Белорусский государственный университет

ПРАКТИКУМ
ПО КУРСУ «ПРОГРАММИРОВАНИЕ»
В ДВУХ ЧАСТЯХ
ЧАСТЬ 2. ЯЗЫК ПРОГРАММИРОВАНИЯ C++

Учебное пособие

Для студентов университетов
специальностей
«Информатика»,
«Прикладная математика»,
«Компьютерная безопасность»,
«Актуарная математика» и
«Экономическая кибернетика

Минск 2008

Авторы: С. П. Бондаренко, А. П. Побегайло

Рецензенты: Л.Ф. Зимянин, Н.А. Разоренов

В практикум включены шесть лабораторных работ по языку программирования C++. Каждая из них содержит необходимые теоретические сведения, примеры и задания для самостоятельной работы.

Приведены также индивидуальные задания, которые охватывают весь учебный материал.

Лабораторный практикум предназначен для студентов ВУЗов, специальностей «Информатика», «Прикладная математика», «Компьютерная безопасность», «Актуарная математика» и «Экономическая кибернетика.

Лабораторная работа №1

Тема. Определение классов. Спецификаторы доступа. Конструкторы и деструкторы. Друзья класса. Вложенные классы.

1.1. Определение класса

Абстрактным типом данных называется множество значений и множество операций, допустимых над этими значениями. В языке программирования C++ программист может определять свои абстрактные типы данных, используя для этого классы. Можно сказать, что класс это реализация абстрактного типа данных в языке программирования C++. Множество значений данных, принадлежащих классу определяется структурой, а множество операций, допустимых над этими значениями, определяется набором функций. Как данные, так и функции, определенные в классе, называются членами класса. Данные класса также называются его атрибутами, а функции – методами. Набор функций класса называется его интерфейсом.

Синтаксически, класс определяется следующим образом:

```
class|struct|union имя_класса
{
  // члены класса
};
```

При определении класса должно использоваться только одно из ключевых слов `class`, `struct` или `union`. Каждое из этих ключевых слов устанавливает различные режимы доступа к членам класса. При использовании ключевого слова `class` доступ ко всем членам класса закрыт, а при использовании ключевых слов `struct` и `union` – открыт. Чтобы открыть или закрыть доступ клиентов к определенным членам класса используются спецификаторы доступа, которые будут рассмотрены в дальнейшем.

Например, определим интерфейс контейнерного класса для хранения одного целого числа.

```
struct IntBox
{
  // атрибуты класса
  bool empty;  // состояние контейнера
  int  room;   // место для хранения целого числа
  // методы класса
```

```
  void put(int n);  // положить целое число в контейнер
  int  get();       // извлечь целое число из контейнера
  bool isEmpty();   // проверить, свободен ли контейнер
};
```

Как правило, интерфейс класса помещается в отдельный заголовочный файл, который называется также как и класс. В нашем случае интерфейс класса нужно поместить в заголовочный файл `IntBox.h`

1.2. Определение функций-членов класса

Функции-члены класса определяются как обычные функции. Но, для того чтобы показать, что эти функции принадлежат классу, нужно перед именем функции записать имя класса и оператор разрешения области видимости.

Определение класса вводит область видимости, которой принадлежат имена членов класса. Определение функции-члена класса также принадлежит области видимости класса. Отсюда следует, что в теле функции-члене класса можно обращаться к другим членам этого класса просто по имени. Например, определим функции, члены класса `IntBox`:

```
void IntBox::put(int n)
{
  room = n;
  empty = false;
}

int  IntBox::get()
{
  empty = true;
  return room;
}

bool IntBox::isEmpty()
{
  return empty;
}
```

Как правило реализация функций также помещается в отдельный файл, назвать который можно IntBox.cpp

Функция, член класса, также может быть определена внутри класса. В этом случае функция, член класса, считается встроенной. Напри-

мер, функцию `isEmpty` можно было бы определить внутри класса следующим образом:

```
struct IntBox
{
  . . .
  bool isEmpty() { return empty; }
};
```

1.3. Доступ к членам класса

Как говорилось в предыдущем параграфе, доступ к членам класса внутри области видимости этого класса выполняется по их именам. При доступе к членам класса вне области видимости этого класса используются операторы `(.)` и (->), как и в случае доступа к членам структуры.

К любому члену класса, принадлежащему текущему объекту, можно также обратиться через указатель `this`. Например, для класса `IntBox` можно написать

```
  this -> empty = false;
```

Но, так как функции-члены класса имеют доступ ко всем членам класса по умолчанию, то указатель `this` используется в основном для возврата указателя или ссылки на объект, к которому применяется функция-член класса.

После определения класса можно объявлять переменные, которые имеют тип этого класса, или другими словами принадлежат этому классу или являются экземплярами этого класса. В языке программирования C++ экземпляры класса называются объектами. Над объектами можно выполнять функции, определенные в классе. Например, в следующем листинге показано, как работать с объектами типа `IntBox`.

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

using namespace std;

int main()
{
```

```
  IntBox box = {true};  // объявление объекта типа IntBox
  int n;

  if (box.isEmpty())
    box.put(10);
  if (!box.isEmpty())
    n = box.get();

  cout << n << endl;   // печатает 10

  return 0;
}
```

Здесь предполагается, что интерфейс класса `IntBox` определен в заголовочном файле `IntBox.h`.

1.4. Спецификаторы доступа

Для ограничения доступа к членам класса вне его области видимости используются спецификаторы доступа: `public`, `protected` и `private`, которые также называются метками и могут использоваться внутри класса произвольное число раз и в любом порядке.

Все члены класса, объявленные после метки `public` и расположенные до следующей метки, доступны как в классе, так и вне этого класса.

Все члены класса, объявленные после метки `protected` и расположенные до следующей метки, доступны в классе и в наследниках этого класса.

Все члены класса, объявленные после метки `private` и расположенные до следующей метки, доступны только внутри класса.

По умолчанию все члены класса, объявленного с помощью ключевого слова `class`, имеют спецификацию доступа `private`, а объявленные с помощью ключевых слов `structure` или `union` – спецификацию доступа `public`.

В нашем примере все члены класса IntBox являются открытыми. Чтобы запретить пользователю класса IntBox прямой доступ к атрибутам этого класса, класс `IntBox` нужно объявить следующим образом:

```
  class IntBox
  {
```

```
  bool empty;  // состояние контейнера
  int  room;   // место для хранения целого числа
public:
  IntBox();          // конструктор
  void put(int n);  // положить целое число в контейнер
  int  get();        // взять целое число из контейнера
  bool isEmpty();    // проверить, свободно ли место
};
```

Но в этом случае в классе должна быть определена функция для инициализации объектов этого класса. Такая функция называется конструктором и имеет такое же имя, как и сам класс. В нашем случае это функция `IntBox`, которая имеет следующее определение:

```
IntBox::IntBox() { empty = true; }
```

Более подробно конструкторы классов рассмотрены в следующем параграфе.

1.5. Конструкторы

Конструктором класса называется функция-член класса, имя которой совпадает с именем этого класса. Конструктор класса всегда вызывается компилятором при создании объекта этого класса. Под созданием объекта будем подразумевать распределение памяти под объект и инициализация данных-членов класса, принадлежащих этому объекту. Отсюда следует, что конструкторы должны использоваться для инициализации данных-членов класса. Значения, которыми инициализируются данные-члены класса, передаются через параметры конструктора.

Конструктор, который не имеет параметров или все параметры которого имеют значения, заданные по умолчанию, называется конструктором по умолчанию. Класс может иметь только один конструктор по умолчанию. Пример конструктора по умолчанию для класса `IntBox` был приведен в предыдущем параграфе.

Если класс совсем не имеет конструкторов, то компилятор сам генерирует конструктор по умолчанию, который имеет следующий прототип:

```
public: inline имя_класса();
```

Конструктором, первый параметр которого является константной ссылкой на объект класса, которому принадлежит конструктор, а остальные параметры либо отсутствуют, либо имеют значения, заданные по умолчанию, называется конструктором копирования. Конструктор копирования используется компилятором для инициализации объекта другим объектом этого же класса.

Если конструктор копирования не определен, то компилятор сам генерирует конструктор копирования. Этот конструктор имеет следующий прототип:

```
public: inline имя_класса(const имя_класса&);
```

и выполняет по членное копирование данных-членов из объекта аргумента конструктора в конструируемый объект. Так как при копировании объекта типа `IntBox` нужно выполнить по членное копирование данных-членов класса, то определять конструктор копирования в классе `IntBox` не обязательно.

1.6. Деструкторы

Деструктор это функция-член класса, имя которой совпадает с именем класса, но перед ним дополнительно ставится символ `'~'` (тильда). Деструктор класса не имеет параметров.

Деструктор автоматически вызывается компилятором перед удалением объекта класса из памяти. Поэтому деструкторы используются для освобождения ресурсов, захваченных во время существования объекта. Например, если во время своего существования объект захватывал память при помощи оператора `new`, то эту память можно освободить в деструкторе при помощи оператора `delete`.

Если ресурсы освобождать не нужно, то деструктор не пишется. В этом случае компилятор генерирует деструктор по умолчанию, который выполняет следующие действия: сначала вызывает деструкторы данных-членов класса, а затем вызывает деструкторы базовых классов.

В нашем случае объекты класса `IntBox` ресурсы не захватывают, поэтому деструктор для этого класса не нужен.

1.7. Друзья класса

Доступ к закрытым членам класса из другого класса или функции допускается только в том случае, если они объявлены как друзья класса. Объявление друга класса начинается с ключевого слова `friend` и может встречаться только внутри определения класса.

Определяются функции-друзья класса как обычные функции внутри или вне класса. Если функция, друг класса, определяется вне класса, то ключевое слово `friend` не используется.

Например, объявим для класса `IntBox` две дружественные функции `in` и `out`, которые соответственно предназначены для ввода целого числа в контейнер с консоли и для вывода целого числа из контейнера на консоль. Тогда определение класса `IntBox` будет выглядеть следующим образом:

```
class IntBox
{
    bool empty;  // состояние контейнера
    int  room;   // место для хранения целого числа
  public:
    IntBox();          // конструктор
    void put(int n);  // положить целое число в контейнер
    int  get();        // взять целое число из контейнера
    bool isEmpty();   // проверить, свободно ли место
    friend void in(IntBox& box);   // ввод числа с консоли
    friend void out(IntBox& box);  // вывод числа на консоль
};
```

Эти функции могут быть реализованы вне класса следующим образом.

