МИНИСТЕРСТВО ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ И КОММУНИКАЦИЙ РЕСПУБЛИКИ УЗБЕКИСТАН

КАРШИНСКИЙ ФИЛИАЛ ТАШКЕНТСКОГО УНИВЕРСИТЕТА ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ

факультет

КОМПЬЮТЕРНЫЙ ИНЖИНИРИНГ

кафедра

ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА

на тему:***«Графические программирования на языке С++»***

Выпускник: _____ И.Т. Тиллаев.

подпись

Руководитель: _____ доц. А Шайкулов.

подпись

“Допущен к защите”

Заведующий кафедры

“Информационных технологий”

д.э.н. Х.С.Мухитдинов _____

«_____»_____2015й

Отправлен в ГАК для защиты:

декан факультета

“Компьютерный нжиниринг”

С.Б.Давронов_____

“_____»_____2015й

Карши 2015

МИНИСТЕРСТВО ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ И КОММУНИКАЦИЙ РЕСПУБЛИКИ УЗБЕКИСТАН

КАРШИНСКИЙ ФИЛИАЛ ТАШКЕНТСКОГО УНИВЕТСИТЕТА ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ

факультет

КОМПЬЮТЕРНЫЙ ИНЖИНИРИНГ

Направление "Информатика и информационные технологии"

«Утверждаю»

Зав.кафедры ИТ"

*Д.э.н.Х.С.Мухитдинов*_____

«____» _____ 2015й

ЗАДАНИЕ

По выпускной квалификационной работе

Студент: **Тиллаев Ислом**

1. Тема ВКР: ***«Графические программирования на языке С++»***

2. Утверждено указом № _____ Каршинского филиала ТУИТ

3. Срок сдачи ВКР _____ *2015й*_____

4. Начальная информация для квалификационной работы: *Теория, методология, технология программирование, Объектно-ориентированное язык программирования С++, Графические программирования на языке С++, техника безопасности при работе с компьютером,*

5.Содержание расчётно-пояснительной записки(перечень подлежащих к разработке вопросов) Графические программирования на языке С++. Технология и методика создания графического программирования.Безопасность жизнедеятельности.

6. Графические материалы, слайды:

7. Консультанты по ВКР:

№	Название раздела	Консультант	Подпись, число	
			Дата для задания	Подпись Консультанта
1	Физические процессы световых волн	Дусаяров А.		

8. Календарный график по выполнению квалификационной работы:

№	Раздела ВКР	Объём ВКР, страницы	По отношению к общему	Заметка о выполнении	Примечания
	Введение				
I	**Теория, методология,**				
1.1.	Теория программирования				
1.2.	Методология программирования				
1.3.	Технология программирования				
II	**Объектно-ориентированное язык программирования С++**				

2.1.	Алфавит и идентификаторы				
2.2.	Массивы				
2.3.	Функция				
2.4.	Классы				
2.5.	Полиморфизм				
III	**Графические программирования на языке**				
3.1	Химическая реакция первого				
3.2					
3.3	Негативная проводимость полупроводника				
3.4	Закон изменения коллекторного тока во времени (t)				
IV	**Безопасность**				
	Заключение				
	Список литературы				
	Приложение				
	Всего				

.Руководитель по ВКР: _____ доц. АШойкулов.

Студент: _____ И Тиллаев.

С О Д Е Р Ж А Н И Е

Заключение

Список литературы

Введение

Актуальность работы: Развитие всех современных точных естественных, общественно гуманитарных наук требует использования современных теорий методологии технологий, математического, графического, физического биологического и других видов моделирования.

Настоящая дипломная работа посвящена графическому моделированию графических программ на современном языке объектно-ориентированного программирования C++ .

В 1 разработаны главе графические программы

1. Химические реакции первого порядка
2. Негативная проводимость полупроводников
3. Закон изменения коллекторного тока

Которые послужит в развитии теории программирования могут быть использованы и в учебном процессе средних школ, колледжей, высших учебных заведений.

ГЛАВА 1:ТЕОРИЯ, МЕТОДОЛОГИЯ, ТЕХНОЛОГИЯ ПРОГРАММИРОВАНИЕ

1.1 ТЕОРИЯ ПРОГРАММИРОВАНИЯ.

Модульное программирование

Модульное программирование – это такой способ программирования при котором вся программа разбивается на группу компонентов называемых модулями, причем каждый из них имеет свой контролируемый размер четкое назначение и детально проработанный интерфейс с внешней средой. Единственная альтернатива модульности – монолитная программа что конечно, неудобна. Таким образом, наиболее интересный вопрос при изучении модульности – определение критерия разбиения на модули.

Концепции модульного программирования. В основе модульного программирования лежат три основных концепции:

Принцип утаивания информации Парнаса. Всякий компонент утаивает единственное проектное решение, т.е. модуль служит для утаивания информации. Подход к разработке программ заключается в том, что сначала формируется список проектных решений, которые особенно трудно принять или которые скорее всего будут меняться. Затем определяются отдельные модули каждый из которых реализует одно из указанных решений.

Аксиома модульности Коуэна. Модуль – независимая программная единица, служащая для выполнения некоторой определенной функции программы и для связи с остальной частью программы. Программная единица должна удовлетворять следующим условиям:

- Блочность организации, т.е. возможность вызвать программную единицу избло любой степени вложенности;
- Синтаксическая обособленность, т. е. выделения модуля в тексте синтаксическими элементами;
- Семантическая независимость. е. независимость от места, где программная единица вызвана;
- Общность данных, т. е. наличие собственных данных, сохраняющихся при каждом обращении;
- Полнота определения, т. е. самостоятельность программной единицы.

Сборочное программирование Цейтина. Модули – это программные кирпичи, из которых строится программа. Существуют три основные предпосылки к модульному программированию:

- Стремление к выделению независимой единицы программного знания. В идеальном случае всякая идея (алгоритм) должна быть оформлена в виде модуля;
- Потребность организационного расчленения крупных разработок;
- Возможность параллельного исполнения модулей (в контексте параллельного программирования).

Определения модуля и его примеры.

Приведем несколько дополнительных определений модуля.

- модуль – это совокупность команд, к которым можно обратится по имени.
- Модуль – это совокупность операторов программы имеющая граничные элементы и идентификатор (возможна агрегатный). Функциональная спецификация модуля должна включать:

- Синтаксическую спецификацию его входов, которая должна позволять построить на используемом языке программирования синтаксически правильное обращение к нему;
- Описание семантики функций, выполняемых модулем по каждому из его входов.

Разновидности модулей. Существуют три основные разновидности модулей:

1) «*маленькие*» (*функциональные*) модули, реализующие, как правило, одну какую-либо определенную функцию. Основным и простейшим модулем практически во всех языках программирования является процедура или функция.

2) «*Средние*» (*информационные*) модули реализующие, как правило несколько операций или функций над одной и той же структурой данных (информационным объектом), которая считается неизвестной вне этого модуля. Примеры «средних» модулей в языках программирования:

a) задачи в языке программирования Ada;

b) кластер в языке программирования CLU;

c) классы в языках программирования C++ и Java.

3) «*Большие*» (*логические*) *модули,* объединяющие набор «средних» или «маленьких» модулей. Примеры «больших» модулей в языках программирования:

a) модуль в языке программирования Modula-2;

b) пакеты в языках программирования AdauJava.

Набор **характеристик модуля** предложен Майерсом [Майерс 1980]. Он состоит из следующих конструктивных характеристик:

1) Размер модуля;

В модуле должна быть 7(+/-2) конструкций (например операторов для функций или функций для пакета). Это число берется на основе представлений психологов средним оперативном буфере памяти человека. Символьные образы в человеческом мозгу объединяется в «чанки» - наборы фактов и связей между ними запоминаемые и извлекаемые как единой целое. В каждой момент времени человек может обрабатывать не более 7 чанков. Модуль (функция) не должен превышать 60 строк. В результате его можно поместить на одну страницу распечатки или легко просмотреть на экране монитора

2) Прочности (связности) модуля;

Существует гипотеза о глобальных данных, утверждающая, что глобальные данные вредны и опасны. Идея глобальных данных дискредитирует себя так же как и идея оператора безусловного перехода **goto.**Локальность данных дает возможность легко читать и понимать модули, а так же легко удалять их из программы.

Связность (прочность) модуля (cohesion) – мера независимости его частей. Чем веще связность модуля – тем лучше, тем больше связей по отношению к оставшейся части программы он упрятывает в себе. Можно выделить типы связности, приведенные ниже.

Функциональная связность. Модуль с функциональной связности реализует одну какую-либо определенную функцию и не может быть разбит на 2 модуля с теми же типами связностей.

Последовательная связность. Модуль с такой связностью может быть разбит на последовательные части выполняющие независимые функции, но совместно реализующие единственную функцию. Например, один и тот же модуль может быть использован сначала для оценки, а затем для обработки данных.

Информационная (коммуникативная) связность. Модуль с информационной связностью – это модуль, который выполняет несколько операции или функций над одной и той же структурой данных

(информационным объектом), которая считается неизвестной вне этого модуля. Эта информационная связность применяется для реализации абстрактных типов данных. Обратим внимание на то, что средства для задания информационного прочных модулей отсутствовали в ранних языках программирования (напримерFORTRANи даже в оригинальной версии языка Pascal). И только позже, в языке программирования Ada, появился пакет – средство задание информационно прочного модуля

3) Сцепления модуля с другими модулями;

Сцепление (coupling) – мера относительной независимости модуля от других модулей. Независимые модули могут быть модифицированы без переделки других модулей. Чем слабее сцепление модуля, тем лучше. Рассмотрим различные типы сцепления.

Независимые модули – это идеальный случай. Модули ничего не знают друг о друге. Организовать взаимодействие таких модулей можно, зная их интерфейс и соответствующим образом перенаправив выходные данные одного модуля на вход другого. Достичь такого сцепления сложно да и не нужно поскольку сцепление по данным (параметрическое сцепление) является достаточна хорошим.

Сцепление *по данным* (параметрическое) – это сцепление, когда данные передаются модулю, как значения его параметров, либо как результат его обращения к другому модулю для вычисления некоторой функции. Этот вид сцепления реализуется в языках программирования при обращении к функциям (процедурам). Две разновидности этого сцепления определяются характером данным.

- Сцепление по простым элементам данных
- Сцепление по структуре данных. В этом случае оба модуля должны знать о внутренней структуре данных.

4) Рутинности (идемпотентность, независимость, от предыдущих обращений) модуля

Рутинность – это независимость модуля от предыдущих обращений к нему (от предыстории). Будем называть модуль рутинном, если в результат его работы зависит только от количество переданных параметров (а не от количество обращения)

Структурное программирование

Структурное программирование(СП) возник как вариант решения проблемы уменьшения сложности разработки программного обеспечения. В начале эры программирования работы программиста ничем не регламентировалось. Решаемые задачи не отличались размахом и масштабностью, используется в основе машинно-ориентированным языке и близкие к ним зек типа Ассемблера разрабатываемые программы редко достигали значительных размеров не ставились жесткие ограничения на время их разработки.

Методология структурного императивного программирования.– подход заключающийся в задании хорошей топологии императивных программ, в том числе отказе от использования глобальных данных и оператора безусловного перехода, разработке модулей с сильной связностью и обеспечении их независимости от других модулей.

метод объектно-ориентированного программирования

Метод структурного программирования оказался эффективен при написании программ «ограниченной сложности». Однако с возрастанием сложности реализуемых программных проектов и, соответственно, объема

кода создаваемых программ возможности метода структурного программирования оказались недостаточными.

Основной причиной возникших проблем можно считать то что в программе не отражалась непосредственно структура явлений и понятий реального мира и связей межу ними. Пи попытке анализа и модификации текста программы программист вынужден был оперировать искусственными категориями. Чтобы писать все более сложные программы, необходим был новый подход к программированию. В итоге были разработаны принципы объектно-ориентированного программированию. В итоге были разработаны принципы объектно-ориентированного программирования. ООР аккумулирует лучшие идеи воплощенные в структурному программировании, и сочетает их с мощными новыми концепциями, которые позволяет по-новому организовывать ваши программы.

1.2. МЕТОДОЛОГИЯ ПРОГРАММИРОВАНИЯ

Приведем основные определения

Программа–завершенный продукт, пригодный для запуска своим автором на системе, на котором он был разработан.

Программный продукт – программа которую любой человек может запускать тестировать исправлять и развивать. Такая программа должна быть написана в обобщенном стиле тщательна оттестирована и сопровождена подробной документацией. (С учетом модной в настоящее время концепции авторских прав, здесь необходима уточнить – любой человек, имеющий разрешение работать с исходными текстами программ)

Программный комплекс - набор взаимодействующих программ, согласованных по функциям и форматам, точно определенным интерфейсом, и вкупе составляющих полное средство для решения больших задач.

Жизненный цикл программного обеспечения – это весь период его разработки и эксплуатации, начиная с момента возникновения замысла и заканчивая прекращениям его использования.

Методология программирования – совокупность методов, применимых в жизненном цикле программного обеспечения и объединенных общим философским подходам.

Существует четыре широко известных в настоящее время методологии программирования – императивного, объектно-ориентированного, логического функционального.

1.3. ТЕХНОЛОГИЯ ПРОГРАММИРОВАНИЯ

Программирования изучает технологические процессы и порядок их прохождения – стадии (с использованием знаний, методов и средств).

Процесс – совокупность взаимосвязанных действий преобразующих некоторые входные данные в выходные. Процессы состоят из набора действий, а каждое действие из набора задач. Вертикальное измерения отражает статические аспекты процессов и оперирует такими понятиями, как рабочие процессы, действия, задачи, результаты деятельности и исполнители.

Стадия – часть действий по создании программного обеспечения ограниченная некаторами временными рамками и заканчивающаяся выпуском конкретного продукта, определяемого заданными для данной стадии требованиями. Стадии состоят из этапов, которые обычно имеют итерационный характер. Иногда стадии объединяют в боле крупные временные рамки, отражает динамические аспекты процессов и оперирует такими понятиями, как фазы, стадии, этапы, итерации и контрольные точки.

Технологический подход – определяется спецификой комбинации стадии и процессов, ориентированной на разные классы программного обеспечения и на особенности коллектива разработчиков.

Глава 2: Объектно-ориентированное язык программирования C++

2.1 АЛФАВИТ И ИДЕНТИФИКАТОРЫ

При написании программ на языке C++ используются символы, составляющие его ***алфавит.***Набор символов зависит от среды выполнения. На ПЭВМ широко используется символьный набор ISO 646-1983, называемый кодом ASCII (American Standart Code for Information Interchange - американский стандартный код обмена информацией). Он содержит *латинские буквы***,** *арабские цифры, специальные* и *управляющие* символы, которые в своем большинстве входят в состав *алфавита языка* C++. Каждый символ кодируется семибитным значением. Для представления *кириллических символов* используется восьмибитный *расширенный*ASCii-код, в котором единичное значение старшего бита говорит об использовании дополнительного символьного набора.

