Лабораторная работа 2. Введение в ООП

1. ЦЕЛЬ РАБОТЫ

Изучить основные понятия ООП: «объект», «класс», «инкапсуляция»; познакомиться со способами описания классов и объектов в языке C++; познакомиться с возможностью перегрузки операторов и использования конструкторов объектов класса; разработать приложения по своим вариантам заданий.

2. КРАТКИЕ ТЕОРЕТИЧЕСКИЕ ПОЛОЖЕНИЯ

Абстрактный тип данных — это множество объектов, определяемое списком компонентов (операций, применимых к этим объектам, и их свойств). Вся внутренняя структура такого типа спрятана от разработчика программного обеспечения - в этом и заключается суть абстракции. Абстрактный тип данных определяет набор функций, независимых от конкретной реализации типа, для оперирования его значениями.

Класс – это набор методов и свойств, описывающих поведение объекта. *Объект* – это конкретный представитель определенного класса. Таким образом, класс подобен стандартному элементу управлению в визуальных средах разработки, но без графического интерфейса. Класс используется аналогично, но его нельзя программировать визуально, как это делается в случае элементов управления.

Программу, в которой созданные классы близко совпадают с понятиями прикладной задачи, обычно легче понять, модифицировать и отладить, чем программу, в которой этого нет.

Инкапсуляция является одним из основных понятий ООП. Инкапсуляция – объединение в одном месте описания всех методов и данных класса. Инкапсуляция облегчает понимание работы программы, а также ее отладку и модификацию, так как внутренняя реализация используемых объектов редко интересует разработчика программного проекта; главное, чтобы объект обеспечивал функции, которые он должен предоставить. Кроме того, инкапсуляция защищает данные и методы класса от внешнего вмешательства или неправильного использования. Другими словами, инкапсуляция означает сокрытие деталей реализации класса внутри него самого.

Взаимодействие с объектом происходит через интерфейс, а детали реализации остаются инкапсулированными. В объектах интерфейсами являются свойства, методы и события. Только они предоставляются данным объектом в распоряжение других объектов. Таким образом, инкапсуляция обеспечивает использование объекта, не зная, как он реализован внутри.

Объявление класса

Заголовок определения. Определение класса начинается с ключевых слов `class`, `struct` или `union`. Правда, `union` применяется крайне редко. И структуры, и классы, и объединения относятся к “классовым” типам C++.

Спецификации доступа. Ключевые слова `private` и `public` называются *спецификаторами доступа.* Спецификатор `private` означает, что элементы

данных и элементы-функции, размещенные под ним, доступны только функциям-элементам данного класса. Это так называемый *закрытый доступ.*

Спецификатор `public` означает, что размещенные под ним элементы доступны как данному классу, так и функциям других классов и вообще любым функциям программы через представителя класса.

Есть еще спецификатор *защищенного* доступа `protected`, означающий, что элементы в помеченном им разделе доступны не только в данном классе, но и для функций-элементов классов, производных от него.

Структуры, классы и объединения. Типы, определяемые с ключевыми словами `struct`, `class` и `union`, являются классами. Отличия их сводятся к следующему:

- Структуры и классы отличаются только доступом по умолчанию. Элементы, не помеченные никаким из спецификаторов, в структурах имеют доступ `public` (открытый); в классах — `private` (закрытый).
- В объединениях по умолчанию принимается открытый доступ.
- Элементы (разделы) объединения, как и в C, перекрываются, т. е. начинаются с одного и того же места в памяти.

Элементы данных и элементы-функции. Элементы данных класса совершенно аналогичны элементам структур в C++, за исключением того, что для них специфицирован определенный тип доступа. Объявления элементов-функций аналогичны прототипам обычных функций. Если определение функции располагается вне тела класса, то к ее имени добавляется префикс, состоящий из имени класса и операции разрешения области действия (`::`).

Класс как область действия. Имена элементов класса расположены в области действия класса, и к ним можно обращаться из функций-элементов данного класса. Кроме того, получить доступ к элементам класса можно в следующих случаях:

- Для существующего представителя класса с помощью операции доступа к элементу (точки).
- Через указатель на существующий представитель класса с помощью операции косвенного доступа к элементу (стрелки).
- С помощью префикса, состоящего из имени класса и операции разрешения области действия (`::`).