```
void in(IntBox& box)
{
  cin >> box.room;
  box.empty = false;
}

void out(IntBox& box)
{
  cout << box.room;
  box.empty = true;
```

```
}
```

1.8. Вложенные классы

Класс, объявленный внутри другого класса, называется вложенным классом. Функции-члены класса, в котором объявлен вложенный класс, не имеют прав доступа к закрытым членам этого вложенного класса. Также и функции-члены вложенного класса не имеет прав доступа к закрытым членам класса, внутри которого он объявлен. Чтобы предоставить такие права вложенному или объемлющему классу, нужно объявить такой класс другом соответственно объемлющего или вложенного класса.

1.9. Задачи для самостоятельного решения

1. Реализовать класс `ArrayStack`, который имеет следующий интерфейс:

```
class ArrayStack  // стек на массиве
{
  int  size;     // размерность массива
  int* p;        // указатель на массив
  int  top;      // верхушка стека
public:
  // конструкторы
  ArrayStack(const int& _size);
  ArrayStack(const ArrayStack& s);  // копирования
  // деструктор
  ~ArrayStack();
  // функции-члены класса
  void push(const int& n);  // втолкнуть элемент в стек
  int pop();                // вытолкнуть элемент из стека
  bool isEmpty();           // пустой стек?
  bool isFull();            // полный стек?
};
```

Определение класса поместить в заголовочный файл `stack.h`, а реализацию функций-членов класса – в файл `stack.cpp`.

Для проверки работоспособности класса `ArrayStack` написать тестирующую программу, в которой вызываются все методы класса.

2. Реализовать класс `ListStack`, который имеет следующий интерфейс:

```
class ListStack  // стек на списке
{
  // вложенный класс
  struct node  // элемент списка
  {
    int item;    // данные
    node*  p;    // указатель на следующий элемент
  };
  node *head;    // указатель на первый элемент в списке
public:
  // конструкторы
  ListStack();                        // по умолчанию
  ListStack(const ListStack& lst);  // копирования
  ~ListStack();           // деструктор
  void push(const int& n);// включить элемент в голову списка
  int pop();              // удалить элемент из головы списка
  bool isEmpty();         // пустой список?
};
```

Определение класса поместить в заголовочный файл `list.h`, а реализацию функций-членов класса – в файл `list.cpp`.

Для проверки работоспособности класса `ListStack` написать тестирующую программу, в которой вызываются все методы класса.

3. Реализовать класс `Student`, который имеет следующий интерфейс:

```
class Student
{
  char name[10];  // фамилия студента
  int  num;       // номер группы
  double grade;   // средний бал
public:
  // конструкторы
  Student();      // конструктор по умолчанию
  Student(const char* _name, const int& _num,
          const double& grade);
  void setName(const char* _name);
  void getName(char* _name);
  bool changeName(const char* oldName, const char* newName);
  void setNum(const int& _num);
  int getNum();
  bool changeNum(const int& oldNum, const int& newNum);
  void setGrade(const double& _grade);
  double getGrade();
```

```
  bool changeGrade(const double& oldGrade,
                   const double& newGrade);
  // функции для ввод-вывода данных в объект типа Student
  void input(istream& in);
  void output(ostream& out);
};
```

Определение класса поместить в заголовочный файл `student.h`, а реализацию функций-членов класса – в файл `student.cpp`.

Для проверки работоспособности класса `Student` написать тестирующую программу, в которой вызываются все методы класса.

1.10. Дополнительные задачи

1. Реализовать класс `ArrayQueue`, который имеет следующий интерфейс:

```
class ArrayQueue    // кольцевая очередь на массиве
{
  int  size;     // размерность массива
  int* p;        // указатель на массив
  int  head;     // индекс первого занятого элемента
  int  n;        // количество элементов в очереди
public:
  ArrayQueue(const int& _size);    // инициализация очереди
  ArrayQueue(const ArrayQueue& q); // конструктор копирования
  ~ArrayQueue();                   // деструктор очереди
  void ins(const int& n);  // включить элемент в очередь
  int del();               // удалить элемент из очереди
  bool isEmpty();          // очередь пустая?
  bool isFull();           // очередь полная?
};
```

Определение класса поместить в заголовочный файл `queue.h`, а реализацию функций-членов класса – в файл `queue.cpp`.

Для проверки работоспособности класса `ArrayQueue` написать тестирующую программу, в которой вызываются все методы класса.

2. Реализовать класс `ListQueue`, который имеет следующий интерфейс:

```
class ListQueue  // очередь на списке
{
  // вложенный класс
```

```
  struct node  // элемент списка
  {
    int item;    // данные
    node*  p;    // указатель на следующий элемент
  };
  node *head;    // указатель на первый элемент в списке
public:
  // конструкторы
  ListQueue();                        // по умолчанию
  ListQueue(const ListQueue& lst);  // копирования
  ~ListQueue();                    // деструктор
  void ins(const int& n);  // включить элемент в хвост списка
  int del();              // удалить элемент из головы списка
  bool isEmpty();                 // пустой список?
};
```

Определение класса поместить в заголовочный файл `list.h`, а реализацию функций-членов класса – в файл `list.cpp`.

Для проверки работоспособности класса `ListQueue` написать тестирующую программу, в которой вызываются все методы класса.

3. Реализовать класс `StudentContainer`, интерфейс которого приведен ниже. Элементы контейнера имеют тип `Student`, который был определен в задаче №3 из раздела 1.9.

```
class StudContainer
{
  int size;    // размерность массива
  Student* p;  // массив записей о студентах
  int count;   // количество студентов в контейнере
public:
  // конструкторы
  StudContainer(const int& _n);
  StudContainer(const StudContainer& c);  // копирования
  ~StudContainer();                       // деструктор
  bool ins(const Student& s);  // включение в контейнер
  bool del(char* name);        // исключить из контейнера
  void sortByName();    // сортировка студентов по именам
  Student* findByName(char* name); // поиск студента по имени
  // сортировка по группам, а внутри групп - именам
  void sortByGroupAndName();
  void report(ostream& out);   // вывести отчет в файл
  int size();    // возвращает размер контейнера
  int count();   // возвращает количество студентов
};
```

Определение класса поместить в заголовочный файл `container.h`, а реализацию функций-членов класса – в файл `container.cpp`.

Для проверки работоспособности класса `StudContainer` написать тестирующую программу, в которой вызываются все методы класса.

1.11. Задачи для индивидуальной работы

1. Реализовать класс `ArraryDeque`, который имеет следующий интерфейс:

```
class ArrayDeque   // дек – двустороння очередь на массиве
{
  int  size;    // размерность массива
  int* p;       // указатель на массив
  int  head;    // индекс первого занятого элемента
  int  n;       // количество элементов в очереди
public:
  ArrayDeque();                     // инициализация
  ArrayDeque(const ArrayDeque& d);  // копирование
  ~ArrayDeque();                    // деструктор
  void pop(const int& n);  // включить элемент в голову дека
  void ins(const int& n);  // включить элемент в хвост дека
  int push();      // исключить элемент из головы дека
  int del();       // исключить элемент из хвоста дека
  bool isEmpty();             // дек пустой?
  bool isFull();              // дек полный?
};
```

Определение класса поместить в заголовочный файл `deque.h`, а реализацию функций-членов класса – в файл `deque.cpp`.

Для проверки работоспособности класса `ArrayDeque` написать тестирующую программу, в которой вызываются все методы класса.

2. Реализовать класс `ListDeque`, который имеет следующий интерфейс:

```
class ListDeque   // дек – двустороння очередь на списке
{
  struct node     // элемент дека
  {
    int  item;    // данные
    node*  next;  // указатель на следующий элемент
    node*  prev;  // указатель на предыдущий элемент
  };
```

```
  // указатели на первый и последний элементы в списке
  struct node *head, *tail;
public:
  // конструкторы
  ListDeque();                    // по умолчанию
  ListDeque(const ListDeque& d);  // копирования
  ~ListDeque();             // деструктор
  void pop(const int& n);  // включить элемент в голову дека
  void ins(const int& n);  // включить элемент в хвост дека
  int push();       // исключить элемент из головы дека
  int del();        // исключить элемент из хвоста дека
  bool isEmpty();           // дек пустой?
};
```

Определение класса поместить в заголовочный файл `deque.h`, а реализацию функций-членов класса – в файл `deque.cpp`.

Для проверки работоспособности класса `ListDeque` написать тестирующую программу, в которой вызываются все методы класса.

Лабораторная работа №2

Тема. Перегрузка функций и операторов.

2.1. Перегрузка функций

В языке C++ разрешается определять функции, которые имеют одинаковое имена, но разные сигнатуры. Напомним, что сигнатурой функции называется список типов параметров этой функции. Такие функции называются перегруженными. Например, перегрузим функцию вычисления абсолютной величины числа:

```
int abs(int n);
double abs(double d);
```

Нельзя перегружать функции, которые отличаются только типом возвращаемого значения. Перегрузка функций упрощает выбор имени для функций, которые реализуют схожие вычисления.

2.2. Перегрузка операторов

В языке программирования C++ операторы рассматриваются как функции, которые имеют следующий синтаксис:

```
тип operator имя_оператора(список_параметров)
```

где `operator` – ключевое слово. Поэтому операторы также как и функции можно перегружать. Перегружаются почти все операторы. Нельзя перегружать следующие операторы:

- оператор доступа к члену класса `(.)`,
- оператор доступа к члену класса через указатель `(.*)`,
- оператор разрешения области видимости `(::)`,
- условный оператор `(:?)`,
- операторы преобразования типов данных: `static_cast`, `const_cast`, `reinterpret_cast`, `dynamic_cast`.

Правила перегрузки различных операторов будем рассматривать на примере класса `even`, который содержит четные числа.

```
class even
{
```

```
  int n;
public:
  // конструктор
  even( );               // по умолчанию
  even(const int& _n);   // копирования
  // операторы присваивания
  even& operator+=(const even& e);
  even& operator-=(const even& e);
  even& operator*=(const even& e);
  // унарный + и –
  friend even operator-(const even& e);
  friend even operator+(const even& e);
  // бинарные операторы сравнения
  friend bool operator<(const even& e1, const even& e2);
  friend bool operator>(const even& e1, const even& e2);
  friend bool operator==(const even& e1, const even& e2);
  friend bool operator!=(const even& e1, const even& e2);
  friend bool operator<=(const even& e1, const even& e2);
  friend bool operator>=(const even& e1, const even& e2);
  // бинарные арифметические операторы
  friend even operator+(const even& e1, const even& e2);
  friend even operator-(const even& e1, const even& e2);
  friend even operator*(const even& e1, const even& e2);
  // бинарные операторы ввода-вывода
  friend istream& operator>>(istream& in, even& e);
  friend ostream& operator<<(ostream& out, const even& e);
};
```

Операторы могут классифицироваться по количеству операндов и местоположению оператора относительно операндов.

Если оператор имеет один операнд, то он называется унарным. Если оператор имеет два операнда, то он называется бинарным и т.д.

Если оператор пишется перед операндами, то он называется префиксным. Если оператор пишется между операндами, то он называется инфиксным. Если оператор пишется после операндов, то он называется постфиксным.

2.3. Перегрузка унарных операторов

Префиксные унарные операторы могут быть перегружены как нестатические члены класса без параметров или как операторы не члены класса с одним параметром. Выбирается тот вариант, который лучше подходит для семантики оператора. Можно оставить два определения префиксного унарного оператора: как члена и как друга класса. В этом

случае при вызове унарного оператора компилятор выбирает тот вариант, для которого преобразования типа считается наилучшим.

Постфиксные унарные операторы могут быть перегружены только как не статические члены класса.

В нашем примере перегружаются унарные и бинарные арифметические операторы, а также бинарные операторы сравнения и ввода-вывода. Например, перегрузим унарный оператор сложения для класса `even`.

```
even& even::operator+=(const even& e)
{
  n += e.n;

  return  *this;
}
```

Заметим, что унарные операторы сложения, вычитания и умножения возвращают ссылку на объект типа `even`, так как в этом случае возвращается сам модифицированный объект, для которого вызывается эта операция. Здесь ключевое слово `this` обозначает указатель на объект класса, к которому применяется функция-оператор. Можно рассматривать `this` как неявный параметр этой функции. Запись `*this` обозначает разыменование этого указателя, поэтому оператор `(+=)` и возвращает объект класса, к которому применяется оператор.

При перегрузке операторов инкремента и декремента следует учитывать, что для отличия префиксных операторов "`++`" и "`--`" от соответствующих постфиксных операторов, в объявлении последних вводят дополнительный фиктивный параметр типа `int`. Например,

```
SomeClass& operator ++()     // префиксный ++
SomeClass operator ++(int)   // постфиксный ++
```

Здесь постфиксный оператор должен возвращать новый объект, который равен объекту, к которому применяется оператор, до изменения этого объекта самим оператором инкремента.

2.4. Перегрузка оператора присваивания

Оператор присваивания может быть перегружен только как нестатический член класса. Перегруженный оператор присваивания должен иметь следующий прототип:

```
имя_класса& operator =(const имя_класса&);
```

При реализации оператора присваивания должна проверяться возможность присваивания объекта самому себе.

Если оператор присваивания не определен в классе, то компилятор генерирует оператор присваивания по умолчанию, который выполняет по членное копирование атрибутов класса. Если при присваивании объектов класса дополнительно к копированию атрибутов требуется выполнить ещё какие-то действия, то оператор присваивания нужно обязательно перегрузить.

Присваивание четных чисел включает только по членное копирование атрибутов класса `even`, поэтому в этом случае перегружать оператор присваивания не нужно.

Аналогично перегружаются составные операторы присваивания, но в этом случае объект, к которому применяется оператор, всегда модифицируется.

2.5. Перегрузка бинарных операторов

Бинарные операторы могут быть перегружены как нестатические члены класса с одним параметром или как операторы не члены класса с двумя параметрами.

При перегрузке бинарных операторов следует учитывать симметричность их параметров. Если аргументами бинарного оператора являются объекты одного класса, то этот оператор лучше перегружать как дружественный оператор с двумя параметрами. Например, перегрузим бинарный оператор сложения для класса `even`.