Алфавит C++ включает л*ашинские* прописные и строчные *буквы*: A,...,Z, a,..., z, *арабские цифры:* **0**,1,..., 9, *специальные символы***:**

+ - * / < > = | & ! \ ~ ' @ # $ % ^ ? _ : ; , . ( ) [ ] { } " .

В качестве *символа-разделителя*элементов (слов) предложений языка используется *пробел*, который на экране не отображается, а для наглядности при записи на бумаге часто обозначается символом Предложения (операторы) языка обычно заканчиваются точкой с запятой. Исключение составляют директивы препроцессора, начинающиеся с символа #, составные операторы и блоки определения функций, которые обрамлены фигурными скобками — {}.

Кроме того имеются *управляющие символы,* которые непосредственно на экране не отображаются. Для их записи используются специальные приемы, которые будут рассмотрены позже. В качестве примера записи управляющего символа «*горизонталь-пая табуляция»*приведем ' \t'. Еще один пример записи управляющего символа ««о-*вая строка**был рассмотрен в первой программе на языке С++.

Использование кириллических символов некоторых случаях возможно и целесообразно (в комментариях, символьных строках, названиях файлов, если это допускает среда выполнения). По пока единого стандарта на кодировку кириллических символов нет. Поэтому могут быть сложности при переносе таких программ с одного компьютера на другой, при переходе из одной среды выполнения в другую.

Специальные символы используются для обозначения (именования) *опкций* и записи *выражений.* Например, запись *((а+ Ь) г)*является *выражением,* задающим вычисление суммы значений переменных ***anbe*** последующим умножением на значение перемешюйе**.**

Совокупность двух и трех специальных символов может задавать *имя (знак) операции.*

Например, ++ и -- являются знаками *унарных* операций *инкремента* (увеличения значения *операнда* на 1) и *декремента* (уменьшения значения *операнда* на 1) соответственно. Так, оператор инкремента переменной *time* имеет вид

time++;

Имена имеют многие элементы программы: константы, переменные, типы данных, функции и ряд других. Такие имена являются *идентификаторами.* Имена вводятся для того, чтобы отличать (идентифицировать) различные элементы одного вида (типа) от других и оперировать (производить действия) с ними. *Идентификатором* называется последовательность символов из латинских букв, символа подчеркивания и арабских цифр, которая начинается с буквы и служит для именования различных элементов программы. Примеры идентификаторов: *varl, TaBle7, badcall, Jimit.*

Идентификаторы могут включать любое число символов, из которых значимыми являются первые 32, т. е. длинные идентификаторы считаются различными, если у них отличаются последовательности из первых 32 символов.

Строчные и заглавные буквы суть разные символы. Поэтому идентификатор *Radius*отличается от идентификатора *radius.*

Некоторые идентификаторы языка зарезервированы в служебных целях и их нельзя использоваться именования переменных, констант и функций. Такие идентификаторы называют *служебными* или *ключевыми (keyword)* словами и входят в алфавит языка. Используемые в стандарте C++ ключевые слова приведены в приложении.

При подключении *стандартных библиотек* добавляется ряд специальных идентификаторов, таких как *cerr, cin, clog, complex, cout, list, map, set, sizet, string, valarray, vector.* Их также не рекомендуется использовать в качестве идентификаторов.

Рекомендации по *именованию:*

• исполкчоиать имена из постановки задачи;

• давать короткие осмысленные имена, отражающие назначение переменной, функции, объекта или типа;

• не начинать с символа подчеркивания, поскольку такой прием широко используется в библиотеках системы программирования;

• следовать единой системе именования; здесь существуют различные варианты, например:

— начинать с прописной буквы, если требуется подчеркнуть уникальность идентификатора;

— использовать символ подчеркивания или прописные буквы внутри идентификатора для построения хорошо читаемых сложных идентификаторов.

Обычно редко удается в именах элементов программы прокомментировать ее содержание. Поэтому для пояснения отдельных частей или всей программы используют *комментарии.* Для введения однострочного комментария используют пару символов //, после которых следует поясняющий текст до конца строки. Многострочные комментарии начинаются с пары символов /* и заканчиваются парой символов */•

Рекомендации по *комментированию:*

• начинать программу с кратких комментариев, описывающих основные этапы алгоритма, переменные для хранения исходных данных, промежуточных и выводимых результатов;

• писать комментарии в терминах постановки задачи и выбранного метода решения;

2.2 МАССИВЫ

Массивы тоже относятся к категории составных типов, поскольку они позволяют сгруппировать несколько переменных, расположенных последовательно друг за другом, под одним идентификатором. Например, следующая запись выделяет память для 10 последовательных переменных int, но без присваивания уникальных идентификаторов:

int a[10]:

Вместо этого все переменные группируются под общим именем a.

При обращении к *элементу массива* используется такая же форма записи с квадратными скобками, как и при определении массива:

a[5] = 47:

Хотя *размер* массива a равен 10, индексирование (нумерация элементов) начинается с нуля, поэтому допустимые индексы элементов находятся в интервале 0-9:

```
//: C03.Arrays.cpp
#include <iostream>
using namespace std:
int ma1n() {
int a[10]:
for(int1 * 0 :  i < 10: i++) {
a[i] = i * 10:
cout « "a[“ « 1 « “] » " « a[1] « endl:
}
} ///:-
```

Обращения к массивам выполняются чрезвычайно быстро. Тем не менее страховка на случай нарушения границ массива не предусмотрена - программа начнет портить содержимое других переменных. Другой недостаток заключается в том, что размер массива должен определяться на стадии компиляции; если вдруг потребуется изменить размер массива во время выполнения программы, то сделать это в приведенном выше синтаксисе не удастся (вообще говоря, в C предусмотрен способ создания динамических массивов, но он громоздок и неудобен). Класс C++ vector, представленный в предыдущей главе, реализует объектный аналог массива с автоматическим изменением размеров. Если размер массива не известен на стадии компиляции, это решение обычно гораздо удобнее.

Элементы массивов могут относиться к произвольному типу, даже к структурному:

```
//: C03:StructArray.cpp
// Массив структур
typedefstruct {
int i. j. k:
```

```
} ThreeDpoint:
intmainO {
ThreeDpointp[10]:
For(int I = 0: i <10: i++) {
p[i].i= i + 1:
 p[i]j = i + 2:
p[i]k = i + 3:
}
} ///:-
```

Обратите внимание: идентификатор поля структуры i никак не связан с односменным счетчиком цикла for.

Чтобы убедиться в том, что элементы массива действительно хранятся в смежных областях памяти, можно вывести их адреса:

```
//: C03:ArrayAddresses.cpp
#include <iostream>
using namespace std:
int main( ) {
int a[10]:
cout « " sizeof (int) =  "« sizeof (int) « endl:
for(int i = 0: i < 10: i++)
cout « ”&a[” « i « "] = "
```

2.3 ФУНКЦИЯ

Функции - используются для упрощения процесса разработки программ в случаях, когда аналогичные преобразования над различными данными необходимо выполнять в нескольких местах программы.

Каждая программа в своем составе должна иметь ***главную функцию main().*** Именно функция *main()* обеспечивает создание точки входа в объектный модуль.

Кроме функции $main()_t$ в программу может входить произвольное число функций, выполнение которых инициализируется либо прямо, либо

опосредованно вызовами из функции *main()*. Каждая функция по отношению к другой является внешней. Для того, чтобы функция была доступной, необходимо, чтобы до ее вызова о ней было известно компилятору.

С понятием функции в языке С++ связано три следующих компонента:

• описание функции;

• прототип;

• вызов функции.

Описание функции состоит из двух частей: заголовка и тела. Описание функции имеет следующую форму записи:

```
/* заголовок функции*/
[тип_результата] <имя>([список_параметров])
{
/* объявления и операторы */
тело_функции
}
```

Здесь *тип результата* — тип возвращаемого значения. В случае отсутствия спецификатора типа предполагается, что функция возвращает целое значение *(int).* Если функция не возвращает никакого значения, то на месте типа записывается спецификатор *void.* В списке параметров для каждого параметра должен быть указан тип. При отсутствии параметров список может быть пустым или иметь спецификатор *void.*

Тело функции представляет собой последовательность объявлений и операторов, описывающих определенный алгоритм. Важным оператором тела функции является оператор возврата в точку вызова: *return [выражение];.*Оператор *return* имеет двойное назначение. Он обеспечивает немедленный возврат в вызывающую функцию и может использоваться для передачи вычисленного значения функции. В теле функции может быть несколько операторов ***return,*** но может не быть и ни одного. В последнем случае возврат в вызывающую программу происходит после выполнения последнего оператора тела функции.

Прототип функции может указываться до вызова функции вместо описания функции для того, чтобы компилятор мог выполнить проверку соответствия типов аргументов и параметров. Прототип функции по форме такой же, как и заголовок функции, в конце его ставится <;>. Параметры функции в прототипе могут иметь имена, но компилятору они не нужны.

Компилятор использует прототип функции для сравнения типов аргументов с типами параметров. Язык С++ не предусматривает автоматического преобразования типов в случаях, когда аргументы не совпадают по типам с соответствующими им параметрами, т. е. язык С++ обеспечивает строгий контроль типов.

При наличии прототипа вызываемые функции не обязаны размещаться в одном файле с вызывающей функцией.

Вызов функции может быть оформлен в виде оператора, если у функции *отсутствует* возвращаемое значение, или в виде выражения, если *существует* возвращаемое значение.

В первом случае оператор имеет следующий формат:

имя_функции (список_аргументов);

Во втором случае выражение записывается следующим образом: имя_функции (список_аргументов)

Описание функции *max* находится в файле *max.cpp,* находящемся в корневом каталоге диска $d:_t$ и имеет следующий вид:

```
int max (int a, int b)
{
int c;
/*рабочая переменная */
if (a>=b) c=a; elsec=b;
returnc;
}
```

2.4. КЛАССЫ

Класс **-** представляет собой абстрактный тип (определяемый программистом), который создается на основе существующих типов. Отдельный класс включт. е.т. е.бядт. е.е, называемые *элементами данных,* и функции, называемые *методами.* Элементы данных и методы являются равноправными компонентами класса.

Описание класса имеет следующий формат:

class | struct | unionимя_класса {список__компонентов};

В этом описании:

• одно из ключевых слов *class, struct* или *union* указывает на начало описания класса, определяет используемый по умолчанию статус доступа к компонентам класса, а также влияет на возможности наследования свойств этого класса;

• *имя_класса* — идентификатор;

• *список_компонентов* — перечень объявлений элементов данных и описаний методов класса.

В соответствии с синтаксисом языка C++ каждый компонент класса обладает статусом доступа. Таких статуса три: общедоступный, собственный и защищенный. В качестве *спецификаторов доступа* используются ключевые слова *public* (общедоступный*), private* (собственный), *protected* (защищенный), за которыми следует двоеточие. Действие спецификатора на компоненты класса начинается с момента его написания до нового спецификатора или до конца описания класса.

Спецификатор доступа *private* используется в основном для задания статуса доступа к элементам данных класса, что позволяет решить проблему защиты данных. Собственные данные являются доступными только для методов своего класса. Спецификатор доступа *public* часто используется для задания общедоступного доступа методам класса, которые организуют связь объекта данного класса с внешним миром. Статус защищенный *(protected)* используется в классах при применении механизма наследования классов.

При отсутствии наследования спецификатор *protected* эквивалентен спецификатору *private*.

Все компоненты класса, введенные с помощью ключевых слов ***struct*** и ***union,*** являются по умолчанию *общедоступными,* а с помощью ключевого слова ***class*** — *собственными,* т. е. недоступными для обращений извне. Для изменения статуса компонентов классов, описанных с помощью ключевых слов *class* и *struct,* необходимо использовать спецификаторы доступа. Классы, описанные с помощью ключевого слова *union,* не могут использоваться в качестве базовых классов при наследовании. Кроме того, у объектов, объявленных на основе подобного класса, для элементов данных выделяется общее место в памяти. Статус компонентов у таких классов изменить нельзя.

Операторы ветвления и циклы

1. *Логические значения, выражения и функции*

В этой лекции подробно рассматриваются операторы ветвлешя ("if " и "switch") и операторы циклов "for" и "while". Для применения всех этих операторов необходимо хорошо знать, что такое логические выражения и как они вичеслаются.

Язык Си++ унаследовал от языка Си соглашение, согласно которому целоезначение 0 считается логическим "false" (ложное значение), а ненулевое целое -логическим "true" (истинным значением). Но выражения вроде;

условие1 = 1

или

условие2 == 0

не слишком удобны при чтении теста программ человеком. Было бы лучше записывать логические выражения в интуитивно понятном виде:

условие 1 ==true

и

условие2 == false

Поэтому в Си++ был добавлен специальный логический тип "bool". Переменные типа "bool" могут принимать значения "true" и "false", которые при необходимости автоматически преобразуются в выражениях в значения 1 и 0.

Тип данных "bool" можно использовать в программах точно так же, как и типы"int", "char" и др. (например, для описания переменных или для создания функций, возвращающих значения типа "bool").

