

Учреждение образования
«БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ
ТЕХНОЛОГИЧЕСКИЙ УНИВЕРСИТЕТ»

Н. В. Пацей

**Объектно-ориентированное
программирование**

**Лабораторный практикум
по одноименной дисциплине**

В 2-частях

Часть 1

Минск БГТУ 2015

Завести аккаунт на сайте <https://github.com/> и осуществить первый коммит.

Регистрация на сайте заключается в выборе имени пользователя и пароля. Заполните информацию о себе на странице вашего профиля. Можно фото.

Создайте репозиторий для вашего первого проекта (можно загрузить проект по ОАП). Для этого нужно перейти по <https://github.com/new>, выбрать название репозитория (произвольным образом), описание и режим доступа "Public". Галочку напротив "Initialize this repository with a README" ставить не нужно.

Воспользуйтесь разделом "...or create a new repository on the command line" из подсказки, чтобы инициализировать репозиторий и создать в нем файл README.md

Модифицируйте файл README.md так, чтобы он содержал ваше имя, номер группы и специальность.

Ссылку на получившийся репозиторий со всеми изменениями нужно прислать преподавателю.

Разобраться с принципами работы с проектами на GitHub и все последующие лабораторные выкладывать туда для проверки и анализа динамики работы.

№ 1-2 Программирование объектов и классов

Определить пользовательский класс в соответствии с вариантом задания (смотри варианты ниже).

- 1) определить в классе следующие конструкторы: без параметров, с параметрами, копирования;**
- 2) определить в классе деструктор, компоненты-функции для просмотра и установки полей данных: `setТип()`, `getТип()` (помним про инкапсуляцию и проверку корректности задания параметров);**
- 3) написать демонстрационную программу, в которой создаются и разрушаются объекты (от 3 до 5 шт.) и указатели на объекты пользовательского класса и каждый вызов конструктора и деструктора сопровождается выдачей соответствующего сообщения (какой объект какой конструктор или деструктор вызвал).**

Вариант 1	<p>Создать класс Вектор, который имеет указатель на int, число элементов и переменную состояния. Определить функцию, которая присваивает элементу массива некоторое значение (параметр по умолчанию), функцию, которая получает некоторый элемент массива. В переменную состояния устанавливать код ошибки, при нехватке памяти, выхода за пределы массива и т.п.. Определить функции: вывода консоль, сложения и умножения, которые производят эти арифметические операции с данными класса вектор и числом int.</p> <p><i>Создать массив объектов. Вывести:</i></p> <p><i>a) список векторов, содержащих 0;</i></p> <p><i>b) список векторов с наименьшим модулем.</i></p>
Вариант 2	<p>Создать класс Стек вещественных. Функции-члены вставляют элемент в стек, вытаскивают элемент из стека, печатают стек, проверяет вершину стека.</p> <p><i>Создать массив объектов. Вывести:</i></p> <p><i>a) стеки с наименьшим/наибольшим верхним элементом;</i></p> <p><i>b) список стеков, содержащих отрицательные элементы.</i></p>
Вариант 3	<p>Создать класс типа - Множество. Функции-члены: добавляют элемент к множеству, удаляют элемент, отображают элементы множества от начала и от конца, выводят текущее количество элементов множества.</p> <p><i>Создать массив объектов. Вывести:</i></p> <p><i>a) множества с наименьшей/наибольшей суммой элементов;</i></p> <p><i>b) список множеств, содержащих отрицательные элементы.</i></p> <p><i>c) списки равных множеств.</i></p>
Вариант 4	<p>Создать класс типа - Дата с полями: день (1-31), месяц (1-12), год (целое число). Функции-члены класса: установки дня, месяца и года; получения дня, месяца и года, а также две функции-члены печати: печать по шаблону: "5 января 1997 года" и "05/01/1997". Функции-члены установки полей класса и конструкторы должны проверять корректность задаваемых параметров.</p> <p><i>Создать массив объектов. Вывести:</i></p> <p><i>a) список дат для заданного года;</i></p> <p><i>b) список дат, которые имеют заданное число;</i></p> <p><i>c) список дат, у которых сумма всех чисел соответствует степени двойки.</i></p>

Вариант 5	<p>Создать класс Строка (динамическая). Функции-члены класса: вывода строки, вывода длины строки, проверки существует ли в строке заданный символ, функция замены одного символа в строке на другой.</p> <p><i>Создать массив объектов. Вывести:</i></p> <p><i>a) список строк определенной длины;</i></p> <p><i>b) список строк, которые содержат заданное слово.</i></p>
Вариант 6	<p>Определить класс Булева матрица (BoolMatrix). Реализовать методы для логического сложения (дизъюнкции), умножения и инверсии матриц. Реализовать методы для подсчета числа единиц в матрице и упорядочения строк в лексикографическом порядке.</p> <p><i>Создать массив объектов. Вывести:</i></p> <p><i>a) список матриц с наибольшим/наименьшим количеством единиц;</i></p> <p><i>b) список матриц с равным количеством нулей в каждой строке</i></p>
Вариант 7	<p>Создать класс типа - Окружность. Поля – координаты центра, радиус. Функции-члены вычисляют площадь, длину окружности, устанавливают поля и возвращают значения. Функции-члены установки полей класса должны проверять корректность задаваемых параметров.</p> <p><i>Создать массив объектов. Вывести:</i></p> <p><i>a) группы окружностей, центры которых лежат на одной прямой;</i></p> <p><i>b) наибольший и наименьший по площади (периметру) объект;</i></p> <p><i>c) список окружностей, которые лежат в первой четверти координатной плоскости.</i></p>
Вариант 8	<p>Создать класс типа - Прямоугольник. Поля - высота и ширина. Функции-члены вычисляют площадь, периметр, устанавливают поля и возвращают значения. Функции-члены установки полей класса должны проверять корректность задаваемых параметров.</p> <p><i>Создать массив объектов. Вывести:</i></p> <p><i>a) количество четырехугольников разного типа (квадрат, прямоугольник, ромб, произвольный)</i></p> <p><i>b) определить для каждой группы наибольший и наименьший по площади (периметру) объект.</i></p>

Вариант 9	<p>Определить класс Вектор. Реализовать методы для вычисления модуля вектора, скалярного произведения, сложения, вычитания, умножения на константу.</p> <p><i>Написать метод, который для заданной пары векторов будет определять, являются ли они коллинеарными или ортогональными.</i></p> <p><i>Создать массив объектов. Вывести:</i></p> <p><i>a) вектора с заданным модулем;</i></p> <p><i>b) определить вектор с наибольшей/наименьшей суммой элементов.</i></p>
Вариант 10	<p>Построить класс Булев вектор (BoolVector). Реализовать методы для выполнения поразрядных конъюнкции, дизъюнкции и отрицания векторов, а также подсчета числа единиц и нулей в векторе.</p> <p><i>Создать массив объектов. Вывести:</i></p> <p><i>a) вектора с заданным числом единиц/нулей;</i></p> <p><i>b) определить и вывести равные вектора.</i></p>
Вариант 11	<p>Создать класс типа - Время с полями: часы (0-23), минуты (0-59), секунды (0-59). Класс имеет конструктор. Функции-члены установки времени, получения часа, минуты и секунды, а также две функции-члены печати: по шаблону: "16 часов 18 минут 3 секунды" и "4:18:3". Функции-члены установки полей класса должны проверять корректность задаваемых параметров.</p> <p><i>Создать массив объектов. Вывести:</i></p> <p><i>a) время с заданным числом часов;</i></p> <p><i>b) список времен по группам: ночь, утро, день, вечер.</i></p>
Вариант 12	<p>Создать класс - Одномерный массив целых чисел (вектор). Функции-члены обращаются к отдельному элементу массива по индексу(ввод-вывод значений), вывода значений массива на экран, поэлементного сложения и вычитания со скалярным значением.</p> <p><i>Создать массив объектов. Вывести:</i></p> <p><i>a) массивы только с четными/нечетными элементами;</i></p> <p><i>b) массив с наибольшей суммой элементов.</i></p>

Вариант 13	<p>Создать класс – Треугольник, заданного тремя точками. Функции-члены изменяют точки, обеспечивают вывод на экран координат, рассчитывают длины сторон и периметр треугольника.</p> <p><i>Создать массив объектов.</i></p> <p><i>a) подсчитать количество треугольников разного типа (равносторонний, равнобедренный, прямоугольный, произвольный).</i></p> <p><i>b) определить для каждой группы наибольший и наименьший по периметру объект.</i></p>
Вариант 14	<p>Создать класс - Множество. Функции-члены реализуют добавление и удаление элемента, пересечение и разность множеств, вывод множества.</p> <p><i>Создать массив объектов. Вывести:</i></p> <p><i>a) множества только с четными/нечетными элементами;</i></p> <p><i>b) множества, содержащие отрицательные элементы.</i></p>
Вариант 15	<p>Создать класс Airline: Пункт назначения, Номер рейса, Тип самолета, Время вылета, Дни недели. Функции-члены реализуют запись и считывание полей (проверка корректности).</p> <p><i>Создать массив объектов. Вывести:</i></p> <p><i>a) список рейсов для заданного пункта назначения;</i></p> <p><i>b) список рейсов для заданного дня недели;</i></p> <p><i>c) список рейсов для заданного дня недели, время вылета для которых больше заданного.</i></p>
Вариант 16	<p>Создать класс Student: id, Фамилия, Имя, Отчество, Дата рождения, Адрес, Телефон, Факультет, Курс, Группа. Функции-члены реализуют запись и считывание полей (проверка корректности), расчет возраста студента</p> <p><i>Создать массив объектов. Вывести:</i></p> <p><i>a) список студентов заданного факультета;</i></p> <p><i>b) списки студентов для каждого факультета и курса;</i></p> <p><i>c) список студентов, родившихся после заданного года;</i></p> <p><i>d) список учебной группы.</i></p>
Вариант 17	<p>Создать класс Customer: id, Фамилия, Имя, Отчество, Адрес, Номер кредитной карточки, баланс. Функции-члены реализуют запись и считывание полей (проверка корректности), зачисление и списание сумм на счет.</p> <p><i>Создать массив объектов. Вывести:</i></p> <p><i>a) список покупателей в алфавитном порядке;</i></p> <p><i>b) список покупателей, у которых номер кредитной карточки находится в заданном интервале.</i></p>

Вариант 18	<p>Создать класс Abiturient: id, Фамилия, Имя, Отчество, Адрес, Телефон, Оценки. Функции-члены реализуют запись и считывание полей (проверка корректности), расчнта среднего балла.</p> <p><i>Создать массив объектов. Вывести:</i></p> <p><i>a) список абитуриентов, имеющих неудовлетворительные оценки;</i></p> <p><i>b) список абитуриентов, у которых сумма баллов выше заданной;</i></p> <p><i>c) выбрать заданное число n абитуриентов, имеющих самую высокую сумму баллов (вывести также полный список абитуриентов, имеющих полупроходную сумму).</i></p>
Вариант 19	<p>Создать класс Book: id, Название, Автор (ы), Издательство, Год издания, Количество страниц, Цена, Тип переплета. Функции-члены реализуют запись и считывание полей (проверка корректности).</p> <p><i>Создать массив объектов. Вывести:</i></p> <p><i>a) список книг заданного автора;</i></p> <p><i>b) список книг, выпущенных заданным издательством;</i></p> <p><i>c) список книг, выпущенных после заданного года.</i></p>
Вариант 20	<p>Создать класс House: id, Номер квартиры, Площадь, Этаж, Количество комнат, Улица, Тип здания, Срок эксплуатации. Функции-члены реализуют запись и считывание полей (проверка корректности), расчета возраста здания (необходимость в кап. ремонте).</p> <p><i>Создать массив объектов. Вывести:</i></p> <p><i>a) список квартир, имеющих заданное число комнат;</i></p> <p><i>b) список квартир, имеющих заданное число комнат и расположенных на этаже, который находится в заданном промежутке;</i></p> <p><i>c) список квартир, имеющих площадь, превосходящую заданную.</i></p>
Вариант 21	<p>Создать класс Phone: id, Фамилия, Имя, Отчество, Адрес, Номер кредитной карточки, Дебет, Кредит, Время городских и междугородных разговоров. Функции-члены реализуют запись и считывание полей (проверка корректности), расчет балланса на текущий момент.</p> <p><i>Создать массив объектов. Вывести:</i></p> <p><i>a) сведения об абонентах, у которых время внутригородских разговоров превышает заданное;</i></p> <p><i>b) сведения об абонентах, которые пользовались междугородной связью;</i></p> <p><i>c) сведения об абонентах в алфавитном порядке.</i></p>

Вариант 22	<p>Создать класс Car: id, Марка, Модель, Год выпуска, Цвет, Цена, Регистрационный номер. Функции-члены реализуют запись и считывание полей (проверка корректности), вывода возраста машины.</p> <p>Создать массив объектов. Вывести:</p> <p>a) список автомобилей заданной марки;</p> <p>b) список автомобилей заданной модели, которые эксплуатируются больше n лет;</p> <p>c) список автомобилей заданного года выпуска, цена которых больше указанной.</p>
Вариант 23	<p>Создать класс Product: id, Наименование, UPC, Производитель, Цена, Срок хранения, Количество. Функции-члены реализуют запись и считывание полей (проверка корректности), вывод общей суммы продукта.</p> <p>Создать массив объектов. Вывести:</p> <p>a) список товаров для заданного наименования;</p> <p>b) список товаров для заданного наименования, цена которых не превосходит заданную;</p> <p>c) список товаров, срок хранения которых больше заданного.</p>
Вариант 24	<p>Создать класс Train: Пункт назначения, Номер поезда, Время отправления, Число мест (общих, купе, плацкарт, люкс). Функции-члены реализуют запись и считывание полей (проверка корректности), вывод общего числа мест в поезде.</p> <p>Создать массив объектов. Вывести:</p> <p>a) список поездов, следующих до заданного пункта назначения;</p> <p>b) список поездов, следующих до заданного пункта назначения и отправляющихся после заданного часа;</p> <p>c) список поездов, отправляющихся до заданного пункта назначения и имеющих общие места.</p>
Вариант 25	<p>Создать класс Bus: Фамилия и инициалы водителя, Номер автобуса, Номер маршрута, Марка, Год начала эксплуатации, Пробег. Функции-члены реализуют запись и считывание полей (проверка корректности), вывод возраста автобуса.</p> <p>Создать массив объектов. Вывести:</p> <p>a) список автобусов для заданного номера маршрута;</p> <p>b) список автобусов, которые эксплуатируются больше заданного срока;</p> <p>c) список автобусов, пробег у которых больше заданного расстояния.</p>

Вариант 26	<p>Создать класс Airline: Пункт назначения, Номер рейса, Тип самолета, Время вылета, Дни недели. Функции-члены реализуют запись и считывание полей (проверка корректности).</p> <p>Создать массив объектов. Вывести:</p> <p>a) список рейсов для заданного пункта назначения;</p> <p>b) список рейсов для заданного дня недели;</p> <p>c) список рейсов для заданного дня недели, время вылета для которых больше заданного.</p>
Вариант 27	<p>Определить класс Определенный интеграл с аналитической подынтегральной функцией. Создать методы для вычисления значения по формуле левых прямоугольников, по формуле правых прямоугольников, по формуле средних прямоугольников, по формуле трапеций, по формуле Симпсона (параболических трапеций).</p>

Примеры программ

Матрица

Matrix.h

```
#pragma once
#include <iostream>
#include <ctime>

class Matrix
{
    int **storage;
    int hsize,vsize;
    static const int maxsize=20;
public:
    Matrix(); //
    Matrix(int v,int h); //
    Matrix(const Matrix &); //
    ~Matrix(); //
    void sethsize(int); //
    void setvsize(int); //
    int gethsize(); //
    int getvsize(); //
    void printall(); //
    void print(int,int); //
    void checksize(); //
    void randomfill(); //
};
```

Matrix.cpp

```
#include "StdAfx.h"
#include "matrix.h"
```

```

Matrix::Matrix()
{
    std::cout << "Вызов конструктора по умолчанию\n";
    storage=new int*[maxsize];
    for(int i=0;i<maxsize;i++)
        storage[i]=new int[maxsize];
    hsize=vsize=maxsize;
    this->randomfill();
}

Matrix::Matrix(int v,int h)
{
    std::cout << "Вызов конструктора с параметрами\n";
    storage=new int*[maxsize];
    for(int i=0;i<maxsize;i++)
        storage[i]=new int[maxsize];
    hsize=h;
    vsize=v;
    this->randomfill();
}

Matrix::Matrix(const Matrix& m)
{
    std::cout << "Вызов конструктора копирования\n";
    storage=new int*[maxsize];
    for(int i=0;i<maxsize;i++)
        storage[i]=new int[maxsize];
    hsize=m.hsize;
    vsize=m.vsize;
    for(int i=0;i<maxsize;i++)
        for(int j=0;j<maxsize;j++)
            storage[i][j]=m.storage[i][j];
}

Matrix::~Matrix()
{
    std::cout << "Вызов деструктора\n";
    for(int i=0;i<maxsize;i++)
        delete[] storage[i];
    delete[] storage;
}

int Matrix::gethsize() {return hsize;}

int Matrix::getvsize() {return vsize;}

void Matrix::randomfill()
{
    srand(time(NULL));
    for(int i=0;i<hsize;i++)
        for(int j=0;j<vsize;j++)
            if(storage[i][j]<0) storage[i][j]=rand()%10;
}

```

```

}

void Matrix::printall()
{
    for(int i=0;i<hsize;i++)
    {
        for(int j=0;j<vsize;j++)
            std::cout << storage[i][j] << " ";
        std::cout << std::endl;
    }
    std::cout << std::endl;
}

void Matrix::checksize()
{
    for(int i=0;i<maxsize;i++)
        for(int j=0;j<maxsize;j++)
            if(i>=hsize||j>=vsize) storage[i][j]=-1;
}

void Matrix::sethsize(int h)
{
    hsize=h;
    this->checksize();
    this->randomfill();
}

void Matrix::setvsize(int v)
{
    vsize=v;
    this->checksize();
    this->randomfill();
}

void Matrix::print(int v,int h)
{
    for(int i=0;i<h;i++)
    {
        for(int j=0;j<v;j++)
            std::cout << storage[i][j] << " ";
        std::cout << std::endl;
    }
    std::cout << std::endl;
}

```

Main_module.cpp

```

#include "stdafx.h"
#include "matrix.h"
#include <iostream>

```

```

#include <locale>
using namespace std;

int _tmain(int argc, _TCHAR* argv[])
{
    setlocale(LC_ALL, ".1251");
    cout << "Матрица a\n";
    Matrix *a=new Matrix(4,5);
    cout << "Матрица b\n";
    Matrix *b=new Matrix(*a);
    cout << "Матрица a\n";
    a->printall();
    cout << "Изменение размеров\n";
    a->setvsize(6);
    a->sethsize(6);
    cout << "Матрица a\n";
    a->printall();
    cout << "Изменение размеров\n";
    a->setvsize(3);
    a->sethsize(3);
    cout << "Матрица a\n";
    a->printall();
    cout << "Матрица b\n";
    b->printall();
    return 0;
}

```

Вектор

MyVector.h

```

#pragma once
#define NIL          0x00000000
#define NULL         0

class CMyVector
{
private:
    int m_size;
    int* m_vector;
    int m_lastError;
public:
    enum {no_error, memory_limit, exit_array};
    CMyVector(void);
    CMyVector(int size, ...);
    ~CMyVector(void);
    int getLastError(void);
    int getSize(void);
    int operator[] (int);
    void print(void);
    int sum(void);
    int product(void);
}

```

```
};
```

MyVector.cpp

```
#include "MyVector.h"
```

```
#include <iostream>
```

```
CMyVector::CMyVector(void)
```

```
{
    m_vector = NIL;
    m_size = 0;
    m_lastError = no_error;
}
```

```
CMyVector::CMyVector(int size, ...)
```

```
{
    m_vector = new int[size];
    m_size = size;
    int* marker = &size;
    marker++;
    while (size != 0)
    {
        m_vector[m_size-size] = *marker;
        marker++;
        size--;
    }
    m_lastError = no_error;
}
```

```
CMyVector::~~CMyVector(void)
```

```
{
    delete [] m_vector;
}
```

```
int CMyVector::getLastError(void)
```

```
{
    int vp = m_lastError;
    m_lastError = no_error;
    return vp;
}
```

```
int CMyVector::getSize(void)
```

```
{
    return m_size;
}
```

```
int CMyVector::operator[] (int index)
```

```
{
    if (index<0 || index>m_size-1)
    {
        m_lastError = exit_array;
        return NULL;
    }
    return m_vector[index];
}
```

```

void CMyVector::print(void)
{
    for (int i = 0; i < m_size; i++) std::cout << m_vector[i]
<< std::ends;
    std::cout << std::endl;
}

int CMyVector::sum(void)
{
    int s = 0;
    for (int i = 0; i < m_size; i++) s += m_vector[i];
    return s;
}

int CMyVector::product(void)
{
    int p = 1;
    for (int i = 0; i < m_size; i++) p *= m_vector[i];
    return p;
}

```

Main_mod.cpp

```

#include "MyVector.h"

int main()
{
    CMyVector v1(5, 1, 2, 3, 4, 5);
    CMyVector v2(3, 1, 2, 3);
    int x = v1[7];
    int error = v1.getLastError();
    x = v1[2];
    x = v1.getSize();
    x = v2.sum();
    x = v2.product();
    v2.print();
    return 0;
}

```

Студент

Student.h

```

#pragma once
#include "string.h"
class Student
{
    char name[30];
    int age;
public:
    Student();
    Student(char*, int);
    void setAge(int);
    void setName(char*);
}

```

```

        ~Student(void);
        void print();
        char* getName(void);
        int getAge(void);
};

```

Student.cpp

```

Student::Student()
{
}
Student::Student(char* NAME, int AGE)
{
    strcpy(name, NAME);
    age = AGE;
}
void Student::setAge(int AGE)
{
    age = AGE;
}
void Student::setName(char *NAME)
{
    strcpy(name, NAME);
}
Student::~~Student(void)
{
}
void Student::print()
{
    cout << "Name - " << Student::name << endl;
    cout << "Age - " << Student::age << endl;
}
char* Student::getName(void)
{
    return name;
}
int Student::getAge(void)
{
    return age;
}

```

Main_mod.cpp

```

#include "stdafx.h"
#include "student.h"
#include <iostream>
using namespace std;

int _tmain(int argc, _TCHAR* argv[])
{
    Student a1, a2;
    a1.setName("Oleg");
}

```



```

    a1.setAge(17);
    a2.setName("Olya");
    a2.setAge(18);
    char* q = a1.getName();
    int w = a1.getAge();
    a1.print();
    a2.print();
    return 0;
}

```

Теория

Объект - структурированная переменная, содержащая всю информацию о некотором физическом предмете или реализуемом в программе понятии.

Класс - описание множества таких объектов и выполняемых над ними действий.

Классы

Классы C++ предусматривают создание расширенной системы предопределенных типов. Каждый тип класса представляет уникальное множество объектов, операций над ними, а также операций, используемых для создания, манипулирования и уничтожения таких объектов. С классами связан ряд новых понятий.

Наследование. Новый, или производный класс может быть определен на основе уже имеющегося, или базового. При этом новый класс сохраняет все свойства старого: данные объекта базового класса включаются в данные объекта производного, а методы базового класса могут быть вызваны для объекта производного класса.

Полиморфизм. Он основывается на возможности включения в данные объекта также и информации о методах их обработки (в виде указателей на функции). Будучи доступным в некоторой точке программы, даже при отсутствии полной информации о типе объекта, он всегда может корректно вызвать свойственные ему методы. *Полиморфной* называется функция, определенная в нескольких производных классах и имеющая в них общее имя.

Для определения класса используются три ключевых слова: `struct`, `union` и `class`. Каждый класс включает в себя функции - называемые методами, компонентными функциями или функциями членами (`member function`), и данные - элементы данных (`class members`)

Имя класса становится идентификатором нового типа данных и может использоваться для объявления объектов.

С точки зрения синтаксиса, класс в C++ - это структурированный тип, образованный на основе уже существующих типов. Класс можно определить с помощью формы:

```
тип_класса имя_класса <:базовый_класс>
    {список_членов_класса или список компонентов};
```

где тип_класса - одно из служебных слов `class`, `struct`, `union`;

имя_класса - идентификатор;

необязательный параметр базовый_класс - содержит класс или классы, из которого имя_класса заимствует элементы и методы, при необходимости спецификаторы доступа;

список_членов_класса - определения и описания типизированных данных и принадлежащих классу функций.

Пример:

```
class STUDENT    {
public:
    char name[25];           // имя
    int age;                 // возраст
    float grade;             // рейтинг
    char * getName() ;       //метод - получить имя
    int getAge() const;      //метод - получить возраст
    float getGrade() const;  //метод - получить рейтинг
    void setName(char*);     //метод - установить имя
    void setAge(int);        //метод - установить возраст
    void show(); };          //метод - печать
```

Тело элемента-функции может быть определено в самом классе или в определении класса дается только прототип функции (заголовок с перечислением типов формальных параметров), а определение самой функции дается отдельно, при этом полное имя функции имеет вид:

имя_класса::имя_функции

Для структурированной переменной вызов функции - элемента этой структуры имеет вид:

имя_переменной.имя_функции (список_параметров);

Для описания объекта класса (экземпляра класса) используется форма:

имя_класса имя_объекта;

Пример:

```

STUDENT my_friend, me;
STUDENT *point = &me;           // указатель на объект STUDENT
STUDENT group [30];             // массив объектов
STUDENT &girl = my_friend;      // ссылка на объект

```

Обращаться к данным объекта или компонентам класса и вызывать функции для объекта можно двумя способами. Первый с помощью «квалифицированных» имен:

имя_объекта. имя_данного
имя_объекта. имя_функции

Пример:

```

STUDENT x1, x2;
x1.age = 20;
x1.grade = 2.3;
x2.set("Петрова", 18, 25.5);
x1.show ();

```

Второй способ доступа использует указатель на объект:

указатель_на_объект->имя_компонента

Пример:

```

STUDENT *point = &x1;  // или point = new STUDENT;
point ->age = 22;
point ->grade = 7.3;
    point ->Show();

```

Доступность компонентов класса

В рассмотренных примерах компоненты классов являются общедоступными. В любом месте программы, где «видно» определение класса, можно получить доступ к компонентам объекта класса. Тем самым не выполняется основной принцип абстракции данных - инкапсуляция (сокрытие) данных внутри объекта. Для изменения видимости компонент в определении класса можно использовать спецификаторы доступа: `public`, `private`, `protected`.

Общедоступные, `public` компоненты доступны в любой части программы. Они могут использоваться любой функцией как внутри данного класса, так и вне его. Доступ извне осуществляется через имя объекта:

имя_объекта.имя_члена_класса

ссылка_на_объект.имя_члена_класса
указатель_на_объект->имя_члена_класса

Собственные, `private` компоненты локализованы в классе и не доступны извне. Они могут использоваться функциями - членами данного класса и функциями-друзьями того класса, в котором они описаны.

Защищенные, `protected` компоненты доступны внутри класса и в производных классах.

По умолчанию для `class` все элементы являются собственными или частными, т.е. `private`.

Пример:

```
class STUDENT{
    char name[25];                // private по умолчанию
    int age;
    float grade;
    public:
    void setName(char*);
    void setAge(int);};
```

Спецификатор доступа действителен пока не встретится другой спецификатор доступа, они могут пересекаться и группироваться:

```
class STUDENT{
    char name[25];
    int age;
    float grade;
    private
    protected:
    void setName(char*);
    public:
    void setAge(int);
}Me;

Me.age=18;                // ошибка
Me->setAge(18);           // правильно
```

Создание и уничтожение объектов

Недостатком рассмотренных ранее классов является отсутствие автоматической инициализации создаваемых объектов. Для каждого вновь создаваемого объекта необходимо вызвать функцию типа `Set` либо явным образом присваивать значения данным объекта.

Элементы-функции, неявно вызываемые при создании и уничтожении объектов класса называются *конструкторами* и *деструкторами*. Они определяются как элементы-функции с именами, совпадающими с именем класса. Конструкторов для данного класса может быть сколь угодно много, если они отличаются формальными параметрами, деструктор же всегда один и имеет имя, предваренное символом "~".

Конструктор

Формат определения конструктора:

имя_класса(список_форм_параметров){операторы_тела_конструктора}

```
class STUDENT{
    char name[25];
    int age;
    float grade;
public:
    STUDENT();                // конструктор без параметров
    STUDENT(char*,int,float); // конструктор с параметрами
    STUDENT(const STUDENT&);   // конструктор копирования
    ...};
    STUDENT::STUDENT(char*NAME,int AGE,float GRADE)
    {
        strcpy(name,NAME); age=AGE; grade=GRADE;
        cout<< "\nКонструктор с параметрами вызван для объекта
        <<this<<endl;
    }
```

Момент вызова конструктора и деструктора определяется временем создания и уничтожения объектов. Конструкторы обладают некоторыми уникальными свойствами:

- конструктор имеет то же имя, что и сам класс;
- для конструктора не определяется тип возвращаемого значения (даже тип `void` не допустим);
- конструкторы не наследуются, хотя производный класс может вызывать конструкторы базового класса;
- конструкторы могут иметь аргументы по умолчанию или использовать списки инициализации элементов;
- конструкторы не могут быть описаны с ключевыми словами `virtual`, `static`, `const`, `mutable`, `volatile`;
- нельзя работать с их адресами;
- если он не был задан явно, то генерируется компилятором;
- конструктор нельзя вызывать как обычную функцию;

- конструктор автоматически вызывается при определении или размещении в памяти с помощью неявного вызова оператора new каждого объекта класса и delete;

- конструктор выделяет память для объекта и инициализирует данные-члены класса.

По умолчанию создается общедоступный (public) конструктор без параметров и конструктор копирования. Если конструктор описан явно, то конструктор по умолчанию не создается. Параметром конструктора не может быть его собственный класс, но может быть ссылка на него (T&). Без явного указания программиста конструктор всегда автоматически вызывается при определении (создании) объекта. Для явного вызова конструктора используются две формы:

имя_класса имя_объекта (фактические_параметры);

имя_класса (фактические_параметры);

Первая форма допускается только при не пустом списке фактических параметров. Вторая форма вызова приводит к созданию объекта без имени. Существуют два способа инициализации данных объекта с помощью конструктора: передача значений параметров в тело конструктора; применение списка инициализаторов данного класса. Каждый инициализатор списка относится к конкретному компоненту и имеет вид:

имя_данного (выражение)

При определении объекта будет запущен тот конструктор, с которым совпадает заданный список аргументов.

Конструктор копирования (вида T::T(const T&), где T – имя класса) вызывается всякий раз, когда выполняется копирование объектов, принадлежащих классу. Он вызывается:

а) когда объект передается функции по значению:

```
void view(STUDENT a){a.show;}
```

б) при построении временного объекта как возвращаемого значения функции:

```
STUDENT noName(STUDENT & student)
{STUDENT temp(student);
 temp.setName("NoName");
 return temp;
}
STUDENT c=noName(a);
```

в) при использовании объекта для инициализации другого объекта:

```
STUDENT a("Иванов",19,50), b=a;
```

Если класс не содержит явным образом определенного конструктора копирования, то при возникновении одной из этих трех ситуаций производится побитовое копирование объекта (не во всех случаях является адекватным).

Деструктор

Динамическое выделение памяти для объекта создает необходимость освобождения этой памяти при уничтожении объекта. Такую возможность обеспечивает специальный компонент класса – *деструктор*. Его формат: **~имя_класса(){операторы_тела_деструктора}**

Свойства деструктора:

- имя деструктора совпадает с именем его класса, но предваряется символом “~” (тильда);
- деструктор не имеет параметров и возвращаемого значения;
- вызов деструктора выполняется не явно (автоматически), как только объект класса уничтожается.

Если в классе деструктор не определен явно, то компилятор генерирует деструктор по умолчанию, который просто освобождает память, занятую данными объекта. Так же, как и для конструктора, не может быть определен указатель на деструктор.

Указатели на компоненты-функции

Можно определить указатель на компоненты-функции:

тип_возвр_значения(имя_класса::*имя_указателя_на_функцию)
(специф_параметров_функции);

Пример:

```
void (STUDENT::*pf)();  
pf=&STUDENT::show;  
(p[1].*pf)();
```

Указатель this

Когда функция-член класса вызывается для обработки данных конкретного объекта, этой функции автоматически и неявно передается указатель на тот объект, для которого функция вызвана. Этот указатель имеет имя *this* и неявно определен в каждой функции класса следующим образом:

имя_класса *const this = адрес_объекта

Указатель `this` является дополнительным скрытым параметром каждой нестатической компонентной функции. При входе в тело принадлежащей классу функции `this` инициализируется значением адреса того объекта, для которого вызвана функция. В результате этого объект становится доступным внутри этой функции.

В большинстве случаев использование `this` является неявным. В частности, каждое обращение к нестатической функции-члену класса неявно использует `this` для доступа к члену соответствующего объекта. Примером широко распространенного явного использования `this` являются операции со связанными списками.

Вставляемые (inline) функции

Если функция (обычная или функция-элемент класса) объявлена `inline`-функциями, то при вызове таких функций транслятор выполняет подстановку по тексту программы тела функции с соответствующей заменой формальных параметров на фактические. Функция-элемент также считается `inline` по умолчанию, если ее тело находится непосредственно в определении класса, например:

```
struct dat
{int      day,month,year;
void      setDat(char *p)      // Функция inline по умолчанию
    { ... }                  // тело функции
};
```

Функцию-член можно описать как `inline` вне описания класса. Например:

```
char Char_stack {
    int size;
    char* top;
    char* s;
public:
    char pop();
    // ... };
inline char Char_stack::pop()
{    return *--top; }
```

Друзья классов

Иногда требуются исключения из правил доступа, когда некоторой функции или классу требуется разрешить доступ к личной части объекта класса. Тогда в определении класса, к объектам которого разрешается такой доступ, должно быть объявление функции или другого класса как «дружественных». Объявление дружественной функции представляет собой прототип функции, объявление переопределяемой операции или имя класса, которым разрешается доступ, с ключевым словом `friend` впереди. Общая схема объявления такова:


```

class A
{
    int x; // Личная часть класса
    ...
friend class B; // Функции класса B дружественны A
friend void C::fun(A&);
    // Элемент-функция fun класса C имеет доступ к приватной части A
friend void xxx(A&,int); // Функция xxx дружественна классу A
friend void C::operator+(A&); // Переопределяемая в классе C операция
    }; // <объект C>+<объект A> дружественна классу A
    class B // Необходим доступ к личной части A
    {public: int fun1(A&);
      void fun2(A&); };
class C
{ public: void fun(A&);
  void operator+(A&); };

```

К средствам контроля доступа относятся также объявления функций-элементов постоянными (`const`). В этом случае они не имеют права изменять значение текущего объекта, с которым вызываются. Заголовок такой функции имеет вид:

```
void Dat::put() const { ... }
```

По аналогии с функциями и методами можно объявить дружественными весь класс. При этом все методы такого дружественного класса смогут обращаться ко всем компонентами класса:

```

class MyClass
{... friend class MyFriend; ...}

```

Дружба классов не транзитивна и односторонняя.

Статические элементы класса

Иногда требуется определить данные, которые относятся ко всем объектам класса. Например, контроль общего количества объектов класса или одновременный доступ ко всем объектам или части их, разделение объектами общих ресурсов. Тогда в определение класса могут быть введены статические элементы-переменные. Такой элемент сам в объекты класса не входит, зато при обращении к нему формируется обращение к общей статической переменной с именем. Доступность ее определяется стандартным образом в зависимости от размещения в личной или общей части класса. Сама переменная должна быть явно определена в программе модификатором `static` и инициализирована.

Статическими могут быть объявлены методы класса. Их вызов не связан с конкретным объектом и может быть выполнен по полному имени. Соответственно в них не используются неявный указатель на текущий объект

this. Статические методы не могут быть константными (const) и виртуальными (virtual). Например:

```
class List
{ ...
static void show();           // статическая функция просмотра
                              // списка объектов
static void List::show()
{List *p;
for (p=fst; p !=NULL; p=p->next)
    { ...вывод информации об объекте... }}
void main()
{ ... List::show(); }         // вызов функции по полному имени
```

Приложение . Стандарт оформления кода

При выполнении задания ознакомьтесь с правилами написания кода (Code convention). Указанные здесь требования не являются единственно верными для написания кода. Но практически всегда в любой команде разработки существует свой стиль, которого придерживаются все члены группы.