```
even operator+(const even& e1, const even& e2)
{
  return even(e1.n + e2.n);
}
```

Отметим, что в этом случае бинарный оператор возвращает новый объект типа `even`, поэтому возвращается не ссылка на объект, а сам объект, который в дальнейшем копируется в переменную, объявленную в вызывающей функции. Другими словами перегруженный бинарный оператор `'+'` возвращает объект по значению, а для копирования объекта в переменную вызывается конструктор копирования.

Операторы ввода-вывода перегружаются как друзья класса. Например, перегрузим оператор для вывода четного числа:

```
ostream& operator<<(ostream& out, const even& e)
{
  return  out << e.n;
}
```

2.6. Задачи для самостоятельного решения

1. Реализовать класс трехмерных векторов, который имеет следующий интерфейс:

```
class Vector
{
  double x, y, z;    // координаты
public:
  // конструкторы
  Vector() {}        // по умолчанию
  Vector(const double& _x, const double& _y,
         const double& _z); // по элементам
  // операторы-члены
  Vector& operator +=(const Vector&);
  Vector& operator -=(const Vector&);
  Vector& operator *=(const double&);
  // операторы-друзья
  friend Vector operator +(const Vector&, const Vector&);
  friend Vector operator -(const Vector&, const Vector&);
  friend Vector operator *(const double&, const Vector&);
  friend Vector operator *(const Vector&, const double&);
  // скалярное произведение
  friend double operator *(const Vector&, const Vector&);
  // длина вектора
  friend double length(const Vector&);
  // угол между двумя векторами
  friend double angle(const Vector&, const Vector&);
  // векторное произведение двух векторов
  friend Vector operator ^(const Vector&, const Vector&);
  // координаты вектора
  friend double x(const Vector&);
  friend double y(const Vector&);
  friend double z(const Vector&);
  // операторы ввода-вывода
  friend istream& operator >>(istream&, Vector&);
  friend ostream& operator <<(ostream&, const Vector&);
};
```

Определение класса поместить в заголовочный файл `vector.h`, а реализацию функций-членов класса – в файл `vector.cpp`.

Для проверки работоспособности класса `Vector` написать тестирующую программу, в которой вызываются все методы класса.

2. Реализовать класс конечное множество целых чисел, который имеет следующий интерфейс:

```
class Set
{
  int size;     // размер множества
  int count;    // количество элементов в множестве
  int* p;       // указатель на массив элементов
public:
  // конструкторы
  Set(const int& _size = 0);  // по умолчанию
  Set(const Set& s);          // копирования
  // деструктор
  ~Set();
  // функции члены класса
  int getSize() const;   // размерность множества
  int getCount() const;  // количество элементов в множестве
  void include(const int& n);  // включить элемент
  bool exclude(const int& n);  // исключить элемент
  void empty();                // очистка множества
  bool isInSet(const int& n) const;  // есть такой элемент?
  // операторы члены класса
  Set& operator =(const Set& s);   // присваивание множеств
  Set& operator +=(const Set& s);  // объединение множеств
  Set& operator -=(const Set& s);  // разность множеств
  Set& operator *=(const Set& s);  // пересечение множеств
  // операторы друзья класса
  // объединение множеств
  friend Set operator +(const Set& s1, const Set& s2);
  // разность множеств
  friend Set operator -(const Set& s1, const Set& s2);
  // пересечение множеств
  friend Set operator *(const Set& s1, const Set& s2);
  // операторы ввода-вывода
  // ввод элемента в множество из потока
  friend istream& operator >>(istream& in, Set& s);
  // вывод множества в поток
  friend ostream& operator <<(ostream& out, const Set& s);
};
```

Определение класса поместить в заголовочный файл `set.h`, а реализацию функций-членов класса – в файл `set.cpp`.

Для проверки работоспособности класса `Set` написать тестирующую программу, в которой вызываются все методы класса.

3. Реализовать класс полином, который имеет следующий интерфейс:

```
class Polynom
{
  int  n;          // степень полинома
  double*  coeff;  // коэффициенты полинома
public:
  // конструкторы
  Polynom();                    // по умолчанию
  Polynom(const int& n, ...);  // по элементам
  Polynom(const Polynom& p);   // копирования
  // деструктор
  ~Polynom();
  // вычисление значения полинома
  double Value(const double& x);
  // операторы присваивания
  Polynom& operator =(const Polynom& p);
  Polynom& operator +=(const Polynom& p);
  Polynom& operator -=(const Polynom& p);
  Polynom& operator *=(const Polynom& p);
  Polynom& operator /=(const Polynom& p);
  // унарный минус и плюс
  friend Polynom operator -( const Polynom& p);
  friend Polynom operator +( const Polynom& p);
  // бинарные операторы
  friend Polynom operator +(const Polynom&, const Polynom&);
  friend Polynom operator -(const Polynom&, const Polynom&);
  friend Polynom operator *(const Polynom&, const Polynom&);
  friend Polynom operator /(const Polynom&, const Polynom&);
};
```

Определение класса поместить в заголовочный файл `polynom.h`, а реализацию функций-членов класса – в файл `polynom.cpp`.

Для проверки работоспособности класса `Polynom` написать тестирующую программу, в которой вызываются все методы класса.

2.7. Дополнительные задачи

1. Разработать класс для описания отрезков, расположенных на целочисленной оси X (оси Y). Объекты класса имеют следующие атрибуты:

- координаты двух точек отрезка – начальной и конечной.

Обеспечить выполнение следующих операций над объектами класса:

- присваивание одного объекта другому;
- получение значения длины отрезка для заданного объекта класса;
- получение абсолютных значений координат начальной и конечной точек объекта;
- ввод-вывод объектов класса в поток;
- сравнение длин отрезков двух объектов класса с использованием для этих целей перегруженных операторов `==`, `!=`, `<`, `<=`, `>`, `>=`;
- увеличение или уменьшение значений координат начальной или концевой точек отрезка на единицу с помощью перегруженных операторов ++ (инкремент) и – (декремент);
- выполнение над двумя объектами класса операций объединения, пересечения и вычитания с использованием для этих целей перегруженных операторов `+`, `*` и `-`.

Для проверки работоспособности класса написать тестирующую программу, в которой вызываются все методы класса.

2. Разработать класс `Point` для работы с точками на плоскости (в пространстве). Объекты класса имеют следующие атрибуты:

- координаты точки.

Обеспечить выполнение следующих действий над объектами класса:

- присваивание одного объекта другому;
- перемещение точки по осям координат на указанную величину;
- определение расстояния от точки до начала координат;
- сравнение точек на совпадение и несовпадение;
- вычисление расстояния между двумя точками;
- сравнение длин радиус-векторов двух точек;
- получение новой точки, симметричной рассматриваемой точке относительно начала координат;
- ввод-вывод объектов класса в поток с использованием для этих целей перегруженных операторов << и >>.

Для проверки работоспособности класса написать тестирующую программу, в которой вызываются все методы класса.

3. Разработать класс бинарное дерево `BynaryTree`. Интерфейс класса разработать самостоятельно.

Для проверки работоспособности класса написать тестирующую программу, в которой вызываются все методы класса.

2.8. Задачи для индивидуальной работы

1. Разработать класс векторов двумерного векторного пространства. Объекты класса имеют следующие атрибуты:

- координаты вектора.

Обеспечить выполнение следующих действий над объектами класса:

- присваивание одного объекта другому;
- ввод-вывод объектов класса с использованием для этих целей перегруженных операторов << и >>;
- определение длины вектора;
- сложение и вычитание двух векторов;
- умножение вектора на скаляр;
- получение скалярного произведения двух векторов;
- сравнение двух векторов на равенство;
- сравнение значений длин двух векторов.

Для проверки работоспособности класса написать тестирующую программу, в которой вызываются все методы класса.

2. Разработать класс для описания отрезков, расположенных на целочисленной плоскости. Объекты класса имеют следующие атрибуты:

- координаты начальной и конечной точек отрезка.

Обеспечить выполнение следующих действий над объектами класса:

- присваивание одного объекта другому;
- получение значения длины отрезка;
- получение координат начальной и конечной точек отрезка;
- ввод-вывод объектов класса в поток;
- сравнение длин двух отрезков с использованием для этих целей перегруженных операторов `==, !=, <, <=, >, >=`;
- увеличение или уменьшение значений координат начальной или конечной точек отрезка на единицу с помощью перегруженных операторов ++ (инкремент) и −− (декремент);

- получение нового отрезка, симметричного заданному отрезку относительно центра системы координат;
- определение взаимного расположения двух отрезков – совпадение, пересечение или параллельность.

Для проверки работоспособности класса написать тестирующую программу, в которой вызываются все методы класса.

Лабораторная работа №3

Тема. Обработка исключений.

3.1. Исключения.

Синтаксически исключение (exception) – это объект произвольного типа, а обработка исключений – это механизм для передачи управления и исключения в специальный блок, который называется обработчиком исключения. Механизм обработки исключения состоит из четырех компонент: исключения, throw -выражения, try-блока и catch-блока, которые организованы следующим образом:

```
try
{
  . . .
  throw  исключение;
  . . .
}
catch(тип_исключения)
{
  . . .
}
```

где инструкция throw выбрасывает исключение; блок try содержит код, который может выбросить исключение; блок catch описывает обработчик исключения и должен следовать непосредственно за блоком try. Допускается использование нескольких последовательных блоков catch. Если внутри блока try произошел выброс исключения, то управление передается первому обработчику исключения, тип которого соответствует типу выброшенного исключения. После обработки исключения управление передается на первую инструкцию, которая следует за последним обработчиком исключения. Например,

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

int main()
{
  int n, m;

  cout << "Input two integers: ";
  cin >> n >> m;
```

```
  try
  {
    if (!m)
      throw "Zero divide";
    cout << "n/m = " << (n/m) << endl;
  }
  catch(char* str)
  {
    cout << str << endl;
  }
  cout << "OK" << endl;

  return 0;
}
```

Часто вся информация об исключении определяется только его типом. В этом случае для типа исключения используют пустые структуры. Например, в нашем случае можно было бы определить исключение типа `Zero_divide`. Тогда наша программа выглядела бы следующим образом.

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

struct Zero_divide {};

int main()
{
  int n, m;

  cout << "Input two integers: ";
  cin >> n >> m;

  try
  {
    if (!m)
      throw Zero_divide();
    cout << "n/m = " << (n/m) << endl;
  }
  catch(Zero_divide)
  {
    cout << "Zero divide" << endl;
  }
  cout << "OK" << endl;

  return 0;
```

```
}
```

Обратим внимание на то, что оператор throw должен выбросить объект типа Zero_divide, а не сам тип. Поэтому в этом операторе используется конструктор по умолчанию для объекта типа Zero_divide.

3.2. Обработка нескольких исключений

Для обработки исключений разных типов допускается использование нескольких последовательных блоков `catch`. Если внутри блока `try` произошел выброс исключения, то управление передается первому обработчику исключения, тип которого соответствует типу выброшенного исключения. Если такого обработчика нет, то вызывается функция `terminate`, которая в свою очередь вызывает функцию `abort`, которая и завершает исполнение программы. С помощью стандартной функции `set_terminate` можно вместо функции `terminate` установить свою функцию для обработки ситуации, возникающей при отсутствии подходящего обработчика для исключения. Прототип функции `set_terminate` описан в заголовочном файле `eh.h`.

3.3. Перехват всех исключений

Для перехвата исключений любого типа используется обработчик исключения, который имеет следующий вид:

```
catch(...)
{
  . . .
}
```

Очевидно, что если такой обработчик исключений используется, то он должен быть последним в последовательности обработчиков исключений. Иначе, он будет перехватывать все исключения. В следующем листинге приведен пример перехвата исключений разных типов.

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

struct Zero {};
struct NonZero {};

int main()
```

```
{
  int n;

  cout << "Input integer: ";
  cin >> n ;

  try
  {
    if (n)
      throw NonZero();
    throw Zero();
  }
  catch(...)
  {
    cout << "Zero or NonZero exception" << endl;
  }
  cout << "OK" << endl;

  return 0;
}
```

3.4. Задачи для самостоятельного решения

1. Реализовать класс рациональных чисел, который имеет следующий интерфейс:

```
// исключения
struct ZeroDivide {};

// интерфейс класса рациональных чисел
class ratio
{
  long n, d;      // числитель и знаменатель
  void reduce();  // сокращение дроби
public:
  // конструкторы
  ratio();                // по умолчанию
  ratio(long n, long d);  // по элементам
  ratio(const ratio& r);  // копирования
  // операторы присваивания
  ratio& operator =(const ratio& r);
  ratio& operator +=(const ratio& r);
  ratio& operator -=(const ratio& r);
  ratio& operator *=(const ratio& r);
  ratio& operator /=(const ratio& r) throw(ZeroDivide);
  // оператор преобразования типа
  operator double(void) const;
```

```
  // унарный минус и плюс
  friend ratio operator -(const ratio& r);
  friend ratio operator +(const ratio& r);
  // операторы сравнения
  friend bool operator <(const ratio& r1, const ratio& r2);
  friend bool operator >(const ratio& r1, const ratio& r2);
  friend bool operator ==(const ratio& r1, const ratio& r2);
  friend bool operator !=(const ratio& r1, const ratio& r2);
  friend bool operator <=(const ratio& r1, const ratio& r2);
  friend bool operator >=(const ratio& r1, const ratio& r2);
  // бинарные операторы
  friend ratio operator +(const ratio& r1, const ratio& r2);
  friend ratio operator -(const ratio& r1, const ratio& r2);
  friend ratio operator *(const ratio& r1, const ratio& r2);
  friend ratio operator /(const ratio& r1, const ratio& r2)
                                           throw(ZeroDivide);
  // операторы ввода-вывода
  friend istream& operator>>(istream& in, ratio& r);
  friend ostream& operator<<(ostream& out, const ratio& r);
};
```

Для проверки работоспособности класса написать тестирующую программу, в которой вызываются все методы класса.

