позволяет добавить свойства к объектам массива. Однако чаще всего в программах на *JavaScript* используются встроенные массивы, в основном графические образы *(Images)* и гипертекстовые ссылки *(Links).*

В новой версии языка появился конструктор для этого типа объектов:

```
new_image = new Image()
new_image = new Image(width, height)
```

Литература:

Воробейкина, И.В. Технологии и методы программирования. Лабораторный практикум / И.В. Воробейкина. – Калининград: Изд-во БГАРФ, 2019. – 97 с.

Глава 2. Язык JavaScript.

10.3. Задание к лабораторной работе

Выполните примеры из п. 10.4.

10.4. Методические указания и порядок выполнения работы

Создание мультипликации с использованием массивов. Часто для создания мультипликации формируют массив графических объектов, которые потом прокручивают один за другим:

```
img_array = new Array()
img_array[0] = new Image(50,100)
img_array[1] =new Image(50,100)
 ...
img_array[99] = new Image(50,100)
```

У объекта *Image* существует *10* свойств, из которых самым важным является *src*. Так, для присваивания конкретных картинок элементам массива *img_array* следует воспользоваться следующей последовательностью команд:

```
img_array[0].src = "image1.gif"
img_array[1].src = "image2.gif"
...
img_array[99].src = "image100.gif"
```

В данном случае можно было воспользоваться и циклом для присвоения имен, так как они могут быть составлены из констант и значения индексной переменной.

Построим документ, в котором будет встроена мультипликация, определенная массивом:

Листинг *Мультипликация.*

```
<HTML>
 <HEAD>
 <SCRIPT>
 <!--//
 function multi_pulti()
 {
 img_array = new Array()
 for (var i=0; i<4; i++)
 img_array[i]  = new Image(50,100)
 img_array[0].src = "e1.jpg"
 img_array[1].src = "e2.jpg"
 img_array[2].src = "e3.jpg"
 img_array[3].src = "e4.jpg"
 var t=new Date()
 p=-1
 }

 function s()
 {
 p=p+1
 document.images[0].src =img_array[p].src
 setTimeout("s()",100)
 if (p==3) p=-1
 }
 //-->
 </SCRIPT>
 </HEAD>
 <BODY onLoad="multi_pulti()">
 <img src="e1.jpg">
 <br>
 <input type="Button" name="But" value="Посмотреть" onClick="s()">
 </BODY>
 </HTML>
```

Далее рассмотрим несколько классических задач, посвященных работе с массивами. Приведем функции работы с массивами, которые ценны сами по себе и могут применяться в различных программах.

Бинарный поиск с формированием таблицы результатов. Напишем функцию, которая реализует алгоритм бинарного поиска таким образом, чтобы во время работы программы формировалась таблица значений переменных *i, j, k* и некоторых выражений.

Листинг *Поиск в упорядоченном массиве с таблицей промежуточных значений.*

```
 <HTML>  <HEAD>
<TITLE>Бинарный поиск. Таблица промежуточных значений</TITLE>
<script language="JavaScript">
<!-- //
var v=new Array (2, 3, 5, 6, 6, 7,10,11, 20, 25)
function testtab(obj,v,t)
{ var res="i j k v[k] t<= v[k]"+"\r\n"
var i=0
var j= v.length-1
var k
while ( i < j )
{ k=Math.round( (i+j)/2+0.5)-1
res = res + i + " "+j+" "+k+" "+"v[" + k + "]=" + v[k] + " " + t + "<=" +
v[k]+"\r\n"
if (t <= v[k] )
j=k
else
i=k+l
}
res += "v[" + i + "]=" +v[i]+"\r\n"
obj.resultl.value=res
if (v[i] == t )
{ return i}
else return -1
}
function test(obj)
{ obj.datal.yalue=v}
//-->
</script>
</HEAD>
<BODY bgcolor=silver>
```

<H4>Реализация алгоритма бинарного поиска</H4>

<FORM name="forml">

<pre>

Массив:<INPUT type="text" size=40 name="datal" ><hr>

Элемент:<INPUT type="text" size=20 name="data2" ><hr>

Результат поиска: <INPUT type="text" size=20 name="result" ><hr>

*Таблица промежуточных значений:
*

<textarea cols=50 rows=7 name="result1" > </textarea><hr>

</PRE>

<input type="button" value=0пределить onClick="test(form1); forml.result.value=testtab(form1,v,forml.data2.value)">

<input type="reset" value=Отменить>

</FORM>

</BODY>

</HTML>

10.5. Индивидуальное задание

Вариативность не предполагается.

1. Задан одномерный массив вещественных чисел. Напишите сценарий, который определяет число положительных элементов массива.

2. Задан одномерный массив вещественных чисел. Напишите сценарий, позволяющий найти максимальный элемент в массиве.

10.6. 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.

11. ЛАБОРАТОРНАЯ РАБОТА № 10. ПРОСТЫЕ ТИПЫ ДАННЫХ. АРИФМЕТИЧЕСКИЕ ВЫРАЖЕНИЯ. ОПЕРАТОРЫ ПРИСВАИВАНИЯ. ИНИЦИАЛИЗАЦИЯ ПЕРЕМЕННЫХ

11.1. Общие сведения

Цель: изучить простые типы данных в C++.

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

Условия допуска к выполнению: показать конспект по теоретической подготовке и решить подготовительный пример.

Критерии положительной оценки: показать выполненную работу: запрограммированный пример; ответить на вопросы преподавателя.

11.2. Теоретическое введение

Перед написанием любой программы надо четко определить, какие исходные данные в нее требуется ввести и что должно получиться в результате. Тип переменных выбирается исходя из возможного диапазона значений и требуемой точности представления данных. Например, нет необходимости объявлять переменную вещественного типа для величины, которая будет принимать целочисленные значения – это сэкономит память, да и операции над целыми числами выполняются быстрее. При неправильном выборе типов данных можно столкнуться с серьезными ошибками, которые приведут к неправильному результату. Часто начинающий программист тратит время на поиск ошибки в алгоритме программы, не удостоверившись, что программа работает с правильными данными. В черновом варианте программы для контроля исходных данных рекомендуется выводить их сразу после объявления или ввода, чтобы исключить ошибки. Самое важное качество любой программы – надежность, поэтому необходимо иметь тестовые примеры, содержащие исходные данные и ожидаемые результаты.

Допустимые имена для переменных

Идентификаторы переменных могут содержать в себе:

- латинские буквы;
- цифры;
- знак нижнего подчёркивания.

При этом название не может начинаться с цифр. Примеры названий:

- age;
- name;
- _sum;
- first_name;
- a1;
- a2;

- a_5.

Все идентификаторы регистрозависимы. Это значит, что *name* и *Name* — разные переменные.

Чаще всего используются следующие типы данных:

- **int** nnn – целое число;
- **byte** – число от 0 до 255;
- **float** – число с плавающей запятой;
- **double** – число с плавающей запятой повышенной точности;
- **char** – символ;
- **bool** – логический тип, который может содержать в себе значения true *(истина)* и false *(ложь)*.

Эти типы данных называются **примитивными** *(значимыми)*, потому что они базово встроены в язык.

Инициализация переменных. Если начальные значения присваиваются переменным (или константам) одновременно с их объявлением, то экономится и память, и время выполнения программы, т.к. в этом случае переменные получают начальные значения во время компиляции программы, а не во время ее выполнения. Заодно облегчается внутреннее документирование программы и обходятся ошибки в случае, когда переменной не было присвоено начальное значение.

Глобальные числовые переменные автоматически инициализируются нулем, локальные – не инициализируются; это необходимо учитывать при создании программы.

Литература:

Воробейкина, И.В. Технологии и методы программирования: учеб. пособие. Часть I / И.В. Воробейкина. – Калининград: Изд-во БГАРФ, 2021. – 108 с.

Контрольные вопросы для самопроверки:

1. Что нужно учитывать при выборе типа переменных?
2. Что такое область действия переменных?

11.3. Задание к лабораторной работе

Откомпилируйте программы из п. 11.4.

11.4. Методические указания и порядок выполнения работы

Рассмотрим программу, переводящей температуру по Фаренгейту в температуру по Цельсию:

```
#include <iostream.h>
```

```
int main()
{
float frg,cls;
cout<<"Введите температуру по Фаренгейту "; cin>>frg;
cls=5/9*(frg-32);
cout<<"Температура по Цельсию  "<<cls;
getchar();    //фиксация кадра
return 0;
}
```

Результат выполнения этой программы стабильно оказывается равным нулю. Это происходит из-за того, что константы 5 и 9 целочисленные, поэтому результат их деления целочисленный (дробная часть всегда отбрасывается). Чтобы исправить эту ошибку достаточно записать хотя бы одну из констант в вещественном виде, например:

cls=5/9.*(frg-32);

Напишите программу для расчета по двум формулам (результат вычисления по первой формуле должен совпадать со второй). С помощью калькулятора подготовьте тестовые примеры. Ввод с клавиатуры числа α предваряйте приглашением. В процессе отладки выводите исходные данные на дисплей. Подготовьте тестовые примеры. Проверьте реакцию программы на неверные исходные данные. При записи выражений обращайте внимание на приоритет операций. Данные при вводе разделяйте пробелами, символами перевода строки или табуляции. Для использования математических функций необходимо подключить заголовочный файл <math.h>. Число π в C++ определяется как M_PI.

$$z_1 = 2\sin^2(3\pi - 2\alpha)\cos^2(5\pi + 2\alpha)$$

$$z_2 = \frac{1}{4} - \frac{1}{4}\sin\left(\frac{5}{2}\pi - 8\alpha\right)$$

11.5. Индивидуальное задание

Напишите программу для расчета по двум формулам (результат вычисления по первой формуле должен совпадать со второй).

1.

$$z_1 = \cos\alpha + \sin\alpha + \cos 3\alpha + \sin 3\alpha$$

$$z_2 = 2\sqrt{2}\cos\alpha \cdot \sin\left(\frac{\pi}{4} + 2\alpha\right)$$

2.

$$z_1 = \frac{\sqrt{2b + 2\sqrt{b^2 - 4}}}{\sqrt{b^2 - 4} + b + 2}$$

$$z_2 = \frac{1}{\sqrt{b + 2}}$$

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. ОПЕРАТОРЫ ВВОДА-ВЫВОДА ЯЗЫКОВ С И С++. УСЛОВНЫЙ ОПЕРАТОР IF. ТЕРНАРНЫЙ IF. ОПЕРАТОР ВЫБОРА SWITCH

12.1. Общие сведения

Цель: изучить операторы ветвления в С++.

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

Условия допуска к выполнению: показать конспект по теоретической подготовке и решить подготовительный пример.

Критерии положительной оценки: показать выполненную работу: запрограммированный пример; ответить на вопросы преподавателя.

12.2. Теоретическое введение

Поточный ввод-вывод в С++ выполняется с помощью объектов *cout, cin*. В С для этих целей используется библиотека stdio.h. В С++ разработана новая библиотека ввода-вывода iostream, использующая концепцию объектно-ориентированного программирования:

#include <iostream>

Библиотека *iostream* определяет три стандартных потока:

- *cin* стандартный входной поток (*stdin* в С)
- *cout* стандартный выходной поток (*stdout* в С)
- *cerr* стандартный поток вывода сообщений об ошибках (*stderr* в С)

Для их использования в Microsoft Visual Studio необходимо прописать строку:

using namespace std;

Для выполнения операций ввода-вывода переопределены две операции поразрядного сдвига:

- >> получить из входного потока
- << поместить в выходной поток

Вывод информации

cout << значение;

Здесь значение преобразуется в последовательность символов и выводится в выходной поток:

cout<<n;

Возможно многократное назначение потоков: ***cout << 'значение1' << 'значение2' << ... << 'значение n';***

```
int n;
char j;
cin >> n >> j;
```

cout << "Значение n равно" << n << "j=" << j;

Ввод информации

cin >> идентификатор;

При этом из входного потока читается последовательность символов до пробела, затем эта последовательность преобразуется к типу идентификатора, и получаемое значение помещается в **идентификатор**:

int n;
cin >> n;

Возможно многократное назначение потоков:

cin >> переменная1 >> переменная2 >>...>> переменнаяn;

При наборе данных на клавиатуре значения для такого оператора должны быть разделены символами (*пробел, \n, \t*).

int n;
char j;
cin >> n >>j;

Особого внимания заслуживает ввод символьных строк. По умолчанию потоковый ввод *cin* вводит строку до пробела, символа табуляции или перевода строки.

Пример

#include <iostream>
using namespace std;
int main()
{
char s[80];
cin >> s;
cout << s << endl;
system("pause");
return 0;
}

Для ввода текста до символа перевода строки используется манипулятор потока *getline():*

#include <iostream>
using namespace std;
int main()
{
char s[80];
cin.getline(s,80);
cout << s << endl;
system("pause");

return 0;
}

Манипуляторы потока

Функцию-манипулятор потока можно включать в операции помещения в поток и извлечения из потока (<<, >>).

В C++ имеется ряд манипуляторов. Рассмотрим основные:

Манипулятор	**Описание**
endl	Помещение в выходной поток символа конца строки '\n'
dec	Установка основания 10-й системы счисления
oct	Установка основания 8-й системы счисления
hex	Установка основания 16-й системы счисления
setbase	Вывод базовой системы счисления
width(ширина)	Устанавливает ширину поля вывода
fill('символ')	Заполняет пустые знакоместа значением символа
precision(точность)	Устанавливает количество значащих цифр в числе (или после запятой) в зависимости от использования fixed
fixed	Показывает, что установленная точность относится к количеству знаков после запятой
showpos	Показывает знак + для положительных чисел
scientific	Выводит число в экспоненциальной форме
get()	Ожидает ввода символа
getline(указатель, количество)	Ожидает ввода строки символов. Максимальное количество символов ограничено полем количество

Пример

```
#include <iostream>
using namespace std;
int main()
{
int n;
cout << "Введите n:";
```

```
cin >> n;
cout << "Значение n равно:" << n << endl;
cin.get(); cin.get();
return 0;
}
```

Для обеспечения выполнения разных последовательностей операторов применяются операторы ветвления *if* или *switch*.

Операции отношений (<, >, ==, <=, >=, !=) являются бинарными и формируют результат типа *bool*, равный *true* или *false* (в компиляторах, не поддерживающих тип *bool*, истинным считается любое значение, не равное нулю). Когда необходимо, чтобы условия выполнялись одновременно, они объединены с помощью операции логического *И (&&)*. Одна из часто встречающихся ошибок – запись подобных условий в виде кальки с математических формул как *a<b<c*. Компилятор не выдает сообщений об ошибке, но результат получается неправильный.

Для присваивания какой-либо переменной в зависимости от выполнения условия двух различных значений можно воспользоваться не оператором *if*, а тернарной условной операцией, например:

```
if (x<0) y=0; else y=x;
y=(x<0) ? 0:x;
```

Первый операнд представляет собой выражение, результат вычисления которого преобразуется в *true* или *false*. После знака вопроса через двоеточие записываются два выражения. Результатом тернарной операции будет первое из них, если выражение в скобках принимает значение true, в противном случае результатом будет второе выражение.