Программа 1.1 приведена в качестве примера использования типа даных"bool". Она запрашивает с клавиатуры возраст кандидата, сдававшего некий тест, и полученную кандидатом оценку в баллах. Затем программа оценивает результат выполнения теста по шкале, зависящей от возраста кандидата и делает вывод о том, сдантест или нет. Для кандидатов до 14 лет порог сдачи теста составляет 50 баллов, для15 лет или 16 лет - 55 баллов, старше 16-ти лет - 60 баллов.

```
#include <iostream.h>

bool acceptable( int age, int score );

int main()

{

intcandidate_age, candidate_score;
```

```
cout«"Введитевозрасткандидата: ";
cin » candidate_age;
cout«"Введитерезультаттестирования:";
cin » candidate_score;
if ( acceptable(candidateage, candidatescore ))
cout«"Этоткандидатсдалтестуспешно.\n";
else
cout«"Этот кандидат тест не прошел.\n ";
return 0; }
// Функция оценки результата тестирования тест сдан/не сдан
bool acceptable( int age, int score)
if ( age <= 14 && score >= 50 )
return true;
else if (age <= 16 && score >= 55 )
return true;
else if (score >= 60)
return true;
else
return false; }
```

2.Циклы "for", "while" и "do...wMle"

Циклы "for" впервые встречались во 2-й лекции, цикл "while" упоминался в 4-й лекции. Любой цикл "for" можно переписать в виде цикла

"while" и наоборот. Рассмотрим программу 2.1 (она похожа на программу 2.2 из 2-й лекции.

```
# include<iostream.h>

int main()

{

int number;

char character;

for ( number = 32; number <= 126; number = number + 1 )

character = number,

cout« "Символ '" « character;

cout« " имееткод" « number << "\n";

return 0;

}
```

Можно переписать с помощью цикла "while" (программа 2.2):

```
#include <iostream h>

#include <iostream.h>

int main();

int number;

char character,

number = 32;

while ( number <= 126 )

{
```

```
character = number,
cout« "Символ '" « character;
cout« "   имееткод " « number « "\n"
number++;
}
return 0;
}
```

Замена цикла "while" на цикл "for" выполняется совсем просто. Например, в программе 2.2 строку

```
while (number<= 126)
```

можно заменить эквивалентным циклом "for" без операторов инициализации и изменения значений:

```
for (; number<= 126;)
```

В Си++ есть еще один, третий, вариант цикла - оператор цикла с постфиксным условием (постусловием) *"do {...} while"*. Он отличается от циклов "for" и "while" тем, что тело цикла внутри скобок "о" обязательно выполняется как минимум один раз, т.к. условие повторения цикла проверяется только после выполнения тела цикла. Циклы "Do ... while" используются довольно редко. Например этот цикл можно применить для проверки корректности введенных с клавиатуры данных (программа 2.2а).

```
…
…
do {
```

```
cout«"Введите результат   тестирования: ",

cin » candidatescore;

  if ( candidatescore > 100 || candidatescore < 0 )

cout«"Допускается  оценка    от 0  до 100.\n";

} while( candidatescore> 100 || candidate_score< 0 );

…

…
```

Фрагмент программы 2.2a.

В программе 2.2a цикл с постусловием позволяет избавиться от дублирования операторов печати приглашения и ввода данных, которое возникает при использовании эквивалентного цикла"while" (программа 2.2b).

```
…

…

cout«"Введите  результат    тестирования:";

cin » candidate_score;

while ( candidatescore > 100 || candidate_score < 0 )

}

cout«"Допускаетсяоценкаот 0до100.\n",

cout«"Введите   результат    тестирования";

cin » candidate _score;

}
```

Фрагмент программы 2.2b.

3. Множественное ветвление и оператор "switch"

Вложенные операторы "if, предназначенные для выполнения "множественного ветвления", уже встречались в 1-й лекции. Упрощенная версия этого примера приведена ниже:

```
if (totalJestscore< 50 )
cout «"Вы не прошли тест. Выучите материал как следует. \n";
elseif (totaltestscore< 60 )
cout «"Вы прошли тест со средним результатом.\n";
elseif (totaltestscore< 80 )
cout «"Выхорошовыполнилитест.\n";
elseif (totaltestscore< 100 )
cout«"Вы показали отличный результат.\n";
else
{
cout«"Вы выполнили тест нечестно";
cout«"(оценка должна быть меньше 100 баллов)!\n";
```

Вложенные операторы "if выглядят слишком громоздко, поэтому в Си++ реализован еще один способ множественного ветвления - оператор "switch" Он позволяет выбрать для выполнения один из нескольких операторов, в зависимости от текущего значения определенной переменной или выражения.

В приведенном выше примере сообщение для печати выбирается в зависимости от значения переменной "totaijest_score". Это может быть произвольное целое число в диапазоне от 0 до 100. Диапазон проверяемых

значений можно сузить, если учесть, что оценка теста проверяется с точностью до 10-ти баллов. Введем дополнительную целочисленную переменную "score_outoften" и присвоим ей значение: scoreoutoften = total_test_score/10;

Теперь проверку в программе можно сформулировать так: (1) если переменная "score_out_of _ten"равна 0, 1, 2, 3 или 4, то печатать сообщение"Вы не прошли

тест. Выучите материал как следует.", (2) если "scoreouiot'_ten" равна 5, то печатать сообщение "Вы прошли тест со средним результатом."и т.д.

В целом оператор "switch" будет выглядеть так:

```
score_out_of_ten = totaltestscore /10;

switch ( score_out_of_ten )

{

case 0:

case 1:

case 2:

case 3:

case 4:

cout«"Вы не прошли тест. Выучите материал как следует. \n";

break,

case 5

cout«"Вы прошли тест со средним результатом \n",
```

```
break;
case 6:
case 7:
cout«"Вы хорошо выполнили тест \n";
break,
case 8 :
case 9 :
case 10:
cout«"Вы показали отличный результат \n",
break;
default:
cout«"Вы выполнили тест нечестно \n",
cout«"(оценка не должна быть больше 100 баллов)!\n";
}
```

Фрагмент программы 3.1.

Оператор "switch" имеет следующий синтаксис:

switch*(селектор)*

{

case*метка*1

< *операторы*>

break;

case*метка*N *:*

*<операторы*N>

break;

default:

<операторы>

}

Сделаем несколько важных замечаний относительно оператора "switch":

- Внутри "switch" выполняются операторы, содержащиеся между меткой, совпадающей с текущим значением селектора, и первым встретившимся после этой метки оператором "break".

- Операторы "break" необязательны, но они улучшают читабельность программ. С ними сразу видно, где заканчивается каждый вариант множественного ветвления. Как только при выполнении операторов внутри "switch" встречается "break", то сразу выполняется переход на первый оператор программы, расположенный после оператора "switch". Иначе продолжается последовательное выполнение операторов внутри "switch".

- Селектор (переменная или выражение) может быть целочисленного (например, "int" или "char") или любого перечислимого типа, но не вещественного типа.

- Вариант "default" ("по умолчанию") необязателен, но для безопасности лучше его предусмотреть.

4. Блоки и область видимости переменных

В Си ++ фигурные скобки "{}" позволяют оформить составной оператор, который содержит несколько операторов, но во всех конструкциях языка может подставляться как один оператор. На описания переменных фигурные скобки также оказывают важное влияние.

Составной оператор, внутри которого описана одна или несколько переменных, называется *блоком. Для* переменных, объявленных внутри блока, этот блок является *областью видимости.* Другими словами, переменные "создаются" каждый раз, когда при выполнении программа входит внутрь блока, и "уничтожаются" после выхода из блока.

Если одно и то же имя используется для переменной внутри и снаружи блока, то это две разных, независимых переменных. При выполнении внутри блока программа по умолчанию полагает, что имя относится к внутренней переменной. Обращение к внешней переменной происходит только в том случае, если переменная с таким именем не описана внутри блока. Действие этого правила продемонстрировано в программе 4.1.

```
#include<iostream.h>

intintegerl = 1;

int integer2 = 2;

 int integer3 = 3,

int main()

{

int integerl =-1;

 int integer2 = -2;

int integerl = 10;

cout« "integerl == " « integerl « "\n";
```

```
cout« "integer2 == " « integer2 « "\n";

cout« "integer3 == " « integer3 « "\n";

}

cout« "integerl == " « integerl « "\n";

cout« "integer2 == " « integer2 « "\n";

cout« "integer3 == " « integer3 « "\n";

return 0;

}
```

Программа 4.1.

Программа 4.1 вывод Инта экран сообщения:

```
integerl == 10

 integer2 == -2

integer3 == 3

integerl ==-1

 integer2 == -2

integer3 — 3
```

Применение локальных переменных иногда объясняется экономией памяти, а иногда необходимостью использования в различных частях программы разных переменных с одинаковыми именами. См. в качестве примера программу 4.2, которая печатает таблицу умножения для чисел от 1 до 10.

```
#include <iostream.h>

int main()
```

```
{
int number;
for ( number = 1; number <= 10; number++ )
{
int multiplier,
for ( multiplier = 1; multiplier <= 10; multiplier++ )
{
cout« number« " x " « multiplier « " = ";
cout« number * multiplier« "\n";    }
cout«"\n";  }
return 0,
}
```

Программа 4.2.

Программу 4.2 можно переписать в более понятном виде спомощью функции (см. программу 4.3).

```
#include <iostream.h>
voidprint_times_table( int value, int lower, int upper);
int main()
{
int number;
for ( number = 1; number <= 10; number++ )
{
```

```
print_times_table( number, 1, 10);

cout«"\n";

}

return 0;

}

voidprint_times_table( int value, int lower, int upper)

{

int multiplier;

for ( multiplier = lower; multiplier <= upper; multiplier++ )

{

cout« value « " x " « multiplier « " = ";

cout« value * multiplier « "\n";    }   }
```

Программа 4.3.

Далее, программу 4.3 можно усовершенствовать, исключивописания всех переменныхиз "main()" идобавивдве функции (см. программу 4.4).

```
#include <iostream.h>

voidprint_tables( int smallest, int largest);

voidprint_times_table( int value, int lower, int upper);

int main()

{

print tables( 1, 10 );

return 0; }
```

```
voidprintjables( int smallest, int largest)
{
int number,
for ( number = smallest; number <= largest; number++ )
{
print_times_table( number, 1, 10);
cout«"\n";
   }
}
voidprint_times_table( int value, int lower, int upper)
{
int multiplier;
for ( multiplier = lower; multiplier <= upper; multiplier++ )
{
cout « value « " x " « multiplier« " = ";
cout« value * multiplier « "\n";
```

Программа 4.4.

5. Замечания вложенных циклов

В первоначальном варианте программы "таблица умножения" (программа 4.2) есть вложенные циклы. В последующих вариантах программы читабельность исходного текста улучшается с помощью

процедурной абстракции. Преобразование тела цикла в вызов функции позволяет производить разработку Лого алгоритма независимо от остальной части программы. Поэтому уменьшается вероятность ошибок, связанных с областью видимости переменных и перегрузкой имен переменных.

Недостаток выноса тела цикла в отдельную функцию заключается в уменьшении быстродействия, поскольку на вызов функции тратится больше времени, чем на итерацию цикла. Пели цикл выполняется не очень часто и не содержит большого количества итераций (больше нескольких десятков), то временными затратами на вызов функции вполне можно пренебречь.

6. *Сводка результатов*

Тип данных "bool" предназначен для использования в логических выражениях и в качестве возвращаемого значения логических функций. Такие функции можно применять в качестве условий в условных операторах и операторах циклов. В Си++ есть три варианта циклов: "for", "while" и "do ... while".

Вложенные операторы "if" в некоторых случаях можно заменить оператором множественного ветвления "switch".

Внутри составного оператора (блока), ограниченного фигурными скобками "{}", допускается описание локальных переменных (внутренних переменных блока**).**

7. *Упражнения*

Упражнение 1

Разработайте функцию, которая принимает целочисленный параметр и возвращает логическое ("bool") значение "true", только если переданное ей число является простым числом из диапазона от 1 до 1000 (число 1 простым

не считается). Проверьте свою функцию на различных входных данных с помощью тестовой программы.

Подсказка: *(I) если число не является простым, то оно имеет как минимум один простой множитель, меньший или равный квадратному корню из числа. (2) (32*32) = 1024 и 1024>1000.*

Упражнение 2

Напишите функцию "prim_pyramid(...)", которая получает целочисленный параметр "height (высота)" и отображает на экране "пирамиду" заданной высоты из символов "*". Проверьте функцию с помощью простой тестовой программы, которая должна воспроизводить следующий примерный диалог с пользователем:

Эта программа печатает на экране "пирамиду"заданной высоты.

Введите высоту пирамиды:37 Введите другое значение (из диапазона от 1 до 30). 6

**

****

******

****

****

****

Упражнение 3

Цикл "for" всегда можно переписать в форме цикла "while", и наоборот. Являются ли две показанных ниже программы эквивалентными? Какие сообщения они печатают на экране? Объясните свой ответ и проверьте его опытным путем.

Программа 3a:

```
#include <iostream.h>
int main()
{
int count = 1;
for (; count <= 5; count++ )
{
int count = 1;
cout« count« "\n";
}
return 0;
}
```

Программа 3b:

```
#include <iostream.h>
 int main()
 {
   int count = 1;
   while ( count <= 5 )
{
   int count = I;
 cout « count « "\n";
  count++;
    }
return 0;
```

Упражнение 4

Приведенная ниже программа должна печатать время закрытия магазина в различные дни недели (ввиду таблицы). В программе объявлен новый перечислимый тип данных "День" и определена функция "ciosmg_iime(..)", которая должна возвращать час закрытия магазина в заданный день (пока эта функция не слишком сложна - для любого дня возвращает значение 17). Программа демонстрирует, как можно использовать типы "int" и "Day" в преобразованиях типов (в заголовке цикла "for" и при ВЫЗОВ функции "closingJime(...)").

```
#include <iostream.h>
enum Day { Monday, Tuesday, Wednesday, Thursday,
```

```
Friday, Saturday, Sunday };

intclosing_time( Day day_of_the_week );

//    Главнаяфункция

intmain()

{

intcount;

//   Печать  заголовка   таблицы

cout.width(17);

cout«"ДЕНЬ";

cout.width(19);

cout« "ВРЕМЯ  ЗАКРЫТИЯ \n\n";

//    Печать   таблицы   от   понедельника (Monday)   до

//    воскресенья (Sunday)

for ( count = int(Monday), count <= int(Sunday), count++ )

{

cout.width(19);

switch ( count)

{

case 0 : cout« "Понедельник"; break;

case 1 : cout «"Вторник"; break;

case 2 : cout « "Среда"; break;
```

```
case 3 : cout« "Четверг"; break;

case 4 ; cout « "Пятница"; break;

case 5 : cout « "Суббота"; break;

case 6 . cout« "Воскресенье", break;

default: cout « "ОШИБКА!";

}

cout.width(9);

cout« closing_time( Day(count))« ":00\n",

}

return 0;

}

//  Конец  главной  функции

//  Функция,  возвращающая  время  закрытия  магазина

//  взаданныйденьнедели

intclosing_time( Day day_of_the_ week )

{

return 17,  }
```

(а) Что произойдет (и почему), если заменить оператор "switch" на строку

```
cout« Day(count); ?
```

Вместо этого замените "switch" на строку prim_day(Day(count), cout); и добавьте описание и определение функции "prim_day(...)", внутри которой

разместите удаленный из главной функции оператор "switch" (поток стандартного вывода "cout" передавайте в функцию как параметр по ссылке типа "ostream&").

(б) Магазин закрывается в воскресенье в 13:00, в субботу в 17:00, в среду в 20:00 и в остальные дни в 18:00. С помощью оператора "switch" внесите соответствующие изменения в функцию "closing_time(...)" и проверьте работу программы.

