Лабораторная работа 15

Классы

В данной лабораторной работе необходимо разработать класс и программу, иллюстрирующую возможности данного класса. Рассмотрим на двух примерах, как создаются классы.

Задание:

Разработать класс (по заданию преподавателя) и программу, иллюстрирующую возможности данного класса

Интерфейс программы:

В программе должны быть:

- два объекта данного класса,
- пункты меню для ввода с клавиатуры параметров (свойств) этих объектов,
- пункты меню для вывода на дисплей параметров (свойств) этих объектов
- пункты меню для вывода на дисплей вычисленных значений этих объектов.

Замечания:

- Все проверки корректности ввода должны проводиться в методах класса.
- Все вычисления параметров должны быть в методах класса, в функции main – минимальные действия

Сначала немного теории…

Класс – *тип данных, определяемый пользователем* и представляющий собой модель реального объекта в виде совокупности данных и функций для работы с ними.

Данные класса называются *членами-данными* или *полями* (по аналогии с полями структуры) или *свойствами*, а функции класса – *членами-функциями* или *методами*.

Члены данные и члены-функции, в свою очередь, называются *элементами класса*.

Объявление класса

В языке C++ описание класса состоит из ключевого слова class, после которого пишется имя класса. Далее идет тело класса, заключенное в фигурные скобки, после которых стоит точка с запятой. Внутри фигурных скобок располагают *объявления свойств и методов*:

```
class Point   // Объявление класса Point
{  // Тело класса
    . . .     // Здесь располагают объявления
       . . .    // свойств и методов
};
```

Объявление класса

Все члены класса делят на закрытые и открытые. Видимостью элементов класса управляют спецификаторы доступа *private* и *public*.

```
class Point   // Объявление класса Point
{
private:
    . . .     // Здесь располагают объявления свойств
        . . .     // и закрытых методов
public:
    . . .     // Здесь располагают объявления
        . . .     // открытых методов
};
```

Создание объектов на базе класса

Для того, чтобы воспользоваться классом, необходимо объявить объект этого класса (**создать экземпляра класса**):

```
Point a;           // статический объект
Point x1, x2, x3; // три статических объекта
Point Array[10];  // статический массив объектов
Point *py = new Point;          // динамический объект
Point *pArray = new Point[20]; // динамический массив объектов
```

`a` – объект класса `Point` или переменная типа `Point`

`Array[10]` – массив из 10 объектов класса `Point`

`py` – указатель на объект класса `Point`

Пример класса

Пример объявления класса:

```
class Point   // Объявление класса Point
{
private:
  int x, y; // Координаты класса
public:
  void SetXY(int i, j); // Задание координат
  int GetX();           // Чтение координаты x
  int GetY();           // Чтение координаты y
  . . . .
};
```

Пример использования класса:

```
Point a;          // Объявление объекта класса Point
a.SetXY(10,20);   // Вызов метода SetXY для объекта a
k = a.GetX(); // Вызов метода GetX для объекта a
```

Заголовочные файлы и файлы определений

В языке C++ используются два типа файлов:

`*.h` — заголовочный файл, в котором принято записывать:

- Объявление классов
- Объявление глобальных функций

`*.cpp` — файл определений или исходный, в котором принято записывать:

- Определения методов класса
- Определения глобальных функций
- Объявления глобальных переменных и объектов

Размещение объявлений и определений

Файл MyProgramm.h

```
class Point   // Объявление класса Point
{
private:
  int x, y; // Координаты класса
public:
  void SetXY(int i, j);   // Объявление метода
  int GetX() {return x;}  // Объявление и определение метода
  int GetY() {return y;}  // Объявление и определение метода
  . . . .
};
```

Файл MyProgramm.cpp

```
void Point::SetXY(int i, j) // Определение метода
{
  if (i >= 0) x = i;
  if (j >= 0) y = j;
}
```

Конструкторы и деструкторы

Конструктор – это функция-член класса, автоматически вызываемая программой *при создании экземпляра* класса.

Конструкторы используются для начальной инициализации членов-данных объекта и выделения динамической памяти под внутренние массивы объекта.

Имя конструктора должно совпадать с именем класса.

При уничтожении объекта автоматически вызывается другая функция – ***деструктор***. Имя деструктора состоит из символа "тильда" и имени класса.