Литература:

Воробейкина, И.В. Технологии и методы программирования: учеб. пособие. Часть I / И.В. Воробейкина. – Калининград: Изд-во БГАРФ, 2021. – 108 с.

Контрольные вопросы для самопроверки:

1. Для чего в операторе *switch* необходима ветвь *default*?
2. Что такое тернарный условный оператор?

12.3. Задание к лабораторной работе

Откомпилируйте программы из п. 12.4.

12.4. Методические указания и порядок выполнения работы

Написать программу, определяющую, какая из курсорных клавиш была нажата.

Функция *getch()* возвращает код нажатой клавиши. В случае нажатия курсорных клавиш эта функция возвращает *0*, а ее повторный вызов позволяет получить код клавиши.

```
#include <stdio.h>
#include <conio.h>
int main()
{
int key;
printf("Нажмите курсорную клавишу\n");
key=getch(); key=getch();
switch(key)
        {
        case 77: printf("Стрелка вправо\n"); break;
        case 75: printf("Стрелка влево\n"); break;
        case 72: printf("Стрелка вверх\n"); break;
        case 80: printf("Стрелка вниз\n"); break;
        default: printf("Не стрелка \n"); break;
        }
getchar();    //фиксация кадра
return 0;
}
```

Если выполняются одни и те же действия при различных значениях констант, метки перечисляются одна за другой:

```
case 77: case 75: case 72: case 80: printf("Стрелки вниз\n"); break;
```

Написать программу печати таблицы значений функции. В этой программе использовались операторы ввода-вывода языка С.

$$y = \begin{cases} t, & x < 0 \\ tx, & 0 \le x < 10 \\ 2t, & x \ge 10 \end{cases}$$

для аргумента, изменяющегося в заданных пределах с заданным шагом. Если *t>0*, должны выводиться целые значения функции. Исходными данными являются: начальное значение аргумента x_n, конечное значение аргумента x_k, шаг изменения аргумента *dx* и параметр *t*. Все величины вещественные. Программа

должна выводить таблицу, состоящую из двух столбцов – значений аргумента и соответствующих им значений функции.

Алгоритм написания программы:

1. Ввести исходные данные.
2. Определить, к какому интервалу принадлежит значение аргумента.
3. Вычислить значение функции по соответствующей формуле.
4. Если *t>0*, преобразовать значение *y* в целое.
5. Вывести строку таблицы.
6. Перейти к следующему значению аргумента.
7. Если оно не превышает конечное значение, повторить шаги 2-6, иначе закончить выполнение.

Шаги 2-6 повторяются многократно, поэтому для их выполнения надо организовать цикл.

```
#include <stdio.h>
#include <conio.h>
main()
{
float xn, xk, t, y, dx;
 printf("Введите xn, xk, dx, t ");
scanf("%f%f%f%f", &xn,&xk, &dx, &t);
printf("--------------------------\n") ;
printf("|     X     |       Y       |\n") ;
printf("--------------------------\n") ;
for(float x= xn ; x<= xk;  x+=dx)
    {
    if(x<0)  y=t;
    if(x>=0 && x<10)          y=t*x;
    if(x>10) y=2*t;
    if(t>100) printf("|%9.2f |%9d |\n", x, (int)y) ;
          else printf("|%9.2f |%9.2f |\n", x, y) ;
    }
printf("--------------------------\n") ;
getch();
return 0;
}
```

Вычислить и вывести на экран значение функции *y*. В этой программе использовались операторы ввода-вывода языка С.

$$y = \begin{cases} \sum_{i=2}^{N} i^2 + \prod_{k=1}^{N+1} k, & N \le 5 \\ \sum_{m=1}^{N-1} \sqrt{m}, & N > 5 \end{cases}$$

Алгоритм:

1. Получить с экрана число *N*.
2. Проверить, к какому промежутку принадлежит *N*.
3. Если *N*≤5, тогда вычислить $\sum_{i=2}^{N} i^2$, вычислить $\prod_{k=1}^{N+1} k$ и сложить результаты.
4. Если *N*>5, тогда вычислить $\sum_{m=1}^{N-1} \sqrt{m}$.
5. Вывести на экран значение *y*.

```
#include <stdio.h>
#include <conio.h>
main()
{
float xn, xk, t, y, sum=0, pr=1;
int N, i, k, m;
 printf("Введите N ");
scanf("%d", &N);
if(N<=5)
   {
for(i=2 ; i<=N; i++)
          sum=sum+i*i;
for(k=1 ;k<=N+1;ki++)
          pr=pr*k;
y=sum+pr ;
}
else
       {
       for(m=1 ; m<=N-1; m++)
                 sum=sum+sqrt(m);
   y=sum ;
   }
printf("%6.2f|\n",y);
getch();
return 0;
```

}

12.5. Индивидуальное задание

Вариативность не предполагается.

1. Создайте таблицу значений функции

$$F = \begin{cases} ax^2 + bx & при\ x < 0\ и\ b \neq 0 \\ \dfrac{x-a}{x-c} & при\ x > 0\ и\ b = 0 \\ \dfrac{x}{c} & в\ остальных\ случаях \end{cases}$$

где *a, b, c* действительные числа. Функция *F* должна принимать действительное значение, если выражение (Ац ИЛИ Вц)И(Ац ИЛИ Сц) не равно нулю и целое в противном случае. Через Ац, Вц, Сц обозначены целые части значений *a, b, c*. Значения *a, b, c, Хнач, Хкон, dx* ввести с клавиатуры.

2. Вычислить и вывести на экран значение функции *y*.

$$y = \begin{cases} \prod_{k=1}^{N+2} \cos\sqrt{k} \times \sin N, & N \leq 10 \\ \sum_{l=2}^{N+10} l^3, & N > 10 \end{cases}$$

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. СЧЕТНЫЙ ЦИКЛ FOR. ЦИКЛ WHILE

13.1. Общие сведения

Цель: изучить циклы в C++.

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

Условия допуска к выполнению: показать конспект по теоретической подготовке и решить подготовительный пример.

Критерии положительной оценки: показать выполненную работу: запрограммированный пример; ответить на вопросы преподавателя.

13.2. Теоретическое введение

Цикл – это многократно повторяющиеся действия в программе.

Если мы знаем точное количество действий (итераций) цикла, то можем использовать **цикл *for***. Синтаксис:

```
for (начало цикла;
    условие окончания цикла;
    действия в конце каждой итерации цикла) {
        оператор1;
        оператор2;
        операторN;
}
```

Итерацией цикла называется один проход этого цикла

Существует частный случай этой записи:

```
for (счетчик = значение; счетчик < значение; шаг цикла) {
   тело цикла;
}
```

Описание синтаксиса

1. Сначала присваивается первоначальное значение счетчику.
2. Затем задается конечное значение счетчика цикла. После того, как значение счетчика достигнет указанного предела, цикл завершится.
3. Задаем шаг цикла.

Цикл *while*

Когда мы не знаем, сколько итераций должен произвести цикл, нам понадобится цикл ***while*** или ***do...while***. Синтаксис цикла ***while***:

```
while (Условие) {
   Тело цикла;
}
```

Данный цикл будет выполняться, пока условие, указанное в круглых скобках является истиной.

Литература:

Воробейкина, И.В. Технологии и методы программирования: учеб. пособие. Часть I / И.В. Воробейкина. – Калининград: Изд-во БГАРФ, 2021. – 108 с.

Контрольные вопросы для самопроверки:

1. Расскажите алгоритм работы циклов *for* и *while.*
2. Как задать бесконечный цикл? Что нужно предусмотреть для выхода из него? В каких случаях задается бесконечный цикл?

13.3. Задание к лабораторной работе

Откомпилируйте программы из п. 13.4.

13.4. Методические указания и порядок выполнения работы

Если нам необходимо, чтобы пример с вычислением функции повторялся, например 5 раз, можно записать:

```
#include <iostream.h>
int main()
{
int i;
float x,y;
for (i=1; i<=5; i++)
        {
        cout<<"Введите значение аргумента ";
        cin>>x;
        if (x<2) y=0;
        if (x>=-2 && x<-1) y=-x-2;
        if (x>=-1 && x<1) y=x;
        if (x>=1 && x<2) y=-x+2;
        if (x>=2) y=0;
        cout<<"y="<<y<<endl;
        }
getchar();    //фиксация кадра
return 0;
}
```

Тот же пример с циклом *while:*

```
#include <iostream.h>
```

```
int main()
{
int i;
float x,y;
(i=1;
while( i<=5)        {
        cout<<”Введите значение аргумента ”;
        cin>>x;
        if (x<2) y=0;
        if (x>=-2 && x<-1) y=-x-2;
        if (x>=-1 && x<1) y=x;
        if (x>=1 && x<2) y=-x+2;
        if (x>=2) y=0;
        cout<<”y=”<<y<<endl;
        i++;
        }
system(“pause”);  //фиксация кадра
return 0;
}
```

Напишем программу, которая будет считать сумму всех чисел от 1 до 1000.

```
#include <iostream>
using namespace std;

int main()
{
   int i; // счетчик цикла
   int sum = 0; // сумма чисел от 1 до 1000.
   for (i = 1; i <= 1000; i++) // задаем начальное значение 1, конечное 1000 и задаем шаг цикла - 1.
      sum += sum;
   cout << "Сумма чисел от 1 до 1000 = " << sum << endl;
   return 0;
}
```

Решим ту же задачу с помощью цикла ***while***. Хотя здесь мы точно знаем, сколько итераций должен выполнить цикл, очень часто бывают ситуации, когда это значение неизвестно.

```
#include <iostream>
using namespace std;
```

```
int main()
{
 setlocale(0, "");
int i = 0; // инициализируем счетчик цикла.
int sum = 0; // инициализируем счетчик суммы.
while (i < 1000)
{
i++;
sum += i;
}
cout << "Сумма чисел от 1 до 1000 = " << sum << endl;
return 0; }
```

Для бесконечного цикла записываем, например: *for(;;) while(true).* Не забудьте организовать корректный выход из бесконечного цикла.

13.5. Индивидуальное задание

Вариативность не предполагается.

Запрограммировать решение квадратных уравнений, имеющих общий вид $ax^2+bx+c=0$, когда нам заранее неизвестно их количество. Выход из цикла сделать при условии, когда коэффициент $a=0$ (тогда квадратное уравнение вырождается в линейное).

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. ОДНОМЕРНЫЕ И ДВУМЕРНЫЕ МАССИВЫ. ОБЪЯВЛЕНИЕ, ТИПЫ, МЕТОДЫ РАБОТЫ

14.1. Общие сведения

Цель: изучить структурированные типы данных (массивы) в C++.

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

Условия допуска к выполнению: показать конспект по теоретической подготовке и решить подготовительный пример.

Критерии положительной оценки: показать выполненную работу: запрограммированный пример; ответить на вопросы преподавателя.

14.2. Теоретическое введение

Одномерный массив – это список переменных одного типа, связанных общим именем. Для объявления одномерного массива используется следующая форма записи:

тип имя_массива [размер];

Здесь с помощью элемента *тип* объявляется базовый тип массива. Количество элементов, которые будут храниться в массиве, определяется элементом *размер*. Например, при выполнении приведенной ниже инструкции объявляется *int*-массив, состоящий из *10* элементов, с именем *sample:*

int sample[10];

Доступ к отдельному элементу массива осуществляется с помощью индекса. В *C++* первый элемент массива имеет нулевой индекс. Поскольку массив *sample* содержит 10 элементов, его индексы изменяются от *0* до *9*. Так, первым элементом массива *sample* является *sample[0],* а последним – *sample[9]*.

Элементы массива в памяти расположены последовательно друг за другом.

В *C++* не выполняется никакой проверки нарушения границ массивов, т.е. программист может обратиться к массиву за его пределами. Если это происходит при выполнении инструкции присваивания, могут быть изменены значения в ячейках памяти, выделенных некоторым другим переменным или даже вашей программе. Обращение к массиву размером *N* элементов за границей *N*-го элемента может привести к разрушению программы при отсутствии каких-либо замечаний со стороны компилятора и без выдачи сообщений об ошибках во время работы программы.

```
// Некорректная программа.
int main()
{
int crash[10], i;
for(i=0; i <100; i++) crash[i]=I;
return 1;
```

}

В данном случае цикл *for* выполнит *100* итераций, несмотря на то, что массив *crash* предназначен для хранения лишь десяти элементов. При выполнении этой программы возможна перезапись важной информации, что может привести к аварийной остановке программы.

Двумерные и многомерные массивы

Чтобы объявить двумерный массив целочисленных значений размером *10×20* с именем *twod*, достаточно записать следующее:

int twod[10][20];

В двумерном массиве позиция любого элемента определяется двумя индексами. Если представить двумерный массив в виде таблицы данных, то один индекс означает строку, а второй – столбец; правый индекс будет изменяться быстрее, чем левый.

Формат инициализации массивов:

тип имя_массива [размер] = {список_значений};

Здесь элемент *список_значений* представляет собой список значений инициализации элементов массива, разделенных запятыми. Например, в следующем фрагменте *10*-элементный целочисленный массив инициализируется числами от *1* до *10*:

int i[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

После выполнения этой инструкции элемент *i[0]* получит значение *1*, а элемент *i[9]* – значение *10*.

Для символьных массивов, предназначенных для хранения строк, предусмотрен сокращенный вариант инициализации:

char имя_массива[размер] = "строка";

Следующий фрагмент кода инициализирует массив *str* фразой *привет*:

char str[7] = "привет";

Это равнозначно поэлементной инициализации:

char str[7] = {'п', 'р', 'и', 'в', 'е', 'т', '\0'};

Поскольку в *C++* строки должны завершаться нулевым символом, убедитесь, что при объявлении массива его размер указан с учетом признака конца. Именно поэтому в предыдущем примере массив *str* объявлен как *7*-элементный, несмотря на то, что в слове *привет* только шесть букв. При использовании строкового литерала компилятор добавляет нулевой признак конца строки автоматически.

Многомерные массивы инициализируются по аналогии с одномерными. В следующем фрагменте программы массив *sqrs* инициализируется числами от *1* до *10* и квадратами этих чисел:

int sqrs[10][2] = {

1, 1,

2, 4,
3, 9,
4, 16,
5, 25,
6, 36,
7, 49,
8, 64,
9, 81,
10, 100};

При инициализации многомерного массива подгруппу инициализаторов можно заключить в фигурные скобки. Вот, например, как выглядит еще один вариант записи предыдущего объявления.

int sqrs[10][2] = { {1, 1}, {2, 4}, {3, 9}, {4, 16}, {5, 25}, {6, 36}, {7, 49}, {8, 64}, {9, 81}, {10, 100} };

При использовании подгрупп инициализаторов недостающие члены подгруппы будут инициализированы нулевыми значениями автоматически.

