



МИНОБРНАУКИ РОССИИ
федеральное государственное бюджетное образовательное учреждение
высшего образования
«Московский государственный технологический университет «СТАНКИН»
(ФГБОУ ВО «МГТУ «СТАНКИН»)

Факультет информационных технологий и систем управления

Кафедра компьютерных систем управления

Пушков Роман Львович

Программирование и основы алгоритмизации

Методические рекомендации по выполнению лабораторных работ по дисциплине
«Программирование и алгоритмизация»
для студентов «МГТУ «СТАНКИН», обучающихся по
направлению 15.03.06 «Мехатроника и робототехника»
направленность Робототехника и робототехнические системы: разработка и применение
(шифр направления, название)

Москва 2016г.

Оглавление

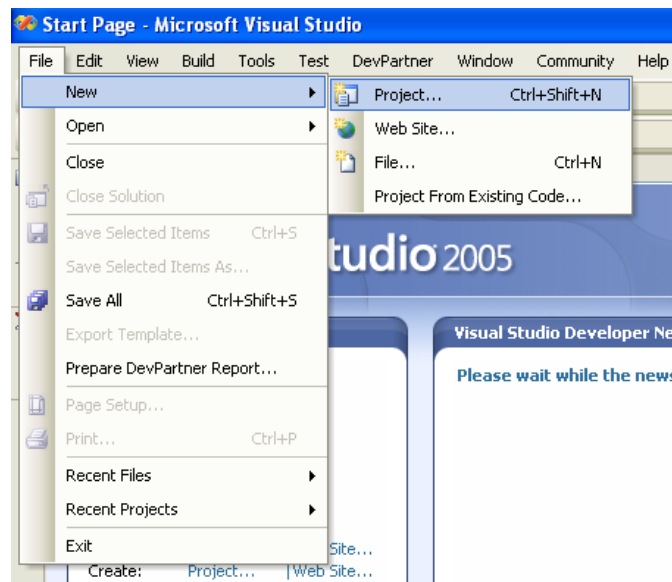
Лабораторная работа №1 «Базовые алгоритмы: очередь и стек»	3
Лабораторная работа №2 «Абстрактные классы. Полиморфизм»	13
Лабораторная работа №3 «Алгоритмы сортировки»	19
Лабораторная работа №4 «Алгоритмы поиска»	26

Лабораторная работа №1 «Базовые алгоритмы: очередь и стек»

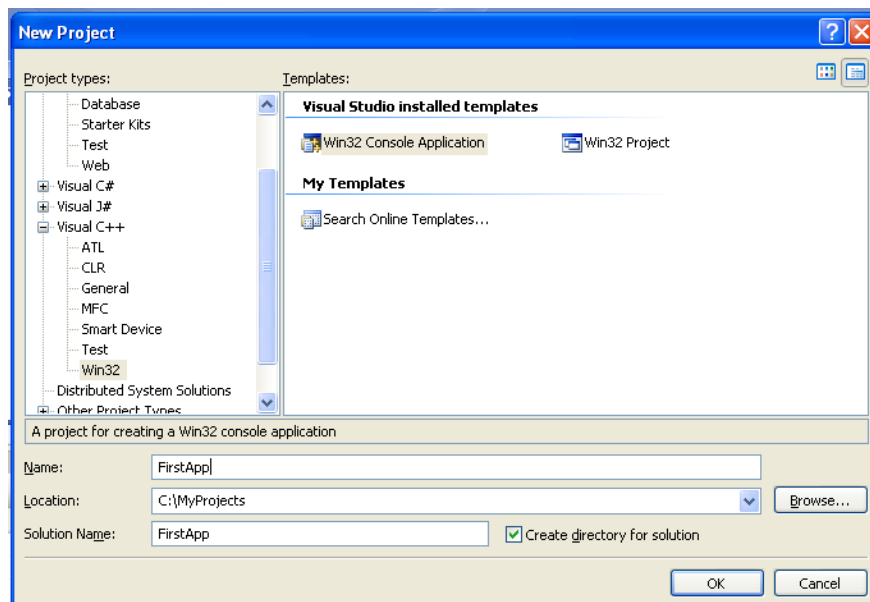
Цель работы: освоить алгоритмы работы очереди и стека.

Ход работы:

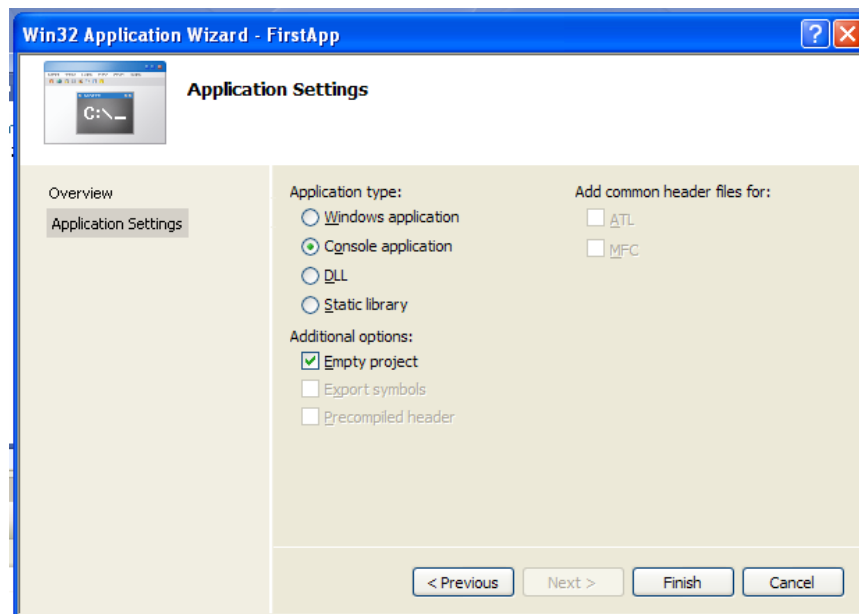
1. Создаем новый проект.



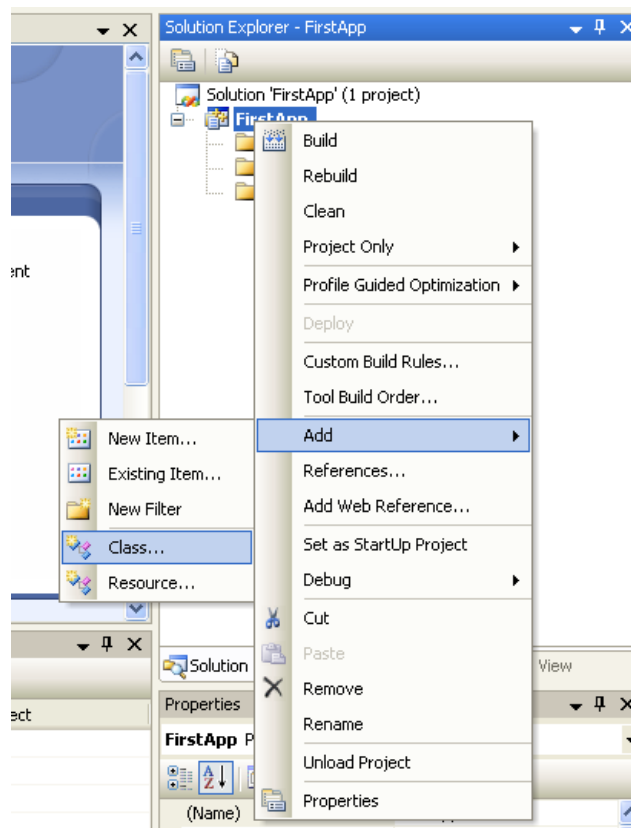
2. Выбираем тип проекта «Win32», консольное приложение «Win32 Console Application». Назначаем имя проекту.