Конструкторы и деструкторы

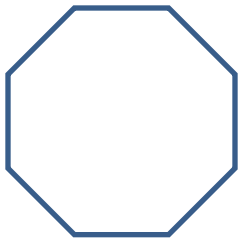
```
class Point   // Объявление класса Point
{
private:
  int x, y; // Координаты класса
public:
  Point() {x = 0; y = 0;}  // Конструктор по умолчанию
                           // Конструктор с параметрами
  Point(int i, int j) {x = 0; y = 0; SetXY(i, j);}

  void SetXY(int i, j);
  int GetX() {return x;}
  int GetY() {return y;}
  . . . .
};
```

Пример использования класса с конструктором:

```
Point a, b(10, 20);
k = a.GetX();
n = b.GetX();
```

Пример разработки класса



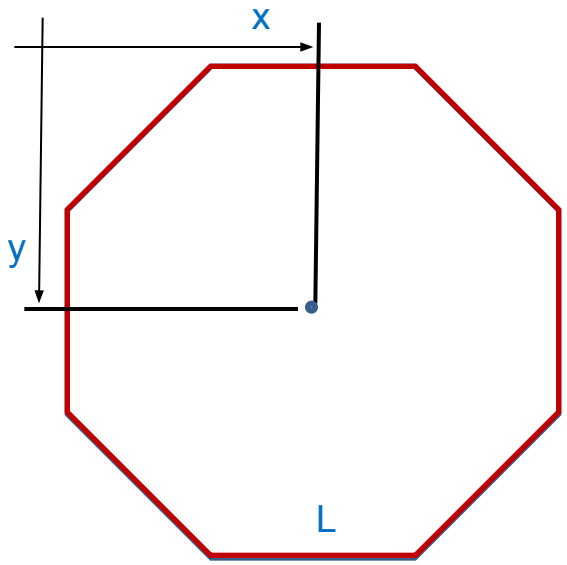
Разработать класс **Равносторонний восьмиугольник** и программу, иллюстрирующую возможности данного класса. Класс должен хранить координаты и размеры восьмиугольника и включать:
методы, позволяющие задавать и читать размеры восьмиугольника;
метод, позволяющие вычислять площадь и периметр восьмиугольника.

Все проверки корректности ввода должны проводиться в методах класса.

Восьмиугольник

Чтобы было удобно работать с восьмиугольниками, нужно правильно определить свойства и методы класса. Если задавать фигуры координатами вершин, то параметров будет очень много (16), да и задавать их будет очень сложно: нужно на бумаге или на дисплее нарисовать восьмиугольник, определить координаты всех вершин, задать их в объект-восьмиугольник, да еще и проверить вводимые значения на корректность (по заданию должен быть равносторонний восьмиугольник).

Поэтому сначала нужно определить минимальное количество параметров, которые однозначно задают равносторонний восьмиугольник. Проще всего задать координаты центра восьмиугольника (x и y) и длину его стороны (L). Все остальные параметры, в том числе координаты вершин можно будет легко вычислить из этих данных.



Координаты x и y в функциях обычно считаются от верхнего левого угла дисплея или окна

Восьмиугольник – свойства

Некоторые параметры в классе должны быть всегда положительные, например длина стороны, поэтому зададим синоним типа **unsigned int**, чтобы покороче писать в классе:

```
typedef unsigned int UINT;
```

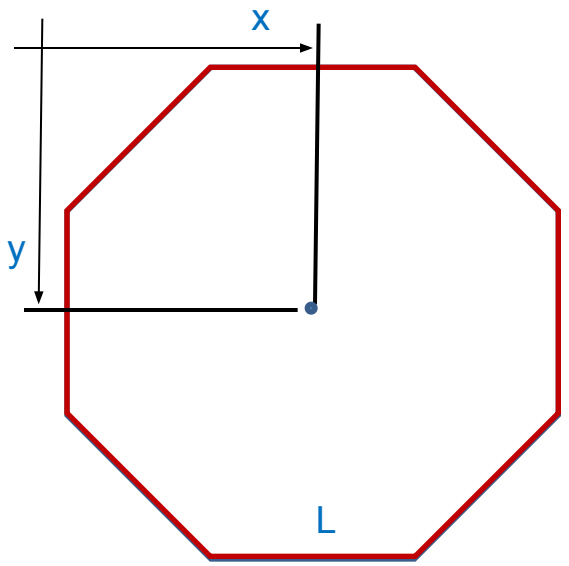
Назовем класс **Octagon** и запишем заголовок класса:

```
class Octagon
{
};
```

В закрытую часть запишем свойства. Координаты пусть будут целого типа, а длина стороны – целое беззнаковое.

```
 class Octagon // 8-угольник
{
private:
   int x, y;     // Координаты центра
   UINT L;        // Длина стороны
public:

};
```