Упражнение 5

Напишите программу, которая отображает в виде таблицы количество строчных английских букв (от v до y) в собственном исходном файле "ex5_5.cpp" (сохраните исходный текст программы именно в этом файле).

При разработке программы предположите, что у компьютера очень мало памяти - используйте только одну переменную типа "ifstream",одну переменную типа "char" и две переменных типа "int". Программа должна выдавать на экран сообщения, похожие на следующие:

СИМВОЛ КОЛИЧЕСТВО ВХОЖДЕНИЙ

A 38

B 5

C 35

D 7

E 58

f 8

…

…

W 4

X 4

Y 0

Z 1

Массивы и символьные строки

1. *Назначение массивов*

В программировании часто возникают задачи, связанные с обработкой больших объемов данных. Для постоянного хранения этих данных удобно пользоваться файлами. Например, в программе для ввода и сортировки длинных числовых списков данные можно ввести с клавиатуры один раз и сохранить в файле для последующего многократного использования. Но до сих пор не было рассмотрено удобного способа представления больших объемов данных внутри программ. Для этой цели в Си++ часто применяются *массивы* - простейшая разновидность *структурных типов данных* (о более сложных структурах данных будет говориться в следующих лекциях).

Массив - это набор переменных одного типа ("im", "char" и др.). При объявлении массива компилятор выделяет для него *последовательность* ячеек памяти, для обращения к которым в программе применяется одно и то же имя. В то же время массив позволяет получить прямой доступ к своим отдельным элементам.

1.1 Объявление массивов

Оператор описания массива имеет следующий синтаксис:

<тип данных><имя переменной>[<целое значение:»];

Допустим, в программе требуется обрабатывать данные о количестве часов, отработанных в течении недели группой из 6-ти сотрудников. Для хранения этих данных можно объявить массив:

inthours[6];

или, лучше, задать численность группы с помощью специальной константы:

constint NO_OF_EMPLOYEES = 6;

inl hours[NO_OF_EMPLOYEES];

Если подобные массивы будут часто встречаться в программе, то целесообразно определить новый тип:

constint NO_OF^EMPLOYEES = 6;

typedefintHours_array[NO_OF_EMPLOYEES];

Hours array hours;

Hours array hoursweekjwo;

В любом изгрех перечисленных вариантов, в программе будет объявлен массив из 6 элементов типа "int", к которым можно обращаться с помощью имен: hours[0] hours[l] hours[2] hours[3] hours[4] hours[5]

Каждое из этих имен является именем *элемента* массива. Числа о,.... 5 называются *индексами* элементов. Отличительная особенность массива заключается в том что его элементы - однотипные переменные - занимают в памяти компьютера последовательные ячейки памяти.

Функция опять же возвращает объект Point, хотя вы могли бы заставить се возвращать любой тип значения по вашему выбору.

В качестве альтернативною примера вы можете создать функцию оператора, которая рассчитывает расстояние между двумя точками и

возвращает результат в формате с плавающей точкой (double). Для этого примера я выбрал оператор %. но вы можете выбрать любой другой бинарный оператор, предусмотренный в C++ (смотрите Приложение Л). Здесь важно то что вы можете выбрать любой тип возвращаемого значения, соответствующий операции, которую вы выполняете.

```
#include <math.h>

double Point::operator%(Point pt) { int dl = pt.x - x; int d2 - pt.y - y;

returnsqrt((double) (dl * dl + d2 * a2));

}
```

При таком определении функции следующий код корректно выведет расстояние между точками (20. 20) и(24. 23) рапное5.0.

```
Point ptl<20,20) ;

Point pt2(24 23);

cout « -Distance between points is. : " « ptl%pt2;
```

Функции операторов как глобальные функции

В предыдущем разделе я указывал, что вы можете объявлять функции операторов как глобальные функции. Однако есть недостаток такого объявления. В этом случае у вас не будет всех необходимых функций в объявлении класса. Но в некоторых случаях (я сейчас опишу их) использование такого подхода становится необходимым.

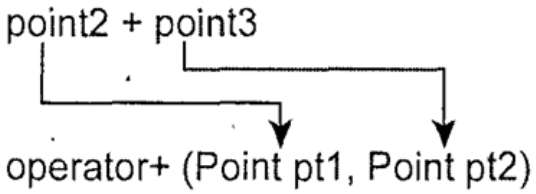
Глобальная функция оператора объявляется вне класса. Типы в списке аргументов определяют. какие типы операндов использует функция. Например, функция оператора сложения для класса Point может быть переписана как глобальная функция. Л вот объявление (прототип), которое должно появиться до вызова функции:

```
Point operator*(Point ptl, Point pt2) ;
```

Ниже приведено определение функции:

```
Pointoperator+{Pointptl.Pointpt2) {
Point new_pt; new_pt.x = ptl.x + pt2.x; new_pt.y = ptl.y + pt2.y;
returnnew_pt;
}
```

Вы можете представить себе вызов этой функции следующим образом:



Прочитав главу 12, вы узнали, как писать классы (типы объектов), которые работают как стандартные типы данных, не так ли?

Да, но только в некоторой степени. С наиболее важными функциями стандартных типов данных, таких как inc. float, double и даже char, вы уже можете производить определенные операции. По сути, без этих операторов было бы очень сложно осуществить какие-либо вычисления в языке С++.

С++ позволяет вам определить, как выполнять тс же самые операции (такие как +, * и /) с объектами вашего собственного класса. Вы также можете описать работу функции проверки на равенство, что позволит вам проверить, являются ли два числа равными.

Преимущество С++ состоит в том, что этот язык позволяет объявлять новые классы, которые почти для всех задач работают так же как основные типы данных.

Введение в функции операторов для класса

Основной синтаксис для записи функций оператора для класса прост, поэтому, когда вы овладеете им, вы сможете использовать столько операторов, сколько захотите.

*return_type**operator@(**argumenenlist)*

Применяя такой синтаксис, вы заменяете символ @ разрешенным оператором C++, например +, * и /. Кроме того, вы не ограничены использованием только этих четырех операторов, на самом деле вы можете использовать в этом случае любой символ оператора, поддерживаемый стандартными типами в C++. Обычные правила ассоциативности и приоритетов выполняются для этих символов соответствующим образом (смотрите Приложение A).

Вы можете определить функцию оператора либо как функцию-член, либо как глобальную функцию (то есть не функцию-член).

- Если вы объявляете функцию оператора как функцию-член, то объект, через который эта функция вызывается, соответствует левому операнду.
- Если вы объявляете функцию оператора как глобальную функцию, то оба операнда соответствуют аргументу.

Это значительно понятнее на примерах. Ниже приведен пример, в котором показано, как можно объявить функции операторов сложения и вычитания (+и -) как часть класса Point:

```
class Point {
//. . . public:
Point operator+(Point pt); Point operator-(Point pt);
};
```

Сейчас оба операнда интерпретируются как аргументы функции. Левый операнд (в этом случае **point2)** передаст свое значение первому аргументу **pt1.** Правый аргумент (в этом случае **point3)** передаст свое значение второму аргументу **pt2.** Концепция «этот объект» отсутствует, и все ссылки на объекты данных класса Point должны быть уточнены.

Это может вызвать проблему. Если объекты данных не объявлены открытыми, то эта функция не может получить к ним доступ. Решением

может быть использование вызовов фу нкции, если таковые имеются, для получения доступа к данным.

```
Point operator+(Point ptl, Point pt2) {
Point new_pt;
int a = ptl.get_x() + pt2.get_x();
int b = ptl.get_y() + pt2.get_y();
new_pt.set(a,b);
returnnew__pt;}
```

Но это не очень хорошее решение, кроме того, для некоторых классов этот вариант может не работать. Например, у вас может быть такой класс, в котором приватные члены данных полностью недоступны, а вы все равно хотите иметь возможность написания функции операторов. Лучшим решением может быть объявление функции как дружественной функции, что означает, что функция является глобальной, но у нее есть доступ к приватным членам класса.

В данном случае функция объявляется как дружественная функция для класса Point.

```
class Point {
//. . .
public:
friend Point operator+(Point ptl, Point pt2);  }
```

Теперь определение функции имеет непосредственный доступ ко всем членам класса Point, даже если они являются приватными.

```
Point operator+(Point ptl. Point pt2) {
  Point new_pt;
int a = ptl.x + pt2.x;
```

```
int b = pt1.y + pt2.y;

new_pt.set(a,b) ;

returnnew_pt;  }
```

Иногда необходимо задать функции операторов как глобальные функции. В функции- члене левый операнд интерпретируется как «этот объект» в определении функции. Л что если левый операнд не объектного типа? Что если вы хотите поддержать подобную операцию?

```
pointl= 3 * point2;
```

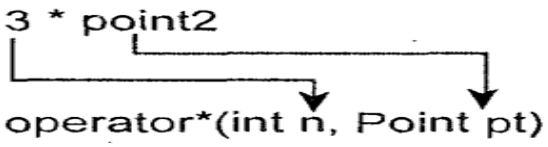
Проблема в данном случае заключается в том, что левый операнд имеет тип int. а не Point. Но вы не можете писать новые операции для типа int, как для класса. Единственным способом поддержать эту операцию является написание глобальной функции.

```
Pointoperator*(intn.Pointpt) {

Point new_pt;

new_pt.x = pt.x * n;

new_pt.y = pt.y * n;

returnnew__pt;

}
```

Как и раньше, для получения доступа к приватным членам данных вам, возможно, понадобится сделать функцию «другом» класса:

```
class Point {

//...

public:

friend Point operator*(int n. Point pt);

}
```

Вызов этой функции можно представить визуально следующим образом:



Повышение эффективности при помаши ссылок

Очевидным способом осуществления операций над объектами является использование простых объектных типов (классов) в качестве аргументов. Но как было указано в главе 12, каждый раз когда объект реализуется или возвращается в виде значения, осуществляется вызов копни конструктора.

Более того, всякий раз когда создается объект, программа должна запросить память системы для создания нового объекта. Все это происходит скрыто, но тем не менее влияет на эффективность программы.

Вы можете повысить эффективность своей программы, записывая классы таким образом, чтобы они минимизировали процесс создания объектов. Для этого есть простой способ: использовать ссылочные типы (referencetypes).

Ниже описана функция сложения для класса Point, а также функция оператора сложения (+), которая ее вызывает. Эта функция написана без использования ссылочных типов.

```
classPoint{
//...
public:
Pointadd(Pointpt);
Point operator*(Point pt);
};
Point Point::add(Point pt) {
```

Point new_pt;

new_pt.x = x + pt.x;

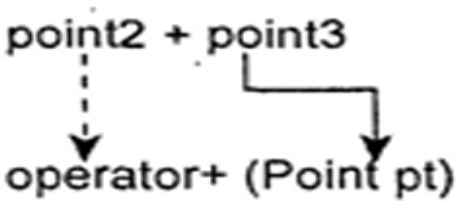
new_pt.y-y+ pt.y;

returnnew_pt;

Имея эти объявления, вы можете применять операции к объекту Point:

Point pointl, point2, point3; pointl = point2 + point3;

Компилятор интерпретирует это выражение, вызывая функцию operator+ через левый операнд - в этом случае **point2.** Правый операнд - в этом случае **point3** - становится аргументом этой функции. Визуально это отношение представлено на рисунке ниже:



Что происходит с операндом **point2?** Его значение игнорируется? Нет. Функция рассматривает **point2** как «этот объект», так что неквалифицированное использование координат **x** и приводит к обращению к копии координат x и y объекта **point2.**

Point Point::operator+{Point pt) {

Point-new_pt;

New_pt.x = x + pt.x;

new_pt.y = y+ pt.y;

returnnew_pt;

}

Неквалифицированное использование членов данных **x** и y обращается к значениям в левом операнде (в этом случае **point2).** Выражения **pt.x**и **pt.y**обращаются к значениям в правом операнде (в этом случае **polnt3).**

Функции оператора здесь объявляются с типом возвращаемого значения Point. Это означает, что они возвращают объект Point. И это правильно: если сложить две точки, то вы получите третью точку; а также если вы отнимаете одну точку от другой, то вы должны получить третью точку. Но C++ позволяет вам указывать любой действительный тип возвращаемого значения функции оператора.

Список аргументов также может содержать любой тип. Здесь возможна перегрузка операций: вы можете объявить функцию оператора, которая взаимодействует с типом int, другая функция будет взаимодействовать с типом doubleи так далее. В случае с классом Point, возможно, имеет смысл разрешить умножение на целое число. Объявление функции оператора будет выглядеть следующим образом:

```
Pointoperator*(intn);
```

Определение функции будет выглядеть следующим образом:

```
Point Point::operator*(int n) {
Point new_pt;
new_pt.x = x * n;
new_pt.y = y* n;
returnnew_pt;
}
}
Point Point::operator+(Point pt)
returnadd(pt);
}
```

Это очевидный способ написания этих функций, но обратите внимание на то, насколько выражение, такое как ptl+ pt2. приводит к созданию нового объекта.

- Правый операнд передастся функции operator+. Создастся копня pt2 и передается этой функции.

- Функция operator* вызывает функцию сложения add. Теперь должна быть создана и передана еще одна копия pt2.

- Функция сложения addсоздает новый объект - new_pt, который вызывает конструктор по умолчанию. Когда функция возвращает значение, программа создаст копию объекта new_.pti возвращает се вызывающему оператору (функция operator+).

- Функция operator+ возвращается вызывающему оператору, требуя создания еще одной копни объекта new_pt.

Так много копирования! Создастся пять новых объектов, что приводит к одному вызову конструктора по умолчанию и четырем вызовам конструкторов копирования. Это неэффективно.

Сегодня при наличии высокоскоростных процессоров вы можете возразить, что эффективность не является настолько критичным фактором. Если говорить о таком простом классе, как Point, то может понадобиться выполнение ты- (ЯН сеч повторяющихся операций (или даже миллионов!), чтобы почувствовать за- мытную временную задержку, если ваша программа работает не очень эффективно. Однако вы не можете быть полностью уверены в том, как именно дет использован ваш класс. Поэтому, если все же существует простой способ повышения эффективности вашего кода, вам следует им воспользоваться.

Вы сможете избежать использования двух из вышеуказанных операций копирования, используя ссылочные аргументы. Вот исправленная версия программы, измененные строки которой выделены полужирным шрифтом:

```
class Point {
//... public:
Point add(const Point &pt);
```

```
Point operator*(const Point &pt);
};
Point Point::add(const Point &pt) {
Point new_pt; new_pt.x = x + pt.x; new_pt.y = y+ pt.y; return new_pt;
}
Point Point::operator*(const Point &pt)
returnadd(pt);  }
```

Одно из преимуществ использования ссылочных типов, таких как Points, в том, что изменяется осуществление вызовов функции, но при этом не требуются другие изменения исходного кода. Помните, что когда вы передаете ссылку, то функция принимает ссылку на оригинальные данные, но без синтаксиса указателей.