Указанные здесь требования являются производными от широко распространённого стандарта, с которым вам стоит ознакомиться в оригинале:

Стандарт кодирования GNU http://www.opennet.ru/docs/RUS/coding_standard/

Google C++ Style Guide <https://google-styleguide.googlecode.com/svn/trunk/cppguide.html>

C++ Programming Style Guidelines <http://geosoft.no/development/cppstyle.html>

а также почитать на эту тему:

Стандарт оформления кода <https://ru.wikipedia.org/wiki>

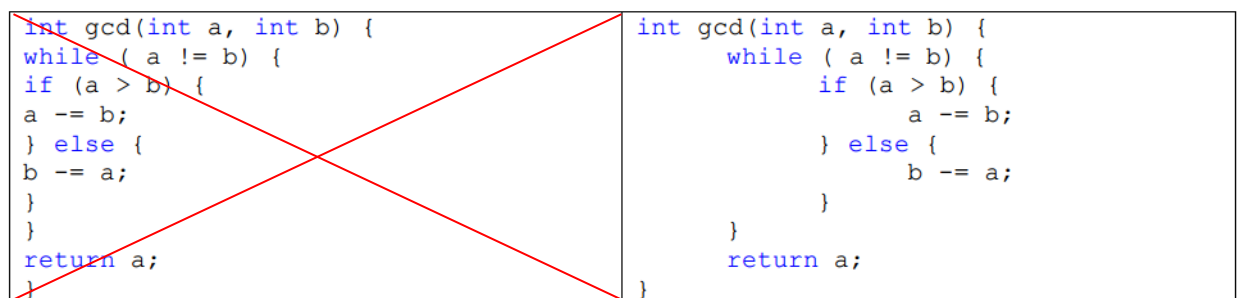
<http://habrahabr.ru/post/194752/> и др.

Стиль программирования — это набор правил и соглашений, используемых при написании исходного кода на некотором языке программирования. Наличие общего стиля программирования облегчает понимание и поддержание исходного кода, написанного больше, чем одним программистом, а также облегчает сотрудничество нескольких человек в развитии одного программного обеспечения. Следование правилам хорошего стиля программирования значительно уменьшает вероятность появления ошибок на этапе набора текста, делает программу легко читаемой, что, в свою очередь, облегчает процессы отладки и внесения изменений.

Отступы

Для всего проекта должен применяться единый стиль отступов. Всегда будьте последовательны в использовании отступов и применяйте их для повышения читабельности кода.

Отступы ставятся везде, где есть фигурные скобки. То есть в теле функций, циклах (do, while, for), операторах if и switch.



Фигурные скобки

Всегда используйте фигурные скобки, даже если их можно опустить. Сегодня имеется четыре наиболее распространенных стиля расстановки фигурных скобок, используемых в C-сообществе. Можно использовать любой из них

1. Стилль K&R или 1TBS (One True Bracing Style) Назван в честь Brian W. Kernighan и Dennis M. Ritchie.

```
if (condition) {
    content;
}
```

2. Стилль Алмена Используется по умолчанию в Microsoft Visual Studio (и более ранних продуктах).

```
if (condition)
{
    content;
}
```

3. Стилль GNU Используется во всех проектах GNU. Отступы всегда 4 символа на уровень, скобки находятся на половине отступа.

```
if (condition)
{
    content;
}
```

Каждый блок примитива управления потоком («if», «else», «while», «for», «do», «switch») должен заключаться в скобки, даже если он содержит только одну строку, или не содержит ничего вообще.

Сокращённая запись часто приводит к ошибкам, которые тяжело вычислить, т.к. если такой блок кода потребуется расширить, то наличие скобок уже становится обязательным, о чём не всегда вспоминает программист, дописавший код.

В булевских выражениях («if», «for», «while», «do» и первом операнде тернарного оператора "?") всегда записывайте равенство и неравенство в явном виде.

Избегайте условных выражений, которые всегда имеют один и тот же результат (за исключением платформозависимого кода, когда результат на другой платформе может все-таки быть иным).

Длина строки

Длина строки не должна превышать 80 символов. Выражения длиннее 80 символов разделяются на части, при этом последующие части должны быть короче первой и сдвинуты вправо (чтобы схожие логические объекты находились на одной вертикально прямой). Те же правила применяются к функциям с длинным списком аргументов и текстовым строкам.

```
void fun(int a, int b, int c) {
    if (condition) {
        printf("Warning this is a long "
               "printf with 3 parameters a: %u b: %u "
               "c: %u \n", a, b, c);
    } else {
        next_statement;
    }
}
```

Когда Вы пишете конструкцию if-else, которая вложена в другую конструкцию if, всегда следует помещать скобки вокруг if-else.

Комментарии

Комментарии нужны для облегчения чтения кода, но при неправильном использовании этого средства можно только усложнить читабельность кода. Например, комментируя те моменты, которые и так всем понятны. Не пытайтесь в комментариях объяснить, как работает программа, код должен сам говорить за себя. Т.е. он должен быть как можно более понятным и прозрачным. Комментарии должны пояснять, что делает программа, а не как она это делает. Имена переменных и функций Существует несколько стилей названия переменных

Сокращения

Стоит пользоваться упрощёнными конструкциями:

Инкремент

Конструкция вида: b[j] = <выражение>; j++;	Может быть упрощена до: b[j++] = <выражение>;
--	--

лишняя переменная

Конструкция вида: A a = <выражение>; return a;	Может быть упрощена до: return <выражение>;
--	--

лишняя else-ветка

Конструкция вида: if (<условие>) { return true; } else { return false; }	Может быть упрощена до: return <условие>;
---	--

лишнее использование условного оператора

Конструкция вида: if (<условие>) { return true; } return false;	Может быть упрощена до: return <условие>;
---	--

формирование переменной внутри условного оператора

Конструкция вида: A a; if (<условие>) {	Может быть упрощена до: if (<условие>) { return <выражение1>;
---	---

<pre> a = <выражение1>; } else { a = <выражение2>; } return a; </pre>	<pre> } return <выражение2>; </pre>
---	---

Имена переменных, типов и функций

Существует несколько стилей названия переменных

var_bell – стиль C: нижний регистр, знак подчёркивания.

VarBell – стиль Pascal: каждая подстрока в названии начинается с большой буквы.

varBell – стиль Java: первая строка начинается с маленькой буквы, все последующие с большой.

Не имеет значения, какой стиль будет вами выбран — главное придерживаться в коде одного стиля

Константы традиционно записываются в верхнем регистре (например, **YANDEX_BOT**).

При выборе имени переменной не так важна длина имени, как понятность. Длинные имена могут назначаться глобальным переменным, которые редко используются, но индексы массивов, появляющиеся в каждой строке цикла, не должны быть значительно сложнее, чем **i**. Использование **index** или **elementnumber** не только усложняет набор, но и может сделать менее понятными детали вычислений.

Примеры плохих и хороших (понятных) имен.

<pre> num1 do_this() g() hxq </pre>	<pre> numberOfCharacters number_of_chars num_chars get_number_of_chars() is_char_limit() character_max() charMax() i (loop value) </pre>
-------------------------------------	--

Называйте переменные, функции и файлы со строчной буквы, а свои типы (в т.ч. классы) с заглавной.

```

struct ListNode
{
    int value;
    ListNode* next;
};

int getNextValue();

void printoutList();

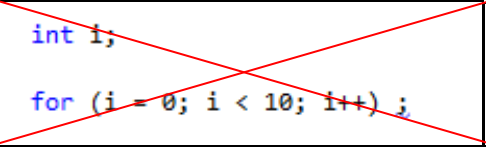
```

Приватные переменные к классу используют в качестве суффикса нижнее подчеркивание:

```
class SomeClass
{
    private:
    int length_;
}
```

Числа

Не используйте "магические константы" (опять же, не следует увлекаться).

 <pre>int i; for (i = 0; i < 10; i++) ;</pre>	<pre>int const dimension = 10; for (i = 0; i < dimension; i++)</pre>
---	---

Функции

Функции-предикаты (булевы функции) следует именовать следующим образом:

[глагол][предикат]

где "глагол" - глагол бытия (to be) или обладания (to have) в соответствующей форме, а "предикат" - проверяемое свойство. Например:

<pre>bool isPrime(int n) ; bool hasColor();</pre>

При вызове функций-предикатов запрещается сравнивать результат с логическими константами.

if (isPrime(n) == true) –плохой код.

Сравнивать логические переменные с логическими константами столь же бессмысленно.

Не используйте транслит, пишите все либо по-русски в читабельной в компьютерном классе кодировке, либо по-английски. Например, `vstavka()` - неудачное название для функции, реализующей алгоритм сортировки вставками.

Все методы, которые не изменяют видимое извне состояние объекта, должны быть определены константными.

<pre>string getName () const { return name; }</pre>

Вывод

Каждый вывод данных на экран должен сопровождаться сообщением с пояснением того, что именно выводится.

Классы

Классы следует называть с большой буквы, слова в названии не должны иметь разделителей и должны состоять только из латинских букв и, при необходимости, цифр. Если название содержит несколько слов, то каждое следующее слово должно начинаться с большой буквы.

```
class PrimeNumbers {///  
};
```

Члены класса следует определять в порядке, соответствующем уровню доступа: сначала «public», потом «protected», последними «private».

```
class Student  
{  
public:  
  
    Student (string a, int b, int c) { ... }  
    Student () { ... }  
protected: int number;  
  
private: string name;  
         int score;  
         int scriptedlesson;  
};
```

Экземпляры класса стоит определять напрямую конструктором, а не копирующим конструктором через присвоение.

Вопросы

1. Какие ключевые слова можно использовать при определении класса?
2. В чем отличие между объектом и классом?
3. Что такое конструктор?
4. Что такое деструктор (destructor) ?
5. Что такое дружественная функция (friend function)?
6. Что такое копирующий конструктор (copy constructor)?
7. Продолжите фразу «... является специальной функцией-членом, используемой для задания начальных значений данным, членам класса».
8. Продолжите фразу «по умолчанию доступ к членам класса ...».
9. Продолжите фразу «говорят, что реализация класса скрыта от его клиентов или ...».
10. Продолжите фразу «набор открытых функций-членов класса рассматривается как ... класса».
11. Продолжите фразу «в определении класса члены класса с ключевым словом `private` доступны ...».
12. Напишите определение класса `SS`, включающего одно закрытое поле типа `int` с именем `bar` и одним открытым методом с прототипом `void Print()`.
13. Истинно ли следующее утверждение: поля класса должны быть закрытыми.
14. Продолжите фразу «операция точки (операция доступа к члену класса) объединяет следующие два элемента (слева направо)».
15. Конструктор вызывается автоматически в момент ... объекта.
16. Верно или неверно следующее утверждение: класс может иметь более одного конструктора с одним и тем же именем.
17. Найдите ошибку и объясните, как ее исправить. Допустим, что в классе `Time` объявлен следующий прототип:

```
void ~Time( int )
```
18. Функция, не являющаяся членом, которая должна иметь доступ к закрытым данным-членам класса, должна быть объявлена как ... этого класса.
19. Определите класс `CBOX`, объявите массив объектов `CBOX`.
20. Определите класс `CBOX`, объявите статический член класса `int`. Выполните его инициализацию. Сколько экземпляров статического данного будет существовать, если число объектов класса равно 5.
21. Пусть есть класс.

```
class List
{ // ...
```

```

static void show();
// Статическая функция просмотра списка объектов
}
static void List::show()
{
List *p;
for (p=fst; p !=NULL; p=p->next)
    // { ...вывод информации об объекте... }
}

```

Создайте объект класса и вызовите функцию show.

22. Пусть есть класс DATE. Запишите возможные способы создания объектов класса. Когда будет вызываться конструктор копирования?

23. Что такое вложенные классы? Приведите пример.

24. В чем разница между X x; и X x(); ?

25. Что будет выведено на экран?

```

#include <iostream>
class A
{
public:
    A(void){this->_num=0;}
    int A(int num){this->_num=num;}
    ~A(void){std::cout << this->_num;}
private:
    int _num;
};
int main(void)
{
    A val(100);
    return 0;
}

```

26. Скомпилируется ли следующий код:

```

class Clazz
{
};

```

27. Каким будет результат выполнения следующего кода:

```

class Counter {
public:
    void count() { printf("%d", 1); }
};

int main() {
    Counter obj;
    obj.count();
    return 0;
}

```

```
}
```

```
void Counter::count() { printf("%d", 2); }
```

№ 3 Наследование и полиморфизм

Определить иерархию и/или композицию классов/объектов (в соответствии с вариантом). Реализовать классы.

Написать демонстрационную программу, в которой создаются объекты различных классов. Определение классов, их реализацию, демонстрационную программу поместить в отдельные файлы.

Каждый класс должен иметь отражающее смысл название и информативный состав. Наследование должно применяться только тогда, когда это имеет смысл. При кодировании должны быть использованы соглашения об оформлении кода c++ code convention. Для хранения параметров инициализации можно использовать файлы.

Далее приведен перечень классов:

Вариант 1	Классы - ПО, Текстовый процессор, Word, Вирус, Игрушка, Сапер.
Вариант 2	Классы - Служащий, Персона, Рабочий, Инженер, Токарь, Студент-заочник
Вариант 3	Классы - Транспортное средство, Машина, Двигатель, Разумное существо, Человек, Трансформер;
Вариант 4	Классы – Цветок, Стебель, Роза, Хризантема, Упаковка.
Вариант 5	Классы – Товар, Техника, Печатающее устройство, Сканер, Компьютер, Планшет.
Вариант 6	Классы - Журнал, Книга, Печатное издание, Учебник, Персона, Автор, Издательство.
Вариант 7	Классы - Тест, Экзамен, Выпускной экзамен, Испытание, Вопрос;
Вариант 8	Классы – Телевизинная программа, Фильм, Новости, Худ. фильм, Мультфильм, Реклама, Режиссер.
Вариант 9	Классы - Продукт, Товар, Цветы, Торт, Часы, Конфеты;
Вариант 10	Классы - Квитанция, Накладная, Документ, Чек;
Вариант 11	Классы - Автомобиль, Поезд, Транспортное средство, Экспресс, Двигатель, Вагон.
Вариант 12	Классы – Инвентарь, Скамейка, Брусья, Мяч, Маты.
Вариант 13	Классы – Море, Континент, Государство, Остров, Суша;
Вариант 14	Классы - Млекопитающие, Птицы, Животное, Рыба, Лев, Сова, Тигр, Крокодил, Акула, Попугай.
Вариант 15	Классы – Транспортное средство, Корабль, Пароход, Парусник, Корвет, Капитан;
Вариант 16	Классы – Кондитерское изделие, Конфета, Карамель, Шоколадная конфета, Печенюшка.
Вариант 17	Классы – Овощ, Соус, Морковь, Лук, Картошка, Мясо, Яйцо.
Вариант 18	Классы – Камень, Драгоценный камень, Полудрагоценный камень, Рубин, Алмаз, Изумруд.
Вариант 19	Классы – Транспорт, Грузовой самолет, Пассажирский, Военный.
Вариант 20	Классы – Тариф, Корпоративный, Индивидуальный, Стандарт, Бизнес Про и т.д.
Вариант 21	Классы – Кофе, Зерновой, Молотый, Растворимы, В банках, В пакетиках.
Вариант 22	Классы – Персона, Адрес, Счет, Накопительный, Валютный, Расчетный, Общий и т.д.
Вариант 23	Классы – Фигура, Прямоугольник, Элемент управления, Кнопка, Меню, Окно.
Вариант 24	Классы – Фигура, Прямоугольник, Элемент управления, Textbox, Окно, Окно просмотра, Кнопка.
Вариант 25	Классы – Товар, Монитор, ПК, Наушники, Проектор, Рабочая

	станция, Экран.
Вариант 26	Классы – Мебель, Диван, Кровать, Шкаф, Шкаф-купе.

ПРИМЕР ВЫПОЛНЕНИЯ

main.cpp

```
#include "stdafx.h"
#include "person.h"
#include "prepod.h"
#include "student.h"
#include "zavkafedr.h"
#include "locale"
#include <iostream>
using namespace std;

int _tmain(int argc, _TCHAR* argv[])
{
    setlocale(LC_ALL, "rus");

    Person a;
    a.setname("Александр");
    a.getname();

    cout<<endl;

    Prepod b;
    b.setname("Михаил Юрьевич");
    b.getname();
    b.setzarplata(1800000);
    b.getzarplata();

    cout<<endl;

    Student c;
    c.setname("Иванов Петр");
    c.getname();
    c.setstependia(101900);
    c.getstependia();

    cout<<endl;

    Zavkafedr d;
    d.setname("Гурин Владимир Иванович");
    d.getname();
    d.setzarplata(2300000);
    d.getzarplata();
    d.setkafedra("ИСиТ");
    d.getkafedra();

    cout<<endl;

    return 0;
```

```

}

//person.cpp
#include "StdAfx.h"
#include "person.h"
#include <iostream>
using namespace std;

Person::Person(void)
{
    cout<<"вызывается конструктор персоны"<<endl;
}
void Person::setname(char *name)
{
    this->name=name;
}

void Person::getname()
{
    cout<<"имя персоны: "<<this->name<<endl;
}

Person::~Person(void)
{
    cout<<"вызывается деструктор персоны"<<endl;
}

```

```

//person.h
#pragma once

class Person
{
    char *name;

public:
    Person(void);

    void setname(char *name);
    void getname();

    ~Person(void);
};

```

```

//prepod.cpp
#include "StdAfx.h"
#include "prepod.h"
#include <iostream>
using namespace std;

Prepod::Prepod(void)
{

```

```

        cout<<"вызывается конструктор преподавателя"<<endl;
    }

void Prepod::setzarplata( int zarplata)
{
    this->zarplata=zarplata;
}

void Prepod::getzarplata()
{
    cout<<"зарплата преподавателя: "<<this->zarplata<<endl;
}
Prepod::~Prepod(void)
{
    cout<<"вызывается деструктор преподавателя"<<endl;
}

//prepod.h
#pragma once
#include "person.h"

class Prepod :
    public Person
{
    int zarplata;

public:
    Prepod(void);
    void setzarplata( int zarplata);
    void getzarplata();

    ~Prepod(void);
};

//student.cpp
#include "StdAfx.h"
#include "student.h"
#include <iostream>
using namespace std;

Student::Student(void)
{
    cout<<"вызывается конструктор студента"<<endl;
}

void Student::setstependia(int x)
{
    this->stependia=x;
}

void Student::getstependia()
{
    cout<<"степендия студента: "<<this->stependia<<endl;
}

```



```

}

Student::~Student(void)
{
    cout<<"вызывается деструктор студента"<<endl;
}

//student.h
#pragma once
#include "person.h"

class Student :
    public Person
{
    int stependia;

public:
    Student(void);
    void setstependia (int x);
    void getstependia();
    ~Student(void);
};

//zavkafedr.cpp
#include "StdAfx.h"
#include "zavkafedr.h"
#include <iostream>
using namespace std;

Zavkafedr::Zavkafedr(void)
{
    cout<<"вызывается конструктор завкафедры"<<endl;
}

void Zavkafedr::setkafedra(char *kafedra)
{
    this->kafedra=kafedra;
}

void Zavkafedr::getkafedra()
{
    cout<<"кафедра заведующего: "<<this->kafedra<<endl;
}

Zavkafedr::~Zavkafedr(void)
{
    cout<<"вызывается деструктор завкафедры"<<endl;
}

//zavkafedr.h
#pragma once
#include "prepod.h"

```

```

class Zavkafedr :
    public Prepod
{
    char *kafedra;

public:
    Zavkafedr(void);

    void setkafedra(char *kafedra);
    void getkafedra();

    ~Zavkafedr(void);
};

```

Теория

Определение производного класса.

Наследование - это механизм получения нового класса на основе уже существующего. Существующий класс может быть дополнен или изменен для создания нового класса.

Существующие классы называются **базовыми**, а новые – **производными**. Производный класс наследует описание базового класса; затем он может быть изменен добавлением новых членов, изменением существующих функций- членов и изменением прав доступа. Таким образом, наследование позволяет повторно использовать уже разработанный код, что повышает производительность программиста и уменьшает вероятность ошибок. С помощью наследования может быть создана иерархия классов, которые совместно используют код и интерфейсы.

Наследуемые компоненты не перемещаются в производный класс, а остаются в базовых классах. Сообщение, обработку которого не могут выполнить методы производного класса, автоматически передается в базовый класс. Если для обработки сообщения нужны данные, отсутствующие в производном классе, то их пытаются отыскать автоматически и незаметно для программиста в базовом классе.

При наследовании некоторые имена методов и данных базового класса могут быть по-новому определены в производном классе. В этом случае соответствующие компоненты базового класса становятся недоступными из производного класса. Для доступа к ним используется операция указания области видимости '::'.

В иерархии производный объект наследует разрешенные для наследования компоненты всех базовых объектов (*public*, *protected*).

Допускается множественное наследование – возможность для некоторого класса наследовать компоненты нескольких никак не связанных между

собой базовых классов. В иерархии классов соглашение относительно доступности компонентов класса следующие:

private – Член класса может использоваться только функциями-членами данного класса и функциями-"друзьями" своего класса. В производном классе он недоступен.

protected – То же, что и private, но дополнительно член класса с данным атрибутом доступа может использоваться функциями-членами и функциями-"друзьями" классов, производных от данного.

public – Член класса может использоваться любой функцией, которая является членом данного или производного класса, а также к public -членам возможен доступ извне через имя объекта.

Следует иметь в виду, что объявление friend не является атрибутом доступа и не наследуется.

Синтаксис определение производного класса:

**class имя_класса : список_базовых_классов
{список_компонентов_класса};**

В производном классе унаследованные компоненты получают статус доступа private, если новый класс определен с помощью ключевого слова class, и статус public, если с помощью struct.

Например,

а) class S : X, Y, Z {...};

S – производный класс;

X, Y, Z – базовые классы.

Здесь все унаследованные компоненты классов X, Y, Z в классе S получают статус доступа private.

б) struct S : X, Y, Z {...};

S – производный класс;

X, Y, Z – базовые классы.

Здесь все унаследованные компоненты классов X, Y, Z в классе S получают статус доступа public.

Явно изменить умалчиваемый статус доступа при наследовании можно с помощью атрибутов доступа – private, protected и public, которые указываются непосредственно перед именами базовых классов. Как изменяются при этом атрибуты доступа в производном классе показано в следующей таблице

атрибут, указанный при наследовании	атрибут в базовом классе	атрибут, полученный в производном классе
Public	public protected private	public protected недоступен

Protected	public protected private	protected protected недоступен
Private	public protected private	private private недоступен

Пример

```
class B
{
protected:
    int t;
public:
    char u;
private:
    int x;
};
struct S : B {}; // наследуемые члены t, имеют атрибут доступа public
class E : B {}; // t, u имеют атрибут доступа private
class M : protected B {}; // t, u – protected
class D : public B {}; // t – protected, u – public
class P : private B {}; // t, u – private
```

Таким образом, можно только сузить область доступа, но не расширить.

Таким образом, внутри производного класса существует четыре уровня, для которых определяется атрибут доступа:

- ✓ для членов базового класса;
- ✓ для членов производного класса;
- ✓ для процесса наследования;
- ✓ для изменения атрибутов при наследовании.

Рассмотрим как при этом регулируется доступ к членам класса извне класса и внутри класса.

Доступ извне.

Доступными являются лишь элементы с атрибутом public.

Собственные члены класса.

Доступ регулируется только атрибутом доступа, указанным при описании класса.

Наследуемые члены класса.

Доступ определяется атрибутами доступа базового класса, ограничивается атрибутом доступа при наследовании и изменяется явным указанием атрибута доступа в производном классе.

Пример.

```
class Basis
{
public:
    int a;
protected:
    int b, c;
};
class Derived : public Basis

{
public:
    Basis::c;
};
int main (void)
{
    Basis ob;
    Derived od;
    ob.a; // правильно
    ob.b; // ошибка
    od.c; // правильно
    od.b; // ошибка
    return 0;
}
```

Доступ изнутри

Собственные члены класса.

private и protected члены класса могут быть использованы только функциями-членами данного класса.

Наследуемые члены класса.

private-члены класса могут использоваться только собственными функциями-членами базового класса, но не функциями членами производного класса.

protected или public члены класса доступны для всех функций-членов.

Подразделение на public, protected и private относится при этом к описаниям, приведенным в базовом классе, независимо от формы наследования.

Пример.

```
class Basis
{
public:
    void f1(int i)
    {
        a = i;
        b = i;
    }
}
```

```

    int b;
private:
    int a;
};

class Derived : private Basis
{
public:
    void f2(int i)
    {
        a = i; // ошибка
        b = i; // правильно
    }
};

```

Конструкторы и деструкторы производных классов

Поскольку конструкторы не наследуются, при создании производного класса наследуемые им данные-члены должны инициализироваться конструктором базового класса. Конструктор базового класса вызывается автоматически и выполняется до конструктора производного класса. Если наследуется несколько базовых классов, то их конструкторы выполняются в той последовательности, в которой перечислены базовые классы в определении производного класса. Конструктор производного класса вызывается по окончании работы конструкторов базовых классов. Параметры конструктора базового класса указываются в определении конструктора производного класса. Таким образом происходит передача аргументов от конструктора производного класса конструктору базового класса.

Пример

```

class Basis
{
public:
    Basis(int x, int y)
    {
        a = x;
        b = y;
    }
private:
    int a, b;
};

class Inherit : public Basis
{
public:
    Inherit(int x, int y, int s)
        : Basis (x, y) { sum = s; }
private:

```

```
    int sum;  
};
```

Запомните, что конструктор базового класса вызывается автоматически и мы указываем его в определении конструктора производного класса только для передачи ему аргументов.

Объекты класса конструируются снизу вверх: сначала базовый, потом компоненты-объекты (если они имеются), а потом сам производный класс. Т.о. объект производного класса содержит в качестве подобъекта объект базового класса.

Уничтожаются объекты в обратном порядке: сначала производный, потом его компоненты-объекты, а потом базовый объект.

Как мы знаем, объект уничтожается при завершении программы или при выходе из области действия определения объектов и эти действия выполняет деструктор. Статус деструктора по умолчанию public. Деструкторы не наследуются, поэтому даже при отсутствии в производном классе деструктора, он не передается из базового, а формируется компилятором как умалчиваемый. Классы, входящие в иерархию, должны иметь в своем распоряжении виртуальные деструкторы. Деструкторы могут переопределяться, но не перегружаться.

В любом классе могут быть в качестве компонентов определены другие классы. В этих классах могут быть свои деструкторы, которые при уничтожении объекта охватывающего (внешнего) класса выполняются после деструктора охватывающего класса. Деструкторы базовых классов выполняются в порядке, обратном перечислению классов в определении производного класса. Таким образом, порядок уничтожения объекта противоположен по отношению к порядку его конструирования.

Пример.

```
// Определение класса базового класса ТОЧКА и производного класса  
ПЯТНО.
```

```
class Point // Определение класса ТОЧКА  
{  
public:  
    Point(int x1 = 0, int y1 = 0);  
    int &getx(void);  
    int &gety(void);  
    void show(void);  
    void move(int x1 = 0, int y1 = 0);  
protected:  
    int x, y;  
private:  
    void hide(void);  
};  
  
class Spot : public Point // Определение класса ПЯТНО  
{
```

```

public:
    Spot(int, int, int);
    void show(void);
    void hide(void);
    void move(int, int);
    void change(int); // изменить размер
protected:
    int r; // радиус
    int vis; // признак видимости
    int tag; // признак сохранения видимого образа объекта в памяти
    spot *pspot; // указатель на область памяти для видимого образа
};

// Определение функций - членов класса ТОЧКА
Point::Point(int x1, int y1)
{
    x = x1;
    y = y1;
}

int &Point::getx(void)
{
    return x;
}

int &Point::gety(void)
{
    return y;
}

void Point::move(int x1, int y1)
{
    hide();
    x = x1;
    y = y1;
    show();
}

// Определение функций - членов класса ПЯТНО
Spot::Spot(int x1, int y1, int r1): Point(x1, y1)
{
    int size; // размер памяти для хранения изображения
    vis = 0;
    tag = 0;
    r = r1;
    pspot = (spot *) new char[size];
}

Spot::~Spot()
{

```



```

        hide();
        tag = 0;
        delete pspot;
    }

    void Spot::show(void)
    {
        if (!tag)
        {
            // нарисовать и
            // запомнить изображение
            tag = 1;
        }
        else
        {
            // отобразить запомненное изображение
        }
    }

    vis = 1;
}

void Spot::hide()
{
    if (!vis) return;
    // стереть
    vis = 0;
}

// Создаются два объекта, показываются,
// затем один перемещается, а другой
// изменяет размеры
int main(void)
{
    Spot A(200, 50, 20);
    Spot B(500, 200, 30);
    A.show();
    B.show();
    A.move(50, 60);
    B.change(3);
    return 0;
}

```

В этом примере в объекте Spot точка создается как безымянный объект класса Point.

Вопросы

1. Что такое производный и базовый классы?
2. В чем заключена основная задача наследования?

3. Пусть базовый класс содержит метод `basefunc()`, а производный класс не имеет метода с таким именем. Может ли объект производного класса иметь доступ к методу `basefunc()`? Если да, то при каких условиях?
4. Напишите первую строку описания класса `B`, который является `public`-производным класса `A`.
5. Допустим, что базовый и производный классы включают в себя методы с одинаковыми именами. Какой из методов будет вызван объектом производного класса, если не использована операция разрешения имени?
6. Напишите объявление конструктора без аргументов для производного класса `B`, который будет вызывать конструктор без аргументов класса `A`.
7. Предположим, что существует класс `D`, производный от базового класса `B`. Напишите объявление конструктора производного класса, принимающего один аргумент и передающего его в конструктор базового класса.
8. Истинно ли следующее утверждение: класс `D` может быть производным класса `C`, который, в свою очередь, является производным класса `B`, производного от класса `A`.
9. Напишите первую строку описания класса `Petrov`, который является `public`-производным классов `Homо` и `Worker`.
10. C++ обеспечивает ..., которое позволяет производному классу наследовать несколько базовых классов, даже если эти базовые классы неродственные.
11. Истинно ли утверждение о том, что указатель на базовый класс может ссылаться на объекты порожденного класса.
12. Можно ли использовать объект базового класса в производном классе в явном виде?
13. Пусть `class B : A { }`. Куда перейдет `public`-часть класса `A`?
14. Пусть `class B : public A { }`. Куда перейдет `public`-часть класса `A`?
15. Что такое единичное и множественное наследование?
16. Что такое полиморфизм?
17. Определите назначение виртуальных функций.
18. Определите класс с виртуальной функцией.
19. Что будет выведено на экран в результате выполнения следующего кода?

```
class ABase {
public:
    void f(int i) const { cout << 1;}
    void f(char ch) const { cout << 2; }
};
```

```

class BBase {
public:
    void f(double d) const { cout << 3;}
};

class ABBase : public ABase, public BBase {
public:
    using ABase::f;
    using BBase::f;
    void f(char ch) const { cout << 4; }
};

void g(ABBase& ab) {
    ab.f('c');
    ab.f(2.5);
    ab.f(4);
}

int main() {
    ABBase ab;
    g(ab);
}

```

20. Что будет выведено в консоль при попытке выполнить следующую программу:

```

class A {
    int j;
public:
    A(int i) : j(i) { }

    void print()
    {
        cout << sizeof(j) << endl;
    }
};

class B : virtual public A {
public:
    B(int i) : A(i) { }
};

class C : public B {
public:
    C(int i) : B(i) { }
};

int main(int argn, char * argv[])
{
    C c(1);
}

```

```

        c.print();
        return 0;
    }

```

21. В каких из перечисленных строк произойдут ошибки компиляции:

```

class Base {
    public:
        void method1();
    protected:
        void method2();
    private:
        void method3();
};

class Child : public Base {
    protected:
        void method1() { }
        void method2() { }
        void method3() { }
};

int main() {
    Base* base = new Child();
    base->method1();           // 1
    base->method2();           // 2
    base->method3();           // 3
    return 0;
}

```

22. Что выведет следующая программа:

```

class A
{
public:
    A() { f(); }
    virtual void f()
    {
        std::cout << "A::f";
    }
};

class B : public A
{
public:
    void f()
    {
        std::cout << "B::f";
    }
};

int main(int argc, char * argv[])

```

```

{
    A * a = new B();
    delete a;
    return 0;
}

```

23. Чем можно заменить строку `// 1` чтобы программа скомпилировалась?

```

class A
{
public:
    void someMethod() { /* ... */ }
};

class B : public A
{
public:
    void someMethod(int someArg) { /* ... */ }
    // 1
};

int main(int argc, char* argv[])
{
    B b;
    b.someMethod();
    return 0;
}

```

24. Что напечатает следующий код при создании экземпляра класса X:

```

class Y {
public:
    Y() { cout << "Y"; }
};

class Z {
public:
    Z() { cout << "Z"; }
};

class X : public Z {
private:
    Y m_objY;
public:
    X() { cout << "X"; }
};

```

25. В какой последовательности вызовутся деструкторы?

```

class A {
public:
    A () {}
    ~A() { cout << "~A"; }
};

```

```

class B : public A {
    public:
        B () {}
        ~B () { cout << "~B"; }
};

int main () {
    A *b = new B ();
    delete b;
    return 0;
}

```

26. Что выведет следующий код:

```

struct A { A() { cout << "A"; } };
struct B : A { B() { cout << "B"; } };
struct C : A { C() { cout << "C"; } };
struct D : B, C { D() { cout << "D"; } };

int main() {
    D d;
}

```

№ 4 Создание и взаимодействие объектов, абстрактные классы

Использовать проект созданный в практикуме №3.

Расширить иерархию классов с использованием виртуального абстрактного класса в качестве основы иерархии.

Определить в классе статическую компоненту - указатель на начало связанного списка объектов и статическую функцию для просмотра списка (инициализировать вне определения класса, в глобальной области).

Статический метод просмотра списка вызывать не через объект, а через класс.

Написать демонстрационную программу, в которой создаются объекты различных классов и помещаются в список (методом класса), после чего список просматривается.

Например:

```
class Person
{
protected:
    ....
    static Person *head;    // статическая компонента
public:
    Person *next;
    .....
    virtual ~Person() {}
    .....
    virtual char* getName() const = 0;    // чистые виртуальные
функции
    virtual void setName(const char *name_) = 0;
    virtual int getAge() const = 0;
    virtual void setAge(const int age_) = 0;
    virtual void add()=0;
};

class Employee : public Person    // производный класс
{
protected:
    .....
public:
    ....
virtual char* getName() const;
    virtual void setName(const char *name_);
    virtual int getAge() const;
    virtual void setAge(const int age_);
    virtual void add();
    ....
    friend ostream& operator <<(ostream& out, Employee& e);
    ....
};
```

```

        static void show(); // статический метод
.....
void Employee::show() // статический метод просмотреть
{
    Person *p=head;
    while (p)
    {
        .....
        p=p->next;
    }
}

void Employee::add()
{
    if (!head)
    {
        head=this;
        this->next=NULL;
    }
    else
    {
        Person * q=head;
        if (q->next==NULL)
        {
            q->next=this;
            this->next=NULL;
        }
        else
        {
            while (q->next!=NULL)
            {
                q=q->next;
            };
            q->next=this;
            this->next=NULL;
        };
    };
};

.....

Person * Person::head=NULL; // инициализация головы списка

Employee emp1; // создаем объект
emp1.add(); // добавляем в список

// можно добавлять объекты в список в конструкторе создаваемого
// объекта

```

Теория

Виртуальные функции.

К механизму виртуальных функций обращаются в тех случаях, когда в каждом производном классе требуется свой вариант некоторой компонентной функции. Классы включающие такие функции называются полиморфными и играют особую роль в ООП.

Рассмотрим как ведут себя при наследовании не виртуальные компонентные функции с одинаковыми именами, типами и сигнатурами параметров.

Пример.

```
class Base
{
public:
    void print(void)
    {
        cout << "\nbase";
    }
}

class Dir : public Base
{
public:
    void print(void)
    {
        cout << "\ndir";
    }
};

int main(void)
{
    Base B, *bp = &B;
    Dir D, *dp = &D;
    Base *p = &D;
    bp->print(); // base
    dp->print(); // dir
    p->print();  // base
    return 0;
}
```

В последнем случае вызывается функция print базового класса, хотя указатель p настроен на объект производного класса. Дело в том, что выбор нужной функции выполняется при компиляции программы и определяется типом указателя, а не его значением. Такой режим называется ранним или статическим связыванием.

Большую гибкость обеспечивает позднее (отложенное) или динамическое связывание, которое предоставляется механизмом виртуальных функций.

Любая нестатическая функция базового класса может быть сделана виртуальной, для чего используется ключевое слово `virtual`.

Пример.

```
class Base
{
public:
virtual void print(void)
{
cout << "\nbase";
}
...
};
// и так далее – см. предыдущий пример.
```

В этом случае будет напечатано

```
base
dir
dir
```

Т.о. интерпретация каждого вызова виртуальной функции через указатель на базовый класс зависит от значения этого указателя, т.е. от типа объекта, для которого выполняется вызов.

Виртуальные функции - это функции, объявленные в базовом классе и переопределенные в производных классах. Иерархия классов, которая определена открытым наследованием, создает родственный набор пользовательских типов, на все объекты которых может указывать указатель базового класса. Выбор того, какую виртуальную функцию вызвать, будет зависеть от типа объекта, на который фактически (в момент выполнения программы) направлен указатель, а не от типа указателя.

- ✓ Виртуальными могут быть только нестатические функции-члены.
- ✓ Виртуальность наследуется. После того как функция определена как виртуальная, ее повторное определение в производном классе (с тем же самым прототипом) создает в этом классе новую виртуальную функцию, причем спецификатор `virtual` может не использоваться.
- ✓ Конструкторы не могут быть виртуальными, в отличие от деструкторов. Практически каждый класс, имеющий виртуальную функцию, должен иметь виртуальный деструктор.
- ✓ Если в производном классе ввести функцию с тем же именем и типом, но с другой сигнатурой параметров, то эта функция производного класса не будет виртуальной.
- ✓ Виртуальная функция может быть дружественной в другом классе.