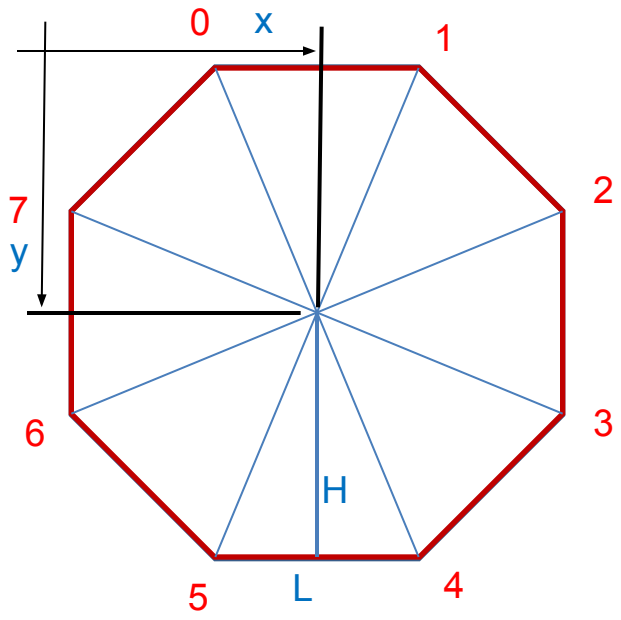
Восьмиугольник

Координаты вершин легко вычислить из этих свойств объекта (L, x и y)

Для этого запишем радиус вписанной окружности: H = L/2 * tg(67.5)

Координаты вершин вычисляются так:

Номер вершины	Координата x	Координата y
0	x – L/2	y – H
1	x + L/2	y – H
2	x + H	y – L/2
3	x + H	y + L/2
4	x + L/2	y + H
5	x –L/2	y + H
6	x – H	y + L/2
7	x – H	y – L/2



Восьмиугольник – свойства

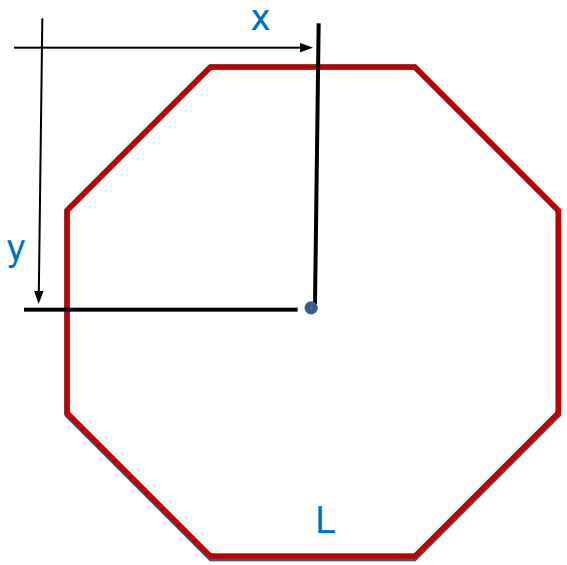
Хранить отдельно в классе радиус вписанной окружности не надо, его всегда можно вычислить, для этого вставим в класс метод, вычисляющий его:

```
class Octagon // 8-угольник
{
private:
  int x, y;      // Координаты центра
  UINT L;        // Длина стороны
  float GetH(); // Радиус вписанной окружности
public:

};
```

Вставим метод, вычисляющий радиус вписанной окружности, в закрытую часть класса,
т.к. он нужен только для использования в методах этого же класса.

```
float Octagon::GetH()
{    // Радиус вписанной окружности
   return L * 1.2071;
}
```



Восьмиугольник – конструкторы

В открытую часть класса вставим конструкторы, которые будут вызываться при создании объектов. Создадим 2 конструктора:

-без параметров (чтобы объявлять объект с некоторыми начальными значениями свойств)

-с 3-мя параметрами (чтобы объявлять объект с конкретными значениями свойств)

```
class Octagon // 8-угольник
{
private:
  int x, y;      // Координаты центра
  UINT L;        // Длина стороны
  float GetH(); // Радиус вписанной окружности
public:
  Octagon() {x = 20; y = 20; L = 10;}
  Octagon(int ix, int iy, UINT il) {x = ix; y = iy; L = il;}
};
```

Теперь можно объявлять объекты, например, так:

```
 Octagon a1;
 Octagon a2(100, 100, 25);
```

Центр восьмиугольника a1 будет в точке 20, 20, длина стороны будет равна 10 .

Центр восьмиугольника a2 будет в точке 100, 100, длина стороны будет равна 25 .