2. Реализовать класс матриц, который имеет следующий интерфейс:

```
// исключения
struct BadDimension {};  // ошибка в размерности матрицы

// интерфейс класса матрица
class matrix
{
  long  n, m;  // размерности матрицы
  double*  p;  // указатель на матрицу
  matrix();    // по умолчанию матрицу создавать нельзя
public:
  // конструкторы
  matrix(long _n, long _m) throw(BadDimensions);
  matrix(const matrix& m);
  // деструктор
  ~matrix();
  // операторы присваивания
  matrix& operator =(const matrix& m) throw (BadDimension);
  matrix& operator +=(const matrix& m) throw (BadDimension);
  matrix& operator -=(const matrix& m) throw (BadDimension);
  matrix& operator *=(const matrix& m) throw (BadDimension);
```

```
  matrix& operator *=(const double& d);
  // оператор индексирования возвращает указатель на строку
  double* operator [] (const long& i);
  const double* operator [](const long& i) const;
  // унарные операторы
  friend matrix operator -(const matrix& m);
  friend matrix operator +(const matrix& m);
  // бинарные операторы
  friend matrix operator *(const double& d, const matrix& m);
  friend matrix operator *(const matrix& m, const double& d);
  friend matrix operator + (const matrix& m1,
                    const matrix& m2) throw(BadDimension)
  friend matrix operator -(const matrix& m1,
                    const matrix& m2) throw(BadDimension)
  friend matrix operator *(const matrix& m1,
                    const matrix& m2) throw(BadDimension)
};
```

Для проверки работоспособности класса написать тестирующую программу, в которой вызываются все методы класса.

3.5. Дополнительные задачи

1. Реализовать класс безопасных матриц, который имеет следующий интерфейс:

```
// исключения
struct BadDimension {};
struct BadFirstIndex {};
struct BadSecondIndex {};

// интерфейс класса матрица
class matrix
{
  Long  n, m;  // размерности матрицы
  double*  p;  // указатель на матрицу
  matrix();    // конструктор по умолчанию закрыт
  class row    // строка матрицы
  {
    long m;    // размерность строки
    double* p; // указатель на строку
  public:
    // конструктор по элементам
    row(const long& _m, const double* _p);
    // оператор индексирования
    double& operator [](const long& j) throw(BadSecondIndex);
```

```
    const double& operator [](const long& j) const
                                        throw(BadSecondIndex);
  };
public:
  // конструкторы
  matrix(const long& _n, const long& _m) throw(BadDimension);
  matrix(const matrix& m);
  // деструктор
  ~matrix();
  // операторы присваивания
  matrix& operator =(const matrix& m) throw(BadDimension);
  matrix& operator +=(const matrix& m) throw(BadDimension);
  matrix& operator -=(const matrix& m) throw(BadDimension);
  matrix& operator *=(const matrix& m) throw(BadDimension);
  matrix& operator *=(const double& d);
  // оператор индексирования
  row operator [](const long& i) throw(BadFirstIndex);
  const row operator [](long i) const throw(BadFirstIndex);
  // унарные операторы
  friend matrix operator -(const matrix& m);
  friend matrix operator +(const matrix& m);
  // бинарные операторы
  friend matrix operator *(const double& d, const matrix& m);
  friend matrix operator *(const matrix& m, const double& d);
  friend matrix operator +(const matrix& m1,
                       const matrix& m2) throw(BadDimension);
  friend matrix operator -(const matrix& m1,
                       const matrix& m2) throw(BadDimension);
  friend matrix operator *(const matrix& m1,
                       const matrix& m2) throw(BadDimension);
};
```

Для проверки работоспособности класса написать тестирующую программу, в которой вызываются все методы класса.

2. Разработать класс `BitWord`, который обеспечивает доступ к битам слова по номеру бита в слове, используя для этого оператор индексирования. Интерфейс класс разработать самостоятельно.

Для проверки работоспособности класса написать тестирующую программу, в которой вызываются все методы класса.

3.6. Задачи для индивидуальной работы

1. Разработать класс для описания прямоугольников на плоскости со сторонами, параллельными осям декартовой системы координат. Объекты класса имеют следующие атрибуты:

- координаты двух точек, расположенных в противоположных углах прямоугольника, которые однозначно определяют объект.

Обеспечить выполнение следующих действий над объектами класса:

- проверку правильности задания координат двух точек, однозначно определяющих прямоугольник, при нарушении этого условия сгенерировать исключение `NonRectangle`;
- присваивание одного объекта другому;
- получение нового объекта, симметричного рассматриваемому объекту относительно центра системы координат;
- ввод-вывод объектов класса с использованием для этих целей перегруженных операторов << и >>;
- выполнение над двумя объектами класса операций объединения, пересечения и вычитания с использованием для этих целей соответственно перегруженных как дружественные функции операторов `+, *, -`; при невозможности выполнения некоторой операции сгенерировать соответственно исключения: `NonUnion`, `NonIntersection` или `NonDifference`;
- сравнение объектов класса с использованием для этих целей перегруженных как дружественные функции операторов сравнения (`==, !=`); два объекта считаются равными меду собой при их полном совпадении;

Для проверки работоспособности класса написать тестирующую программу, в которой вызываются все методы класса.

2. Создать класс для описания квадратов на плоскости. Объекты класса имеют следующие атрибуты:

- координаты двух противоположных точек квадрата.

Обеспечить выполнение следующих действий над объектами класса:

- проверку правильности задания координат двух точек, однозначно определяющих объект, при нарушении этого условия сгенерировать исключение `NonSquare`;
- присваивание одного объекта другому;
- ввод-вывод объектов класса с использованием для этих целей перегруженных операторов << и >>;

- получение длины диагонали квадрата;
- получение значения координаты точки объекта, ближайшей к началу координат.
- сравнение двух квадратов по площади с использованием для этих целей перегруженных как дружественные функции операторов `==, !=, <, <=, >, >=`.

Для проверки работоспособности класса написать тестирующую программу, в которой вызываются все методы класса.

3. Разработать класс для описания кругов на плоскости. Объекты класса имеют следующие атрибуты:

- координаты центра круга и его радиус.

Обеспечить выполнение следующих действий над объектами класса:

- проверку правильности задания значения радиуса, в случае ошибки генерировать исключение типа `BadRadius`;
- присваивание одного объекта другому;
- ввод-вывод объектов класса с использованием для этих целей перегруженных операторов << и >>;
- получение длины окружности, ограничивающей круг;
- получение значения площади круга;
- выполнение над двумя объектами класса операций объединения и пересечения с использованием для этих целей перегруженных как дружественные функции операторов + и -. Операции считаются возможными, если один из объектов находится полностью внутри другого объекта. При невозможности выполнения операции сгенерировать соответственно исключения типов `NonUnion` или `NonIntersection`.
- выдачу значений координат центра;
- увеличение или уменьшение значения радиуса объекта на единицу с помощью перегруженных операторов `++` (инкремент) или `--` (декремент) соответственно.
- сравнение площадей двух кругов с использованием для этих целей перегруженных как дружественные функции операторов `==, !=, <, <=, >, >=`.

Для проверки работоспособности класса написать тестирующую программу, в которой вызываются все методы класса.

4. Разработать класс для описания эллиптических областей на плоскости, фокусы которых лежат на прямой, параллельной одной из осей координат. Объекты класса имеют следующие атрибуты:

- координаты двух фокусов эллипса;
- большой и малый радиусы эллипса.

Обеспечить выполнение следующих действий над объектами класса:

- проверку правильности задания значений координат двух фокусов эллипса и величин радиусов, однозначно определяющих объект, при нарушении этого условия сгенерировать исключение `NonEllipse`;
- присваивание одного объекта другому;
- получение нового объекта, симметричного рассматриваемому объекту относительно центра системы координат;
- получение значения площади эллипса;
- ввод-вывод объектов класса с использованием для этих целей перегруженных операторов << и >>;
- выполнение над двумя объектами класса операций пересечения и объединения с использованием для этих целей перегруженных как дружественные функции операторов `'&'` и `'|'` соответственно. Операция считается возможной, если один из объектов находится полностью внутри другого объекта. При невозможности выполнения операции пересечения сгенерировать исключение `NonIntersect`, при невозможности выполнения операции объединения сгенерировать исключение `NonUnion`.
- сравнение площадей двух эллипсов с использованием для этих целей перегруженных как дружественные функции операторов `==, !=, <, <=, >, >=`;
- выдачу значений координат фокусов эллипса;
- увеличение или уменьшение значения радиусов эллипса на единицу с помощью перегруженных операторов `++` (инкремент) и `--` (декремент).

Для проверки работоспособности класса написать тестирующую программу, в которой вызываются все методы класса.

5. Создать класс `Prime` для описания простых чисел. Объектами класса являются простые числа, для хранения которых используются без знаковые длинные целые переменные. Обеспечить выполнение следующих действий над объектами класса:

- проверку, является рассматриваемое число простым; предусмотреть генерацию исключения `NonPrime` при нарушении данного требования;
- присваивание одного объекта другому;
- поиск близнеца для заданного объекта класса; предусмотреть генерацию исключения `NonPossible` при невозможности получения такого числа;
- поиск в натуральном ряду чисел простого числа, следующего по порядку по отношению к рассматриваемому простому числу;
- поиск простого числа, предшествующего в натуральном ряде чисел рассматриваемому простому числу; предусмотреть генерацию исключения `NotHavePrev` при невозможности получения такого числа;
- определения, является ли объект класса сверхпростым числом (число является сверхпростым, если при любой перестановке своих цифр число остается простым);
- ввод-вывод объектов класса с использованием для этих целей перегруженных как дружественные функции операторов << и >>;
- сравнение простых чисел с использованием перегруженных как дружественные функции операторов сравнения (`==`, `!=`, `>`, `<`, `>=`, `<=`).

Для проверки работоспособности класса написать тестирующую программу, в которой вызываются все методы класса.

Лабораторная работа №4

Тема. Наследование классов.

4.1. Определение наследования

Наследование – это отношение между классами, которое позволяет классу потомку использовать члены одного или нескольких классов предков. В языке программирования C++ класс потомок называется производным классом от наследуемых классов, а классы предки называются базовыми классами. В производном классе нет необходимости заново определять атрибуты и методы, которые наследуются им от базового класса. Синтаксически наследование между двумя классами определяется следующим образом:

```
class имя_производного_класса: имя_базового_класса
{
  // тело производного класса
};
```

В производном классе возможен доступ к членам базового класса, которые объявлены в разделах `public` и `protected`. В производном классе запрещен доступ к членам базового класса, которые объявлены в разделе `private`.