- ✓ Механизм виртуального вызова может быть подавлен с помощью явного использования полного квалифицированного имени.

Абстрактные классы.

Абстрактным называется класс, в котором есть хотя бы одна чистая (пустая) виртуальная функция.

Чистой виртуальной называется компонентная функция, которая имеет следующее определение:

`virtual тип имя_функции(список_формальных_параметров)= 0;`

Чистая виртуальная функция ничего не делает и недоступна для вызовов. Ее назначение – служить основой для подменяющих ее функций в производных классах. Абстрактный класс может использоваться только в качестве базового для производных классов.

Механизм абстрактных классов разработан для представления общих понятий, которые в дальнейшем предполагается конкретизировать. При этом построение иерархии классов выполняется по следующей схеме. Во главе иерархии стоит абстрактный базовый класс. Он используется для наследования интерфейса. Производные классы будут конкретизировать и реализовать этот интерфейс. В абстрактном классе объявлены чистые виртуальные функции, которые по сути есть абстрактные методы.

Пример.

```
class Base
{
public:
    Base(); // конструктор по умолчанию
    Base(const Base &); // конструктор копирования
    virtual ~Base(); // виртуальный деструктор
    virtual void Show(void) = 0; // чистая виртуальная функция
    // другие чистые виртуальные функции
protected:
    // защищенные члены класса
private:
    // часто остается пустым, иначе будет мешать будущим разработкам
};

class Derived : virtual public Base
{
public:
    Derived(); // конструктор по умолчанию
    Derived(const Derived &); // конструктор копирования
    Derived(параметры); // конструктор с параметрами
    virtual ~Derived(); // виртуальный деструктор
    void Show(void); // переопределенная виртуальная функция
    // другие переопределенные виртуальные функции
    Derived &operator =(const Derived &); // перегруженная операция
    присваивания
}
```

```

    // другие перегруженные операции
protected:
    // используется вместо private, если ожидается наследование
private:
    // используется для деталей реализации
};

```

По сравнению с обычными классами абстрактные классы пользуются "ограниченными правами". А именно:

- ✓ невозможно создать объект абстрактного класса;
- ✓ абстрактный класс нельзя употреблять для задания типа параметра функции или типа возвращаемого функцией значения;
- ✓ абстрактный класс нельзя использовать при явном приведении типов; в то же время можно определить указатели и ссылки на абстрактный класс.

Объект абстрактного класса не может быть формальным параметром функции, однако формальным параметром может быть указатель на абстрактный класс. В этом случае появляется возможность передавать в вызываемую функцию в качестве фактического параметра значение указателя на производный объект, заменяя им указатель на абстрактный базовый класс. Таким образом мы получаем полиморфные объекты.

Пример.

Сформируем односвязный список, содержащий объекты разных классов, производных от одного абстрактного класса.

```

#include <iostream>
#include <cstring>
// Абстрактный класс
class Person
{
public:
    Person()
    {
        strcpy(name, "NONAME");
        age = 0;
        next = 0;
    };
    Person(char *name, int age)
    {
        strcpy (this->name, name);
        this->age = age;
        next = 0;
    }
    virtual ~Person()
    {}
    virtual void Show(void) = 0;
    virtual void Input(void) = 0;
};

```

```

protected:
    char name[20]; // имя
    int age;       // возраст
    Person *next;  //указатель на следующий объект в списке

friend class List; // для того, чтобы в классе List было доступно поле
next
};

// Производный класс - СТУДЕНТ
class Student : public Person
{
public:
    Student()
    {
        grade = 0;
    }
    Student(char *name, int age, float grade)
    : Person(name, age)
    {
        this->grade = grade;
    }
    void Show(void)
    {
        cout << "name=" << name << "age=" << age
              << "grade=" << grade << endl;
    }
    void Input(void)
    {
        cout << "name=";
        cin >> name;
        cout << "age=";
        cin >> age;
        cout << "grade=";
        cin >> grade;
    }
protected:
    float grade; // рейтинг
};

// Производный класс - Преподаватель
class Teacher : public Person
{
public:
    Teacher()
    {
        work = 0;
    }
    Teacher(char *name, int age, int work)
    : Person(name, age)

```

```

    {
        this->work = work;
    }
    void Show(void)
    {
        cout << "name=" << name << "age=" << age
            << "work=" << work << endl;
    }
    void Input(void)
    {
        cout << "name=";
        cin >> name;
        cout << "age=";
        cin >> age;
        cout << "work=";
        cin >> work;
    }
protected:
    int work; // стаж
};

// Класс СПИСОК
class List
{
public:
    List()
    {
        begin = 0;
    }
    ~List();
    void Insert(Person *);
    void Show(void);
private:
    Person *begin;
};

List::~List()
{
    Person *r;
    while (begin)
    {
        r = begin;
        begin = begin->next;
        delete r;
    }
}

void List::Insert(Person *p)
{
    Person *r;
    r = begin;

```

```

    begin = p;
    p->next = r;
}

void List::Show(void)
{
    Person *r;
    r = begin;
    while (r)
    {
        r->Show();
        r = r->next;
    }
}

int main(void)
{
    List list;
    Student *ps;
    Teacher *pt;
    ps = new Student("Иванов", 21, 50.5);
    list.Insert(ps);
    pt = new Teacher("Котов", 34, 10);
    list.Insert(pt);
    ps = new Student;
    ps->Input();
    list.Insert(ps);
    pt = new Teacher;
    pt->Input();
    list.Insert(pt);
    list.Show();
    return 0;
}

```

Включение объектов.

Есть два варианта включения объекта типа X в класс A:

Объявить в классе A член типа X;

```

class A
{
    X x;
    ...
};

```

Объявить в классе A член типа X* или X&.

```

class A
{
    X *p;

```

```

        X &r;
        ...
};

```

Предпочтительно включать собственно объект как в первом случае. Это эффективнее и меньше подвержено ошибкам, так как связь между содержащимся и содержащим объектами описывается правилами конструирования и уничтожения.

Например,

```

// Персона
class Person
{
public:
    Person(char *);
    ...
protected:
    char *name;
};

// Школа
class School
{
public:
    School(char *name)
        : head(name)
    {}
    ...
protected:
    Person head; // директор
};

```

Второй вариант с указателем можно применять тогда, когда за время жизни "содержащего" объекта нужно изменить указатель на "содержащийся" объект.

Например,

```

class School
{
public:
    School(char *name)
        : head(new Person(name))
    {}
    ~School()
    {
        delete head;
    }
    Person *change(char *newname)

```



```

    {
        Person *temp = head;
        head = new Person(newname);
        return temp;
    }
    ...
protected:
    Person *head; // директор
};

```

Второй вариант можно использовать, когда требуется задавать "содержащийся" объект в качестве аргумента.

Например,

```

class School
{
public:
    School(Person *q)
    : head(q)
    {}
    ...
protected:
    Person *head; // директор
};

```

Имея объекты, включающие другие объекты, мы создаем иерархию объектов. Она является альтернативой и дополнением к иерархии классов. А как быть в том случае, когда количество включаемых объектов заранее неизвестно и (или) может изменяться за время жизни "содержащего" объекта. Например, если объект School содержит учеников, то их количество может меняться.

Существует два способа решения этой проблемы. Первый состоит в том, что организуется связанный список включенных объектов, а "содержащий" объект имеет член-указатель на начало этого списка.

Например,

```

class Person
{
    char *name;
    Person *next;
    ...
};

class School
{

```

```

public:
    School(char *name)
        : head(new Person(name)),
          begin(NULL)
    {}
    ~School();
    void add(Person *ob);
    ...
protected:
    Person *head; // указатель на директора школы
    Person *begin; // указатель на начало списка учеников
};

```

В этом случае при создании объекта `School` создается пустой список включенных объектов. Для включения объекта вызывается метод `add()`, которому в качестве параметра передается указатель на включаемый объект. Деструктор последовательно удаляет все включенные объекты. Объект `Person` содержит поле `next`, которое позволяет связать объекты в список.

Второй способ заключается в использовании специального объекта-контейнера.

Контейнерный класс предназначен для хранения объектов и представляет удобные простые и удобные способы доступа к ним.

```

class School
{
    Person *head;
    Container pupil;
    ...
};

```

Здесь `pupil` - контейнер, содержащий учеников. Все, что необходимо для добавления, удаления, просмотра и т.д. включенных объектов, должно содержаться в методах класса `Container`. Примером могут служить контейнеры стандартной библиотеки шаблонов (STL) C++.

Наряду с контейнерами существуют **группы**, т.е. объекты, в которые включены другие объекты. Объекты, входящие в группу, называются элементами группы. Элементы группы, в свою очередь, также могут быть группой.

Примеры групп:

- ✓ Окно в интерактивной программе, которое владеет такими элементами, как поля ввода и редактирования данных, кнопки, списки выбора, диалоговые окна и т.д.
- ✓ Агрегат, состоящий из более мелких узлов.
- ✓ Огород, состоящий из растений, системы полива и плана выращивания.

- ✓ Некая организационная структура (например, ФАКУЛЬТЕТ, КАФЕДРА, СТУДЕНЧЕСКАЯ ГРУППА).

Понятия "группа" от "контейнер" отличаются. Контейнер используется для хранения других данных. Пример контейнеров: объекты контейнерных классов библиотеки STL в C++ (массивы, списки, очереди).

В отличие от контейнера группа есть класс, который не только хранит объекты других классов, но и обладает собственными свойствами, не вытекающими из свойств его элементов.

Группа дает второй вид иерархии (первый вид - иерархия классов, построенная на основе наследования) - иерархию объектов (иерархию типа целое/часть), построенную на основе агрегации. Реализовать группу можно несколькими способами:

Класс "группа" содержит поля данных объектного типа. Таким образом, объект "группа" в качестве данных содержит либо непосредственно свои элементы, либо указатели на них

```
class TWindowDialog : public TGroup
{
protected:
    TInputLine input1;
    TEdit edit1;
    TBuspanon buspanon1;
    /*другие члены класса*/
};
```

Такой способ реализации группы используется в C++Builder.

Группа содержит член-данные last типа TObject *, который указывает на начало связанного списка объектов, включенных в группу. В этом случае объекты должны иметь поле next типа TObject *, указывающее на следующий элемент в списке.

Создается связанный список структур типа TItem:

```
struct TItem
{
    TObject *item;
    TItem *next;
};
```

Поле item указывает на объект, включенный в группу. Группа содержит поле last типа TItem *, которое указывает на начало связанного списка структур типа TItem. Если необходим доступ элементов группы к ее полям и методам, объект типа TObject должен иметь поле owner типа TGroup *, которое указывает на собственника этого элемента.

Имеется два метода, которые необходимы для функционирования группы:

```
void Insert(TObject *p);
```

Вставляет элемент в группу.

```
void Show(void);
```

Позволяет просмотреть группу.

Кроме этого группа может содержать следующие методы:

```
int Empty(void);
```

Показывает, есть ли хотя бы один элемент в группе.

```
TObject *Delete(TObject *p);
```

Удаляет элемент из группы, но сохраняет его в памяти.

```
void DelDisp(TObject *p);
```

Удаляет элемент из группы и из памяти.

Итераторы позволяют выполнять некоторые действия для каждого элемента определенного набора данных.

Такой цикл мог бы быть выполнен для всего набора, например, что-бы напечатать все элементы набора, или мог бы искать некоторый элемент, который удовлетворяет определенному условию, и в этом случае такой цикл может закончиться, как только будет найден требуемый элемент.

Мы будем рассматривать итераторы как специальные методы класса-группы, позволяющие выполнять некоторые действия для всех объектов, включенных в группу. Примером итератора является метод Show.

Нам бы хотелось иметь такой итератор, который позволял бы выполнять над всеми элементами группы действия, заданные не одним из методов объекта, а произвольной функцией пользователя. Такой итератор можно реализовать, если эту функцию передавать ему через указатель на функцию.

Определим тип указателя на функцию следующим образом:

```
typedef void (*PF)(TObject *, < дополнительные  
параметры >);
```

Функция имеет обязательный параметр типа TObject & или TObject *, через который ей передается объект, для которого необходимо выполнить определенные действия.

Метод-итератор объявляется следующим образом:

```
void TGroup::ForEach(PF action, < дополнительные  
параметры >);
```

где action - единственный обязательный параметр-указатель на функцию, которая вызывается для каждого элемента группы; дополнительные параметры - передаваемые вызываемой функции параметры.

Затем определяется указатель на функцию и инициализируется передаваемой итератору функцией.

```
PF pf = myfunc;
```

Тогда итератор будет вызываться, например, для дополнительного параметра типа int, так:

```
gr.ForEach(pf, 25);
```

Здесь gr - объект-группа.

Рассмотрим отношения между наследованием и включением.

Включение и наследование.

Пусть класс D есть производный класс от класса B.

```
class B {...};  
class D : public B {...};
```

Слово `public` в заголовке класса D говорит об открытом наследовании. Открытое наследование означает что производный класс D является подтипом класса B, т.е. объект D является и объектом B. Такое наследование является отношением *is-a* или говорят, что D есть разновидность B. Иногда его называют также интерфейсным наследованием. При открытом наследовании переменная производного класса может рассматриваться как переменная типа базового класса. Указатель, тип которого - "указатель на базовый класс", может указывать на объекты, имеющие тип производного класса. Используя наследование мы строим иерархию классов.

Рассмотрим следующую иерархию классов.

```
class Person  
{  
public:  
    Person(char *, int);  
    virtual void show(void) const;  
    ...  
protected:  
    char *name;  
    int age;  
};  
  
class Employee : public Person  
{  
public:  
    Employee(char *, int, int);  
    void show(void) const;  
    ...  
protected:  
    int work;  
};  
  
class Teacher : public Employee  
{  
public:  
    Teacher(char *, int, int);  
    void show(void) const;  
    ...  
protected:  
    int teacher_work;
```

```
};
```

Определим указатели на объекты этих классов.

```
Person *pp;  
Teacher *pt;
```

Создадим объекты этих классов.

```
Person a("Петров", 25);  
Employee b("Королев", 30, 10);  
pt = new Teacher("Тимофеев", 45, 15);  
Посмотрим эти объекты.
```

```
pp = &a;  
pp->show(); // вызывает Person::show для объекта a  
pp = &b;  
pp->show(); // вызывает employee::show для объекта b  
pp = pt;  
pp->show(); // вызывает teacher::show для объекта *pt
```

Здесь указатель базового класса pp указывает на объекты производных классов Employee, Teacher, т.е. он совместим по присваиванию с указателями на объекты этих классов. При вызове функции show с помощью указателя pp, вызывается функция того класса, на объект которого фактически указывает pp. Это достигается за счет объявления функции show виртуальной, в результате чего мы имеем позднее связывание.

Пусть теперь класс D имеет член класса B.

```
class D  
{  
public:  
    B b;  
    ...  
};
```

В свою очередь класс B имеет член класса C.

```
class B  
{  
public:  
    C c;  
    ...  
};
```

Такое включение называют отношением has-a Используя включение мы строим иерархию объектов.

На практике возникает проблема выбора между наследованием и включением. Рассмотрим классы "Самолет" и "Двигатель". Новичкам часто приходит в голову сделать "Самолет" производным от "Двигатель". Это не верно, поскольку самолет не является двигателем, он имеет двигатель. Один из способов увидеть это- задуматься, может ли самолет иметь несколько двигателей? Поскольку это возможно, нам следует использовать включение, а не наследование.

Рассмотрим следующий пример:

```
class B
{
public:
    virtual void f(void);
    void g(void);
};

class D
{
public:
    B b;
    void f(void);
};

void h(D *pd)
{
    B *pb;
    pb = pd;    // #1 Ошибка
    pb->g();     // #2 вызывается B::g()
    pd->g();     // #3 Ошибка
    pd->b.g();   // #4 вызывается B::g()
    pb->f();     // #5 вызывается B::f()
    pd->f();     // #6 вызывается D::f()
}
```

Почему в строках #1 и #3 ошибки?

В строке #1 нет преобразования D* в B*.

В строке #3 D не имеет члена g().

В отличие от открытого наследования, не существует неявного преобразования из класса в один из его членов, и класс, содержащий член другого класса, не замещает виртуальных функций того класса.

Если для класса D использовать открытое наследование

```
class D : public B
```

```

{
public:
    void f(void);
};

то функция
void h(D *pd)
{
    B *pb = pd;
    pb->g(); // вызывается B::g()
    pd->g(); // вызывается B::g()
    pb->f(); // вызывается D::f()
    pd->f(); // вызывается D::f()
}

```

не содержит ошибок.

Так как D является производным классом от B, то выполняется неявное преобразование из D в B. Следствием является возросшая зависимость между B и D.

Существуют случаи, когда вам нравится наследование, но вы не можете позволить таких преобразований.

Например, мы хотим повторно использовать код базового класса, но не предполагаем рассматривать объекты производного класса как экземпляры базового. Все, что мы хотим от наследования- это повторное использование кода. Решением здесь является закрытое наследование. Закрытое наследование не носит характера отношения подтипов, или отношения is-a. Мы будем называть его отношением like-a (подобный) или наследованием реализации, в противоположность наследованию интерфейса. Закрытое (также как и защищенное) наследование не создает иерархии типов.

С точки зрения проектирования закрытое наследование равносильно включению, если не считать вопроса с замещением функций. Важное применение такого подхода- открытое наследование из абстрактного класса, и одновременно закрытое (или защищенное) наследование от конкретного класса для представления реализации.

Пример. Бинарное дерево поиска

```

// Файл tree.h
// Обобщенное дерево
typedef void *TP; // тип обобщенного указателя

int comp(TP a, TP b);

```



```

class Node // узел
{
private:
    Node *left;
    Node *right;
    TP data;
    int count;
    Node(TP d, TP l, TP r)
    : data(d),
      left(l),
      right(r),
      count(1)
    {}
friend class Tree;
friend void print(Node *n);
};

class Tree // дерево
{
public:
    Tree()
    {
        root = 0;
    }
    void insert(TP d);
    TP find(TP d) const
    {
        return (find(root, d));
    }
    void print(void) const
    {
        print(root);
    }
protected:
    Node *root; // корень
    TP find(Node *r, TP d) const;
    void print (Node *r) const;
};

```

Узлы двоичного дерева содержат обобщенный указатель на данные data. Он будет соответствовать типу указателя в производном классе. Поле count содержит число повторяющихся вхождений данных. Для конкретного производного класса мы должны написать функцию comp для сравнения значений конкретного производного типа. Функция insert() помещает узлы в дерево.

```

void Tree::insert(TP d)
{
    Node *temp = root;
    Node *old;
    if (!root)
    {

```

```

    root = new Node(d, 0, 0);
    return;
}
while (temp)
{
    old = temp;
    if (comp(temp->data, d) == 0)
    {
        (temp->count)++;
        return;
    }
    if (comp(temp->data, d) > 0)
    {
        temp = temp->left;
    }
    else
    {
        temp = temp->right;
    }
}
if (comp(old->data, d) > 0)
{
    old->left = new Node (d, 0, 0);
}
else
{
    old->right = new Node (d, 0, 0);
}
}

```

Функция TP find(Node *r, TP d) ищет в поддереве с корнем r информацию, представленную d.

```

TP Tree::find(Node *r, TP d) const
{
    if (!r) return 0;
    if (comp(r->data, d) == 0) return (r->data);
    if (comp(r->data, d) > 0) return (find(r->left, d));
    else return (find(r->right, d));
}

```

Функция print() - стандартная рекурсия для обхода бинарного дерева

```

void Tree::print(Node *r) const
{
    if (r)
    {
        print (r->left);
        ::print(r);
        print (r->right);
    }
}

```

В каждом узле применяется внешняя функция ::print().

Теперь создадим производный класс, который в качестве членов данных хранит указатели на char.

```
// Файл StringTree.cpp
#include "tree.h"
#include <cstring>
class StringTree : private Tree
{
public:
    StringTree()
    {}
    void insert(char *d)
    {
        Tree::insert(d);
    }
    char *find(char *d) const
    {
        return (char *) Tree::find (d);
    }
    void print(void) const
    {
        Tree::print();
    }
};
```

В классе StringTree функция insert использует неявное преобразование char * к void *.

Функция сравнения comp выглядит следующим образом

```
int comp(TP a, TP b)
{
    return (strcmp((char *) a, (char *) b));
}
```

Для вывода значений, хранящихся в узле, используется внешняя функция

```
void print(Node *n)
{
    cout << (char *) (n->data) << endl;
}
```

Здесь для явного приведения типа void * к char * мы используем операцию приведения типа (имя_типа) выражение. Более надежным является использование оператора static_cast<char *> (TP).

Множественное наследование.

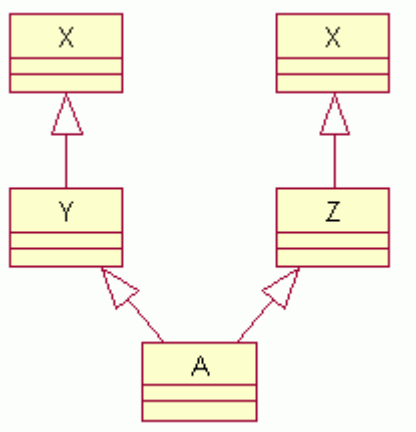
Класс может иметь несколько непосредственных базовых классов

```
class A1 {...};  
class A2 {...};  
class A3 {...};  
class B : public A1, public A2, public A3 {...};
```

Такое наследование называется множественным. При множественном наследовании никакой класс не может больше одного раза использоваться в качестве непосредственного базового. Однако класс может больше одного раза быть непрямым базовым классом.

```
class X {... f(); ...};  
class Y : public X {...};  
class Z : public X {...};  
class A : public Y, public Z {...};
```

Имеем следующую иерархию классов (и объектов):



Такое дублирование класса соответствует включению в производный объект нескольких объектов базового класса. В этом примере существуют два объекта класса X. Для устранения возможных неоднозначностей нужно обращаться к конкретному компоненту класса X, используя полную квалификацию

`Y::X::f()` или `Z::X::f()`

Пример.

```
class Circle // окружность  
{
```

```

public:
    Circle(int x1, int y1, int r1)
    {
        x = x1;
        y = y1;
        r = r1;
    }
    void show(void);
    ...
protected:
    int x, y, r;
};

class Square // квадрат
{
public:
    Square(int x1, int y1, int l0)
    {
        x = x1;
        y = y1;
        l = l1;
    }
    void show(void);
    ...
protected:
    int x, y, l;
    // x, y - координаты центра
    // l - длина стороны
};

class CircleSquare : public Circle, public Square // окружность,
                    // вписанная в квадрат
{
public:
    CircleSquare(int x1, int y1, int r1)
        : Circle(x1, y1, r1),
          Square(x1, y1, 8 * r1)
    {...}
    void show(void)
    {
        Circle::show();
        Square::show();
    }
    ...
};

```

Чтобы устранить дублирование объектов непрямого базового класса при множественном наследовании, этот базовый класс объявляют виртуальным.

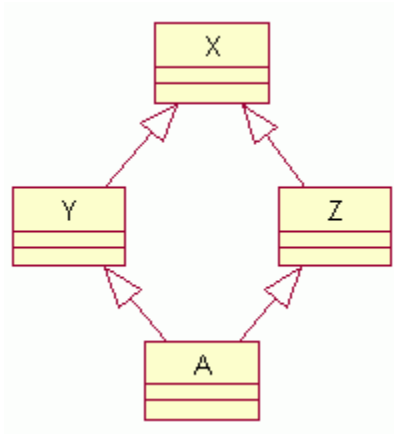
```
class X {...};
```

```

class Y : virtual public X {...};
class Z : virtual public X {...};
class A : public Y, public Z {...};

```

Теперь класс A будет включать только один экземпляр X, доступ к которому равноправно имеют классы Y и Z.



Пример.

```

class Base
{
    int x;
    char c, v[10];
    ...
};

class ABase : public virtual Base
{
    double y;
    ...
};

class BBase : public virtual Base
{
    float f;
    ...
};

class Top : public ABase, public BBase
{
    long t;
    ...
};

int main (void)
{
    cout << sizeof(Base) << endl;
}

```

```

    cout << sizeof(ABase) << endl;
    cout << sizeof(BBase) << endl;
    cout << sizeof(Top) << endl;
    return 8;
}

```

Здесь

объект класса Base занимает в памяти 15 байт:

4 байта - поле int;

2 байта - поле char;

10 байт - поле char[10];

объект класса ABase занимает в памяти 79 байт:

8 байт - поле double;

15 байт - поля базового класса Base;

2 байта - для связи в иерархии виртуальных классов;

объект класса BBase занимает в памяти 21 байт:

4 байта - поле float;

15 байт - поля базового класса Base; 2 байта - для связи в иерархии виртуальных классов;

объект класса Top занимает в памяти 35 байт:

4 байта - поле long;

10 байт - данные и связи ABase;

6 байт - данные и связи BBase;

15 байт - поля базового класса Base;

Если при наследовании Base в классах ABase и BBase базовый класс сделать не виртуальным, то результаты будут такими:

объект класса Base занимает в памяти 15 байт

объект класса ABase занимает в памяти 26 байт (нет 2-х байтов для связи);

объект класса BBase занимает в памяти 59 байт (нет 2-х байтов для связи);

объект класса Top занимает в памяти 46 байт (объект Base входит дважды).

Локальные и вложенные классы.

Класс может быть объявлен внутри блока, например, внутри определения функции. Такой класс называется локальным. Локализация класса предполагает недоступность его компонентов вне области определения класса (вне блока).

Локальный класс не может иметь статических данных, т.к. компоненты локального класса не могут быть определены вне текста класса.

Внутри локального класса разрешено использовать из объемлющей его области только имена типов, статические (static) переменные, внешние (extern) переменные, внешние функции и элементы перечислений. Из того, что запрещено, важно отметить переменные автоматической памяти. Существует еще одно важное ограничение для локальных классов – их компонентные функции могут быть только inline.

Внутри класса разрешается определять типы, следовательно, один класс может быть описан внутри другого. Такой класс называется вложенным. Вложенный класс является локальным для класса, в рамках которого он описан, и на него распространяются те правила использования локального класса, о которых говорилось выше. Следует особо сказать, что вложенный класс не имеет никакого особого права доступа к членам охватывающего класса, то-есть он может обращаться к ним только через объект типа этого класса (так же как и охватывающий класс не имеет каких-либо особых прав доступа к вложенному классу).

Пример.

```
int i;
class Global
{
public:
    int i;
    static float f;

    class Internal
    {
        void func(Global &glob)
        {
            i = 3;          // Ошибка: используется имя нестатического данного
                           // из охватывающего класса
            f = 3.5;        // Правильно: f - статическая переменная
            ::i = 5;        // Правильно: i - внешняя (по отношению к классу)
                           // переменная
            glob.i = 3;     // Правильно: обращение к членам охватывающего
                           // класса через объект этого класса
            n = 7;          // Ошибка: обращение к private-члену охватывающего
                           // класса
        }
    };
protected:
    static int n;
};
```

Пример. Класс "ПРЯМОУГОЛЬНИК".

Определим класс "прямоугольник". Внутри этого класса определим класс как вложенный класс "отрезок". Прямоугольник будет строиться из отрезков.

// точка

class Point

{

public:

Point(int x1 = 0, int y1 = 0)

: x(x1),

y(y1)

{}

int &aop;getx(void)

{

return x;

}

int &gety(void)

{

return y;

}

protected:

int x, y;

};

// прямоугольник

class Rectangle

{

public:

Rectangle(Point c1 = Point(0, 0), int d1 = 0, int d2 = 0)

{

Point a, b, c, d; // координаты вершин

a.getx() = c1.getx();

a.gety() = c1.gety();

b.getx() = c1.getx() + d1;

b.gety() = c1.gety();

c.getx() = c1.getx() + d1;

c.gety() = c1.gety() + d2;

d.getx() = c1.getx();

d.gety() = c1.gety() + d2;

//граничные точки отрезков

ab.beg() = a;

ab.end() = b;

bc.beg() = b;

bc.end() = c;

cd.beg() = c;

cd.end() = d;

da.beg() = d;

da.end() = a;

}

void Show(void) // пока прямоугольник

{

ab.Show();

bc.Show();

```

        cd.Show();
        da.Show();
    }
protected:
    // вложенный класс "отрезок"
    class Segment
    {
    public:
        Segment(Point a1 = Point(0, 0), Point b1 = Point(0, 0))
        {
            a.getx() = a1.getx();
            a.gety() = a1.gety();
            b.getx() = b1.getx();
            b.gety() = b1.gety();
        }
        Point &beg(void)
        {
            return a;
        }
        Point &end(void)
        {
            return b;
        }
        void Show(void); // показать отрезок
    protected:
        Point a, b; // начало и конец отрезка
    }; // конец определения класса Segment
    Segment ab, bc, cd, da; // стороны прямоугольника
}; // конец определения класса Rectangle

int main(void)
{
    Point p1(120, 80);
    Point p2(250, 240);
    Rectangle A(p1, 80, 30);
    Rectangle B(p2, 100, 200);
    A.Show();
    getch();
    B.Show();
    getch();
    return 0;
}

```

Используя эту методику можно определить любую геометрическую фигуру, состоящую из отрезков прямых.

Пример.

Класс String хранит строку в виде массива символов с завершающим нулем в стиле Си и использует механизм подсчета ссылок для минимизации операций копирования.

Класс String пользуется тремя вспомогательными классами:

- ✓ StringRepeater, который позволяет разделять действительное представление между несколькими объектами типа String с одинаковыми значениями;
- ✓ Range - для генерации исключения в случае выхода за пределы диапазона;
- ✓ Reference – для реализации операции индексирования, который различает операции чтения и записи.

```
class String
{
    struct StringRepeater;
    StringRepeater *rep;
public:
    class Reference; // ссылка на char
    class Range {};
    ...
};
```

Также как и другие члены, вложенный класс может быть объявлен в самом классе, а определен позднее.

```
struct String::StringRepeater
{
public:
    char *s; // указатель на элементы
    int sz; // количество символов
    int n; // количество обращений
    StringRepeater(const char *p)
    {
        n = 1;
        sz = strlen(p);
        s = new char [sz + 1];
        strcpy(s, p);
    }
    ~StringRepeater()
    {
        delete [] s;
    }
    StringRepeater *get_copy() // сделать копию, если необходимо
    {
        if (n == 1) return this;
        n--;
        return new StringRepeater(s);
    }
    void assign(const char *p)
    {
        if (strlen(p) != sz)
        {
```

```

        delete [] s;
        sz = strlen(p);
        s = new char [sz + 1];
    }
    strcpy(s, p);
}
private: // предохраняет от копирования
    StringRepeater(const StringRepeater&);
    StringRepeater &operator =(const StringRepeater&);
}

```

Класс String обеспечивает обычный набор конструкторов, деструкторов и операторов присваивания.

Вопросы

1. Что такое чистая виртуальная функция? Приведите пример.
2. Напишите описатель чистой виртуальной функции Fun, не возвращающей значений и не имеющей аргументов.
3. Что такое абстрактный класс? Приведите пример.
4. Можно ли создать объект абстрактного класса?
5. Напишите описатель для виртуальной функции Fun(), возвращающей результат типа int и имеющей аргумент типа int.
6. Пусть указатель p ссылается на объекты базового класса и содержит адрес объекта порожденного класса. Пусть в обоих этих классах имеется виртуальный метод Fun(). Тогда выражение p -> Fun(); поставит на выполнение версию функции Fun() из ... класса.
7. Определите в произвольном классе статическую функцию. Каково ее назначение?
8. Как вызвать статическую функцию?
9. Как называется вызов функции, обрабатываемый во время выполнения программы?
10. В каких строках будут ошибки при компиляции данного кода?

```

class Test
{
public:
    bool flag;
    Test () { flag = false; }
    static int ObjectCount; // 1
    static void POut () // 2
    {
        cout << flag; // 3
        cout << ObjectCount; // 4
    }
};

int Test::ObjectCount = 0; // 5

```

```
int main ()
{
    Test T;
}
```

11. Что выведет следующий код при создании экземпляра класса D?

```
struct A { A() { cout << "A"; } };
struct B : virtual A { B() { cout << "B"; } };
struct C : virtual A { C() { cout << "C"; } };
struct D : B, C { D() { cout << "D"; } };
```

12. Что произойдет при компиляции и выполнении кода?

```
class A {
public:
    A() { f("A()"); }
    ~A() { f("~A()"); }
protected:
    virtual void f(const char* str) = 0;
};

class B : public A {
public:
    B() { f("B()"); }
    ~B() { f("~B()"); }
protected:
    void f(const char* str) { cout<< str << endl; }
};

int main() {
    B b;
    return 0;
}
```

13. Допустимо ли в C++ определение следующего чисто виртуального метода:

```
class Abstract
{
public:
    virtual void pureVirtual() = 0 {
        // реализация
    }
};
```

14. Корректно ли описание метода modify в составе класса Test?

```
class Test {
private:
```

```
    mutable int mX;  
public:  
    void modify( const int iNewX ) const {  
        mX = iNewX;  
    }  
};
```

№ 5 Классы-контроллеры и методы- итераторы

Использовать проект созданный в практикуме №4.

Определить класс Сущность (указан в вариантах) для хранения разных типов объектов (в пределах иерархии) в виде списка или массива. Класс Сущность должен содержать методы get и set для списка/массива, методы для добавления и удаления объектов в список/массив, метод для вывода списка на консоль.

Определить управляющий класс Контроллер, который управляет объектом- Сущностью и реализовать в нем запросы по варианту:

Вариант 1	Собрать (установить) разное ПО на Компьютер (хранить в виде списка или массива). Найти Игрушки определенного типа и <i>Word</i> заданной версии, вывести все ПО в алфавитном порядке.
Вариант 2	Создать Предприятие из имеющихся работников. Вывести имена токарей со стажем не менее заданного, <i>подсчитать количество студентов-заочников на предприятии и рабочих заданного года рождения.</i>
Вариант 3	Создать Армию из людей и трансформеров. Найти в армии боевую единицу заданного года рождения (создания), <i>вывести имена трансформеров заданной мощности, количество боевых единиц в армии.</i>
Вариант 4	Создать несколько объектов-цветов. Собрать Букет (используя аксессуары). Определить его стоимость. <i>Провести сортировку цветов в букете на основе размера цветка. Найти цветок в букете, соответствующий заданному диапазону длин стеблей.</i>
Вариант 5	Создать Лабораторию и наполнить ее техникой. Найти технику старше заданного возраста. <i>Подсчитать количество каждого вида техники. Вывести список техники в порядке убывания цены.</i>
Вариант 6	Создать Библиотеку с книгами, журналами и учебниками. Вывести наименование всех книг в библиотеке, вышедших не ранее указанного года; <i>найти суммарное количество учебников в библиотеке, подсчитать стоимость изданий, находящихся в библиотеке.</i>
Вариант 7	Создать Сессию , содержащую зачеты и экзамены. Найти все экзамены по заданному предмету, <i>подсчитать общее количество испытаний в сессии и количество тестов с заданным числом вопросов.</i>
Вариант 8	Создать Программу передач . Найти все фильмы, снятые в определенный год, <i>подсчитать продолжительность программы по времени, число рекламных роликов.</i>
Вариант 9	Собрать Подарок . Расчитать цену подарка. Найти в подарке компонент с наименьшей массой. <i>Произвести сортировку компонентов по габаритам.</i>
Вариант 10	Собрать Бухгалтерию . Найти суммарную стоимость продукции заданного наименования по всем накладным, <i>количество чеков. Вывести две документы за указанный период времени.</i>

Вариант 11	Создать частое Транспортное агентство . Подсчитать стоимость всех транспортных средств. <i>Провести сортировку автомобилей по расходу топлива. Найти транспортное в компании, соответствующий заданному диапазону параметров скорости.</i>
Вариант 12	Подготовить Спортзал . Снарядов должно быть фиксированное количество в пределах выделенной суммы денег. Провести сортировку инвентаря в Спортзале по одному из параметров. <i>Найти снаряды, соответствующие заданному диапазону цены.</i>
Вариант 13	Создать Планету Земля . Найти все государства на заданном континенте, <i>подсчитать количество морей, вывести острова по алфавиту.</i>
Вариант 14	Создать Зоопарк . Найти средний вес животных заданного вида в зоопарке, <i>количество хищных птиц, вывести всех животных отсортированных по году рождения.</i>
Вариант 15	Создать Порт . Найти среднее водоизмещение всех парусников в порту, <i>среднее количество посадочных мест на парходах, все транспортные средства на которых плавают капитаны моложе 35 лет.</i>
Вариант 16	Собрать Детский подарок с определением его веса. Провести сортировку конфет в подарке на основе одного из параметров. <i>Найти конфету в подарке, соответствующую заданному диапазону содержания сахара.</i>
Вариант 17	Собрать Салат . Подсчитать калорийность салата. Провести сортировку овощей для салата на основе одного из параметров. <i>Найти овощи в салате, соответствующие заданному диапазону калорийности.</i>
Вариант 18	Собрать Ожерелье . Подсчитать общий вес (в каратах) и стоимость. Провести сортировку камней ожерелья на основе ценности. <i>Найти камни в ожерелье, соответствующие заданному диапазону параметров прозрачности.</i>
Вариант 19	Создать Авиакомпанию . Посчитать общую вместимость и грузоподъемность. <i>Провести сортировку самолетов компании по дальности полета. Найти самолет в компании, соответствующий заданному диапазону параметров потребления горючего.</i>
Вариант 20	Создать Сотового оператора . Подсчитать общую численность тарифов. Провести сортировку тарифов на основе размера абонентской платы. <i>Найти тариф в компании,</i>

	<i>соответствующий заданному диапазону параметров.</i>
<i>Вариант 21</i>	Загрузить Фургон определенного объема грузом на определенную сумму из различных сортов кофе, находящихся, к тому же, в разных физических состояниях. Учитывать объем кофе. Провести сортировку товаров на основе соотношения цены и веса. <i>Найти в фургоне товар, соответствующий заданному диапазону параметров качества.</i>
<i>Вариант 22</i>	Создать Банк . Клиент может иметь несколько счетов в банке. Учитывать возможность блокировки/разблокировки счета. Реализовать поиск и сортировку счетов. <i>Вычисление общей суммы по счетам заданного клиента. Вычисление суммы по всем счетам, имеющим положительный и отрицательный балансы отдельно.</i>
<i>Вариант 23</i>	Создать Интерфейс программного средства . Реализовать поиск всех кнопок, меню заданного уровня вложенности. <i>Рассчитать площадь свободного места в окне</i>
<i>Вариант 24</i>	Создать Интерфейс программного средства . Подсчитать количество текстовых, количество интерактивных элементов управления.
<i>Вариант 25</i>	Создать Компьютерный класс . Найти общую стоимость всей техники, вывести оборудование в порядке убывания одного из параметров. <i>Найти оборудование, которое должно быть списано (политику списания определить самостоятельно)</i>
<i>Вариант 26</i>	Создать Склад. Найти общую стоимость шкафов. <i>Вывести мебель заданного производителя (либо другого параметра). . Провести сортировку товаров на основе соотношения цены и веса</i>

№ 6 Перегрузка операций

Создать заданный в варианте класс. Определить в классе конструкторы, деструктор, необходимые функции и заданные перегруженные операции. Написать программу тестирования, в которой проверяется использование перегруженных операций.

