МИНИСТЕРСТВО ПО РАЗВИТИЮ ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ И КОММУНИКАЦИЙ РЕСПУБЛИКИ УЗБЕКИСТАН

НУКУССКИЙ ФИЛИАЛ ТАШКЕНТСКОГО УНИВЕРСИТЕТА ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ ИМЕНИ МУХАММАДА АЛ-ХОРАЗМИ

Кафедра информационные образовательные технологии

«Профессиональное образования в сфере ИКТ»

Допуск к защите

зав. кафедрой_____

2017 г. «____» _____

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА

на тему «**МЕТОДИКА ОБУЧЕНИЯ ГРАФИЧЕСКОМУ ПРОГРАММИРОВАНИЮ НА ЯЗЫКЕ С++**»

Выпускник: Хожамбергенов Ж.

Научный руководитель: к.ф.-м.н. Алламуратов Ш.

Рецензент: д.т.н. Сейтназаров К.

НУКУС - 2017 г.

СОДЕРЖАНИЕ

Введение

Язык программирования C++ является одним из наиболее популярных средств объектно-ориентированного программирования, позволяющим разрабатывать программы, эффективные по объему кода и скорости выполнения. Последнее объясняется близостью языковых конструкций архитектуре ЭВМ и высокой выразительностью языка.

C++ включает большое число операций и типов данных, средства управления вычислительными процессами, механизмы модификации типов данных, средства управления вычислительными процессами, механизмы модификации типов данных и методы их обработки и, как следствие, является мощным языком программирования. Он позволяет описывать процессы обработки информации, начиная с уровня отдельных разрядов, видов и адресов памяти, переходя на основе механизмов объектно-ориентированного программирования к близким конкретным предметным областям понятиям.

Выпускная квалификационная работа состоит из трех глав. Первая глава посвящена алфавитам и идентификаторам в C++. Во второй главе были рассмотрены функции и операторы, где были подробно приведены операторы разветвления и цикла.

Третья глава было посвящено графическим программированиям на языке C++ где досконально были изучены функции графического режима. Кроме того были созданы графики некоторых функции на языке C++

ГЛАВА.1. АЛФАВИТЫ И ИДЕНТИФИКАТОРЫ В С++

1.1. Алфавит и идентификаторы

При написании программ на языке С++ 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* С++. 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 С++ включает латинские 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 «*горизонтальная 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). 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) *опций* и записи *выражений.* Например, запись *(a*+b*)e* является *выражением,* задающим вычисление суммы значений переменных ***a+b*** последующим умножением на значение e**.**

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 С++ ключевые слова приведены в приложении.

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

Рекомендации по *именованию:*

• истолковать 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;

1.2. 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 « "] = "
```

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 С++ (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.

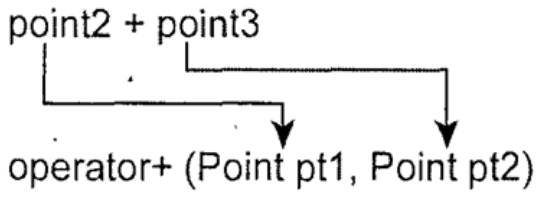
Глобальная функция оператора объявляется вне класса. Типы в списке аргументов определяют. какие типы операндов использует функция. Например, функция оператора сложения для класса Point может быть переписана как глобальная функция. Л вот объявление (прототип), которое должно появиться до вызова функции:

```
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 С++, 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 С++. Обычные правила ассоциативности и приоритетов выполняются для этих символов соответствующим образом (смотрите Приложение А).

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 = ptl.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.

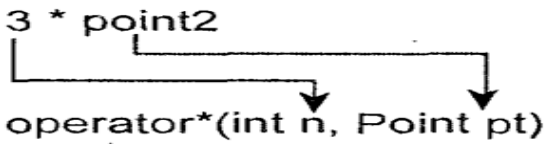
```
Pointoperator*(intn.Pointpt) {
Point new_pt;
```

```
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).

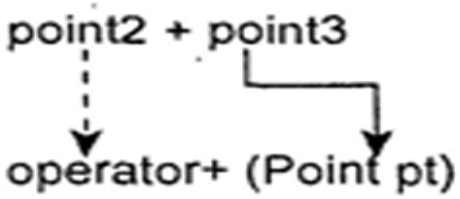
Ниже описана функция сложения для класса Point, а также функция оператора сложения (+), которая ее вызывает. Эта функция написана без использования ссылочных типов.

```
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. 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 addisubвыполняют 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).

ГЛАВА 2. ФУНКЦИИ И ОПЕРАТОРЫ

2.1. Функции и классы.

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 непосредованно 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 C++ 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;
}
```

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 | union имя_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.

2.2. Операторы условия (if, switch) и цикла (for, while)

1. *Логические значения, выражения и функции*

Оператор используется в основном для выполнения алгоритма данной задачи. Операторы делятся на линейную и управлении. Операторы во

многих случаях заканчиваются знаком «точка-запятой» (‘;’) и он со стороны компилятора принимается как отдельный оператор.

Во время создания программы используется пустой оператор ‘;’ . Вид оператора условия **if** задается следующим образом.

If(<условия>)<оператор>

Если условия "true" истинно то выполняется <оператор> , если "false" ложное то никакой операция не выполняется и управления переходит к следующему оператору (рис.1).

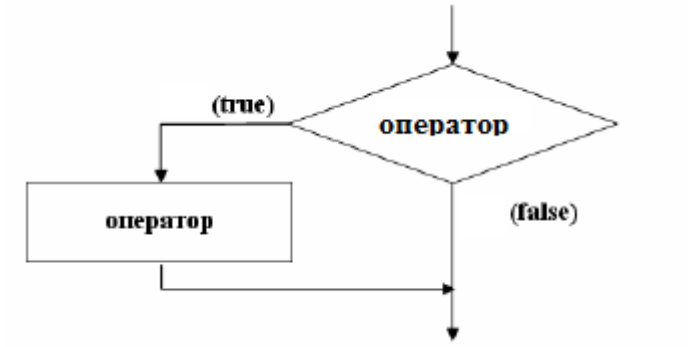


Рис.1

Внизу приведен программа использования оператора **if.**