Безразмерная инициализация массивов

Предположим, что мы используем следующий вариант инициализации массивов для построения таблицы сообщений об ошибках:

char e1[14] = "Деление на 0\n";
char e2[23] = "Конец файла\n";
char e3[21] = "В доступе отказано\n";

В *C++* предусмотрена возможность автоматического определения длины массивов путем использования их безразмерного формата. Если в инструкции инициализации массива не указан его размер, *C++* автоматически создаст массив, размер которого будет достаточным для хранения всех значений инициализаторов:

char e1[] = "Деление на 0\n";
char e2[] = "Конец файла\n";
char e3[] = "В доступе отказано\n";

Метод безразмерной инициализации позволяет изменить любое сообщение без пересчета его длины. При инициализации многомерных массивов необходимо указать все данные, за исключением крайней слева размерности, чтобы *C++*-компилятор мог должным образом индексировать массив. В следующем примере массив *sqrs* объявляется как безразмерный:

int sqrs[][2] = {
1, 1,
2, 4,
3, 9,
4, 16,

5, 25,
6, 36,
7, 49,
8, 64,
9, 81,
10, 100 };

Преимущество такой формы объявления перед «габаритной» (с точным указанием всех размерностей) состоит в том, что программист может удлинять или укорачивать таблицу значений инициализации, не заботясь об изменении размерности массива.

Литература:

Воробейкина, И.В. Технологии и методы программирования: учеб. пособие. Часть I / И.В. Воробейкина. – Калининград: Изд-во БГАРФ, 2021. – 108 с.

Глава 2. Массивы и строки.

Контрольные вопросы для самопроверки:

1. Как можно обратиться к элементу массива?
2. В каком случае можно не указывать размерность массива при его объявлении?
3. Можно ли при объявлении с инициализацией многомерного массива использовать пустые квадратные скобки?

14.3. Задание к лабораторной работе

Откомпилируйте программы из п. 14.4.

14.4. Методические указания и порядок выполнения работы

Следующая программа помещает в *sample* числа от *0* до *9*.

```
#include <iostream>
using namespace std;
int main()
{
// резервируем область памяти для 10 элементов типа int
int sample[10];
int t;
// Помещаем в массив значения
for(t=0; t<10;++t)
cout<< sample[t] << ' ';
return 0;
```

```
}
```

В следующем примере в двумерный массив помещаются последовательные числа от *1* до *12*.

```
#include <iostream>
using namespace std;
int main()
{
int t,i, num[3][4];
for(t=0; t<3; ++t)
        {
        for(i=0; i<4; ++i)
                {
                num[t][i]=t*4+i+1;
                cout<< num[t][i] << ' ';
                }
        cout << '\n';
        }
return 0;
}
```

В этом примере элемент *num[0][0]* получит значение *1*, элемент *num[0][1]* – значение *2*, элемент *num[0][2]* – значение *3* и т.д. Значение элемента *num[2][3]* будет равно числу *12*.

Вывод двумерного массива в виде таблицы:

```
#include <iostream>
using namespace std;
int main()
{
int t,i, num[3][4]={1,2,3,4,5,6,7,8};
for(t=0; t<3; ++t)
        {
        for(i=0; i<4; ++i)
                {
                cout.width(2);
                cout<< num[t][i]';
                }
        cout << '\n';
        }
return 0;
```

}

Самостоятельно разберитесь, как работает объект *cout.width(2).*

14.5. Индивидуальное задание

Вариативность не предполагается.

1. Создайте программу сложения матриц. Матрицы объявить, как двумерные массивы, инициализировать их с экрана. Результат вывести в виде таблицы.

2. Создайте целочисленный массив из *n* элементов. Вычислить количество положительных элементов массива, расположенных после первого отрицательного элемента.

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. МЕТОДЫ СОРТИРОВКИ МАССИВОВ, ПОИСК МИНИМАЛЬНОГО И МАКСИМАЛЬНОГО ЭЛЕМЕНТОВ

15.1. Общие сведения

Цель: освоить основные алгоритмы обработки массивов – сортировка, поиск минимального и максимального значений.

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

Условия допуска к выполнению: показать конспект по теоретической подготовке и решить подготовительный пример.

Критерии положительной оценки: показать выполненную работу: запрограммированный пример; ответить на вопросы преподавателя.

15.2. Теоретическое введение

Сортировка одномерного массива

Одной из самых распространенных операций, выполняемых над массивами, является сортировка. Существует множество различных алгоритмов сортировки. Широко применяется, например, сортировка перемешиванием и сортировка методом Шелла. Известен также алгоритм *Quicksort* (быстрая сортировка с разбиением исходного набора данных на две половины так, что любой элемент первой половины упорядочен относительно любого элемента второй половины). Однако самым простым считается алгоритм сортировки пузырьковым методом. Несмотря на то, что пузырьковая сортировка не отличается высокой эффективностью (его производительность неприемлема для сортировки больших массивов), его можно применять для сортировки массивов малого размера. Алгоритм сортировки пузырьковым методом получил свое название от способа, используемого для упорядочивания элементов массива. Здесь выполняются повторяющиеся операции сравнения и при необходимости меняются местами смежные элементы. При этом элементы с меньшими значениями постепенно перемещаются к одному концу массива, а элементы с большими значениями – к другому. Этот процесс напоминает поведение пузырьков воздуха в резервуаре с водой. Пузырьковая сортировка выполняется путем нескольких проходов по массиву, во время которых при необходимости осуществляется перестановка элементов, оказавшихся не на своем месте. Количество проходов, гарантирующих получение отсортированного массива, равно количеству элементов в массиве, уменьшенному на единицу.

Универсальным считается алгоритм *Quicksort*. В стандартную библиотеку *C++* включена функция *qsort()*, которая реализует одну из версий этого алгоритма.

Алгоритм поиска минимального и максимального элементов

Поиск *min* и *max:*

```
int n=10;
int A[n]={.....}; //инициализация массива
int i, imax, imin;
for(i=imax=imin=0; i<n; i++)
        {
        if (A[i]> A[imax]) imax=i;
        if (A[i]< A[imin]) imin=i;
        }
cout<<"Минимальный элемент: " << A[imin] << endl <<"Максимальный элемент: "<<A[imin]<<endl;
```

Литература:

Воробейкина, И.В. Технологии и методы программирования: учеб. пособие. Часть I / И.В. Воробейкина. – Калининград: Изд-во БГАРФ, 2021. – 108 с.

Глава 2. Массивы и строки.

Контрольные вопросы для самопроверки:

1. Можно ли использовать пузырьковый метод сортировки массивов для сортировки больших массивов?
2. Назовите известные вам методы сортировки массивов.

15.3. Задание к лабораторной работе.

Откомпилируйте программы из п. 15.4.

15.4. Методические указания и порядок выполнения работы

В следующей программе реализована сортировка массива (целочисленного типа), содержащего случайные числа.

```
// Использование метода пузырьковой сортировки
// для упорядочения массива.
#include <iostream>
#include <cstdlib>
using namespace std;
int main()
{
int nums[10];
int a, b, t, size;
size = 10; // Количество элементов, подлежащих сортировке.
```

```
// Помещаем в массив случайные числа.
for(t=0; t<size; t++) nums[t]=rand();
cout<< "Исходный массив: ";
for(t=0; t<size; t++) cout<< nums[t] << ' ';
cout << '\n';
// Реализация метода пузырьковой сортировки.
for(a=1; a<size; a++)
        for(b=size-1; b>=a; b--)
        {
        if(nums[b-1] > nums[b])
                {
                // Меняем элементы местами.
                t = nums[b-1];
                nums[b-1] = nums[b];
                nums[b] = t;
                }
        }
}
// Конец пузырьковой сортировки.
// Отображаем отсортированный массив.
cout << "Отсортированный массив: ";
for(t=0; t<size; t++) cout<< nums[t] << ' ';
return 0;
}
```

Пусть существует целочисленный массив из *n* элементов. Вычислить количество положительных элементов массива, расположенных между его минимальным и максимальным элементами.

```
#include <iostream>
using namespace std;
int main()
{
const int n=10;
int A[n]={.....}; //инициализация массива
int i, imax, imin, count;
for(i=imax=imin=0; i<n; i++)
{
if (A[i]> A[imax]) imax=i;
if (A[i]< A[imin]) imin=i;
}
```

```
cout<<"Минимальный элемент: " << A[imin] << endl <<"Максимальный элемент: "<<A[imin]<<endl; //отладка
//что встретилось раньше – максимум или минимум?
int ibeg=imax<imin ? imax : imin;
int iend=imax<imin ? imin : imax;
cout<<"ibeg=" << ibeg << endl <<"iend= "<<iend<<endl; //отладка
for(count=0, i=ibeg+1; i<iend; i++)
    if(A[i]>0) count++;
cout<<"Количество положительных элементов =" << count;
return 0;
}
```

Здесь массив просматривается, начиная с элемента, следующего за минимальным (максимальным), до элемента, предшествующего максимальному (минимальному). Индексы границ просмотра хранятся в переменных *ibeg* и *iend,* для определения значений которых используется тернарная условная операция.

15.5. Индивидуальное задание

Вариативность не предполагается.

1. Создайте программу сортировки одномерного массива по возрастанию и убыванию.

2. Создайте целочисленный массив из *n* элементов. Вычислить произведение отрицательных элементов массива, расположенных между его минимальным и максимальным элементами. Предусмотреть случай однородного массива и отсутствие отрицательных элементов между минимумом и максимумом.

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. ЛАБОРАТОРНАЯ РАБОТА № 15. СТРОКИ, ФУНКЦИИ РАБОТЫ СО СТРОКАМИ

16.1. Общие сведения

Цель: изучить методы работы со строками.

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

Условия допуска к выполнению: показать конспект по теоретической подготовке и решить подготовительный пример.

Критерии положительной оценки: показать выполненную работу: запрограммированный пример; ответить на вопросы преподавателя.

16.2. Теоретическое введение

Строки

Чаще всего одномерные массивы используются для создания символьных строк. В *C++* строка определяется как символьный массив, который завершается нулевым символом (*'\0'*). При определении длины символьного массива необходимо учитывать признак ее завершения и задавать его длину на единицу больше длины самой большой строки из тех, которые предполагается хранить в этом массиве. Например, объявляя массив *str*, предназначенный для хранения *10*-символьной строки, следует использовать следующую инструкцию. *char str [11];* Заданный здесь размер (*11*) позволяет зарезервировать место для нулевого символа в конце строки. *C++* позволяет определять строковые литералы. Строковый литерал – это список символов, заключенный в двойные кавычки. Вот несколько примеров. *"Привет", "Мне нравится C++", "#$%@@#$", "".* Строка, приведенная последней (*""*), называется пустой или нулевой. Она состоит только из одного нулевого символа (признака завершения строки). Нулевые строки используются для представления пустых строк. Вам не нужно вручную добавлять в конец строковых констант нулевые символы, *C++*-компилятор делает это автоматически.

Некоторые библиотечные функции обработки строк

Язык *C++* поддерживает множество функций обработки строк. Самыми распространенными из них являются следующие. *strcpy() strcat() strlen() strcmp()* Для вызова всех этих функций в программу необходимо включить заголовок *<cstring>*.

Функция *strcpy().* Общий формат вызова:

strcpy (to, from);

Функция *strcpy()* копирует содержимое строки *from* в строку to. Помните, что массив, используемый для хранения строки *to*, должен быть достаточно большим, чтобы в него можно было поместить строку из массива *from*. В противном случае массив to переполнится, что может привести к разрушению программы.

Функция *strcat().* Обращение к функции *strcat()* имеет следующий формат. *strcat(s1, s2);* Функция *strcat()* присоединяет строку *s2* к концу строки *s1*, при этом строка *s2* не изменяется. Обе строки должны завершаться нулевым символом. Результирующая строка *s1* также будет завершаться нулевым символом.

Функция *strcmp()*. Обращение к функции *strcmp()* имеет следующий формат:

strcmp(s1, s2);

Функция *strcmp()* сравнивает строку *s2* со строкой *s1* и возвращает значение *0*, если они равны. Если строка *s1* лексикографически (т.е. в соответствии с алфавитным порядком) больше строки *s2*, возвращается положительное число. Если строка *s1* лексикографически меньше строки *s2,* возвращается отрицательное число.

Функция *strlen()*. Общий формат вызова функции *strlen():*

strlen(s);

Здесь *s* – строка. Функция *strlen()* возвращает длину строки, указанной аргументом *s*.

Массивы строк

Для создания массива строк используется двумерный символьный массив, в котором размер левого индекса определяет количество строк, а размер правого – максимальную длину каждой строки. При выполнении следующей инструкции объявляется массив, предназначенный для хранения *30* строк длиной *80* символов,

char str_array[30][80];

Чтобы получить доступ к отдельной строке, достаточно указать только левый индекс. Следующая инструкция вызывает функцию *gets()* для записи третьей строки массива *str_array:*

gets(str_array[2]);

Рассмотрим пример.

```
// Вводим строки текста и отображаем их на экране.
#include <iostream>
#include <cstdio>
using namespace std;
int main()
{
int t, i;
char text[100][80];
```

```
for(t=0; t<100; t++)
        {
        cout<< t << ": ";
        gets(text[t]);
        if(!text[t][0]) break; // Выход из цикла по пустой строке.
        }
// Отображение строк на экране.
for(i=0; i<<t; i++) cout<< text[i] << '\n';
return 0;
}
```

Обратите внимание на то, как в программе выполняется проверка на ввод пустой строки. Функция *gets()* возвращает строку нулевой длины, если единственной нажатой клавишей оказалась клавиша *[ENTER]*. Это означает, что первым байтом в строке будет нулевой символ. Нулевое значение всегда интерпретируется как ложное, но, взятое с отрицанием *(!)*, дает значение *истина*, которое позволяет выполнить немедленный выход из цикла с помощью инструкции *break*.

Литература:

Воробейкина, И.В. Технологии и методы программирования: учеб. пособие. Часть I / И.В. Воробейкина. – Калининград: Изд-во БГАРФ, 2021. – 108 с.

Глава 2. Массивы и строки.

Контрольные вопросы для самопроверки:

1. Каким символом завершается любая строка?
2. Перечислите функции работы со строками.

16.3. Задание к лабораторной работе

Откомпилируйте программы из п. 16.4.

16.4. Методические указания и порядок выполнения работы

```
// Использование cin-инструкции для считывания строки с клавиатуры
#include <iostream>
using namespace std;
int main()
{
char str[80];
cout << "Введите строку: ";
cin >> str; // Считываем строку с клавиатуры.
cout << "Вот ваша строка: ";
```

```
cout << str;
return 0;
}
```

Несмотря на то, что эта программа формально корректна, она не лишена недостатков. Рассмотрим следующий результат ее выполнения.