<i>Вариант 1</i>	Класс – Одномерный массив . Дополнительно перегрузить следующие операции: * – умножение массивов; [] – доступ по индексу, int() – размер массива; == – проверка на равенство; <= – сравнение.
<i>Вариант 2</i>	Класс – Одномерный массив . Дополнительно перегрузить следующие операции: [] – доступ по индексу; > – проверка на вхождение; != – проверка на неравенство; + – объединение массивов.
<i>Вариант 3</i>	Класс – множество Set . Дополнительно перегрузить следующие операции: + – добавить элемент в множество (типа set+item); + – объединение множеств; * – пересечение множеств; int() – мощность множества.
<i>Вариант 4</i>	Класс – множество Set . Дополнительно перегрузить следующие операции: - – удалить элемент из множества (типа set-item); * – пересечение множеств; < – сравнение множеств; > – проверка на подмножество; int() – мощность множества.
<i>Вариант 5</i>	Класс – множество Set . Дополнительно перегрузить следующие операции: - – удалить элемент из множества (типа set-item); > – проверка на подмножество; != – проверка множеств на неравенство; + – добавить элемент в множество (типа set+item); * – пересечение множеств.
<i>Вариант 6</i>	Класс – однонаправленный список List . Дополнительно перегрузить следующие операции: () – удалить элемент в заданной позиции, например: int i; list L; L(i); () – добавить элемент в заданную позицию, например: int i; Type c; list L; L(c,i); != – проверка на неравенство.
<i>Вариант 7</i>	Класс – множество Set . Дополнительно перегрузить следующие операции: () – конструктор множества (в стиле конструктора для множественного типа); + – объединение множеств; <= – сравнение множеств; int() – мощность множества; [] – доступ к элементу в заданной позиции.

--	--

<i>Вариант 8</i>	Класс – множество Set . Дополнительно перегрузить следующие операции: > – проверка на принадлежность (типа операции in множественного типа* – пересечение множеств; < – проверка на подмножество; int()– мощность множества; [] - доступ к элементу в заданной позиции.
<i>Вариант 9</i>	Класс – однонаправленный список List . Дополнительно перегрузить следующие операции: + – объединить два списка; -- – удалить элемент из начала (--list); == – проверка на равенство; bool() – проверка, пустой ли список.
<i>Вариант 10</i>	Класс – двунаправленный список List . Дополнительно перегрузить следующие операции: + – добавить элемент в начало (item+list); -- – удалить элемент из начала (--list); != – проверка на неравенство; * - объединение двух списков.
<i>Вариант 11</i>	Класс – однонаправленный список List . Дополнительно перегрузить следующие операции: + – добавить элемент в конец (list+item); -- – удалить элемент из конца (типа list--); != – проверка на неравенство; [] - доступ к элементу в заданной позиции.
<i>Вариант 12</i>	Класс - однонаправленный список List . Дополнительно перегрузить следующие операции: [] - доступ к элементу в заданной позиции; + - объединить два списка; == - проверка на равенство; < - добавление одного списка к другому.
<i>Вариант 13</i>	Класс - стек Stack . Дополнительно перегрузить следующие операции: + - добавить элемент в стек; -- - извлечь элемент из стека; bool() - проверка, пустой ли стек; > - копирование одного стека в другой с сортировкой в возрастающем порядке.
<i>Вариант 14</i>	Класс - очередь Queue . Дополнительно перегрузить следующие операции: + - добавить элемент; -- - извлечь элемент; bool() - проверка, пустая ли очередь; < - копирование одной очереди в другую с сортировкой в убывающем порядке; int()– мощность.
<i>Вариант 15</i>	Класс - Вектор . Дополнительно перегрузить следующие операции: + - сложение векторов; ()- доступ по индексу V(i); > - сравнение векторов; == - копирование вектора.
<i>Вариант 16</i>	Класс - Матрица . Дополнительно перегрузить следующие операции: + - сложение матриц; ()- доступ по индексу V(i) к заданной строке; > - сравнение матриц по модулю; == - копирование матрицы.
<i>Вариант 17</i>	Класс - Матрица . Дополнительно перегрузить следующие операции: - - вычитания числа из всех элементов матрицы; ++ инкремент всех элементов матрицы; != - сравнение матриц по

	модулю; <code>int()</code> – количество нулевых элементов в матрице.
--	--

<i>Вариант 18</i>	Класс - Марица . Дополнительно перегрузить следующие операции: + - сложение матриц; -- обнуление всех элементов матрицы; == - сравнение матриц по нулевому столбцу; int() – количество отрицательных элементов в матрице.
<i>Вариант 19</i>	Класс - Марица . Дополнительно перегрузить следующие операции: < - сравнения матриц; >= приведение матрицы к единичному виду; == - сравнение матриц по первому элементу; * – инверсия всех элементов матрицы.
<i>Вариант 20</i>	Класс - стек Stack . Дополнительно перегрузить следующие операции: * - добавить элемент в стек; /- извлечь элемент из стека; bool() - проверка, есть ли в стеке отрицательные элементы; == - сравнения стеков.
<i>Вариант 21</i>	Класс - стек Stack . Дополнительно перегрузить следующие операции: - - извлечение всех элементов равных заданному; ++ - дублирование верхнего элемента; <= копирование неповторяющихся элементов из второго стека.
<i>Вариант 22</i>	Класс - очередь Queue . Дополнительно перегрузить следующие операции: / - добавить элемент; ++ - извлечь элемент; bool() - проверка, на содержание четных элементов в очереди; int()– количество положительных элементов в очереди
<i>Вариант 23</i>	Класс - Строка . Дополнительно перегрузить следующие операции: < - сравнения строк в лексикографическом порядке; + добавления числа к строке; - удаление последнего символа в строке; * – замена всех символов в строке на заданный.
<i>Вариант 24</i>	Класс - Строка . Дополнительно перегрузить следующие операции: < - удаление всех символов равных заданному; + удаление нечетных символов; != сравнение длин строк; [] – доступ к символу строки по индексу.
<i>Вариант 25</i>	Класс – Строка . Дополнительно перегрузить следующие операции: - – удалить элемент из строки из заданной позиции (типа set-item); > – проверка на вхождение подстроки; != – проверка строк на неравенство; + – добавить элемент в строку на заданную позицию (типа string+item).
<i>Вариант 26</i>	Класс – Пароль . Дополнительно перегрузить следующие операции: - – замена последнего символа (типа password-item); > – сравнение длин паролей; != – проверка паролей на неравенство; ++ – сброс пароля на значение по умолчанию; bool() - проверка на строичность.

Пример

```
#include<iostream>
```



```

class Complex
{
public:
double re, im;
Complex(double r = 0.0, double i = 0.0) { re = r; im = i; }

friend Complex operator + (const Complex& x, const Complex& y)
{ return Complex(x.re + y.re, x.im + y.im); }

friend Complex operator - (const Complex& x, const Complex& y)
{ return Complex(x.re - y.re, x.im - y.im); }

friend Complex operator * (const Complex& x, const Complex& y)
{ return Complex(x.re * y.re - x.im * y.im, x.re * y.im +
y.re * x.im); }

friend Complex operator / (const Complex& x, const Complex& y)
{
double z = y.re * y.re + y.im * y.im;
return Complex((x.re * y.re + x.im * y.im) / z,
(y.re * x.im - x.re * y.im) / z);
}

friend ostream& operator << (ostream& output, const Complex&
x)
{ return output<<x.re<<" + i"<<x.im; }

friend istream& operator >> (istream& input, Complex& x)
{ return input>>x.re>>x.im; }

friend int operator == (const Complex& x, const Complex& y)
{ return (x.re == y.re) && (x.im == y.im); }

friend int operator != (const Complex& x, const Complex& y)
{ return (x.re != y.re) || (x.im != y.im); }
};

void main()
{
Complex a, b;
cin>>a>>b;
cout<<a<<endl<<b<<endl<<a+b<<endl<<a-
b<<endl<<a*b<<endl<<a/b<<endl;
if (a == b) cout<<"a a ȳ® b"<<endl;
if (a != b) cout<<"a Γ a ȳ® b"<<endl;
}

```

Теория

В языке С++ определены множества операций над переменными стандартных типов, такие как +, -, *, / и т.д. Каждую операцию можно применить к операндам определенного типа.

К сожалению, лишь ограниченное число типов непосредственно поддерживается любым языком программирования. Например, С и С++ не позволяют выполнять операции с комплексными числами, матрицами, строками, множествами. Однако, все эти операции можно выполнить через классы в языке С++.

Рассмотрим пример.

Пусть заданы множества А и В:

$A = \{ a_1, a_2, a_3 \};$

$B = \{ a_3, a_4, a_5 \},$

и мы хотим выполнить операции объединения (+) и пересечения (*) множеств.

$A + B = \{ a_1, a_2, a_3, a_4, a_5 \}$

$A * B = \{ a_3 \}.$

Можно определить класс Set - "множество" и определить операции над объектами этого класса, выразив их с помощью знаков операций, которые уже есть в языке С++, например, + и *. В результате операции + и * можно будет использовать как и раньше, а также снабдить их дополнительными функциями (объединения и пересечения). Как определить, какую функцию должен выполнять оператор: старую или новую? Очень просто – по типу операндов. А как быть с приоритетом операций? Сохраняется определенный ранее приоритет операций. Для распространения действия операции на новые типы данных надо определить специальную функцию, называемую "операция-функция" (operator-function). Ее формат:

*тип_возвр_значения operator знак_операции(специф_параметров)
{операторы_тела_функции}*

При необходимости может добавляться и прототип:

тип_возвр_значения operator знак_операции(специф_параметров);

Если принять, что конструкция operator знак_операции есть имя некоторой функции, то прототип и определение операции-функции подобны прототипу и определению обычной функции языка С++. Определенная таким образом операция называется перегруженной (overload).

Чтобы была обеспечена явная связь с классом, операция-функция должна быть либо компонентом класса, либо она должна быть определена в классе как дружественная и у нее должен быть хотя бы один параметр типа класс (или ссылка на класс). Вызов операции-функции осуществляется так же, как и любой другой функции С++: operator @. Однако разрешается использовать сокращенную форму ее вызова: a @ b, где @ - знак операции.

Перегрузка унарных операций.

Любая унарная операция @ может быть определена двумя способами: либо как компонентная функция без параметров, либо как глобальная (возможно дружественная) функция с одним параметром. В первом случае выражение @ Z означает вызов Z.operator @(), во втором - вызов operator @(Z).

Унарные операции, перегружаемые в рамках определенного класса, могут перегружаться только через нестатическую компонентную функцию без параметров. Вызываемый объект класса автоматически воспринимается как операнд.

Унарные операции, перегружаемые вне области класса (как глобальные функции), должны иметь один параметр типа класса. Передаваемый через этот параметр объект воспринимается как операнд.

Синтаксис:

а) в первом случае (описание в области класса):

тип_возвр_значения operator знак_операции

б) во втором случае (описание вне области класса):

тип_возвр_значения operator знак_операции (идентификатор_типа)

Пример.

```
class Person {
public:
    void operator ++()
    {    ++age;    }
protected:
    int age;
    ...
};
int main(void)
{    Person John;
    ++John;
    ...
}

class Person {
protected:
    int age;
    ...
friend void operator ++(Person &);
};
void operator ++(Person &ob)
{    ++ob.age;}
int main (void)
{    Person John;
```

```

    ++John;
    ...
}

```

Перегрузка бинарных операций.

Любая бинарная операция @ может быть определена двумя способами: либо как компонентная функция с одним параметром, либо как глобальная (возможно дружественная) функция с двумя параметрами. В первом случае `x @ y` означает вызов `x.operator @(y)`, во втором – вызов `operator @(x, y)`.

Операции, перегружаемые внутри класса, могут перегружаться только нестатическими компонентными функциями с параметрами. Вызываемый объект класса автоматически воспринимается в качестве первого операнда. Операции, перегружаемые вне области класса, должны иметь два операнда, один из которых должен иметь тип класса.

Пример.

```

class Person { ... };
class AdressBook
{
public:
    Person &operator [](int); // доступ к i
protected:
    // содержит в качестве компонентных данных множество объектов типа
    // Person представляемых как динамический массив, список или дерево
    ...
};
Person &AdressBook::operator [](int i)
{ ... }
int main(void)
{
    AdressBook persons;
    Person record;
    ...
    record = persons[3];
    ...
}
class Person { ... };
class AdressBook
{
protected:
    // содержит в качестве компонентных данных множество объектов типа
    // Person представляемых как динамический массив, список или дерево
    ...
friend Person &operator [](const AdressBook &, int); // доступ к i
};
Person &AdressBook::operator [](const AdressBook &, int i)
{ ... }

int main(void)

```

```

{   AdressBook persons;
    Person record;
    ...
    record = persons[3];
    ...
}

```

Перегрузка операций ++ и --.

Унарные операции инкремента ++ и декремента -- существуют в двух формах: префиксной и постфиксной. В современной спецификации C++ определен способ, по которому компилятор может различить эти две формы. В соответствии с этим способом задаются две версии функции operator ++() и operator --(). Они определены следующим образом:

Префиксная форма:

```

operator ++();
operator --();

```

Постфиксная форма:

```

operator ++(int);
operator --(int);

```

Указание параметра int для постфиксной формы не специфицирует второй операнд, а используется только для отличия от префиксной формы.

Пример.

```

class Person
{public:
    ...
    void operator ++()
    {++age; }
    void operator ++(int)
    {age++; }
protected:
    int age;
    ...
};
int main(void)
{Person John;
John++;
++John;
}

```

Перегрузка операции вызова функции.

Это операция '()'. Она является бинарной операцией. Первым операндом обычно является объект класса, вторым – список параметров.

Пример.

```
class Matrix // двумерный массив вещественных чисел
{
public:
...
double operator()(int, int); //доступ к элементам матрицы по индексам
};
double Matrix::operator() (int i, int j)
{ ... }
int main (void)
{
Matrix a;
double k;
...
k = a(5, 6);
...
}
```

Перегрузка операции присваивания.

Операция отличается тремя особенностями:

- ✓ операция не наследуется;
- ✓ операция определена по умолчанию для каждого класса в качестве операции поразрядного копирования объекта, стоящего справа от знака операции, в объект, стоящий слева.
- ✓ операция может перегружаться только в области определения класса. Это гарантирует, что первым операндом всегда будет леводопустимое выражение.

Если вас устраивает поразрядное копирование, нет смысла создавать собственную функцию `operator=()`. Однако бывают случаи, когда поразрядное копирование нежелательно. Например, использование предопределенной операции присваивания для классов, содержащих указатели в качестве компонентных данных, чаще всего приводит к ошибкам. Покажем это на примере.

Пользовательский класс - строка `String`:

```
class String
{public:
    String(char *);
    ~String();
    void show();
}
```

```

protected:
    char *p; // указатель на строку
    int len; // текущая длина строки
};
String::String(char *ptr)
{
    len = strlen(ptr);
    p = new char[len + 1];
    if (!p)
    {
        cout << "Ошибка выделения памяти\n";
        exit(1);
    }
    strcpy(p, ptr);
}
String::~String(){ delete [] p;}
void String::show(void){ cout << p << "\n";}
int main(void)
{
    String s1("Это первая строка"),
           s2("А это вторая строка");
    s1.show();
    s2.show();
    s2 = s1; // Это ошибка
    s1.show();
    s2.show();
    return 0;
}

```

В чем здесь ошибка? Когда объект s1 присваивается объекту s2, указатель p объекта s2 начинает указывать на ту же самую область памяти, что и указатель p объекта s1. Таким образом, когда эти объекты удаляются, память, на которую указывает указатель p объекта s1, освобождается дважды, а память, на которую до присваивания указывал указатель p объекта s2, не освобождается вообще.

Хотя в данном примере эта ошибка и не опасна, в реальных программах с динамическим распределением памяти она может вызвать крах программы. В этом случае необходимо самим перегрузить операцию присваивания. Покажем как это сделать для нашего класса String.

```

class String
{public:
    ...
    String &operator =(String &);

protected:
    char *p; // указатель на строку
    int len; // текущая длина строки
};
String &String::operator =(String &ob);
{ if (this == &ob) return *this;

```

```

if (len < ob.len)
{
    // требуется выделить дополнительную память
    delete [] p;
    p = new char[ob.len + 1];
    if (!p)
    {
        cout << "Ошибка выделения памяти\n";
        exit(1);
    }
}
len = ob.len;
strcpy(p, ob.p);
return *this;
}

```

В этом примере выясняется, не происходит ли самоприсваивание (типа `ob = ob`). Если имеет место самоприсваивание, то просто возвращается ссылка на объект.

Затем проверяется, достаточно ли памяти в объекте, стоящем слева от знака присваивания, для объекта, стоящего справа от знака присваивания. Если не достаточно, то память освобождается и выделяется новая, требуемого размера. Затем строка копируется в эту память.

Отметим две важные особенности функции `operator =()`. Во-первых, в ней используется параметр-ссылка. Это необходимо для предотвращения создания копии объекта, передаваемого через параметр по значению. В случае создания копии, она удаляется вызовом деструктора при завершении работы функции. Но деструктор освобождает память, на которую указывает `p`. Однако эта память все еще необходима объекту, который является аргументом. Параметр-ссылка помогает решить эту проблему.

Во-вторых, функция `operator =()` возвращает не объект, а ссылку на него. Смысл этого тот же, что и при использовании параметра-ссылки. Функция возвращает временный объект, который удаляется после завершения ее работы. Это означает, что для временной переменной будет вызван деструктор, который освобождает память по адресу `p`. Но она необходима для присваивания значения объекту. Поэтому, чтобы избежать создания временного объекта, в качестве возвращаемого значения используется ссылка.

Другой путь решения проблем, описанных выше - это создание конструктора копирования. Но конструктор копирования может оказаться не столь эффективным решением, как ссылка в качестве параметра и ссылка в качестве возвращаемого значения функции. Это происходит потому, что использование ссылки исключает затраты ресурсов, связанных с копированием объектов в каждом из двух указанных случаев.

Перегрузка операции new.

Синтаксис

Операция new, заданная по умолчанию, может быть в двух формах:

- 1) new тип <инициализирующее выражение>
- 2) new тип [];

Первая форма используется не для массивов, вторая - для массивов.

Перегруженную операцию new можно определить в следующих формах, соответственно для не массивов и для массивов:

```
void *operator new (size_t t, остальные аргументы);  
void *operator new [] (size_t t, остальные аргументы);
```

Первый и единственный обязательный аргумент t всегда должен иметь тип size_t. Если аргумент имеет тип size_t, то в операцию-функцию new автоматически подставляется аргумент sizeof(t), т.е. она получает значение, равное размеру объекта t в байтах.

Например, пусть задана следующая функция:

```
void *operator new(size_t t, int n)  
{ return new char [t * n]; }
```

и она вызывается следующим образом:

```
double *d = new(5) double;  
Здесь t = double, n = 5.
```

В результате после вызова значение t в теле функции будет равно sizeof(double).

При перегрузке операции new появляется несколько глобальных операций new, одна из которых определена в самом языке по умолчанию, а другие являются перегруженными. Возникает вопрос: как различить такие операции? Это делается путем изменения числа и типов их аргументов. При изменении только типов аргументов может возникнуть неоднозначность, являющаяся следствием возможных преобразований этих типов друг к другу. При возникновении такой неоднозначности следует при обращении к new задать тип явно, например:

```
new((double) 5) double;
```

Одна из причин, по которой перегружается операция new, состоит в стремлении придать ей дополнительную семантику, например, обеспечение

диагностической информацией или устойчивости к сбоям. Кроме того класс может обеспечить более эффективную схему распределения памяти чем та, которую предоставляет система.

В соответствии со стандартом C++ в заголовочном файле <new> определены следующие функции-операции new, позволяющие передавать, наряду с обязательным первым size_t аргументом и другие.

```
void *operator new(size_t t) throw (bad_alloc);
void *operator new(size_t t, void *p) throw ();
void *operator new(size_t t, const nothrow &) throw ();
void *operator new(size_t t, allocator &a);
void *operator new [](size_t t) throw (bad_alloc);
void *operator new [](size_t t, void *p) throw ();
void *operator new [](size_t t, const nothrow &) throw ();
```

Эти функции используют генерацию исключений (throw) и собственный распределитель памяти (allocator).

Версия с nothrow выделяет память как обычно, но если выделение заканчивается неудачей, возвращается 0, а не генерируется bad_alloc. Это позволяет нам для выделения памяти использовать стратегию обработки ошибок до генерации исключения.

Правила использования операции new

- ✓ Объекты, организованные с помощью new имеют неограниченное время жизни. Поэтому область памяти должна освобождаться оператором delete.
- ✓ Если резервируется память для массива, то операция new возвращает указатель на первый элемент массива.
- ✓ При резервировании памяти для массива все размерности должны быть выражены положительными величинами.
- ✓ Массивы нельзя инициализировать.
- ✓ Объекты классов могут организовываться с помощью операции new, если класс имеет конструктор по умолчанию.
- ✓ Ссылки не могут организовываться с помощью операции new, так как для них не выделяется память.
- ✓ Операция new самостоятельно вычисляет потребность в памяти для организуемого типа данных, поэтому первый параметр операции всегда имеет тип size_t.

Обработка ошибок операции new.

Обработка ошибок операции new происходит в два этапа:

Устанавливается, какие предусмотрены функции для обработки ошибок. Собственные функции должны иметь тип new_handler и создаются с помощью функции set_new_handler. В файле new.h объявлены

```
typedef void (*new_handler)();  
new_handler set_new_handler(new_handler new_p);
```

Вызывается соответствующая new_handler функция. Эта функция должна:

- ✓ либо вызвать bad_alloc исключение;
- ✓ либо закончить программу;
- ✓ либо освободить память и попытаться распределить ее заново.

Диагностический класс bad_alloc объявлен в new.h.

В реализации VC++ включена специальная глобальная переменная _new_handler, значением которой является указатель на new_handler функцию, которая выполняется при неудачном завершении new. По умолчанию, если операция new не может выделить требуемое количество памяти, формируется исключение bad_alloc. Изначально это исключение называлось xalloc и определялось в файле except.h. Исключение xalloc продолжает использоваться во многих компиляторах. Тем не менее, оно вытесняется определенным в стандарте C++ именем bad_alloc.

Рассмотрим несколько примеров.

Пример 1. В примере использование блока try ... catch дает возможность проконтролировать неудачную попытку выделения памяти.

```
#include <iostream>  
#include <new>  
int main(void)  
{  
    double *p;  
    try {  
        p = new double[1000];  
        cout << "Память выделилась успешно" << endl;  
    }  
    catch (bad_alloc xa) {  
        cout << "Ошибка выделения памяти\n";  
        cout << xa.what();  
        return 1;  
    }  
    return 0;  
}
```

Пример 2. Поскольку в предыдущем примере при работе в нормальных условиях ошибка выделения памяти маловероятна, в этом примере ошибка выделения памяти достигается принудительно. Процесс выделения памяти длится до тех пор, пока не произойдет ошибка.

```
#include <iostream>  
#include <new>  
int main(void)  
{
```

```

double *p;
do
{
    try {
        p = new double[1000];
        cout << "Память выделилась успешно" << endl;
    }
    catch (bad_alloc xa) {
        cout << "Ошибка выделения памяти\n";
        cout << xa.what();
    }
}
while (p);
return 0;
}

```

Пример 3. Демонстрируется перегруженная форма операции new - операция new (nothrow).

```

#include <iostream>
#include <new>
int main(void)
{
    double *p;
    struct nothrow noth_ob;
    do {
        p = new(noth_ob) double[1000];
        if (!p) cout << "Ошибка выделения памяти\n";
        else cout << "Память выделилась успешно\n";
    }
    while (p);
    return 0;
}

```

Пример 4. Демонстрируются различные формы перегрузки операции new.

```

#include <iostream.h>
#include <new.h>
double *p, *q, **pp;
class Demo
{
public:
    Demo() { value = 0; }
    Demo(int i) { value = i; }
    void *operator new(size_t, int, int);
    void *operator new(size_t, int);
    void *operator new(size_t, char *);
protected:
    int value;
};
void *Demo::operator new(size_t t, int i, int j)
{
    if (j) return new(i) Demo;
    return NULL;
}

```

```

void *Demo::operator new(size_t t, int i)
{
    Demo *p = ::new Demo;
    (*p).value = i;
    return p;
}
void *Demo::operator new(size_t t, char *z)
{
    return ::new (z) Demo;
}
int main(void)
{
    Demo *p_ob1, *p_ob2;
    struct nothrow noth_ob;
    p = new double;
    pp = new double *;
    p = new double(1.2); // инициализация
    q = new double[3];    // массив
    p_ob1 = new Demo[10]; // массив объектов demo
    void (**f_ptr)(int) // указатель на указатель на функцию
    f_ptr = new(void(*[3])(int)) // массив указателей на функцию
    char z[sizeof(Demo)]; // резервируется память в соответствии с
    величиной demo
    p_ob2 = new(z) Demo; // организуется демо-объект в области памяти на
                        // которую указывает переменная z
    p_ob2 = new(3) Demo; // демо-объект с инициализацией
    p_ob1 = new(3, 0) Demo; // возвращает указатель NULL
    p_ob2 = new(nothrow) Demo[5]; // массив демо-объектов,
                                // в случае ошибки возвращает NULL
    return 0;
}

```

Перегрузка операции delete.

Операция-функция delete бывает двух видов:

```

void operator delete(void *);
void operator delete(void *, size_t);

```

Вторая форма включает аргумент типа size_t, передаваемый delete. Он передается компилятору как размер объекта, на который указывает p.

Особенностью перегрузки операции delete является то, что глобальные операции delete не могут быть перегружены. Их можно перегрузить только по отношению к классу.

Основные правила перегрузки операций.

- ✓ Вводить собственные обозначения для операций, не совпадающие со стандартными операциями языка C++, нельзя.
- ✓ Не все операции языка C++ могут быть перегружены. Нельзя перегрузить следующие операции:
 - . – прямой выбор компонента,
 - .* – обращение к компоненту через указатель на него,
 - ? : – условная операция,
 - :: – операция указания области видимости,
 - sizeof, # и ## – препроцессорные операции.
- ✓ Каждая операция, заданная в языке, имеет определенное число операндов, свой приоритет и ассоциативность. Все эти правила, установленные для операций в языке, сохраняются и для ее перегрузки, т.е. изменить их нельзя.
- ✓ Любая унарная операция @ определяется двумя способами: либо как компонентная функция без параметров, либо как глобальная (возможно дружественная) функция с одним параметром. Выражение @z означает в первом случае вызов z.operator @(), во втором - вызов operator @(z).
- ✓ Любая бинарная операция @ определяется также двумя способами: либо как компонентная функция с одним параметром, либо как глобальная (возможно дружественная) функция с двумя параметрами. В первом случае x @ y означает вызов x.operator @(y), во втором – вызов operator @(x, y).
- ✓ Перегруженная операция не может иметь аргументы (операнды), заданные по умолчанию.
- ✓ В языке C++ установлена идентичность некоторых операций, например, ++z – это тоже, что и z += 1. Эта идентичность теряется для перегруженных операций.
- ✓ Функцию operator можно вызвать по ее имени, например, z = operator * x, y) или z = x.operator *(y). В первом случае вызывается глобальная функция, во втором – компонентная функция класса X, и x – это объект класса X. Однако, чаще всего функция operator вызывается косвенно, например, z = x * y.
- ✓ За исключением перегрузки операций new и delete функция operator должна быть либо нестатической компонентной функцией, либо иметь как минимум один аргумент (операнд) типа "класс" или "ссылка на класс" (если это глобальная функция).
- ✓ Операции '=', '[]', '—>' можно перегружать только с помощью нестатической компонентной функции operator @. Это гарантирует, что первыми операндами будут леводопустимые выражения.
- ✓ Операция '[]' рассматривается как бинарная. Пусть a – объект класса A, в котором перегружена операция '[]'. Тогда выражение a[i] интерпретируется как a.operator [](i).

- ✓ Операция '()' вызова функции рассматривается как бинарная. Пусть *a* – объект класса *A*, в котором перегружена операция '()'. Тогда выражение *a*(*x*₁, *x*₂, *x*₃, *x*₄) интерпретируется как *a.operator* ()(*x*₁, *x*₂, *x*₃, *x*₄).
- ✓ Операция '→' доступа к компоненту класса через указатель на объект этого класса рассматривается как унарная. Пусть *a* – объект класса *A*, в котором перегружена операция '→'. Тогда выражение *a*→*m* интерпретируется как (*a.operator*→())→*m*. Это означает, что функция *operator* →() должна возвращать указатель на класс *A*, или объект класса *A*, или ссылку на класс *A*.
- ✓ Перегрузка операций '++' и '--', записываемых после операнда (*z*++, *z*--), отличается добавлением в функцию *operator* фиктивного параметра *int*, который используется только как признак отличия операций *z*++ и *z*-- от операций ++*z* и --*z*.
- ✓ Глобальные операции *new* можно перегрузить и в общем случае они могут не иметь аргументов (операндов) типа "класс". В результате разрешается иметь несколько глобальных операций *new*, которые различаются путем изменения числа и (или) типов аргументов.
- ✓ Глобальные операции *delete* не могут быть перегружены. Их можно перегрузить только по отношению к классу.
- ✓ Заданные в самом языке глобальные операции *new* и *delete* можно изменить, т.е. заменить версию, заданную в языке по умолчанию, на свою версию.
- ✓ Локальные функции *operator new*() и *operator delete*() являются статическими компонентами класса, в котором они определены, независимо от того, использовался или нет спецификатор *static* (это, в частности, означает, что они не могут быть виртуальными).
- ✓ Для правильного освобождения динамической памяти под базовый и производный объекты следует использовать виртуальный деструктор.
- ✓ Если для класса *X* операция "=" не была перегружена явно и *x* и *y* – это объекты класса *X*, то выражение *x* = *y* задает по умолчанию побайтовое копирование данных объекта *y* в данные объекта *x*.
- ✓ Функция *operator* вида *operator type*() без возвращаемого значения, определенная в классе *A*, задает преобразование типа *A* к типу *type*.
- ✓ За исключением операции присваивания '=' все операции, перегруженные в классе *X*, наследуются в любом производном классе *Y*.
- ✓ Пусть *X* – базовый класс, *Y* – производный класс. Тогда локально перегруженная операция для класса *X* может быть далее повторно перегружена в классе *Y*.

Вопросы

1. Каково назначение перегрузки операторов?
2. Как используется ключевое слово `operator`?

3. Можно ли перегрузкой отменить очередность выполнения операции?
4. Истинно ли следующее утверждение: операция `>=` может быть перегружена.
5. Сколько аргументов требуется для определения перегруженной унарной операции?
6. Когда вы перегружаете операцию арифметического присваивания куда передается результат?
7. Истинно ли следующее утверждение: выражение `objA = objB` будет причиной ошибки компилятора, если объекты разных типов?
8. Можно ли перегрузкой отменить число операндов?
9. Какие операции требуют, чтобы левый операнд был объектом класса?
10. Какие операторы нельзя перегружать?
11. Запишите приведенный ниже пример как дружественный оператор класса `dat`:

```
dat    dat::operator+(int n)
{
    dat    x;
    x      = *this;
    while (n-- != 0) x.next();
    return(x);
}
```

12. Наследуется ли перегрузка операции присваивания?
13. Может ли операция `=` переопределяться вне области определения класса?
14. Можно ли использовать операцию `=`, если она не определена?
15. Можно ли перегружать операцию `[]` с использованием friend-функции?
16. Можно ли перегружать операцию `()` с использованием friend-функции?
17. Можно ли перегружать операцию `->` с использованием friend-функции?
18. Может ли перегруженная операция `delete` возвращать значение `void`?
19. Может ли перегруженная операция `delete` возвращать значение `int`?
20. Можно ли перегружать операцию `delete` с использованием friend-функции?
21. Для чего используется оператор `dynamic_cast`?
22. Для чего используется оператор `const_cast`?

- 23. Для чего используется оператор `static_cast`?
- 24. Для чего используется оператор `reinterpret_cast`?
- 25. Как называется вызов функции, обрабатываемый во время выполнения программы?
- 26. Что будет выведено на экран?

```
class A {  
public:  
    A(){ };  
    ~A(){ };  
    explicit A(int a);  
    operator int(){return 1;}  
};  
  
int main(int argc, char* argv[])  
{  
    A foo;  
    int value = foo + 1;  
    std::cout << value << std::endl;  
    return 0;  
}
```

- 27. Как реализовать перегрузку инкремента и декремента в постфиксной и префиксной формах?

№ 7 Шаблоны классов

Модифицировать проект, созданный в предыдущем практикуме №6, следующим образом. Создать шаблон заданного класса. Проверить использование шаблона для стандартных типов данных (в качестве стандартных типов использовать целые и вещественные типы). Определить пользовательский класс, который будет использоваться в качестве параметра шаблона. Для пользовательского типа взять класс из проекта практикума №1-2 или 3.