```
#include<iostream.h>
int main()
{
int b;
cin>>b;
if (b>0)
{                    // выполнения условия  b>0
…
cout<< “b-положительное число”;
….
}
if (b<0)
cout<< “b-отрицательное число”; // выполнения условия  b<0
```

return 0;

}

Оператор if – else

Вид оператора **if – else** выглядит следуюшим образом:

if(<условия >)<operator$_1$>;else<operator$_2$>;

где <условия > true истинно то выполняется <operator$_1$>, иначе выполняется <operator$_2$> рис.2.

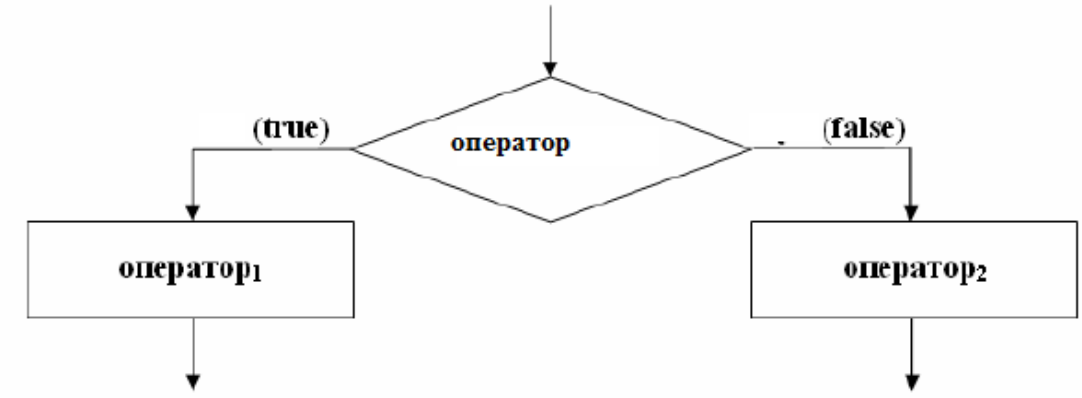


Рис.2. Блок схема оператора **if-else.**

Например рассмотрим задачу нахождения корни квадратного уравнения $ax^2+bx+c=0$:

```
#include <iostream.h>
#include <math.h>
int main()
{
float a, b, c;
float d, x1, x2;
cout<<“ax^2+bx+c=0  найти корни уравнения.”;
cout<< “\n  a – вводите  коэфициент:”;
cin>>a;
cout<< “\n  b – вводите  коэфициент:”;
cin>>b;
cout<< “\n  c – вводите  коэфициент:”;
cin>>c;
```

```
D=b*b-4*a*c;
if (D<0)
{cout<< “ уравнения не имеет  вещественных  корней! ”;
return 0;
}
if (D==0)
{cout<< “ уравнения имеет единственное решения: ”;
x1=-b/(2*a);
cout<< “\n x= ” <<x1;
return 0;
}
else
{cout<< “ уравнения имеет  двух корней: ”;
x1=(-b+sqrt(D)) / (2*a);
x2=(-b-sqrt(D)) / (2*a);
cout<< “\n x1= ” <<x1;
cout<< “\n x2= ” <<x2;
}
return 0;
}
```

Сначала вводится значения коэффициентов - a, b, c, потом вычисляется дискриминант –D. Если дискриминант отрицательный на экран выходит сообщение « уравнения не имеет вещественных корней!» и программа остановит свою работу. Если дискриминант не меньше нуля , то следующий оператор проверяет его на равнению на нуль. Если условия выполняется то на экран выходит сообщение “уравнения имеет единственное решения: ” и значения корня x, и программа остановит свою работу. Иначе если D>0 то выполняется следующий оператор и на экран выходит сообщения “ уравнения имеет двух корней:” и значения корней x_1 и x_2

с этим программа остановит свою работу и выход осуществляется с помощью оператора return.

Рассмотрим второй пример на нахождению максимальную значению, из данных трёх чисел.:

```
…
int x, y, z, max;
cin>>x>>y>>z;
if (x>y)
        if (y<z) max=z;
        else max=y;
else
        if (x<z) max=z;
        else max=x;
…
```

В этом пункте подробно рассматриваются операторы ветвления ("if " и "switch") и операторы циклов "for" и "while". Для применения всех этих операторов необходимо хорошо знать, что такое логические выражения и как они вычисляются.

Язык С++ унаследовал от языка Си соглашение, согласно которому целое значение 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").

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, сдан тест 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«"Введите возраст кандидата: ";
cin » candidate_age;
cout«"Введите результат тестирования:";
cin » candidate_score;
if ( acceptable(candidate age, candidate score ))
cout«"Этот кандидат сдал тест успешно.\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; }
```

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.