В следующем листинге приведен пример определения наследования классов и доступа к членам базового класса через объект производного класса.

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

struct Base
{
  int n;
  int count() { return ++n; }
};

struct Derived: Base
```

```
{
  int m;
  int discount() { return --m; }
};

int main()
{
  Derived d;

  d.n = 10;
  cout << d.count() << endl;     // печатает 11
  d.m = 20;
  cout << d.discount() << endl; // печатает 19

  return 0;
}
```

При создании нового объекта производного класса компилятор всегда вызывает конструктор по умолчанию базового класса. Чтобы изменить такую инициализацию объекта производного класса, нужно в списке инициализации конструктора производного класса явно вызвать требуемый конструктор базового класса. Список инициализации конструктора записывается после его списка параметров и отделяется от него символом `':'`. В следующем листинге показан пример определения конструкторов со списками инициализации в классах `Base` и `Derived`.

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

class Base
{
  int n;
public:
  Base(const int& _n): n(_n) {}
  int count() { return ++n; }
};

class Derived: public Base
```

```
{
  int m;
public:
  Derived(const int& _n, const int& _m): Base(_n), m(_m) {}
  int discount() { return --m; }
};

int main()
{
  Derived d(10, 20);

  cout << d.count() << endl;    // печатает 11
  cout << d.discount() << endl; // печатает 19

  return 0;
}
```

4.2. Доступа к членам, наследуемым от базового класса

При определении производного класса можно также определить права доступа к членам, наследуемым от базового класса. Для этого в определении производного класса перед именем базового класса записывается один из спецификаторов доступа `public`, `protected` или `private`.

Если класс наследуется как `public`, то права доступа к членам, наследуемым от базового класса, в производном классе не изменяется.

Если класс наследуется как `protected`, то все `public` члены, наследуемые от базового класса, становятся `protected` членами производного класса, а доступ к остальным членам, наследуемым от базового класса, не изменяется.

Если класс наследуется как `private`, то все члены, наследуемые от базового класса, становятся `private` членами производного класса.

Если спецификатор доступа перед базовым классом опущен, то по умолчанию для классов он устанавливается в `private`, а для структур – в `public`.

4.3. Наследование и оператор присваивания

Оператор присваивания не наследуется, так как при отсутствии этого оператора в производном классе, компилятор генерирует для этого класса оператор присваивания по умолчанию. Поэтому нетривиальные операторы присваивания нужно переопределять в производных классах.

4.4. Виртуальные функции

В языке программирования C++ существует механизм для вызова функции производного класса в случае, если доступ к объекту производного класса осуществляется через указатель или ссылку на базовый класс. Для этого в производном классе должна быть определена функция, имя и сигнатура которой совпадают с именем и сигнатурой функции базового класса, причем функция базового класса должна быть объявлена с ключевым словом `virtual`. В этом случае говорят, что функция производного класса замещает функцию базового класса. Функции, объявленные с ключевым словом `virtual` называются виртуальными.

В следующем листинге приведен пример вызова виртуальной функции производного класса через указатель на базовый класс.

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

class Base
{
public:
  virtual int what() { return 10; }
};

class Derived: public Base
{
public:
  virtual int what() { return 20; }
};

int main()
{
```

```
  Derived d;

  Base *b = &d;
  cout << b->what() << endl;  // печатает 20

  Base &c = d;
  cout << c.what() << endl;   // печатает 20

  return 0;
}
```

Механизм замещения виртуальных функций можно обойти, если при вызове виртуальной функции указать имя класса, которому она принадлежит.

В следующем листинге приведен пример вызова виртуальной функции базового класса, через указатель на этот класс.

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

class Base
{
public:
  virtual int what() { return 10; }
};

class Derived: public Base
{
public:
  virtual int what() { return 20; }
};

int main()
{
  Base *b = new Derived;
```

```
  cout << b->Base::what() << endl;  // 10

  return 0;
}
```

При уничтожении объекта всегда вызываются деструкторы базовых классов. Поэтому деструкторы базовых классов должны быть виртуальными и всегда иметь реализацию.

4.5. Абстрактные классы

Виртуальная функция называется чистой, если она определена только для спецификации интерфейса класса. Чисто виртуальные функции не имеют реализации и определяются следующим образом:

```
virtual тип имя_функции(параметры) = 0;
```

Предполагается, что чистая виртуальная функция переопределяется в производных классах. Класс, который содержит хотя бы одну чисто виртуальную функцию, называется абстрактным классом.

Так как у чисто виртуальных функций отсутствует реализация, то невозможно создать объект, принадлежащий абстрактному классу. Поэтому абстрактный класс может использоваться только в качестве базового класса для других классов.

Однако можно определить указатель или ссылку на абстрактный класс. Это позволяет использовать абстрактные классы для описания интерфейсов базовых классов и работы с производными классами через указатели или ссылки на базовый класс.

Если класс, производный от абстрактного класса, не определяет все чисто виртуальные функции, то он также является абстрактным классом.

В следующем листинге приведен пример использования ссылки на абстрактный класс в качестве параметра функции.

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

class Abstract
{
public:
```

```
  virtual ~Abstract() {}
  virtual what() = 0;
};

class Concrete: public Abstract
{
public:
  virtual ~Concrete() {}
  virtual what() { return 10; }
};

int foo(Abstract& a)
{
  return a.what();
}

int main()
{
  Concrete c;
  cout << foo(c) << endl;  // печатает 10

  return 0;
}
```

4.6. Задачи для самостоятельного решения

1. Реализовать классы контейнеры и итераторы, иерархии которых приведены на рисунках 1 и 2. Классы, показанные в иерархиях, имеют следующие интерфейсы:

```
////////// Абстрактные контейнеры

// абстрактный Контейнер
class AbstractContainer
{
public:
  virtual ~AbstractContainer() {};
  virtual bool isEmpty() const = 0;  // контейнер пуст?
  virtual bool isFull() const = 0;   // контейнер полный?
};

// абстрактный Стек
class AbstractStack: public AbstractContainer
{
public:
  virtual void push(const int& n) = 0;  // вставить первым
  virtual void pop(int& n) = 0;         // удалить первый
```

```
};

// абстрактная Очередь
class AbstractQueue: public AbstractContainer
{
public:
  virtual void ins(const int& n) = 0;  // вставить последним
  virtual void del(int& n) = 0;        // удалить первый
};
```

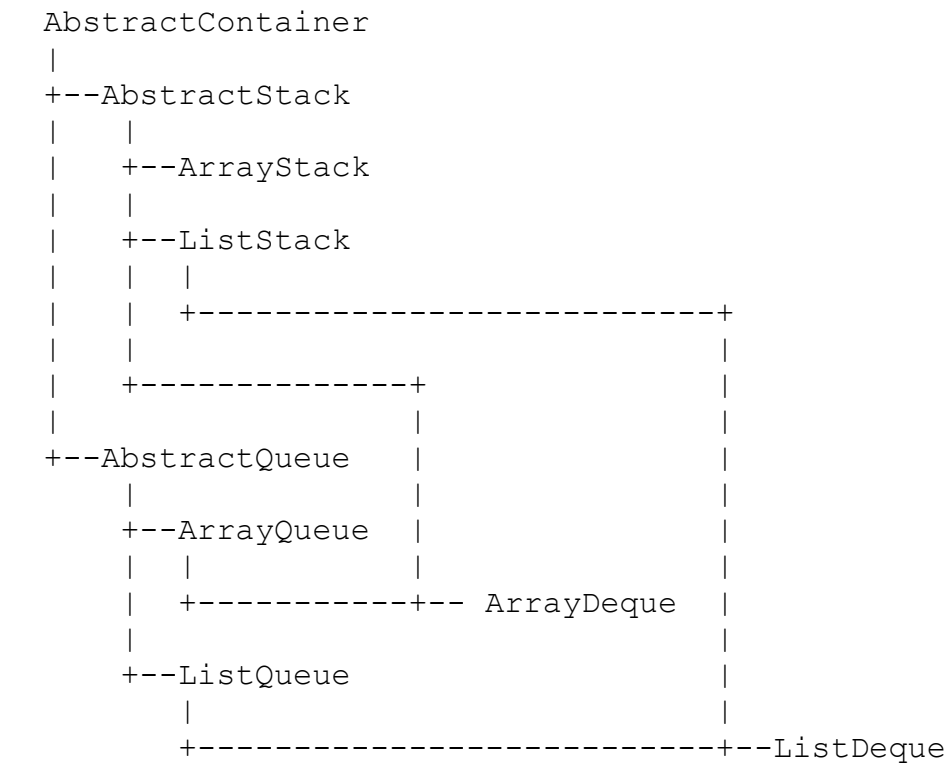

Рис. 1. Иерархия контейнеров

```
///////// Абстрактные итераторы

// абстрактный Итератор
class AbstractIterator
{
public:
  virtual ~AbstractIterator() {};
  virtual bool inRange() const = 0;  // итератор в пределах?
  virtual void reset() = 0;          // сбросить на начало
  virtual int& operator *() const = 0;  // разыменовать
};
```

```
// абстрактный Итератор для продвижения вперед
class AbstractForwardIt
{
public:
  virtual ~AbstractForwardIt() {};
  virtual void operator ++() = 0;  // сдвиг на элемент вперед
};

// абстрактный Итератор для продвижения назад
class AbstractBackwardIt
{
public:
  virtual ~AbstractBackwardIt() {};
  virtual void operator --() = 0;  // сдвиг на элемент назад
};
```

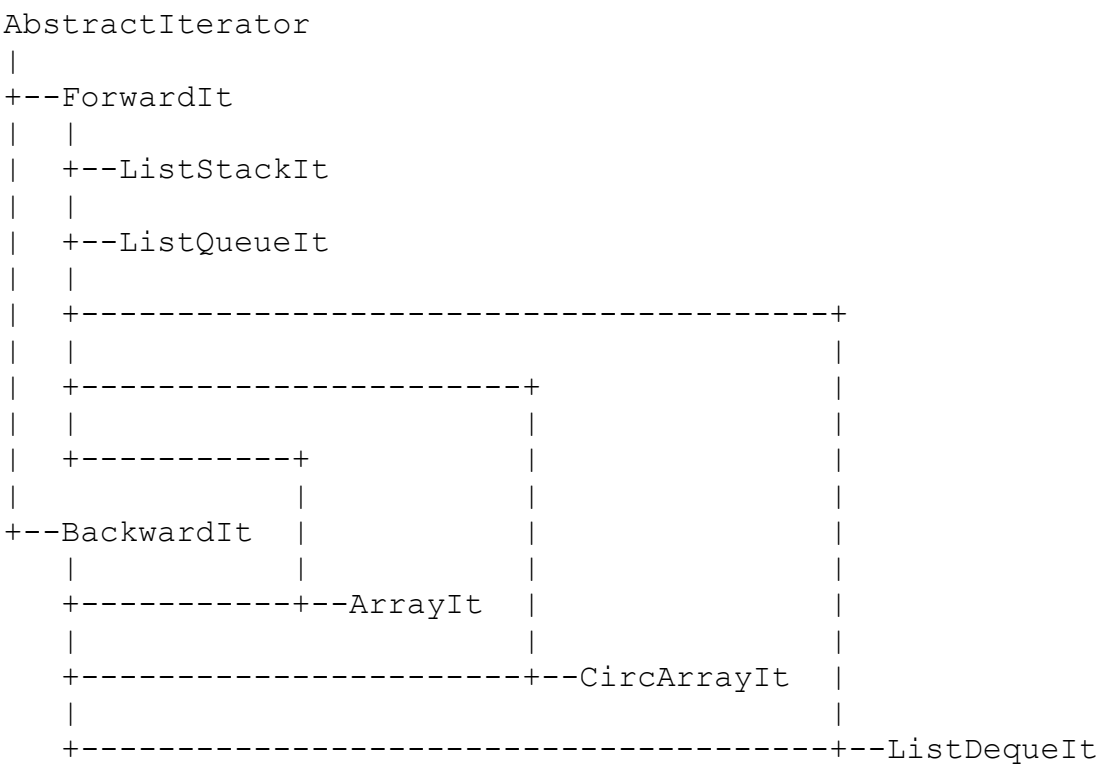

Рис. 2. Иерархия итераторов