Восьмиугольник – методы

Далее вставляем в класс методы, описанные в задании:
методы, позволяющие задавать свойства восьмиугольника (x, y, L).
Эти методы ничего не должны возвращать, следовательно перед именем функции пишем void. Аргументы зависят от того, какие свойства задаем, например:

```
void SetX(int ix) {x = ix;} // Задает x типа - int
void SetY(int iy) {y = iy;} // Задает y типа - int
void SetL(UINT il) {L = il;} // Задает L типа – unsigned int
```

Может понадобиться метод, задающий сразу обе координаты:

```
void SetXY(int ix, int iy) {x = ix; y = iy;}
```

Вставляем эти методы в класс:

```
class Octagon // 8-угольник
{ . . . .
public:
. . . .
  void SetX(int ix) {x = ix;}
  void SetY(int iy) {y = iy;}
  void SetXY(int ix, int iy) {x = ix; y = iy;}
  void SetL(UINT il) {L = il;}
};
```

В эти методы можно будет вставить проверки на корректность вводимых данных, если она потребуется. Вызываться методы будут так:

```
Octagon a1;
a1.SetXY(33, 44); // задаем координаты центра объекта a1
```

Восьмиугольник – методы

Вставляем в класс следующие методы, описанные в задании:
методы, позволяющие читать свойства восьмиугольника;
Аргументов у методов нет, т.к. они возвращают значения свойств, хранящиеся в том объекте, для которого будут вызываться. Возвращаемое значение зависит от того, какие значения мы читаем.

```
int GetX() {return x;} // Читаем значение x, хранящееся в
объекте
int GetY() {return y;} // Читаем значение y
UINT GetL() {return L;} // Читаем значение L
```

Вставляем эти методы в класс:

```
class Octagon // 8-угольник
{
. . . .
public:
. . . .
  int GetX() {return x;}
  int GetY() {return y;}
  UINT GetL() {return L;}
};
```

Возвращать сразу обе координаты нельзя

~~`int GetXY() {return x, y;}`~~

Т.к. функция может возвращать только одно значение!

Вызываться методы будут так:

```
Octagon a1;
a1.SetXY(33, 44); // задаем координаты центра объекта a1
. . .
k = a1.GetX(); // из объекта a1 читается значение координаты x
```

Восьмиугольник – методы

Вставляем в класс другие методы, описанные в задании:
методы, позволяющие вычислять площадь и периметр восьмиугольника;
Аргументов у методов нет, т.к. они вычисляют площадь и периметр по значению свойств, хранящихся в том объекте, для которого будут вызываться.
Возвращаемое значение зависит от того, что мы читаем.

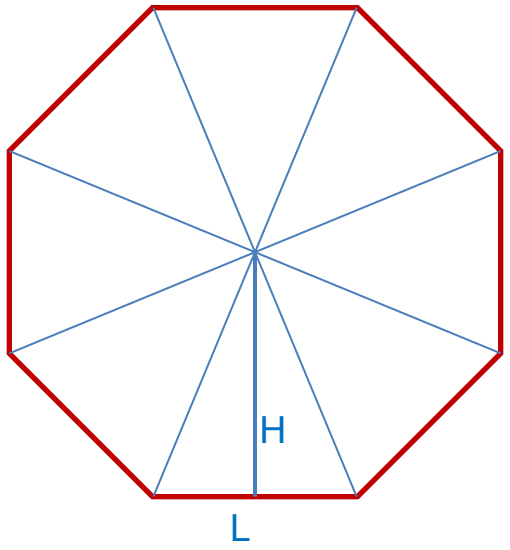
```
float GetSquare();   // вычисляем площадь восьмиугольника
UINT GetPerimeter(); // вычисляем периметр восьмиугольника
```

Площадь вычисляем через треугольники:

```
float Octagon::GetSquare()
{
  return 8 * GetH()*L/2;
}
```

Периметр вычисляем через L:

```
UINT Octagon::GetPerimeter()
{
  return 8 * L;
}
```



Восьмиугольник – методы

Вставляем эти методы в класс:

```
class Octagon // 8-угольник
{
. . . .
public:
. . . .
  float GetSquare();    // вычисляем площадь восьмиугольника
  UINT GetPerimeter(); // вычисляем периметр восьмиугольника
};
```

Вызываться методы будут так:

```
Octagon a1;
a1.SetXY(33, 44); // задаем координаты центра объекта a1
. . .
f = a1.GetSquare(); // вычисляется площадь объекта a1
k = a1.GetSquare(); // вычисляется периметр объекта a1
```