Введите строку: Это проверка

Вот ваша строка: Это

Как видите, при выводе строки, введенной с клавиатуры, программа отображает только слово *Это*, а не всю строку. Дело в том, что оператор ">>" прекращает считывание строки, как только встречает символ пробела, табуляции или новой строки. Для решения этой проблемы можно использовать другие функции, например, *gets()*. Формат ее вызова:

gets(имя_массива);

Функция *gets()* считывает вводимые пользователем символы до тех пор, пока он не нажмет клавишу *<ENTER>*. Для вызова функции *gets()* в программу необходимо включить заголовок *<cstdio>*.

```
/* Использование функции gets() для считывания строки с клавиатуры */
#include <iostream>
#include <cstdio>
using namespace std;
int main()
{
char str[80];
cout << "Введите строку: ";
gets(str); // Считываем строку с клавиатуры.
cout << "Вот ваша строка: ";
cout << str;
return 0;
}
```

На этот раз после запуска новой версии программы на выполнение и ввода с клавиатуры текста *Это простой тест* строка считывается полностью, а затем так же полностью и отображается:

Введите строку: Это простой тест

Вот ваша строка: Это простой тест

В этой программе следует обратить внимание на инструкцию

```
cout << str;
```

Здесь (вместо привычного литерала) используется имя строкового массива. Заметим, что имя символьного массива, который содержит строку, можно использовать везде, где допустимо применение строкового литерала.

Самостоятельно рассмотрите другие функции чтения строк.

Использование функции *strcpy()* демонстрируется в следующей программе, которая копирует строку *"Привет"* в строку *str*.

```
#include <iostream>
#include <cstring>
using namespace std;
int main()
{
char str[80];
strcpy(str, "Привет");
cout << str;
return 0;
}
```

Использование функции *strcat()* демонстрируется в следующей программе, которая должна вывести на экран строку *Привет всем!*

```
#include <iostream>
#include <cstring>
using namespace std;
int main()
{
char s1[20], s2[10];
strcpy(s1, "Привет");
strcpy(s2, " всем!");
strcat (s1, s2);
cout << s1;
return 0;
}
```

Использование функции *strcmp()* демонстрируется в следующей программе, которая служит для проверки правильности пароля, введенного пользователем (для ввода пароля с клавиатуры и его верификации служит функция *password()*).

```
#include <iostream>
#include <cstring>
#include <cstdio>
using namespace std;
bool password();
int main()
{
if(password()) cout << "Вход разрешен.\n";
else cout << "В доступе отказано.\n";
return 0; }
/* Функция возвращает значение true, если пароль принят, и значение false в противном случае */
bool password()
{
```

```
char s[80];
cout << "Введите пароль: ";
gets(s);
if(strcmp(s, "пароль"))
        {
        // Строки различны
        cout << "Пароль недействителен.\n";
        return false;
        }
        // Сравниваемые строки совпадают.
return true;
}
```

При использовании функции *strcmp()* важно помнить, что она возвращает число *0* (т.е. значение *false*), если сравниваемые строки равны. Следовательно, если вам необходимо выполнить определенные действия при условии совпадения строк, вы должны использовать оператор *НЕ (!).*

При выполнении следующей программы будет показана длина строки, введенной с клавиатуры.

```
#include <iostream>
#include <cstring>
#include <cstdio>
using namespace std;
int main()
{ char str[80];
cout << "Введите строку: ";
gets(str);
cout << "Длина строки равна: " << strlen(str);
return 0;
}
```

Если пользователь введет строку *Привет всем!*, программа выведет на экране число *12*. При подсчете символов, составляющих заданную строку, признак завершения строки (нулевой символ) не учитывается.

В следующем примере продемонстрируем использование всех этих четырех строковых функций.

```
#include <iostream>
#include <cstring>
#include <cstdio>
using namespace std;
int main()
{
char s1[80], s2 [80];
cout << "Введите две строки: ";
```

```
gets (s1);
gets(s2);
cout <<"Их длины =: "<< strlen (s1)<< ' '<< strlen(s2) << '\n';
if(!strcmp(s1, s2))
        cout << "Строки равны \n";
else
        cout << "Строки не равны \n";
strcat(s1, s2);
cout << s1 << '\n';
strcpy(s1, s2);
cout << s1 << " и " << s2 << ' ';
cout << "теперь равны\n";
return 0;
}
```

Результат работы программы (если по приглашению ввести строки *привет* и *всем*):

```
Их длины =: 6   4
Строки не равны
привет всем
всем и всем теперь равны
```

Следующая программа при выполнении выведет на экран слово *TEST*. Здесь используется библиотечная функция *toupper()*, которая возвращает прописной эквивалент своего символьного аргумента. Для вызова функции *toupper()* необходимо включить в программу заголовок *<cctype>*.

```
// Преобразование символов строки в их прописные эквиваленты.
#include <iostream>
#include <cstring>
#include <cctype>
using namespace std;
int main()
{
char str[80];
int i;
strcpy(str, "test");
for(i=0; str[i]; i++) str[i] = toupper(str[i]);
cout << str;
return 0;
}
```

Рассмотрим упрощенную базу данных служащих, в которой хранится имя, номер телефона, количество часов, отработанных служащим, и размер почасо-

вого оклада. Чтобы создать такую программу для коллектива, состоящего из десяти служащих, определим четыре массива. Обратите особое внимание на то, как реализуется доступ к каждому массиву.

```
#include <iostream>
using namespace std;
char name[10][80]; // Массив имен служащих.
char phone[10][20]; // Массив телефонных номеров служащих.
float hours[10]; // Массив часов, отработанных за неделю.
float wage[10]; // Массив окладов.
    // Функция возвращает команду, выбранную пользователем.
    int menu()
    {
    int choice;
    cout << "0. Выход из программы\n";
    cout << "1. Ввод информации\n";
    cout << "2. Генерирование отчета\n";
    cout << "\n Выберите команду: ";
    cin >> choice;
    return choice;
    }
    // Функция ввода информации в базу данных.
void enter()
{
int i;
char temp[80];
for(i=0; i<10; i++)
    {
    cout<< "Введите фамилию: ";
    cin >> name[i];
    cout << "Введите номер телефона: ";
    cin >> phone[i];
    cout << "Введите количество отработанных часов: ";
    cin >> hours[i];
    cout << "Введите оклад: ";
    cin >> wage[i];
    }
}
// Отображение отчета.
void report()
{
int i;
for(i=0; i<10; i++)
```

```
        {
        cout<< name[i] << ' ' << phone[i] << '\n';
        cout << "Заработная плата за неделю: "<< wage[i] * hours[i];
        cout << '\n'; }
        }
        }
int main()
{
int choice;
do
        {
        choice = menu(); // Получаем команду, выбранную пользователем.
        switch(choice)
                {
                case 0: break;
                case 1: enter(); break;
                case 2: report(); break;
                default: cout << "Попробуйте еще раз.\n\n";
                }
        }while(choice != 0);
        return 0;
        }
```

16.5. Индивидуальное задание

Вариативность не предполагается.

Создать базу данных студентов, в которой хранится фамилия, год рождения, пол, средняя успеваемость для каждого студента.

16.6. 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.

17. ЛАБОРАТОРНАЯ РАБОТА № 16. УКАЗАТЕЛИ И МАССИВЫ

17.1. Общие сведения

Цель: научиться работать с массивами через указатели.

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

Условия допуска к выполнению: показать конспект по теоретической подготовке и решить подготовительный пример.

Критерии положительной оценки: показать выполненную работу: запрограммированный пример; ответить на вопросы преподавателя.

17.2. Теоретическое введение

Указатели и массивы

В *C++* указатели и массивы тесно связаны между собой; зачастую понятия указатель и массив взаимозаменяемы.

Рассмотрим следующий фрагмент программы.

```
char str[80];
char *p1;
p1 = str;
```

Здесь *str* представляет собой имя массива, содержащего *80* символов, а *p1* – указатель на тип *char*. В третьей строке переменной *p1* присваивается адрес первого элемента массива *str*. Другими словами, после этого присваивания *p1* будет указывать на элемент *str[0]* (в *C++* использование имени массива без индекса генерирует указатель на первый элемент этого массива). Таким образом, при выполнении присваивания *p1 = str* адрес *str[0]* присваивается указателю *p1*. Необходимо понимать: неиндексированное имя массива, использованное в выражении, означает указатель на начало этого массива. Имя массива без индекса образует указатель на начало этого массива. Поскольку *p1* будет указывать на начало массива *str*, указатель *p1* можно использовать для доступа к элементам этого массива. Например, если нужно получить доступ к пятому элементу массива *str*, используйте одно из следующих выражений: *str[4]* или **(p1+4)*. В обоих случаях будет выполнено обращение к пятому элементу. Помните, что индексирование массива начинается с нуля, поэтому при индексе, равном четырем, обеспечивается доступ к пятому элементу. Такой же эффект производит суммирование значения исходного указателя *(p1)* с числом *4*, поскольку *p1* указывает на первый элемент массива *str*. Необходимость использования круглых скобок, в которые заключено выражение *p1+4*, обусловлена тем, что оператор "*" имеет более высокий приоритет, чем оператор "+". Без этих круглых скобок выражение бы свелось к получению значения, адресуемого указателем *p1*, т.е. значения первого элемента массива, которое затем было бы увеличено на *4*.

Убедитесь в правильности использования круглых скобок в выражении с указателями. В противном случае ошибку будет трудно отыскать, поскольку внешне программа может выглядеть вполне корректной.

В *C++* предусмотрено два способа доступа к элементам массивов: с помощью индексирования массивов и арифметики указателей. Арифметические операции над указателями иногда выполняются быстрее, чем индексирование массивов, особенно при доступе к элементам, расположение которых отличается строгой упорядоченностью. Кроме того, иногда указатели позволяют написать более компактный код по сравнению с использованием индексирования массивов.

Как уже было сказано, указатели и массивы часто взаимозаменяемы. Однако в общем случае они не являются взаимозаменяемыми. Рассмотрим фрагмент кода:

```
int num[10];
int i;
for(i=0; i<10; i++) << strlen("C++-компилятор");
        {
        *num = i; // Здесь все правильно.
        num++; // ОШИБКА – num модифицировать нельзя.
        }
```

Здесь используется массив целочисленных значений с именем *num*. Как отмечено в комментарии, несмотря на то, что приемлемо применить к имени *num* оператор * (который обычно применяется к указателям), абсолютно недопустимо модифицировать значение *num*. Дело в том, что *num* – это константа, которая указывает на начало массива. Следовательно, инкрементировать ее нельзя. Несмотря на то, что имя массива (без индекса) действительно генерирует указатель на начало массива, его значение изменению не подлежит.

Индексирование указателя

В *C++* указатель, который ссылается на массив, можно индексировать так, как если бы это было имя массива. Соответствующий такому подходу синтаксис обеспечивает альтернативу арифметическим операциям над указателями, поскольку он более удобен в некоторых ситуациях.

Соглашение о нулевых указателях

Объявленный, но не инициализированный указатель будет содержать произвольное значение. При попытке использовать указатель до присвоения ему конкретного значения можно разрушить не только собственную программу, но даже и операционную систему. Существует соглашение: если указатель содержит нулевое значение, считается, что он ни на что не ссылается. Если всем неиспользуемым указателям присваивать нулевые значения, можно избежать случай-

ного использования неинициализированного указателя. При объявлении указатель любого типа можно инициализировать нулевым значением, например, как это делается в следующей инструкции:

*float *p = 0; // p – теперь нулевой указатель.*

Для тестирования указателя используется инструкция *if* (любой из следующих ее вариантов):

if(p) // Выполняем что-то, если p – не нулевой указатель.

if(!p) // Выполняем что-то, если p – нулевой указатель.

Соблюдая упомянутое выше соглашение о нулевых указателях, вы можете избежать многих серьезных проблем, возникающих при использовании указателей.

Трудности выявления ошибок, связанных с указателями, объясняются тем, что сам по себе указатель не обнаруживает проблему. Проблема может проявиться только косвенно, возможно, даже в результате выполнения нескольких инструкций после крамольной операции с указателем. Например, если один указатель случайно получит адрес не тех данных, то при выполнении операции с этим сомнительным указателем адресуемые данные могут подвергнуться нежелательному изменению, и, что самое здесь неприятное, это тайное изменение, как правило, становится явным гораздо позже. Такое запаздывание существенно усложняет поиск ошибки, поскольку отправляет вас по ложному следу. К тому моменту, когда проблема станет очевидной, вполне возможно, что указатель-виновник внешне будет выглядеть безобидно, и вам придется затратить еще немало времени, чтобы найти истинную причину проблемы. Рассмотрим наиболее частые ошибки, допускаемые при работе с указателями.

Неинициализированные указатели

Эта программа некорректна:

```
int main()
{
int x, *p;
x = 10; *p = x; // На что указывает переменная p?
return 0;
}
```

Здесь указатель *p* содержит неизвестный адрес, поскольку он нигде не определен. У вас нет возможности узнать, где записано значение переменной *x*. При небольших размерах программы (например, как в данном случае) ее странности (которые заключаются в том, что указатель *p* содержит адрес, не принадлежащий ни коду программы, ни области данных) могут никак не проявляться, т.е. программа внешне будет работать нормально. Но по мере ее развития и, соответственно, увеличения ее объема, вероятность того, что *p* станет указывать либо на код программы, либо на область данных, возрастет. В один прекрасный

день программа вообще перестанет работать. Способ не допустить создания таких программ: прежде чем использовать указатель, позаботьтесь о том, чтобы он ссылался на что-нибудь действительное.

Литература:

Воробейкина, И.В. Технологии и методы программирования: учеб. пособие. Часть I / И.В. Воробейкина. – Калининград: Изд-во БГАРФ, 2021. – 108 с.

Глава 3. Указатели.

Контрольные вопросы для самопроверки:

1. Что такое указатели?
2. Что означает разыменование указателя?
3. В чем опасность неинициализированных указателей?
4. Является ли имя массива указателем на этот массив?
5. Можно ли сравнивать указатели?
6. В чем опасность объявленного, но неинициализированного указателя?

17.3. Задание к лабораторной работе

Откомпилируйте программы из п. 17.4.

17.4. Методические указания и порядок выполнения работы

Рассмотрим две версии одной и той же программы. В этой программе из строки текста выделяются слова, разделенные пробелами. Например, из строки *Численные методы* программа должна выделить слова *Численные* и *методы*. Такие разграниченные символьные последовательности называют *лексемами* (*token*). При выполнении программы входная строка посимвольно копируется в другой массив (с именем *token*) до тех пор, пока не встретится пробел. После этого выделенная лексема выводится на экран, и процесс продолжается до тех пор, пока не будет достигнут конец строки. Например, если в качестве входной строки использовать строку *Это лишь простой тест*, программа отобразит следующее:

Это
лишь
простой
тест.