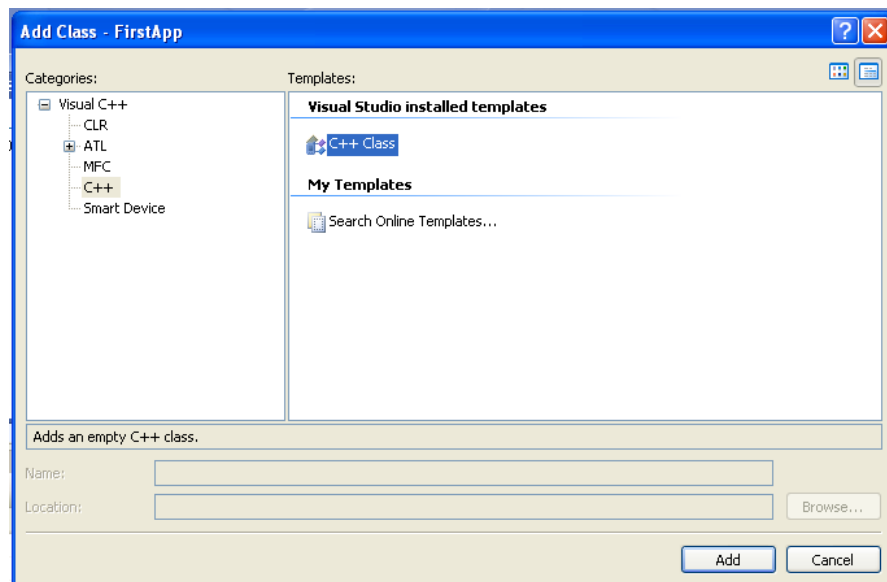


3. На вкладке «Application Settings» выбираем тип «Console Application». Проект должен быть пустым. За это отвечает галочка «Empty project».

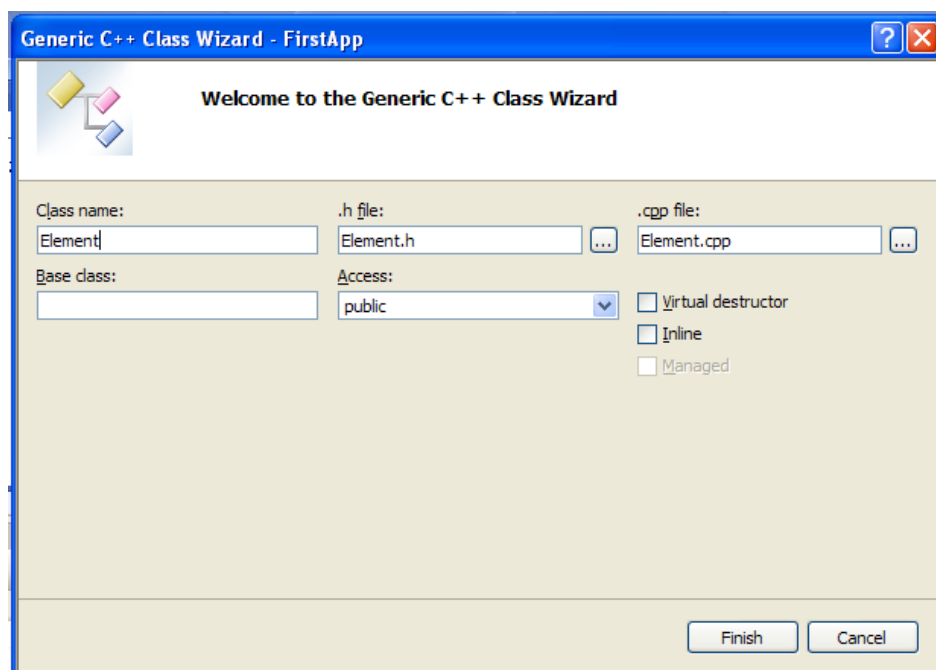


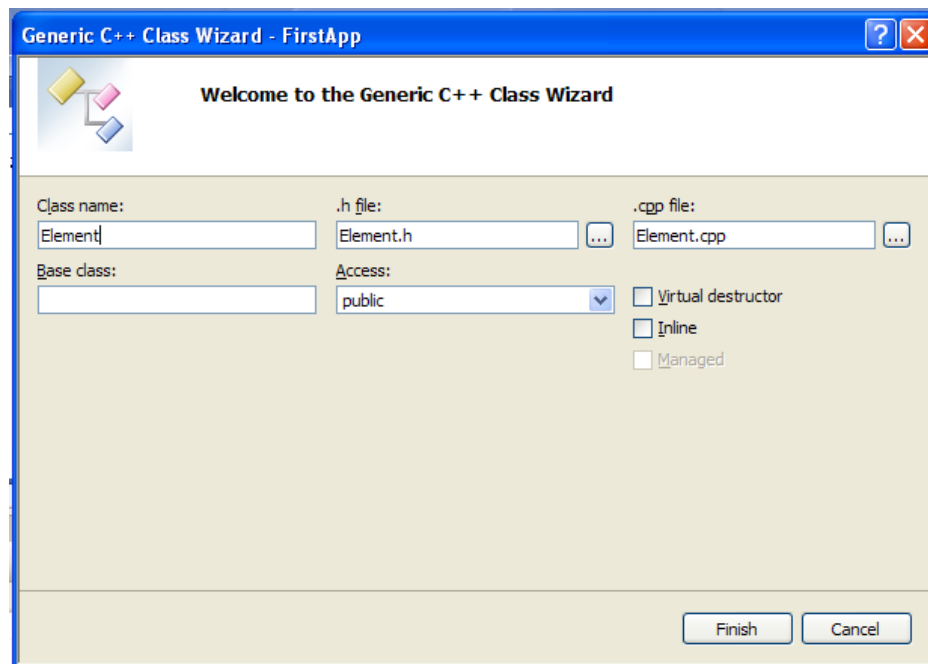
4. В окне «Solution Explorer» добавляем к проекту новый класс.



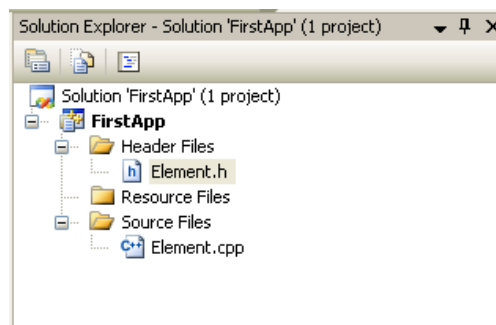


5. Даем классу имя Element. Необходимые имена файлов заполнятся автоматически.





6. Файлы Element.h и Element.cpp будут добавлены в проект.



7. Удалите из проекта файл Element.cpp, так как класс Element будет содержать только данные.

Создайте в классе необходимые данные для реализации класса Element (файл Element.h):

```
#pragma once
```

```
class Element
```

```
{
```

```
public:
```

```
    Element* prev;
```

```
    Element* next;
```

```
    int value;
```

```
    Element() { prev = NULL; next = NULL; value = 0; };
```

```
};
```

8. Создаем новый класс, отвечающий за очередь. Даем ему имя Queue. Описание класса будет размещаться в файле Queue.h, реализация – в файле Queue.cpp.
9. Добавляем в описание методы Put, Get и Print для помещения элемента в очередь, удаления из очереди и распечатки содержимого очереди. Файл Queue.h должен выглядеть так:

```
#pragma once

#include "Element.h"

class Queue
{
public:
    Element* first;
    Element* last;
    int count;

    Queue(void);
    ~Queue(void);
    void Put(Element* e);
    Element* Get();
    void Print();
};
```

10. Обратите внимание на строку #include "Element.h". Описание класса Element подключается, т.к. в классе Queue используются переменные класса Element.