Класс Восьмиугольник

Теперь получаем класс:

```
class Octagon // 8-угольник
{
private:
  int x, y;      // Координаты центра
  UINT L;        // Длина стороны
  float GetH(); // Радиус вписанной окружности
public:
  Octagon() {x = 20; y = 20; L = 10;}
  Octagon(int ix, int iy, UINT il) {x = ix; y = iy; L = il;}

  void SetX(int ix) {x = ix;} // Задаем значение x
  void SetY(int iy) {y = iy;} // Задаем значение y
  void SetXY(int ix, int iy) {x = ix; y = iy;} // Задаем x и y
  void SetL(UINT il) {L = il;} // Задаем значение L
  int GetX() {return x;}     // Читаем значение x
  int GetY() {return y;}     // Читаем значение y
  UINT GetL() {return L;}    // Читаем значение L
  float GetSquare();   // вычисляем площадь восьмиугольника
  UINT GetPerimeter(); // вычисляем периметр восьмиугольника
};
```

Добавление методов

Можно добавить метод, позволяющий читать координаты вершин восьмиугольника. Для этого нужно указать методу номер вершины.

```
int GetTopX(UINT i); // Координата x i-той вершины
int GetTopY(UINT i); // Координата y i-той вершины
```

Вернуть сразу обе координаты так нельзя:

~~`int GetTopXY(UINT i);`~~

Т.к. функция может возвращать только одно значение.

Если нужно вернуть два значения, придется использовать структуру или класс, например так:

```
struct Point
{
  int x;
  int y;
};
....
Point GetTopXY(UINT i);
```

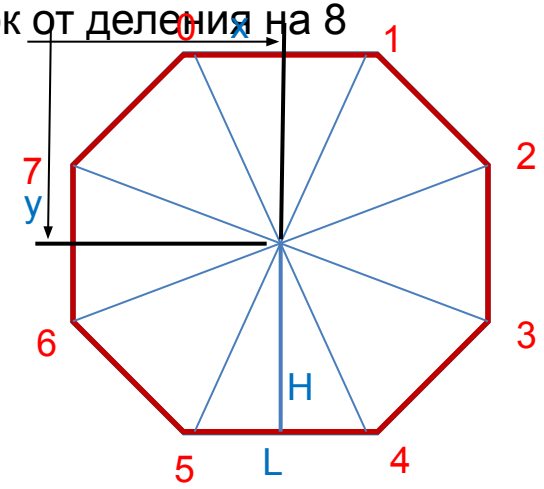
Вычислить координаты вершин можно по данным, приведенным в таблице.

Номер вершины	Координата x	Координата y
0	x – L/2	y – H
1	x + L/2	y – H
2	x + H	y – L/2
3	x + H	y + L/2
4	x + L/2	y + H
5	x –L/2	y + H
6	x – H	y + L/2
7	x – H	y – L/2

Чтение координат вершин восьмиугольника

```
int Octagon::GetTopX(UINT i) // Координата x i-той вершины
{
  float fl = L/2, fh = GetH();
  float coord[8] = {-fl, fl, fh, fh, fl, -fl, -fh, -fh};
  return (x + coord[i % 8]);
}
int Octagon::GetTopY(UINT i) // Координата y i-той вершины
{
  float fl = L/2, fh = GetH();
  float coord[8] = {-fl, fl, fh, fh, fl, -fl, -fh, -fh};
  return (y + coord[7 - (i % 8)]);
}
```

Можно проверить номер вершины на корректность, но можно просто вычислять остаток от деления на 8



Номер вершины	Координата x	Координата y
0	x – L/2	y – H
1	x + L/2	y – H
2	x + H	y – L/2
3	x + H	y + L/2
4	x + L/2	y + H
5	x –L/2	y + H
6	x – H	y + L/2
7	x – H	y – L/2

Чтение координат вершин восьмиугольника

Если нужно вернуть два значения, придется использовать структуру

```
struct Point
{
  int x;
  int y;
};
....
Point Octagon::GetTop(UINT i) // Координаты x и y i-той вершины
{
  Point p;
  float fl = L/2, fh = GetH();
  float coord[8] = {-fl, fl, fh, fh, fl, -fl, -fh, -fh};

  p.x = x + coord[i % 8];
  p.y = y + coord[7 - (i % 8)];
  return p;
}
```