Вот как выглядит версия программы разбиения строки на слова с использованием арифметики указателей.

```
// Программа разбиения строки на слова:
// версия с использованием указателей.
```

```
#include <iostream>
#include <cstdio>
using namespace std;
int main()
{
char str[80];
char token[80];
char *p, *q;
cout << "Введите предложение: ";
gets(str);
p = str;
// Считываем лексему из строки.
while(*p)
        {
        q = token; // Устанавливаем q для указания на начало массива token.
        /* Считываем символы до тех пор, пока не встретится либо пробел,
либо нулевой символ (признак завершения строки). */
        while(*p != ' ' && *p)
        {
        *q = *p;
        q++; p++;
        }
        if(*p) p++; // Перемещаемся за пробел.
        *q = '\0'; // Завершаем лексему нулевым символом.
        cout << token << '\n';
        }
        return 0;
        }
```

А вот как выглядит версия той же программы с использованием индексирования массивов.

```
        // Программа разбиения строки на слова:
        // версия с использованием индексирования массивов.
#include <iostream>
#include <cstdio>
        using namespace std;
        int main()
        {
        char str[80];
        char token[80];
        int i, j;
```

```
        cout << "Введите предложение: ";
        gets(str); // Считываем лексему из строки.
        for(i=0; ; i++)
        {
                /* Считываем символы до тех пор пока не встретится либо
        пробел, либо нулевой символ (признак завершения строки). */
        for(j=0; str[i]!=' ' && str[i]; j++, i++)
                token[j] = str[i];
        token[j] = '\0'; // Завершаем лексему нулевым символом.
        cout << token << '\n';
        if(!str[i]) break;
        }
return 0;
}
```

У этих программ может быть различное быстродействие, что обусловлено особенностями генерирования кода компиляторами. Как правило, при использовании индексирования массивов генерируется более длинный код, чем при выполнении арифметических действий над указателями.

```
// Индексирование указателя подобно массиву.
#include <iostream>
#include <cctype>
using namespace std;
int main()
{
char str[20] = "I drink water";
char *p;
int i;
p = str; // Индексируем указатель.
for(i=0; p[i]; i++) p[i] = toupper(p[i]);
cout << p; // Отображаем строку.
return 0;
}
```

При выполнении программа отобразит на экране следующее. *I DRINK WATER.*

Выражение *p[i]* по своему действию идентично выражению **(p+i)*.

```
// Пример сравнения указателей.
#include <iostream>
using namespace std;
int main()
```

```
{
int num[10];
int *start, *end;
start = num;
end = &num[9];
while(start <= end)
        {
        cout << "Введите число: ";
        cin >> *start;
        start++;
        }
        start << num; //Восстановление исходного значения указателя
        while(start <= end)
        cout << *start << ' '; start++;
        return 0;
        }
```

Как показано в этой программе, поскольку *start* и *end* указывают на общий объект (в данном случае им является массив *num*), их сравнение может иметь смысл.

17.5. Индивидуальное задание

Вариативность не предполагается. Задания выполнить с помощью указателей.

1. Создайте программу сортировки одномерного массива по возрастанию и убыванию.

2. Создайте целочисленный массив из *n* элементов. Вычислить произведение отрицательных элементов массива, расположенных между его минимальным и максимальным элементами. Предусмотреть случай однородного массива и отсутствие отрицательных элементов между минимумом и максимумом.

17.6. 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.

18. ЛАБОРАТОРНАЯ РАБОТА № 17. РАБОТА С ДИНАМИЧЕСКИМИ МАССИВАМИ. ВЕКТОРЫ

18.1. Общие сведения

Цель: познакомиться с динамическими массивами и векторами.

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

Условия допуска к выполнению: показать конспект по теоретической подготовке и решить подготовительный пример.

Критерии положительной оценки: показать выполненную работу: запрограммированный пример; ответить на вопросы преподавателя.

18.2. Теоретическое введение

Работа с динамическими массивами

Если до начала работы программы неизвестно, сколько в массиве элементов, в программе следует использовать динамические массивы. Память под них выделяется с помощью операции *new* в динамической области памяти во время выполнения программы. Адрес начала массива хранится в указателе. Например:

```
int n=10;
int *a=new int[n];
double *b=new double[n];
```

Во второй строке описан указатель на величину *int*, которому присваивается адрес динамической памяти, выделенной с помощью операции *new*. Выделяется столько памяти, сколько необходимо для хранения *n* величин типа *int*. Величина *n* может быть переменной. В третьей строке описан указатель типа *double,* здесь выделяется столько памяти, сколько необходимо для хранения *n* величин типа *double.*

Обнуление памяти при ее выделении не происходит. Инициализировать динамический массив нельзя.

Обращение к элементу динамического массива осуществляется так же, как и к элементу статического массива, например, *a[3]*. Можно обратиться к этому элементу массива и другим способом: **(a+3)*. В этом случае мы **явно** задаем те же действия, что выполняются при обращении к массиву обычным образом.

В указателе хранится адрес начала массива (имя статического массива так же является указателем на его первый элемент, только константным, то есть, ему нельзя присвоить новое значение). Для получения адреса третьего элемента к адресу, хранящемуся в указателе, прибавляется смещение *3*.

Освободить используемую динамическую память можно с помощью оператора *delete[],* например:

```
delete[]a;
```

Размерность массива при этом не указывается. Квадратные скобки в операторе *delete[]* обязательны, при их отсутствии программа поведет себя непредсказуемо.

Известно, что локальная переменная при выходе из блока, в котором она объявлена, теряется. Если эта переменная – указатель, и в ней хранится адрес выделенной динамической памяти, при выходе из блока эта память перестает быть доступной, но не помечается как свободная, поэтому не может быть использована в дальнейшем. Это называется *утечкой памяти:*

```
{
int n;
cin>>n;
int *pmas=new int[n];
}
... //после выхода из блока указатель *pmas недоступен
```

Векторы

В программировании термин *вектор* соответствует массиву. Вектор – это набор однотипных значений, к которым можно обращаться в произвольном порядке. Установить размер объекта *vector* можно во время выполнения программы. Вообще *vector* – альтернатива операции *new* для создания динамического массива. На самом деле класс *vector* использует операции *new* и *delete,* но делает это автоматически. При работе с объектом *vector* в программу включается заголовочный файл <*vector*>. Кроме того, идентификатор *vector* является частью пространства имен *std*, поэтому используется директива *using namespace std* или *std::vector*. Формат объявления вектора:

vector <тип данных> имя вектора;

Например:

```
vector <string> ivector; // создание вектора строк
vector <int> vi;                  // создание вектора int нулевого размера
……
int n;
cin>>n;
vector <double> vd[n];// создание вектора double из n элементов
```

Объекты *vector* изменяют свои размеры при вставке или добавлении значений к ним, поэтому можно при объявлении объекта *vector* указать размер *0.* Для изменения размера вектора используются специальные методы, входящие в состав пакета *vector.*

Проинициализировать вектор можно при объявлении. Формат инициализации:

vector <тип> *имя={<элемент[0]>,<элемент[1]>,…,<элемент [n]>};*

Например, *vector <int> ivector = {1, 2, 3};*

В векторе для обращения к ячейке используются индексы. Но в C++ есть еще один способ это сделать благодаря функции ***at()***. В скобках указываеися индекс той ячейки, к которой нужно обратиться:

```
vector <int> ivector = {1, 2, 3};
ivector.at(1) = 5;  // изменили значение второго элемента
cout << ivector.at(1);  // вывели его на экран
```

Указывать размер вектора можно по-разному. Можно это сделать еще при его объявлении, а можно хоть в самом конце программы.

Объявляем длину вектора на старте:

```
vector <int> vector_first(5);
```

Здесь в круглых скобках после имени вектора указываем первоначальную длину.

Второй способ:

```
vector <int> vector_second;  // создали вектор
vector_second.reserve(5);  // указали число элементов
```

При использовании первого способа ячейки вектора автоматически обнуляются.

Методы и итераторы

Метод *size()* возвращает длину вектора (эта функция практически всегда используется вместе с циклом *for*), *empty()* используется, если требуется узнать, пуст ли вектор. При отсутствии в ячейках какого-либо значения это функция возвратит *true*, в противном случае – *false*.

```
if (ivector.empty()) {
  // ...
}
```

С помощью функции *push_back()* добавляется ячейка в конец вектора, а функция *pop_back()* удаляет одну ячейку в конце вектора.

В методах часто используются итераторы. Итератор – это обобщение указателя. В действительности он может быть указателем или объектом, для которого определены операции над указателем (разыменование, инкремент). Например, *begin()* возвращает итератор, который указывает на первый элемент в контейнере, *end()* возвращает итератор, который ссылается на область памяти, следующую за последним элементом контейнера. Идея итератора *end()* сходна с идеей терминатора *(\0)*, находящегося в конце строки. То есть *end()* определяет позицию, следующую **за** последним элементом контейнера.

Объявление итератора для класса *vector:*

```
vector <mun>::iterator имя итератора;
```

Например:

vector <double>::iterator pd; // это итератор

vector <double> scores;// это объект vector <double>

Теперь итератор *pd* можно применять в коде:

pd= scores.begin(); // pd указывает на первый элемент

//разыменование pd и присвоение значения первому элементу:

** pd=22.3;*

++ pd; // pd указывает на следующий элемент

То есть итератор ведет себя подобно указателю.

Вместо записи *vector <double>::iterator pd=scores.begin();* можно использовать оператор *auto pd=scores.begin();*

Все содержимое контейнера можно определить с помощью следующего кода:

*for(pd=scores.begin(); pd=!scores.end(); pd++) cout<<*pd<<endl;*

Метод *erase()* удаляет указанный диапазон элементов вектора. В качестве аргументов он принимает два итератора, которые определяют границы диапазона удаляемых элементов. Первый итератор указывает на начало диапазона, второй – на элемент, **следующий за концом** диапазона. Например, следующий код удаляет первый и второй элементы – те, на которые ссылаются *begin()* и *begin()+1:*

scores.erase()(scores.begin(),scores.begin()+2);

В документации часто используют нотацию *[p1,p2),* где *p1* и *p2* – итераторы, описывающая диапазон, начинающийся с *p1* и заканчивающийся, но не включающий, *p2.* Таким образом, диапазон *[begin(),end())* охватывает все содержимое вектора. Запись *[p1,p1)* определяет пустой диапазон.

Метод *insert()* выполняет те же функции, что и *push_back(),* кроме того, с помощью *insert()* можно добавлять элементы в начало вектора. В качестве аргументов *insert()* принимает три итератора: первый указывает позицию, после которой будут добавляться новые элементы; второй и третий итераторы описывают добавляемый диапазон. Обычно этот диапазон является частью другого вектора.

Следующий код вставляет все элементы вектора *new_v* (за исключением первого) после первого элемента вектора *old_v*:

vector <int> new_v;

vector <int> old_v;

……..

old_v.insert(old_v.begin(), new_v.begin()+1, new_v.end());

В этом операторе все данные добавляются **после** последнего элемента вектора *old_v:*

old_v.insert(old_v.end(), new_v.begin()+1, new_v.end());

Цикл for, основанный на диапазоне

Рассмотрим пример:

```
double prices[5]={4.99, 10.99, 6.87, 7.99, 8.49};
for(double x : prices) cout<<x<<endl;
```

Здесь *x* изначально представляет первый элемент массива *prices*. После вывода на экран первого элемента цикл затем проходит по *x* для вывода остальных элементов массива.

Еще пример:

```
int math[] = { 0, 1, 4, 5, 7, 8, 10, 12, 15, 17, 30, 41};
for (int number : math) // итерация по массиву math
        cout << number << ' '; // получаем доступ к элементу массива в этой
                               //итерации через переменную number
```

Чтобы изменить значения в массиве применяется другой синтаксис для переменной цикла:

```
for(double &x : prices) x*=0.8;
```

Символ & идентифицирует *x* как ссылочную переменную, и такая форма объявления позволяет изменять содержимое массива, тогда как первая форма не разрешает этого.

Цикл *for*, основанный на диапазоне, также может использоваться со списками инициализации:

```
for(int x : {3,5,2,8,6}) cout<<x<<' ';
```

Литература:

Воробейкина, И.В. Технологии и методы программирования: учеб. пособие. Часть I / И.В. Воробейкина. – Калининград: Изд-во БГАРФ, 2021. – 108 с.

Глава 4. Динамические массивы и векторы.

Контрольные вопросы для самопроверки:

1. Можно ли инициализировать динамический массив?
2. Что такое утечка памяти?
3. В чем преимущество объекта *vector* по сравнению с обычным массивом?
4. Что такое итераторы?

18.3. Задание к лабораторной работе

Откомпилируйте программы из п. 18.4.

18.4. Методические указания и порядок выполнения работы

Объявить числовой динамический массив. Количество элементов массива получить с экрана. Вывести исходный массив на экран. Просуммировать все положительные элементы массива, если таковых нет, вывести соответствующее сообщение.

```
int main()
```

```
{
int i,n,count=0;
cout << " Сколько элементов в массиве? ";
cin >> n;
int *a = new int[n];
for (i = 0; i < n; i++)
        {
        if(i%2==0)
                *(a + i) = i;
        else
                *(a + i) = -i;
        }
for (i = 0; i < n; i++) cout<<*(a + i)<<' '<<(a+i)<<endl;
for (i = 0; i < n; i++) if (*(a + i) > 0) count+=*(a+i);
if (count == 0)
        cout << "Положительных чисел нет\n";
else
        cout << "Сумма=" << count << endl;
delete[] a;
return 0;
}

#include <iostream>
#include <vector>
using namespace std;
int main()
{
 setlocale(0, "");
 vector <int> vector_first(3);
 vector <int> vector_second;
 vector_second.reserve(3);
 cout << "Значения первого вектора (с помощью скобок): ";
 for (int i = 0; i < 3; i++)  cout << vector_first[i] << ' ';
 cout << "Значения второго вектора (с помощью reserve): " << endl;
 for (int i = 0; i < 3; i++)  cout << vector_second[i] << " ";
 return 0;
}
```

Результат работы программы:

Значения первого вектора (с помощью скобок): 0 0 0
Значения второго вектора (с помощью reserve): 17 3412 0

Элементы первого вектора обнулились, элементы второго вектора – «мусор» из оперативной памяти. При использовании второго способа есть некоторый плюс – экономия времени: для первого способа компилятор тратит время, чтобы заполнить все ячейки нулями.

```
int main()
{
vector  vec_string(2);
vec_string[0] = 2;  // заполнили две
vec_string[1] = 3;  // ячейки
for (int i = 0; i< vec_string.size(); i++) cout<<vec_string[i]<< " ";
cout << endl;
vec_string.insert(vec_string.begin(), 1);  // добавили элемент в начало
for (int i = 0; i< vec_string.size(); i++) cout<<vec_string[i] << " ";
cout << endl;
vec_string.insert(vec_string.end(), 4);  // добавили элемент в конец
for (int i = 0; i <vec_string.size(); i++) cout<<vec_string[i] << " ";
return 0;
}
```

Результат:

```
2 3
1 2 3
1 2 3 4
```

Для увеличения скорости работы программы добавлять элементы нужно именно в конец вектора: при добавлении элемента в начало, с помощью той же функции *insert()*, в векторе происходит смещение всех ячеек вправо. К тому же, смещение идет по одной ячейке, а это – линейный поиск, который работает достаточно медленно. При этом, чем больше вектор, тем медленнее будет происходить добавление элементов.

Создать пустой вектор типа *string*. Написать программу, которая записывает в вектор предложения, и вывести содержимое вектора на экран.

```
#include "stdafx.h"
#include <iostream>
#include <vector>
#include <string>
using namespace std;
int main()
{
string str1;
vector <string> vec;
cout << "Введите предложение \n";
```

```
while (getline(cin, str1) && (vec.size() < 4))
            vec.push_back(str1);
for (int i = 0; i < 4; i++) cout << vec[i] << endl;
return 0;
}
```

Или

```
for (int i = 0; i < 4; i++)
    {
    getline(cin, str1);
    vec.push_back(str1);
    }
for (int i = 0; i < vec.size(); i++) cout << vec[i] << endl;
```

18.5. Индивидуальное задание

Вариативность не предполагается.