Оператор Switch

Еще один вид оператора условии это **switch** оператор разветвления. Его вид:

```
switch (<выражения>)
{
case<постоянное выражения1>:<группа операторов 1>; break;
case< постоянное выражения 2>:< группа операторов 2>; break;
…
case< постоянное выражения n>:< группа операторов n>; break;
default: < группа операторов n+1>;
}
```

В первую очередь вычисляется значения <выражения> , патом это значения сравнивается с помощью ключевого слова case с <постоянное выражения$_1$> . Если они совпадут то начиная со знака «:» этой строки до ключевого слова break выполняется <группа операторов $_1$> и управления

переходит к следующему оператору который расположен после операторов разветвления. Если <выражения> не совпадает некотором <постоянное выражения $_i$ > то выполняется строка в ключевом слове default < группа операторов $_{n+1}$>

Рассмотрим следующий пример ответ на вопрос «продолжит процесс? » Если будет взят положительный ответ то на экран выходит сообщения «процесс продолжается» и программа продолжает свою работу следующим оператором, иначе с ответом «процесс закончен» программа завершает свою работу.

```
#include <iostream.h>
int main()
{
char ответ= ‘ ’;
cout<< “продолжит процесс? (‘y’, ‘Y’): ”;
cin>>ответ;
switch(ответ)
{
case ‘Y’:
case ‘y’:
cout<< “продолжается процесс! \n”;
break;
default:
cout<< “процесс закончен! \n”
return 0;
}
…      // protsess
return 0;
}
```

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. 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 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 3.

Программу 2 можно переписать в более понятном виде спомощью функции.

```
#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.

Далее, программу 3 можно усовершенствовать, исключив описания всех переменных из "main()" и добавив две функции.

```
#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";
```

Оператор цикла for и while

Один из сильнейших механизмов управление программой это операторы цикла.

Оператор повторения в истинном значения выражения «условия цикла» выполняет тело цикла несколько (процесс итерации) раз (рис.3).

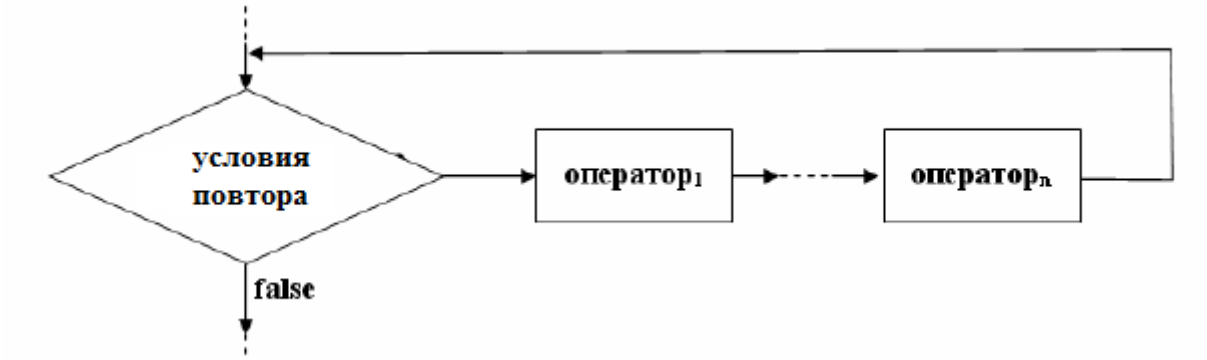


Рис.3. Блок схема оператора цикла.

Проверка условии повтора может быт перед выполнением операторов повтора в теле цикла (for, while) или после выполнения операторов в теле цикла (do-while).

Операторы повтора могут расположиться внутри один за другим.

Оператор цикла for

Синтаксис оператора for следующее:

for(<выражение$_1$>; <выражение$_2$>; <выражение$_3$>;) <operator или блок$_1$>;

Это оператор начинает свою работу с выполнение < выражение$_1$> . Патом начинается шаги повтора. На каждом шагу выполняется < выражение$_2$>, если результат отличен от 0 или true-истина, то цикл тело - < operator или блок > выполняется и в конце выполняется < выражение $_3$>. Если значение

< выражение $_2$> равно 0 (false), то процесс повтора остановится и управление передается следующему оператору.

Рассмотрим пример нахождение суммы чисел от 10 до 20.

```
#include <iostream.h>
int main()
{
int Summa=0;
for(int i=10; i<=20; i++)
Summa+=i;
cout<< “Сумма= ”<<Summa;
return 0;
}
```

В операторе for могут и не быть цикл повторения. Например если необходимо приостановит выполнение программы на некоторое время то это можно осуществить без дополнительных работ.:

```
#include <iostream.h>
int main()
{int delay;
…
for (delay=5000; delay>0; delay--);  // bos operator
…
return 0;}
```

В выше приведенном примере который вычисляет сумму от 10 до 20 можно вычислит с телом свободного оператора:

```
…
for(int i=10; i<=20; Summa+=i++);
…
```

На примере вычисление факториала можно показать работу блок операторов как тело оператора цикла:

```
#include <iostream.h>
int main()
{
int a;
unsigned long fact=1;
cout<< “Pu’tin sandi kiritin’:_ ”;
cin>>a;
if((a>=0)&&(a<33))
{
        for(int i=1; i<=a; i++) fact*=i;
        cout<<a<< “!= ”<<fact<< ‘\n’;
}
return 0;
}
```

Программа работает в диапазоне чисел от 0 до 33. Потому что 34! не вмещается на разряд unsigned long.

Оператор цикла while

Оператор цикла **while** работает в блоке повторно до результата (false или 0). Его синтаксис следующее:

while (<выражение>) <оператор или блок >;

Если < выражение > имеет вещественное значение, повторение будет бесконечным.

Оператор **while** условию повтора проверяет наперед. Если в начале повтора < выражение > фальшивый, то <оператор или блок > не выполняется.

В некоторых случаях < выражение > приходит как оператор присвоение. Где выполняется операция присвоение и результат сравнивается с нулем. Если результат отличен от нуля то продолжим цикл.

Точно как оператор for с помощью ‘,’ на <выражения> несколько действие могут выполняться синхронно. Например в программе «число и его квадрат» показан это случай:

```
#include <iostream.h>
int main()
{
int n, n2;
cout<< “Введите число (1…10): _ ”;
cin>>n;
n++;
while(n--, n2=n*n, n>0)
cout<< “n= ”<<n<< “n^2= ”<< n2<<endl;
return 0;
}
```

В выполнении оператора цикла в программе число n убывает до 1. На каждом шагу вычисляется n и его квадрат. Надо обратит внимание на последовательность написании операторов, потому что последний оператор принимается как условия повтора и значение n равняется к нулю цикл закончится.

С помощью оператора цикла **while** напишем программу нахождения наибольший общий делитель двух чисел. Используем алгоритм Евклида:

```
int main()
{
int a, b;
cout<< “Нахождение НОД двух чисел. \n”;
cout<< “ Введите натуральное число A и B’: ”;
cin>>a>>b;
while(a!=b)a>b?a-=b:b-=a;
cout<< “Это число НОД = ”<<a;
```

```
return 0;
}
```

На каждом шагу из чисел a и b отнимается с большого маленький. После повтора значение чисел a задается как результат.

Замечания вложенных циклов

В первоначальном варианте программы "таблица умножения" (программа 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.

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**).**

Цикл "for" всегда можно переписать в форме цикла "while", и наоборот.

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;
```

ГЛАВА 3. ГРАФИЧЕСКИЕ ПРОГРАММИРОВАНИЯ НА ЯЗЫКЕ С++

Язык программирования С++ обладает возможностями логического программирования программирования простых и сложных математических функции модели применяемых в точных, естественных и общественных наук.

3.1. Функции графического режима

Arc

Синтаксис:

void arc(int x, int y, int УголНачала, int УголКонца, int Радиус);

Вычерчивает дугу с центром в точке с координатами (x, y). Параметры Угол Начала и Угол Конца задают круговые координаты начальной и конечной точек линии дуги, которая вычерчивается против часовой стрелки от начальной точки к конечной. Уголовые координаты задаются в градусах.

Значение уголовые координаты возрастает против часовой стрелки. Параметр Радиус задает радиус дуги.

Линия дуги вычерчивается цветом, заданным функцией setcolor.

Заголовочный файл: <graph.h>

Bar

Синтаксис:

void bar(int x1, int y1, int x2, int y2);

Вычерчивает закрашенный прямоугольник. Параметры x1 и y1 задают положение левого верхнего угла прямоугольника, x2 и y2 – правого нижнего. Цвет и стиль заливки прямоугольника задаются функцией setfillstyle.

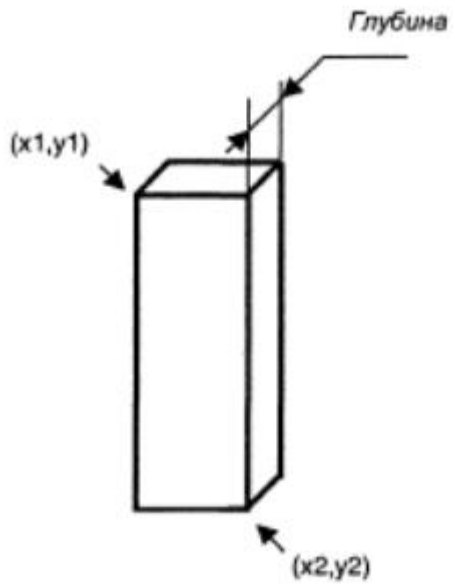
Заголовочный файл: <graph.h>

Bar3d

Синтаксис:

void bar3d(int x1, int y1, int x2, int y2, int Глубина, int B_Грань);

Вычерчивает параллелепипед. Параметры x1 и y1 задают положение левого верхнего, а x2 и y2 – правого нижнего угла ближней грани параллелепипеда. Параметр Глубина задает расстояние между передней и задней гранями, параметр B_Грань определяет, нужно ли вычерчивать границу верхней грани. Если параметр B_Грань равен нулю, то линия границы верхней грани не вычерчивается.



Цвет и стиль закраски ближней грани параллелепипеда можно задать при помощи функции setfillstyle, цвет линий границы – при помощи функции setcolor.

Заголовочный файл: <graph.h>

circle

Синтаксис:

void circle(int x, int y, int r)

Вычерчивает окружность радиуса r с центром в точке с координатами (x, y).

Цвет окружности можно задать при помощи функции setcolor.

Заголовочный файл: <graph.h>

drawpoly

Синтаксис:

void drawpoly(int КолТочек, int * Координаты);

Вычерчивает замкнутую ломаную линию, состоящую из отрезков прямых. Параметр КолТочек задает количество точек в результате последовательного соединения которых получается ломаная. Параметр Координаты задает массив координат узловых точек ломаной. Нулевой и первый элементы массива Координаты содержат координаты первой точки (x и y), второй и третий элементы содержат координаты второй точки и т.д.

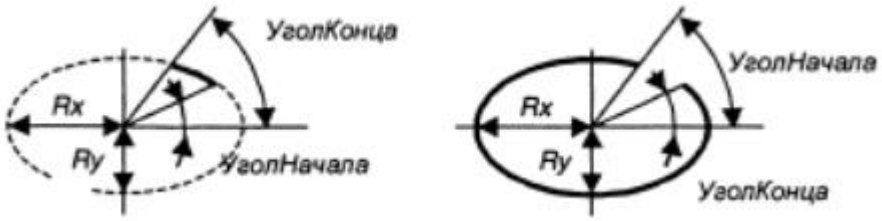
Заголовочный файл: <graph.h>

ellipse

Синтаксис:

void ellipse(int x, int y, int Угол Начала, int Угол Конца, int РадиусX, int РадиусY);

Вычерчивает эллипс или дугу эллипса с центром в точке с координатами (x, y). Параметры Угол Начала и Угол Конца задают круговые координаты начальной и конечной точек линии эллипса, которая вычерчивается против часовой стрелки от начальной точки к конечной. Угловые координаты задаются в градусах. Значение угловой координаты возрастает против часовой стрелки. Параметры РадиусX и РадиусY задают горизонтальный и вертикальный радиусы эллипса.