Доработанный класс Восьмиугольник

```
class Octagon // 8-угольник
{
private:
  int x, y;      // Координаты центра
  UINT L;        // Длина стороны
  float GetH(); // Радиус вписанной окружности
public:
  Octagon() {x = 20; y = 20; L = 10;}
  Octagon(int ix, int iy, UINT il) {x = ix; y = iy; L = il;}

  void SetX(int ix) {x = ix;} // Задаем значение x
  void SetY(int iy) {y = iy;} // Задаем значение y
  void SetXY(int ix, int iy) {x = ix; y = iy;} // Задаем x и y
  void SetL(UINT il) {L = il;} // Задаем значение L
  int GetX() {return x;}     // Читаем значение x
  int GetY() {return y;}     // Читаем значение y
  UINT GetL() {return L;}    // Читаем значение L
  float GetSquare();   // вычисляем площадь восьмиугольника
  UINT GetPerimeter(); // вычисляем периметр восьмиугольника
  int GetTopX(UINT i); // Координата x i-той вершины
  int GetTopY(UINT i); // Координата y i-той вершины
  Point GetTop(UINT i); // Координаты x и y i-той вершины
};
```

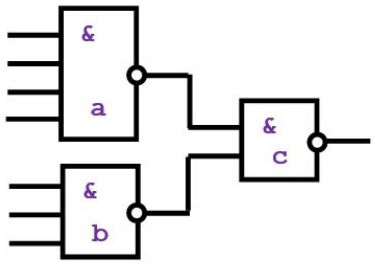
Применение класса Восьмиугольник

```
void main()
{
  Octagon a1, a2;
  Point p;
  a1.SetXY(100,200);
  a2.SetXY(55, 55);
  a1.SetL(25);
  a2.SetL(50);

  printf("Площадь a1 = %f \n", a1.GetSquare());
  printf("Площадь a2 = %f \n", a2.GetSquare());
  printf("Периметр a1 = %d \n", a1.GetPerimeter());
  printf("Периметр a2 = %d \n", a1.GetPerimeter());
  printf("a1 - координаты вершин:\n");
  for(int i=0; i<8; i++)
  {
    p = a1.GetTop(i);
   printf("%d. x = %d, y = %d\n", i, p.x, p.y);
  }
  return 0;
};
```

Класс Логический элемент

Для начала, нужно понять, для чего нужен такой логический элемент. Он нужен для создания схем, следовательно он должен хранить количество входов логического элемента (2, 3, 4 или 8 входов), значения входных сигналов и позволять:



- задавать количество входов логического элемента;
- задавать сигналы 0 или 1 на конкретном входе или сразу на всех входах массивом;
- считывать сигналы с конкретного входа;
- вычислять выходной сигнал в соответствии с логической функцией, выполняемой данным элементом.

Логический элемент - свойства

Значения входных сигналов, поданных на входы элемента, можно хранить в статическом (а лучше в динамическом) массиве типа bool (хотя можно и BYTE). Кроме того, нужно хранить число входов элемента (число входов всегда положительное, поэтому выберем тип для этого свойства – **`unsigned int`** или **`UINT`**)

```
class LogElement
{
private:
  bool* InpArray;  // Указатель на массив входных
сигналов
  UINT Count;      // Число входов
public:
};
```

В этом варианте нужно динамически выделять память под массив и не забывать освобождать ее. Для этого лучше сделать метод, выделяющий память под массив, и метод, освобождающий память.

Логический элемент - конструкторы

Кроме того, класс *Логический элемент* должен иметь конструкторы, позволяющие создавать элементы с заданным числом входов или числом входов по умолчанию

```
LogElement();
LogElement(UINT s);
```

в которых, кроме того, будет выделяться память и задаваться начальные значения входных сигналов.

В случае динамического массива не забыть сделать деструктор.

```
~LogElement();
```

В нем можно вызывать метод, освобождающий память:

Класс Логический элемент

Предварительно класс будет выглядеть так.

```
class LogElement
{
private:
  bool* InpArray;  // Указатель на массив входных
сигналов
  UINT Count;      // Число входов
public:
LogElement();
LogElement(UINT s);
  ~LogElement();

  void Clear();
};
```

Логический элемент - методы

В соответствии с заданием, класс *Логический элемент* должен иметь соответствующие методы, например:

- Метод, задающий число входов:

```
void SetCount(UINT s);
```

где s - число входов. В этом методе нужно проверять число входов на корректность и выделять память (в случае динамического массива).

- Метод, задающий сигнал на i-том входе:

```
void Set(UINT i, bool b);
```

где i – номер входа, но котором нужно установить значение b. В этом методе нужно проверить i, чтобы не выйти за пределы массива.

- Метод, задающий сигнал на всех входах массивом данных:

```
void Set(bool* ArrB);
```

где `ArrB` – массив переменных типа `bool` со значениями входных сигналов

- Метод, считывающий сигнал с i-того входа:

```
bool Get(UINT i);
```

где i – номер входа, но с которого нужно считать значение. В этом методе нужно проверить i, чтобы не выйти за пределы массива.