Реализуем методы класса в файле Queue.cpp:

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

using namespace std;

Queue::Queue(void)
{
    first = 0;
    last = 0;
    count = 0;
```

```
}
```

```
Queue::~Queue(void)
```

```
{
```

```
}
```

```
void Queue::Put(Element* e)
```

```
{
```

```
    if (e == 0)
```

```
        return;
```

```
    e->prev = last;
```

```
    if (count < 1)
```

```
    {
```

```
        last = e;
```

```
        first = e;
```

```
    }
```

```
    last->next = e;
```

```
    last = e;
```

```
    count++;
```

```
    cout << "Element " << e->value << "added to queue" << endl;
```

```
}
```

```
Element* Queue::Get()
```

```
{
```

```
    Element* e;
```

```
    if (first == 0)
```

```
        return 0;
```

```
    e = first;
```

```
    first = first->next;
```

```
    first->prev = 0;
```



```

    e->next = 0;
    count--;

    cout << "Element " << e->value << "removed from queue" <<
endl;
    return e;
}

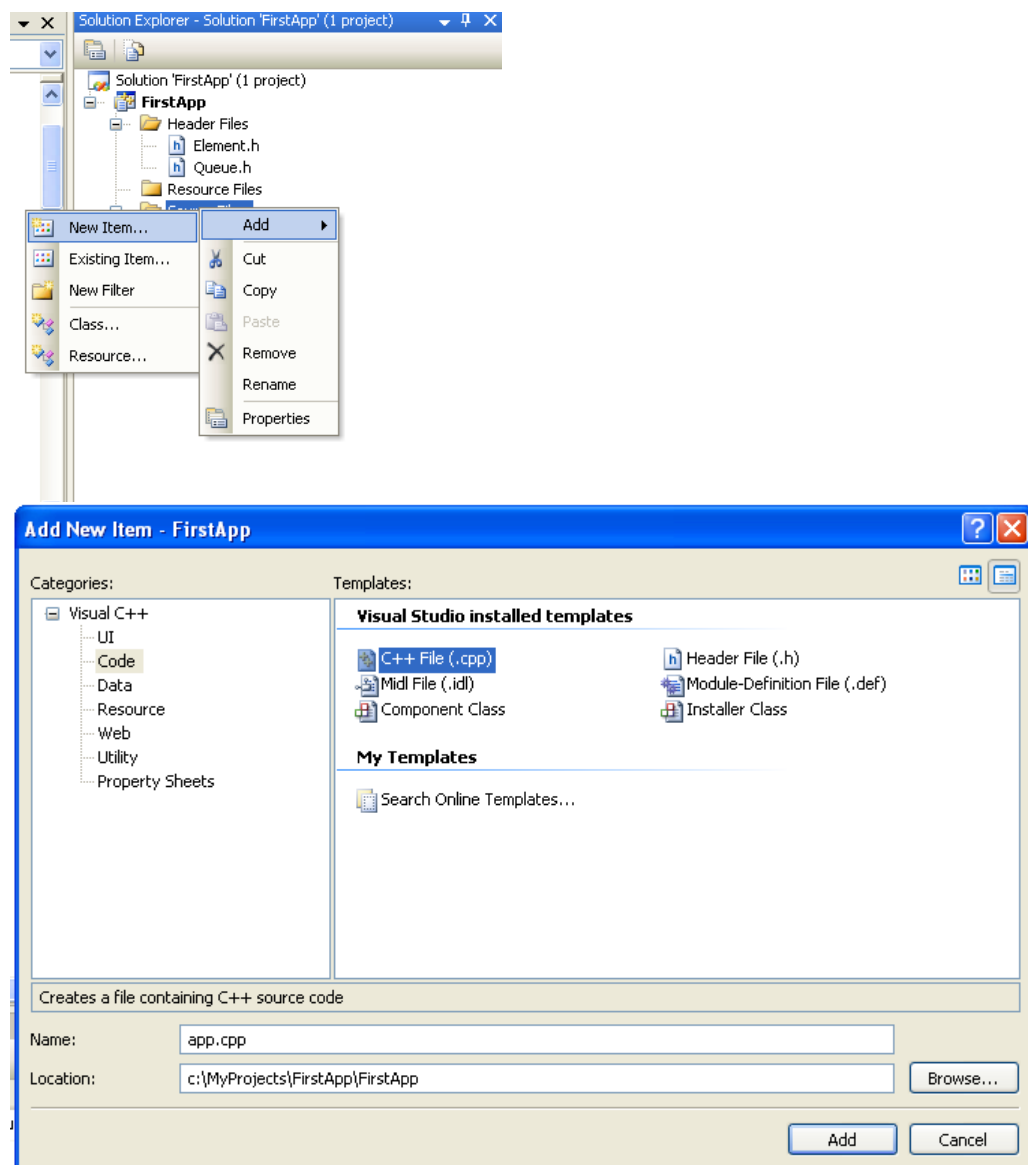
void Queue::Print()
{
    if(count < 1)
    {
        cout << "Queue is empty";
        return;
    }

    Element* e;
    e = first;
    while(e != 0)
    {
        cout << e->value << ", ";
        e = e->next;
    }
    cout << endl;
}

```

11. Обратите внимание, что при использовании cout необходимо подключить библиотеку <iostream> и объявить использование пространства имен std (using namespace std;).

12. Добавим к проекту новый файл:



Присвоим ему имя app.cpp. В этом файле будет располагаться основная программа.

13. Создадим 5 элементов и добавим их в очередь в порядке: e2, e3, e1, e5, e4.

Далее распечатываем очередь. После чего забираем 2 элемента из очереди и распечатываем очередь снова.

```
#include "Element.h"
```

```
#include "Queue.h"
```

```
#include <conio.h>
```

```
void main ()
```

```
{
```

```
    Element e1, e2, e3, e4, e5;
```

```
    e1.value = 10;
```

```
e2.value = 20;  
e3.value = 30;  
e4.value = 40;  
e5.value = 50;
```

```
Queue q;
```

```
q.Put (&e2) ;  
q.Put (&e3) ;  
q.Put (&e1) ;  
q.Put (&e5) ;  
q.Put (&e4) ;
```