Доступ к элементам данных. Поскольку функции-элементы класса находятся в его области действия, они могут обращаться к элементам данных непосредственно по имени. Обычные функции или функции-элементы других классов могут обращаться к элементам существующего представителя класса с помощью операций “.” или “->”. Пример:

```
class Time {public:
                int hour;
                int min;   } ;
int main()
{Time start; // Объявление локального объекта класса Time.
 Time *pTime = &start; //Создаем указатель на локальный объект.
 start.hour = 17; // Операция доступа к элементу.
```

```
    pTime->min = 30; // Косвенный доступ к элементу.
    return 0;}
```

Вызов функций-элементов класса. Совершенно аналогично тому, что имеет место в случае элементов-данных, функции-элементы класса могут вызываться функциями-элементами того же класса просто по имени. Обычные функции и элементы других классов могут вызывать функции-элементы данного класса для существующих его представителей с помощью операций “.” или “->” (через указатель).

Добавим в класс Time функции-члены класса, позволяющие устанавливать время и выводить информацию о текущем времени на экран. Теперь информацию о классе Time можно представить с помощью следующей таблицы:

`class Time`		
Поля/Свойства (элементы данных) класса		
Название	тип	Описание
`hour`	`int`	количество часов
`min`	`int`	количество минут
Методы (функции-элементы) класса		
Название и тип возвращаемого значения	Аргументы	Описание
`void SetTime`	`int h` `int m`	Устанавливает значение часов *h* и минут *m*.
`void ShowTime`	нет	Выводит время на экран

```
#include <stdio.h>
class Time {int hour;
            int min;
        public:
           void SetTime(int h, int m)
                  {hour = h; min = m; }
           void ShowTime(void)
                  {printf("Time: %02d:%02d\n", hour, min); }
                   };
int main()
{Time start;  Time *pStart = &start;
 int hr, min;
 start.SetTime(17, 15); // Вызов элемента для объекта start.
 pStart->ShowTime(); // вызов элемента через указатель на объект.
  return 0;}
```

Указатель `this`. Любая функция-элемент класса, не являющаяся статической имеет доступ к объекту, для которого она вызвана, через посредство ключевого слова `this`. Типом `this` является *имя_класса**. Например:

```
class Dummy {void SomeFunc(void) {...};
             public:
             Dummy();
             };
Dummy::Dummy()
{SomeFunc();  this->SomeFunc();  (*this).SomeFunc();
}
```

В этом примере каждый оператор конструктора (функция `Dummy()`) вызывает одну и ту же функцию `SomeFunc()`. Поскольку функции-элементы могут обращаться к элементам класса просто по имени, подобное использование указателя `this` довольно бессмысленно. Это ключевое слово чаще всего применяется для возврата из функции-элемента указателя или ссылки на текущий объект.

Перегрузка операций

Язык C++ позволяет переопределять для классов существующие обозначения операций. Это называется *перегрузкой операций.* Благодаря ей класс можно сделать таким, что он будет вести себя подобно встроенному типу. Например, в классе можно перегрузить такие операци: `+, -, *, /, =, >, <, >=, <=, +=` и другие.

Функции-операции, реализующие перегрузку операций, имеют вид

```
тип_возвращаемого_значения operator знак_операции
                ([операнды]) {тело функции}
```

Если функция является элементом класса, то первый операнд соответствующей операции будет самим объектом, для которого вызвана функция-операция (его не надо указывать). В случае одноместной операции список параметров будет пуст. Для двухместных операций функция будет иметь один параметр, соответствующий второму операнду. Если функция-операция не является элементом класса, она будет иметь один параметр в случае одноместной операции и два — в случае двухместной.

Для перегрузки операций существуют такие правила:

- Приоритет и правила ассоциации для перегруженных операций остаются теми же самыми, что и для операций над встроенными типами.
- Нельзя изменить поведение операции по отношению к встроенному типу.
- Функция-операция должна быть либо элементом класса, либо иметь один или несколько параметров типа класса.
- Функция-операция не может иметь аргументов по умолчанию.
- Обычно операцию присваивания определяют так, чтобы она возвращала ссылку на свой объект. В этом случае сохраняется семантика арифметических присваивании, допускающая последовательные присваивания в выражении (т.е. `c = b = a;`).

- Невозможно изменить синтаксис перегруженных операций. Одноместные операции должны быть одноместными, а двухместные — двухместными.
- Нельзя изобретать новые обозначения операций. Возможные операции ограничиваются тем списком, что приведен в начале этого раздела.
- Желательно сохранять смысл перегружаемой операции.