- Метод, вычисляющий сигнал в соответствии с логической функцией :

```
bool Get();
```

Класс Логический элемент

Теперь класс будет выглядеть так.

```
class LogElement
{
private:
  bool* InpArray;  // Указатель на массив входных сигналов
  UINT Count;      // Число входов
  void Clear();
public:
LogElement();
LogElement(UINT s);
  ~LogElement();

  void SetCount(UINT s); // Задаем число входов
  void Set(UINT i, bool b); // Задаем сигнал на i-том
входе
  void Set(bool* ArrB); // Задаем сигнал на всех входах
  bool Get(UINT i); // Читаем сигнал с i-того входа
  bool Get(); // Вычисляем сигнал  выходе элементан
};
```

Логический элемент - методы

Теперь определяем метод, задающий число входов:

```
void LogElement::SetCount(UINT s)
{   // Сначала проверяем, корректно ли значение s
  if (s == 2 || s == 3 || s == 4 || s == 8)
  { // если корректно
    Count = s; // Запоминаем новое число входов
    Clear(); // Освобождаем старую память
    InpArray = new bool[Count]; // выделяем новую память
    for(UINT i = 0; i < Count; i++)
 InpArray[i] = false; // задаем начальные значения
  }
}
```

Метод, освобождающий память:

```
void LogElement::Clear()
{
  if (InpArray)  // Если память выделялась
  {
    delete InpArray; // освобождаем ее
    InpArray = 0; // и указателю присваиваем адрес 0
  }
}
```

Логический элемент - конструкторы

Используя метод задающий число входов и метод освобождающий память запишем конструкторы.

Конструктор по умолчанию, он не имеет аргументов. Если его не создавать, то по умолчанию будет неопределенное число входов, память не будет выделена и т.д. Если задать число входов 0, то это будет нелогично. Поэтому пусть с его помощью создается элемент с числом входов, например, 2.

```
LogElement::LogElement()
{
  InpArray = 0; // Указателю присваиваем 0, чтобы
        // в методе SetCount не освобождалась
память
  SetCount(2); // Задаем число вхо
}
```

Создание объекта:
`LogElement a;`
У объекта `a` – 2 входа

Логический элемент - конструкторы

Второй конструктор будем использовать, когда захотим создать объект с заданным числом входов (отличным от 2) :

```
LogElement::LogElement(UINT s) // s – число входов
{
  InpArray = 0; // Указателю присваиваем 0, чтобы
        // в методе SetCount не освобождалась
память
  SetCount(2); // Сначала создаем элемент с числом
      // входов равным 2; если s будет
накорректним
      // то останется число входов = 2
  SetCount(s); // Пытаемся выделить
число
               // входов s
}
```

Создание объекта:
`LogElement b(4);`
У объекта `b` – 4 входа

Не забываем создать деструктор:

```
LogElement::~LogElement()
{
  Clear(); // Освобождаем память
```

Логический элемент - методы

Теперь определяем метод, задающий сигнал на i-том входе:

```
void LogElement::Set(UINT i, bool b) // i = 0 ÷ Count-1
{   // проверим i, чтобы не выйти за пределы массива
  if (i < Count)
    InpArray[i] = b;
}
```

Номера входов будут начинаться с 0, т.к. индексация массива начинается с 0.

Прнименение метода:

```
LogElement a;
a.Set(1, true);
```

На входе с индексом 1 объекта `a` – true

Если нужно, чтобы номера входов начинались с 1 (более привычно), то индекс надо сдвинуть на 1:

```
void LogElement::Set(UINT i, bool b) // i = 1 ÷ Count
{   // проверим i, чтобы не выйти за пределы массива
  if (i > 0 && i <= Count)
    InpArray[i - 1] = b;
}
```

Номера входов будут начинаться с 1.

Логический элемент - методы

Теперь определяем метод, задающий сигнал на всех входах массивом данных:

```
void LogElement::Set(bool* ArrB)
{
  for (UINT i = 0; i < Count; i++)
    InpArray[i] = ArrB[i];
}
```

Здесь может быть источник ошибок, т.к. `ArrB` может содержать значений меньше `Count`.