Я также буду использовать тут ключевое слово const, которое не допускает изменений передаваемого аргумента. Когда функция получает свою собственную копию аргумента, то она не может изменить значение оригинальной копии, вне зависимости от того, какие операции выполняются. Но ссылочный аргумент, такой как указатель, потенциально может изменить оригинальную копию. Ключевое слово const восстанавливает защиту данных, так что функция не может случайно изменить значение аргумента.

Изменение устраняет две операции копирования объекта. Но каждый раз после возврата значений этими функциями создастся копия объекта. Вы можете сократить количество этих копий, сделав одну или обе эти функции встраиваемыми. Функция operator*, которая просто вызывает функцию сложения add. является хорошим претендентом на то, чтобы стать встраиваемой.

```
class Point {
//...
```

public:

Point add(const Point &pt);

Point operator+(const Point &ot) {return add(pt);}

};

Когда функция operator* встраивается таким способом, то операции, такие как

ptl+ pt2, транслируются непосредственно в вызовы функции сложения add.

Пример 13.1. Операторы класса Point

Теперь у вас есть все необходимые инструменты для написания эффективных и полезных функций операторов для класса Point. Нижеуказанный код показывает полное объявление класса Point, а также код, который тестирует его объявляя и выполняя операции над объектами.

Код, который остался неизменным из главы 12. написан обычным шрифтом, а новый и измененный код выделен полужирным шрифтом.

Листинг 13.1. Polnt3.cpp

#include <iostream> using namespace std; class Point {

private: *11* Data members (private)

int x, y;

public:// Constructors

Point() {}

Point(intnew_x, intnew_y) {set(new_x, new_y);} **Point(const Point &sre) {set(src.x, sre.y);}**

// Операции

Point add(const Point &pt);

Point sub(const Point &pt);

Point operator*(const Point &pt) {return add(pt);) Point operator-(const Point &pt) {return sub(pt);}

// Другим функции-члены

void set (intnew_x, intnew__y) ; intget._x() const {return x;} intget_y() const {return y;}

};

intmainO {

Point pointl(20, 20);

Point point2(0, 5);

Point point3{-10, 25);

Point point4 - pointl + point2 + point3; cout« "The point is " « point4.get_x(); cout<< ", << point4.get_y() <<<<endl; return 0;

}

void Point::set(intnew_x, intnew_y) { if (new_x< 0)

new_x *= -1; if (new_y< 0)

new_y *= -1;

X = new_x; y= new_y;

}

Point Point::add(const Point &pt) {

Point new_pt; newj?t.x = x + pt.x; new_pt.y» y+ pt.y; return new_pt;

}

Point Point::sub(const Point &pt) {

Point new_pt; new_pt.x * x - pt.x; new_pt.y = y- pt.y; return new_pt;

}

Как это работает

В этом примере к классу Pointдобавляется серия функций-членов:

Point add(const Point &pt);

Point sub(const Point &pt);

Point operator*(const Point &pt) {return add(pt);} Point operator-(const Point &pt) (return sub(pt);}

Функции addиsubвыполняют операции сложения и вычитания координат, таким образом, вы можете записать выражение следующего вида:

Pointpoint1 = point2.add(point3);

В этом выражении объекты **point2** и **point3** складываются для получения нового объекта класса Point. Функция operator* является встраиваемой функцией, которая транслирует такие выражения, как представлено ниже, в вызов функции сложения.

Pointpointl= point2 + point3;

При такой записи функция выполняется с минимальными вычислительными затратами, потому что данная функция является встраиваемой и используется ссылка на параметр (constPoint£c). Выражение point2 + point3 транслируется в вызов функции operator*, которая, в свою очередь, вызывает функцию add.

Функция add, в свою очередь, создаст новый объект класса point (new_pt), инициализируя его путем добавления координаты «этого объекта» к координатам объектного аргумента. «Этот объект» - это объект, через который происходит вызов функции. Другими словами, это объект **point2** в следующем выражении:

point2.add(point3);

Функции operator- и subработают по такому же принципу.

Также в этом примере к объявлениям функций **get_x**и **get_y**прибавляется ключевое слово **const**. Ключевое слово добавляется после

оставшейся части объявления, но перед открывающейся фигурной скобкой ({). В этом контексте ключевое слово **const**означает, что «функция не разрешает изменение каких-либо объектов данных».

intget_x() const {return x;} intget_y() const {return y;}

Данное изменение полезно по ряду причин. Так вы предотвращаете нежелательные изменения объектов данных, позволяете выполнять вызов функций через другие **const**функции, а также разрешаете вызов функций через функции, в которых разрешается не изменять объект Fraction(так как они имеют аргумент **const**объекта Fraction).

Упражнения

Упражнение 13.1,1. Напишите тестовую программу, чтобы увидеть сколько раз вызываются конструктор по умолчанию и конструктор копирования. (Подсказка: вставьте выражения, которые передадут вывод объекту cout; вы можете использовать несколько строк, если необходимо, при условии, что описания функции синтаксически правильны). Потом запустите программу в таком виде, а затем запустите се со ссылочными аргументами (constPoints), замененными на обычные аргументы (Point). Насколько эффективным является старый подход?

2.5 ПОЛИМОРФИЗМ

Дословно *полиморфизм* в переводе с греческого означает многообразие форм *(poly* — много, *morphos* — форм). *Полиморфизм* можно определить как свойство, позволяющее использовать одно имя для обозначения действий, общих для родственных классов. При этом конкретизация выполняемых действий осуществляется в зависимости от типа обрабатываемых данных. К важнейшим формам полиморфизма в C++ можно отнести следующее:

- перегрузку функций и операций;
- виртуальные функции;
- обобщенные функции, или шаблоны.

Перегрузку функций и операций можно определить как *анатический* (*static*) полиморфизм, поскольку он поддерживается на этапе компиляции. Виртуальные функции относятся к *динамическому (runtime*) полиморфизму, поскольку он реализуется при выполнении программ.

Достоинством полиморфизма является то, что позволяет использовать многократно один раз составленные алгоритмы и, как следствие, обеспечивает уменьшение избыточного кода.

Значение вычисленного выражения является возвращаемым значением функции. Возвращаемое значение передается в место вызова функции и является результатом ее работы.

Число и типы аргументов должны совпадать с числом и типом параметров функции. При вызове функции параметры подставляются вместо аргументов. Ниже приводится пример функции с возвращаемым значением.

Пример 1. Составить программу, содержащую обращение к функции вычисления максимума из двух чисел:

Возможное решение данной задачи имеет вид:

```
#include<iostream.h>
intmax(int,int);        /*прототип функции */
void main()
{
int x,y,z;
cout « "\ nпоочередно  введите x и y \ n";
cin » x; cin » y;
z=max(x,y);
cout « "z=" « z;
}
#include "d:\max.cpp" // включение  файла  max.cpp с функцией  max
 public:// Constructors
 Point() {}
```

Point(intnew_x, intnew_y) {set(new_x, new_y);} **Point(const Point &sre) {set(src.x, sre.y);}**

// Операции

Point add(const Point &pt);

Point sub(const Point &pt);

Point operator*(const Point &pt) {return add(pt);) Point operator-(const Point &pt) {return sub(pt);}

// Другим функции-члены

void set (intnew_x, intnew__y) ; intget._x() const {return x;} intget_y() const {return y;}

};

intmainO {

Point pointl(20, 20);

Point point2(0, 5);

Point point3{-10, 25);

Point point4 - pointl + point2 + point3; cout« "The point is " « point4.get_x(); cout<< ", << point4.get_y() <<<<endl; return 0;

}

void Point::set(intnew_x, intnew_y) { if (new_x< 0)

new_x *= -1; if (new_y< 0)

new_y *= -1;

X = new_x; y= new_y;

}

Point Point::add(const Point &pt) {

Point new_pt; new?t.x = x + pt.x; new_pt.y» y+ pt.y; return new_pt;

}

Point Point::sub(const Point &pt) {

Point new_pt; new_pt.x * x - pt.x; new_pt.y = y- pt.y; return new_pt;

}

Как это работает

В этом примере к классу Pointдобавляется серия функций-членов:

Point add(const Point &pt);

Point sub(const Point &pt);

Point operator*(const Point &pt) {return add(pt);} Point operator-(const Point &pt) (return sub(pt);}

Функции addи subвыполняют операции сложения и вычитания координат, таким образом, вы можете записать выражение следующего вида:

Pointpoint1 = point2.add(point3);

В этом выражении объекты **point2** и **point3** складываются для получения нового объекта класса Point. Функция operator* является встраиваемой функцией, которая транслирует такие выражения, как представлено ниже, в вызов функции сложения.

Pointpointl= point2 + point3;

При такой записи функция выполняется с минимальными вычислительными затратами, потому что данная функция является встраиваемой и используется ссылка на параметр (constPoint£c). Выражение point2 + point3 транслируется в вызов функции operator*, которая, в свою очередь, вызывает функцию add.

Функция add, в свою очередь, создаст новый объект класса point (new_pt), инициализируя его путем добавления координаты «этого объекта» к координатам объектного аргумента. «Этот объект» - это объект, через

который происходит вызов функции. Другими словами, это объект **point2** в следующем выражении:

point2.add(point3);

Функции operator- и subработают по такому же принципу.

Также в этом примере к объявлениям функций **get_x**и **get_y**прибавляется ключевое слово **const**. Ключевое слово добавляется после оставшейся части объявления, но перед открывающейся фигурной скобкой ({). В этом контексте ключевое слово **const**означает, что «функция не разрешает изменение каких-либо объектов данных».

intget_x() const {return x;} intget_y() const {return y;}

Данное изменение полезно по ряду причин. Так вы предотвращаете нежелательные изменения объектов данных, позволяете выполнять вызов функций через другие **const**функции, а также разрешаете вызов функций через функции, в которых разрешается не изменять объект Fraction(так как они имеют аргумент **const**объекта Fraction).

Упражнение 13.1.1. Напишите тестовую программу, чтобы увидеть сколько раз вызываются конструктор по умолчанию и конструктор копирования. (Подсказка: вставьте выражения, которые передадут вывод объекту cout; вы можете использовать несколько строк, если необходимо, при условии, что описания функции синтаксически правильны). Потом запустите программу в таком виде, а затем запустите се со ссылочными аргументами (constPoints), замененными на обычные аргументы (Point). Насколько эффективным является старый подход?

Упражнение 13.1.2. Напишите и протестируйте расширенный класс Point, который поддерживает умножение объекта Pointна целое число. Используйте глобальную функцию, поддерживаемую дружественными объявлениями **friend,** как это описано **в** предыдущем разделе.

Пример 13.2. Операторы класса Fraction

В этом примере используется техника, аналогичная технике примера 13.1. для расширения поддержки базовых операторов для класса Fraction. Как и раньше, в коде для эффективности используются ссылочные аргументы (constFraction&).

Листинг13.2.fract6.cpp

tinclude<iostream> using namespace std;

class Fraction { private:

int num. den; // Числительизнаменатель, public:

Fraction{) {set(0, 1);}

Fraction{int n, int d) {set(n, d);>

Fraction(const Fraction bsrc);

void set(int n, int d) {num = n; den = d;

normalize();} intget_num() const {return num;} intget_den() const {return den;}

Fraction add(const Fraction bother);

Fraction mult(const Fraction bother);

Fraction operator*(const Fraction bother)

{return add(other);}

Fraction operator*(const Fraction bother)

{returnmult(other);}

private:

voidnormalizeO; // Перевести дробьв стандартную форму, intgcf(int a, int b) ; // Наибольшийобщийделитель, int lcm(int a, int b) ; // Наименьшийобщий

// знаменатель.

};

```
int main() {
Fraction £1(1, 2);
Fraction £2(lf 3);
Fraction f3 * fl + £2;
cout<<"1/2 + 1/3  “;
cout<< f3.get_nvm() <<"/";
cout<< f3.get_den() <<   <<endl;
return 0;
}
// ФУНКЦИИКЛАССAFRACTION
Fraction::Fraction(Fraction const&src) { num = src.num; den = src.den; }
// Нормализация: перевести дробь в стандартную форму,
// уникальную для каждого математически отличного значения. //
voidFraction::normalize(>{
11 Проверка на равенство О
if (den == 0 || num= = 0 )  { num = 0; den = 1;
}
// Поставить знак минус только в числителе. if (den< 0) {
num*= -1; den * = -1;
}
// Вынести за скобки наибольший общий делитель в
// числителе и знаменателе.
intn= gcf(num, den);
num= num / n;
```

```
den = den / n;
}
//Наибольшийобщийделитель //
int Fraction::gcf(int a, int b) { if (a % b == 0)
return abs(b);
else
returngcf(b, a % b);
}
// Наименьшийобщиймножитель //
int Fraction::lcm(int a, int b){ return (a / gcf(a, b)) * b;
}
Fraction Fraction::add(const Fraction &other) {
Fraction fract;
int led = lcm(den, other.den);
intquotl = led/den;
int quot2 = led/other.den;
fract.set(num * quotl + other.num * quot2, led);
fract.normalize(); return fract;
}
Fraction Fraction::mult(const Fraction bother)
{
Fraction fract;
fract.set(num * other.num, den * other.den); fract.normalize(); ' return fract;
}
```

Как это работает

Функции addи multперенесены из ранее существующего кола в класс Fraction. Все, что мы сделали, - это изменили тип аргумента, поэтому каждая из этих функции использует ссылочные аргументы, обеспечивая более эффективное выполнение.

Fraction add(const Fraction bother);

Fraction mult(const Fraction bother);

При изменении объявлений данных функций, описания функций также должны быть изменены, чтобы отобразить измененный тип аргумента. Но это изменение задевает только заголовок функции (ниже выделено жирным шрифтом). Остальноеописаниенеизменяется.

FractionFraction::add(constFractionbother) {

Fractionfract;

intled = lcm(den, other.den);

intquotl = led/den;

int quot2 = led/other.den;

fract.set(num * quotl + other.num * quot2, led); fract.normalize(); return fract;

}

Fraction Fraction::mult(const Fraction bother) {

Fraction fract;

fract.set(num * other.num, den * other.den); fract.normalize(); return fract;

}

Чтобы понять, как работают эти функции, вы.возможно, захотите кратко просмотреть главу 11.