Пример класса VECTOR с перегруженными операциями

Определим класс для работы с трехмерными векторами в евклидовом пространстве. В этом классе будут использоваться перегруженные операции сложения (знак +) и присваивания (знак =) как операции с трехмерными векторами. Сумма двух векторов будет вычисляться как вектор, компоненты которого равны суммам соответствующих компонент слагаемых. Операция = будет выполнять, как и положено, покомпонентное присваивание векторов.

Вначале создадим файл Vector.h:

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

class vector
{int x,y,z ;
 public:
 vector operator+(vector t);
 vector operator=(vector t);
 void show(void);
 void assign(int , int , int);
 };
//функция для инициализации вектора
void vector::assign(int mx, int my, int mz)
{    x=mx; y=my; z=mz;
 }
//функция для вывода вектора на экран
void vector::show(void)
{cout<<x<<", "; cout<<y<<", "; cout<<z<<"\n";
 }
//перегрузка оператора присваивания
vector vector::operator=(vector t)
{x=t.x; y=t.y; z=t.z;
 return *this;
 }
//перегрузка оператора «плюс»
vector vector::operator+(vector t)
{vector temp;
 temp.x=x+t.x; temp.y=y+t.y; temp.z=z+t.z;
 return temp; }
```

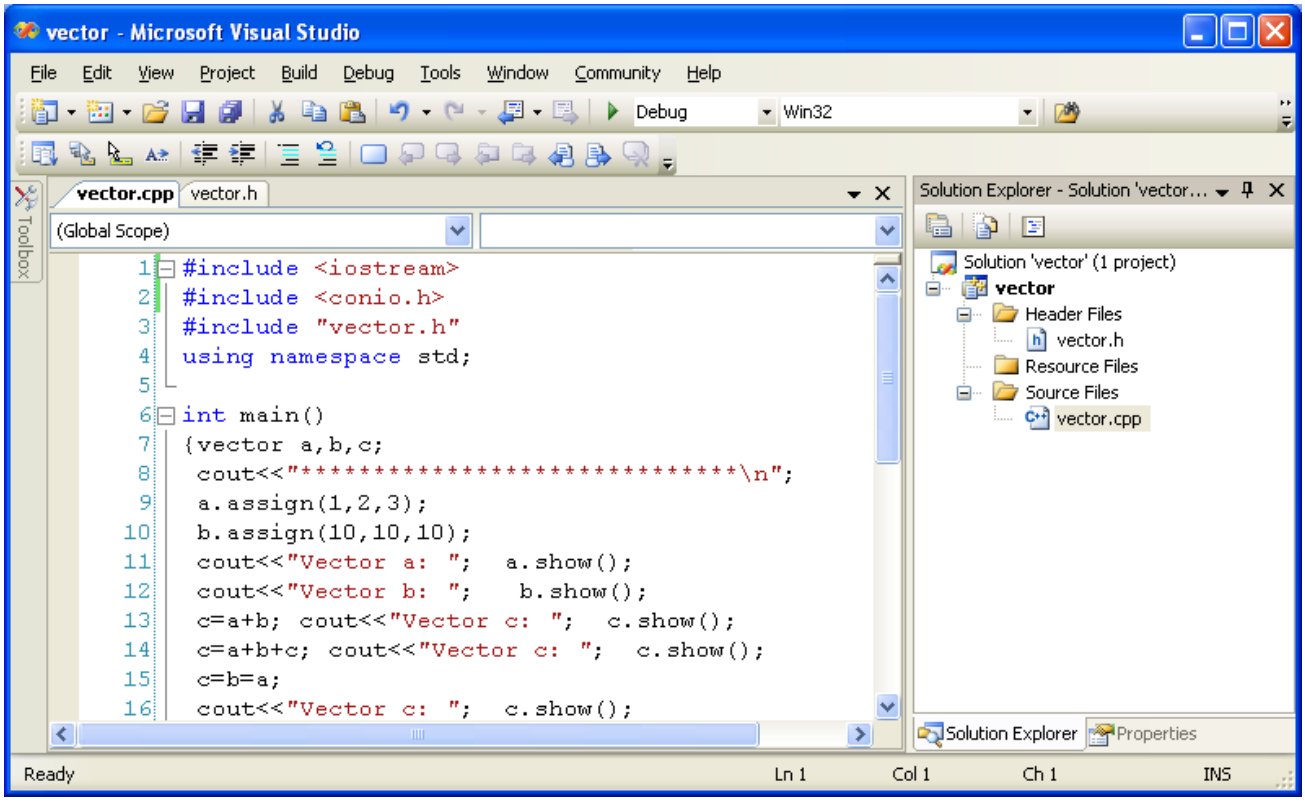
Затем создадим файл Vector.cpp:

```
#include <iostream>
#include <conio.h>
#include "vector.h"
```

```
using namespace std;

int main()
{vector a,b,c;
 cout<<"*****************************\n";
 a.assign(1,2,3);
 b.assign(10,10,10);
 cout<<"Vector a: ";  a.show();
 cout<<"Vector b: ";   b.show();
 c=a+b; cout<<"Vector c: ";  c.show();
 c=a+b+c; cout<<"Vector c: ";  c.show();
 c=b=a;
 cout<<"Vector c: ";  c.show();
 cout<<"Vector b: ";  b.show();
 getch();
 return 0; }
```

Окно IDE Visual Studio с проектом может выглядеть примерно так:



После запуска программы получим на экране:

```
*****************************
Vector a: 1, 2, 3
Vector b: 10, 10, 10
Vector c: 11, 12, 13
Vector c: 22, 24, 26
Vector c: 1, 2, 3
Vector b: 1, 2, 3
```

Конструкторы и деструкторы. Как известно, в классе могут быть объявлены две специальные функции-элемента – *конструктор* и *деструктор*.

Конструктор отвечает за создание представителей данного класса. Его объявление записывается без типа возвращаемого значения и ничего не возвращает, а имя должно совпадать с именем класса. Конструктор может иметь любые параметры, необходимые для *конструирования,* т. е. создания, нового представителя класса. Если конструктор не определен, компилятор генерирует *конструктор по умолчанию,* который просто выделяет память, необходимую для размещения представителя класса.

С помощью конструктора можно при создании объекта элементам данных класса присвоить некоторые начальные значения, определяемые в программе. Это реализуется путем передачи аргументов конструктору объекта.

Конструкторы можно перегружать, чтобы обеспечить различные варианты задания начальных значений объектов класса. Конструктор может содержать значения аргументов по умолчанию. Для каждого класса может существовать только один конструктор с умолчанием.

Когда объявляется объект класса, между именем объекта и точкой с запятой в скобках можно указать *список инициализации элементов*. Эти начальные значения передаются в конструктор класса при вызове конструктора.

Деструктор отвечает за уничтожение представителей класса. Если деструктор не определен, генерируется деструктор по умолчанию, который просто возвращает системе занимаемую объектом память. Деструктор объявляется без типа возвращаемого значения, ничего не возвращает и не имеет параметров. Имя деструктора совпадает с именем класса, но перед ним ставится символ ~ (тильда). Класс может иметь только один деструктор, то есть перегрузка деструктора запрещена.

Представленные в качестве примеров классы не были до сих пор обеспечены деструкторами. На самом деле, деструкторы редко используются с простыми классами. Деструкторы имеют смысл в классах, использующих динамическое распределение памяти под объекты (например, для массивов или списков, …).

Если для класса не определено никакого конструктора, компилятор создает сам конструктор с умолчанием. Такой конструктор не задает никаких начальных значений, так что после создания объекта нет никакой гарантии, что он находится в непротиворечивом состоянии.

Области памяти, занятые данными базовых типов, таких как `int`, `float`, `double` и др., выделяются и освобождаются автоматически и не нуждаются в помощи конструктора и деструктора.

Пример. Усовершенствуем класс `vector`. Объявим в нем конструктор с аргументами по умолчанию (конструктор с умолчанием). Задание в конструкторе аргументов по умолчанию позволяет гарантировать, что объект будет находиться в непротиворечивом состоянии, даже если в вызове конструктора не указаны никакие значения.

Вначале изменим файл Vector.h:

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

```
class vector
{int x,y,z ;
 public:
 vector operator+(vector t);
 vector operator=(vector t);
 void show(void);
 vector(int , int , int);//конструктор с параметрами
 ~vector();              //деструктор
 };