Линия эллипса или дуги вычерчивается цветом, установленным функцией setcolor.

Заголовочный файл: <graph.h>

getmaxx, getmaxy

Синтаксис:

int getmaxx(void);

int getmaxy(void);

Функция getmaxx возвращает координату x крайней правой точки экрана, функция getmaxy – координату y крайней нижней точки экрана.

Заголовочный файл: <graph.h>

getx, gety

Синтаксис:

int getx(void);

int gety(void);

Возвращает координату x(y) указателя вывода.

Заголовочный файл: <graph.h>

graphresult

Синтаксис:

int graphresult(void);

Возвращает результат (код ошибки) последней выполненной графической операции. Если операция выполнена успешно, функция возвращает ноль. Код ошибки выполнения графической операции устанавливают функции: bar, bar3d, initgraph, pieslice, setfillpattern, setfillstyle, setlinestyle, settextstyle и др.

Заголовочный файл: <graph.h>

grapherrormsg

Синтаксис:

char * grapherrormsg(int КодОшибки);

Возвращает указатель на строку, содержащую сообщение, соответствующее коду ошибки выполнения графической операции, указанному при вызове функции.

Заголовочный файл: <graph.h>

initgraph

Синтаксис:

void initgraph(int * Driver, int * Mode, char* Path);

Инициализирует графический режим. Параметр Driver определяет драйвер видеосистемы, параметр Mode – режим работы видеосистемы, параметр Path – путь к файлу драйвера.

Замечание

Обычно в качестве параметра Driver используют указатель на целую константу, значение которой равно DETECT. В этом случае функция initgraph сама определяет тип графического адаптера и устанавливает для него наилучший режим.

Заголовочный файл: <graph.h>

line

Синтаксис:

void line(int x1, int y1, int x2, int y2);

Вычерчивает линию из точки с координатами x1, y1 в точку с координатами x2, y2.

Цвет линии можно задать при помощи функции setcolor, стиль – при помощи функции setlinestyle.

Заголовочный файл: <graph.h>

lineto

Синтаксис:

void lineto(int x, int y);

Вычерчивает линию от текущего положение указателя вывода до точки, координаты которой указаны при вызове. Линия вычерчивается стилем, установленным функцией setlinestyle. Цвет линии можно задать, вызвав функцию setcolor.

Заголовочный файл: <graph.h>

linerel

Синтаксис:

void linerel(int dx, int dy);

Вычерчивает линию из точки текущего положения указателя вывода (xt, yt) в точку с координатами (xt+dx, yt+dy), т.е. координаты конца линии задаются в приращениях относительно текущих координат указателя вывода. Линия вычерчивается стилем, который устанавливается функцией setlinestyle. Цвет линии можно задать, вызвав функцию setcolor.

Замечание

Координаты указателя вывода можно получить при помощи функций getx и gety.

Заголовочный файл: <graph.h>

moveto

Синтаксис:

void moveto(int x, int y);

Перемещает указатель вывода в точку с указанными координатами.

Заголовочный файл: <graph.h>

moverel

Синтаксис:

void moverel(int dx, int dy);

Перемещает указатель вывода на dx и dy пикселов. Если значение параметра dx(dy) положительное, то указатель перемещается вниз (влево), если отрицательное, то – вверх (вправо).

Заголовочный файл: <graph.h>

outtext

Синтаксис:

void outtext(const char* Текст);

Выводит строку символов Текст от текущего положения указателя вывода и перемещает указатель вывода в точку, расположенную за последним выведенным символом.

Замечание

Строка, передаваемая функции outtext, не должна содержать символов форматирования, например \n.

Цвет выводимых символов можно задать при помощи функции setcolor, шрифт – settextstyle.

Заголовочный файл: <graph.h>

outtextxy

Синтаксис:

void outtextxy(int x, int y, const char* Текст);

Устанавливает указатель вывода в точку с координатами (x, y) и выводит от нее строку Текст, при этом указатель вывода своего положения не меняет, т.е. остается в точке с координатами (x, y).

Цвет выводимых символов можно задать при помощи функции setcolor, шрифт – settextstyle.

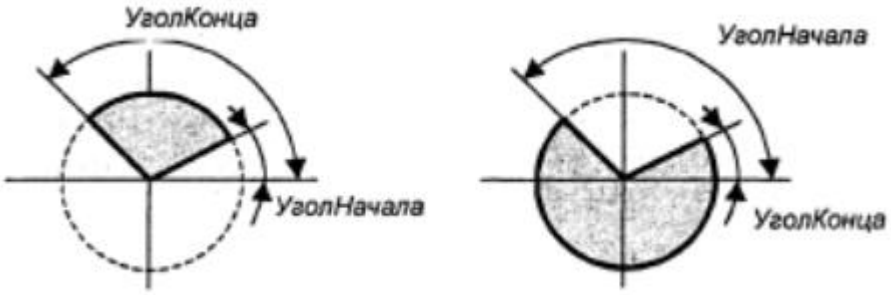
Заголовочный файл: <graph.h>

pieslice

Синтаксис:

void pieslice(int x, int y, int УголНачала, int УголКонца, int Радиус);

Вычерчивает круговой сектор радиуса Радиус с центром а точке с координатами (x, y). Параметры УголНачала и УголКонца задают круговые координаты начальной и конечной точек линии окружности, которая вычерчивается против часовой стрелки от начальной к конечной точке. Угловые координаты задаются в градусах. Значение угловой координаты возрастает против часовой стрелки. Нулевому углу соответствует горизонтальный отрезок, проведенный из точки (x, y) в сторону возрастания координаты x. Если УголНачала=0, а УголКонца=360, то функция pieslice вычерчивает круг.