```
////////// Конкретные контейнеры на массиве

// Стек на массиве
class ArrayStack: public AbstractStack
```

```
{
protected:
  int  size;  // размерность массива
  int* p;     // указатель на массив
  int  top;   // верхушка стека
public:
  ArrayStack(int _size);
  ArrayStack(const ArrayStack &s);
  virtual ~ArrayStack();
  virtual void push(const int& n);
  virtual void pop(int& n);
  virtual bool isEmpty() const;
  virtual bool isFull() const ;
  friend class ArrayIt;
};

// Кольцевая очередь на массиве
class ArrayQueue: public AbstractQueue
{
protected:
  int  size;  // размерность массива
  int* p;     // указатель на массив
  int  head;  // индекс первого занятого элемента
  int  n;     // количество элементов в очереди
public:
  ArrayQueue(int _size);
  ArrayQueue(const ArrayQueue &q);
  ~ArrayQueue();
  void ins(const int& n);
  void del(int& n);
  bool isEmpty() const;
  bool isFull() const;
  friend class ArrayIt;
};

// Дек на массиве
class ArrayDeque: public ArrayQueue
{
public:
  ArrayDeque(int _size);
  ArrayDeque(const ArrayDeque &d);
  ~ArrayDeque();
  virtual void push(const int& n);
  virtual void pop(int& n);
  friend class ArrayQueueIterator;
  friend class CircularArrayIt;
};
```

```
////////// Конкретные контейнеры на списке

// Стек на списке
class ListStack: public AbstractStack
{
  // вложенный класс
  struct node  // элемент списка
  {
    int item;    // данные
    node*  p;    // указатель на следующий элемент
  };
  node *head;    // указатель на первый элемент в списке
public:
  ListStack();
  ListStack(const ListStack& lst);
  virtual ~ListStack();
  virtual void push(const int& n);
  virtual int pop();
  virtual bool isEmpty();
  virtual bool isFull();
  friend class ListStackIt;
};

// Очередь на списке
class ListQueue: public AbstractQueue
{
  // вложенный класс
  struct node  // элемент списка
  {
    int item;    // данные
    node*  p;    // указатель на следующий элемент
  };
  node *head;    // указатель на первый элемент в списке
public:
  // конструкторы
  ListQueue();
  ListQueue(const ListQueue& lst);
  virtual ~ListQueue();
  virtual void ins(const int& n);
  virtual int del();
  virtual bool isEmpty();
  virtual bool isFull();
  friend class ListQueueIt;
};

// Дек на списке
```

```
class ListDeque: public ListStack, public ListQueue
{
  struct node     // элемент дека
  {
    int  item;    // данные
    node*  next;  // указатель на следующий элемент
    node*  prev;  // указатель на предыдущий элемент
  };
  // указатели на первый и последний элементы в списке
  struct node *head, *tail;
public:
  // конструкторы
  ListDeque();
  ListDeque(const ListDeque& d);
  virtual ~ListDeque();
  virtual void pop(const int& n);
  virtual void ins(const int& n);
  virtual void push(int& n);
  virtual void del(int& n);
  virtual bool isEmpty();
  virtual bool isFull();
  friend class ListDequeIt;
};

////////// Конкретные итераторы на массиве

// Итератор массива
class ArrayIt: public AbstractForwardIt, public AbstractBack-
wardIt
{
  ArrayStack&  a;  // ссылка на стек на массиве
  int pos;         // текущая позиция итератора
  ArrayIt();
public:
  ArrayIt(const ArrayStack& _a);
  virtual bool inRange() const;
  virtual void reset();
  virtual int& operator *() const;
  virtual void operator ++();
  virtual void operator --();
};

// Итератор кольцевого массива
class CircularArrayIt: public AbstractForwardIt, public Ab-
stractBackwardIt
{
  ArrayQueue& a;  // ссылка на очередь
```

```
  int pos;        // текущая позиция итератора
  CircularArrayIt();
public:
  CircularArrayIt(const ArrayQueue& _a);
  virtual bool inRange() const;
  virtual void reset();
  virtual int& operator *() const;
  virtual void operator ++();
  virtual void operator --();
};

////////// Конкретные итераторы на списке

// Итератор для прохода стека на списке вперед
class ListStackIt: public AbstractForwardIt
{
  ListStack&  s;         // ссылка на стек на массиве
  ListStack::node* pos;  // текущая позиция итератора
  ListStackIt();
public:
  ArrayIt(const ListStack& _s);
  virtual bool inRange() const;
  virtual void reset();
  virtual int& operator *() const;
  virtual void operator ++();
};

// Итератор для прохода очереди на списке вперед
class ListQueueIt: public AbstractForwardIt
{
  ListQueue&  q;         // ссылка на стек на массиве
  ListQueue::node* pos;  // текущая позиция итератора
  ListQueueIt();
public:
  ListQueueIt(const ListQueue& _q);
  virtual bool inRange() const;
  virtual void reset();
  virtual int& operator *() const;
  virtual void operator ++();
};

// Итератор для прохода дека на списке вперед и назад
class  ListDequeIt:  public  AbstractForwardIt,  public  Ab-
stractBackwardIt
{
  ListDeque&  dq;        // ссылка на стек на массиве
  ListDeque::node* pos;  // текущая позиция итератора
```

```
  ListDequeIt();
public:
  ListQueueIt(const ListDeque& _q);
  virtual bool inRange() const;
  virtual void reset();
  virtual int& operator *() const;
  virtual void operator ++();
  virtual void operator --();
};
```

Для проверки работоспособности классов написать тестирующую программу, в которой вызываются все методы классов.

2. Разработать класс `Person` для описания человека. Класс содержит следующие атрибуты:

- фамилию;
- имя;
- отчество;
- пол;
- год рождения.

Обеспечить над объектами класса выполнение следующих действий:

- получение фамилии человека;
- изменение фамилии человека;
- получение имени человека;
- получение отчества человека;
- получение пола человека;
- получение года рождения человека.

Создать на базе класса `Person` класс-потомок `Teacher` для описания преподавателей. Объекты класса `Teacher` имеют дополнительно следующие атрибуты:

- должность преподавателя;
- название подразделения;
- стаж работы;

Обеспечить выполнение следующих действий над объектами класса `Teacher`:

- получение должности преподавателя;
- перевод преподавателя на другую должность;
- получение названия подразделения, где работает преподаватель;
- перевод преподавателя в другое подразделение;

- получение стажа работы преподавателя;
- увеличение стажа работы преподавателя на один год.

Создать на базе класса `Person` класс-потомок `Student` для описания студентов. Объекты класса `Student` имеют дополнительно следующие атрибуты:

- название факультета, на котором учится студент;
- номер курса, на котором учится студент;
- номер группы, в которой учится студент.

Обеспечить выполнение следующих действий над объектами класса `Student`:

- получение названия факультета, на котором учится студент;
- перевод студента на другой факультет;
- получение номера курса, на котором учится студент;
- перевод студента на следующий курс;
- получение номера группы, в которой учится студент;
- перевод студента в другую группу.

Для проверки работоспособности классов написать тестирующую программу, в которой вызываются все методы классов.

4.7. Дополнительные задачи

1. Разработать классы четырехугольников, иерархия которых приведена на рисунке 3.

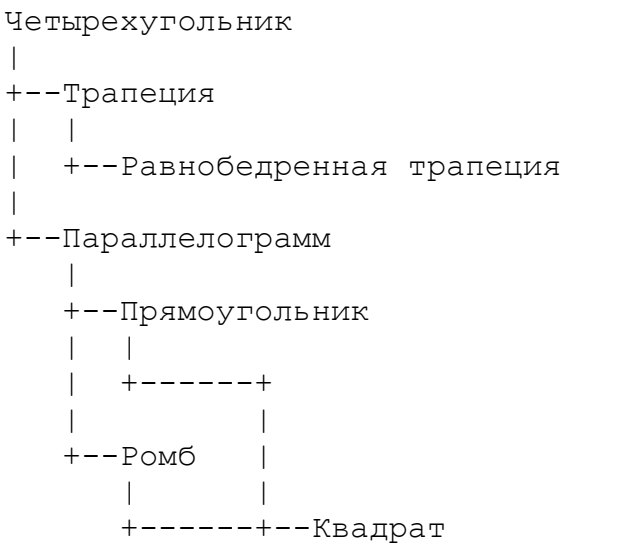

Рис. 3. Иерархия четырехугольников

Базовый класс иерархии имеет следующий интерфейс:

```
class Rectangle
{
     virtual ~Rectangle() {}
     virtual double perimeter() = 0;  // периметр
     virtual double area() = 0;       // площадь
};
```

Интерфейсы остальных классов разработать самостоятельно.

Для проверки работоспособности классов написать тестирующую программу, в которой вызываются все методы классов.

2. Разработать классы пространственных областей, иерархия которых приведена на рисунке 4.

```
Область в пространстве
|
+--Эллипсоид
|  |
|  +--Шар
|
+--Многогранник
   |
   +--Правильный многогранник
```

Рис. 4. Иерархия областей в пространстве

Базовый класс иерархии имеет следующий интерфейс:

```
class Volume
{
     virtual ~ Volume () {}
     virtual double surfaceArea() = 0; // площадь поверхности
     virtual double volume() = 0;      // объем
};
```

Интерфейсы остальных классов разработать самостоятельно.

Для проверки работоспособности классов написать тестирующую программу, в которой вызываются все методы классов.

4.8. Задачи для самостоятельного решения

1. Разработать классы арифметических и геометрических прогрессий, иерархия которых приведена на рисунке 5.

```
Прогрессия
|
+--Арифметическая прогрессия
|
+--Геометрическая прогрессия
```

Рис. 5. Иерархия прогрессий

Базовый класс иерархии имеет следующий интерфейс:

```
class Progression
{
  virtual ~Progression() {}
  virtual double nMember(const int& n) = 0;
  virtual double nSum(const int& n) = 0;
};
```

Интерфейсы остальных классов разработать самостоятельно.

Для проверки работоспособности классов написать тестирующую программу, в которой вызываются все методы классов.

2. Разработать классы областей на плоскости, иерархия которых приведена на рисунке 6.

```
Область на плоскости
|
+--Эллипс
|  |
|  +--Круг
|
+--Многоугольник
   |
   +--Правильный многоугольник
```

Рис. 6. Иерархия областей на плоскости

Базовый класс иерархии имеет следующий интерфейс:

```
class Area
```

```
{
     virtual ~Area() {}
     virtual double perimeter() = 0;  // периметр
     virtual double area() = 0;       // площадь
};
```

Интерфейсы остальных классов разработать самостоятельно.

Для проверки работоспособности классов написать тестирующую программу, в которой вызываются все методы классов.

3. Разработать классы треугольников, иерархия которых приведена на рисунке 7.

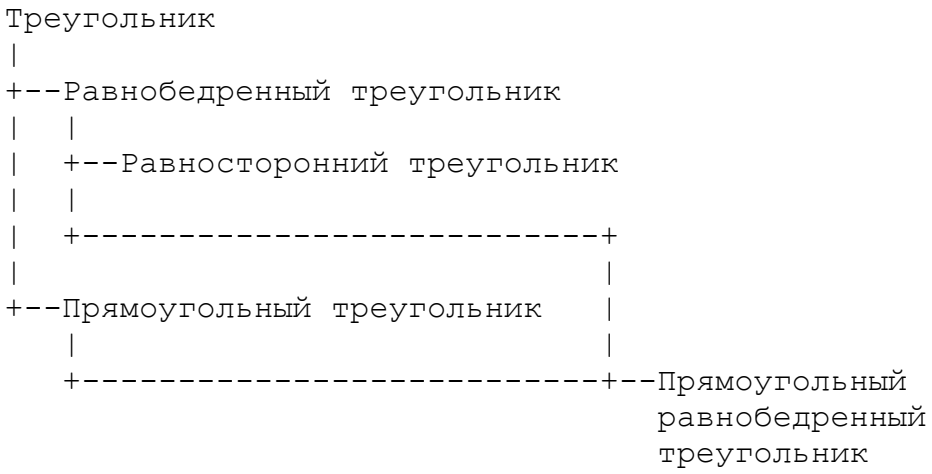

Рис. 7. Иерархия треугольников

Для проверки работоспособности классов написать тестирующую программу, в которой вызываются все методы классов.

Лабораторная работа №5

Тема. Шаблоны функций.

5.1. Определение шаблона функции

Функция, которая абстрагируется от типа хотя бы одного из своих параметров, называется обобщённой или родовой функцией. В языке программирования C++ родовые функции определяются шаблонами функций.

Определение шаблона функции имеет следующий вид:

```
template <список_параметров> определение_функции
```

где в угловых скобках указан список параметров шаблона функции, который не может быть пустым. Параметрами шаблона функции могут быть:

- типы;
- не типы;
- шаблоны класса.

Объявление параметра-типа имеет следующий вид:

```
class|typename имя_параметра
```

где ключевые слова `class` и `typename` являются взаимозаменяемыми.

Параметр-тип представляет в определении функции тип данных и его также можно использовать для задания типа, возвращаемого функцией значения.

Например, в следующем листинге определен шаблон функции `min`, находящей минимальный элемент массива. При этом тип массива является параметром `T` шаблона функции.