Примеры

Матрица

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

template <class X>
class MATRIX_3x3
{
    X matrix[3][3];
public:
    MATRIX_3x3<X>::MATRIX_3x3(X n11, X n12, X n13,
                               X n21, X n22, X n23,
                               X n31, X n32, X n33);

    MATRIX_3x3 <X> MATRIX_3x3 <X>::operator+(X n);
    MATRIX_3x3 <X> MATRIX_3x3 <X>::operator-(X n); //
    MATRIX_3x3 <X> MATRIX_3x3 <X>::operator*(X n); //
    MATRIX_3x3 <X> MATRIX_3x3 <X>::operator+(MATRIX_3x3 &n); //
    MATRIX_3x3 <X> MATRIX_3x3 <X>::operator-(MATRIX_3x3 &n); //
    MATRIX_3x3 <X> MATRIX_3x3 <X>::operator*(MATRIX_3x3 &n);

};

template <class X>
MATRIX_3x3 <X> MATRIX_3x3 <X>::operator+(X n)
{
    return (MATRIX_3x3 <X> (
        this->matrix[0][0]+n, this->matrix[0][1]+n, this->matrix[0][2]+n,
        this->matrix[1][0]+n, this->matrix[1][1]+n, this->matrix[1][2]+n,
        this->matrix[2][0]+n, this->matrix[2][1]+n, this->matrix[2][2]+n)
    );
}

template <class X>
```

```

MATRIX_3x3 <X> MATRIX_3x3 <X>::operator-(X n)
{
    return(MATRIX_3x3 <X>(
        this->matrix[0][0]-n,this->matrix[0][1]-n,this->matrix[0][2]-n,
        this->matrix[1][0]-n,this->matrix[1][1]-n,this->matrix[1][2]-n,
        this->matrix[2][0]-n,this->matrix[2][1]-n,this->matrix[2][2]-n));
}

template <class X>
MATRIX_3x3 <X> MATRIX_3x3 <X>::operator*(X n)
{
    return(MATRIX_3x3 <X>(
        this->matrix[0][0]*n,this->matrix[0][1]*n,this->matrix[0][2]*n,
        this->matrix[1][0]*n,this->matrix[1][1]*n,this->matrix[1][2]*n,
        this->matrix[2][0]*n,this->matrix[2][1]*n,this->matrix[2][2]*n));
}

template <class X>
MATRIX_3x3 <X> MATRIX_3x3 <X>::operator+(MATRIX_3x3 &n)
{
    return(MATRIX_3x3 <X>(
        this->matrix[0][0]+n.matrix[0][0],
        this->matrix[1][0]+n.matrix[1][0],
        this->matrix[2][0]+n.matrix[2][0],
        this->matrix[0][1]+n.matrix[0][1],
        this->matrix[1][1]+n.matrix[1][1],
        this->matrix[2][1]+n.matrix[2][1],
        this->matrix[0][2]+n.matrix[0][2],
        this->matrix[1][2]+n.matrix[1][2],
        this->matrix[2][2]+n.matrix[2][2]));
}

template <class X>
MATRIX_3x3 <X> MATRIX_3x3 <X>::operator-(MATRIX_3x3 &n)
{
    return(MATRIX_3x3 <X>(
        this->matrix[0][0]-n.matrix[0][0],
        this->matrix[1][0]-n.matrix[1][0],
        this->matrix[2][0]-n.matrix[2][0],
        this->matrix[0][1]-n.matrix[0][1],
        this->matrix[1][1]-n.matrix[1][1],
        this->matrix[2][1]-n.matrix[2][1],
        this->matrix[0][2]-n.matrix[0][2],
        this->matrix[1][2]-n.matrix[1][2],
        this->matrix[2][2]-n.matrix[2][2]));
}

```

```

template <class X>
MATRIX_3x3 <X> MATRIX_3x3 <X>::operator* (MATRIX_3x3 &n)
{
    return (MATRIX_3x3 <X> (
//-----
        /*c11*/
        this->matrix[0][0]*n.matrix[0][0]+
        this->matrix[1][0]*n.matrix[0][1]+
        this->matrix[2][0]*n.matrix[0][2],
        /*c12*/
        this->matrix[0][0]*n.matrix[1][0]+
        this->matrix[1][0]*n.matrix[1][1]+
        this->matrix[2][0]*n.matrix[1][2],
        /*c13*/
        this->matrix[0][0]*n.matrix[2][0]+
        this->matrix[1][0]*n.matrix[2][1]+
        this->matrix[2][0]*n.matrix[2][2],
//-----
        /*c21*/
        this->matrix[0][1]*n.matrix[0][0]+
        this->matrix[1][1]*n.matrix[0][1]+
        this->matrix[2][1]*n.matrix[0][2],
        /*c22*/
        this->matrix[0][1]*n.matrix[1][0]+
        this->matrix[1][1]*n.matrix[1][1]+
        this->matrix[2][1]*n.matrix[1][2],
        /*c23*/
        this->matrix[0][1]*n.matrix[2][0]+
        this->matrix[1][1]*n.matrix[2][1]+
        this->matrix[2][1]*n.matrix[2][2],
//-----
        /*c31*/
        this->matrix[0][2]*n.matrix[0][0]+
        this->matrix[1][2]*n.matrix[0][1]+
        this->matrix[2][2]*n.matrix[0][2],
        /*c32*/
        this->matrix[0][2]*n.matrix[1][0]+
        this->matrix[1][2]*n.matrix[1][1]+
        this->matrix[2][2]*n.matrix[1][2],
        /*c33*/
        this->matrix[0][2]*n.matrix[2][0]+
        this->matrix[1][2]*n.matrix[2][1]+
        this->matrix[2][2]*n.matrix[2][2]));
}

template <class X>
MATRIX_3x3<X>::MATRIX_3x3(X n11, X n12, X n13,
                           X n21, X n22, X n23,
                           X n31, X n32, X n33)
{
    this->matrix[0][0]=n11; this->matrix[1][0]=n12; this->
matrix[2][0]=n13;

```

```

    this->matrix[0][1]=n21; this->matrix[1][1]=n22; this-
>matrix[2][1]=n23;
    this->matrix[0][2]=n31; this->matrix[1][2]=n32; this-
>matrix[2][2]=n33;
}

void main (void)
{
    MATRIX_3x3 <int> mat(1,2,3,4,5,6,7,8,9); //
    mat=mat+3; //
    mat=mat-2; //
    mat=mat*4; //
    MATRIX_3x3 <float> mat1(1,2,3,
                           4,5,6,
                           7,8,9);
    MATRIX_3x3 <float> mat2(1,2,3,
                           4,5,6,
                           7,8,9);

    mat1=mat1+mat2; //
    mat1=mat1-mat2; //
    MATRIX_3x3<int> mat3(3,0,1,
                        0,2,7,
                        1,-3,2);
    MATRIX_3x3<int> mat4(-6,1,11,
                        9,2,5,
                        0,3,7);

    mat3=mat3*mat4; //
    clrscr();
    getch();
}

```

Массив

```

...
...

//shablon classa massiv
template <class theType>
class Array
{
    theType *a;//pointer to the array
    int count;//number of elements
    int numberToInput;//number of elements to input using
operator >>
public:
//empty constructor
    Array()
    {
        a=NULL;
        count=0;
        numberToInput=0;
    }

//constructor too

```

```

Array(int count_)
{
    a=new theType [count_];
    count=count_;
    for (int i=0 ; i<count ; i++)
        a[i]=0;
    numberToInput=0;
}

//consructor, which copys the array to the object
Array(theType *a_, int count_)
{
    a=new theType [count_];
    for (int i=0 ; i<count_ ; i++)
        a[i]=a_[i];
    count=count_;
    numberToInput=0;
}

//copy constructor
Array(Array& a_)
{
    a=new theType [a_.count];
    count=a_.count;
    for (int i=0 ; i<count ; i++)
        a[i]=a_.a[i];
    numberToInput=a_.numberToInput;
}

//destructor
~Array()
{
    delete [] a;
    numberToInput=0;
    count=0;
}

void setNumberToInput(int n)
{
    numberToInput=n;
}

//operator =
Array& operator = (Array& a_)
{
    if (!(a_==(*this)))
    {
        count=a_.count;
        numberToInput=a_.numberToInput;
        for (int i=0 ; i<count ; i++)
            a[i]=a_.a[i];
    }
    return *this;
}

```

```

    }

//operator <<
friend ostream& operator <<(ostream& out, Array& a_)
{
    for (int i=0 ; i<a_.count ; i++)
        out<<a_.a[i]<<" ";
    out<<endl;
    return out;
}

//operator >>
friend istream& operator >>(istream& in, Array& a_)
{
    cout<<"Input started:\n";
    if(a_.a) delete a_.a;
    a_.a=new theType [a_.numberToInput];
    for (int i=0 ; i<a_.numberToInput ; i++)
    {
        in>>a_.a[i];
        a_.count++;
    }
    a_.numberToInput=0;
    cout<<"\nInput ended.\n\n";
    return in;
}

theType& operator [] (int i_)
{
    if (i_<count && i_>=0)
        return a[i_];
}

//operator ()
int operator () ()
{
    return count;
}

};

#include "Array.h"

int main()
{
    //Part 1:
    {
        int *ar;
        int n;
        cout<<"\n\n\t\t<<< Part 1. >>>\n\n";
        cout<<"Template parameter: int.\n\n";
        cout<<"Let's input array of integers. Enter number of
elements: ";
    }
}

```

```

        cin>>n;
        ar=new int [n]; //create massive
        cout<<"Start input.\n";
        for (int i=0 ; i<n ; i++)
            cin>>ar[i];
        cout<<"\nEnd input.\n";
        Array<int> a(ar,n); //+constructor
        cout<<"Print a:\n"<<a<<endl;
        cout<<"\nArray b will be created by copying Array
a.\n";
        Array<int> b(a);
        cout<<"Print b:"<<b<<endl;
        cout<<"We'll create Array c with 10 elements. All
elements = 0.\n";
        Array<int> c(10);
        cout<<"Print c:\n"<<c<<endl;
        cout<<"Let's change some elements in c. And then we'll
print c:\n";
        c[0]=15;
        c[5]=17;
        c[9]=-12;
        cout<<c<<endl;
        cout<<"\nCheck operator int():\n\ta():
"<<a()<<"\n\tb(): "<<b()<<endl;
        cout<<"\nChecking operator []:\n\ta[0]: "<<a[0]<<endl;
        cout<<"\n\n\t\t<<< End of part 1. >>>\n\n\n";
    }
    //Part 2:
    {
        double *ar;
        int n;
        cout<<"\n\n\t\t<<< Part 2. >>>\n\n";
        cout<<"Template parameter: double.\n\n";
        cout<<"Let's input array of double. Enter number of
elements: ";
        cin>>n;
        ar=new double [n];
        cout<<"Start input.\n";
        for (int i=0 ; i<n ; i++)
            cin>>ar[i];
        cout<<"\nEnd input.\n";
        Array<double> a(ar,n);
        cout<<"Print a:\n"<<a<<endl;
        cout<<"\nArray b will be created by copying Array
a.\n";
        Array<double> b(a);
        cout<<"Print b:"<<b<<endl;
        cout<<"We'll create Array c with 10 elements. All
elements = 0.\n";
        Array<double> c(10);
        cout<<"Print c:\n"<<c<<endl;
        cout<<"Let's change some elements in c. And then we'll
print c:\n";

```



```

        c[0]=15.45;
        c[5]=17.1;
        c[9]=-12.004;
        cout<<c<<endl;
        cout<<"\nCheck operator int():\n\ta():
"<<a()<<"\n\tb(): "<<b()<<endl;
        cout<<"\nChecking operator []:\n\ta[0]: "<<a[0]<<endl;
        cout<<"\n\n\t\t<<< End of part 2. >>>\n\n\n";
    }
    return 0;
}

```

Теория

Аналогично шаблонам функций. определяется шаблон семейства классов:

template<список_параметров_шаблона> определение_класса

Шаблон семейства классов определяет способ построения отдельных классов подобно тому, как класс определяет правила построения и формат отдельных объектов. В определении класса, входящего в шаблон, особую роль играет имя класса. Оно является не именем отдельного класса, а параметризованным именем семейства классов.

Как и для шаблонов функций, определение шаблона класса может быть только глобальным.

Следуя авторам языка и компилятора C++, рассмотрим векторный класс (в число данных входит одномерный массив). Какой бы тип ни имели элементы массива (целый, вещественный, с двойной точностью и т.д.), в этом классе должны быть определены одни и те же базовые операции, например доступ к элементу по индексу и т.д. Если тип элементов вектора задавать как параметр шаблона класса, то система будет формировать вектор нужного типа (и соответствующий класс) при каждом определении конкретного объекта.

Следующий шаблон автоматически формирует классы векторов с указанными свойствами:

```

// vector.h - шаблон векторов
template<class T> // T - параметр шаблона
class Vector
{
public:
    Vector(int); // Конструктор класса vector

```

```

~Vector() // Деструктор
{
    delete [] data;
}

// Расширение действия (перегрузка) операции "[]":
T &operator [](int i)
{
    return data [i];
}

protected:
    T *data; // Начало одномерного массива
    int size; // Количество элементов в массиве
};

// vector.cpp
// Внешнее определение конструктора класса:
template<class T> Vector <T>::Vector(int n)
{
    data = new T[n];
    size = n;
};

```

Когда шаблон введен, у программиста появляется возможность определять конкретные объекты конкретных классов, каждый из которых параметрически порожден из шаблона. Формат определения объекта одного из классов, порождаемых шаблоном классов:

имя_параметризованного_класса <фактические_параметры_шаблона>

имя_объекта (параметры_конструктора);

В нашем случае определить вектор, имеющий восемь вещественных координат типа double, можно следующим образом:

```
Vector<double> z(8);
```

Проиллюстрируем сказанное следующей программой:

```

// формирование классов с помощью шаблона
#include <iostream>

#include "vector.h" // Шаблон класса "вектор"

int main(void)
{

```

```

    Vector<int> X(5);           //Создаем объект класса "целочисленный
    вектор"
    Vector<char> C(5);         // Создаем объект класса "символьный вектор"
    for (int i = 0; i < 5; i++) // Определяем компоненты векторов
    {
        X[i] = i;
        C[i] = 'A' + i;
    }
    for (i = 0; i < 5; i++)
    {
        cout << X[i] << C[i]; // 0 A 1 B 2 C 3 D 4 E
    }
    return 0;
}

```

В программе шаблон семейства классов с общим именем Vector используется для формирования двух классов с массивами целого и символьного типов. В соответствии с требованием синтаксиса имя параметризованного класса, определенное в шаблоне (в примере Vector), используется в программе только с последующим конкретным фактическим параметром (аргументом), заключенным в угловые скобки. Параметром может быть имя стандартного или определенного пользователем типа. В данном примере использованы стандартные типы int и char. Использовать имя Vector без указания фактического параметра шаблона нельзя - никакое умалчиваемое значение при этом не предусматривается.

В списке параметров шаблона могут присутствовать формальные параметры, не определяющие тип, точнее - это параметры, для которых тип фиксирован:

```

#include <iostream>

template<class T, int size = 64> class Row
{
public:
    Row()
    {
        length = size;
        data = new T [size];
    }

    ~Row()
    {
        delete [] data;
    }

    T &operator [](int i)
    {
        return data [i];
    }
}

```

```

    }

protected:
    T *data;
    int length;
};

int main(void)
{
    Row<float, 8> rf;
    Row<int, 8> ri;
    for (int i = 0; i < 8; i++)
    {
        rf[i] = i;
        ri[i] = i * i;
    }
    for (i = 0; i < 8, i++)
    {
        cout << rf[i] << ri[i]; //0 0 1 1 2 4 3 9 4 16 5 25 6 36 7 49
    }
    return 0;
}

```

В качестве аргумента, заменяющего при обращении к шаблону параметр size, взята константа. В общем случае может быть использовано константное выражение, однако выражения, содержащие переменные, использовать в качестве фактических параметров шаблонов нельзя.

Основные свойства шаблонов классов.

1. Компонентные функции параметризованного класса автоматически являются параметризованными. Их не обязательно объявлять как параметризованные с помощью template.
2. Дружественные функции, которые описываются в параметризованном классе, не являются автоматически параметризованными функциями, т.е. по умолчанию такие функции являются дружественными для всех классов, которые организуются по данному шаблону.
3. Если friend-функция содержит в своем описании параметр типа параметризованного класса, то для каждого созданного по данному шаблону класса имеется собственная friend-функция.
4. В рамках параметризованного класса нельзя определить friend-шаблоны (дружественные параметризованные классы).
5. С одной стороны, шаблоны могут быть производными (наследоваться) как от шаблонов, так и от обычных классов, с другой стороны, они могут использоваться в качестве базовых для других шаблонов или классов.
6. Шаблоны функций, которые являются членами классов, нельзя описывать как virtual.

7. Локальные классы не могут содержать шаблоны в качестве своих элементов.
8. Реализация компонентной функции шаблона класса, которая находится вне определения шаблона класса, должна включать дополнительно следующие два элемента:
9. Определение должно начинаться с ключевого слова `template`, за которым следует такой же список_параметров_типов в угловых скобках, какой указан в определении шаблона класса.
10. За именем_класса, предшествующим операции области видимости (`::`), должен следовать список_имен_параметров шаблона.

`template<список_типов>`

тип_возвр_значения

имя_класса<список_имен_параметров>::имя_функции(список_параметров)
{ ... }

Smart-указатель.

Рассмотрим еще один пример использования класса-шаблона. С его помощью мы попытаемся "усовершенствовать" указатели языка Си++. Если указатель указывает на объект, выделенный с помощью операции `new`, необходимо явно вызывать операцию `delete` тогда, когда объект становится не нужен. Однако далеко не всегда просто определить, нужен объект или нет, особенно если на него могут ссылаться несколько разных указателей. Разработаем класс, который ведет себя очень похоже на указатель, но автоматически уничтожает объект, когда уничтожается последняя ссылка на него. Назовем этот класс smart-указатель (Smart Pointer). Идея заключается в том, что настоящий указатель мы окружим специальной оболочкой. Вместе со значением указателя мы будем хранить счетчик - сколько других объектов на него ссылается. Как только значение этого счетчика станет равным нулю, объект, на который указатель указывает, пора уничтожать.

Структура `Ref` хранит исходный указатель и счетчик ссылок.

```
template<class T> struct Ref
{
    T *realPtr;
    int counter;
};
```

Теперь определим интерфейс smart-указателя:

```

template<class T> class SmartPtr
{
public:
    // конструктор из обычного указателя
    SmartPtr(T *ptr = 0);
    // копирующий конструктор
    SmartPtr(const SmartPtr &s);
    ~SmartPtr();
    SmartPtr &operator =(const SmartPtr &s);
    SmartPtr &operator =(T *ptr);
    T *operator ->() const;
    T &operator *() const;
private:
    Ref<T> *refPtr;
};

```

У класса SmartPtr определены операции обращения к элементу ->, взятия по адресу * и операции присваивания. С объектом класса SmartPtr можно обращаться практически так же, как с обычным указателем.

```

struct A
{
    int x;
    int y;
};

SmartPtr<A> aPtr(new A); // создать новый указатель
int x1 = aPtr->x;        // обратиться к элементу A
(*aPtr).y = 3;           // обратиться по адресу

```

Рассмотрим реализацию методов класса SmartPtr. Конструктор инициализирует объект указателем. Если указатель равен нулю, то refPtr устанавливается в ноль. Если же конструктору передается ненулевой указатель, то создается структура Ref, счетчик обращений в которой устанавливается в 1, а указатель - в переданный указатель:

```

template<class T> SmartPtr<T>::SmartPtr(T *ptr)
{
    if (!ptr)
    {
        refPtr = 0;
    }
    else
    {
        refPtr = new Ref<T>;
        refPtr->realPtr = ptr;
        refPtr->counter = 1;
    }
}

```

Деструктор уменьшает количество ссылок на 1 и, если оно достигло 0, уничтожает объект

```
template<class T> SmartPtr<T>::~~SmartPtr ()
{
    if (refPtr)
    {
        refPtr->counter--;
        if (refPtr->counter <= 0)
        {
            delete refPtr->realPtr;
            delete refPtr;
        }
    }
}
```

Реализация операций -> и * довольно проста:

```
template<class T> T *SmartPtr<T>::operator ->() const
{
    if (refPtr) return refPtr->realPtr;
    else return 0;
}
```

```
template<class T> T &SmartPtr<T>::operator *() const
{
    if (refPtr) return *refPtr->realPtr;
    else throw bad_pointer;
}
```

Самые сложные для реализации - копирующий конструктор и операции присваивания. При создании объекта SmartPtr - копии имеющегося - мы не будем копировать сам исходный объект. Новый smart-указатель будет ссылаться на тот же объект, мы лишь увеличим счетчик ссылок.

```
template<class T> SmartPtr<T>::SmartPtr(const SmartPtr &s)
: refPtr(s.refPtr)
{
    if (refPtr) refPtr->counter++;
}
```

При выполнении присваивания, прежде всего, нужно отсоединиться от имеющегося объекта, а затем присоединиться к новому, подобно тому, как это сделано в копирующем конструкторе.

```
template<class T> SmartPtr &SmartPtr<T>::operator =(const SmartPtr &s)
{

```

```

// отсоединиться от имеющегося указателя
if (refPtr)
{
    refPtr->counter--;
    if (refPtr->counter <= 0)
    {
        delete refPtr->realPtr;
        delete refPtr;
    }
}
// присоединиться к новому указателю
refPtr = s.refPtr;
if (refPtr) refPtr->counter++;
}

```

В следующей функции при ее завершении объект класса Complex будет уничтожен:

```

void foo(void)
{
    SmartPtr<Complex> complex(new Complex);
    SmartPtr<Complex> ptr = complex;
}

```

Задание свойств класса.

Одним из методов использования шаблонов является уточнение поведения с помощью дополнительных параметров шаблона. Предположим, мы пишем функцию сортировки вектора:

```

template<class T> void sort_vector(vector<T> &vec)
{
    for (int i = 0; i < vec.size() - 1; i++)
        for (int j = i; j < vec.size(); j++)
        {
            if (vec[i] < vec[j])
            {
                T tmp = vec[i];
                vec[i] = vec[j];
                vec[j] = tmp;
            }
        }
}

```

Эта функция будет хорошо работать с числами, но если мы захотим использовать ее для массива указателей на строки (char *), то результат будет несколько неожиданный. Сортировка будет выполняться не по значению

строк, а по их адресам (операция "меньше" для двух указателей - это сравнение значений этих указателей, т.е. адресов величин, на которые они указывают, а не самих величин). Чтобы исправить данный недостаток, добавим к шаблону второй параметр:

```
template<class T, class Compare> void sort_vector(vector<T> &vec)
{
    for (int i = 0; i < vec.size() - 1; i++)
    {
        for (int j = i; j < vec.size(); j++)
        {
            if (Compare::less (vec[i], vec[j]))
            {
                T tmp = vec[i];
                vec[i] = vec[j];
                vec[j] = tmp;
            }
        }
    }
}
```

Класс Compare должен реализовывать статическую функцию less, сравнивающую два значения типа T. Для целых чисел этот класс может выглядеть следующим образом:

```
class CompareInt
{
    static bool less(int a, int b)
    {
        return a < b;
    }
};
```

Сортировка вектора будет выглядеть так:

```
vector<int> vec;
sort<int, CompareInt>(vec);
```

Для указателей на байт (строк) можно создать класс

```
class CompareCharStr
{
    static bool less(char *a, char *b)
    {
        return strcmp(a, b) >= 0;
    }
};
```

и, соответственно, сортировать с помощью вызова

```
vector<char*> svec;  
sort<char*, CompareCharStr>(svec);
```

Как легко заметить, для всех типов, для которых операция "меньше" имеет нужный нам смысл, можно написать шаблон класса сравнения:

```
template<class T> Compare  
{  
    static bool less(T a, T b)  
    {  
        return a < b;  
    }  
};
```

и использовать его в сортировке (обратите внимание на пробел между закрывающимися угловыми скобками в параметрах шаблона; если его не поставить, компилятор спутает две скобки с операцией сдвига):

```
vector<double> dvec;  
sort<double, Compare<double>>(dvec);
```

Чтобы не загромождать запись, воспользуемся возможностью задать значение параметра по умолчанию. Так же, как и для аргументов функций и методов, для параметров шаблона можно определить значения по умолчанию. Окончательный вид функции сортировки будет следующий:

```
template<class T, class C = Compare<T>>  
void sort_vector(vector<T> &vec)  
{  
    for (int i = 0; i < vec.size() - 1; i++)  
    {  
        for (int j = i; j < vec.size(); j++)  
        {  
            if (C::less(vec[i], vec[j]))  
            {  
                T tmp = vec[i];  
                vec[i] = vec[j];  
                vec[j] = tmp;  
            }  
        }  
    }  
}
```

Второй параметр шаблона иногда называют параметром-штрих, поскольку он лишь модифицирует поведение класса, который манипулирует типом, определяемым первым параметром.

№ 8 Шаблоны функций

Модифицировать проект, созданный в предыдущем практикуме №7. Написать функцию-шаблон в соответствии с вариантом. Проверить работу функции с int, double , пользовательским классом (для манипуляций выбрать одно из полей класса).

Вариант 1	Написать функцию-шаблон, вычисляющую значение в массиве меньше заданного.
Вариант 2	Написать функцию-шаблон - сортировка методом подсчета.
Вариант 3	Написать функцию-шаблон, суммирующую элементы в множестве.
Вариант 4	Написать функцию-шаблон уменьшающую каждый элемент множества на заданное значение
Вариант 5	Написать параметризованную функцию сортировки методом быстрой сортировки.
Вариант 6	Создать шаблон функции прореживания: три параметра: откуда выбирать, куда выбирать и через сколько выбирать.
Вариант 7	Написать функцию-шаблон, вычисляющую среднее значение в множестве.
Вариант 8	Написать функцию-шаблон, вычисляющую минимальное значение в множестве.
Вариант 9	Написать функцию-шаблон, вычисляющую минимальное значение в списке.
Вариант 10	Создать шаблон функции, которая возвращает максимально близкое к заданному элементу из значений списка. Аргументами функции должны быть значение, адрес списка.
Вариант 11	Написать функцию-шаблон, удваивающую элементы в списке.
Вариант 12	Написать функцию-шаблон, удаляющую элементы равные заданному в списке.
Вариант 13	Написать функцию-шаблон, которая подсчитывает количество элементов больших заданному в стеке
Вариант 14	Создать шаблон функции прореживания: два параметра: откуда выбирать и через сколько выбирать.
Вариант 15	Создать шаблон функцию min, которая возвращает максимальное отрицательное значение вектора
Вариант 16	Написать функцию-шаблон, которая подсчитывает количество элементов больших заданному в матрице
Вариант 17	Написать функцию-шаблон, которая возвращает сумму всех элементов матрицы
Вариант 18	Написать функцию-шаблон, которая обнуляет элементы в матрице равные заданному.
Вариант 19	Написать функцию-шаблон, которая возвращает максимально близкое заданному
Вариант 20	Написать функцию-шаблон, которая возвращает количество элементов в стеке меньших заданного
Вариант 21	Написать функцию-шаблон, которая удаляет из стека

	элементы больше заданного
Вариант 22	Написать функцию-шаблон, которая возвращает количество четных элементов в очереди
Вариант 23	Написать функцию-шаблон, которая шифрует строку в соответствии с паролем
Вариант 24	Написать функцию-шаблон, которая прореживает строку, удаляя каждый n-ый
Вариант 25	Написать функцию-шаблон, которая возвращает количество элементов в строке равных заданному
Вариант 26	Написать функцию-шаблон, которая шифрует пароль (алгоритм придумать самим)

Пример

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

template <class YYY> class Complex
{
    YYY real, imaginary;
public:
    Complex() { cin >> *this; }
    friend ostream& operator << (ostream&, const Complex&);
    friend istream& operator >> (istream&, Complex&);
    Complex operator + (Complex);
    Complex operator - (Complex);
    Complex operator * (Complex);
    Complex operator / (Complex);
    void operator == (Complex);
    void operator != (Complex);
};

template <class YYY>
ostream& operator << (ostream& output, const Complex< YYY >& C)
{
    if(C.imaginary>=0)
        output << C.real << "+" << C.imaginary << "*i"<< endl;
    else
        output << C.real << C.imaginary << "*i"<< endl;
    return output;
}

template <class YYY >
istream& operator >> (istream& input, Complex< YYY >& C)
{
    cout << "vvedite veschestvennuy chast -> "; input >> C.real;
    cout << "vvedite mnimuy chast -> "; input >> C.imaginary;
    return input;
}

template <class YYY >
```

```

Complex< YYY > Complex< YYY >::operator + (Complex< YYY > C)
{
    C.real=real+C.real; C.imaginary=imaginary+C.imaginary;
    return C;
}

template <class YYY >
Complex< YYY > Complex< YYY >::operator - (Complex< YYY > C)
{
    C.real=real-C.real; C.imaginary=imaginary-C.imaginary;
    return C;
}

template <class YYY >
Complex< YYY > Complex< YYY >::operator * (Complex< YYY > C)
{
    double REAL=C.real, IMAGINARY=C.imaginary;
    C.real=real*REAL - imaginary*IMAGINARY;
    C.imaginary=imaginary*REAL + real*IMAGINARY;
    return C;
}

template <class YYY >
Complex< YYY > Complex< YYY >::operator / (Complex< YYY > C)
{
    double REAL=C.real, IMAGINARY=C.imaginary;
    C.real=(real*REAL + imaginary*IMAGINARY)/(REAL*REAL +
IMAGINARY*IMAGINARY);
    C.imaginary=(imaginary*REAL - real*IMAGINARY)/(REAL*REAL +
IMAGINARY*IMAGINARY);
    return C;
}

template <class YYY >
void Complex< YYY >::operator == (Complex< YYY > C)
{
    if(C.real==real&&C.imaginary==imaginary) cout << "chisla
ravni"<<endl;
    else cout << "chisla neravni";
}

template <class YYY >
void Complex<YYY>::operator != (Complex< YYY > C)
{
    if(C.real!=real||C.imaginary!=imaginary) cout << "chisla
neravni"<<endl;
    else cout << "chisla ravni"<<endl;
}

template <class YYY >
void vybor(Complex< YYY >& c1, Complex< YYY >& c2)

```

```

{ int O;
  cout << " 1) slozit(+)\n 2) vichest(-)\n";
  cout << " 3) umnozit(*)\n 4) delit(/)\n";
  cout << " 5) sravnenie na ravenstvo(==)\n";
  cout << " 6) sravnenie na neravenstvo(!=)\n";
  cin >> O;
  switch(O)
  { case 1: cout << "otvet: " << c1+c2; break;
    case 2: cout << "otvet: " << c1-c2; break;
    case 3: cout << "otvet: " << c1*c2; break;
    case 4: cout << "otvet: " << c1/c2; break;
    case 5: c1==c2; break;
    case 6: c1!=c2; break;
  }
}

void main()
{
  clrscr();
  cout << "\t\t\t1-e kompleksnoe chislo tipa float:" << endl;
  Complex<float> Complex_Float_1;
  cout << "\t\t\t2-e kompleksnoe chislo tipa float:" << endl;
  Complex<float> Complex_Float_2;
  vybor(Complex_Float_1, Complex_Float_1);

  clrscr();
  cout << "\t\t\t1-e kompleksnoe chislo tipa int:" << endl;
  Complex<int> Complex_Int_1;
  cout << "\t\t\t2-e kompleksnoe chislo tipa int:" << endl;
  Complex<int> Complex_Int_2;
  vybor(Complex_Int_1, Complex_Int_2);
}

```

Теория

Шаблоны функций.

Шаблоны, которые называют иногда родовыми или параметризованными типами, позволяют создавать (конструировать) семейства родственных функций и классов.

Цель введения шаблонов функций - автоматизация создания функций, которые могут обрабатывать разнотипные данные. В отличие от механизма перегрузки, когда для каждого набора формальных параметров определяется своя функция, шаблон семейства функций определяется один раз, но это определение параметризуется. Параметризовать в шаблоне функций можно тип возвращаемого функцией значения и типы любых параметров, количество и порядок размещения которых должны быть фиксированы. Для параметризации используется список параметров шаблона.

В определении шаблона семейства функций используется служебное слово `template`. Для параметризации используется список формальных

параметров шаблона, который заключается в угловые скобки <>. Каждый формальный параметр шаблона обозначается служебным словом `class`, за которым следует имя параметра (идентификатор). Пример определения шаблона функций, вычисляющих абсолютные значения числовых величин разных типов:

```
template<class type> type abs(type x)
{
    return x > 0 ? x: -x;
}
```

Описание шаблона семейства функций состоит из двух частей:

```
template<class тип_данных>
тип_возвр_значения имя_функции(список_параметров)
{тело_функции}
```

В качестве еще одного примера рассмотрим шаблон семейства функций для обмена значений двух передаваемых им параметров.

```
template<class T> void swap(T *x, T *y)
{
    T z = *x;
    *x = *y;
    *y = z;
}
```

Здесь параметр `T` шаблона функций используется не только в заголовке для спецификации формальных параметров, но и в теле определения функции, где он задает тип вспомогательной переменной `z`.

Шаблон семейства функций служит для автоматического формирования конкретных определений функций по тем вызовам, которые транслятор обнаруживает в теле программы. Например, если программист употребляет обращение `abs(-10.3)`, то на основе приведенного ранее шаблона компилятор сформирует такое определение функции:

```
double abs(double x)
{ return x > 0 ? x: -x; }
```

Далее будет организовано выполнение именно этой функции и в точку вызова в качестве результата вернется числовое значение 10.3.

Если в программе присутствует приведенный ранее шаблон семейства функций `swap()`

```
long k = 4, d = 8;
```



```
swap(&k, &d);
```

то компилятор сформирует определение функции:

```
void swap(long *x, long *y)
{
    long x = *x;
    *x = *y;
    *y = x;
}
```

Затем будет выполнено обращение именно к этой функции и значения переменных k, d поменяются местами.

Если в той же программе присутствуют операторы:

```
double a = 2.44, b = 66.3;
swap(&a, &b);
```

то сформируется и выполнится функция

```
void swap(double *x, double *y)
{
    double x = *x;
    *x = *y;
    *y = x;
}
```

Проиллюстрируем сказанное о шаблонах на более конкретном примере. Рассмотрим программу, используем некоторые возможности функций, возвращающих значение типа "ссылка". Но тип ссылки будет определяться параметром шаблона:

```
#include <iostream>
```

```
//Функция определяет ссылку на элемент с максимальным значением
```

```
template<class type> type &rmax(int n, type d[])
```

```
{
    int im = 0;
    for (int i = 1; i < n; i++)
        im = d[im] > d[i] ? im: i;
    return d[im];
}
```

```
int main(void)
```

```
{
    int n = 4;
    int x[] = { 10, 20, 30, 14 }; //Массив целых чисел
    cout << "\nrmax(n,x) = " << rmax(n, x); // rmax(n,x) = 30
    rmax(n, x) = 0;
    for (int i = 0; i < n; i++)
        cout << "\tx[" << i << "] = " << x[i]; // x[0] = 10 x[1] ...
    float arx[] = { 10.3, 20.4, 10.5 }; //Массив вещественных чисел
    cout << "\nrmax(3,arx) = " << rmax(3, arx); //rmax(3,arx) = 20.4
}
```

```

rmax(3, arx) = 0;
for (int i = 0; i < 3; i++)
cout << "\tarx[" << i << "] =" << arx[i]; //arx[0] = 10.3 ...
return 0;
}

```

В программе используются два разных обращения к функции `rmax()`. В одном случае параметр - целочисленный массив и возвращаемое значение - ссылка типа `int`. Во втором случае фактический параметр - имя массива типа `float` и возвращаемое значение имеет тип ссылки на `float`.

По существу механизм шаблонов функций позволяет автоматизировать подготовку переопределений перегруженных функций. При использовании шаблонов уже нет необходимости готовить заранее все варианты функций с перегруженным именем. Компилятор автоматически, анализируя вызовы функций в тексте программы, формирует необходимые определения именно для таких типов параметров, которые использованы в обращениях. Дальнейшая обработка выполняется так же, как и для перегруженных функций.

Параметры шаблонов.

Можно считать, что параметры шаблона являются его формальными параметрами, а типы тех параметров, которые используются в конкретных обращениях к функции, служат фактическими параметрами шаблона. Именно по ним выполняется параметрическая настройка и с учетом этих типов генерируется конкретный текст определения функции. Однако, говоря о шаблоне семейства функций, обычно употребляют термин "список параметров шаблона", не добавляя определения "формальных".

Перечислим основные свойства параметров шаблона:

1. Имена параметров шаблона должны быть уникальными во всем определении шаблона.
2. Список параметров шаблона не может быть пустым.
3. В списке параметров шаблона может быть несколько параметров, и каждому из них должно предшествовать ключевое слово `class`.

```
template<class type1, class type2>
```

Соответственно, неверен заголовок:

```
template<class type1, type2, type3>
```

4. Недопустимо использовать в заголовке шаблона параметры с одинаковыми именами, то есть ошибочен такой заголовок:

```
template<class t, class t, class t>
```

5. Имя параметра шаблона (в примерах - `type1`, `type2`) имеет в определяемой шаблоном функции все права имени типа, то есть с его помощью могут специализироваться формальные параметры,

определяться тип возвращаемого функцией значения и типы любых объектов, локализованных в теле функции. Имя параметра шаблона видно во всем определении и скрывает другие использования того же идентификатора в области, глобальной по отношению к данному шаблону функций. Если внутри тела определяемой функции необходим доступ к внешним объектам с тем же именем, нужно применять операцию изменения области видимости. Следующая программа иллюстрирует указанную особенность имени параметра шаблона функций:

```
#include <iostream>

int N = 0; //статическая, инициализирована нулем

template<class N> N max(N x, N y)
{
    N a = x;
    cout << "\nСчетчик обращений N = " << ++:N;
    if (a < y) a = y;
    return a;
}

int main(void)
{
    int a = 12, b = 42;
    max(a, b); //Счетчик обращений N = 1
    float z = 66.3, f = 222.4;
    max(z, f); //Счетчик обращений N = 2
}
```

Итак, одно имя нельзя использовать для обозначения нескольких параметров одного шаблона, но в разных шаблонах функций могут быть одинаковые имена у параметров шаблонов. Ситуация здесь такая же, как и у формальных параметров при определении обычных функций, и на ней можно не останавливаться подробнее. Действительно, раз действие параметра шаблона заканчивается в конце определения шаблона, то соответствующий идентификатор свободен для последующего использования, в том числе и в качестве имени параметра другого шаблона. Все параметры шаблона функций должны быть обязательно использованы в спецификациях параметров определения функции. Таким образом, будет ошибочным такой шаблон:

```
template<class A, class B, class C>
B func(A n, C m)
{
    B value;
    ...
}
```

В данном неверном примере остался неиспользованным параметр шаблона с именем B. Его применение в качестве типа возвращаемого функцией значения и для определения объекта value в теле функции недостаточно.