Сектор закрашивается стилем и цветом, установленными функцией setfillstyle, линия границы вычерчивается цветом, установленным функцией setcolor.

Заголовочный файл: <graph.h>

putpixel

Синтаксис:

void putpixel(int x, int y, int цвет);

Окрашивает пиксел, точку с координатами (x, y), цветом Цвет. В качестве параметра Цвет обычно используют именованную константу (см. setcolor).

Заголовочный файл: <graph.h>

rectangle

Синтаксис:

void rectangle(int x1, int y1, int x2, int y2);

Вычерчивает прямоугольник. Параметры x1 и y1 задают положение левого верхнего угла прямоугольника, x2 и y2 – правого нижнего.

Вид (стиль линии) контура прямоугольника можно задать при помощи функции setlinestyle, цвет – при помощи функции setcolor.

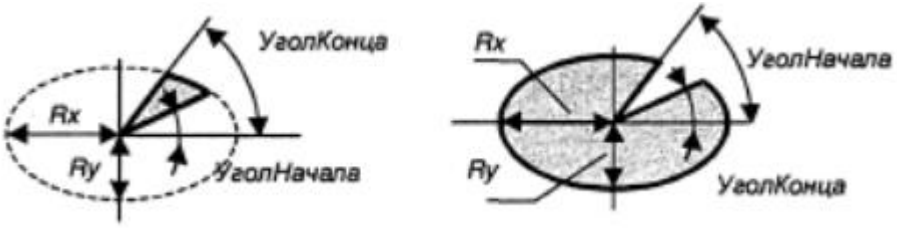
Заголовочный файл: <graph.h>

sector

Синтаксис:

void sector(int x, int y, int Угол1, int Угол2, int РадиусX, int РадиусY);

Вычерчивает эллиптический (РадиусX ≠ РадиусY) или круговой (РадиусX = РадиусY) сектор. Параметры x и y задают координаты центра сектора. Параметры Угол1 и Угол2 – углы прямых, ограничивающих сектор, параметры РадиусX и РадиусY – радиусы эллипса по осям X и Y, из которого «вырезается» сектор. Нулевому углу соответствует горизонтальный отрезок,



проведенный из точки (x, y) в сторону возрастание координаты x. Если Угол1=0, а Угол2=360, то функция sector вычерчивает полный круг (эллипс).

3.2. Создание графиков некоторых функции на C++

Цвет и стиль заливки можно задать при помощи функции setfillstyle, цвет границы сектора – при помощи функции setcolor.

Заголовочный файл: <graph.h>

setcolor

Синтаксис:

void setcolor(int Цвет);

Задает цвет вывода текста (функции outtextxy и outtext), вычерчивания линий и фигур (функции line, circle, rectangle и др.). В качестве параметра Цвет обычно используют именованную константу.

Цвет	**Константа**	**Значение константы**
Черный	BLACK	0
Синий	BLUE	1
Зеленый	GREEN	2
Бирюзовый	CYAN	3
Красный	RED	4
Сиреневый	MAGENTA	5

Коричневы	BROWN	6
Светло-серый	LIGHTGRAY	7
Серый	DARKGRAY	8
Голубой	LIGHTBLUE	9
Светло-зеленый	LIGHTGREEN	10
Светло-бирюзовый	LIGHTCYAN	11
Алый	LIGHTRED	12
Светло-сиреневый	LIGHTMAGENTA	13
Желтый	YELLOW	14
Белый (яркий)	WHITE	15

Заголовочный файл: <graph.h>

setfillstyle

Синтаксис:

void setfillstyle(int Стиль, int Цвет);

Устанавливает стиль и цвет заливки (закрашивания), используемый функциями вывода областей (bar, bar3d, sector и др.). В качестве параметра Стиль обычно используют одну из именованных констант, список которых приведен ниже. Параметр Цвет также задается именованной константой (см. setcolor).

Константа	**Стиль заполнения области**
EMPTY_FILL	Без заливки (сплошная заливка цветом фона)
SOLID_FILL	Сплошная заливка текущим цветом
LINE_FILL	Горизонтальная штриховка
LTSLASH_FILL	Штриховка под углом 45 градусов влево тонкими линиями
SLASH_FILL	Штриховка под углом 45 градусов влево
BKSLASH_FILL	Штриховка под углом 45 градусов вправо тонкими линиями

LTBKSLASH_FILL	Штриховка под углом 45 градусов вправо штриховка клеткой
HATCH_FILL	Штриховка клеткой
XHATCH_FILL	Штриховка под углом 45 градусов редкой косой клеткой
INTERLEAVE_FILL	Штриховка под углом 45 градусов частой косой клеткой
WIDEDOT_FILL	Заполнение редкими точками
CLOSEDOT_FILL	Заполнение частыми точками
USER_FILL	Тип заполнения определяется программистом