```
template <class T> T min(T *a, const int size)
{
  T m = a[0];
  for (int i = 1; i < size; ++i)
    if (a[i] < m)
      m = a[i];
  return m;
}
```

Параметр-не-тип шаблона функции объявляется как переменная, которая используется в определении функции. Эта переменная может иметь один из следующих типов:

- целочисленный тип;
- тип перечисления;
- указатель на объект, функцию или член класса;
- ссылка на объект или функцию.

Например, в следующем листинге определен шаблон функции `min`, для нахождения минимального элемента массива, параметр `size` которого не является типом и определяет размерность массива.

```
template <int size, class T> T min(T *a)
{
  const int n = size;
  T m = a[0];
  for (int i = 1; i < n; ++i)
    if (a[i] < m)
      m = a[i];
  return m;
}
```

Параметром шаблона функции может быть также шаблон класса. Работа с шаблонами классов рассмотрена в следующей лабораторной работе.

Шаблоны функций могут перегружаться также как и обычные функции. Т.е. допускается объявление шаблонов функций с одинаковыми именами, но разными параметрами.

5.2. Конкретизация шаблона функции

Процесс создания из шаблона функции определения конкретной функции называется конкретизацией или инстанциированием шаблона. Определения функции, созданные из шаблона функции, называются специализациями шаблона функции или шаблонными функциями. Конкретизация шаблона функции может быть явной или неявной.

При явной конкретизации шаблона функции после имени функции нужно сначала указать параметры шаблона в угловых скобках, а затем – параметры функции в круглых скобках.

Шаблон функции конкретизируется неявно, если опущен хотя бы один его параметр. Это может быть как при вызове функции, так и при определении адреса функции. Если какой-то параметр шаблона опущен, то также должны быть опущены и все последующие параметры.

Примеры явной и неявной конкретизации шаблона функции приведены в следующем листинге.

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

template <class T> T min(T *a, const int size)
{
  T m = a[0];
  for (int i = 1; i < size; ++i)
    if (a[i] < m)
      m = a[i];
  return m;
}

int main()
{
  int a[3] = {3, 1, 2};
  double b[3] = {3.1, 1.2, 2.3};

  int n = min<int>(a, 3);  // явная конкретизация
  cout << n << endl;       // печатает 1

  double m = min(b, 3);    // неявная конкретизация
  cout << m << endl;       // печатает 1.2

  return 0;
}
```

5.3. Явная специализация шаблона функции

Явной специализацией шаблона функции называется определение шаблона функции для конкретных значений параметров этого шаблона. Явная специализация шаблона функции имеет следующий синтаксис:

```
template<> имя_функции<список_аргументов_шаблона>
                        (параметры_функции)
```

```
блок_функции
```

где имя_функции задает имя шаблона функции, а список_аргументов_шаблона содержит аргументы, для которых выполняется явная специализация этого шаблона функции.

Явная специализация шаблона функции используется в двух случаях:

- какой-то тип данных не может быть параметром шаблона функции в силу невозможности такой реализации шаблона, которая была бы совместима с другими типами данных;
- специальная реализация шаблона функции для какого-то типа данных более эффективна, чем общая реализация шаблона.

Например, рассмотрим шаблон функции max, которая находит максимальное значение из двух аргументов:

```
template <class T> T max(T t1, T t2)
{
  return  t1 > t2 ? t1 : t2;
}
```

Так как общая реализация этого шаблона некорректна для строк, то для них пишут специальную реализацию шаблона. В этом случае параметры шаблона не указываются, а после имени функции в угловых скобках записываются аргументы шаблона для его явной специализации.

```
template <>
char* max<char*>(char* s1, char* s2)
{
  return strcmp(s1, s2) > 0 ? s1 : s2;
}
```

Так как в этом случае шаблон не содержит дополнительных параметров, то явную специализацию шаблона можно заменить нешаблонной функцией, которая никак не связана с шаблоном и не является его явной специализацией:

```
char* max(char* s1, char* s2)
{
```

```
    return strcmp(s1, s2) > 0 ? s1 : s2;
  }
```

При поиске нужной функции компилятор сначала просматривает нешаблонные функции с точным соответствием типов параметров типам аргументов. Поэтому в этом случае для строк будет вызываться нешаблонная функция max.

5.4. Модели компиляции шаблонов

В языке программирования C++ поддерживаются две модели компиляции шаблонов:

- модель компиляции с включением,
- модель компиляции с разделением.

В модели компиляции с включением определение шаблона должно включаться в каждый файл, в котором этот шаблон конкретизируется. Обычно, в этом случае определение шаблона помещается в заголовочный файл, который затем включается директивой `#include` в каждый исходный файл, в котором он используется.

В модели компиляции с разделением объявление шаблона функции помещаются в заголовочный файл, а определение – в исходный файл. Причем определению шаблона должно предшествовать ключевое слово `export`. В этом случае шаблон называется экспортируемым. Шаблон функции должен быть определен как экспортируемый только один раз во всей программе. После этого заголовочный файл с объявлением шаблона включается во все исходные файлы, в которых требуется конкретизация шаблона. Эта модель компиляции не всегда поддерживается компиляторами по причине сложности её реализации.

5.5. Задачи для самостоятельного решения

1. Разработать шаблон функции для нахождения в не отсортированном массиве заданного элемента. Шаблон функции имеет следующий прототип:

```
template<class T> bool search(T* arr, const int& size,
                              const T& item);
```

Для проверки работоспособности шаблона функции написать тестирующую программу, в которой функция вызывается для разнотипных массивов.

2. Разработать шаблон функции для обмена значениями максимального и минимального элементов массива. Элементы массива могут иметь любой тип, для которого определен оператор сравнения на «меньше». Шаблон функции имеет следующий прототип:

```
template<class T> void swap(T* arr, const int& size);
```

Для проверки работоспособности шаблона функции написать тестирующую программу, в которой функция вызывается для разнотипных массивов.

3. Разработать шаблон функции для сортировки элементов массива по возрастанию. Элементы массива могут иметь любой тип, для которого определен оператор сравнения на «меньше». Шаблон функции имеет следующий прототип:

```
template<class T> void sort(T* arr, const int& size);
```

Для проверки работоспособности шаблона функции написать тестирующую программу, в которой функция вызывается для разнотипных массивов.

4. Разработать шаблон функции для бинарного поиска элемента в отсортированном массиве. Элементы массива могут иметь любой тип, для которого определен оператор сравнения на «меньше». Шаблон функции имеет следующий прототип:

```
template<class T> int bsearch(const T& item,
                              const T* arr, const int& size);
```

Для проверки работоспособности шаблона функции написать тестирующую программу, в которой функция вызывается для разнотипных отсортированных массивов.

5. Разработать шаблон функции для слияния двух отсортированных массивов в один отсортированный массив. Элементы массивов должны иметь тип сравнимый на «меньше». Шаблон функции имеет следующий прототип:

```
template<class T> T* merge(T* a1, const int& size1,
                           T* a2, const int& size2);
```

Для проверки работоспособности шаблона функции написать тестирующую программу, в которой функция вызывается для разнотипных отсортированных массивов.

5.6. Дополнительные задачи

1. Разработать шаблон оператора для объединения двух массивов в один массив. Результирующий массив не должен иметь повторяющихся элементов. Шаблон функции имеет следующий прототип:

```
template<class T> T* operator +(T* a1, const int& size1,
                                T* a2, const int& size2);
```

Для проверки работоспособности шаблона функции написать тестирующую программу, в которой функция вызывается для разнотипных массивов.

2. Разработать шаблон оператора для нахождения пересечения двух массивов. Результирующий массив должен содержать только элементы, которые входят в каждый из исходных массивов. Шаблон функции имеет следующий прототип:

```
template<class T> T* operator *(T* a1, const int& size1,
                                T* a2, const int& size2);
```

Для проверки работоспособности шаблона функции написать тестирующую программу, в которой функция вызывается для разнотипных массивов.

3. Разработать шаблон оператора для нахождения разности двух массивов. Результирующий массив должен содержать только элементы, которые входят только в первый из массивов. Шаблон функции имеет следующий прототип:

```
template<class T> T* operator -(T* a1, const int& size1,
                                T* a2, const int& size2);
```

Для проверки работоспособности шаблона функции написать тестирующую программу, в которой функция вызывается для разнотипных массивов.

4. Разработать шаблон функции для применения к каждому элементу массива некоторой шаблонной функции. Шаблон функции имеет следующий прототип:

```
template<class T> void forEach(T* arr, const int& size,
                               void (*f)(T& item));
```

Для проверки работоспособности шаблона функции написать тестирующую программу, в которой функция вызывается для разнотипных массивов и шаблонных.

5.7. Задачи для индивидуальной работы

1. Разработать шаблон функции для нахождения первого вхождения одного массива в другой. Шаблон функции имеет следующий прототип:

```
template<class T> int subarray(T* whole, const int& size1,
                               T* part, const int& size2);
```

Для проверки работоспособности шаблона функции написать тестирующую программу, в которой функция вызывается для разнотипных массивов.

2. Разработать шаблон функции для нахождения последнего вхождения одного массива в другой. Шаблон функции имеет следующий прототип:

```
template<class T> int subarray(T* whole, const int& size1,
                               T* part, const int& size2);
```

Для проверки работоспособности шаблона функции написать тестирующую программу, в которой функция вызывается для разнотипных массивов.

3. Разработать шаблон функции для проверки массивов на равенство. Массивы считаются равными, если совпадают их размерности и соответствующие элементы. Шаблон функции имеет следующий прототип:

```
template<class T> bool equal(T* a1, const int& size1,
                             T* a2, const int& size2);
```

Для проверки работоспособности шаблона функции написать тестирующую программу, в которой функция вызывается для разнотипных массивов.

4. Разработать шаблон функции для проверки массивов на равенство. Массивы считаются равными, если совпадают их размерности и соответствующие элементы. Шаблон функции имеет следующий прототип:

```
template<class T> int equal(T* a1, const int& size1,
                             T* a2, const int& size2);
```

В случае неравенства массивов шаблонная функция `equal` должна возвращать индекс первых несовпадающих элементов массивов, а в случае равенства – произвольное отрицательное число.

Для проверки работоспособности шаблона функции написать тестирующую программу, в которой функция вызывается для разнотипных массивов.

Лабораторная работа №6

Тема. Шаблоны классов.

6.1. Определение шаблона класса

Класс, который абстрагируется от типа хотя бы одного своего атрибута или от типа параметра хотя бы одного своего метода, называется родовым или обобщенным классом. В языке программирования C++ родовые классы определяются шаблонами классов. Определение шаблона класса имеет следующий вид:

```
template <список_параметров>
определение_класса
```

где в угловых скобках указан список параметров шаблона класса, который не может быть пустым. Параметры шаблона класса имеют тот же смысл и подчиняются тем же требованиям, что и параметры шаблона функции.

Функции-члены шаблона класса могут быть определены как в шаблоне класса, так и вне шаблона класса. В последнем случае они должны быть явно объявлены как члены шаблона класса.