В любом случае, функции операторов класса Fractionвыполняют только вызов соответствующей функции-члена (в нашем случае addили mult) и возвращают значение. Например, если компилятор видит выражение

fl+ f2

он транслирует это выражение путем вызова следующей функции: fl.operator+(f2)

Функция operator+ класса Fraction- это встраиваемая функция, описанная следующим образом:

Fraction operator*(const Fraction bother)

{returnadd(other);}

Поэтому вызов функции транслируется, в конечном итоге, так:

£l.add(£2)

Операции умножения обрабатываются таким же способом.

Выражения в функции maintестируют код функции оператора путем объявления дробей, их суммирования и вывода результатов.

```
Fractionfl(l,2);
Fractionf2(l,3);
Fractionf3 = fl + f2;
cout« "1/2 + 1/3 = ";
cout« f3.get_num() « */*;
cout« f3.get_den() « *. * « endl;
```

Упражнения

Упражнение 13.2.1. Измените функцию mainв примере 13.1 так, чтобы она запрашивала последовательность дробных значений, а цикл ввода прекращался при вводе 0 в качестве знаменателя. Напишите программу так,

чтобы она вычисляла сумму всех дробей, которые вводятся, и выводила результат на экран.

Упражнение 13.2.2. Напишите функцию operator- (вычитание) для класса Fraction. Упражнение 13.2.3. Напишите функцию operator/ (деление) для класса Fraction.

Работа с другими типами

Благодаря перегрузке, вы можете написать много различных функций для каждого из операторов, в которых каждая функции работает с различными типами. Например.вы можете написать несколько версий функции operator*. которая работает с классом Fraction:

```
class Fraction {
//. . . public:
operator*(const Fraction bother);
friend operator*(int n, const Fraction bfr);
friend operator*(const Fraction bfr,intn);
}
```

Каждая из этих функций (которые, кстати, должны быть где-нибудь описаны) работает с различными комбинациями операндов типов int и Fraction, позволяя вам записывать выражения следующего вида:

```
Fractionfractl= 1 + Fractiond, 2) + Fraction (3, 4) + 4;
```

Но существует более легкий способ поддержки операций с целыми числами. Что вам действительно необходимо, - это функция для преобразования целых чисел в объекты класса Fraction. Если бы такая операция использовалась, тогда от вас бы потребовалось написание только одной версии функции operator*. В выражении, таком как следующее, компилятор преобразует число 1 в формат класса Fraction, а затем вызовет функцию Fraction::operator-*-, чтобы сложить две дроби.

Fractionfractl= 1 + Fractiond, 2);

Оказывается, что такую функцию преобразования легко написать - она поставляется конструктором класса Fraction, который принимает один аргумент тина int!

Это простой конструктор и его можно эффективно использовать как встраиваемую функцию.

Fraction(int n) {set(n, 1) ; }

Если есть такое объявление, то все операции, объявленные дня двух объектов класса Fraction, автоматически расширяются, включая в себя операции, касающиеся объекта Fractionи целых чисел.

Функция присваивания класса (=)

Когда вы пишете класс, компилятор C++ автоматически обеспечивает вас тремя специальными функциями-членами. Пока я вас познакомил с двумя из них.

- Конструктор по умолчанию. Работа автоматической версии (поставляемой компилятором) состоит в присваивании каждому члену инициального значения, равного 0. Необходимо отмстить, что компилятор не будет использовать этот конструктор, если вы напишете свой собственный... поэтому вам следует всегда писать свой собственный конструктор по умолчанию, даже если он ничего не делает.
- Конструктор копирования. Работа автоматической версии состоит в выполнении простого копирования всех членов исходного объекта.
- Функция оператора присваивания (=). Это новая функция.

Функция оператора присваивания является специальной функцией, потому что компилятор сам се поставляет, если вы не делаете этого. Полому мы могли выполнять такие операции, как:

Fractionfl; /

fl= f2 + f3 ;

Работа функции operator= по умолчанию очень похожа на работу конструктора копирования: она также выполняет простое копирование всех членов. У вас может возникнуть вопрос: является ли функция оператора присваивания тоже конструктором копирования?

Нет, не является, хотя на первый взгляд кажется именно так. В обоих случаях всс значения одного объекта копируются (по умолчанию) в другой. Различие состоит в том, что конструктор копирования инициализирует новый объект, в то время как оператор присваивания копирует значения в уже существующий объект. В некоторых случаях (например классы, которые включают запрос к памяти или открытие файла) конструктору

копирования, возможно, придется выполнить большую работу, чем функция оператора присваивания.

Когда вы пишете вашу собственную функцию оператора присваивания, используйте следующий синтаксис:

class_name&operator=(const*class_name&source_arg)*

Такое объявление обладает интересной уловкой: оно похоже на конструктор копирования но функция operator= должна возвращать ссылку на объект класса, а также принимать ссылочный аргумент.

Здесь функция operatorsвозможно, похожа на класс Fraction:

class Fraction {

//. . . public:

Fraction &operator=(const Fraction &src) { set(src.num, sre.den); return *this;

} ;

};

В этом коде используется новое ключевое слово **this.** Я объясню использование ключевого слова **this** и других непонятных вещей функции оператора присваивания в следующей главе.

Между тем достаточно знать, что для класса, такого как этот, вам не нужно писать функцию оператора присваивания вообще. Работы по умолчанию здесь вполне достаточно, и компилятор всегда поставляет эту функцию оператора, если вы этого не делаете.

ГЛАВА 3. ГРАФИЧЕСКИЕ ПРОГРАММИРОВАНИЯ НА ЯЗЫКЕ С++

Язык программирования С++ обладает возможностями логического программирования программирования простых и сложных математических функции мадели применяемых в точных, естейтвенных и обшественных наук. Пользуясь С++ и его математическими алгаритмическими и другими возможностями написания программ описовающих физические процессы, явления по механики электричество и магнитизм оптика и атомная ядро. Квантовой физики, квантовой механики.

Химические процессы, реакции. Органические связонные, биохимические сложные процессы. В качестве премера приводим графические программы.

Графический программы написанный на языке С ++ содержать.

Имена #Include – <graphics.h> библиотека графических программ;

#Include<math.h>библиотека математических программ.

Пространство имен ползователя – usingnamespace. Std

Проссесорыdefine GetMax x 600

Define Get Max y 400

Точность переменных констант double, dublex = ø.øø1

Каждая программа C++ начинается объевление главной функции – intmain ( )

Главная программа завершается {

Returnø

}

Между главный функции

intmain ( )

{

- - - - - - -

- - - - - - -

- - - - - - -

Return ø }

Выполняемая программа в данном случае графической программы создание окна для графической программе initwindow (GetMaxX, GetMaxY) “FirstSample”

Написание математической модели формулы к примеру на следованного нами графической программы химической реакции 1 порядка – уравнение кинетики

Ni = Nø * exp (R * t1)

Горизонтальные и вертикальные оси графики

Out text xy (2ø 2ø “N0)

Outtextxy (570,370 “t”)

Установление света – Setcolor (7)

Использование графических операторах к примеру line

Line (60.30 , 30,15);

Завершение графической программы

Closegroph ( );

White (1 kbhit ());

Delay (200) – пауза

Returnø – завершение главной программы

Логической объединение теории программирование возможности современного обьектно–ориентированного программирований теории химических, физических, биологических, биофизических, биохимических, физических-биологохимических, и других простецов возможность научному исследование нами перечисленных простецов служится развития современном науки технике. В данном контрольном случае глубокого научечного демостронного и другие исследование математической модели уравнение кинетики Ni = N0 использованного для химического реакции первого порядка и другие.

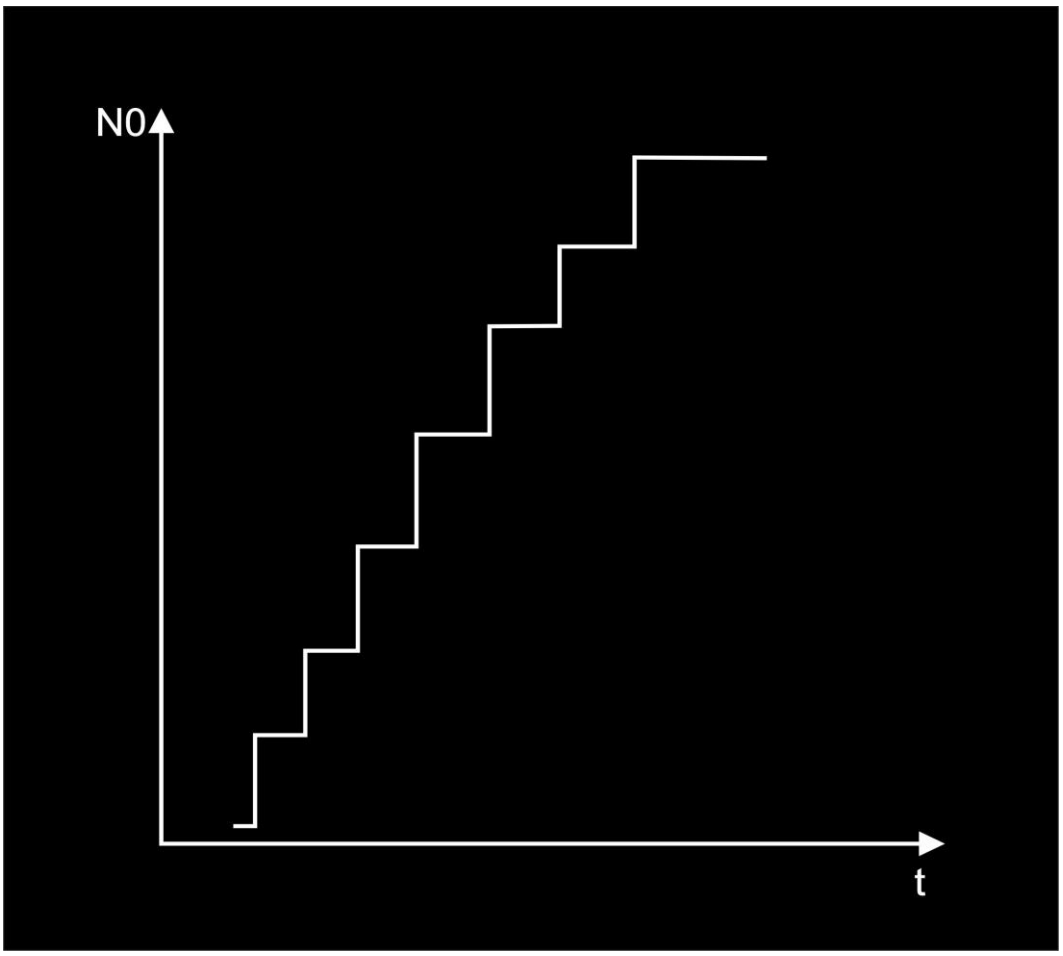
3.1. ХИМИЧЕСКАЯ РЕАКЦИЯ ПЕРВОГО ПОРЯДКА

$$No \cdot e^{kt}$$

```
#include <iostream>

#include <graphics.h>

#include <math.h>

#define GetMaxX 600
```

```
#define GetMaxY 400

using namespace std;

const double k=.001;

int main()

{

  double N0, t;

  N0=100;

  t=1;

  initwindow(GetMaxX, GetMaxY, "First Sample");

  double N1, t1;

  while(t<=60)

  {

       t1=t+.025;

       N1=N0*exp(k*t1);

       outtextxy(20,20,"N0");

       outtextxy(570,370,"t");

       setcolor(9);

       line(int(N0), GetMaxY-50*int(t), int(N1), GetMaxY-50*int(t1));

       N0=N1;

       t=t1;

       setcolor(7);
```

```
        line(50, 15, 50, GetMaxY-10);
        line(60, 30, 50, 15);
        line(40, 30, 50, 15);
        line(30, GetMaxY-40, GetMaxX-10, GetMaxY-40);
        line(GetMaxX-25, GetMaxY-30, GetMaxX-10, GetMaxY-40);
        line(GetMaxX-25, GetMaxY-50, GetMaxX-10, GetMaxY-40);
    }
    //closegraph();
    while (!kbhit( ))
        delay(200);
  return 0;
}
```



(3.2. масала)

...

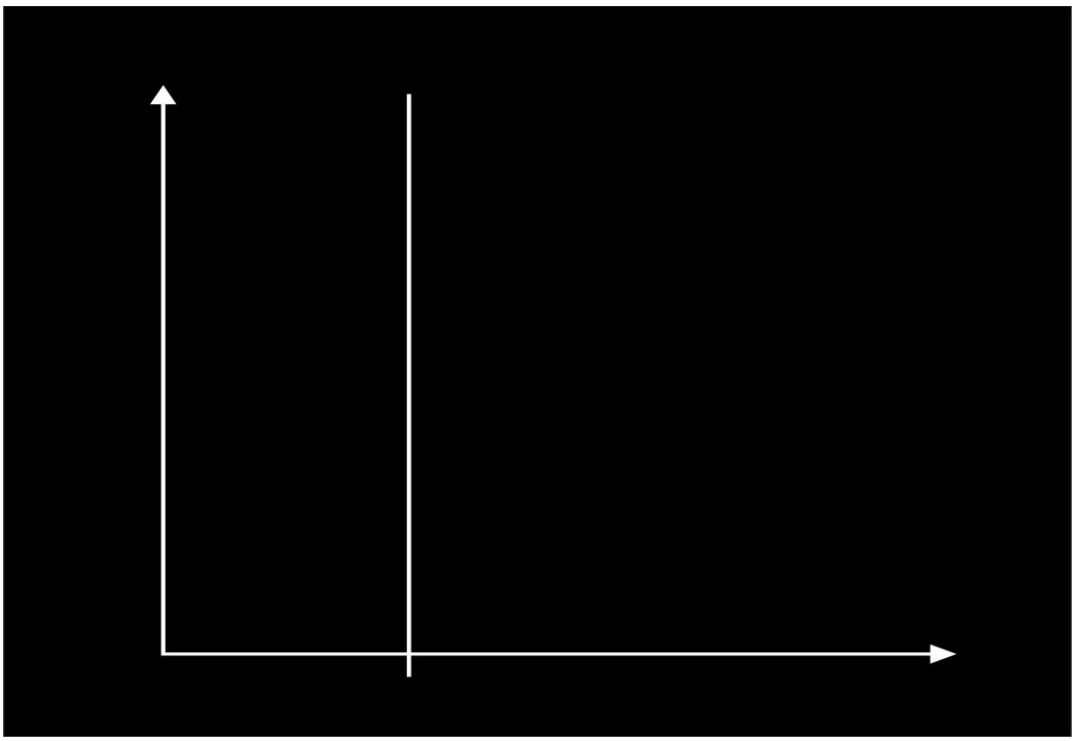
3.3. НЕГАТИВНАЯ ПРОВОДИМОСТЬ ПОЛУПРОВОДНИКА