Заголовочный файл: <graph.h>

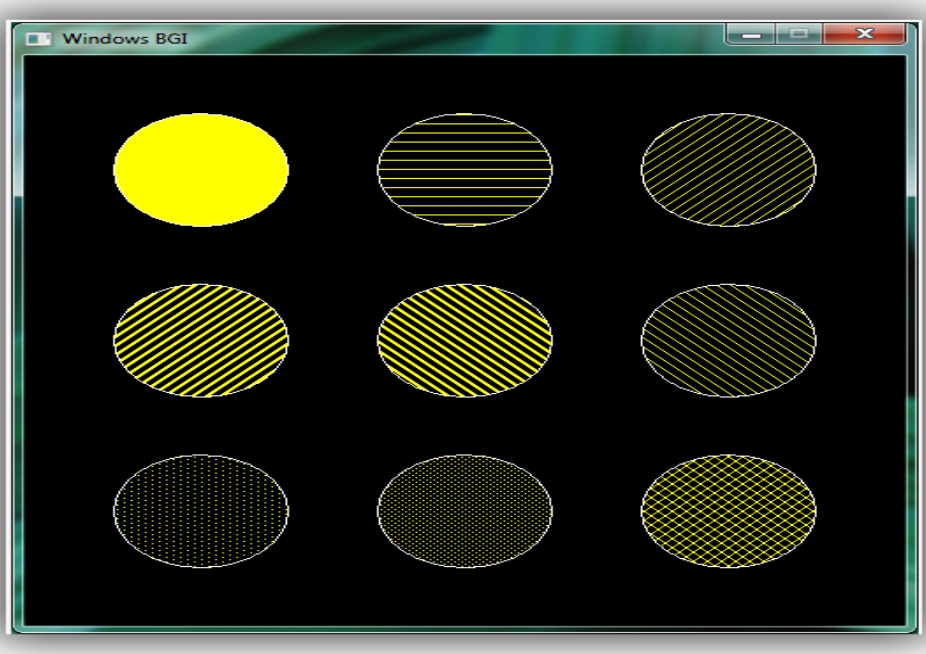
Рассмотрим пример создания цветных кругов расположенных внутри прямоугольника.

Программа 1.

```
#include <iostream.h>
#include <graph.h>
#include <conio.h>
using namespace std;
intmain()
{
initwindow(500, 500);

circle(100, 100, 50);
setfillstyle(1, YELLOW);
floodfill(100, 100, WHITE);

circle(250, 100, 50);
setfillstyle(2, YELLOW);
```

```
floodfill(250, 100, WHITE);

circle(400, 100, 50);
setfillstyle(3, YELLOW);
floodfill(400, 100, WHITE);

circle(100, 250, 50);
setfillstyle(4, YELLOW);
floodfill(100, 250, WHITE);

circle(250, 250, 50);
setfillstyle(5, YELLOW);
floodfill(250, 250, WHITE);

circle(400, 250, 50);
setfillstyle(6, YELLOW);
floodfill(400, 250, WHITE);

circle(100, 400, 50);
setfillstyle(10, YELLOW);
floodfill(100, 400, WHITE);

circle(250, 400, 50);
setfillstyle(11, YELLOW);
floodfill(250, 400, WHITE);

circle(400, 400, 50);
setfillstyle(8, YELLOW);
floodfill(400, 400, WHITE);
```

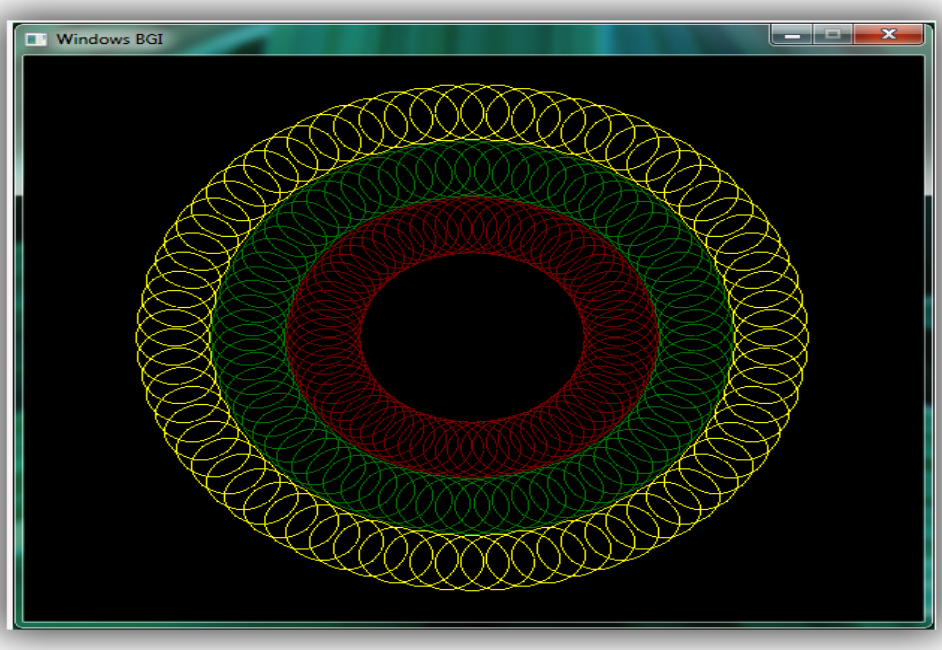
```
getch();
closegraph();
return 0;
}
```



Пример 2. Рассмотрим создания концентрических цветных кругов.

```
#include <iostream>
#include <graph.h>
#include <conio.h>
using namespace std;
int main()
{
int x, y, color, angle=0;
structarccoordstype a;
initwindow (600, 600);
delay (2000);
while (angle <= 360)
   {
```