1. Создать пустой вектор типа *double*. Написать программу, которая записывает в вектор числа до тех пор, пока не введен ноль. Вывести на экран с **помощью итератора** только положительные элементы вектора.

2. Создать пустой вектор типа *int*. Написать программу, которая записывает в вектор только положительные числа. При вводе отрицательного числа заполнение вектора прекратить. Вывести содержимое вектора на экран **с помощью цикла, основанного на диапазоне**.

18.6. 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.

19. ЛАБОРАТОРНАЯ РАБОТА № 18. ПЕРЕДАЧА ФУНКЦИИ АРГУМЕНТОВ ПО ЗНАЧЕНИЮ И ПО ССЫЛКЕ. ПЕРЕГРУЗКА ФУНКЦИЙ И ИСПОЛЬЗОВАНИЕ АРГУМЕНТОВ ПО УМОЛЧАНИЮ

19.1. Общие сведения

Цель: изучить методы работы с функциями.

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

Условия допуска к выполнению: показать конспект по теоретической подготовке и решить подготовительный пример.

Критерии положительной оценки: показать выполненную работу: запрограммированный пример; ответить на вопросы преподавателя.

19.2. Теоретическое введение

Вызов функций с указателями

В *C++* разрешается передавать функции указатели. Для этого достаточно объявить параметр типа указатель.

```
// Передача функции указателя
void f (int *j)
{ *j = 100; // Переменной, адресуемой указателем j, присваивается число 100. }
int main()
{
int i;
int *p;
p = &i; // Указатель p теперь содержит адрес переменной i.
f(p);
cout << i; // Переменная i теперь содержит число 100.
return 0;
}
```

В этой программе функция *f()* принимает указатель на целочисленное значение. В функции *main()* указателю *p* присваивается адрес переменной *i*. Затем из функции *main()* вызывается функция *f()*, а указатель *p* передается ей в качестве аргумента. После того как параметр-указатель *j* получит значение аргумента *p*, он (так же, как и *p*) будет указывать на переменную *i*, определенную в функции *main()*. Таким образом, при выполнении операции присваивания **j = 100;* переменная *i* получает значение *100*. Поэтому программа отобразит на экране число *100*. В общем случае приведенная здесь функция *f()* присваивает число *100* переменной, адрес которой был передан этой функции в качестве аргумента.

В примере 5.3.1 необязательно было использовать переменную *p*. Вместо нее при вызове функции *f()* достаточно использовать переменную *i*, предварив ее оператором *&* (при этом генерируется адрес переменной *i*):

```
int main()
{
int i;
f(&i);
cout << i; // Переменная i теперь содержит число 100.
return 0;
}
```

При выполнении некоторой операции в функции, которая использует указатель, эта операция выполняется над переменной, адресуемой этим указателем. Таким образом, такая функция может изменить значение объекта, адресуемого ее параметром.

Вызов функций с массивами

Если массив является аргументом функции, то при вызове такой функции ей передается только адрес первого элемента массива, а не полная его копия. В *C++* имя массива без индекса представляет собой указатель на первый элемент этого массива.

Существует три способа объявить параметр, который принимает указатель на массив. Во-первых, параметр можно объявить как массив, тип и размер которого совпадает с типом и размером массива, используемого при вызове функции. Этот вариант объявления параметра-массива продемонстрирован в следующем фрагменте:

```
void display(int num[10])
{
int i;
for(i=0; i<10;i++) cout<< num[i] <<' ';
}
int main()
{
int t[10], i;
for(i=0; i<10; ++i) t[i]=I;
display(t); // передаем функции массив t
return 0;
}
```

Несмотря на то, что параметр *num* объявлен здесь как целочисленный массив, состоящий из *10* элементов, компилятор автоматически преобразует его в указатель на целочисленное значение. Необходимость этого преобразования объясняется тем, что никакой параметр в действительности не может принять

массив целиком. А так как будет передан один лишь указатель на массив, то функция должна иметь параметр, способный принять этот указатель.

Второй способ объявления параметра-массива состоит в его представлении в виде безразмерного массива, как показано ниже:

```
void display(int num[])
{
int i;
for(i=0; i<10; i++) cout<< num[i] << ' ';
}
```

Здесь параметр *num* объявляется как целочисленный массив неизвестного размера. Целочисленный массив при таком способе объявления также автоматически преобразуется компилятором в указатель на целочисленное значение.

Третий способ объявления параметра-массива. При передаче массива функции ее параметр можно объявить как указатель. Этот вариант используется чаще всего:

```
void display(int *num)
{
int i;
for(i=0; i<10; i++) cout<< num[i] << ' ';
}
```

Возможность такого объявления параметра (в данном случае *num*) объясняется тем, что любой указатель (подобно массиву) можно индексировать с помощью символов квадратных скобок (*[]*). Таким образом, все три способа объявления параметра-массива приводятся к одинаковому результату, который можно выразить одним словом: указатель. Однако отдельный элемент массива, используемый в качестве аргумента, обрабатывается подобно обычной переменной.

Например, рассмотренную выше программу можно было бы переписать, не используя передачу целого массива:

```
void display(int num)
{
cout<< num << ' ';
}
int main()
{
int t[10],i;
for(i=0; i<10; ++i) t[i]=I;
for(i=0; i<10; i++) display(t[i]);
return 0;
}
```

Параметр, используемый функцией *display()*, имеет тип *int*. Здесь не важно, что эта функция вызывается с использованием элемента массива, поскольку ей передается только один его элемент. Если массив используется в качестве аргумента функции, то функции передается адрес этого массива. Это означает, что код функции может потенциально изменить реальное содержимое массива, используемого при вызове функции.

Передача функциям строк

Строки в *C++* – это обычные символьные массивы, которые завершаются нулевым символом. Таким образом, при передаче функции строки реально передается только указатель (типа *char**) на начало этой строки. Рассмотрим, например, следующую программу. В ней определяется функция *stringupper()*, которая преобразует строку символов в ее прописной эквивалент.

```
// Передача функции строки.
#include <iostream>
#include <cstring>
#include <cctype>
using namespace std;
void stringupper(char *str)
{
while(*str)
        {
        *str = toupper(*str); // Получаем прописной эквивалент одного символа
        str++; // Переходим к следующему символу
        }
}
int main()
{
char str[80];
strcpy(str, "Мне нравится C++");
stringupper(str);
cout << str; // Отображаем строку с использованием прописного написания символов.
return 0;
}
```

Результаты выполнения этой программы:

МНЕ НРАВИТСЯ C++

Обратите внимание на то, что параметр *str* функции *stringupper()* объявляется с использованием типа *char**. Это позволяет получить указатель на символьный массив, который содержит строку.

Инструкция return

Инструкция *return* выполняет две важные операции: она обеспечивает возвращение управления к инициатору вызова функции; ее можно использовать для передачи значения, возвращаемого функцией.

Управление от функции передается инициатору ее вызова в двух ситуациях: либо при обнаружении закрывающейся фигурной скобки, либо при выполнении инструкции *return*. Инструкцию *return* можно использовать с некоторым заданным значением либо без него. Но если в объявлении функции указан тип возвращаемого значения (т.е. не тип *void*), то функция должна возвращать значение этого типа. Только *void*-функции могут использовать инструкцию *return* без значения. Для *void*-функций инструкция *return* используется как элемент программного управления. В приведенной ниже функции выводится результат возведения числа в положительную целочисленную степень. Если же показатель степени окажется отрицательным, инструкция *return* обеспечит выход из функции, прежде чем будет сделана попытка вычислить такое выражение. В этом случае инструкция *return* действует как управляющий элемент, предотвращающий нежелательное выполнение определенной части функции.

```
void power(int base, int exp)
{
int i;
if(exp<0 return;
i=1;
for(; exp; exp--) i*=base;
cout<< "Результат равен: " << i;
}
```

Функция может содержать несколько инструкций *return* и будет завершена при выполнении хотя бы одного из них:

```
void f()
{
// ...
switch(c)
        {
        case 'a': return;
        case 'b': // ...
        case 'c': return;
        }
if(count 100) return;
}
```

Однако слишком большое количество инструкций *return* может ухудшить ясность алгоритма. Несколько инструкций *return* стоит использовать только в том случае, если они способствуют ясности функции.

Каждая функция, кроме типа *void*, возвращает какое-нибудь значение. Это значение явно задается с помощью инструкции *return*. Значит, любую не *void*-функцию можно использовать в качестве операнда в выражении. Каждое из следующих выражений допустимо:

```
x = power(y);
if(max(x, y)) > 100) cout << "больше";
switch(abs (x)) {…
```

функция, может иметь несколько инструкций *return*.

```
#include <iostream>
using namespace std;
// Функция возвращает индекс искомой подстроки или -1, если она не была найдена.
int find_substr(char *sub, char *str)
{
int t; char *p, *p2;
for(t=0; str[t]; t++)
        {
        p = &str[t]; // установка указателей
        p2 = sub;
        while(*p2 && *p2==*p)
                {
                // проверка совпадения
                p++; p2++;
                }
                /* Если достигнут конец p2-строки (т.е. подстроки), то подстрока была найдена. */
        if(!*p2) return t; // Возвращаем индекс подстроки.
        }
return -1; // Подстрока не была обнаружена.
}
int main()
{
int index;
index = find_substr("три", "один два три четыре");
cout << "Индекс равен " << index; // Индекс равен 9.
return 0;
```

}

Результаты выполнения программы:

Индекс равен 9

Поскольку искомая подстрока существует в заданной строке, выполняется первая инструкция *return*.

Два способа передачи аргументов

В общем случае в языках программирования, как правило, предусматривается два способа, которые позволяют передавать аргументы в подпрограммы. Первый называется вызовом по значению (*call-by-value*). В этом случае значение аргумента копируется в формальный параметр подпрограммы. Следовательно, изменения, внесенные в параметры подпрограммы, не влияют на аргументы, используемые при ее вызове. При вызове по ссылке функции передается адрес аргумента. Второй способ передачи аргумента подпрограмме называется вызовом по ссылке (*call-by-reference*). В этом случае в параметр копируется адрес аргумента, а не его значение. В пределах вызываемой подпрограммы этот адрес используется для доступа к реальному аргументу, заданному при ее вызове. Это значит, что изменения, внесенные в параметр, окажут воздействие на аргумент, используемый при вызове подпрограммы.

Использование указателя для обеспечения вызова по ссылке

Вызов по значению можно «вручную» заменить вызовом по ссылке. В этом случае функции будет передаваться адрес аргумента (т.е. указатель на аргумент). Это позволит внутреннему коду функции изменить значение аргумента, которое хранится вне функции. Примером такого «дистанционного» управления значениями переменных является вызов функции с указателями.

Ссылочные параметры

Ссылочный параметр автоматически получает адрес соответствующего аргумента. Несмотря на возможность вручную организовать вызов по ссылке с помощью оператора получения адреса, такой подход не всегда удобен. Во-первых, он вынуждает программиста выполнять все операции с использованием указателей. Во-вторых, вызывая функцию, программист должен не забыть передать ей адреса аргументов, а не их значения. Вместо этого можно сориентировать компилятор на автоматическое использование вызова по ссылке для одного или нескольких параметров конкретной функции. Такая возможность реализуется с помощью ссылочного параметра (*reference parameter*). При использовании ссылочного параметра функции автоматически передается адрес (а не значение) аргумента. При выполнении кода функции, а именно при выполнении операций над ссылочным параметром, обеспечивается его автоматическое разыменование, и

поэтому программисту не нужно использовать операторы, работающие с указателями. Ссылочный параметр объявляется с помощью символа, который должен предшествовать имени параметра в объявлении функции. Операции, выполняемые над ссылочным параметром, оказывают влияние на аргумент, используемый при вызове функции, а не на сам ссылочный параметр.

Перегрузка функций

В *C++* несколько функций могут иметь одинаковые имена, но при условии, что их параметры будут различными. Такую ситуацию называют перегрузкой функций (*function overloading*), а функции, которые в ней задействованы, – перегруженными (*overloaded*). Перегрузка функций – один из способов реализации полиморфизма в *C++*.

Аргументы по умолчанию. Перегрузка функций и неоднозначность

Чтобы передать аргумент по умолчанию, нужно в инструкции определения функции задать параметру то значение, которое необходимо передать, когда при вызове функции соответствующий аргумент не будет указан.

```
//параметрам по умолчанию присваивается 0
void f(int a=0, int b=0);
```

Теперь эту функцию можно вызывать тремя различными способами. Она может вызываться

– с двумя заданными аргументами;

– только с первым заданным аргументом (в этом случае параметр *b* по умолчанию станет равен нулю);

– вообще без аргументов (параметры *a* и *b* по умолчанию станут равными нулю).

Таким образом, все следующие вызовы функции *f()* правильны:

```
f(); //по умолчанию a=0 и b=0
f(10); // a=10 и по умолчанию b=0
f(10,99) // a=10 и b=99
```

При создании функции, имеющей аргументы по умолчанию, эти аргументы должны задаваться только один раз: либо в прототипе функции, либо в ее определении.

Аргументы по умолчанию нельзя задавать одновременно и в определении, и в прототипе функции.

Все параметры, задаваемые по умолчанию, должны указываться правее параметров, передаваемых обычным путем. И после того, как вы начали определять параметры по умолчанию, параметры, которые по умолчанию не передаются, уже определять нельзя. Такой фрагмент приведет к ошибке компиляции:

void f(int a=0, int b); //b тоже должен задаваться по умолчанию

Аргументы по умолчанию должны быть константами или глобальными переменными. Они не могут быть локальными переменными или другими параметрами.

Перегрузка функций и неоднозначность. Неоднозначность возникает тогда, когда компилятор не может определить различие между двумя перегруженными функциями. Инструкции, создающие неоднозначность, являются ошибочными, а программы, которые их содержат, скомпилированы не будут. Основной причиной неоднозначности в *C++* является автоматическое преобразование типов. В *C++* делается попытка автоматически преобразовать тип аргументов, используемых для вызова функции, в тип параметров, определенных функцией.

int myfunc(double d);

. . .

cout << myfunc('c'); // Ошибки нет, выполняется преобразование типов

Как отмечено в комментарии, ошибки здесь нет, поскольку *C++* автоматически преобразует символ *'c'* в его *double*-эквивалент. Автоматическое преобразование типов является главной причиной неоднозначности.

Литература:

Воробейкина, И.В. Технологии и методы программирования: учеб. пособие. Часть I / И.В. Воробейкина. – Калининград: Изд-во БГАРФ, 2021. – 108 с.

Глава 5. Функции. Основы.

Глава 6. Функции: ссылки, перегрузка и использование аргументов по умолчанию.

Контрольные вопросы для самопроверки:

1. Чем передача аргументов по значению отличается от передачи аргументов по ссылке?
2. Что такое аргументы по умолчанию?
3. Как создается ограниченный массив?
4. Расскажите о перегрузке функций.