```
q.Print () ;
```

```
q.Get () ;  
q.Get () ;
```

```
q.Print () ;
```

```
getch () ; }
```

14. Обратите внимание на то, что к программе подключены описания классов `Element` и `Queue`, т.к. они используются.

14. Запустите программу на выполнение. Убедитесь, что она ведет себя правильно.

15. Создайте новый проект, в котором будет реализован класс стека (`Stack`) вместо класса очереди. Создайте основную программу и покажите, что класс работает.

16. Выполните индивидуальное задание.

17. Оформите отчет по работе, описав принцип работы алгоритмов очереди и стека, поместите в отчет блок-схему реализации метода из индивидуального задания, ответьте в отчете на контрольные вопросы.

Индивидуальные задания:

1. Реализуйте в классе `Stack` метод, который выводит на экран сумму всех элементов стека.

2. Реализуйте в классе Queue метод, который выводит на экран сумму всех элементов.
3. Реализуйте в классе Stack метод, который выводит на экран произведение всех элементов.
4. Реализуйте в классе Queue метод, который выводит на экран произведение всех элементов.
5. Реализуйте в классе Stack метод, который выводит сумму первых трех элементов стека. Если элементов меньше, то сумму всех элементов.
6. Реализуйте в классе Queue метод, который выводит сумму первых трех элементов очереди. Если элементов меньше, то сумму всех элементов.
7. Реализуйте в классе Stack метод, который выводит произведение первых трех элементов стека. Если элементов меньше, то выводит сообщение о невозможности расчета.
8. Реализуйте в классе Queue метод, который выводит произведение первых трех элементов очереди. Если элементов меньше, то выводит сообщение о невозможности расчета.
9. Реализуйте в классе Stack метод, который выводит сумму всех нечетных элементов стека. Если элементов меньше, то сумму всех элементов.
10. Реализуйте в классе Queue метод, который выводит сумму всех четных элементов очереди. Если элементов меньше, то выводит сообщение о невозможности расчета.

Контрольные вопросы:

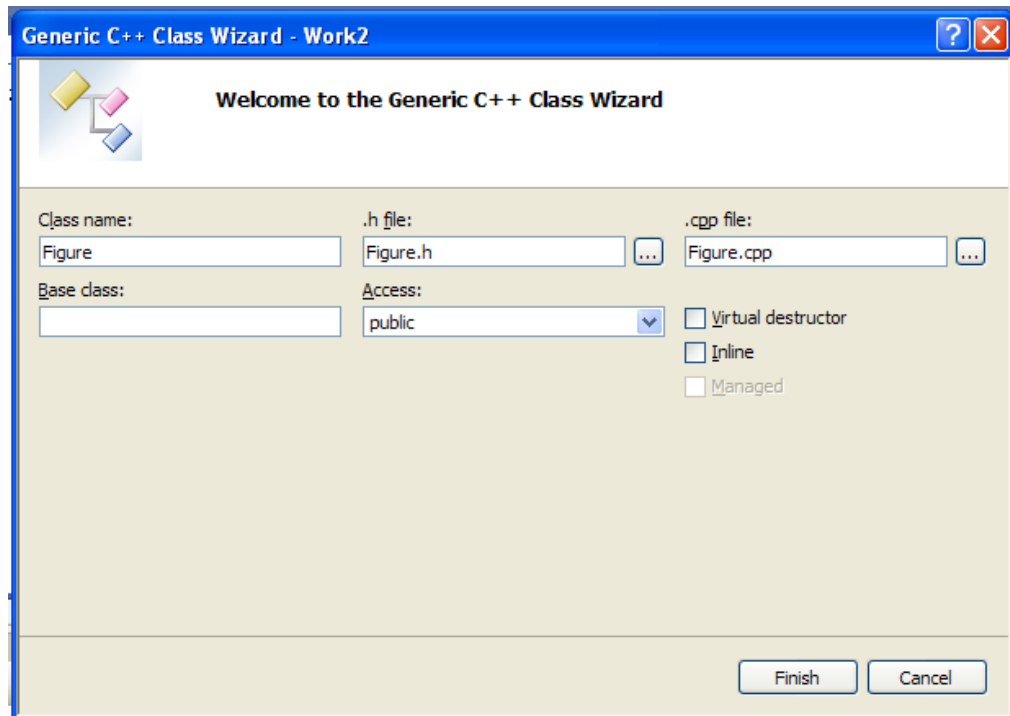
1. Нарисуйте блок-схему реализации метода Queue::Get. Объясните работу метода.
2. Нарисуйте блок-схему реализации метода Queue::Put. Объясните работу метода.
3. Нарисуйте блок-схему реализации метода Queue::Print. Объясните работу метода.
4. Нарисуйте блок-схему реализации метода Stack::Get. Объясните работу метода.
5. Нарисуйте блок-схему реализации метода Stack::Put. Объясните работу метода.
6. Нарисуйте блок-схему реализации метода Stack::Print. Объясните работу метода.

Лабораторная работа №2 «Абстрактные классы. Полиморфизм»

Цель работы: освоить основные принципы полиморфизма и позднего связывания в C++.

Ход работы:

1. Создайте новый консольный проект (см л.р. 1).
2. Добавьте в проект новый класс, отвечающий за общую структуру объекта – геометрической фигуры.



3. Кроме автоматически созданных конструктора и деструктора добавьте к классу еще 2 метода:

GetSquare – возвращающий площадь фигуры;

PrintName – печатающий на экране название фигуры.

Описание класса должно выглядеть примерно следующим образом:

```
class Figure
{
public:
    Figure(void) ;
    ~Figure(void) ;
    double GetSquare() ;
    void PrintName() ;
```

```
};
```

4. Реализуйте методы класса следующим образом:

```
Figure::Figure(void)
{
    cout << "Figure Constructor called!" << endl;
}

Figure::~~Figure(void)
{
    cout << "Figure Destructor called!" << endl;
}

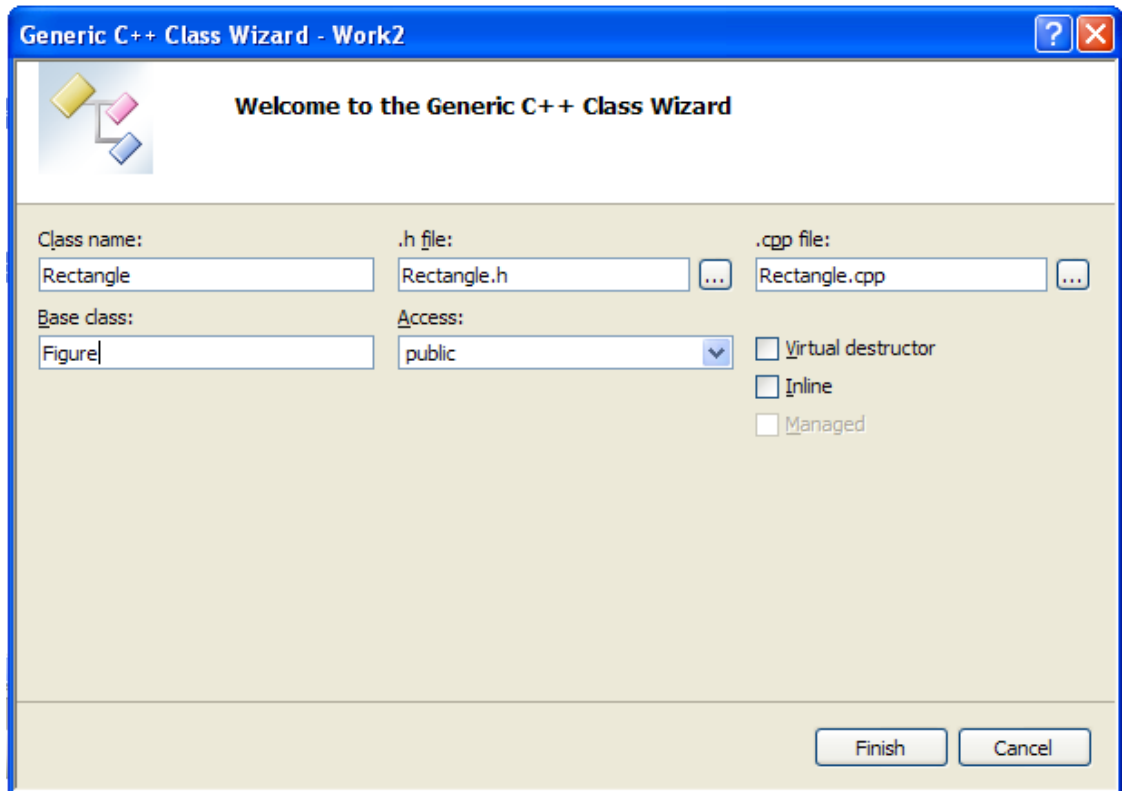
double Figure::GetSquare()
{
    cout << "Common Figure can't have a square!" << endl;
    return 0;
}