В следующем листинге приведен пример шаблона контейнерного класса `Stack`, предназначенного для хранения значений определенного типа `T`, который является параметром шаблона класса. Размерность стека задается целочисленным параметром `n` шаблона класса. Этот параметр можно было бы определить и в конструкторе класса, а не в списке параметров шаблона.

```
template <class T, int n> class Stack
{
  int size;  // размер стека
  int top;   // верхушка стека
  T* a;      // указатель на массив
public:
  Stack(): size(n), top(0), a(new int[size]) {}
  ~Stack() { delete[] a; }

  void push(const T& n) { a[top++] = n; }
```

```
  void pop(T& n) { n = a[--top]; }

  bool isEmpty() const { return !top; }
  bool isFull() const { return top == size; }
};
```

Для шаблонов классов поддерживаются те же модели компиляции, что и для шаблонов функций, а именно, модели компиляции с включением и разделением.

6.2. Конкретизация шаблона класса

Процесс создания определения класса из шаблона класса называется конкретизацией или инстанциированием шаблона класса. Класс, созданный из шаблона, называется специализацией шаблона класса или шаблонным классом. Шаблон класса конкретизируется только явно.

В следующем листинге приведен пример конкретизации шаблона класса `Stack`, определенного в предыдущем параграфе. Предполагается, что определение этого шаблона хранится в заголовочном файле `Stack.h`.

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

using namespace std;

int main()
{
  int n;
  Stack<int, 2> s;  // стек для хранения двух целых чисел

  cout << s.isEmpty() << ' ' << s.isFull() << endl;
  s.push(10);
  s.push(20);
  cout << s.isEmpty() << ' ' << s.isFull() << endl;

  s.pop(n);
  cout << n << endl;
  s.pop(n);
```

```
  cout << n << endl;
  cout << s.isEmpty() << ' ' << s.isFull() << endl;

  return 0;
}
```

Шаблон класса конкретизируется только при определении объекта класса. При объявлении ссылок и указателей на класс шаблон класса не конкретизируется, так как в этом случае объект класса может и не существовать.

По определению считается, что функция-член шаблона класса сама является шаблоном. Стандарт языка программирования C++ требует, чтобы она конкретизировалась только при вызове или при взятии её адреса. Поэтому при конкретизации шаблона класса компилятор генерирует код только тех функций-членов класса, которые явно или неявно вызываются в программе.

6.3. Явная специализация шаблона класса

Явной специализацией шаблона класса называется определение шаблона класса для конкретных значений всех параметров этого шаблона. Явная специализация шаблона класса имеет следующий синтаксис:

```
template<>
class|struct|union имя_класса<список_аргументов_шаблона>
блок_класса
```

где `имя_класса` задает имя шаблона класса, а в угловых скобках указан список аргументов шаблона, для которых выполняется явная специализация этого шаблона класса.

Явная специализация шаблона класса используется в тех же случаях, что и явная специализация шаблонов функций.

В следующем листинге приведен пример явной специализации шаблона класса.

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

using namespace std;
```

```
template <class T> struct Demo
{
  T t;
  T max(T _t) { return t > _t ? t : _t; }
};

// явная специализация шаблона для строк
template<> struct Demo<char*>
{
  char* t;
  char* max(char* _t) { return strcmp(t, _t) > 0 ? t : _t; }
};

int main()
{
  Demo<int> d;
  d.t = 10;
  cout << d.max(20) << endl;     // печатает 20

  Demo<char*> s;
  s.t = "aaa";
  cout << s.max("bbb") << endl;  // печатает bbb

  return 0;
}
```

Шаблон класса можно специализировать для одного или нескольких параметров, оставляя другие параметры неспециализированными. Такая специализация шаблона класса называется частичной. Частичную специализацию шаблонов классов поддерживают не все компиляторы.

6.4. Задачи для самостоятельного решения

1. Реализовать шаблон класса `ArrayStack`, который имеет следующий интерфейс:

```
template <class T> class ArrayStack  // стек на массиве
```

```
{
  int  size;    // размерность массива
  T* p;         // указатель на массив
  int  top;     // верхушка стека
public:
  // конструкторы
  ArrayStack(const int& _size);
  ArrayStack(const ArrayStack<T>& s);  // копирования
  // деструктор
  ~ArrayStack();
  // функции-члены класса
  void push(const T& n);  // втолкнуть элемент в стек
  T pop();                // вытолкнуть элемент из стека
  bool isEmpty();         // пустой стек?
  bool isFull();          // полный стек?
};
```

Определение и реализацию шаблона класса поместить в заголовочный файл `stack.h`.

Для проверки работоспособности шаблона класса `ArrayStack` написать тестирующую программу, в которой из шаблона класса `ArrayStack` создаются различные шаблонные классы и вызываются все методы этих шаблонных классов.

2. Реализовать шаблон класса `ListStack`, который имеет следующий интерфейс:

```
template <class T> class ListStack  // стек на списке
{
  // вложенный класс
  struct node  // элемент списка
  {
    T item;    // данные
    node* p;   // указатель на следующий элемент
  };
  node *head;    // указатель на первый элемент в списке
public:
  // конструкторы
  ListStack();                          // по умолчанию
  ListStack(const ListStack<T>& lst);  // копирования
  ~ListStack();           // деструктор
  void push(const T& n);// включить элемент в голову списка
  T pop();              // удалить элемент из головы списка
  bool isEmpty();       // пустой список?
};
```

Определение и реализацию шаблона класса поместить в заголовочный файл `list.h`.

Для проверки работоспособности шаблона класса `ListStack` написать тестирующую программу, в которой из шаблона класса `ListStack` создаются различные шаблонные классы и вызываются все методы этих шаблонных классов.

3. Реализовать шаблон класса `Set`, который имеет следующий интерфейс:

```
template <class T> class Set
{
  int size;   // размер множества
  int count;  // количество элементов в множестве
  T* p;       // указатель на массив элементов
public:
  // конструктор по умолчанию
  Set(const int& _size = 0);
  // конструктор копирования
  Set(const Set<T>& s);
  // деструктор
  ~Set();
// функции члены класса
  int getSize() const;  // размерность множества
  int getCount() const;  // количество элементов в множестве
  void include(const T& n);  // включить элемент
  void exclude(const T& n);  // исключить элемент
  void empty();  // удаление из множества всех элементов
  bool isInSet(const T& n) const;  // элемент в множестве?
  // операторы члены класса
  Set<T>& operator =(const Set<T>& s);  // присваивание
  Set<T>& operator +=(const Set<T>& s); // объединение
  Set<T>& operator -=(const Set<T>& s); // разность
  Set<T>& operator *=(const Set<T>& s); // пересечение
  // операторы друзья класса
  friend Set<T> operator +(const Set<T>& s1,
                           const Set<T>& s2);  // объединение
  friend Set<T> operator -(const Set<T>& s1,
                           const Set<T>& s2);  // разность
  friend Set<T> operator *(const Set<T>& s1,
                           const Set<T>& s2);  // пересечение
  // операторы ввода-вывода
  friend istream& operator >>(istream& in, Set<T>& s);
  friend ostream& operator <<(ostream& out, const Set<T>& s);
};
```

Определение и реализацию шаблона класса поместить в заголовочный файл `set.h`.

Для проверки работоспособности шаблона класса `Set` написать тестирующую программу, в которой из шаблона класса `Set` создаются различные шаблонные классы и вызываются все методы этих шаблонных классов.

6.5. Дополнительные задачи

1. Реализовать шаблон класса `ArrayQueue`, который имеет следующий интерфейс:

```
// кольцевая очередь на массиве
template <class T> class ArrayQueue
{
  int  size;     // размерность массива
  T*   p;        // указатель на массив
  int  head;     // индекс первого занятого элемента
  int  n;        // количество элементов в очереди
public:
  ArrayQueue(const int& _size);        // инициализация
  ArrayQueue(const ArrayQueue<T>& q);  // копирование
  ~ArrayQueue();          // деструктор очереди
  void ins(const T& n);   // включить элемент в очередь
  T del();                // удалить элемент из очереди
  bool isEmpty();         // очередь пустая?
  bool isFull();          // очередь полная?
};
```

Определение и реализацию шаблона класса поместить в заголовочный файл `queue.h`.

Для проверки работоспособности шаблона класса `ArrayQueue` написать тестирующую программу, в которой из шаблона класса `ArrayQueue` создаются различные шаблонные классы и вызываются все методы этих шаблонных классов.

2. Реализовать шаблон класса `ListQueue`, который имеет следующий интерфейс:

```
template <T> class ListQueue  // очередь на списке
{
  // вложенный класс
```

```
  struct node  // элемент списка
  {
    T item;    // данные
    node*  p;  // указатель на следующий элемент
  };
  node *head;    // указатель на первый элемент в списке
public:
  // конструкторы
  ListQueue();                          // по умолчанию
  ListQueue(const ListQueue<T>& lst);  // копирования
  ~ListQueue();          // деструктор
  void ins(const T& n);  // включить элемент в хвост списка
  T del();               // удалить элемент из головы списка
  bool isEmpty();        // пустой список?
};
```

Определение и реализацию шаблона класса поместить в заголовочный файл `list.h`.

Для проверки работоспособности шаблона класса `ListQueue` написать тестирующую программу, в которой из шаблона класса `ListQueue` создаются различные шаблонные классы и вызываются все методы этих шаблонных классов.

6.6. Задачи для индивидуальной работы

1. Реализовать шаблон класса `ArraryDeque`, который имеет следующий интерфейс:

```
// дек – двустороння очередь на массиве
template <T> class ArrayDeque
{
  int  size;    // размерность массива
  T* p;         // указатель на массив
  int  head;    // индекс первого занятого элемента
  int  n;       // количество элементов в очереди
public:
  ArrayDeque();                          // инициализация
  ArrayDeque(const ArrayDeque<T>& d);  // копирование
  ~ArrayDeque();         // деструктор
  void pop(const T& n);  // включить элемент в голову дека
  void ins(const T& n);  // включить элемент в хвост дека
  T push();      // исключить элемент из головы дека
  T del();       // исключить элемент из хвоста дека
  bool isEmpty();        // дек пустой?
  bool isFull();         // дек полный?
```

```
};
```

Определение и реализацию шаблона класса поместить в заголовочный файл `deque.h`.

Для проверки работоспособности шаблона класса `ArrayDeque` написать тестирующую программу, в которой из шаблона класса `ArrayDeque` создаются различные шаблонные классы и вызываются все методы этих шаблонных классов.

2. Реализовать шаблон класса `ListDeque`, который имеет следующий интерфейс:

```
// дек – двустороння очередь на списке
template <class T> class ListDeque
{
  struct node     // элемент дека
  {
    T  item;      // данные
    node*  next;  // указатель на следующий элемент
    node*  prev;  // указатель на предыдущий элемент
  };
  // указатели на первый и последний элементы в списке
  struct node *head, *tail;
public:
  // конструкторы
  ListDeque();                          // по умолчанию
  ListDeque(const ListDeque<T>& d);  // копирования
  ~ListDeque();           // деструктор
  void pop(const T& n);  // включить элемент в голову дека
  void ins(const T& n);  // включить элемент в хвост дека
  T push();       // исключить элемент из головы дека
  T del();        // исключить элемент из хвоста дека
  bool isEmpty();         // дек пустой?
};
```

Определение и реализацию шаблона класса поместить в заголовочный файл `deque.h`.

Для проверки работоспособности шаблона класса `ListDeque` написать тестирующую программу, в которой из шаблона класса `ListDeque` создаются различные шаблонные классы и вызываются все методы этих шаблонных классов.

Список литературы

1. Б. Страуструп. Язык программирования C++, 3-е изд. – СПб. – М.: Невский диалект – Бином, 1999. – 991 с.

2. С. Б. Липпман, Ж. Лажойе. Язык программирования C++. Вводный курс, 3-е изд. – СПб. – М.: Невский диалект – ДМК Пресс, 2001. – 1024 с.

3. Х. Дейтел, П. Дейтел. Как программировать на C++. – М.: Бином, 1998. – 1024 с.

4. В. Штерн. Основы C++. Методы программной инженерии. – М.: Лори, 2003. – 880 с.

5. А. П. Побегайло. C/C++ для студента. – СПб. – БХВ-Петербург, 2006. – 528 с.

6. Д. Вандевурд, Н. М. Джосаттис. Шаблоны C++: справочник разработчика. – М. Вильямс, 2003. – 544 с.