//определение конструктора с аргументами по умолчанию
//для задания начальных значений закрытых данных
vector::vector(int mx=0, int my=0, int mz=0)
{x=mx; y=my; z=mz;
 cout<< "Вектор с координатами ("<<x<<","<<y<<","<<z<< ")
                              создан.\n";
 }
//определение деструктора
vector::~vector()
{cout<< "Вектор с координатами ("<<x<<","<<y<<","<<z<< ")
                              разрушен.\n";
 }
void vector::show(void)
{cout<<x<<", "; cout<<y<<", "; cout<<z<<"\n";
 }

vector vector::operator=(vector t)
{x=t.x; y=t.y; z=t.z;
 return *this; }

vector vector::operator+(vector t)
{vector temp;
 temp.x=x+t.x; temp.y=y+t.y; temp.z=z+t.z;
 return temp;
 }
```

Затем изменим файл Vector.cpp:

```
#include <iostream>
#include "vector1.h"
using namespace std;

int main()
{ cout<<"*****************************\n";
//создадим три объекта класса vector
 vector a(1,2,3), b(10,10,10),c;
 cout<<"Vector a: ";
 a.show();
 cout<<"Vector b: ";
 b.show();
 c=a+b; cout<<"Vector c: ";   c.show();
```

```
c=a+b+c; cout<<"Vector c: ";  c.show();
c=b=a;
cout<<"Vector c: ";  c.show();
cout<<"Vector b: ";  b.show();
}
```

Данная программа создает три объекта класса `vector`: два объекта со всеми тремя указанными аргументами, а один (*c*) – с аргументами по умолчанию.

Перегрузка операций «поместить в поток» и «взять из потока»

C++ способен вводить и выводить стандартные типы данных, используя операции «поместить в поток» >> и операцию «взять из потока» <<. Эти операции уже перегружены в библиотеках классов, которыми снабжены компиляторы C++, чтобы обрабатывать каждый стандартный тип данных, включая строки и адреса памяти. Операции «поместить в поток» и «взять из потока» можно также перегрузить для того, чтобы выполнять ввод и вывод типов, определенных пользователем.

Необходимо обратить внимание, что ни функция вывода, ни функция ввода не могут быть членами класса. Это связано с тем, что, если операторная функция является членом класса, левый операнд (неявно передаваемый с помощью указателя `this`) должен быть объектом класса, который сгенерировал обращение к этой операторной функции. И это изменить нельзя. Однако при перегрузке операторов вывода левый операнд должен быть потоком, а правый – объектом класса, данные которого подлежат выводу (например: `cin >> phone`). Следовательно, перегруженные операторы вывода не могут быть функциями-членами класса. В связи с этим операторные функции ввода и вывода делают «друзьями» класса. Если функция является «другом» класса, она получает легальный доступ к его `private`-данным.

Чтобы объявить функцию, дружественную данному классу, необходимо включить ее прототип в объявления класса и предварить его ключевым словом `friend`. Спецификаторы доступа к элементам `private`, `protected` и `public` не имеют отношения к объявлению дружественности, так что объявления дружественных функций могут помещаться в любом месте описания класса.

Следующая программа демонстрирует перегрузку операций «поместить в поток» и «взять из потока» для обработки данных класса телефонных номеров `PhoneNumber`. В этой программе предполагается, что номера вводятся правильно в виде: 4952 12-34-56 (то есть вначале четырехзначный код области, а затем шестизначный городской номер с дефисами между парами цифр).

```
//Перегрузка операторов << и >>
#include <iostream>
#include <conio.h>
using namespace std;

class PhoneNumber
   {friend ostream & operator << (ostream &, const PhoneNumber &);
     friend istream & operator >> (istream &, PhoneNumber &);
```

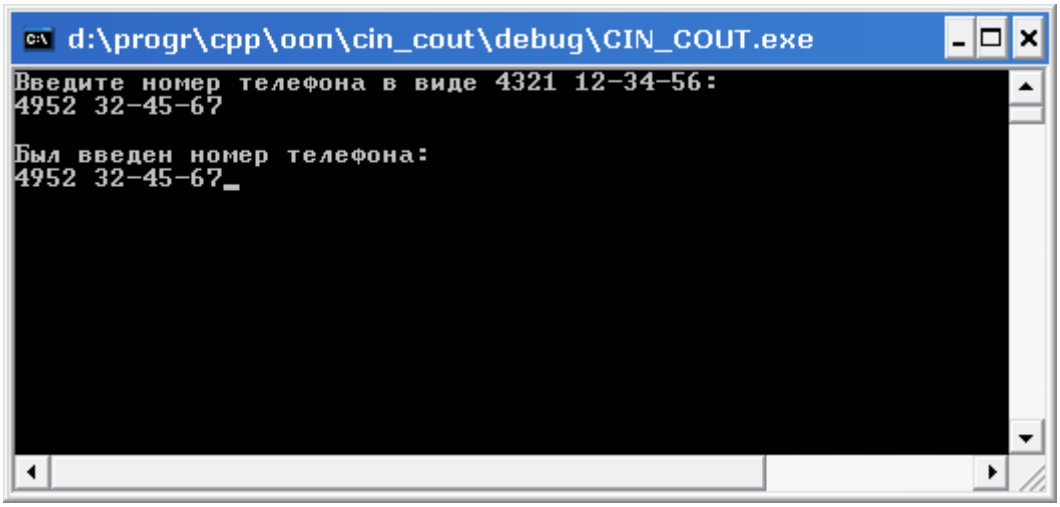
```
private:
    char areaCode[5]; //четырехцифровой код области и нулевой символ
    char cityNumber[9]; //восьмицифровой городской номер
                        //в виде 12-34-56 и нулевой символ
  };