19.3. Задание к лабораторной работе

Откомпилируйте программы из п. 19.4.

19.4. Методические указания и порядок выполнения работы

В следующей программе функция *cube()* преобразует значение каждого элемента массива в куб этого значения. При вызове функции *cube()* в качестве

первого аргумента необходимо передать адрес массива значений, подлежащих преобразованию, а в качестве второго – его размер.

```
#include <iostream>
using namespace std;
void cube(int *n, int num)
{
while(num)
        {
        *n = (*n) * (*n)* (*n);
        num--;
        n++;
        }
}
int main()
{ int i,
nums[10];
for(i=0; i<10; i++) nums[i]=i+1;
cout<< "Исходное содержимое массива: ";
for(i=0; i<10; i++) cout<< nums[i] << ' ';
cout << '\n';
cube(nums, 10); // Вычисляем кубы значений.
cout << "Измененное содержимое: ";
for(i=0; i<10; i++) cout<< nums[i] << ' ';
return 0; }
```

Результаты выполнения этой программы:

Исходное содержимое массива: 1 2 3 4 5 6 7 8 9 10

Измененное содержимое: 1 8 27 64 125 216 343 512 729 1000

Как видите, после обращения к функции *cube()* содержимое массива *nums* изменилось: каждый элемент стал равным кубу исходного значения. Элементы массива *nums* были модифицированы инструкциями, составляющими тело функции c*ube(),* поскольку ее параметр *n* указывает на массив *nums*.

В следующей программе показан один из возможных вариантов реализации функции *strlen().*

```
// Нестандартная реализация функции strlen().
int mystrlen(char *str)
{
int i;
for(i=0; str[i]; i++); // Находим конец строки
return i;
}
```

```
#include <iostream>
using namespace std;
int mystrlen(char *str);
int main()
{ cout << "Длина строки ПРИВЕТ ВСЕМ равна: ";
cout << mystrlen("ПРИВЕТ ВСЕМ");
return 0; }
```

Результат выполнения этой программы:

Длина строки ПРИВЕТ ВСЕМ равна: 11

Рассмотрим следующую программу, в которой используется стандартная библиотечная функция *abs()*.

```
#include <iostream>
#include <cstdlib>
using namespace std;
int main()
{
int i;
i = abs(-10); // строка 1
cout << abs(-23); // строка 2
abs(100); // строка 3
return 0;
}
```

Функция *abs()* возвращает абсолютное значение своего целочисленного аргумента. Она использует заголовок *cstdlib*. В строке *1* значение, возвращаемое функцией *abs()*, присваивается переменной *i*. В строке *2* значение, возвращаемое функцией *abs()*, ничему не присваивается, но используется инструкцией *cout*. В строке *3* значение, возвращаемое функцией *abs()*, теряется, поскольку не присваивается никакой другой переменной и не используется как часть выражения.

```
// Новая версия функции find_substr() возвращает указатель на подстроку
#include <iostream>
using namespace std;
/* Функция возвращает указатель на искомую подстроку или нуль, если таковая не будет найдена */
char *find_substr(char *sub, char *str)
        {
        int t;
        char *p, *p2, *start;
        for(t=0; str[t]; t++)
                {
```

```
        p = &str[t]; // установка указателей
        start = p;
        p2 = sub;
        while(*p2 && *p2==*p)
            { // проверка совпадения
            p++; p2++;
            }
            /* Если достигнут конец p2-подстроки, то эта подстрока была найдена. */ if(!*p2) return start;
            // Возвращаем указатель на начало найденной подстроки
            }
    return 0; // подстрока не найдена
    }
int main()
{
char *substr;
substr = find_substr("три", "один два три четыре");
cout << "Найденная подстрока: " << substr;
return 0;
}
```

При выполнении этой версии программы получен следующий результат:

Найденная подстрока: три четыре

В данном случае, когда подстрока *три* была найдена в строке *один два три четыре*, функция *find_substr()* возвратила указатель на начало искомой подстроки *три*, который в функции *main()* был присвоен переменной *substr*. Таким образом, при выводе значения *substr* на экране отобразился остаток строки, т.е. *три четыре*.

Рассмотрим следующую функцию.

```
#include <iostream>
using namespace std;
int sqr_it(int x) { x = x*x; return x; }
int main()
{
int t=10;
cout << sqr_it(t) << ' ' << t;
return 0;
}
```

В этом примере значение аргумента *t=10*, передаваемого функции *sqr_it()*, копируется в параметр *x*. При выполнении присваивания *x = x*x* изменяется

лишь локальная переменная *x*. Переменная *t*, используемая при вызове функции *sqr_it()*, по-прежнему будет иметь значение *10*, и на нее никак не повлияют операции, выполняемые в этой функции. Следовательно, после запуска рассматриваемой программы на экране будет выведен такой результат: *100 10*.

Рассмотрим функцию *swap()*; она меняет значения двух переменных, на которые указывают ее аргументы.

```
void swap(int *x, int *y)
{
int temp;
temp = *x; // Временно сохраняем значение, расположенное по адресу x
*x = *y; // Помещаем значение, хранимое по адресу y, в адрес x
*y = temp; // Помещаем значение, которое раньше хранилось по адресу x, в адрес y
}
```

Здесь параметры **x* и **y* являются адресами переменных *x* и *y*, используемых при вызове функции *swap()*. Следовательно, при выполнении этой функции будет совершен **реальный** обмен содержимым переменных, используемых при ее вызове. Поскольку функция *swap()* в качестве аргументов ждет два указателя, ее необходимо вызывать с адресами переменных, значения которых нужно обменять.

```
#include <iostream>
using namespace std;
// Обмен аргументами
void swap(int *x, int *y)
        {
        int temp;
        temp = *x;
        *x = *y;
        *y = temp;
        }
int main()
{
int i = 10, j = 20;
cout << "Исходные значения переменных i и j: "<<i<<' ' << j << '\n';
swap(&j, &i); // Вызываем swap() с адресами переменных i и j
cout << "Значения переменных i и j после обмена: "<<i << ' ' <<j;
return 0;
}
```

Результаты выполнения этой программы:

Исходные значения переменных i и j: 10 20

Значения переменных i и j после обмена: 20 10

В этом примере переменной *i* было присвоено начальное значение *10*, а переменной *j* – начальное значение *20*. Затем была вызвана функция *swap()* с адресами переменных *i* и *j*. Для получения адресов здесь используется унарный оператор &. Следовательно, функции *swap()* при вызове были переданы адреса переменных *i* и *j*, а не их значения. После выполнения функции *swap()* переменные *i* и *j* обменялись своими значениями.

В следующей программе функция *f()* принимает один ссылочный параметр типа *int*.

```
// Использование ссылочного параметра
#include <iostream>
using namespace std;
void f(int &i)
        { i = 10; // Модификация аргумента, заданного при вызове
        }
int main()
{
int val = 1;
cout << "Старое значение переменной val: " << val << '\n';
f(val); // Передаем адрес переменной val функции f()
cout << "Новое значение переменной val: " << val << '\n';
return 0;
}
```

При выполнении этой программы получаем такой результат:

Старое значение переменной val: 1

Новое значение переменной val: 10

Рассмотрим объявление параметра *i*. Его имени предшествует символ &, который превращает переменную *i* в ссылочный параметр. Инструкция *i = 10;* **не присваивает** переменной *i* значение *10*. В действительности значение *10* присваивается переменной, на которую ссылается переменная *i* (в нашей программе ею является переменная *val*). В этой инструкции не используется оператор, который необходим при работе с указателями. Применяя ссылочный параметр, вы тем самым уведомляете компилятор о передаче адреса (т.е. указателя), и компилятор автоматически разыменовывает его. Поскольку переменная *i* была объявлена как ссылочный параметр, компилятор автоматически передает функции *f()* адрес любого аргумента, с которым вызывается эта функция. Таким образом, в функции *main()* инструкция

f(val); // Передаем адрес переменной val функции f()

передает функции *f()* адрес переменной *val* (а не ее значение). При вызове функции *f()* не нужно предварять переменную *val* оператором &. Более того, это

было бы ошибкой! Поскольку функция *f()* получает адрес переменной *val* в форме ссылки, она может модифицировать значение этой переменной. Чтобы проиллюстрировать реальное применение ссылочных параметров, перепишем пример *6.1.2* с использованием ссылок. В следующей программе обратите внимание на то, как функция *swap()* объявляется и вызывается.

```
#include <iostream>
using namespace std;
/* Функция swap() с использованием ссылочных параметров.
Здесь функция swap() определяется в расчете на вызов по ссылке, а не на вызов по значению. Поэтому она может выполнить обмен значениями двух аргументов, с которыми она вызывается. */
void swap(int &x, int &y)
{
int temp;
temp = x; // Сохраняем значение, расположенное по адресу x
x = y; // Помещаем значение, хранимое по адресу y, в адрес x
y = temp; // Помещаем значение, которое раньше хранилось по адресу x, в адрес y
}
int main()
{
int i = 10, j = 20;
cout<<"Исходные значения i и j: "<< i <<' ' << j << '\n';
swap (j, i);
cout<<"Значения i и j после обмена: "<< i <<' ' << j;
return 0;
}
```

Обратите внимание на то, что объявление *x* и *y* ссылочными параметрами избавляет вас от необходимости использовать оператор при организации обмена значениями. Компилятор автоматически генерирует адреса аргументов, используемых при вызове функции *swap()* и автоматически разыменовывает ссылки *x* и *y*.

```
// Неоднозначность вследствие перегрузки функций
#include <iostream>
using namespace std;
float myfunc(float i) { return i; }
double myfunc(double i) { return -i; }
int main()
{
// Неоднозначности нет, вызывается функция myfunc(double)
```

```
cout << myfunc (10.1) << " ";
// Неоднозначность
cout << myfunc(10);
return 0;
}
```

Здесь благодаря перегрузке функция *myfunc()* может принимать аргументы либо типа *float*, либо типа *double*. При выполнении строки кода

```
cout << myfunc (10.1) << " ";
```

не возникает никакой неоднозначности: компилятор обеспечивает вызов функции *myfunc(double),* поскольку все литералы с плавающей точкой в *C++* автоматически получают тип *double*. Но при вызове функции *myfunc()* с аргументом, равным целому числу *10*, в программу вносится неоднозначность, поскольку компилятору неизвестно, в какой тип ему следует преобразовать этот аргумент – *float* или *double*. Оба преобразования допустимы. В такой неоднозначной ситуации будет выдано сообщение об ошибке, и программа не скомпилируется. Неоднозначность в программе вызвана не перегрузкой функции *myfunc(),* объявленной дважды для приема *double*- и *float*-аргумента, а использованием при конкретном вызове функции *myfunc()* аргумента неопределенного для преобразования типа.

19.5. Индивидуальное задание

Ссылочный тип в качестве типа значения, возвращаемого функцией, можно применить для создания ограниченного массива. Известно, что при выполнении *C++*-кода проверка нарушения границ при индексировании массивов не предусмотрена. Значит, может быть задан индекс, превышающий размер массива. Однако путем создания ограниченного, или безопасного, массива выход за его границы можно предотвратить. При работе с таким массивом любой выходящий за установленные границы индекс не допускается для индексирования массива.

Создайте ограниченный массив типа *int*, введите в него данные и выведите их на экран.

19.6. 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.

20. ЛАБОРАТОРНАЯ РАБОТА № 19. РЕКУРСИВНЫЕ ФУНКЦИИ

20.1. Общие сведения

Цель: изучить методы работы с рекурсивными функциями.

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

Условия допуска к выполнению: показать конспект по теоретической подготовке и решить подготовительный пример.

Критерии положительной оценки: показать выполненную работу: запрограммированный пример; ответить на вопросы преподавателя.

20.2. Теоретическое введение

Рекурсия (или циклическое определение) представляет собой процесс определения чего-либо на собственной основе. В программировании под рекурсией понимается вызов функцией самой себя. Функцию, которая вызывает саму себя, называют рекурсивной. Классическим примером рекурсии является вычисление факториала числа.

Когда функция вызывает сама себя, в системном стеке выделяется память для новых локальных переменных и параметров, и код функции с самого начала выполняется с этими новыми переменными. Рекурсивный вызов не создает новой копии функции. Новыми являются только аргументы. При возвращении каждого рекурсивного вызова из стека извлекаются старые локальные переменные и параметры, и выполнение функции возобновляется с внутренней точки ее вызова. О рекурсивных функциях можно сказать, что они *выдвигаются* и *задвигаются*. В большинстве случаев использование рекурсивных функций не дает значительного сокращения объема кода. Кроме того, рекурсивные версии многих процедур выполняются медленнее, чем их итеративные эквиваленты, из-за дополнительных затрат системных ресурсов, связанных с многократными вызовами функций. Слишком большое количество рекурсивных обращений к функции может вызвать переполнение стека. Поскольку локальные переменные и параметры сохраняются в системном стеке и каждый новый вызов создает новую копию этих переменных, может настать момент, когда память стека будет исчерпана. В этом случае могут быть разрушены другие данные. Но если рекурсия построена корректно, об этом вряд ли стоит волноваться.

Основное достоинство рекурсии состоит в том, что некоторые типы алгоритмов рекурсивно реализуются проще, чем их итеративные эквиваленты. При написании рекурсивной функции необходимо включить в нее инструкцию проверки условия, которая бы обеспечивала выход из функции без выполнения рекурсивного вызова. Если этого не сделать, то, вызвав однажды такую функцию,

из нее уже нельзя будет вернуться. При работе с рекурсией это самый распространенный тип ошибки. Поэтому при разработке программ с рекурсивными функциями стоит использовать инструкции *cout*, чтобы быть в курсе того, что происходит в конкретной функции, и иметь возможность прервать ее работу в случае обнаружения ошибки.

Литература:

Воробейкина, И.В. Технологии и методы программирования: учеб. пособие. Часть I / И.В. Воробейкина. – Калининград: Изд-во БГАРФ, 2021. – 108 с.

Глава 5. Функции. Основы.

Контрольные вопросы для самопроверки:

Расскажите, что такое рекурсия и приведите пример рекурсивной программы.

20.3. Задание к лабораторной работе

Откомпилируйте программы из п. 19.4.