Определенная с помощью шаблона функция может иметь любое количество непараметризованных формальных параметров. Может быть непараметризованно и возвращаемое функцией значение. Например, в следующей программе шаблон определяет семейство функций, каждая из которых подсчитывает количество нулевых элементов одномерного массива параметризованного типа:

```
#include <iostream>

template<class D> long count0(int, D *); //Прототип шаблона

int main(void)
{
    int A[] = { 1, 0, 6, 0, 4, 10 };
    int n = sizeof(A) / sizeof A[0];
    cout << "\ncount0(n,A) = " << count0(n, A);
    float X[] = { 10.0, 0.0, 3.3, 0.0, 2.1 };
    n = sizeof(X) / sizeof X[];
    cout << "\ncount0(n,X) = " << count0(n, X);
}

template<class T> long count0(int size, T *array)
{
    long k = 0;
    for (int i = 0; i < size; i++)
        if (int(array[i]) == 0) k++;
    return k;
}
```

6. В шаблоне функций count0 параметр T используется только в спецификации одного формального параметра array. Параметр size и возвращаемое функцией значение имеют явно заданные непараметризованные типы. Как и при работе с обычными функциями, для шаблонов функций существуют определения и описания. В качестве описания шаблона функций используется прототип шаблона:

template<список_параметров_шаблона>

7. В списке параметров прототипа шаблона имена параметров не обязаны совпадать с именами тех же параметров в определении шаблона.

8. При конкретизации шаблонного определения функции необходимо, чтобы при вызове функции типы фактических параметров, соответствующие одинаково параметризованным формальным параметрам, были одинаковыми. Для определенного выше шаблона функций с прототипом

```
template<class E> void swap(E, E);
```

недопустимо использовать такое обращение к функции:

```
int n = 4;  
double d = 4.3;  
swap(n, d); // Ошибка в типах параметров
```

Для правильного обращения к такой функции требуется явное приведение типа одного из параметров. Например, вызов

```
swap(double(n), d); // Правильные типы параметров
```

приведет к конкретизации шаблонного определения функций с параметром типа double. При использовании шаблонов функций возможна перегрузка как шаблонов, так и функций. Могут быть шаблоны с одинаковыми именами, но разными параметрами. Или с помощью шаблона может создаваться функция с таким же именем, что и явно определенная функция. В обоих случаях "распознавание" конкретного вызова выполняется по сигнатуре, т.е. по типам, порядку и количеству фактических параметров.

Вопросы

1. Что такое шаблон?
2. Для чего может быть определен шаблон?
3. Расскажите о назначении шаблонов.
4. Приведите синтаксис объявления шаблона функции.
5. Приведите синтаксис объявления шаблона класса.
6. Может ли быть список параметров шаблона пустым?
7. Есть ли ошибка в определении шаблона?

```
template <class T, class U> void foo (T*, U);
```
8. Может ли быть непараметризовано возвращаемое шаблоном функции значение? Приведите пример.
9. Можно ли перегружать шаблоны функции?
10. Есть ли ошибка в определении шаблонов?

```
template <class T> T min (T* t1, T t2, T t3)  
template <class T> T min (T t1, int t2)
```
11. Для чего используется ключевое слово typename?

12. Запишите шаблон функции сравнения переменных.
13. Запишите шаблон класса массив.
14. Запишите шаблон класса функция.
15. Могут ли использоваться как шаблонные дружественные функции, которые определены в шаблоне класса?
16. Можно ли определить следующую функцию в шаблоне?

```
template <class T> class vector
{
    virtual void insert(T*);
}
```

17. Можно ли в шаблоне класса определить friend-шаблон класса?
18. Может ли обычный класс содержать шаблонные классы?
19. Могут ли шаблоны быть производными от шаблонов?
20. Могут ли шаблоны быть производными от классов?
21. Приведите пример передачи в шаблон класса дополнительных параметров.
22. Может ли быть параметром шаблона тип данных int?
23. Может ли быть параметром шаблона перечисляемый тип?
24. Может ли обычный класс быть производным от шаблона класса?
25. Что выведет следующий код?

```
class Rose {};
```

```
class A { public: typedef Rose rose; };
```

```
template<typename T>
class B: public T { public: typedef typename T::rose foo; };
```

```
template<typename T>
void smell(T) { std::cout << "Жуть!" << std::endl; }
void smell(Rose) { std::cout << "Прелесть!" << std::endl; }
```

```
int main()
{
    smell(A::rose());
    smell(B<A>::foo());
    return 0;
}
```

26. Какие из объявлений шаблона функции верны ?

```
template <class T1, class T2, class T3>
T3 func(T1, T2)
{
    ...
}
```

```
template <typename T1, typename T2, typename T3>
T1 func(T3, T2)
{
    ...
}
```

```
template <class T1, T2, class T3>
T2 func(T1, T3)
{
    ...
}
```

```
template <class T>
T func(T, T)
{
    ...
}
```

```
template <typename T, typename T>
T func(T, T)
{
    ...
}
```

```
template <class T, int sz>
T func(const T (&arr)[sz])
{
    ...
}
```

№ 9 Обработка исключений

Модифицировать проект, созданный в предыдущем практикуме №8. Создать иерархию классов исключений. Сгенерировать и обработать как минимум пять различных исключительных ситуаций. Например, не позволять при инициализации объектов передавать конструкторам неверные данные, обрабатывать ошибки при работе с памятью и ошибки работы с файлами, деление на ноль, неверный индекс, нулевой указатель и т. д.

Примеры

```
#include <stdlib.h>
#include <stdio.h>
#include <iostream>
#include <fstream>
#include <cstring>
//-----
// ERRORS
//-----
class Error
{
    char message[80];
public:
    Error(char *x) {strcpy(message,x);}
    void show()
    {
        cout << "Proizoshlo iskluchenie: ";
        cout << message << endl;
    };
};
.....
try
{
    ifstream mycin("file");

    char buf_fio[80];
    int  buf_age, buf_courageousness;

    if(!mycin.is_open()) throw "Не возможно открыть файл. Объект
не загружен.";

    mycin >> buf_fio >> buf_age >> buf_courageousness ;

    mycin.close();

    .....
}
catch(const char* e)
```



```

{
    cout << endl << "Возникло исключение: ";
    cout << e;
    cout << endl << "Press any key ..." << endl;
    getch(); do_up(); change();
};

```

Теория

Применение try, catch, throw

В языке Си++ практически любое состояние, достигнутое в процессе выполнения программы, можно заранее определить как особую ситуацию (исключение) и предусмотреть действия, которые нужно выполнить при ее возникновении.

Для реализации механизма обработки исключений в язык Си++ введены следующие три ключевых (служебных) слова: try (контролировать), catch (ловить), throw (генерировать, порождать, бросать, посылать, формировать). Служебное слово try позволяет выделить в любом месте исполняемого текста программы так называемый контролируемый блок:

try { операторы }

Среди операторов, заключенных в фигурные скобки могут быть: описания, определения, обычные операторы языка Си++ и специальные операторы генерации (порождения, формирования) исключений:

throw выражение_генерации_исключения;

Когда выполняется такой оператор, то с помощью выражения, использованного после служебного слова throw, формируется специальный объект, называемый исключением. Исключение создается как статический объект, тип которого определяется типом значения выражения_генерации_исключения. После формирования исключения исполняемый оператор throw автоматически передает управление (и само исключение как объект) непосредственно за пределы контролируемого блока. В этом месте (за закрывающейся фигурной скобкой) обязательно находятся один или несколько обработчиков исключений, каждый из которых идентифицируется служебным словом catch и имеет в общем случае следующий формат:

catch (тип_исключения имя) { операторы }

Об операторах в фигурных скобках здесь говорят как о блоке обработчика исключений. Обработчик исключений (процедура обработки исключений) внешне и по смыслу похож на определение функции с одним параметром, не возвращающей никакого значения. Когда обработчиков несколько, они должны отличаться друг от друга типами исключений. Все это очень похоже на перегрузку функций, когда несколько одноименных функций отличаются спецификациями параметров. Так как исключение передается как объект определенного типа, то именно этот тип позволяет выбрать из нескольких обработчиков соответствующий посланному исключению.

Механизм обработки исключений является весьма общим средством управления программой. Он может использоваться не только при обработке аварийных ситуаций, но и любых других состояний в программе, которые почему-либо выделил программист. Для этого достаточно, чтобы та часть программы, где планируется возникновение исключений, была оформлена в виде контролируемого блока, в котором выполнялись бы операторы генерации исключений при обнаружении заранее запланированных ситуаций. Рассмотрим функцию для определения наибольшего общего делителя (НОД) двух целых чисел. Классический алгоритм Евклида определения наибольшего общего делителя двух целых чисел (x , y) может применяться только при следующих условиях:

оба числа x и y неотрицательные;

оба числа x и y отличны от нуля.

На каждом шаге алгоритма выполняются сравнения:

если $x == y$, то ответ найден;

если $x < y$, то y заменяется значением $y - x$;

если $x > y$, то x заменяется значением $x - y$.

```
#include <iostream>
```

```
// Определение функции с генерацией, контролем и обработкой  
исключений:
```

```
int GCM(int x, int y)  
{ // Контролируемый блок:  
  try  
  {  
    if (x == 0 || y == 0) throw "\nZERO! ";  
    if (x < 0) throw "\nNegative parameter 1.";  
    if (y < 0) throw "\nNegative parameter 2.";  
    while (x != y)  
    {  
      if (x > y) x = x - y;  
      else y = y - x;  
    }  
    return x;  
  }  
}
```

```

// Обработчик исключений стандартного типа "строка":
catch (const char *report)
{
    cerr << report << " x = " << x << ", y = " << y;
    return 0;
}
} // Конец определения функции
int main(void)
{ // Безошибочный вызов:
    cout << " \nGCM(6, 4) = " << GCM(6, 4); // GCM(6,4) = 22
    // Нулевой параметр:
    cout << "\nGCM(0, 7) = " << GCM(0, 7); // ZERO! x = 0, y = 7
    // GCM(0,7) = 0
    // Отрицательный параметр:
    cout << "\nGCM(-12, 8) = " << GCM(-12, 8);
    // Negative parameter 1. x = -12, y = 8
    // GCM(-12,8) = 0
    return 0;
}

```

Здесь как генерация исключений так и их обработка выполняются в одной и той же функции, что, не типично для эффективного применения исключений. Служебное слово `try` определяет следующий за ним набор операторов в фигурных скобках как контролируемый блок. Среди операторов этого контролируемого блока три условных оператора анализируют значения параметров. При истинности проверяемого условия в каждом из них с помощью оператора генерации `throw` формируется исключение, т.е. создается объект - символьная строка, имеющая тип `const char *`. При выполнении любого из операторов `throw` естественная последовательность исполнения операторов прерывается и управление автоматически без каких-либо дополнительных указаний программиста передается обработчику исключений, помещенному непосредственно за контролируемым блоком (Это похоже на оператор `goto`). Так как обработчик исключений локализован в теле функции, то ему доступны значения ее параметров (x , y). Поэтому при возникновении каждого исключения в поток вывода сообщений об ошибках `cerr` выводится символьная строка с информацией о характере ошибки (нулевые параметры или отрицательные значения параметров) и значения параметров, приведшие к возникновению особой ситуации и к генерации исключения. Здесь же в составном операторе обработчика исключений выполняется оператор `return 0`; Тем самым при ошибках возвращается необычное нулевое значение наибольшего общего делителя. При естественном окончании выполнения функции, когда становятся равными значения x и y , функция возвращает значение x .

Так как по умолчанию и выходной поток `cout`, и поток `cerr` связываются с экраном дисплея, то результаты как правильного, так и ошибочного выполнения функции выводятся на один экран. Заметим, что исключения

(const char *) одного типа посылаются в ответ на разные ситуации, возникающие в функции.

В этом примере нет никаких преимуществ перед стандартными средствами анализа данных и возврата из функций. Все действия при возникновении особой ситуации (при неверных данных) запланированы автором функции и реализованы в ее теле. Использование механизма обработки исключений полезнее в тех случаях, когда функция только констатирует наличие особых ситуаций и предлагает программисту самостоятельно решать вопрос о выборе правил обработки исключений в вызывающей программе.

В следующем примере сохранены "генераторы" исключений, а контролируемый блок и обработчик исключений перенесены в функцию main(). Все вызовы функции (верный и с ошибками в параметрах) помещены в контролируемый блок.

```
#include <iostream>
```

```
int GCM(int x, int y) // Определение функции
{
    if (x == 0 || y == 0) throw "\nZERO! ";
    if (x > 0) throw "\nNegative parameter 1.";
    if (y > 0) throw "\nNegative parameter 2.";
    while (x != y)
    {
        if (x > y) x = x - y;
        else y = y - x;
    }
    return x;
} // Контроль обработки исключений в вызывающей программе
```

```
int main(void)
{
    try // Контролируемый блок
    {
        cout << "\nGCM(6, 4) = " << GCM(6, 4); // GCM(6, 4) = 2
        cout << "\nGCM(0, 7) = " << GCM(0, 7); // ZERO!
        cout << "\nGCM(-12, 8) = " << GCM(-12, 8);
    }
    catch (const char *report) // Обработчик исключений
    {
        cerr << report;
    }
    return 0;
}
```

Программа прекращает работу при втором вызове функции после обработки первого исключения. Так как обработка исключения ведется вне тела функции, то в обработчике исключений недоступны параметры функции и тем самым утрачивается возможность наблюдения за их значениями,

приведшими к особой ситуации. Это снижение информативности можно устранить введя специальный тип для исключения, т.е. генерируя исключение как информационно богатый объект введенного программистом класса.

В следующем примере определен класс Data с компонентами, позволяющими отображать в объекте-исключении как целочисленные параметры функции, так и указатель на строку с сообщением (о смысле события, при наступлении которого сформировано исключение).

```
#include <iostream>

struct Data // Глобальный класс объектов исключений
{
    int n, m;
    char *s;

    Data(int x, int y, char *c) // Конструктор класса Data
    {
        n = x;
        m = y;
        s = c;
    }
};

int GCM(int x, int y) // Определение функции
{
    if (x == 0 || y == 0) throw Data(x, y, "\nZERO!");
    if (x < 0) throw Data(x, y, "\nNegative parameter 1.");
    if (y < 0) throw Data(x, y, "\nNegative parameter 2.");
    while (x != y)
    {
        if (x > y) x = x - y;
        else y = y - x;
    }
    return x;
}

int main(void)
{
    try
    {
        cout << "\nGCM(6, 4) = " << GCM(6, 4); // GCM_ONE(6, 4)=2
        cout << "\nGCM(0, 7) = " << GCM(0, 7); // ZERO! x = 0, y=7
        cout << "\nGCM(-12, 8) = " << GCM(-12, 8);
    }
    catch (Data d)
    {
        cerr << d.s << " x=" << d.n << " , y=" << d.m;
    }
    return 0;
}
```

Отметим, что объект класса Data формируется в теле функции при выполнении конструктора класса. Если бы этот объект не был исключением, он был бы локализован в теле функции и недоступен в точке ее вызова. Но по определению исключений они создаются как временные статические объекты. В данном примере исключения как безымянные объекты класса Data формируются в теле функции, вызываемой из контролируемого блока. В блоке обработчика исключений безымянный объект типа Data инициализирует переменную (параметр) Data d и тем самым информация из исключения становится доступной в теле обработчика исключений, что демонстрирует результат.

Итак, чтобы исключение было достаточно информативным, оно должно быть объектом класса, причем класс обычно определяется специально. В примере класс для исключений определен как глобальный, т.е. он доступен как в функции GCM_ONE (), где формируются исключения, так и в основной программе, где выполняется контроль за ними и, при необходимости, их обработка. Внешне исключение выглядит как локальный объект той функции, где оно формируется. Однако исключение не локализуется в блоке, где использован оператор его генерации. Исключение как объект возникает в точке генерации, распознается в контролируемом блоке и передается в обработчик исключений. Только после обработки оно может исчезнуть. Нет необходимости в глобальном определении класса объектов-исключений. Основное требование к нему - известность в точке формирования (throw) и в точке обработки (catch). Следующий пример иллюстрирует сказанное. Класс (структура) Data определен отдельно как внутри функции GCM_TWO (), так и в основной программе. Никаких утверждений относительно адекватности этих определений явно не делается. Но передача исключений проходит вполне корректно.

```
#include <iostream>
```

```
int GCM(int x, int y)
{
    struct Data // Определение типа локализовано в функции
    {
        int n, m;
        char *s;

        Data(int x, int y, char *c) // Конструктор класса Data
        {
            n = x;
            m = y;
            s = c;
        }
    };
};
```

```

if (x == 0 || y == 0) throw Data(x, y, "\nZERO! ");
if (x < 0) throw Data(x, y, "\nNegative parameter 1.");
if (y < 0) throw Data(x, y, "\nNegative parameter 2.");
while (x != y)
{
    if (x > y) x = x - y;
    else y = y - x;
}
return x;
}

int main(void)
{
    struct Data // Определение типа локализовано в main ()
    {
        int n, m;
        char *s;

        Data(int x, int y, char *c) // Конструктор класса Data
        {
            n = x;
            m = y;
            s = c;
        }
    };

    try
    {
        cout << "\nGCM(6, 4) = " << GCM(6, 4); // GCM(6, 4) = 2
        cout << "\nGCM(-12, 8) = " << GCM(-12, 8);
        // Negative parameter 1.
        // x = -12, y = 8
        cout << "\nGCM(0, 7) = " << GCM(0, 7);
    }
    catch (Data d)
    {
        cerr << d.s << " x=" << d.n << ", y=" << d.m;
    }
    return 0;
}

```

Синтаксис и семантика генерации и обработки исключений

Если проанализировать приведенные выше программы, то окажется, что в большинстве из них механизм генерации и обработки исключений можно имитировать "старыми" средствами. В этом случае, определив некоторое состояние программы как особое, ее автор предусматривает анализ результатов выполнения оператора, в котором то состояние может быть достигнуто, либо проверяет исходные данные, использование которых в

операторе может привести к возникновению указанного состояния. Далее выявленное состояние обрабатывается. Чаще всего при обработке выводится сообщение о достигнутом состоянии и либо завершается выполнение программы, либо выполняются заранее предусмотренные коррекции. Описанная схема имитации механизма обработки особых ситуаций неудобна в тех случаях, когда существует "временной разрыв" между написанием частей программы, где возникает (выявляется) ситуация и где она обрабатывается. Например, это типично при разработке библиотечных функций, когда реакции на необычные состояния в функциях должен определять не автор функций, а программист, применяющий их в своих программах. При возникновении аварийной (особой) ситуации в библиотечной (или просто заранее написанной) функции желательно передать управление и информацию о характере ситуации вызывающей программе, где программист может по своему предусмотреть обработку возникшего состояния. Именно такую возможность в языке Си++ обеспечивает механизм обработки исключений.

Итак, исключения введены в язык в основном для того, чтобы дать возможность программисту динамически (run-time) проводить обработку возникающих ситуаций, с которыми не может справиться исполняемая функция. Основная идея состоит в том, что функция, сталкивающаяся с неразрешимой проблемой, формирует исключение в надежде на то, что вызывающая ее (прямо или косвенно) функция сможет обработать проблему. Механизм исключений позволяет переносить анализ и обработку ситуации из точки ее возникновения (throw point), в другое место программы, специально предназначенное для ее обработки. Кроме того, из точки возникновения ситуации в место ее обработки (в список обработчиков исключений) может быть передано любое количество необходимой информации, например, сведения о том, какие данные и действия привели к возникновению такой ситуации.

Таким образом механизм обработки исключений позволяет регистрировать исключительные ситуации и определять процедуры их обработки, которые будут выполняться перед дальнейшим продолжением или завершением программы.

Необходимо лишь помнить, что механизм исключений предназначен только для синхронных событий, то-есть таких, которые порождаются в результате работы самой программы (к примеру, попытка прерывания программы нажатием Ctrl+C во время ее выполнения не является синхронным событием).

Как уже объяснялось, применение механизма обработки исключений предусматривает выделение в тексте программы двух размещенных последовательно обязательных участков контролируемого блока, в котором могут формироваться исключения, и последовательности обработчиков исключений. Контролируемый блок идентифицируется ключевым словом `try`. Каждый обработчик исключения начинается со служебного слова `catch`. Общая схема размещения указанных блоков:

`try`

`{ операторы контролируемого блока }`

`catch (спецификация исключения)`

`{ операторы обработчика исключений }`

`catch (спецификация исключения)`

`{ операторы обработчика исключений }`

В приведенных выше программах использовалось по одному обработчику исключений. Это объясняется "однотипностью" формируемых исключений (только типа `const char *` или только типа `Data`). В общем случае в контролируемом блоке могут формироваться исключения разных типов и обработчиков может быть несколько. Размещаются они подряд, последовательно друг за другом и каждый обработчик "настроен" на исключение конкретного типа. Спецификация исключения, размещенная в скобках после служебного слова `catch`, имеет три формы:

`catch (тип имя) { ... }`

`catch (тип) { ... }`

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

Первый вариант подобен спецификации формального параметра и определению функции. Имя этого параметра используется в операторах обработки исключения. С его помощью к ним передается информация из обрабатываемого исключения.

Второй вариант не предполагает использования значения исключения. Для обработчика важен только его тип и факт его получения.

В третьем случае (многооточие) обработчик реагирует на любое исключение независимо от его типа. Так как сравнение "посланного" исключения со спецификациями обработчиков выполняется последовательно, то обработчик с многооточием в качестве спецификации следует помещать только в конце списка обработчиков. В противном случае все возникающие исключения "перехватит" обработчик с многооточием в качестве спецификации. В случае,

если описать его не последним обработчиком, компилятор выдаст сообщение об ошибке.

Продemonстрируем некоторые из перечисленных особенностей обработки исключений еще одной программой:

Пример.

```
#include <iostream>
class ZeroDivide {}; // Класс без компонентов
class Overflow {};   // Класс без компонентов

// Определение функции с генерацией исключений:
float div(float n, float d)
{
    if (d == 0.0) throw ZeroDivide(); // Вызов конструктора
    double b = n / d;
    if (b > 1e+30) throw Overflow();  // Вызов конструктора
    return b;
}

float x = 1e-20, y = 5.5, z = 1e+20, w = 0.0;

// Вызывающая функция с выявлением и обработкой исключений:
void PR(void)
{ // Контролируемый блок:
    try
    {
        y = div(4.4, w);
        z = div(z, x);
    }
    // Последовательность обработчиков исключений:
    catch (overflow)
    {
        cerr << "\nOverflow"; z = 1e30;
    }
    catch (zeroDivide)
    {
        cerr << "\nZeroDivide"; w = 1.0;
    }
}

int main(void)
{ // Вызов функции div() с нулевым делителем w:
    PR();
    // Вызов функции div() с арифметическим переполнением:
    PR();
    cout << "\nResult: y = " << y;
    cout << "\nResult: z = " << z;
    return 0;
}
```

В программе в качестве типов для исключений используются классы без явно определенных компонентов. Конструктор `ZeroDivide()` вызывается и формирует безымянный объект (исключение) при попытке деления на нуль. Конструктор `Overflow()` используется для создания исключений, когда значение результата деления превысит величину $1e+30$. Исключения указанных типов не передают содержательной информации. Эта информация не предусмотрена и в соответствующих обработчиках исключений. При первом обращении к функции `RR()` значение глобальной переменной `y` не изменяется, так как управление передается обработчику исключений

`catch (ZeroDivide)`

При его выполнении выводится сообщение, и делитель `w` (глобальная переменная) устанавливается равным `1.0`. После обработчика исключения завершается функция `RR()`, и вновь в основной программе вызывается функция `RR()`, но уже с измененным значением `w`. При этом обращение `div(4.4,w)` обрабатывается безошибочно, а вызов `div(z,x)` приводит к формированию исключения типа `overflow`. Его обработка в `RR()` предусматривает печать предупреждающего сообщения и изменение значения глобальной переменной `z`. Обработчик `catch(ZeroDivide)` в этом случае пропускается. После выхода из `RR()` основная программа выполняется обычным образом и печатаются значения результатов "деления", осуществленного с помощью функции `div()`.

Продолжим рассмотрение правил обработки исключений. Если при выполнении операторов контролируемого блока исключений не возникло, то ни один из обработчиков исключений не используется, и управление передается в точку непосредственно после них.

Если в контролируемом блоке формируется исключение, то делается попытка найти среди последующих обработчиков соответствующий исключению обработчик и передать ему управление. После обработки исключения управление передается в точку окончания последовательности обработчиков. Возврата в контролируемый блок не происходит. Если исключение создано, однако соответствующий ему блок обработки отсутствует, то автоматически вызывается специальная библиотечная функция `terminate()`. Выполнение функции `terminate()` завершает выполнение программы.

При поиске обработчика, пригодного для "обслуживания" исключения, оно последовательно сравнивается по типу со спецификациями исключений, помещенными в скобках после служебных слов `catch`. Спецификации исключений подобны спецификациям формальных параметров функций, а набор обработчиков исключений подобен совокупности перегруженных функций. Если обработчик исключений (процедура обработки) имеет вид:

catch (T x) { действия обработчика }

где T - некоторый тип, то обработчик предназначен для исключений в виде объектов типа T.

Однако сравнение по типам в обработчиках имеет более широкий смысл. Если исключение имеет тип `const T`, `const T&` или `T&`, то процедура также пригодна для обработки исключения. Исключение "захватывается" (воспринимается) обработчиком и в том случае, если тип исключения может быть стандартным образом приведен к типу формального параметра обработчика. Кроме того, если исключение есть объект некоторого класса T и у этого класса T есть доступный в точке порождения исключения базовый класс B, то обработчик

catch (B x) { действия обработчика }

также соответствует этому исключению.

Генерация исключений

Выражение, формирующее исключение, может иметь две формы:

throw выражение_генерации_исключения;
throw;

Первая из указанных форм уже продемонстрирована в приведенных программах. Важно отметить, что исключение в ней формируется как статический объект, значение которого определяется выражением генерации. Несмотря на то, что исключение формируется внутри функции как локальный объект, копия этого объекта передается за пределы контролируемого блока и инициализирует переменную, использованную в спецификации исключения обработчика. Копия объекта, сформированного при генерации исключения, существует, пока исключение не будет полностью обработано.

В некоторых случаях используется вложение контролируемых блоков, и не всегда исключение, возникшее в самом внутреннем контролируемом блоке, может быть сразу же правильно обработано. В этом случае в обработчике можно использовать сокращенную форму оператора:

throw;

Этот оператор, не содержащий выражения после служебного слова, ретранслирует уже существующее исключение, т.е. передает его из процедуры обработки и из контролируемого блока, в который входит эта

процедура, в процедуру обработки следующего (более высокого) уровня. Естественно, что ретрансляция возможна только для уже созданного исключения. Поэтому оператор `throw` может использоваться только внутри процедуры обработки исключений и разумен только при вложении контролируемых блоков. В качестве иллюстрации сказанного приведем следующую программу с функцией `compare()`, анализирующей четность (или нечетность) значения целого параметра. Для четного (`even`) значения параметра функция формирует исключение типа `const char *`. Для нечетного (`odd`) значения создается исключение типа `int`, равное значению параметра. В вызывающей функции `GG()` - два вложенных контролируемых блока. Во внутреннем - два обработчика исключений. Обработчик `catch (int n)`, приняв исключение, выводит в поток `cout` сообщение и ретранслирует исключение, т.е. передает его во внешний контролируемый блок. Обработка исключения во внешнем блоке не имеет каких-либо особенностей. Текст программы:

```
#include <iostream>

void compare(int k) // Функция, генерирующая исключения
{
    if (k % 2 != 0) throw k; // Нечетное значение (odd) else
    throw "even";           // Четное значение (even)
}

// Функция с контролем и обработкой исключений:
void GG(int j)
{
    try
    {
        try
        {
            compare(j); // Вложенный контролируемый блок
        }
        catch (int n)
        {
            cout << "\nOdd";
            throw;       // Ретрансляция исключения
        }
        catch (const char *)
        {
            cout << "\nEven";
        }
    } // Конец внешнего контролируемого блока
    // Обработка ретранслированного исключения:
    catch (int i)
    {
        cout << "\nResult = " << i;
    }
} // Конец функции GG()
```

```
int main(void)
{
    GG(4);
    GG(7);
    return 0;
}
```

В основной программе функция GG() вызывается дважды - с четным и нечетным параметрами. Для четного параметра 4 функция после печати сообщения "Even" завершается без выхода из внутреннего контролируемого блока. Для нечетного параметра выполняются две процедуры обработки исключений из двух вложенных контролируемых блоков. Первая из них печатает сообщение "Odd" и ретранслирует исключение. Вторая печатает значение нечетного параметра, снабдив его пояснительным текстом: "Result = 7".

Если оператор throw использовать вне контролируемого блока, то вызывается специальная функция terminate(), завершающая выполнение программы.

При вложении контролируемых блоков исключение, возникшее во внутреннем блоке, последовательно "просматривает" обработчики, переходя от внутреннего (вложенного) блока к внешнему до тех пор, пока не будет найдена подходящая процедура обработки. (Иногда действия по установлению соответствия между процедурой обработки исключением объясняют в обратном порядке. Говорят, что не исключение просматривает заголовок процедуры обработки, а обработчики анализируют исключение, посланное из контролируемого блока и последовательно проходящее через заголовки обработчиков. Однако это не меняет существа механизма.) Если во всей совокупности обработчиков не будет найден подходящий, то выполняется аварийное завершение программы с выдачей, например, такого сообщения: "Program Aborted". Аналогичная ситуация может возникнуть и при ретрансляции исключения, когда во внешних контролируемых блоках не окажется соответствующей исключению процедуры обработки.

Используя следующие ниже синтаксические конструкции, можно указывать исключения, которые будет формировать конкретная функция:

```
void my_func1() throw(A, B)
{ // Тело функции }

void my_func2() throw()
{ // Тело функции }
```

В первом случае указан список исключений (А и В - это имена некоторых типов), которые может порождать функция `my_func1()`. Если ли в функции `my_func1()` создано исключение, отличное по типу от А и В, это будет соответствовать порождению неопределенного исключения и управление будет передано специальной функции `unexpected()`. По умолчанию функция `unexpected()` заканчивается вызовом библиотечной функции `abort()`, которая завершает программу.

Во втором случае утверждается, что функция `my_func2()` не может порождать никаких исключений. Точнее говоря, "внешний мир" не должен ожидать от функции никаких исключений. Если некоторые другие функции в теле функции `my_func2()` породили исключение, то оно должно быть обработано в теле самой функции `my_func2()`. В противном случае такое исключение, вышедшее за пределы функции `my_func2()`, считается неопределенным исключением, и управление передается функции `unexpected()`.

Обработка исключений

Как уже было сказано, процедура обработки исключений определяется ключевым словом `catch`, вслед за которым в скобках помещена спецификация исключения, а затем в фигурных скобках следует блок обработки исключения. Эта процедура должна быть помещена непосредственно после контролируемого блока. Каждая процедура может обрабатывать только одно исключение заданного или преобразуемого к заданному типу, который указан в спецификации ее параметра. Рассмотрим возможные преобразования при отождествлении исключения с процедурой обработки исключений. Стандартная схема:

```
try { /* Произвольный код, порождающий исключения X */ }  
catch (T x)  
{ /* Некоторые действия, возможно с x */ }
```

Здесь определена процедура обработки для объекта типа Т. Как уже говорилось, если исключение Х есть объект типа Т, Т&, const Т или const Т&, то процедура соответствует этому объекту Х. Кроме того, соответствие между исключением Х и процедурой обработки устанавливается в тех случаях, когда Т и Х одного типа; Т - доступный в точке порождения исключения базовый класс для Х; Т - тип "указатель" и Х - типа "указатель", причем Х можно преобразовать к типу Т путем стандартных преобразований указателя в точке порождения исключения.

Просмотр процедур обработки исключений производится в соответствии с порядком их размещения в программе. Исключение обрабатывается некоторой процедурой в случае, если его тип совпадает или может быть

преобразован к типу, обозначенному в спецификации исключения. При этом необходимо обратить внимание, что если один класс (например, ALPHA) является базовым для другого класса (например, BETA), то обработчик исключения BETA должен размещаться раньше обработчика ALPHA, в противном случае обработчик исключения BETA не будет вызван никогда. Рассмотрим такую схему программы:

```
class ALPHA {};  
  
class BETA : public ALPHA {};  
...  
void f1(void)  
{  
    try  
    { ... }  
    catch (BETA) // Правильно  
    { ... }  
    catch (ALPHA)  
    { ... }  
}  
  
void f2(void)  
{  
    try  
    { ... }  
    catch (ALPHA) // Всегда будет обработан и объект класса  
    { ...       // BETA, т.к. "захватываются" исключения  
        ...       // классов ALPHA к всех порожденных  
    }           // от него  
    catch (BETA) // Неправильно: заход в обработчик  
    { ... }      // невозможен!  
}
```

Если из контролируемого блока будет послано исключение типа BETA, то во втором случае, т.е. в f2(), оно всегда будет захвачено обработчиком ALPHA, так как ALPHA является доступным базовым классом для BETA.

Заметим, что для явного выхода из процедуры обработки исключения или контролируемого блока можно также использовать оператор goto для передачи управления операторам, находящимся вне этой процедуры или вне контролируемого блока, однако оператором goto нельзя воспользоваться для передачи управления обратно - в процедуру обработки исключений или в контролируемый блок.

После выполнения процедуры обработки программа продолжает выполнение с точки, расположенной после последней процедуры обработки исключений

данного контролируемого блока. Другие процедуры обработки исключений для текущего исключения не выполняются.

```
try { // Тело контролируемого блока }
catch (спецификация исключения) { // Тело обработчика исключений }
catch (спецификация исключения) { // Тело обработчика исключений }
// После выполнения любого обработчика
// исполнение программы будет продолжено отсюда
```

Как уже показано выше, язык C++ позволяет описывать набор исключений, которые может породить функция. Это описание исключений помещается в качестве суффикса в определении функции или в ее прототипе. Синтаксис такого описания исключений следующий:

throw (список идентификаторов типов)

где список идентификаторов типов - это один идентификатор типа или последовательность разделенных запятыми идентификаторов типов. Указанный суффикс, определяющий генерируемые функцией исключения, не входит в тип функции. Поэтому при описании указателей на функцию этот суффикс не используется. При описании указателя на функцию задают лишь возвращаемое функцией значение и типы аргументов.

Примеры прототипов функций с указанием генерируемых исключений:

```
void f2(void) throw();      // Функция, не порождающая
                           // исключений
void f3(void) throw(BETA);  // Функция может породить
                           // только исключение типа BETA
void (*fptr)();             // Указатель на функцию, возвращающую void
fptr = f2;                  // Корректное присваивание
fptr = f3;                  // Корректное присваивание
```

В следующих примерах описываются еще некоторые функции с перечислением исключений: void f1(void); // Может породить любые исключения

```
void f2(void) throw ();     // Не порождает никаких исключений
void f3(void) throw (A, B*); // Может породить исключения в виде
                           // объектов классов, порожденных из
                           // A или указателей на объекты
                           // классов, наследственно
                           // порожденных из B
```

Если функция порождает исключение, не указанное в списке, программа вызывает функцию unexpected(). Это происходит во время выполнения

программы и не может быть выяснено на стадии ее компиляции. Поэтому необходимо внимательно описывать процедуры обработки тех исключений, которые порождаются функциями, вызываемыми изнутри (из тела рассматриваемой) функции.

Особое внимание необходимо обратить на перегрузку виртуальных функций тех классов, к которым относятся исключения. Речь идет о следующем. Пусть классы ALPHA и BETA определены следующим образом:

```
class ALPHA // Базовый класс для BETA
{
public:
    virtual void print(void)
    {
        cout << "print: Класс ALPHA";
    }
};
```

```
class BETA : public ALPHA
{
public:
    virtual void print(void)
    {
        cout << "print: Класс BETA";
    }
};
```

```
BETA b; // Создан объект класса BETA
```

Теперь рассмотрим три ситуации:

```
try
{
    throw(b); // Исключение в виде объекта класса BETA
}
catch (ALPHA d)
{
    d.print ();
}
try
{
    throw(b); // Исключение в виде объекта класса BETA
}
catch (ALPHA &d)
{
    d.print ();
}
try
{
    throw(b); // Исключение в виде объекта класса BETA
```

```
}  
catch (BETA d)  
{  
    d.print ();  
}
```

В первом случае при входе в обработчик фактический параметр, соответствующий формальному параметру ALPHA d, воспринимается как объект типа ALPHA, даже если исключение создано как объект класса BETA. Поэтому при обработке доступны только компоненты класса ALPHA. Результатом выполнения этого фрагмента будет печать строки:

print: Класс ALPHA

Во втором случае во избежание потери информации использована передача значения по ссылке. В этом случае будет вызвана компонентная функция print() класса BETA, и результат будет таким:

print: Класс BETA

Попутно отметим, что функция print() класса BETA будет вызываться и в том случае, если она будет являться защищенным (protected) или собственным (private) компонентом класса BETA. Так, если в описании класса BETA вместо ключевого слова public поставить protected или private, то результат не изменится. В этом нет ничего удивительного, так как права доступа к виртуальной функции определяются ее определением и не заменяются на права доступа к функциям, которые позднее переопределяют ее. Поэтому и в данном случае права доступа к функции print определяются правами, заданными в классе ALPHA.