void Figure::PrintName()
{
    cout << "Abstract Figure" << endl;
}
```

5. Так как в классе Figure используется оператор cout, не забудьте добавить в файл с реализацией класса подключение библиотеки iostream:

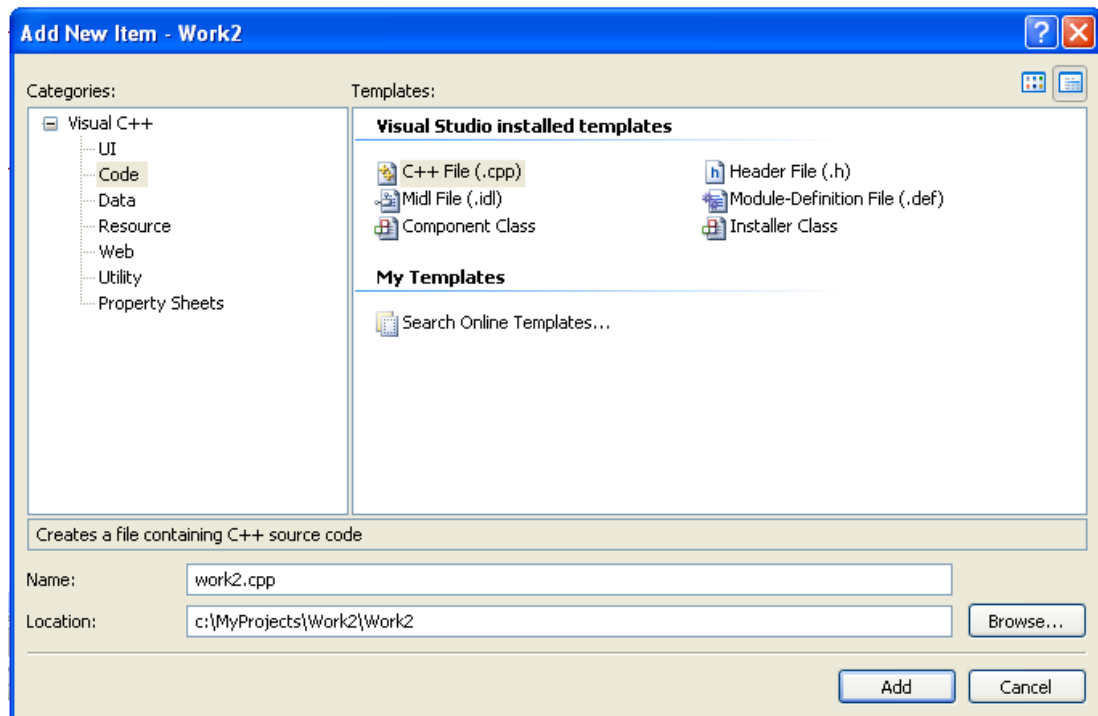
```
#include <iostream>
using namespace std;
```

6. По аналогии добавьте класс Rectangle, унаследованный от класса Figure.



В этом классе будут переопределены все методы таким образом, чтобы они сообщали об операциях с классом Rectangle, а не Figure. Что касается площади, то необходимо добавить в класс еще два поля: double a, b. Эти поля будут отвечать за длину и ширину прямоугольника. Метод GetSquare должен рассчитывать и печатать площадь.

7. Добавьте к проекту новый файл, в котором будет располагаться основная программа:



8. Создайте основную программу, в которой будут динамически создаваться объекты класса Figure и Rectangle. Далее будут вызываться различные методы, а в конце эти объекты будут уничтожаться.

```
#include <conio.h>

#include "Figure.h"
#include "Rectangle.h"

void main ()
{
    Figure* f;
    Rectangle* r;

    f = new Figure();
    r = new Rectangle();

    r->a = 10;
    r->b = 20;

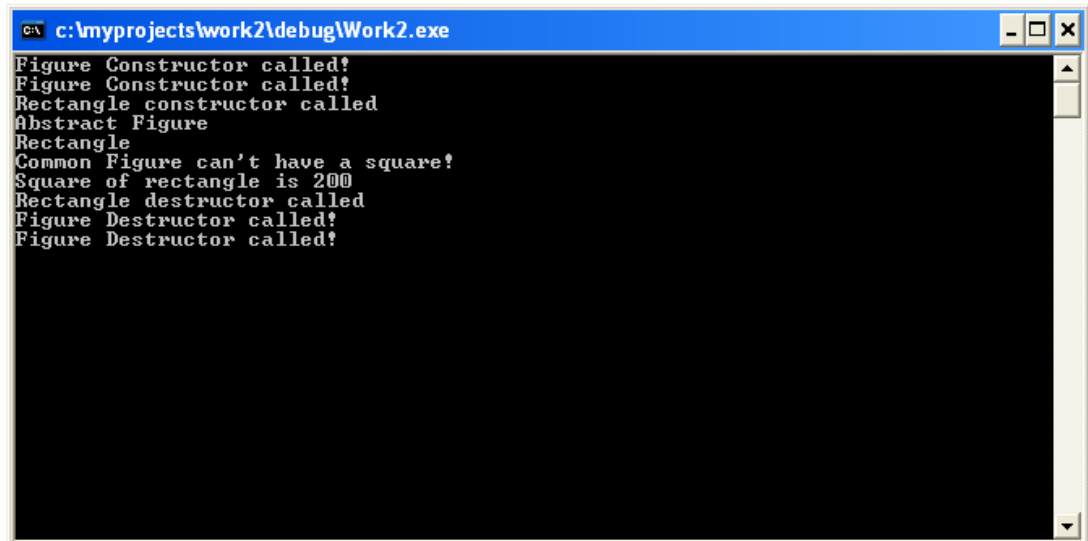
    f->PrintName();
    r->PrintName();

    f->GetSquare();
    r->GetSquare();

    delete r;
    delete f;

    getch();
}
```


9. Обратите внимание на порядок вызова методов, конструкторов и деструкторов:



```
c:\myprojects\work2\debug\Work2.exe
Figure Constructor called!
Figure Constructor called!
Rectangle constructor called
Abstract Figure
Rectangle
Common Figure can't have a square!
Square of rectangle is 200
Rectangle destructor called
Figure Destructor called!
Figure Destructor called!
```

10. Измените основную программу так, чтобы можно было понаблюдать явления, происходящие после преобразования дочернего класса Rectangle к классу Figure.

```
#include <conio.h>

#include "Figure.h"
#include "Rectangle.h"

void main ()
{
    Figure* f;
    Figure* f2;
    Rectangle* r;

    f = new Figure();
    r = new Rectangle();

    r->a = 10;
    r->b = 20;

    f2 = (Figure*)r;

    f->PrintName();
    f2->PrintName();

    f->GetSquare();
    f2->GetSquare();

    delete f;
    delete f2;

    getch();
}
```

11. Пронаблюдайте за результатом. Что изменилось. Свои наблюдения и причину изменений изложите в отчете.

12. Добавьте в описание всех методов класса Figure ключевое слово virtual, кроме методов PrintName и конструктора:

```
class Figure
{
public:
    Figure(void);
    virtual ~Figure(void);
    virtual double GetSquare();
    void PrintName();
};
```

13. Пронаблюдайте за результатом. Что изменилось. Свои наблюдения и причину изменений изложите в отчете.
14. Приведите класс Figure в полный порядок, добавив ключевое слово virtual к методу PrintName.
15. Реализуйте массив из 10 элементов типа Figure. В цикле, при помощи cin запрашивайте какую фигуру хочет добавить в массив пользователь, в зависимости от того, какая фигура выбрана, запросите ее необходимые параметры.
16. В цикле выведите площади всех фигур, добавленных в массив.

Индивидуальное задание

1. Реализуйте класс Circle для работы с кругом. В главной программе поставьте в очередь несколько кругов и прямоугольников.
2. Реализуйте класс Triangle для работы с треугольником. В главной программе поставьте в очередь несколько треугольников и прямоугольников.
3. Реализуйте класс Trapezia для работы с трапецией. В главной программе поставьте в очередь несколько трапеций и прямоугольников.
4. Реализуйте класс Romb для работы с ромбом. В главной программе поставьте в очередь несколько ромбов и прямоугольников.
5. Выполните задание №1, реализовав во всех классах вместо нахождения площади – нахождение периметра фигуры (для круга – длины окружности).
6. Выполните задание №2, реализовав во всех классах вместо нахождения площади – нахождение периметра фигуры (для круга – длины окружности).
7. Выполните задание №3, реализовав во всех классах вместо нахождения площади – нахождение периметра фигуры (для круга – длины окружности).
8. Выполните задание №4, реализовав во всех классах вместо нахождения площади – нахождение периметра фигуры (для круга – длины окружности).

Лабораторная работа №3 «Алгоритмы сортировки»

Цель работы: освоить основные алгоритмы сортировки данных. Проанализировать быстродействие алгоритмов.

Вводная часть (виды сортировок, используемых в работе):

Сортировка Шелла

Сортировка Шелла является довольно интересной модификацией алгоритма сортировки простыми вставками.

Рассмотрим следующий алгоритм сортировки массива $a[0]..a[15]$.

1. Вначале сортируем простыми вставками каждые 8 групп из 2-х элементов ($a[0]$, $a[8]$), ($a[1]$, $a[9]$), ... , ($a[7]$, $a[15]$).

2. Потом сортируем каждую из четырех групп по 4 элемента ($a[0]$, $a[4]$, $a[8]$, $a[12]$), ..., ($a[3]$, $a[7]$, $a[11]$, $a[15]$).

В нулевой группе будут элементы 4, 12, 13, 18, в первой - 3, 5, 8, 9 и т.п.

3. Далее сортируем 2 группы по 8 элементов, начиная с ($a[0]$, $a[2]$, $a[4]$, $a[6]$, $a[8]$, $a[10]$, $a[12]$, $a[14]$).

4. В конце сортируем вставками все 16 элементов.

Очевидно, лишь последняя сортировка необходима, чтобы расположить все элементы по своим местам. Так зачем нужны остальные ?

На самом деле они продвигают элементы максимально близко к соответствующим позициям, так что в последней стадии число перемещений будет весьма невелико.

Последовательность и так почти отсортирована. Ускорение подтверждено многочисленными исследованиями и на практике оказывается довольно существенным.

Единственной характеристикой сортировки Шелла является приращение - расстояние между сортируемыми элементами, в зависимости от прохода. В конце приращение всегда

равно единице - метод завершается обычной сортировкой вставками, но именно последовательность приращений определяет рост эффективности.

Использованный в примере набор ..., 8, 4, 2, 1 - неплохой выбор, особенно, когда количество элементов - степень двойки. Однако гораздо лучший вариант предложил Р.Седжвик. Его последовательность имеет вид

$$\text{inc}[s] = \begin{cases} 9 \cdot 2^s - 9 \cdot 2^{s/2} + 1, & \text{если } s \text{ четно} \\ 8 \cdot 2^s - 6 \cdot 2^{(s+1)/2} + 1, & \text{если } s \text{ нечетно} \end{cases}$$

При использовании таких приращений среднее количество операций: $O(n^{7/6})$, в худшем случае - порядка $O(n^{4/3})$.

Обратим внимание на то, что последовательность вычисляется в порядке, противоположном используемому: $\text{inc}[0] = 1$, $\text{inc}[1] = 5$, ... Формула дает сначала меньшие числа, затем все большие и большие, в то время как расстояние между сортируемыми элементами, наоборот, должно уменьшаться. Поэтому массив приращений inc вычисляется перед запуском собственно сортировки до максимального расстояния между элементами, которое будет первым шагом в сортировке Шелла. Потом его значения используются в обратном порядке.

При использовании формулы Седжвика следует остановиться на значении $\text{inc}[s-1]$, если $3 \cdot \text{inc}[s] > \text{size}$.

```
int increment(long inc[], long size) {
    int p1, p2, p3, s;

    p1 = p2 = p3 = 1;
    s = -1;
    do {
        if (++s % 2) {
            inc[s] = 8*p1 - 6*p2 + 1;
        } else {
            inc[s] = 9*p1 - 9*p3 + 1;
            p2 *= 2;
            p3 *= 2;
        }
        p1 *= 2;
    }
```

```

    } while(3*inc[s] < size);

    return s > 0 ? --s : 0;
}

void shellSort(int a[], long size) {
    long inc, i, j, seq[40];
    int s;

    // вычисление последовательности приращений
    s = increment(seq, size);
    while (s >= 0) {
        // сортировка вставками с инкрементами inc[]
        inc = seq[s--];

        for (i = inc; i < size; i++) {
            int temp = a[i];
            for (j = i-inc; (j >= 0) && (a[j] > temp); j -= inc)
                a[j+inc] = a[j];
            a[j+inc] = temp;
        }
    }
}

```

Часто вместо вычисления последовательности во время каждого запуска процедуры, ее значения рассчитывают заранее и записывают в таблицу, которой пользуются, выбирая начальное приращение по тому же правилу: начинаем с $\text{inc}[s-1]$, если $3 \cdot \text{inc}[s] > \text{size}$.

Сортировка пузырьком

Идея метода: шаг сортировки состоит в проходе снизу вверх по массиву. По пути просматриваются пары соседних элементов. Если элементы некоторой пары находятся в неправильном порядке, то меняем их местами.

После нулевого прохода по массиву "вверх" оказывается самый "легкий" элемент - отсюда аналогия с пузырьком. Следующий проход делается до второго сверху элемента, таким образом второй по величине элемент поднимается на правильную позицию... Делаем проходы по все уменьшающейся нижней части массива до тех пор, пока в ней не останется только один элемент. На этом сортировка заканчивается, так как последовательность упорядочена по возрастанию.

Качественно улучшение алгоритма можно получить из следующего наблюдения. Хотя легкий пузырек снизу поднимется вверх за один проход, тяжелые пузырьки опускаются со минимальной скоростью: один шаг за итерацию. Так что массив 2 3 4 5 6 1 будет отсортирован за 1 проход, а сортировка последовательности 6 1 2 3 4 5 потребует 5 проходов.

Чтобы избежать подобного эффекта, можно менять направление следующих один за другим проходов. Получившийся алгоритм иногда называют "**шейкер-сортировкой**".

Сортировка вставками

Сортировка простыми вставками в чем-то похожа на вышеизложенные методы.

Аналогичным образом делаются проходы по части массива, и аналогичным же образом в его начале "вырастает" отсортированная последовательность...

Однако в сортировке пузырьком или выбором можно было четко заявить, что на i -м шаге элементы $a[0] \dots a[i]$ стоят на правильных местах и никуда более не переместятся. Здесь же подобное утверждение будет более слабым: последовательность $a[0] \dots a[i]$ упорядочена. При этом по ходу алгоритма в нее будут вставляться (см. название метода) все новые элементы.

Будем разбирать алгоритм, рассматривая его действия на i -м шаге. Как говорилось выше, последовательность к этому моменту разделена на две части: готовую $a[0] \dots a[i]$ и неупорядоченную $a[i+1] \dots a[n]$.

На следующем, $(i+1)$ -м каждом шаге алгоритма берем $a[i+1]$ и вставляем на нужное место в готовую часть массива.

Поиск подходящего места для очередного элемента входной последовательности осуществляется путем последовательных сравнений с элементом, стоящим перед ним. В зависимости от результата сравнения элемент либо остается на текущем месте (вставка завершена), либо они меняются местами и процесс повторяется.

Сортировка вставками со сторожевым элементом

Алгоритм можно слегка улучшить. Заметим, что на каждом шаге внутреннего цикла проверяются 2 условия. Можно объединить их в одно, поставив в начало массива специальный сторожевой элемент. Он должен быть заведомо меньше всех остальных элементов массива.

Тогда при $j=0$ будет заведомо верно $a[0] \leq x$. Цикл остановится на нулевом элементе, что и было целью условия $j \geq 0$.

Таким образом, сортировка будет происходить правильным образом, а во внутреннем цикле станет на одно сравнение меньше. Однако, отсортированный массив будет не полон, так как из него исчезло первое число. Для окончания сортировки это число следует вернуть назад, а затем вставить в отсортированную последовательность $a[1] \dots a[n]$.

Сортировка выбором

Идея метода состоит в том, чтобы создавать отсортированную последовательность путем присоединения к ней одного элемента за другим в правильном порядке.

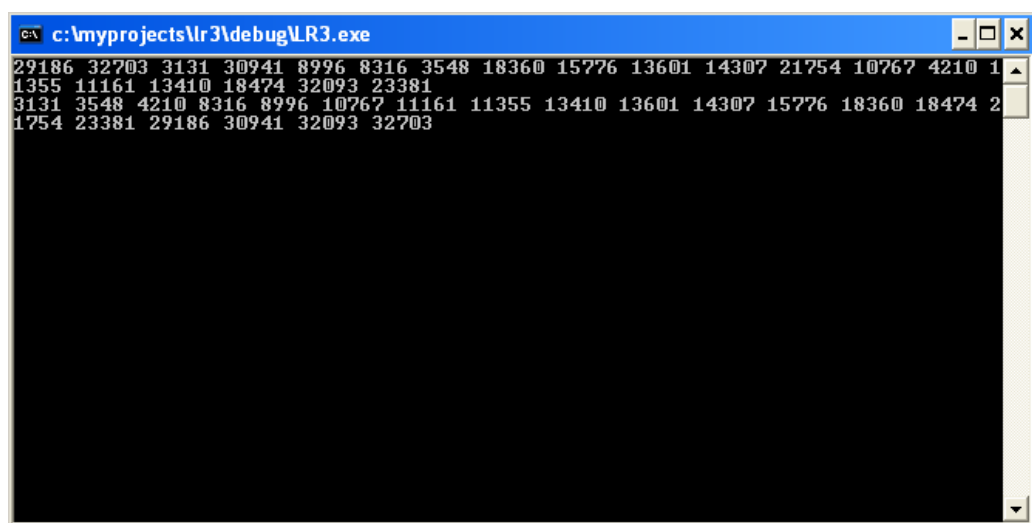
Будем строить готовую последовательность, начиная с левого конца массива. Алгоритм состоит из n последовательных шагов, начиная от нулевого и заканчивая $(n-1)$ -м.

На i -м шаге выбираем наименьший из элементов $a[i] \dots a[n]$ и меняем его местами с $a[i]$.

Вне зависимости от номера текущего шага i , последовательность $a[0] \dots a[i]$ (выделена курсивом) является упорядоченной. Таким образом, на $(n-1)$ -м шаге вся последовательность, кроме $a[n]$ оказывается отсортированной, а $a[n]$ стоит на последнем месте по праву: все меньшие элементы уже ушли влево.

Ход работы:

1. Создайте новый консольный проект (см л.р. 1).
2. Добавьте в проект сpp-файл, в котором будет располагаться функция `main()`.
3. Для удобства будем сортировать последовательности целых положительных чисел. Создайте функцию `Generate`, которая будет создавать массив заданного количества произвольных чисел от 0 до 32767, расположенных в произвольном порядке. Для генерации чисел можно воспользоваться функцией `int rand(void)` из библиотеки `stdlib.h`. Эта функция возвращает псевдослучайное число из диапазона 0 – 32767. Перед стартом цикла генерации чисел необходимо инициализировать систему псевдослучайных чисел. Сделать это можно вызовом функции `srand(GetTickCount());`
Эта функция берет текущее системное время и инициализирует им систему псевдослучайных чисел. Такая инициализация снижает вероятность повторения последовательностей.
Для работы функции `GetTickCount` необходимо подключить также библиотеку `windows.h`.
4. Создайте функцию `Print`, которая будет выводить на экран в строчку через пробел все числа заданной последовательности.
5. Реализуйте функцию сортировки Шелла, приведенную в приложении.
6. Составьте функцию `main` так, чтобы генерировалось 20 случайных чисел, затем они выводились на экран, сортировались и снова выводились на экран. Убедитесь, что сортировка работает:



```
c:\myprojects\lr3\debug\LR3.exe
29186 32703 3131 30941 8996 8316 3548 18360 15776 13601 14307 21754 10767 4210 1
1355 11161 13410 18474 32093 23381
3131 3548 4210 8316 8996 10767 11161 11355 13410 13601 14307 15776 18360 18474 2
1754 23381 29186 30941 32093 32703
```

7. Замерьте время, требуемое для сортировки. Для этого можно воспользоваться уже знакомой функцией `GetTickCount()` по следующему принципу:


```

DWORD t1, t2;
t1 = GetTickCount();
// ... Делаем что-то
t2 = GetTickCount();

```

Разница $t2-t1$ даст время процесса в миллисекундах.

8. Результатом замера скорей всего будет 0, т.к. чисел очень мало для того, чтобы сортировка заняла сколько-нибудь ощутимое время. Количество сортируемых чисел необходимо увеличить. Сделайте замеры для массива из 10000000 элементов. Обратите внимание, что, если массив задается статически в стиле `int m[10000000]`, то скорей всего произойдет ошибка переполнения стека. Для того, чтобы можно было работать с большим количеством чисел, воспользуемся динамическим выделением:

```

int *m;
m = new int[10000000];
// ... Делаем что-то
delete [] m;

```

9. Сделайте 10 замеров для количества 10000000 чисел. Посчитайте среднее время работы алгоритма сортировки. Замеры и результаты расчета отразите в отчете.
10. Сделайте такие же замеры для различного числа элементов так, чтобы по полученным данным можно было получить 5-7 точек для построения графика зависимости времени работы от количества данных. В отчете приведите таблицу с данными о замерах, о среднем времени и график зависимости.
11. Реализуйте алгоритм из индивидуального задания. Описания алгоритмов даны в приложении.
12. Проведите все необходимые замеры, приведите в отчет таблицы замеров и график на котором отражены зависимости для алгоритма Шелла и вашего алгоритма.

Индивидуальные задания:

1. Сортировка пузырьком.
2. Сортировка вставками.
3. Сортировка выбором.
4. Сортировка вставками со сторожевым элементом.
5. Шейкер-сортировка.

Лабораторная работа №4 «Алгоритмы поиска»

Цель работы: освоить основные алгоритмы поиска данных. Проанализировать быстродействие алгоритмов.

Вводная часть (виды алгоритмов поиска, используемые в работе):

Поиск значения перебором