$A \cdot T\sqrt{T} \cdot e^{-\Delta E/2kt}$

```
#include <iostream>

#include <graphics.h>

#include <math.h>

#define GetMaxX 500

#define GetMaxY 400
```

```
using namespace std;

const double k=1.38e-6;  // Boltsman doimiysi.

const double A=8;  // Emissiya doimiysi.

int main()

{

  double delta_E, T,N0,N1;

  delta_E=5, T=.05;

  initwindow(GetMaxX, GetMaxY, "First Sample");

  double T1;

  N0=A*T*sqrt(T)*exp(-delta_E/(2*k*T));

  while(T<=50)

  {

      T1=T+.05;

      N1=A*T1*sqrt(T1)*exp(-delta_E/(2*k*T1));

      line(int(5*N0), GetMaxY-10*int(T), int(5*N1), GetMaxY-10*int(T1));

      N0=N1;
```

```
    T=T1;
    line(50, 15, 50, GetMaxY-10);
    line(60, 30, 50, 15);
    line(40, 30, 50, 15);
    line(30, GetMaxY-40, GetMaxX-10, GetMaxY-40);
    line(GetMaxX-25, GetMaxY-30, GetMaxX-10, GetMaxY-40);
    line(GetMaxX-25, GetMaxY-50, GetMaxX-10, GetMaxY-40);
  }
  //closegraph();
  while (!kbhit( ))
    delay(200);
return   ;     }
```



3.4. ЗАКОН ИЗМЕНЕНИЯ КОЛЛЕКТОРНОГО ТОКА ВО ВРЕМЕНИ (T)

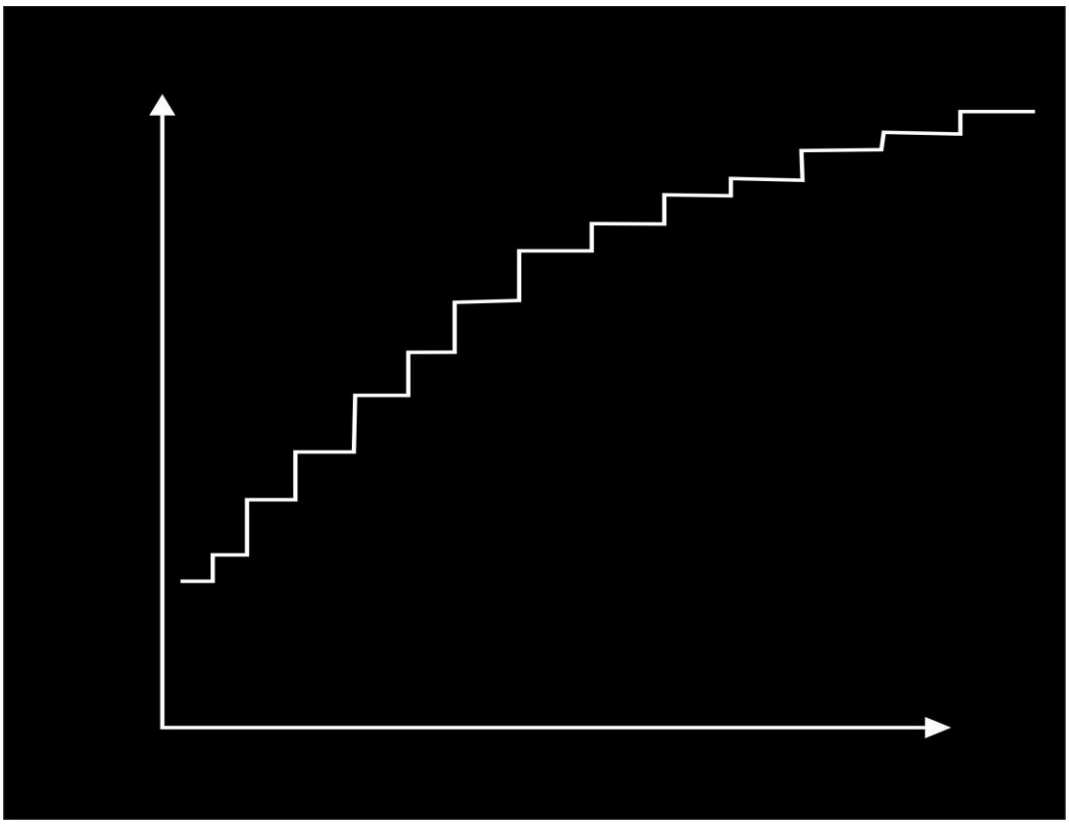
$K = I_0 \cdot e^{0{,}08 \cdot (T - T0)}$

```
#include <iostream>

#include <graphics.h>

#include <math.h>

#define GetMaxX 500

#define GetMaxY 400

using namespace std;

const double I0=15;

const double T0=25;

int main()
```

```
{
  double T,N0,N1;
  T=13;
  initwindow(GetMaxX, GetMaxY, "First Sample");
  double T1;
  N0=I0*exp(.08*(T-T0));
  while(T<=100)     {
       T1=T+.05;
       N1=I0*exp(.08*(T1-T0));
       line(int(10*N0), GetMaxY-10*int(T), int(10*N1), GetMaxY-10*int(T1));
       N0=N1;
       T=T1;
       line(50, 15, 50, GetMaxY-10);
       line(60, 30, 50, 15);
       line(40, 30, 50, 15);
       line(30, GetMaxY-40, GetMaxX-10, GetMaxY-40);
       line(GetMaxX-25, GetMaxY-30, GetMaxX-10, GetMaxY-40);
       line(GetMaxX-25, GetMaxY-50, GetMaxX-10, GetMaxY-40);
    }
   //closegraph();
   while (!kbhit( ))
```

delay(200);

return 0; }



ГЛАВА 4. БЕЗОПАСНОСТЬ ЖИЗНИДЕЯТЕЛЬНОСТИ

Требования к помещениям при работе за компьютером

Помещения должны иметь естественное и искусственное освещение. Расположение рабочих мест за мониторами для взрослых пользователей в подвальных помещениях не допускается. Площадь на одно рабочее место с компьютером для взрослых пользователей должна составлять не менее 6 м2, а объем не менее -20 м3.

Помещения с компьютерами должны оборудоваться системами отопления, кондиционирования воздуха или эффективной приточно-вытяжной вентиляцией.

Для внутренней отделки интерьера помещений с компьютерами должны использоваться диффузно-отражающие материалы с коэффициентом отражения для потолка — 0,7-0,8; для стен — 0,5-0,6; для пола — 0,3-0,5.

Поверхность пола в помещениях эксплуатации компьютеров должна быть ровной, без выбоин, нескользкой, удобной для очистки и влажной уборки, обладать антистатическими свойствами.

В помещении должны находиться аптечка первой медицинской помощи, углекислотный огнетушитель для тушения пожара.

Требования к микроклимату, ионному составу и концентрации вредных химических веществ в воздухе помещений

На рабочих местах пользователей персональных компьютеров должны обеспечиваться оптимальные параметры микроклимата в соответствии Сан Пен 2.2.4.548-96. Согласно этому документу для категории тяжести работ 1а температура воздуха должна быть в холодный период года не более 22-24оС, в теплый период года 20-25оС. Относительная влажность должна составлять 40-60%, скорость движения воздуха — 0,1 м/с. Для поддержания оптимальных значений микроклимата используется система отопления и кондиционирования воздуха. Для повышения влажности воздуха в помещении следует применять увлажнители воздуха с дистиллированной или кипяченой питьевой водой.

Ионный состав воздуха должен одержать следующее количество отрицательных и положительных аэрофонов; минимально необходимый уровень 600 и 400 ионов в 1 см3 воздуха; оптимальный уровень 3 000-5 000 и 1 500-3 000 ионов в 1 см3 воздуха; максимально допустимый — 50 000 ионов в 1 см3 воздуха. Для поддержания оптимального ионного состава воздуха, обеспыливания и обеззараживания воздуха в помещении рекомендуется применять аппараты завода «Диод» серии «Эллион».

Требования к освещению помещений и рабочих мест

В компьютерных залах должно быть естественное и искусственное освещение. Естественное освещение обеспечивается через оконные проемы с коэффициентом естественного освещения КЕО нениже 1,2% в зонах с устойчивым снежным покровом и нениже 1,5% на остальной территории. Световой поток из оконного проема должен падать на рабочее место оператора с левой стороны.

Искусственное освещение в помещениях эксплуатации компьютеров должно осуществляться системой общего равномерного освещения.

Освещенность на поверхности стола в зоне размещения документа должна быть 300-500 лк. Допускается установка светильников местного освещения для подсветки документов. Местное освещениене должно создавать бликов на поверхности экрана и увеличивать освещенность экрана более 300 лк. Прямую блескость от источников освещения следует ограничить. Яркость светящихся поверхностей (окна, светильники), находящихся в поле зрения, должна быть не более 200 кд/м2.

Отраженная блескость на рабочих поверхностях ограничивается за счет правильного выбора светильника и расположения рабочих мест по отношению к естественному источнику света. Яркость бликов на экране монитора не должна превышать 40 кд/м2. Показатель ослепленности для источников общего искусственного освещения в помещениях должен быть не более 20, показатель дискомфорта в административно-общественных помещениях не более 40. Соотношение яркости между рабочими поверхностями не должно превышать 3:1 — 5:1, а между рабочими поверхностями и поверхностями стен и оборудования 10:1.

Для искусственного освещения помещений с персональными компьютерами следует применять светильники типа ЛПО36 с зеркализованными решетками, укомплектованные высокочастотными пускорегулирующими аппаратами. Допускается применять светильники прямого света, преимущественно отраженного света типа ЛПО13, ЛПО5,

ЛСО4, ЛПО34, ЛПО31 с люминесцентными лампами типа ЛБ. Допускается применение светильников местного освещения с лампами накаливания. Светильники должны располагаться в видесплошных или прерывистых линий сбоку от рабочих мест параллельно линии зрения пользователя при разном расположении компьютеров. При периметра льном расположении — линии светильников должны располагаться локализованонад рабочим столом ближе к его переднему краю, обращенному к оператору. Защитный угол светильников должен быть не менее 40 градусов. Светильники местного освещения должны иметь не просвечивающийся отражатель с защитным углом не менее 40 градусов.

Для обеспечения нормативных значений освещенности в помещениях следует проводить чистку стекол оконных проемов и светильников не реже двух раз в год и проводить своевременную замену перегоревших ламп.

Требования к шуму и вибрации в помещениях

Уровни шума на рабочих местах пользователей персональных компьютеров не должны превышать значений, установленных СанПиН 2.2.4/2.1.8.562-96 и составляют не более 50 дБА. На рабочих местах в помещениях для размещения шумных агрегатов уровень шума не должен превышать 75 дБА, а уровень вибрации в помещениях допустимых значений по СН 2.2.4/2.1.8.566-96 категория 3, тип «в».

Снизить уровень шума в помещениях можно использованием звукопоглощающих материалов с максимальными коэффициентами звукопоглощения в области частот 63-8000 Гц для отделки стен и потолка помещений. Дополнительный звукопоглощающий эффект создают однотонные занавески из плотной ткани, повешенные в складку на расстоянии 15-20 см от ограждения. Ширина занавески должна быть в 2 раза больше ширины окна.

Требования к организации и оборудованию рабочих мест

Рабочие места с персональными компьютерами по отношению к световым проемам должны располагаться так, чтобы естественный свет

падал сбоку, желательно слева.

Схемы размещения рабочих мест с персональными компьютерами должны учитывать расстояния между рабочими столами с мониторами: расстояние между боковыми поверхностями мониторов не менее 1,2 м, а расстояние между экраном монитора и тыльной частью другого монитора не менее 2,0 м.

Рабочий стол может быть любой конструкции, отвечающей современным требованиям эргономики и позволяющей удобно разместить на рабочей поверхности оборудование учетом его количества, размеров и характера выполняемой работы. Целесообразно применение столов, имеющих отдельную от основной столешницы специальную рабочую поверхность для размещения клавиатуры. Используются рабочие столы с регулируемой и нерегулируемой высотой рабочей поверхности. При отсутствии регулировки высота стола должна быть в пределах от 680 до 800 мм.

Глубина рабочей поверхности стола должна составлять 800 мм (допускаемая не менее 600 мм), ширина — соответственно 1 600 мм и 1 200 мм. Рабочая поверхность стола не должна иметь острых углов и краев, иметь матовую или полуматовую фактору.

Рабочий стол должен иметь пространство для ног высотой не менее 600 мм, шириной — не менее 500 мм, глубиной на уровне колен — не менее 450 мм и на уровне вытянутых ног — не менее 650 мм.

Быстрое и точноесчитывание информации обеспечивается при расположении плоскости экрана ниже уровня глаз пользователя, предпочтительно перпендикулярно к нормальной линии взгляда (нормальная линия взгляда 15 градусов вниз от горизонтали).

Клавиатура должна располагаться на поверхности стола на расстоянии 100-300 мм от края, обращенного к пользователю.

Для удобства считывания информации с документов применяются подвижные подставки (пюпитры), размеры которых по длине и ширине

соответствуют размерам устанавливаемых на них документов. Пюпитр размещается в одной плоскости и на одной высосет экраном.

Для обеспечения физиологически рациональной рабочей позы, создания условий для ее изменения в течение рабочего дня применяются подъемно-поворотные рабочие стулья сиденьем и спинкой, регулируемыми по высоте и углам наклона, а также расстоянию спинки от переднего края сидения.

Конструкция стула должна обеспечивать:

ширину и глубину поверхности сиденья не менее 400 мм;

поверхность сиденья с закругленным передним краем;

регулировку высоты поверхности сиденья в пределах 400-550 мм и углом наклона вперед до 15 градусов и назад до 5 градусов.;

высоту опорной поверхности спинки 300±20 мм, ширину — не менее 380 мм и радиус кривизны горизонтальной плоскости 400 мм;

угол наклона спинки в вертикальной плоскости в пределах 0±30 градусов;

регулировку расстояния спинки от переднего края сидения в пределах 260-400 мм;

стационарные или съемные подлокотники длиной не менее 250 мм и шириной 50-70 мм;

регулировку подлокотников по высотенад сиденьем в пределах 230±30 мм и внутреннего расстояния между подлокотниками в пределах 350-500 мм.;

поверхность сиденья, спинки и подлокотников должна быть полумягкой, нескользящим неэлектризующимся, воздухонепроницаемым покрытием, легко очищаемым от загрязнения.

Рабочее место должно быть оборудовано подставкой для ног, имеющей ширину не менее 300 мм, глубину не менее 400 мм, регулировку по высоте в пределах до 150 мм и по углу наклона опорной поверхности подставки до 20 град. Поверхность подставки должна быть рифленой и иметь по переднему краю бортик высотой 10 мм.