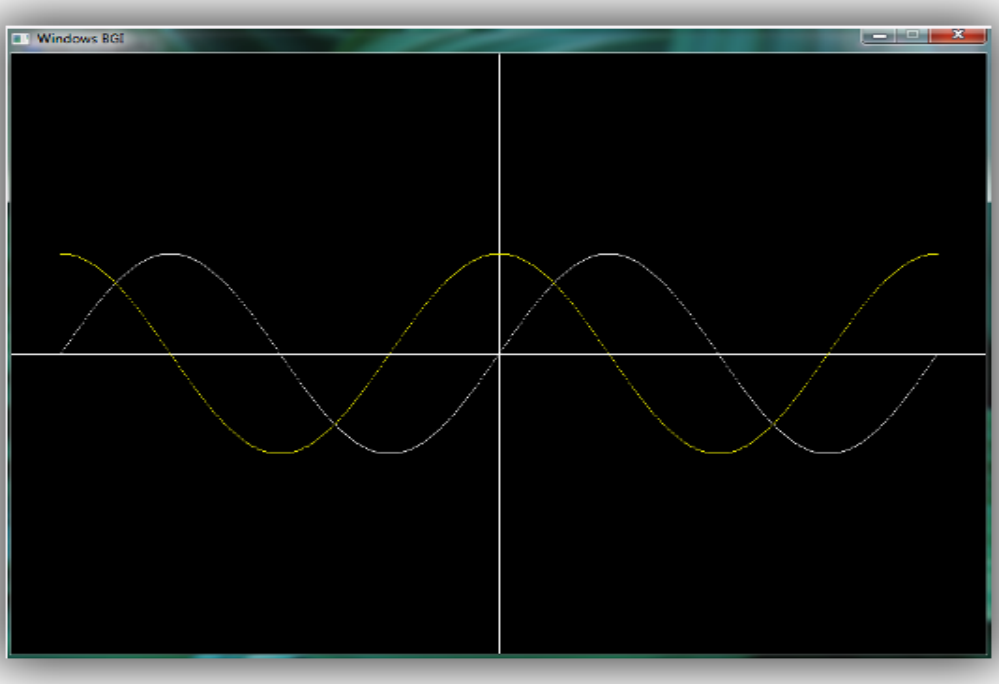
```
setcolor (BLACK);
arc (getmaxx()/2, getmaxy()/2, angle, angle+2, 100);
setcolor (RED);
getarccoords (&a);
circle (a.xstart, a.ystart, 25);
setcolor (BLACK);
arc (getmaxx()/2, getmaxy()/2, angle, angle+2, 150);
getarccoords (&a);
setcolor (GREEN);
circle (a. xstart, a.ystart, 25);
setcolor (BLACK);
arc (getmaxx()/2, getmaxy()/2, angle, angle+2, 200);
getarccoords (&a);
setcolor (YELLOW);
circle (a. xstart, a.ystart, 25);
angle = angle+5;
delay (50);
   }
getch();
closegraph();
return 0;
}
```



Пример 3. Рассмотрим создания графиков тригонометрических функции $y = 300 - \sin\left(\frac{\pi x}{180}\right) \cdot 100;$ $y = 300 - \cos\left(\frac{\pi x}{180}\right) \cdot 100$

```
#include <iostream>
#include <graph.h>
#include <conio.h>
#include <math.h>
using namespace std;
int main()
{
initwindow (800, 600);
line (0, 300, getmaxx(), 300);
line (400, 0, 400, getmaxy());
int x, y;
float pi=3.1415;
for(int a=-360; a<=360; a++)
   {
```

```
        x=400+a;
        y=300-sin(a*pi/180)*100;
putpixel (x, y, WHITE);
        y=300-cos(a*pi/180)*100;
putpixel (x, y, YELLOW);
delay(10);
    }
getch();
closegraph();
return 0;
}
```



Заключение

В последнее время язык С++ пополнился рядом нововведений. Среди них- стандартная библиотека шаблонов (Standard Template Library-STL), пространства имен (namespaces), идентификация типов на этапе выполнения (Run-Time Type Identification-RTTI) и булевский тип. Все эти новшества без труда могут быть использованы опытными программистами на базовом С++.

С++ был разработан в середине 80-х Бьерном Страуструпом из компании Bell Labs. Этот язык стал современным и могучим наследником языка С. С++ дополняет стандартный С концепцией классов –механизмом для создания типов данных, определяемых пользователем.

Выпускная квалификационная работа состоит из трех глав. Первая глава посвящена алфавитам идентификаторам в С++. Во второй главе были рассмотрены функции и операторы, где были подробно приведены операторы разветвления и цикла.

Третья глава было посвящено графическим программированиям на языке С++ где досконально были изучены функции графического режима. Кроме того были созданы графики некоторых функции на языке С++

Использованные литературы

Техника безопасности

1. Строго запрещается расположить кабинета информатики на бетонированную комнату и на подвале помещении. Полы должны быть деревянными и все компьютеры должны быть заземлены.

2. Требуется двойная изоляция всех электрических (220 В) линии проводов в том числе удлинители и компьютеры.

3. Надо расположить всех компьютеров в середине комнаты или рядом стены в двойном ряде друг против друга.

4. В кабинете должен быть единый отключатель для всех компьютеров .

5. Монитор компьютера должна быть на уровне глаз, и студенты должны иметь возможность соблюдать дистанцию от монитора на 40-80 см.

6. Клавиатуры компьютера должны быть на уровне согнутых колен. Для мышки должен быть место на обоих сторонах клавиатуры и они должны быть на одной высоте.

7. Продолжительность работы не должен превышать 60 минут на каждого студента.

8. Площадь компьютерного кабинета должна в шесть раз больше от количество компьютеров.

9. Компьютерные кабинеты должны быть оснащены системой вентиляции.

10. Надо предотвратит попадание пыли на клавиатуру.