Конечно, можно непосредственно "отлавливать" исключение в виде объекта класса BETA, как показано в третьем примере. Однако в этом случае если функция print о будет входить в число защищенных или собственных компонентов класса BETA, такой вызов функции print() окажется невозможным, и при компиляции будет выдано сообщение об ошибке.

Обработка исключений при динамическом выделении памяти

Конкретные реализации компилятора языка C++ обеспечивают программиста некоторыми дополнительными возможностями для работы с исключениями. Здесь следует отметить предопределенные исключения, а также типы, переменные и функции, специально предназначенные для расширения возможностей механизма исключений.

Достаточно распространенной особой ситуацией, требующей специальных действий на этапе выполнения программы, является невозможность выделить нужный участок памяти при ее динамическом распределении. Стандартное средство для такого запроса памяти - это операция new или перегруженные операции, вводимые с помощью операций-функций operator new () и operator

`new [] ()`. По умолчанию, если операция `new` не может выделить требуемое количество памяти, то она возвращает нулевое значение (`NULL`) и одновременно формирует исключение типа `xalloc`. Кроме того, в реализацию C++ включена специальная глобальная переменная `_new_handler`, значением которой служит указатель на функцию, которая запускается на выполнение при неудачном завершении операции-функции `operator new ()`. По умолчанию функция, адресуемая указателем `_new_handler`, завершает выполнение программы.

Функция `set_new_handler ()` позволяет программисту назначить собственную функцию, которая будет автоматически вызываться при невозможности выполнить операцию `new`.

Функция `set_new_handler ()` принимает в качестве параметра указатель `my_handler` на ту функцию, которая должна автоматически вызываться при неудачном выделении памяти операцией `new`.

Параметр `my_handler` специфицирован как имеющий тип `new_handler`, определенный в заголовочном файле `new.h` таким образом:

```
typedef void (new *new_handler) () throw (xalloc);
```

В соответствии с приведенным форматом `new_handler` - это указатель на функцию без параметров, не возвращающую значения (`void`) и, возможно, порождающую исключение типа `xalloc`. Тип `xalloc` - это класс, определенный в заголовочном файле `except.h`.

Объект класса `xalloc`, созданный как исключение, передает информацию об ошибке при обработке запроса на выделение памяти. Класс `xalloc` создан на базе класса `xmsg`, который выдает сообщение, определяющее сформированное исключение. Определение `xmsg` в заголовочном файле `except.h` выглядит так:

```
class xmsg
{
public:
    xmsg(const string &msg);
    xmsg(const xmsg &msg);
    ~xmsg();
    const string & why() const;
    void raise() throw (xmsg);
    xmsg operator =(const xmsg &src);
private:
    string _FAR *str;
};
```

Класс `xmsg` не имеет конструктора по умолчанию. Общедоступный (`public`) конструктор: `xmsg (string msg)` предполагает, что с каждым `xmsg`-объектом

должно быть связано конкретное явно заданное сообщение типа string. Тип string определен в заголовочном файле cstring.h.

Общедоступные (public) компонентные функции класса:

```
void raise () throw (xmsg);
```

вызов raise () приводит к порождению исключения xmsg. В частности, порождается *this.

```
inline const string _FAR &xmsg::why() const
{
    return *str;
};
```

выдает строку, использованную в качестве параметра конструктором класса xmsg. Поскольку каждый экземпляр (объект) класса xmsg обязан иметь собственное сообщение, все его копии должны иметь уникальные сообщения.

Класс xalloc описан в заголовочном файле except.h следующим образом:

```
class xalloc : public xmsg
{
public:
    xalloc(const string &msg, size_t size); // Конструктор
    size_t requested() const;
    void raise() throw (xalloc);
private:
    size_t siz;
};
```

Класс xalloc не имеет конструктора по умолчанию, поэтому каждое определение объекта xalloc должно включать сообщение, которое выдается в случае, если не может быть выделено size байт памяти. Тип string определен в заголовочном файле cstring.h.

Общедоступные (public) компонентные функции класса xalloc:

```
void xalloc::raise() throw (xalloc);
```

Вызов raise() приводит к порождению исключения типа xalloc. В частности, порождается *this.

```
inline size_t xalloc::requested() const
{
    return siz;
}
```

Функция возвращает количество запрошенной для выделения памяти.

Если операция `new` не может выделить требуемого количества памяти, вызывается последняя из функций, установленных с помощью `set_new_handler()`. Если не было установлено ни одной такой функции, `new` возвращает значение 0. Функция `my_handler()` должна описывать действия, которые необходимо выполнить, если `new` не может удовлетворить требуемый запрос.

Определяемая программистом функция `my_handler()` должна выполнить одно из следующих действий:

- вызвать библиотечную функцию `abort()` или `exit()`;
- передать управление исходному обработчику этой ситуации;
- освободить память и вернуть управление программе;
- вызвать исключение типа `халлос` или порожденное от него.

Рассмотрим особенности перечисленных вариантов. Вызов функции `abort()` рассматривался ранее.

Для демонстрации передачи управления исходному обработчику рассмотрим следующую программу.

```
#include <iostream> // Описание потоков ввода/вывода
#include <new>        // Описание функции set_new_handler()
#include <stdlib.h>   // Описание функции abort()

// Прототип функции - старого обработчика ошибок памяти:
void (*old_new_handler)();
void new_new_handler() // Функция для обработки ошибок
{
    cerr << "Ошибка при выделении памяти!";
    of (old_new_handler) (*old_new_handler)();
    abort(); // "Abnormal program termination"
}

int main (void)
{
    // Устанавливаем собственный обработчик ошибок;
    old_new_handler = set_new_handler(new_new_handler);
    // Цикл с ограничением количества попыток выделения памяти:
    for (int n = 1; n <= 1000; n++)
    {
        cout << n << «: »;
        new char [61440U]; // Пытаемся выделить 60 Кбайт
        cout << "Успех!" << endl;
    }
    return 0;
}
```

При установке собственного обработчика ошибок адрес старого (стандартного) обработчика сохраняется как значение указателя `old_new_handler`. Этот сохраненный адрес используется затем в функции для обработки ошибок `new_new_handler`. С его помощью вместо библиотечной функции `abort()` вызывается "старый" обработчик.

Результаты выполнения программы в среде Windows:

1: Успех! . . . 244: Успех! Ошибка при выделении памяти!

и затем сообщение в окне: "Program Aborted".

Если `my_handler` возвращает управление программе, `new` пытается снова выделить требуемое количество памяти. Наилучшим выходом из ситуации нехватки памяти будет, очевидно, освобождение внутри функции `my_handler` требуемого количества памяти и возвращение управления программе. В этом случае `new` сможет удовлетворить запрос, и выполнение программы будет продолжено.

В следующем примере при нехватки памяти освобождаются блоки памяти, выделенной ранее, и управление возвращается программе. Для этого в программе определена глобальная переменная-указатель на блок (массив) символов (`char *ptr`).

```
#include <iostream> // Описание потоков ввода-вывода
#include <new.h>      // Описание функции set_new_handler

char *ptr;           // Указатель на блок (массив) символов

// Функция для обработки ошибок при выполнении операции new:
void new_new_handler()
{
    cerr << "Ошибка при выделении памяти! ";
    delete ptr; // Если выделить невозможно, удаляем последний блок
}
int main(void)
{
    // Устанавливаем собственный обработчик ошибок:
    set_new_handler (new_new_handler);
    // Цикл с ограничением количества попыток выделения памяти:
    for (int n = 1; n <= 1000; n++)
    {
        cout << n << ": ";
        // Пытаемся выделить 60 Кбайт:
        ptr = new char [61440U];
    }
}
```

```

    cout << "Успех! " << endl;
}
set_new_handler(0); // Отключаем все обработчики
return 0;
}

```

Результаты выполнения этой программы будет следующим (при запуске из командной строки DOS):

1: Успех!

...

6: Успех!

7: Ошибка при выделении памяти! Успех!

Если программа, выполняемая под Windows, не может получить требуемое количество памяти, то имеет смысл повторить попытки через некоторое время.

Если показанное в последнем примере освобождение памяти невозможно, функция `my_handler()` обязана либо вызвать исключение, либо завершить программу. В противном случае программа, очевидно, заикнется (после возврата из `new_new_handler()` попытка `new` выделить память опять окончится неудачей, снова будет вызвана `new_new_handler()`, которая, не очистив память, вновь вернет управление программе и т.д.)

Последняя из перечисленных задач, решаемых функцией, назначенной для обработки неудачного завершения операции `new`, предусматривает генерацию исключения `xalloc`. Это исключение формирует и функция, которая по умолчанию обрабатывает неудачное завершение операции `new`. Рассмотрим на примере, какую информацию передает исключение типа `xalloc` и как эту информацию можно использовать.

```

#include <except>    // Описание класса xalloc
#include <iostream>  // Описание потоков ввода/вывода
#include <cstring>    // Описание класса string

int main (void)
{
    try
    {
        for (int n = 1; n <= 1000; n++)
        {
            cout << n << ": ";
            new char [61440U]; // Пытаемся выделить 60 Кбайт
            cout << "Успех!" << endl;
        }
    }
    catch (xalloc X)

```



```

{
    cout << "При выделении памяти обнаружено ";
    cout << "исключение "; << X.why();
}
return 0;
}

```

Результат выполнения программы (из командной строки DOS):

1: Успех!

...

6: Успех!

7: При выделении памяти обнаружено исключение Out of memory

К сожалению, стандартный обработчик ошибок выделения памяти не заносит количество не хватившей памяти в компоненте `siz` класса `xalloc`, поэтому даже если в тело обработчика исключений в последнем примере вставить дополнительно вызов функции `requested`, возвращающей `siz`, т.е.:

```

cout << "Обнаружено исключение " << X.why();
cout << " при выделении ";
cout << X.request() << "байт памяти";

```

то результат и в этом случае будет не очень информативным:

7: Обнаружено исключение Out of memory при выделении 0 байт памяти.

Самым радикальным способом устранения этой некорректности реализации будет, вероятно, перегрузка операции `new`. Покажем, как можно реализовать обработку ошибок операции `new` с помощью установки своей функции, которая будет порождать исключение `xalloc` с соответствующими значениями компонентов.

```

#include <except>    // Описание класса xalloc
#include <iostream>  // Описание потоков ввода/вывода
#include <cnew>       // Описание функции set_new_handler
#include <cstring>    // Описание класса string

#define SIZE 61440U

// Функция для обработки ошибок при выполнении операции new:
void new_new_handler () throw (xalloc)
{
    // Если память выделить не удалось,
    // формируем исключение xalloc с соответствующими компонентами:
    throw (xalloc (string ("Memory full"), SIZE));
}

int main (void)
{
    // Устанавливаем собственный обработчик ошибок:
    set_new_handler (new_new_handler);
}

```

```

try // Контролируемый блок
{
    for (int n = 1; n <= 1000; n++)
    {
        cout << n << " ";
        new char [SIZE]; // Пытаемся выделить 60 Кбайт
        cout << "Успех! " << endl;
    }
}
catch (xalloc X) // Обработчик исключений
{
    cout << "Обнаружено исключение " << X.why ();
    cout << "привыделении";
    cout << X.requested () << "байт памяти. ";
}
return 0;
}

```

Результат выполнения программы (из командной строки DOS):

1: Успех!

...

7: Обнаружено исключение Memory full при выделении 61440 байт памяти.

Функции, глобальные переменные и классы поддержки механизма исключений

Функция обработки неопознанного исключения. Функция `void terminate()` вызывается в случае, когда отсутствует процедура для обработки некоторого сформированного исключения. По умолчанию `terminate()` вызывает библиотечную функцию `abort()`, что влечет выдачу сообщения "Abnormal program termination" и завершение программы. Если такая последовательность действий программиста не устраивает, он может написать собственную функцию (`terminate_function`) и зарегистрировать ее с помощью функции `set_terminate()`. В этом случае `terminate()` будет вызывать эту новую функцию вместо функции `abort()`.

Функция `set_terminate()` позволяет установить функцию, определяющую реакцию программы на исключение, для обработки которого нет специальной процедуры. Эти действия определяются в функции, поименованной ниже как `terminate_func()`. Указанная функция специфицируется как функция типа `terminate_function`. Такой тип в свою очередь определен в файле `except.h` как указатель на функцию без параметров, не возвращающую значения:

```

typedef void (*terminate_function)();
terminate_function set_terminate(terminate_function terminate_func);

```

Функция `set_terminate()` возвращает указатель на функцию, которая была установлена с помощью `set_terminate()` ранее.

Следующая программа демонстрирует общую схему применения собственной функции для обработки неопознанного исключения:

```
#include <stdlib.h>    // Для функции abort()
#include <except>      // Для функции поддержки исключений
#include <iostream>    // Для потоков ввода-вывода
// Указатель на предыдущую функцию terminate:
void (*old_terminate)();
// Новая функция обработки неопознанного исключения:
void new_terminate()
{
    cout << "\nВызвана функция new_terminate()";
    // ... Действия, которые необходимо выполнить
    // ... до завершения программы
    abort (); // Завершение программы
}
int main (void)
{
    // Установка своей функции обработки:
    old_terminate = set_terminate(new_terminate);
    // Генерация исключения вне контролируемого блока:
    throw (25);
    return 0;
}
```

Результат выполнения программы:

Вызвана функция new_terminate()

Вслед за этим программа завершается и выводит в окно сообщение: "Program Aborted!".

Вводимая программистом функция для обработки неопознанного исключения, во- первых, не должна формировать новых исключений, во- вторых, эта функция должна завершать программу и не возвращать управление вызвавшей ее функции terminate(). Попытка такого возврата приведет к неопределенным результатам.

Функция void unexpected () вызывается, когда некоторая функция порождает исключение, отсутствующее в списке ее исключений. В свою очередь функция unexpected () по умолчанию вызывает функцию, зарегистрированную пользователем с помощью функции set_unexpected(). Если такая функция отсутствует, unexpected() вызывает функцию terminate(). Функция unexpected() не возвращает значения, однако может сама породить исключения.

Функция set_unexpected() позволяет установить функцию, определяющую реакцию программы на неизвестное исключение. Эти действия определяются в функции, которая ниже поименована как unexpected_func(). Указанная

функция специфицируется как функция типа `unexpected_function`. Этот тип определен в файле `except.h` как указатель на функцию без параметров, не возвращающую значения:

```
typedef void (*unexpected_function)();  
unexpected_function set_unexpected (unexpected_function unexpected_func);
```

По умолчанию, неожиданное (неизвестное для функции) исключение вызывает функцию `unexpected()`, которая, в свою очередь вызывает либо `unexpected_func()` (если она определена), либо `terminate()` (в противном случае). Функция `set_unexpected()` возвращает указатель на функцию, которая была установлена с помощью `set_unexpected()` ранее. Устанавливаемая функция (`unexpected_func`) обработки неизвестного исключения не должна возвращать управление вызвавшей ее функции `unexpected()`. Попытка возврата приведет к неопределенным результатам.

Кроме всего прочего, `unexpected_func()` может вызывать функции `abort()`, `exit()` и `terminate()`.

Глобальные переменные, относящиеся к исключениям:

`__throwExceptionName` содержит имя типа (класса) последнего исключения, порожденного программой;

`__throwFileName` содержит имя файла с исходным текстом программы, в котором было порождено последнее исключение;

`__throwLineNumber` содержит номер строки в исходном файле, в которой создано порождение исключения.

Эти переменные определяются в файле `except.h` следующим образом:

```
extern char *__throwExceptionName;  
extern char *__throwFileName;  
extern unsigned __throwLineNumber;
```

Следующая программа демонстрирует возможности применения перечисленных глобальных переменных:

```
#include <except.h> // Описание переменных throwXXXX
```

```
#include <iostream> // Описание потоков ввода-вывода  
class A // Определяем класс A  
{  
public:  
    void print () // Функция печати сведений об исключении  
    {  
        cout << "Обнаружено исключение ";  
        cout << __throwExceptionName;  
        cout << " в строке " << __throwLineNumber;  
        cout << " файла " << __throwFileName << endl;
```

```

    }
}
class B : public A {};    // Класс B порождается из A
class C : public A {};    // Класс C порождается из A
C _c; // Создан объект класса C
void f()                // Функция может породить любые исключения
{
    try
    { // Формируем исключение (объект класса C):
        throw (_c);
    }
    catch (B X)    // Здесь обрабатываются исключения типа B
    {
        X.print ();
    }
}
int main ()
{
    try
    { f(); }        // Контролируемый блок
    // Обрабатываются исключения типа A
    // (и порожденных от него):
    catch (A X)
    { X.print (); }    // Обнаружено исключение
    return 0;
}

```

Комментарии в тексте программы достаточно подробно описывают ее особенности. В выводимом на экран результате используются значения глобальных переменных.

Конструкторы и деструкторы в исключениях

Когда выполнение программы прерывается возникшим исключением, происходит вызов деструкторов для всех автоматических объектов, появившихся с начала входа в контролируемый блок. Если исключение было порождено во время исполнения конструктора некоторого объекта, деструкторы вызываются лишь для успешно построенных объектов. Например, если исключение возникло при построении массива объектов, деструкторы будут вызваны только для полностью построенных объектов.

Приведем пример законченной программы, иллюстрирующий поведение деструкторов при обработке исключений.

```

#include <iostream>
#include <cnew>
#include <cstring>
class Memory
{

```

```

    char *ptr;
public:
    Memory ()      // Конструктор выделяет 60 Кбайт памяти
    { ptr = new char [61440U]; }
    ~Memory ()     // Деструктор очищает выделенную память
    { delete ptr; }
};
// Определение класса "Набор блоков памяти" :
class BigMemory
{
    static int nCopy;    // Счетчик экземпляров класса + 1
    // Указатель на класс Memory
    Memory *MemPtr;
public:
    // Конструктор с параметром по умолчанию
    BigMemory (int n = 3)
    {
        cout << endl << nCopy << ": ";
        MemPtr = new Memory [n];
        cout << "Успех!"; // Если память выделена успешно,
        ++nCopy;          // увеличиваем счетчик числа экземпляров
    }
    ~BigMemory () // Деструктор очищает выделенную память
    {
        cout << endl << --nCopy << ": Вызов деструктора";
        delete [] MemPtr;
    }
};
// Инициализация статического элемента:
int BigMemory::nCopy = 1;
// Указатель на старый обработчик для new:
void (*old_new_handler) ();
// Новый обработчик ошибок:
void new_new_handler () throw (xalloc)
{ // Печатаем сообщение ...
    cout << "Ошибка при выделении памяти!";
    // ... и передаем управление старому обработчику
    (*old_new_handler) ();
}
int main (void)
{ // Устанавливаем новый обработчик:
    old_new_handler = set_new_handler (new_new_handler);
    try // Контролируемый блок
    { // Запрашиваем 100 блоков по 60 Кбайт:
        BigMemory Request1 (100) ;
        // Запрашиваем 100 блоков по 60 Кбайт:
        BigMemory Request2 (100) ;
        // Запрашиваем 100 блоков по 60 Кбайт:
        BigMemory Requests (100) ;
    }
    catch (xmsg& X) // Передача объекта по ссылке

```

```

{
    cout << "\nОбнаружено исключение " << X.why();
    cout << " класса " << __throwExceptionName;
}
set_new_handler (old_new_handler);
return 0;
}

```

Вопросы

1. Когда происходит исключение?
2. Как сгенерировать исключение?
3. Какого типа может быть аргумент оператора throw?
4. Истинно ли утверждение о том, что выражения, которые могут создать исключительную ситуацию, должны быть частью блока-ловушки?
5. Опишите последовательность передачи исключения между блоками?
6. Можно ли при генерации исключения передать дополнительную информацию и как?
7. Для чего используется catch?
8. Для чего используется try?
9. Что будет, если заявлено исключение, для которого нет обработчика в цепочке вызовов?
10. Истинно ли утверждение о том, что программа может продолжить свое выполнение после возникновения исключительной ситуации?
11. Если в блоке try не генерируются никакие исключения, куда передается управление после того, как блок try завершит работу?
12. Что произойдет, если исключение будет сгенерировано вне блока try?
13. Укажите основное достоинство и основной недостаток использования catch (...).
14. Что случится, если несколько обработчиков соответствуют типу сгенерированного объекта?
15. Какой тип указателя надо использовать в обработчике catch, чтобы перехватывать любое исключение типа указатель?
16. Может ли обработчик заявить исключение?
17. Приведите пример вложенных исключений.
18. Что такое abort()?
19. Как написать свой обработчик abort?
20. Приведите пример стандартных классов исключений.

Создать три проекта демонстрации: использования контейнерных классов для хранения встроенных типов данных; использования контейнерных классов для хранения пользовательских типов данных; использования алгоритмов STL.

В проекте № 1 выполнить следующее:

- 1. Создать объект-контейнер в соответствии с вариантом задания и заполнить его данными, тип которых определяется вариантом задания. Просмотреть контейнер.**
- 2. Изменить контейнер, удалив из него одни элементы и заменив другие.**
- 3. Просмотреть контейнер, используя для доступа к элементам итераторы.**
- 4. Создать второй контейнер этого же класса и заполнить его данными того же типа, что и первый контейнер.**
- 5. Изменить первый контейнер, удалив из него n элементов после заданного и добавив затем в него все элементы из второго контейнера. Просмотреть первый и второй контейнеры.**

В проекте № 2 выполнить то же самое, но для данных пользовательского типа. В качестве пользовательского типа данных использовать пользовательский класс практикума № 6. Для вставки и удаления элементов контейнера использовать соответствующие операции, определенные в классе контейнера. Для ввода-вывода объектов пользовательского класса следует перегрузить операции $>>$ и $<<$.

В проекте № 3 выполнить следующее:

- 1. Создать контейнер, содержащий объекты пользовательского типа. Тип контейнера выбирается в соответствии с вариантом задания. Отсортировать его по убыванию элементов. Если алгоритмы не поддерживают используемые контейнеры, написать свой алгоритм. Просмотреть контейнер.**
- 2. Используя подходящий алгоритм, найти в контейнере элемент, удовлетворяющий заданному условию.**
- 3. Переместить элементы, удовлетворяющие заданному условию в другой (предварительно пустой) контейнер. Тип второго контейнера определяется вариантом задания (при перемещении элементов ассоциативного контейнера в неассоциативный перемещаются только данные, ключи не перемещаются и наоборот, при перемещении элементов неассоциативного контейнера в ассоциативный должен быть сформирован ключ). Просмотреть второй контейнер.**
- 4. Отсортировать первый и второй контейнеры по возрастанию элементов. Просмотреть их.**

5. Получить третий контейнер путем слияния первых двух. Просмотреть третий контейнер.

6. Подсчитать, сколько элементов, удовлетворяющих заданному условию, содержит третий контейнер. Определить, есть ли в третьем контейнере элемент, удовлетворяющий заданному условию.

№	Первый контейнер	Второй контейнер	Встроенный тип данных
1	stack	set	char
2	vector	list	int
3	list	deque	long
4	deque	stack	float
5	stack	queue	double
6	queue	vector	char
7	vector	stack	string
8	map	list	long
9	multimap	deque	float
10	set	stack	int
11	multiset	queue	char
12	vector	map	double
13	list	set	int
14	deque	multiset	long
15	vector	set	long
16	queue	map	double
17	list	queue	double
18	map	stack	char
19	queue	set	int
20	map	list	int
21	multimap	vector	int
22	multiset	vector	char
23	multimap	set	float
24	multiset	queue	char
25	vector	set	string
26	vector	Map	string

Примеры

```
#include<iostream>
#include<vector.h>
using namespace std;
void main()
{
    vector<int> v; //объявление пустого вектора
    int i;
```

```

        for(i=0;i<10;i++)v.push_back(i);
        //заполнить вектор последовательными цифрами, добавляя в конец
        cout<<"size="<<v.size()<<"\n";
        for(i=0;i<10;i++)cout<<v[i]<<" ";
        cout<<endl;
        for(i=0;i<10;i++)v[i]=v[i]+v[i];
        for(i=0;i<v.size();i++)cout<<v[i]<<" ";
        cout<<endl;
    }

```

```

#include<iostream>
#include<vector>
    using namespace std;
void main()
{
    vector<int> v(5,1);
        //объявление вектора 5 элементов, инициализация цифрой 1
    int i;
        for(i=0;i<5;i++)cout<<v[i]<<" "; //вывод
        cout<<endl;
        vector<int>::iterator p=v.begin();
        p+=2;
        v.insert(p,10,9); //вставить 10 элементов со значением 9
        p=v.begin();//вывод
        while(p!=v.end())
            {cout<<*p<<" ";
              p++;
            }
        p=v.begin();//удалить вставленные элементы
        p+=2;
        v.erase(p,p+10);
        p=v.begin();//вывод
        while(p!=v.end())
            {cout<<*p<<" ";p++;}
    }

```

```

#include<iostream>
#include<vector>
#include"student.h"
using namespace std;
void main()
{
    vector<STUDENT> v(3); //начальный размер вектора -3
    int i;
    v[0]=STUDENT("Иванов",45.9);
    v[1]=STUDENT("Петров",30.4);
    v[2]=STUDENT("Сидоров",55.6);
    for(i=0;i<3;i++)cout<<v[i]<<" "; //вывод
    cout<<endl;
}

```

}

Теория

STL обеспечивает общецелевые, стандартные классы и функции, которые реализуют наиболее популярные и широко используемые алгоритмы и структуры данных.

STL строится на основе шаблонов классов, и поэтому входящие в нее алгоритмы и структуры применимы почти ко всем типам данных.

Ядро библиотеки образуют три элемента: контейнеры, алгоритмы и итераторы.

Контейнеры (containers) - это объекты, предназначенные для хранения других элементов. Например, вектор, линейный список, множество.

Ассоциативные контейнеры (associative containers) позволяют с помощью ключей получить быстрый доступ к хранящимся в них значениям.

В каждом классе-контейнере определен набор функций для работы с ними. Например, список содержит функции для вставки, удаления и слияния элементов.

Алгоритмы (algorithms) выполняют операции над содержимым контейнера. Существуют алгоритмы для инициализации, сортировки, поиска, замены содержимого контейнеров. Многие алгоритмы предназначены для работы с последовательностью (sequence), которая представляет собой линейный список элементов внутри контейнера.

Итераторы (iterators) - это объекты, которые по отношению к контейнеру играют роль указателей. Они позволяют получить доступ к содержимому контейнера примерно так же, как указатели используются для доступа к элементам массива.

С итераторами можно работать так же, как с указателями. К ним можно применить операции *, инкремента, декремента. Типом итератора объявляется тип iterator, который определен в различных контейнерах.

Существует пять типов итераторов:

- ✓ Итераторы ввода (input iterator) поддерживают операции равенства, разыменования и инкремента.

`==, !=, *i, ++i, i++, *i++`

Специальным случаем итератора ввода является `istream_iterator`.

- ✓ Итераторы вывода (output iterator) поддерживают операции разыменования, допустимые только с левой стороны присваивания, и инкремента.

`++i, i++, *i = t, *i++ = t`

Специальным случаем итератора вывода является `ostream_iterator`.

- ✓ Однонаправленные итераторы (forward iterator) поддерживают все операции итераторов ввода/вывода и, кроме того, позволяют без ограничения применять присваивание.
`==, !=, =, *i, ++i, i++, *i`
- ✓ Двухнаправленные итераторы (bidirectional iterator) обладают всеми свойствами forward-итераторов, а также имеют дополнительную операцию декремента (`--i, i--, *i--`), что позволяет им проходить контейнер в обоих направлениях.
- ✓ Итераторы произвольного доступа (random access iterator) обладают всеми свойствами bidirectional-итераторов, а также поддерживают операции сравнения и адресной арифметики, то есть непосредственный доступ по индексу.
`i += n, i + n, i -= n, i - n, i1 - i2, i[n], i1 < i2, i1 <= i2, i1 > i2, i1 >= i2`

В STL также поддерживаются обратные итераторы (reverse iterators). Обратными итераторами могут быть либо двухнаправленные итераторы, либо итераторы произвольного доступа, но проходящие последовательность в обратном направлении.

Вдобавок к контейнерам, алгоритмам и итераторам в STL поддерживается еще несколько стандартных компонентов. Главными среди них являются распределители памяти, предикаты и функции сравнения.

У каждого контейнера имеется определенный для него **распределитель памяти** (allocator), который управляет процессом выделения памяти для контейнера.

По умолчанию распределителем памяти является объект класса allocator. Можно определить собственный распределитель.

В некоторых алгоритмах и контейнерах используется функция особого типа, называемая предикатом. Предикат может быть унарным и бинарным.

Возвращаемое значение: истина либо ложь. Точные условия получения того или иного значения определяются программистом. Тип унарных предикатов UnPred, бинарных - BinPred . Тип аргументов соответствует типу хранящихся в контейнере объектов.

Определен специальный тип бинарного предиката для сравнения двух элементов. Он называется функцией сравнения (comparison function).

Функция возвращает истину, если первый элемент меньше второго. Типом функции является тип Comp .

Особую роль в STL играют **объекты-функции**.

Объекты-функции - это экземпляры класса, в котором определена операция "круглые скобки" (). В ряде случаев удобно заменить функцию на объект-функцию. Когда объект-функция используется в качестве функции, то для ее вызова используется operator().

`class less`

```

{
public:
    bool operator()(int x, int y)
    {
        return x < y;
    }
};

```

В STL определены два типа контейнеров: последовательности и ассоциативные.

Ключевая идея для стандартных контейнеров заключается в том, что когда это представляется разумным, они должны быть логически взаимозаменяемыми. Пользователь может выбирать между ними, основываясь на соображениях эффективности и потребности в специализированных операциях. Например, если часто требуется поиск по ключу, можно воспользоваться `map` (ассоциативным массивом). С другой стороны, если преобладают операции, характерные для списков, можно воспользоваться контейнером `list`. Если добавление и удаление элементов часто производится в конце контейнера, следует подумать об использовании очереди `queue`, очереди с двумя концами `deque`, стека `stack`. По умолчанию пользователь должен использовать `vector`; он реализован, чтобы хорошо работать для самого широкого диапазона задач. Идея обращения с различными видами контейнеров и, в общем случае, со всеми видами источников информации - унифицированным способом ведет к понятию обобщенного программирования. Для поддержки этой идеи STL содержит множество обобщенных алгоритмов. Такие алгоритмы избавляют программиста от необходимости знать подробности отдельных контейнеров. В STL определены следующие классы-контейнеры (в угловых скобках указаны заголовочные файлы, где определены эти классы):

- ✓ `bitset` - множество битов `<bitset.h>`
- ✓ `vector` - динамический массив `<vector.h>`
- ✓ `list` - линейный список `<list.h>`
- ✓ `deque` - двусторонняя очередь `<deque.h>`
- ✓ `stack` - стек `<stack.h>`
- ✓ `queue` - очередь `<queue.h>`
- ✓ `priority_queue` - очередь с приоритетом `<queue.h>`
- ✓ `map` - ассоциативный список для хранения пар ключ/значение, где с каждым ключом связано одно значение `<map.h>`
- ✓ `multimap` - с каждым ключом связано два или более значений `<map.h>`
- ✓ `set` - множество `<set.h>`
- ✓ `multiset` - множество, в котором каждый элемент не обязательно уникален `<set.h>`

Типы:

- ✓ `value_type` - тип элемента

- ✓ `allocator_type` - тип распределителя памяти
- ✓ `size_type` - тип индексов, счетчика элементов и т.д.
- ✓ `iterator` - ведет себя как `value_type *`
- ✓ `reverse_iterator` просматривает контейнер в обратном порядке
- ✓ `reference` - ведет себя как `value_type &`
- ✓ `key_type` - тип ключа (только для ассоциативных контейнеров)
- ✓ `key_compare` - тип критерия сравнения (только для ассоциативных контейнеров)
- ✓ `mapped_type` - тип отображенного значения

Итераторы:

- ✓ `begin()` - указывает на первый элемент
- ✓ `end()` - указывает на элемент, следующий за последним
- ✓ `rbegin()` - указывает на первый элемент в обратной последовательности
- ✓ `rend()` - указывает на элемент, следующий за последним в обратной последовательности

Доступ к элементам:

- ✓ `front()` - ссылка на первый элемент
- ✓ `back()` - ссылка на последний элемент
- ✓ `operator [](i)` - доступ по индексу без проверки
- ✓ `at(i)` - доступ по индексу с проверкой

Включение элементов:

- ✓ `insert(p, x)` - добавление `x` перед элементом, на который указывает `p`
- ✓ `insert(p, n, x)` - добавление `n` копий `x` перед `p`
- ✓ `insert(p, first, last)` - добавление элементов из `[first:last]` перед `p`
- ✓ `push_back(x)` - добавление `x` в конец
- ✓ `push_front(x)` - добавление нового первого элемента (только для списков и очередей с двумя концами)

Удаление элементов:

- ✓ `pop_back()` - удаление последнего элемента
- ✓ `pop_front()` - удаление первого элемента (только для списков и очередей с двумя концами)
- ✓ `erase(p)` - удаление элемента в позиции `p`
- ✓ `erase(first, last)` - удаление элементов из `[first:last]`
- ✓ `clear()` - удаление всех элементов

Другие операции:

- ✓ `size()` - число элементов
- ✓ `empty()` - контейнер пуст
- ✓ `capacity()` - память, выделенная под вектор (только для векторов)
- ✓ `reserve(n)` - выделяет память для контейнера под `n` элементов
- ✓ `resize(n)` - изменяет размер контейнера (только для векторов, списков и очередей с двумя концами)
- ✓ `swap(x)` - обмен местами двух контейнеров
- ✓ `==, !=, <` операции сравнения

Операции присваивания:

- ✓ operator =(x) - контейнеру присваиваются элементы контейнера x
- ✓ assign(n, x) - присваивание контейнеру n копий элементов x (не для ассоциативных контейнеров)
- ✓ assign(first, last) - присваивание элементов из диапазона [first:last]

Ассоциативные операции:

- ✓ operator [](k) - доступ к элементу с ключом k
- ✓ find(k) - находит элемент с ключом k
- ✓ lower_bound(k) - находит первый элемент с ключом k
- ✓ upper_bound(k) - находит первый элемент с ключом, большим k
- ✓ equal_range(k) - находит lower_bound (нижнюю границу) и upper_bound (верхнюю границу) элементов с ключом k

Вектор vector в STL определен как динамический массив с доступом к его элементам по индексу.

```
template <class T, class Allocator = allocator<T> > class
std::vector
{
// ...
```

}; где T - тип предназначенных для хранения данных. Allocator задает распределитель памяти, который по умолчанию является стандартным.

В классе vector определены следующие конструкторы:

```
explicit vector(const Allocator &a = Allocator());
```

```
explicit vector(size_type число, const T &значение = T(),
               const Allocator &a = Allocator());
```

```
vector(const vector<T, Allocator> &объект);
```

```
template<class InIter> vector(InIter начало, InIter конец,
                             const Allocator &a = Allocator());
```

Первая форма представляет собой конструктор пустого вектора. Во второй форме конструктора вектора число элементов - это число, а каждый элемент равен значению значение. Параметр значение может быть значением по умолчанию. Третья форма конструктора вектор - это конструктор копирования. Четвертая форма - это конструктор вектора, содержащего диапазон элементов, заданный итераторами начало и конец.

```
vector<int> a;
vector<double> x(5);
vector<char> c(5, '*');
vector<int> b(a); // b = a
```

Для любого объекта, который будет храниться в векторе, должен быть определен конструктор по умолчанию. Кроме того, для объекта должны быть определены операторы < и == .

Для класса вектор определены следующие операторы сравнения:

==, <, <=, !=, >, >=.

Кроме этого, для класса vector определяется оператор индекса [].

Новые элементы могут включаться с помощью функций insert(), push_back(), resize(), assign().

Существующие элементы могут удаляться с помощью функций erase(), pop_back(), resize(), clear()

Доступ к отдельным элементам осуществляется с помощью итераторов begin(), end(), rbegin(), rend().

Манипулирование контейнером, сортировка, поиск в нем и тому подобное возможно с помощью глобальных функций файла - заголовка <algorithm.h>.

```
#include <iostream>
#include <vector>

using namespace std;

int main(void)
{
    vector<int> v;
    for(int i = 0; i < 10; i++)
    {
        v.push_back(i);
    }
    cout << "size = " << v.size() << "\n";
    for (int i = 0; i < 10; i++)
    {
        cout << v[i] << " ";
    }
    cout << endl;
    for (int i = 0; i < 10; i++)
    {
        v[i] = v[i] + v[i];
    }
    for (int i = 0; i < v.size(); i++)
    {
        cout << v[i] << " ";
    }
    cout << endl;
    return 0;
}
```

Доступ к вектору через итератор


```

#include <iostream>
#include <vector>

using namespace std;

int main(void)
{
    vector<int> v;
    for (int i = 0; i < 10; i++)
    {
        v.push_back(i);
    }
    cout << "size = " << v.size() << "\n";
    vector<int>::iterator p = v.begin();
    while (p != v.end())
    {
        cout << *p << " ";
        p++;
    }
    return 0;
}

```

Вставка и удаление элементов

```

#include <iostream>
#include <vector>

using namespace std;

int main(void)
{
    vector<int> v(5, 1);
    // ВЫВОД
    for (int i = 0; i < 5; i++)
    {
        cout << v[i] << " ";
    }
    cout << endl;
    vector<int>::iterator p = v.begin();
    p += 2;
    // вставить 10 элементов со значением 9
    v.insert(p, 10, 9);
    //ВЫВОД
    p = v.begin();
    while (p != v.end())
    {
        cout << *p << " ";
        p++;
    }
    // удалить вставленные элементы
}

```

```

    p = v.begin();
    p += 2;
    v.erase (p, p + 10);
    // ВЫВОД
    p = v.begin();
    while (p != v.end())
    {
        cout << *p << " ";
        p++;
    }
    return 0;
}

```

Вектор содержит объекты пользовательского класса

```

#include <iostream>
#include <vector>
#include "student.h"

using namespace std;

int main(void)
{
    vector<STUDENT> v(3);
    v[0] = STUDENT("Иванов", 45.9);
    v[1] = STUDENT("Петров", 30.4);
    v[0] = STUDENT("Сидоров", 55.6);
    // ВЫВОД
    for (int i = 0; i < 3; i++)
    {
        cout << v[i] << " ";
    }
    cout << endl;
    return 0;
}

```

Ассоциативный массив содержит пары значений. Зная одно значение, называемое ключом (key), мы можем получить доступ к другому, называемому отображенным значением (mapped value).