Режим труда и отдыха при работе с компьютером

Режим труда и отдыха предусматривает соблюдение определенной длительности непрерывной работы на ПК и перерывов, регламентированных с учетом продолжительности рабочей смены, видов и категории трудовой деятельности.

Виды трудовой деятельности на ПК разделяются на 3 группы: группа А — работа по считыванию информации с экрана с предварительным запросом; группа Б — работа по вводу информации; группа В — творческая работа в режиме диалога с ПК .

Если в течение рабочей смены пользователь выполняет разные виды работ, то его деятельность относят к той группе работ, на выполнение которой тратится не менее 50% времени рабочей смены.

Категории тяжести и напряженности работы на ПК определяются уровнем нагрузки за рабочую смену: для группы А — по суммарному числу считываемых знаков; для группы Б — по суммарному числу считываемых или вводимых знаков; для группы В — по суммарному времени непосредственной работы на ПК. В таблице приведены категории тяжести и напряженности работ в зависимости от уровня нагрузки за рабочую смену.

Виды категорий трудовой деятельности с ПК

Категория работы по тяжести и напряженности	Уровень нагрузки за рабочую смену при видах работы на ПК		
	Группа А Количество знаков	Группа Б Количество знаков	Группа В Время работы, ч
I	До 20000	До 15000	До 2,0
II	До 40000	До 30000	До 4,0
III	До 60000	До 40000	До 6,0

Количество и длительность регламентированных перерывов, их

распределение в течение рабочей смены устанавливается в зависимости от категории работ на ПК и продолжительности рабочей смены.

При 8-часовой рабочей смене и работенка ПК регламентированные перерывы следует устанавливать:

для первой категории работ через 2 часа от начала смены и через 2 часа после обеденного перерыва продолжительностью 15 минут каждый;

для второй категории работ — через 2 часа от начала рабочей смены и через 1,5-2,0 часа после обеденного перерыва продолжительностью 15 минут каждый или продолжительностью 10 минут через каждый час работы;

для третьей категории работ — через 1,5- 2,0 часа от начала рабочей смены и через 1,5-2,0 часа после обеденного перерыва продолжительностью 20 минут каждый или продолжительностью 15 минут через каждый час работы.

При 12-часовой рабочей смене регламентированные перерывы должны устанавливаться в первые 8 часов работы аналогично перерывам при 8-часовой рабочей смене, а в течение последних 4 часов работы, независимо от категории и вида работ, каждый час продолжительностью 15 минут.

Продолжительность непрерывной работы на ПК без регламентированного перерыва не должна превышать 2 часа.

При работена ПК в ночную смену продолжительность регламентированных перерывов увеличивается на 60 минут независимо от категории и вида трудовой деятельности.

Эффективными являются нерегламентированные перерывы (микро паузы) длительностью 1-3 минуты.

Регламентированные перерывы и микро паузы целесообразно использовать для выполнения комплекса упражнений и гимнастики для глаз, пальцев рук, а также массажа. Комплексы упражнений целесообразно менять через 2-3 недели.

Пользователям ПК, выполняющим работу с высоким уровнем напряженности, показана психологическая разгрузка во время

регламентированных перерывов и в конце рабочего дня в специально оборудованных помещениях (комнатах психологической разгрузки).

Медико-профилактические и оздоровительные мероприятия. Все профессиональные пользователи ПК должны проходить обязательные предварительные медицинские осмотры при поступлении на работу, периодические медицинские осмотры с обязательным участием терапевта, невропатолога и окулиста, а также проведением общего анализа крови и ЭКГ.

Не допускаются к работена ПК женщины со времени установления беременности и в период кормления грудью.

Близорукость, дальнозоркость и другие нарушения рефракции должны быть полностью корригированы очками. Для работы должны использоваться очки, подобранные-с учетом рабочего расстояния от глаз до экрана дисплея. При более серьезных нарушениях состояния зрения вопрос о возможности работы на ПК решается врачом-офтальмологом.

Для снятия усталости аккомодационных мышц и их тренировки используются компьютерные программы типа Relax.

Интенсивно работающим целесообразно использовать такие новей шиесредства профилактики зрения, как очки ЛПО-тренер и офтальмологические тренажеры ДАК и «Снайпер-ультра».

Досуг рекомендуется использовать для пассивного и активного отдыха (занятия на тренажерах, плавание, езда на велосипеде, бег, игра в теннис, футбол, лыжи, аэробика, прогулки по парку, лесу, экскурсии, прослушивание музыки и т.п.). Дважды в год (весной и поздней осенью) рекомендуется проводить курс витаминотерапии в течение месяца. Следует отказаться от курения. Категорически должно быть запрещено курениена рабочих местах и в помещениях с ПК.

Обеспечение электро безопасности и пожарной безопасности на рабочем месте

Электро безопасность.

На рабочем месте пользователя размещены дисплей, клавиатура и системный блок. При включении дисплея на электронно-лучевой трубке создается высокое напряжение в несколько киловольт. Поэтому запрещается прикасаться к тыльной стороне дисплея, вытирать пыль с компьютера при его включенном состоянии, работать на компьютере во влажной одежде и влажными руками.

Перед началом работы следует убедиться в отсутствии свешивающихся со стола или висящих под столом проводов электропитания, в целостности вилки и провода электропитания, в отсутствии видимых повреждений аппаратуры и рабочей мебели, в отсутствии повреждений и наличии заземления приэкранного фильтра.

Токи статического электричества, наведенные в процессе работы компьютера на корпусах монитора, системного блока и клавиатуры, могут приводить к разрядам при прикосновении к этим элементам. Такие разряды опасности для человека не представляют, но могут привести к выходу из строя компьютера. Для снижения величин токов статического электричества используются нейтрализаторы, местное и общее увлажнение воздуха, использование покрытия полов с антистатической пропиткой.

Пожарная безопасность

Пожарная безопасность — состояние объекта, при котором исключается возможность пожара, а в случайного возникновения предотвращается воздействиена людей опасных его факторов и обеспечивается защита материальных ценностей.

Противопожарная защита — это комплекс организационных и технических мероприятий, направленных на обеспечение безопасности людей, предотвращение пожара, ограничение его распространения, а такжена создание условий для успешного тушения пожара.

Пожарная безопасность обеспечивается системой предотвращения пожара и системой пожарной защиты. Во всех служебных помещениях обязательно должен быть «План эвакуации людей при пожаре»,

регламентирующий действия персонала в случае возникновения очага возгорания и указывающий места расположения пожарной техники.

Пожары в ВЦ представляют особую опасность, так как сопряжены с большими материальными потерями. Характерная особенность

ВЦ — небольшие площади помещений. Как известно, пожар может возникнуть при взаимодействии горючих веществ, окислителя и источников зажигания. В помещениях ВЦ присутствуют все три основные фактора, необходимые для возникновения пожара.

Горючими компонентами на ВЦ являются: строительные материалы для акустической и эстетической отделки помещений, перегородки, двери, полы, перфокарты и перфоленты, изоляция кабелей и др.

Источниками зажигания в ВЦ могут быть электрические схемы от ЭВМ, приборы, применяемые для технического обслуживания, устройства электропитания, кондиционирования воздуха, где в результате различных нарушений образуются перегретые элементы, электрические искры и дуги, способные вызвать загорания горючих материалов.

В современных ЭВМ очень высокая плотность размещения элементов электронных схем. В непосредственной близости друг от друга располагаются соединительные провода, кабели. При протекании по ним электрического тока выделяется значительное количество теплоты. При этом возможно оплавление изоляции. Для отвода избыточной теплоты от ЭВМ служат системы вентиляции и кондиционирования воздуха. При постоянном действии эти системы представляют собой дополнительную пожарную опасность.

Для большинства помещений ВЦ установлена категория пожарной опасности В.

Одна из наиболее важных задач пожарной защиты — защита строительных помещений от разрушений и обеспечение их достаточной прочности в условиях воздействия высоких температур при пожаре. Учитывая высокую стоимость электронного оборудования ВЦ, а также

категорию его пожарной опасности, здания для ВЦ и части здания другого назначения, в которых предусмотрено размещение ЭВМ, должны быть первой и второй степени огнестойкости. Для изготовления строительных конструкций используются, как правило, кирпич, железобетон, стекло, металл и другие негорючие материалы. Применение дерева должно быть ограничено, а в случае использования необходимо пропитывать его огнезащитными составами.

Заключения

В развитии современной науки и техники а также всех отраслей техники технологии образование экономики, медицины и других особая роль пред надлежит теории, методологии, технологии, программирование в частности математическому моделированию оптимизации компьютерного графического программирование. В работе освещены некоторые вопросы теории, методологии, технологии программирование. Освещены основные операторы и другие программные средства С++. Разработаны графические компьютерные модельные программы показывающие изменение физических, биологических, химических процессов протекающих во времени –**t** возможность разработанных компьютерных программ в учебных процессах.

СПИСОК ЛИТЕРАТУРЫ

1. Керниган Б., Ритчи Д., Фьюэр А. Язык программирования

Си. -— М.: Финансы и статистика, 1985.

2. Уинер Р. Язык Turbo Си: Пер. с англ. — М.: Мир, 1991. —

384 с: йл.

3. Уэйт М., Прата С, Мартин Д. Язык Си. Руководство для начинающих:

Пер. с англ. — М.: Мир, 1988. — 512 с: ил.

4. Вирт И. Алгоритмы и структуры данных: Пер. с англ. — М.:

Мир, 1989. — 360 с: ил.

5. Зелковиц М., Шоу А., Гэннон Дж. Принципы разработки

программного обеспечения: Пер. с англ. — М.: Мир, 1982. —

386 с: ил.

6. Мик Б. и др. Практическое руководство по программированию:

Пер. с англ. — М.: Радио и связь, 1986. — 168 с: ил.

7. Фокс Дж. Программное обеспечение и его разработка: Пер. с

англ. — М.: Мир, 1985. — 368 с: ил.

8. Язык компьютера/ Под ред. и с предисл. В. М. Курочкина.

Пер. с англ. — М.: Мир, 1989. — 240 с: ил.

1. Александреску А. Современное проектирование на C++. Серия C++ In-Depth, т. 3 / Пер. с англ. — М.: Вильяме, 2002. — 336 с., ил.

2. Аммерааль Л. STL для программистов на C++ / Пер. с англ. — М.:

ДМК, 1999. - 240 с., ил.

3. АхоА., Сети Р., Ульман Дж. Компиляторы: принципы, технологии, инструменты / Пер. с англ. — М,: Вильяме, 2001. — 768 с., ил.

4. Бадд Т. Объектно-ориентированное программирование в действии / Пер. с англ. — СПб.: Питер, 1997. — 464 с., ил.

5. Бентли Дж. Жемчужина программирования. 2-е издание / Пер. с англ. -

СПб.: Питер, 2002. - 272 с., ил.

6. Берри Р. Микинз Б. Язык С: введение для программистов / Пер. с англ. -

М.: Финансы и статистика, 1988. — 191 с., ил.

7. Браунси К. Основные концепции структур данных и реализация в C++ / Пер. с англ. — М.: Вильяме, 2002. — 320 с., ил.

8. Буч Г. Объектно-ориентированный анализ и проектирование с примерами приложений на C++, 2-е изд. / Пер. с англ. — М.: Бином; СПб.: Невский диалект, 1998. — 560 с., ил.

9. Вирт Н. Алгоритмы + структуры данных = программы / Пер. с англ. М.: Мир, 1985. — 406 с., ил.

10. Гамма Э., Хелм Р., Джонсон Р., Влиссидес Дж. Приемы объектно-ориентированного проектирования. Паттерны проектирования. Серия Библиотека программиста / Пер. с англ. — СПб.: Питер, 2001. — 368 сил.496 Часть IV. Приложения

11. Голуб А. И. С и C++. Правила программирования. - М.: Бином, 1996.-272с.

12. Гринзоу Л. Философия программирования для Windows 95/NT / Пер.с англ. — СПб.: Символ-Плюс, 1997. — 640 с., ил.

13. Дейтел П. Дж., Дейтел Х. М. Как программировать на C++. Введение в объектно-ориентированное проектирование с использованием UML Пер. с англ. —- М.: Бином, 2002. — 1152 с.

14. Каррано Ф. М., Причард Дж. Дж. Абстракция данных и решение задач на C++. Стены и зеркала, 3-е издание/ Пер. с англ. — М.: Вильяме,

2003. - 848 с., ил.

15. Кениг Э., Му Б. Э. Эффективное программирование на C++. Серия

C++ In-Depth, т. 2 / Пер. с англ. — М.: Вильяме, 2002. — 384 с., ил.

16. Керниган Б. В., Пайк Р. Практика программирования / Пер. с англ.

СПб.: Невский Диалект, 2001. — 381 с., ил.

17. Керниган Б., Ритчи Д. Язык программирования Си / Пер. с англ.

3-е изд., испр. — СПб.: Невский Диалект, 2001. — 352 с., ил.

18. Киммел П. и др. BorlandC++ 5 / Пер. с англ. — СПб.: БХВ-Петербург,

2000.-976с., ил.

19. Круглински Д., Уингоу С., Шеферд Дж. Программирование на Microsoft

VisualC++ 6.0 для профессионалов/ Пер. с англ. — СПб.: Питер; М.:

Русская Редакция, 2001. — 864 с., ил.

20. Липпман С. Б. Основы программирования на C++. Серия C++ In-

Depth, т. 1 / Пер. с англ. — М.: Вильяме, 2002. — 256 с., ил.

21. Луис Д. BorlandC++ 5. Справочник / Пер. с нем. — М.: Бином, 3997. —560 с., ил.

22. Марченко А. Л. C++. Бархатный путь. — М.: Горячая линия - - Телеком, 1999. - 400 с.

23. Мейерс С. Эффективное использование C++. 50 рекомендаций по

улучшению наших программ и проектов. Серия Для программистов. /

Пер. с англ. — М.: ДМК Пресс, 2000. — 240 с., ил.

24. Мейерс С. Наиболее эффективное использование C++. 35 новых рекомендаций по улучшению наших программ и проектов. Серия Для

программистов / Пер. с англ. — М.: ДМК Пресс, 2000. — 304 с., ил.

25. Мейерс С. Эффективное использование STL. Библиотека программиста / Пер. с англ. — СПб.: Питер, 2002. — 224 с., ил.

26. Мешков А. В., Тихомиров Ю. В. VisualC++ и MFC / Пер. с англ.—

СПб.: БХВ-Петербург, 2000. - 1040 с., ил.

27. Москвин П.