```
int FindPos(int a[], long size, int val)
{
    For(int i=0;i<size,i++)
    {
        if(a[i] == val)
            return i;
    }
    return -1; // значение не найдено
}
```

Бинарный поиск

В бинарном поиске исходное множество должно быть упорядочено по возрастанию. Иными словами, каждый последующий ключ больше предыдущего, т. е. $\{K_1 \leq K_2 \leq K_3 \leq \dots \leq K_{n-1} \leq K_n\}$.

Отыскиваемый ключ сравнивается с центральным элементом множества, если он меньше центрального, то поиск продолжается в левом подмножестве, в противном случае — в правом.

Номер центрального элемента находится по формуле:

$$\text{№ эл-та} = \lfloor n/2 \rfloor + 1,$$

где квадратные скобки обозначают, что отделения берется только целая часть (всегда округляется в меньшую сторону). В методе бинарного поиска анализируются только центральные элементы.

Пример. Дано множество

{7, 8, 12, 16, 18, 20, 30, 38, 49, 50, 54, 60, 61, 69, 75, 79, 80, 81, 95, 101, 123, 198}.

Найти во множестве ключ $K = 61$.

Шаг 1. № эл-та = $\lfloor n/2 \rfloor + 1 = \lfloor 22/2 \rfloor + 1 = 12$.

$K \sim K2$

61 > 60. Дальнейший поиск в правом подмножестве:

{61, 69, 75, 79, 80, 81, 95, 101, 123, 198}.

Значок «~» обозначает сравнение элементов (чисел, значений).

Шаг 2. № эл-та = $\lfloor n/2 \rfloor + 1 = \lfloor 10/2 \rfloor + 1 = 6$.

$K \sim K18$

61 < 81. Дальнейший поиск в левом подмножестве

{61, 69, 75, 79, 80}

(относительно предыдущего подмножества).

Шаг 3. № эл-та = $\lfloor n/2 \rfloor + 1 = \lfloor 5/2 \rfloor + 1 = 3$.

$K \sim K15$

61 < 75. Дальнейший поиск в левом подмножестве

{61, 69}.

Шаг 4. № эл-та = $\lfloor n/2 \rfloor + 1 = \lfloor 2/2 \rfloor + 1 = 2$.

$K \sim K14$

61 < 69. Дальнейший поиск в левом подмножестве

{61}.

Шаг 5. № эл-та = $\lfloor n/2 \rfloor + 1 = \lfloor 1/2 \rfloor + 1 = 1$.

$K \sim K13$

61=61.

Вывод: искомый ключ найден под номером 13.

Фибоначчиев поиск

В этом поиске анализируются элементы, находящиеся в позициях, равных числам Фибоначчи. Числа Фибоначчи получаются по следующему правилу:

каждое последующее число равно сумме двух предыдущих чисел, например:

{ 1, 2, 3, 5, 8, 13, 21, 34, 55, ... }

Поиск продолжается до тех пор, пока не будет найден интервал между двумя ключами, где может располагаться отыскиваемый ключ.

Пример. Дано исходное множество ключей
{3, 5, 8, 9, 11, 14, 15, 19, 21, 22, 28, 33, 35, 37, 42, 45, 48, 52}.

Пусть отыскиваемый ключ равен 42 ($K = 42$).

Последовательное сравнение отыскиваемого ключа будет проводиться в позициях, равных числам Фибоначчи: {1, 2, 3, 5, 8, 13, 21, ...}.

Шаг 1. $K \sim K_1$, $42 > 3 \Rightarrow$ отыскиваемый ключ сравнивается с ключом, стоящим в позиции, равной числу Фибоначчи.

Шаг 2. $K \sim K_2$, $42 > 5 \Rightarrow$ сравнение продолжается с ключом, стоящим в позиции, равной следующему числу Фибоначчи.

Шаг 3. $K \sim K_3$, $42 > 8 \Rightarrow$ сравнение продолжается.

Шаг 4. $K \sim K_5$, $42 > 11 \Rightarrow$ сравнение продолжается.

Шаг 5. $K \sim K_8$, $42 > 19 \Rightarrow$ сравнение продолжается.

Шаг 6. $K \sim K_{13}$, $42 > 35 \Rightarrow$ сравнение продолжается.

Шаг 7. $K \sim K_{18}$, $42 < 52 \Rightarrow$ найден интервал, в котором находится отыскиваемый ключ: от 13 до 18 позиции, т. е. {35, 37, 42, 45, 48, 52}.

В найденном интервале поиск вновь ведется в позициях, равных числам Фибоначчи.

Интерполяционный поиск

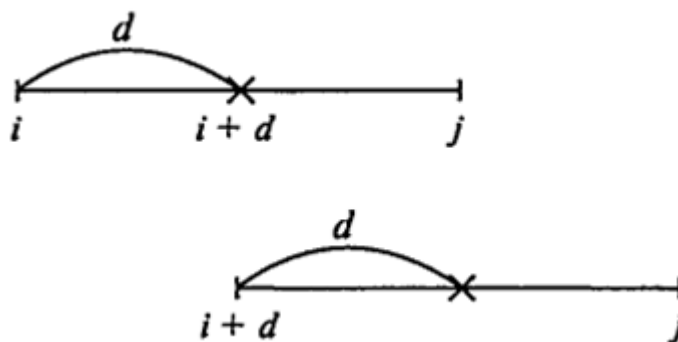
Исходное множество должно быть упорядочено по возрастанию весов.

Первоначальное сравнение осуществляется на расстоянии шага d который определяется по формуле:

$$d = \left[\frac{(j-i)(K - K_i)}{K_j - K_i} \right],$$

где i — номер первого рассматриваемого элемента; j — номер последнего рассматриваемого элемента; K — отыскиваемый ключ; K_i , K_j — значения ключей в i и j позициях; $[]$ — целая часть от числа.

Идея метода заключается в следующем: шаг d меняется после каждого этапа по формуле, приведенной выше. Алгоритм заканчивает работу при $d = 0$, при этом анализируются соседние элементы, после чего делается окончательное решение о результатах поиска.



Этот метод прекрасно работает, если исходное множество представляет собой арифметическую прогрессию или множество, приближенное к ней.

Пример. Дано множество ключей:

{2, 9, 10, 12, 20, 24, 28, 30, 37, 40, 45, 50, 51, 60, 65, 70, 74, 76}.

Пусть искомым ключ равен 70 ($K = 70$).

Шаг 1. Определим шаг d для исходного множества ключей:

$$d = [(18 - 1)(70 - 2)/(76 - 2)] = 15.$$

Сравниваем ключ, стоящий под шестнадцатым порядковым номером в данном множестве, с искомым ключом:

$K_{16} \sim K, 70 = 70$, ключ найден.

Ход работы:

1. Создайте новый консольный проект (см л.р. 1).
2. Добавьте в проект сpp-файл, в котором будет располагаться функция `main()`.
3. Для удобства будем производить поиск в последовательности целых положительных чисел. Создайте функцию `Generate`, которая будет создавать массив заданного количества произвольных чисел от 0 до 32767, расположенных в произвольном порядке. Для генерации чисел можно воспользоваться функцией `int rand(void)` из библиотеки `stdlib.h`. Эта функция возвращает псевдослучайное число из диапазона 0 — 32767.

Перед стартом цикла генерации чисел необходимо инициализировать систему псевдослучайных чисел. Сделать это можно вызовом функции `srand(GetTickCount());`

Эта функция берет текущее системное время и инициализирует им систему псевдослучайных чисел. Такая инициализация снижает вероятность повторения последовательностей.

Для работы функции `GetTickCount` необходимо подключить также библиотеку `windows.h`.

4. Создайте функцию `Print`, которая будет выводить на экран в строчку через пробел все числа заданной последовательности.
5. Реализуйте функцию поиска заданного значения перебором (приведенную в приложении), возвращающую позицию найденного элемента или сообщаящую, что искомого элемента нет.
6. Составьте функцию `main` так, чтобы генерировалось 20 случайных чисел, затем они выводились на экран, с клавиатуры запрашивался элемент для поиска, производился поиск и выводилась на экран позиция найденного элемента. Убедитесь, что поиск работает.
7. Замерьте время, требуемое для поиска. Для этого можно воспользоваться уже знакомой функцией `GetTickCount()` по следующему принципу:

```
DWORD t1, t2;  
t1 = GetTickCount();  
// ... Делаем что-то  
t2 = GetTickCount();
```

Разница `t2-t1` даст время процесса в миллисекундах.

8. Результатом замера скорее всего будет 0, т.к. чисел очень мало для того, чтобы поиск занял сколько-нибудь ощутимое время. Количество чисел в выборке для поиска необходимо увеличить. Сделайте замеры для массива из 10000000 элементов. Обратите внимание, что, если массив задается статически в стиле `int m[10000000]`, то скорее всего произойдет ошибка переполнения стека. Для того, чтобы можно было работать с большим количеством чисел, воспользуемся динамическим выделением:

```
int *m;  
m = new int[10000000];  
// ... Делаем что-то  
delete [] m;
```

9. Сделайте 10 замеров для количества 10000000 чисел. Посчитайте среднее время работы алгоритма поиска. Замеры и результаты расчета отразите в отчете.

10. Сделайте такие же замеры для различного числа элементов так, чтобы по полученным данным можно было получить 5-7 точек для построения графика зависимости времени работы от количества данных. В отчете приведите таблицу с данными о замерах, о среднем времени и график зависимости.
11. Реализуйте алгоритм из индивидуального задания. Описания алгоритмов даны в приложении.
12. Проведите все необходимые замеры, приведите в отчет таблицы замеров и график на котором отражены зависимости для алгоритма последовательного поиска и вашего алгоритма. Замеры производите таким образом, чтобы можно было отразить в отчете время, требуемое на весь алгоритм поиска (включая сортировку), а также время непосредственно поиска (исключая сортировку).

Индивидуальные задания:

1. Бинарный поиск. Сортировка пузырьком.
2. Бинарный поиск. Сортировка вставками.
3. Бинарный поиск. Сортировка Шелла.
4. Бинарный поиск. Сортировка выбором.
5. Фибоначиев поиск. Сортировка пузырьком.
6. Фибоначиев поиск. Сортировка вставками.
7. Фибоначиев поиск. Сортировка Шелла.
8. Фибоначиев поиск. Сортировка выбором.
9. Интерполяционный поиск. Сортировка пузырьком.
10. Интерполяционный поиск. Сортировка вставками.
11. Интерполяционный поиск. Сортировка Шелла.
12. Интерполяционный поиск. Сортировка выбором.