Прнименение метода:

```
LogElement a(4);
bool Inp[4] = {true, true, false, true};

a.Set(Inp);
```

Логический элемент - методы

Метод, считывающий сигнал с i-того входа:

```
bool LogElement::Get(UINT i) // i = 0 ÷ Count-1
{  // проверим i, чтобы не выйти за пределы массива
  if (i < Count)
    return InpArray[i];
}                    // Номера входов будут  начинаться с
0
```

Если оставить так, то в случае, когда i выходит за пределы массива, ничего возвращаться не будет, и это вызовет ошибку выполнения программы. Чтобы исправить это, можно возвращать, например, значение последнего входа (правда оно ничем не лучше первого) или просто false.

```
bool LogElement::Get(UINT i)
{  // проверим i, чтобы не выйти за пределы массива
  if (i < Count)
    return InpArray[i];
  return InpArray[Count - 1];
}
```

Но в этом случае никак не понять, что был задан неправильный номер входа. Лучший вариант – вызывать исключительную ситуацию (Exception) и потом корректно ее обрабатывать.

Логический элемент - методы

Чтобы номера входов начинались с 1, индекс надо сдвинуть на 1:
Метод, считывающий сигнал с i-того входа:

```
bool LogElement::Get(UINT i) // i = 1 ÷ Count
{  // проверим i, чтобы не выйти за пределы массива
  if (i > 0 && i <= Count)
    return InpArray[i - 1];
  return InpArray[Count];
}
```

Вариант с использованием исключительной ситуации (Exception) :

```
bool LogElement::Get(UINT i) // i = 1 ÷ Count
{  // проверим i, чтобы не выйти за пределы массива
  if (i > 0 && i <= Count)
    return InpArray[i - 1];
  throw "Неверный номер входа";
}
```

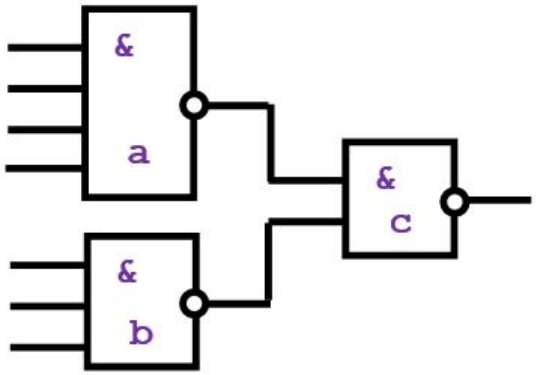
Логический элемент - методы

Метод, вычисляющий сигнал в соответствии с логической функцией определяется этой самой функцией, например И-НЕ:

```
bool LogElement::Get()
{
  bool b = 1;
  for (UINT i = 0; i < Count; i++)
    b &= InpArray[i]; // Операция И всех входных
сигналов
  return !b;          // Отрицание
}
```

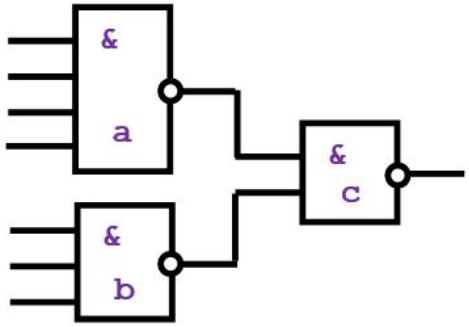
Класс Логический элемент

Тогда можно будет работать просто и удобно с логическими элементами, например так:
Рассчитать схему:




```
LogElement a(4), b(3), c; // Создаем элементы с
// заданным числом входов
// У элемента с число входов – по умолчанию - 2
```

Класс Логический элемент




```
// Задаем сигналы на входах:
a.Set(1, 0); // 0 на 1-ом входе элемента a
a.Set(2, 1); // 1 на 2-ом входе элемента a
a.Set(3, 1); // 1 на 3-ем входе элемента a
a.Set(4, 1); // 1 на 4-ом входе элемента a

b.Set(1, 1); // 1 на 1-ом входе элемента b
b.Set(2, 0); // 0 на 2-ом входе элемента b
b.Set(3, 1); // 1 на 3-ем входе элемента b
// Сигналы с выходов элем-в a и b подаем на входы элем. c:
c.Set(1, a.Get()); // С выхода элем. a – на 1-й вход элем. c
c.Set(2, b.Get()); // С выхода элем. b – на 2-й вход элем. c
bool rez = c.Get(); // Результат - на выходе элемента c
printf("Сигнал на выходе = %d\n", rez);  // rez = 0

a.Set(1, 1); // Изменим сигнал на 1-ом входе элемента a
c.Set(1, a.Get()); // Пересчитаем вых. сигнал элемента a
rez = c.Get(); // Пересчитаем вых. сигнал элемента c
printf("Сигнал на выходе = %d\n", rez);  // rez = 1
```