20.4. Методические указания и порядок выполнения работы

Рекурсивный способ вычисления факториала числа демонстрируется в следующей программе. Для сравнения сюда же включен и его нерекурсивный (итеративный) эквивалент.

```
#include <iostream>
using namespace std;
// Рекурсивная версия
int factr(int n)
        {
        int answer;
        if(n==1) return(1);
        answer = factr(n-1)*n;
         return(answer);
        }
// Итеративная версия.
int fact(int n)
        {
        int t, answer;
        answer =1;
        for(t=1; t<=n; t++) answer = answer* (t);
        return (answer);
```

```
        }
int main()
{
// Использование рекурсивной версии
cout << "Факториал числа 4 равен " << factr(4); cout << '\n';
// Использование итеративной версии
cout << "Факториал числа 4 равен " << fact(4); cout << '\n';
return 0;
}
```

Нерекурсивная версия функции *fact()* довольно проста: в ней используется цикл, в котором организовано перемножение последовательных чисел, начиная с *1* и заканчивая числом *n*, заданным в качестве параметра; на каждой итерации цикла текущее значение управляющей переменной цикла умножается на текущее значение произведения, полученное в результате выполнения предыдущей итерации цикла. Рекурсивная функция *factr()* несколько сложнее. Если она вызывается с аргументом, равным *1*, то сразу возвращает значение *1*. В противном случае она возвращает произведение *factr(n-1)*n*. Для вычисления этого выражения вызывается метод *factr()* с аргументом *n-1*. Этот процесс повторяется до тех пор, пока аргумент не станет равным *1*, после чего вызванные ранее методы начнут возвращать значения. Например, при вычислении факториала числа *2* первое обращение к методу *factr()* приведет ко второму обращению к тому же методу, но с аргументом, равным *1*. Второй вызов метода *factr()* возвратит значение *1*, которое будет умножено на *2* (исходное значение параметра *n*). Можно вставить в функцию *factr()* инструкции *cout*, чтобы показать промежуточные результаты.

Рассмотрим функцию *reverse()*, которая использует рекурсию для отображения своего строкового аргумента в обратном порядке.

```
// Отображение строки в обратном порядке с помощью рекурсии.
#include <iostream>
using namespace std;
// Вывод строки в обратном порядке
void reverse(char *s)
        {
        if(*s)reverse(s+1);
        else return;
        cout << *s;
        }
int main() {
char str[] = "Это тест";
reverse(str);
```

```
return 0;
}
```

Функция *reverse()* проверяет, не передан ли ей в качестве параметра указатель на нуль, которым завершается строка. Если нет, то функция *reverse()* вызывает саму себя с указателем на следующий символ в строке. Этот процесс повторяется до тех пор, пока той же функции не будет передан указатель на нуль. Когда обнаружится символ конца строки, пойдет обратный процесс, т.е. вызванные ранее функции начнут возвращать значения, и каждый возврат будет сопровождаться «довыполнением» метода, т.е. отображением символа *s*. В результате исходная строка посимвольно отобразится в обратном порядке.

20.5. Индивидуальное задание

Вариативность не предполагается.

Используя рекурсию, найти все числа Фибоначчи, порядковые номера которых меньше либо равны введенного с экрана числа.

Пример рекурсивной функции для нахождения чисел Фибоначчи:

```
unsigned long fibonacci(unsigned long entered_number) // функция принимает один аргумент
{
    if ( entered_number == 1 || entered_number == 2) // частный случай
        return (entered_number -1); // ряд чисел Фибоначчи всегда начинается с 0, 1, ...
    return fibonacci(entered_number-1) + fibonacci(entered_number-2); // формула поиска n-го числа, например найти восьмое по счёту число, и оно равно 7-е + 6-е
}
```

20.6. 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.

21. ЛАБОРАТОРНАЯ РАБОТА № 20. ИСПОЛЬЗОВАНИЕ УКАЗАТЕЛЯ ДЛЯ ОБЕСПЕЧЕНИЯ ВЫЗОВА ПО ССЫЛКЕ

21.1. Общие сведения

Цель: познакомиться с ссылками в C++.

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

Условия допуска к выполнению: показать конспект по теоретической подготовке и решить подготовительный пример.

Критерии положительной оценки: показать выполненную работу: запрограммированный пример; ответить на вопросы преподавателя.

21.2. Теоретическое введение

Вызов по значению можно «вручную» заменить вызовом по ссылке. В этом случае функции будет передаваться адрес аргумента (т.е. указатель на аргумент). Это позволит внутреннему коду функции изменить значение аргумента, которое хранится вне функции. Примером такого «дистанционного» управления значениями переменных является вызов функции с указателями.

Рассмотрим функцию *swap()*; она меняет значения двух переменных, на которые указывают ее аргументы.

```
void swap(int *x, int *y)
{
int temp;
temp = *x; // Временно сохраняем значение, расположенное по адресу x
*x = *y; // Помещаем значение, хранимое по адресу y, в адрес x
*y = temp; // Помещаем значение, которое раньше хранилось по адресу x, в адрес y
}
```

Здесь параметры **x* и **y* являются адресами переменных *x* и *y*, используемых при вызове функции *swap()*. Следовательно, при выполнении этой функции будет совершен **реальный** обмен содержимым переменных, используемых при ее вызове. Поскольку функция *swap()* в качестве аргументов ждет два указателя, ее необходимо вызывать с адресами переменных, значения которых нужно обменять.

Ссылочный параметр автоматически получает адрес соответствующего аргумента. Несмотря на возможность вручную организовать вызов по ссылке с помощью оператора получения адреса, такой подход не всегда удобен. Во-первых, он вынуждает программиста выполнять все операции с использованием указателей. Во-вторых, вызывая функцию, программист должен не забыть передать ей адреса аргументов, а не их значения. Вместо этого можно сориентировать ком-

пилятор на автоматическое использование вызова по ссылке для одного или нескольких параметров конкретной функции. Такая возможность реализуется с помощью ссылочного параметра (reference parameter). При использовании ссылочного параметра функции автоматически передается адрес (а не значение) аргумента. При выполнении кода функции, а именно, при выполнении операций над ссылочным параметром, обеспечивается его автоматическое разыменование, и поэтому программисту не нужно использовать операторы, работающие с указателями. Ссылочный параметр объявляется с помощью символа, который должен предшествовать имени параметра в объявлении функции. Операции, выполняемые над ссылочным параметром, оказывают влияние на аргумент, используемый при вызове функции, а не на сам ссылочный параметр.

Ссылочный параметр автоматически получает адрес соответствующего аргумента. Несмотря на возможность вручную организовать вызов по ссылке с помощью оператора получения адреса, такой подход не всегда удобен. Во-первых, он вынуждает программиста выполнять все операции с использованием указателей. Во-вторых, вызывая функцию, программист должен не забыть передать ей адреса аргументов, а не их значения. Вместо этого можно сориентировать компилятор на автоматическое использование вызова по ссылке для одного или нескольких параметров конкретной функции. Такая возможность реализуется с помощью ссылочного параметра (reference parameter). При использовании ссылочного параметра функции автоматически передается адрес (а не значение) аргумента. При выполнении кода функции, а именно при выполнении операций над ссылочным параметром, обеспечивается его автоматическое разыменование, и поэтому программисту не нужно использовать операторы, работающие с указателями. Ссылочный параметр объявляется с помощью символа, который должен предшествовать имени параметра в объявлении функции. Операции, выполняемые над ссылочным параметром, оказывают влияние на аргумент, используемый при вызове функции, а не на сам ссылочный параметр.

Литература:

Воробейкина, И.В. Технологии и методы программирования: учеб. пособие. Часть I / И.В. Воробейкина. – Калининград: Изд-во БГАРФ, 2021. – 108 с.

Глава 6. Функции: ссылки, перегрузка и использование аргументов по умолчанию.

21.3. Задание к лабораторной работе

Откомпилируйте программы из п. 21.4.

21.4. Методические указания и порядок выполнения работы

```
#include <iostream>
```

```
using namespace std;
// Обмен аргументами
void swap(int *x, int *y)
        {
        int temp;
        temp = *x;
        *x = *y;
        *y = temp;
        }
int main()
{
int i = 10, j = 20;
cout << "Исходные значения переменных i и j: "<<i<<' ' << j << '\n';
swap(&j, &i); // Вызываем swap() с адресами переменных i и j
cout << "Значения переменных i и j после обмена: "<<i << ' ' <<j;
return 0;
}
```

Результаты выполнения этой программы:

Исходные значения переменных i и j: 10 20

Значения переменных i и j после обмена: 20 10

В этом примере переменной *i* было присвоено начальное значение *10*, а переменной *j* – начальное значение *20*. Затем была вызвана функция *swap()* с адресами переменных *i* и *j*. Для получения адресов здесь используется унарный оператор &. Следовательно, функции *swap()* при вызове были переданы адреса переменных *i* и *j*, а не их значения. После выполнения функции *swap()* переменные *i* и *j* обменялись своими значениями.

21.5. Индивидуальное задание

Создайте двумерный ограниченный массив типа *int*, введите в него данные и выведите их на экран.

21.6. 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.

22. ЛАБОРАТОРНАЯ РАБОТА № 21. КЛАССЫ, СТРУКТУРЫ, ОБЪЕДИНЕНИЯ. КОНСТРУКТОРЫ И ДЕСТРУКТОРЫ

22.1. Общие сведения

Цель: познакомиться с классами в C++.

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

Условия допуска к выполнению: показать конспект по теоретической подготовке и решить подготовительный пример.

Критерии положительной оценки: показать выполненную работу: запрограммированный пример; ответить на вопросы преподавателя.

22.2. Теоретическое введение

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
```

Конструкторы и деструкторы

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.

```
#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.

Функцией, обратной конструктору, является деструктор. Эта функция вызывается при удалении объекта. Обычно при работе с объектом в момент его удаления должны выполняться некоторые действия. Например, при создании объекта для него выделяется память, которую необходимо освободить при его удалении. Имя деструктора совпадает с именем класса, но с символом ~ (тильда) в начале.

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

```
        myclass(); // конструктор
        ~myclass(); // деструктор
        void show();
} ;
myclass::myclass()
        {
        cout << "Содержимое конструктора\n";
        a = 10;
        }
};
myclass::~myclass()
        {cout << "Удаление...\n";}
void myclass::show()
        {cout << a << "\n";}
int main()
{
myclass ob;
ob.show();
return 0;
}
```

Деструктор класса вызывается при удалении объекта. Локальные объекты удаляются тогда, когда они выходят из области видимости. Глобальные объекты удаляются при завершении программы. Адреса конструктора и деструктора получить невозможно.

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

Конструкторы с параметрами

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

```
#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.

```
#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;
}
```

Литература:

Воробейкина, И.В. Технологии и методы программирования: учеб. пособие / И.В. Воробейкина. – Калининград: Изд-во БГАРФ, 2021. – 155 с.

Глава 1. Классы.

Контрольные вопросы для самопроверки:

1. Какие функции выполняет конструктор и когда он вызывается?
2. Может ли конструктор выполнять другие функции, кроме инициализации полей класса?

22.3. Задание к лабораторной работе

Откомпилируйте программы из п. 22.4.

22.4. Методические указания и порядок выполнения работы

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

```
#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.

В следующем примере представлена улучшенная версия класса *stack*, где для автоматической инициализации объекта стека при его создании используется конструктор.

```
#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.

Далее 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.

22.5. Индивидуальное задание

Вариативность не предполагается.

1. 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.

2. 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.

3. 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.*

4. 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.

22.6. 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.

23. ЛАБОРАТОРНАЯ РАБОТА № 22. ДРУЖЕСТВЕННЫЕ ФУНКЦИИ

23.1. 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: показать конспект по теоретической подготовке и решить подготовительный пример.

Критерии положительной оценки: показать выполненную работу: запрограммированный пример; ответить на вопросы преподавателя.

23.2. 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*. В качестве аргументов дружественным функциям передаются объекты классов, для которых эти функции дружественны.

```
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;
}
```

Поскольку дружественная функция не является членом класса, то 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. Таким образом, в отличие от функции-члена *myclass*, в которой можно непосредственно обращаться к переменным *n* и *d*, дружественная функция имеет доступ к этим переменным только через объект, который объявлен внутри функции или передан ей.

Дружественная функция не наследуется. Поэтому функция, дружественная базовому классу, не является таковой для производного класса. Функция может быть дружественна более, чем к одному классу.

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

Литература:

Воробейкина, И.В. Технологии и методы программирования: учеб. пособие / И.В. Воробейкина. – Калининград: Изд-во БГАРФ, 2021. – 155 с.

Глава 4. Дружественные функции.

Контрольные вопросы для самопроверки:

1. В какиз случаях целесообразно применять дружественные функции?
2. Может ли функция, являясь членом одного класса быть дружественной другому классу?
3. Наследуются ли дружественные функции?

23.3. Задание к лабораторной работе

Откомпилируйте программы из п. 23.4.

23.4. Методические указания и порядок выполнения работы

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 так называемое *предварительное объявление* или *ссылку 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 – способ сообщить компилятору имя класса без его фактического объявления. Это делается следующим образом: перед первым использованием имени класса пишется строка:

```
class имя_класса;
```

В предыдущей программе предварительным объявлением является инструкция *class truck;* после которой класс *truck* можно использовать в прототипе дружественной функции *sp_greater().*

Функция может быть членом одного класса и быть дружественной другому.

```
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 функциця *sp_greater()* – член класса *car*.

При упоминании в программе члена класса можно полностью (с именем класса и оператором расширения области видимости) задать его имя. Но при использовании объекта для вызова функции-члена или для доступа к переменной-члену полное имя излишне и употребляется редко. Например, инструкция *t=c1.sp_greater(t1);* может быть написана с избыточным указанием оператора расширения области видимости и именем класса: *t=c1.car::sp_greater(t1);*

Поскольку объект *c1* имеет тип *car*, компилятор уже и так «знает», что функция *sp_greater()* – член класса *car*.

23.5. Индивидуальное задание

Вариативность не предполагается.

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.

23.6. 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.

24. ЗАКЛЮЧЕНИЕ

В данном пособии на примерах рассматриваются различные методы и приемы написания программ, которые можно применять в языках HTML, JavaScript и C++. Каждое занятие состоит из теоретических положений, упражнений. Пособие рассчитано как на начинающих программистов, так и на 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 «Технологии и методы программирования» для студентов специальности 10.05.03 ИБАС.

25. ЛИТЕРАТУРА

1. Спейнаур, С. Справочник WEB-мастера. Практическое пособие / С. Спейнаур, В. Куэрсиа; пер. с англ. – Киев: Изд. группа «BHV», 1997. – 367 с.

2. Беляев, С. А. Разработка игр на языке JavaScript: учеб. пособие / С. А. Беляев. – Санкт-Петербург: Лань, 2018. – 128 с.

3. Шилдт, Г. Самоучитель C++ / Г. Шилдт; пер. с англ. – 3-е изд. – Санкт-Петербург: БХВ-Петербург, 2011. – 688 с.

4. Эккель, Б. Философия C++. Практическое программирование / Б. Эккель. – Санкт-Петербург: Питер, 2014. – 608 с.

5. Шилдт, Г. Полный справочник по C++ / Г. Шилдт. – 4-е издание; пер. с англ. – Москва: Издательский дом «Вильямс», 2006. – 800 с.

6. Воробейкина, И.В. Технологии и методы программирования: учеб. пособие / И.В. Воробейкина. – Калининград: Изд-во БГАРФ, 2021. – 155 с.

7. Воробейкина, И.В. Технологии и методы программирования. Лабораторный практикум / И.В. Воробейкина. – Калининград: Изд-во БГАРФ, 2019. – 97 с.

Локальный электронный методический материал

Ирина Владимировна Воробейкина

ТЕХНОЛОГИИ И МЕТОДЫ ПРОГРАММИРОВАНИЯ

Редактор М. А. Дмитриева

Уч.-изд. л. 7,1. Печ. л. 9,8.

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