//Перегруженная операция поместить в поток
ostream & operator << (ostream &output, const PhoneNumber &num)
   {output <<  num.areaCode << " " << num.cityNumber;
     return output; // разрешает cout << a << b << c;
     }

//Перегруженная операция взять из потока
istream & operator >> (istream &input, PhoneNumber &num)
   { input >> num.areaCode;
input >> num.cityNumber;
     return input;            // разрешает cin >> a >> b >> c;
 }

int main()
   {setlocale(LC_ALL, "Russian");
     PhoneNumber phone;             // создание объекта phone
     cout << "Введите номер телефона в виде 4321 12-34-56:\n";
     cin >> phone;        // вызвается функция operator>>
                          // путем вызова operator>> (cin, phone).
     cout << "\nБыл введен номер телефона: \n";
     cout << phone; // вызвается функция operator<<
                          // путем вызова operator<< (cout, phone).
     getch();
    return 0;  }
```

Результат работы программы:



Таким образом, функции ввода и вывода не должны быть членами класса, для обработки данных которого они предназначены. Они могут быть «друзьями» класса или просто независимыми функциями.

3. ЗАДАНИЕ НА РАБОТУ

1. Проверить работу программ, приведенных в кратких теоретических положениях.

2. Разработать самостоятельно приложения для решения задач по своему варианту.

4. ПОРЯДОК ВЫПОЛНЕНИЯ РАБОТЫ

Задание 1. Ознакомиться с теоретическим материалом, приведенным в пункте «Краткие теоретические положения» данных методических указаний, а также с конспектом лекций и рекомендуемой литературой по данной теме.

Задание 2. Разработайте программу по своему варианту. Для этого надо вначале создать h-файл с объявлением и определением класса, а затем разработать основную программу (`cpp`-файл с функцией **`main`**), в которой используется созданный класс. Программа должна содержать меню, позволяющее осуществить проверку всех методов класса.

Покажите результаты работы преподавателю. Оформите отчет по работе.

5. ОФОРМЛЕНИЕ ОТЧЕТА

Отчет по работе должен содержать:

- название и цель работы;
- номер варианта;
- файлы *.h и *.cpp, содержащие описание и реализацию методов класса в соответствии с заданием преподавателя;
- текст основной программы с комментариями;
- описание разработанных структур данных и функций,
- текст кода программы, результаты работы программы.

6. БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. Шилдт Г. C++: базовый курс, 3-е издание. : Пер. с англ. – М.: «Издательский дом «Вильямс», 2005. – 624 с.

2. Пахомов Б.И. C/C++ и MS Visual C++ для начинающих. – СПб.: БХВ-Петербург, 2008. – 624 с.

7. КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Какие типы данных называются абстрактными в C++?

2. Каким образом производиться объявление класса?

3. Какие спецификаторы доступа используются при объявлении класса?

4. Будет ли ошибка, если при создании нового типа данных не будет перегружен оператор присваивания?

5. В чем заключается инкапсуляция? Как реализуется инкапсуляция при создании абстрактных типов данных?

6. Какие правила существуют для перегрузки операций для классов?

7. Чем обусловлено введение дружественных функций в языке C++?

8. Чем отличается реализация дружественной функции от функции-элемента класса?

9. Какие правила существуют для перегрузки операций для классов?

10. Почему перегруженные функции-операции «поместить в поток» и «взять из потока» не могут быть членами класса, для обработки данных которого они предназначены?