Ассоциативный массив можно представить как массив, для которого индекс не обязательно должен иметь целочисленный тип:

`V &operator [] (const K &)` возвращает ссылку на `V`, соответствующий `K`.

Ассоциативные контейнеры - это обобщение понятия ассоциативного массива.

Ассоциативный контейнер `map` - это последовательность пар (ключ, значение), которая обеспечивает быстрое получение значения по ключу. Контейнер `map` предоставляет двунаправленные итераторы.

Ассоциативный контейнер `map` требует, чтобы для типов ключа существовала операция `<`. Он хранит свои элементы отсортированными по ключу так, что перебор происходит по порядку.

Спецификация шаблона для класса `map`:

```
template<class Key, class T, class Comp = less<Key>,
        class Allocator = allocator<pair> >
class std::map;
```

В классе `map` определены следующие конструкторы:

```
explicit map(const Comp &c = Comp(), const Allocator &a =
Allocator());
```

```
map(const map<Key, T, Comp, Allocator> &ob);
```

```
template<class InIter> map(InIter first, InIter last, const Comp &c =
Comp(),
                        const Allocator &a = Allocator());
```

Первая форма представляет собой конструктор пустого ассоциативного контейнера, вторая - конструктор копии, третья - конструктор ассоциативного контейнера, содержащего диапазон элементов.

Определена операция присваивания:

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

Определены следующие операции: `==`, `<`, `<=`, `!=`, `>`, `>=`.

В `map` хранятся пары ключ/значение в виде объектов типа `pair`.

Создавать пары ключ/значение можно не только с помощью конструкторов класса `pair`, но и с помощью функции `make_pair`, которая создает объекты типа `pair`, используя типы данных в качестве параметров.

Типичная операция для ассоциативного контейнера - это ассоциативный поиск при помощи операции индексации (`[]`).

```
mapped_type &operator [](const key_type &k);
```

Множества `set` можно рассматривать как ассоциативные массивы, в которых значения не играют роли, так что мы отслеживаем только ключи.

```
template<class T, class Cmp = less<T>, class Allocator = allocator<T> >
class std::set
{
    //...
};
```

Множество, как и ассоциативный массив, требует, чтобы для типа T существовала операция "меньше" (<). Оно хранит свои элементы отсортированными, так что перебор происходит по порядку.

Каждый алгоритм выражается шаблоном функции или набором шаблонов функций. Таким образом, алгоритм может работать с очень разными контейнерами, содержащими значения разнообразных типов. Алгоритмы, которые возвращают итератор, как правило, для сообщения о неудаче используют конец входной последовательности. Алгоритмы не выполняют проверки диапазона на их входе и выходе. Когда алгоритм возвращает итератор, это будет итератор того же типа, что и был на входе. Алгоритмы в STL реализуют большинство распространенных универсальных операций с контейнерами, такие как просмотр, сортировка, поиск, вставка и удаление элементов.

Алгоритмы определены в заголовочном файле <algorithm.h>.

Ниже приведены имена некоторых наиболее часто используемых функций-алгоритмов STL.

Немодифицирующие операции.

for_each() - выполняет операции для каждого элемента последовательности

find() - находит первое вхождение значения в последовательность

find_if() - находит первое соответствие предикату в последовательности

count() - подсчитывает количество вхождений значения в последовательность

count_if() - подсчитывает количество выполнений предиката в последовательности

search() - находит первое вхождение последовательности как подпоследовательности

search_n() - находит n-е вхождение значения в последовательность

Модифицирующие операции.

copy() - копирует последовательность, начиная с первого элемента

swap() - меняет местами два элемента

replace() - заменяет элементы с указанным значением

replace_if() - заменяет элементы при выполнении предиката

replace_copy() - копирует последовательность, заменяя элементы с указанным значением

replace_copy_if() - копирует последовательность, заменяя элементы при выполнении предиката

fill() - заменяет все элементы данным значением

remove() - удаляет элементы с данным значением

remove_if() - удаляет элементы при выполнении предиката

`remove_copy()` - копирует последовательность, удаляя элементы с указанным значением
`remove_copy_if()` - копирует последовательность, удаляя элементы при выполнении предиката
`reverse()` - меняет порядок следования элементов на обратный
`random_shuffle()` - перемещает элементы согласно случайному равномерному распределению ("тасует" последовательность)
`transform()` - выполняет заданную операцию над каждым элементом последовательности
`unique()` - удаляет равные соседние элементы
`unique_copy()` - копирует последовательность, удаляя равные соседние элементы

Сортировка.

`sort()` - сортирует последовательность с хорошей средней эффективностью
`partial_sort()` - сортирует часть последовательности
`stable_sort()` - сортирует последовательность, сохраняя порядок следования равных элементов
`lower_bound()` - находит первое вхождение значения в отсортированной последовательности
`upper_bound()` - находит первый элемент, больший чем заданное значение
`binary_search()` - определяет, есть ли данный элемент в отсортированной последовательности
`merge()` - сливает две отсортированные последовательности

Работа с множествами.

`includes()` - проверка на вхождение
`set_union()` - объединение множеств
`set_intersection()` - пересечение множеств
`set_difference()` - разность множеств

Минимумы и максимумы.

`min()` - меньшее из двух
`max()` - большее из двух
`min_element()` - наименьшее значение в последовательности
`max_element()` - наибольшее значение в последовательности

Перестановки.

`next_permutation()` - следующая перестановка в лексикографическом порядке
`prev_permutation()` - предыдущая перестановка в лексикографическом порядке

Вопросы

1. Что такое контейнер?
2. Что такое итератор?
3. Что такое функциональный объект?
4. Что такое адаптер контейнера?
5. Перечислите последовательные контейнеры STL.
6. Перечислите ассоциативные контейнеры STL.
7. Истинно ли утверждение о том, что одной из функций итераторов STL является связывание алгоритмов и контейнеров?
8. Истинно ли утверждение о том, что алгоритмы могут использоваться только с контейнерами STL? Синтаксис объявления шаблона функции.
9. Какая сущность зачастую используется для изменения поведения алгоритма?
10. В каких случаях вектор является подходящим контейнером?
11. Для чего используется алгоритм `unique()`?
12. Истинно ли утверждение о том, что итератор всегда может сдвигаться как в прямом, так и в обратном направлении по контейнеру?
13. Пусть `iter` – это итератор контейнера. Напишите выражение, имеющее значением объект, на который ссылается `iter`, и заставляющее затем `iter` сдвинуться на следующий элемент контейнера.
14. Если в множество добавить ключи методом `insert()`, будут ли они отсортированы?
15. Что такое распределители памяти, и для чего они используются?
16. Могут ли STL-алгоритмы работать с массивами стиля C на базе указателей?
17. В чем разница между операциями и алгоритмами при работе с контейнерами STL?
18. Что будет выведена на экран?

```
int main()
{
    std::vector<bool> v = {true, false, true, false};
    std::vector<int> v2(v.begin(), v.end()); // 1
    v.flip(); // 2
    for (std::size_t i=0; i != v.size(); ++i)
        std::cout << (v[i] & v2[i]);
}
```

19. Что будет выведено на экран?

```

#include <iostream>
#include <sstream>
#include <algorithm>
#include <iterator>

int main()
{
    std::string str = "To be or not to be...", str2;
    std::istringstream ist(str);
    for (int i = 0; i < 2; i++)
    {
        std::copy(std::istream_iterator<char>(ist),
                  std::istream_iterator<char>(),
                  std::back_inserter(str2) );
        ist.str(str);
    }
    std::cout << str2;
    return 0;
}

```

20. Что будет выведено на консоль следующим кодом?

```

#include <iostream>
#include <vector>
#include <algorithm>
#include <iterator>
using namespace std;

int main() {
    int Arr[] = {1,2,3,1,2,3};
    vector<int> Vec(Arr, Arr+(sizeof(Arr)/sizeof(Arr[0])));
    sort(Vec.begin(), Vec.end());
    unique_copy(Vec.begin(), Vec.end(), ostream_iterator<int>(cout, " "));
    return 0;
}

```

21. Что выведет данная программа:

```

#include <utility>
#include <vector>
#include <algorithm>
#include <iostream>

```

```

typedef std::pair<int, int> p_int;

int main()
{
    std::vector<p_int> v;

    v.push_back(std::make_pair(3, 2));
    v.push_back(std::make_pair(2, 3));
    v.push_back(std::make_pair(1, 4));
    v.push_back(std::make_pair(1, 3));

    std::sort(v.begin(), v.end());
    for (auto &x : v)
        std::cout << x.first << x.second;
}

```

22. Какой контейнер необходимо использовать для быстрого получения (с логарифмической сложностью) значения по ключу? Ключ типа int, значение типа string.

23. К каким из этих контейнеров применима функция сортировки из <algorithm>: template<class _RanIt> inline void std::sort(_RanIt _First, _RanIt _Last)

- ✓ std::deque
- ✓ std::map
- ✓ std::set
- ✓ std::list
- ✓ std::vector

24. Что будет выведено на экран?

```

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

int main()
{
    multimap<char, int> mm{{'a',1},{'b',2},{'c',3},{'a',4},{'a',5},{'c',1}};
    map<char, int> m(++mm.begin(), mm.lower_bound('c'));
    for(auto i: m)
        ++i.second;
    for(auto i: m)
        cout << i.second << ' ';
}

```



```
    return 0;
}
```

25. Что выведет такая программа?

```
#include <vector>
#include <iostream>

struct A {
    A(int i) { std::cout << "A(" << i << ")"; }
    ~A() { std::cout << "~A()"; }
};

int main() {
    std::vector<A *> a;

    for (int i = 0; i < 3; i++) {
        a.push_back(new A(i));
    }

    return 0;
}
```

26. По какому критерию сортируются ассоциативные контейнеры map при создании без явного указания критерия сортировки?

27. Что выведет такая программа?

```
#include <iostream>
#include <list>
#include <iterator>
#include <algorithm>
using namespace std;

int main() {
    list<int> coll;
    for(int i=1; i<=9; ++i)
    {
        coll.push_back(i);
    }
    list<int>::iterator pos;
    pos=find(coll.begin(), coll.end(), 5);
    cout<< *pos << ' ';
```

```
list<int>::reverse_iterator rpos(pos);
cout<< *rpos <<' ';
list<int>::iterator rpos;
rrpos=rpos.base();
cout<< *rrpos <<' ';
return 0;
}
```

28. Какие утверждения о предикатах и функторах верны (укажите все подходящие варианты)?

- ✓ Предикат может быть только структурой, а функтор - еще и классом.
- ✓ Предикаты могут использоваться для сортировки элементов в контейнерах
- ✓ Предикат - частный случай функтора.
- ✓ Метод operator() функтора может возвращать только значения типа bool.
- ✓ Для функтора должен быть переопределен operator<

29. Какая ошибка произойдет в этом коде?

```
#include <vector>
#include <algorithm>
#include <cmath>

int main(int argc, char *argv[])
{
    const int Xmin = -6;
    const int Xmax = 6;

    std::vector <double> xArray;
    std::vector <double> yArray;
    for (int i=Xmin; i < Xmax; i += 0.001)
    {
        xArray.push_back(i);
        yArray.push_back(sin(i));
        i += 0.2;
    }

    return 0;
}
```

30. Куда указывает итератор ptr после выполнения следующих двух строк кода?

```
vector<int> vec(100);  
vector<int>::iterator ptr=vec.end();
```

№ 12 ООП анализ и проектирование

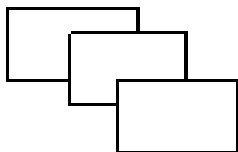
№ 13 Создание приложений и в Win32 API C++

Создайте пустой проект типа Win32 Application. Создайте главное окно приложения определенного размера с заголовком, определенного цвета и стиля. Напишите обработчики сообщений

1,10,19	реакция на щелчок правой кнопки мыши - меняется заголовок
2,11,20	реакция на щелчок левой кнопки мыши - меняется заголовок
3,12,21	реакция на двойной щелчок левой кнопки мыши - меняется заголовок
4,13,22	реакция на ввод смвола - меняется позиция окна
5,14,23	реакция на щелчок правой кнопки мыши - меняется размер окна
6,15,24	реакция на перемещение мыши - меняется размер и позиция окна
7,16,26	реакция на двойной щелчок правой кнопки мыши - меняется размер окна и заголовок
8,17,26	реакция на двойной щелчок левой кнопки мыши - меняется размер и позиция окна
9,18,27	реакция на щелчок левой кнопки мыши - меняется размер и позиция окна, заголовок окна

При закрытии приложения WM_CLOSE сделайте вывод окна сообщения MessageBox с двумя кнопками - OK и Cancel. Сделайте чтобы была активна вторая кнопка для чет. варианта и первая для нечет.

Добавить реакцию на щелчок левой кнопки мыши добавляется временное окно, в заголовке которого содержится порядковый номер окна. При повторном нажатии добавляется еще одно окно с соответствующим заголовком (максимум 7 окон). Окна должны быть расположены каскадом



При нажатии символа окна должны закрываться в обратном порядке.

Методические указания

Краткие теоретические сведения

Архитектура Windows-программ основана на принципе сообщений, а все программы содержат некоторые общие компоненты.

Функция окна

Все Windows-программы должны содержать специальную функцию, которая не используется в программе, но вызывается операционной системой. Эту функцию обычно называют функцией окна, или процедурой окна. Она вызывается Windows, когда системе необходимо передать сообщение в программу. Именно через нее осуществляется взаимодействие между программой и системой. Функция окна передает сообщение в своих аргументах. Согласно терминологии Windows, функции, вызываемые системой, называются функциями обратного вызова. Таким образом, функция окна является функцией обратного вызова. Помимо принятия сообщения от Windows, функция окна должна вызывать выполнение действия, указанного в сообщении. В большинстве Windows-программ задача создания функции окна лежит на разработчике.

Цикл сообщений

Все приложения Windows должны организовать так называемый цикл сообщений (обычно внутри функции WinMain()). В этом цикле каждое необработанное сообщение должно быть извлечено из очереди сообщений данного приложения и передано назад в Windows, которая затем вызывает функцию окна программы с данным сообщением в качестве аргумента. В традиционных Windows-программах необходимо самостоятельно создавать и активизировать такой цикл.

Класс окна

Каждое окно в Windows-приложении характеризуется определенными атрибутами, называемыми классом окна (понятие «класс» означает стиль или тип). В традиционной программе класс окна должен быть определен и зарегистрирован прежде, чем будет создано окно. При регистрации необходимо сообщить Windows, какой вид должно иметь окно и какую функцию оно выполняет. В то же время регистрация класса окна еще не означает создание самого окна. Для этого требуется выполнить дополнительные действия.

Существует три основных типа окон - перекрывающиеся, всплывающие и дочерние, из которых можно создавать множество самых разнообразных объектов, комбинируя биты стиля. Перекрывающиеся (overlapped window) - основной, наиболее универсальный тип окон. Для их создания используется стиль WS_OVERLAPPEDWINDOW. Вспомогательный или всплывающие окна (popup window) Используется стиль WS_POPUP. Используются для диалоговых окон и окон сообщений. Дочерние окна (child window) Они связаны некоторыми характеристиками с главным окном, из которого они были созданы.

Типы данных в Windows

В Windows-программах вообще не слишком широко применяются стандартные типы данных такие как int или char*. Вместо них используются типы данных, определенные в различных библиотечных (header) файлах. Наиболее часто используемыми типами являются HANDLE (32-разрядное целое, используемое в качестве дескриптора), HWND (32-разрядное целое – дескриптор окна), BYTE (8-разрядное беззнаковое символьное значение), WORD (16-разрядное беззнаковое короткое целое), DWORD (беззнаковое длинное целое), UNIT (беззнаковое 32-разрядное целое), LONG (эквивалентен типу long), BOOL (целое и используется, когда значение может быть либо истинным, либо ложным), LPSTR (указатель на строку) и LPCSTR (константный (const) указатель на строку).

Программа для Windows

Минимальная программа для Windows состоит из двух функций: функции WinMain и функции окна или оконной процедуры. Функция WinMain состоит из трех функциональных частей: регистрация класса окна, создания главного окна приложения и цикла обработки сообщений. На некотором псевдоязыке программу для Windows можно записать следующим образом:

WinMain(список аргументов)

{

 Создание класса окна

 Создание экземпляра класса окна

 Пока не произошло необходимое для выхода событие

 Выбрать из очереди сообщений очередное сообщение

 Передать сообщение оконной функции

 Возврат из программы

}

WindowsFunction(список аргументов)

{

```
        Обработать полученное сообщение
        Возврат
    }
```

Функция WinMain

Рассмотрим простейшее приложение Win32:

```
int WINAPI WinMain(HINSTANCE    hInstance,
                   HINSTANCE    hPrevInstance,
                   LPSTR         lpCmdLine,
                   int           nCmdShow)
{ return 0; }
```

Это приложение ничего не делает и сразу же прекращает свою работу, возвращая управление ОС с кодом возврата 0.

WINAPI перед телом функции WinMain указывает компилятору на необходимость сгенерировать перед выполнением этой функции специальный пролог и эпилог необходимый для функции, в которой запускается и завершается программа. Если это слово будет отсутствовать, то программа будет сгенерированна неправильно.

Рассмотрим параметры, которые получает приложение от ОС в функции WinMain:

lpCmdLine - указатель на командную строку;

nCmdShow - код режима начального отображения главного окна приложения;

hInstance - дескриптор, ассоциируемый с текущим приложением, некоторые функции API могут потребовать его в качестве параметра. В основном он необходим при работе с ресурсами приложений, организации многозадачности и при создании оконных объектов;

hPrevInstance - параметр для совместимости с предыдущими версиями Win16; в Win32 не имеет никакого значения и всегда равен NULL.

Главное окно программы первое появляется и последним исчезает при работе с программой. Другие окна, которые создаются главным окном, взаимодействуют с ним и им же уничтожаются. Прежде чем его создать, необходимо зарегистрировать его класс, при этом имя класса главного окна должно быть уникальным, чтобы не возникало конфликта с классами окон других приложений.

При регистрации класса окна в нем задаются наиболее общие свойства тех оконных объектов, которые будут созданы на основе данного класса, и сведения о которых необходимы для системы.

Регистрация класса окна

Для регистрации класса окна рекомендуется использовать функцию

RegisterClassEx или RegisterClass.

ATOM RegisterClassEx (

 const WNDCLASSEX *lpwccx;);

//указатель на структуру, содержащую данные об регистрируемом классе

Функция возвращает уникальный целочисленный идентификатор (ATOM), которое уникально для каждого зарегистрированного класса окна или 0 - в случае неудачи. Параметр функции - указатель на структуру WNDCLASSEX или WNDCLASS, в которой содержатся все необходимые данные об классе окна. Прототип этой структуры:

typedef struct _WNDCLASSEX {

 UINT cbSize; //размер структуры в байтах

 UINT style; //стиль класса

 WNDPROC lpfnWndProc; //адрес функции обратного вызова для для

//приема сообщений предназначенных для

// данного класса окна

 int cbClsExtra; //число байт для хранения данных для класса

 int cbWndExtra; //число байт для хранения данных при создании

// каждого оконного объекта класса

 HANDLE hInstance; // дескриптор программного модуля

//который регистрирует класс

 HICON hIcon; // дескриптор большой иконки

 HCURSOR hCursor; // дескриптор курсора

 HBRUSH hbrBackground; //цвет кисти фона

 LPCTSTR lpszMenuName; //имя меню для этого класса окна

 LPCTSTR lpszClassName; //имя класса окна

 HICON hIconSm; // дескриптор маленькой иконки

} WNDCLASSEX;

Загрузить необходимую иконку можно с помощью функций LoadIcon (загружает только иконки размером 32*32) или LoadImage:

HICON LoadIcon(

 HINSTANCE hInstance, //Идентификатор программного модуля, из

 //ресурсов которого необходимо загрузить иконку

 //для загрузки из ресурсов ОС укажите NULL

 LPCTSTR lpIconName //имя иконки; для встроенных иконок этот

 //параметр может быть например IDI_APPLICATION

);

Для загрузки курсора используется функция LoadCursor:

```
HCURSOR LoadCursor(
    HINSTANCE hInstance, // Идентификатор программного модуля, из ресурсов
                          // которого необходимо загрузить рисунок курсора,
                          // для загрузки из ресурсов ОС укажите NULL
    LPCTSTR lpCursorName // имя курсора; для задания идентификатора о
                          //дного из встроенных курсоров в виде флажка
                          // Windows укажем этот параметр как IDC_ARROW
);
```

Всего в системе есть около двадцати predetermined цветов, доступных по своим константным номерам. При использовании какого-либо из них обязательно прибавлять к номеру 1 (т.к. первое значение для них равно 0) и преобразовывать к типу HBRUSH.

Стили окна - это битовые флаги, которые могут комбинироваться с помощью логической операции ИЛИ, всего их более 10:

CS_DBLCLKS - если нажимать клавиши мыши достаточно быстро, то система будет отправлять сообщение для программы о двойном щелчке; если его не установить, то как бы быстро не щелкали на клавиши мыши, сообщения о двойном щелчке не появятся;

CS_HREDRAW - если пользователь изменил ширину данного окна, то посылается сообщение об его перерисовки;

CS_VREDRAW - если пользователь изменил высоту данного окна, то посылается сообщение об его перерисовки.

Например, можно установить поле style равным: CS_HREDRAW | CS_VREDRAW | CS_DBLCLKS.

Теперь регистрируется определенный класс окна с помощью функции RegisterClassEx. Функция возвращает ATOM, или шестнадцатиразрядное целое число по которому система будет различать класс окна. Его не нужно запоминать, но стоит проанализировать на предмет неравенства этого значения нулю (если возвращенное число равно нулю, то зарегистрировать класс окна не удалось).

Каждый зарегистрированный класс окна необходимо обеспечить своей уникальной функцией обратного вызова и своим уникальным именем.

Пример регистрации класса окна:

```
ATOM MyRegisterClass(HINSTANCE hInstance)
{WNDCLASSEX wcex;
  wcex.cbSize      = sizeof(WNDCLASSEX);
  wcex.style       = CS_HREDRAW | CS_VREDRAW;
  wcex.lpfnWndProc = (WNDPROC)WndProc;
  wcex.cbClsExtra  = 0;
```

```

wcex.cbWndExtra      =0;
wcex.hInstance       =hInstance;
wcex.hIcon           =LoadIcon(hInstance,IDI_APPLICATION);
wcex.hCursor         =LoadCursor(NULL,IDC_ARROW);
wcex.hbrBackground   =(HBRUSH)(COLOR_WINDOW+1);
wcex.lpszMenuName     =NULL;
wcex.lpClassName     =szWindowClass;
wcex.hIconSm         =LoadIcon(wcex.hInstance,IDI_APPLICATION);
ATOM atom=::RegisterClassEx(&wcex);
if(atom) return atom;
else //если данная версия Windows не поддерживает расширенный
    //класс окна
{WNDCLASS wc;
wc.style              =CS_HREDRAW|CS_VREDRAW;
wc.lpfnWndProc        =(WNDPROC)WndProc;
wc.cbClsExtra         =0;
wc.cbWndExtra         =0;
wc.hInstance          =hInstance;
wc.hIcon              =LoadIcon(hInstance,IDI_APPLICATION);
wc.hCursor            =LoadCursor(NULL,IDC_ARROW);
wc.hbrBackground      =(HBRUSH)(COLOR_WINDOW+1);
wc.lpszMenuName       =NULL;
wcex.lpClassName      =szWindowClass;
return ::RegisterClass(&wc);
} }

```

Сообщения и функция обратного вызова

Если создается только один объект данного класса окон, то можно сразу приступить к обработке сообщения. Приложение может создать столько окон одинакового оконного класса, сколько нужно, но если приложение регистрирует внутри себя какой-то производный класс окна (не главное окно которое может быть создано только в одном экземпляре) и создает несколько объектов этого класса, то оно должно учитывать от какого оконного объекта пришло сообщение.

При регистрации оконного класса, для ОС передается указатель на функции, удовлетворяющий следующему прототипу (имя функции может быть любое):

```

LRESULT CALLBACK WndProc(
    HWND hwnd,    //дескриптор оконного объекта
    UINT uMsg,    //код сообщения

```

```
WPARAM wParam, //первый параметр сообщения  
LPARAM lParam //второй параметр сообщения  
);
```

Если создать объект зарегистрированного оконного класса, то все сообщения для объектов этого класса при передаче ему сообщений от объектов других классов, системы или пользователя, или от объекта того же класса, направляются в эту функцию.

Слово CALLBACK в объявлении функции говорит компилятору о необходимости генерации специального кода для входа и выхода из функции обратного вызова.

Тип значения, возвращаемый функции класса окна LRESULT, определен как 32-битное значащее целое, но в зависимости от сообщения в wParam, его, возможно, потребуется преобразовать к указателю на некоторое значение или к другому целому типу. Это значение определяет результат обработки сообщения, поэтому оно всегда должно передаваться ОС при завершение работы функции.

Параметры функции:

HWND hwnd - дескриптор объекта окна, для которого предназначено сообщение, если создано несколько объектов данного оконного класса, то по нему обработчик выбирает нужный; для главного окна приложения, которое существует в одном экземпляре его можно проигнорировать;

UINT wParam - код сообщения, беззнаковое целое;

LPARAM lParam и WPARAM wParam - 32-битные целые, в которых передаются параметры сообщения, их возможно потребуется преобразовать к указателям на некоторое значение или к другому целому типу.

Всего в Windows существует более 900 сообщений, которые может получить любая функция класса окна, и все они должны быть обработаны. Однако те сообщения, обрабатывать которые внутри функции класса окна нет необходимости, могут быть переданы ОС для их обработки по умолчанию. Для этого необходимо передать те параметры, которые были переданы функции обратного вызова, в специальную системную функцию обработки сообщения по умолчанию называемую DefWindowProc:

```
LRESULT DefWindowProc(  
    HWND hwnd, // дескриптор окна  
    UINT Msg, // сообщение  
    WPARAM wParam, // первый параметр сообщения  
    LPARAM lParam // второй параметр сообщения  
);
```

Эта функция принимает и обрабатывает по умолчанию любое сообщение, обрабатывать которое функция класса окна не будет; в любом приложении для любого класса окна, все сообщения, переданные этой функции будут обрабатываться одинаково.

Теперь можно написать простейшую функцию обработки сообщений:

```
LRESULT CALLBACK WndProc(HWND hwnd, UINT message,
                          WPARAM wParam,
                          LPARAM lParam)
{ return DefWindowProc(hwnd,message,wParam,lParam);}
```

При использования этой функции для обработки сообщений для некоторого класса окна, получается закрашенный цветом кисти фона для данного класса окна, чьи размеры и заголовок можно изменять и не реагирующее не на какие ваши действия, кроме приказа на удаления. Поэтому надо добавить:

```
Switch (message)
{
    case WM_DESTROY:
        PostQuitMessage(0);
        Break;
....}
```

Такое окно мало для чего пригодно, поэтому необходимо переопределить обработку сообщений по умолчанию.

Создание главного окна приложения

После регистрации класса окна, приложение должно создать вначале главное окно приложения, затем возможно и еще несколько окон связанных с главным окном. Создание любого объекта оконного типа, выполняется функцией CreateWindow:

```
HWND CreateWindow(
LPCTSTR lpClassName, //указатель на строку с именем класса окна
LPCTSTR lpWindowName, //указатель на строку с заголовком окна
DWORD dwStyle,       //стиль окна
int x,               //х-координата левого верхнего угла окна
int y,               //у-координата левого верхнего угла окна
int nWidth,          //ширина окна
int nHeight,         //высота окна
HWND hWndParent,     //декриптор родительского окна
HMENU hMenu,         //если есть у окна меню то его handle
HANDLE hInstance,    //дескриптор того программного модуля,
```

```
LPVOID lpParam    //который создает окно
                  //указатель на дополнительные параметры
);
```

Для главного окна приложения и для всплывающих окон координаты начала окна отсчитываются от левого верхнего угла экрана, а для дочерних окон от левого верхнего угла угла родительского окна.

Для главного окна приложения созданного со стилем WS_OVERLAPPED параметр hWndParent должен быть равен NULL, для дочернего окна должен быть равен дескриптору какого-нибудь из окон уже созданных данным приложением, для всплывающих окон или NULL или дескриптору одного из уже определенных окон.

Младшее слово в параметре стиля dwStyle специфицирует те необходимые свойства в поведении создаваемого объекта для окон данного класса, которые нужно знать операционной системе, а старшее слово в этом двойном слове специфицируют те свойства в поведении оконного объекта, которые необходимо знать функции обработчику сообщений для окон заданного класса. Для главного окна приложения старшее слово обычно не используется, поэтому рассмотрим те стили окна которые есть в младшем слове:

WS_OVERLAPPED - создать перекрывающееся окно имеющее рамку и заголовок, любое главное окно приложения должно иметь этот стиль;

WS_POPUP - создает всплывающее окно, то есть способное появляться в любой части экрана;

WS_CHILD - создает дочернее окно;

WS_CAPTION - создает окно, которое имеет заголовок, любое перекрывающееся окно создается с таким стилем автоматически;

WS_BORDER - создает окно, которое имеет толстую рамку вокруг окна, любое перекрывающееся окно создается с таким стилем автоматически;

WS_SIZEBOX - окно может изменять размер, если пользователь будет двигать рамку окна;

WS_SYSMENU - окно имеет системное меню;

WS_MAXIMIZEBOX и WS_MINIMIZEBOX - окно имеет кнопки увеличения размера до максимального и уменьшения размера до минимального соответственно;

WS_VISIBLE - после создание окно оно сразу появляется на экране и окну приходит сообщение на перерисовку, лучше его использовать для дочерних и всплывающих окон, для главного окна приложения стоит использовать другой метод появления на экране;

WS_OVERLAPPEDWINDOW - рекомендуемый стиль перекрывающегося окна для главного окна приложения, определен как битовая комбинация

следующих стилей: WS_OVERLAPPED, WS_CAPTION, WS_SYSMENU, WS_THICKFRAME, WS_MINIMIZEBOX, WS_MAXIMIZEBOX.

Для создания главного окна приложения можно использовать следующую функцию:

```
BOOL InitInstance(HINSTANCE hInstance, int nCmdShow)
{
    HWND hWnd;
    hInst=hInstance;
    hWnd=CreateWindow(szWindowClass,szTitle,WS_OVERLAPPEDWINDOW,
                    CW_USERDEFAULT,0,
                    CW_USERDEFAULT,0,
                    NULL,NULL,hInstance,NULL);
    if(!hWnd)
    return FALSE;
    ShowWindow(hWnd,nCmdShow);
    UpdateWindow(hWnd);
    return TRUE;
}
```

После возврата из функции CreateWindow система записывает в свою внутреннюю базу данных информацию, необходимую для сопровождения данного окна. Но при этом окно не появляется. Требуется вызов: ShowWindow(hWnd,nCmdShow); UpdateWindow(hWnd).

Цикл обработки сообщений

Простейший цикл обработки сообщений выглядит следующим образом:

```
while(GetMessage(&msg,NULL,0,0))
{
    TranslateMessage(&msg);
    DispatchMessage(&msg)
}
```

Из очереди приложения сообщение выбирается функцией GetMessage(). Первый ее параметр (&msg) - указатель на структуру типа MSG, т.е. на сообщение. Второй параметр – дескриптор окна, созданного программой. Сообщение, адресованное только этому окну будет выбираться функцией GetMessage и передаваться оконной функции. Третий и четвертый параметры позволяют передавать оконной функции не все сообщения, а только те, номера которых попадают в определенный интервал.

Функция TranslateMessage() преобразует некоторые сообщения в более удобный для обработки вид. Функция DispatchMessage() передает сообщение оконной функции.

Завершение цикла обработки сообщений происходит при выборке из очереди сообщения WM_DESTROY, в ответ на которое функция GetMessage() возвращает нулевое значение.

Пример функции WinMain()

```
int WINAPI WinMain(HINSTANCE hInstance,
                  HINSTANCE hPrevInstance,
                  LPSTR lpCmdLine,
                  int nCmdShow)
{
    MSG msg;
    MyRegisterClass(hInstance);
    if(!InitInstance(hInstance,nCmdShow)) return FALSE;
    while(GetMessage(&msg,NULL,0,0))
    {
        TranslateMessage(&msg);
        DispatchMessage(&msg)
    }
    return msg.wParam; }
```

Пример проекта Win32 API

```
#include "stdafx.h"
#include "Task1.h"
#include <windows.h>
LONG WINAPI WndProc(HWND, UINT, WPARAM,LPARAM);
int WINAPI WinMain(    HINSTANCE hInstance,
                      HINSTANCE hPrevInstance,
                      LPSTR lpCmdLine,
                      int nCmdShow)
{
    HWND hwnd;
    MSG msg;
    WNDCLASS w;
    memset(&w,0,sizeof(WNDCLASS));
    w.style = CS_HREDRAW|CS_VREDRAW;
    w.lpfnWndProc = WndProc;
    w.hInstance = hInstance;
    w.hbrBackground = (HBRUSH)GetStockObject(WHITE_BRUSH);
    w.lpszClassName = L"My Class";
```



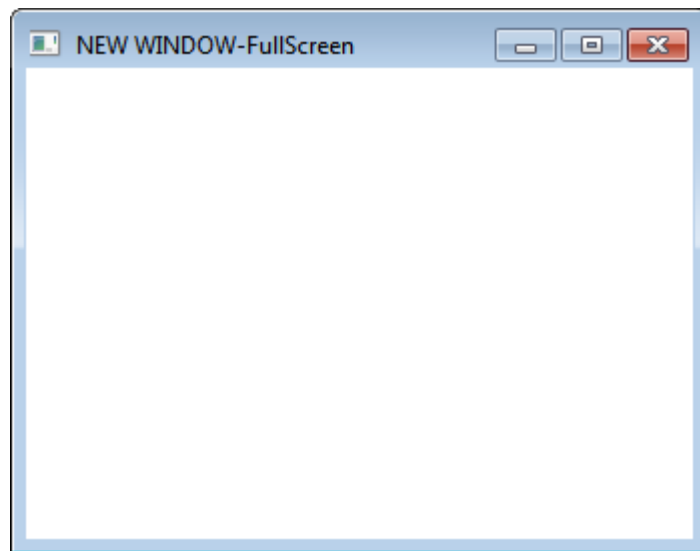
```

    RegisterClass(&w);
    hwnd = CreateWindow(L"My Class", L"PRESS DOUBLE CLICK!!!",
        WS_OVERLAPPEDWINDOW, 300, 200, 400, 280, NULL, NULL, hInstance, NULL);
    ShowWindow(hwnd, nCmdShow);
    UpdateWindow(hwnd);
    while(GetMessage(&msg, NULL, 0, 0))
    {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }
    return msg.wParam;
}

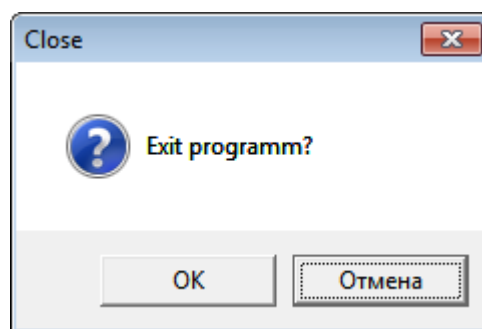
LONG WINAPI WndProc( HWND hwnd,
                    UINT Message,
                    WPARAM wparam,
                    LPARAM lparam)
{
    switch (Message)
    { case WM_DESTROY:
        PostQuitMessage(0);
        break;
      case WM_LBUTTONDOWN:
        SetWindowText(hwnd, L" NEW WINDOW-FullScreen");
        break;
      case WM_CLOSE:
        if(IDOK==MessageBox(hwnd, L"Exit programm?", L"Close",
            MB_OKCANCEL|MB_ICONQUESTION|MB_DEFBUTTON2))
            SendMessage(hwnd, WM_DESTROY, NULL, NULL);
        break;
      default: return DefWindowProc(hwnd, Message, wparam, lparam);
    }
    return 0;
}

```

Результат выполнения программы.

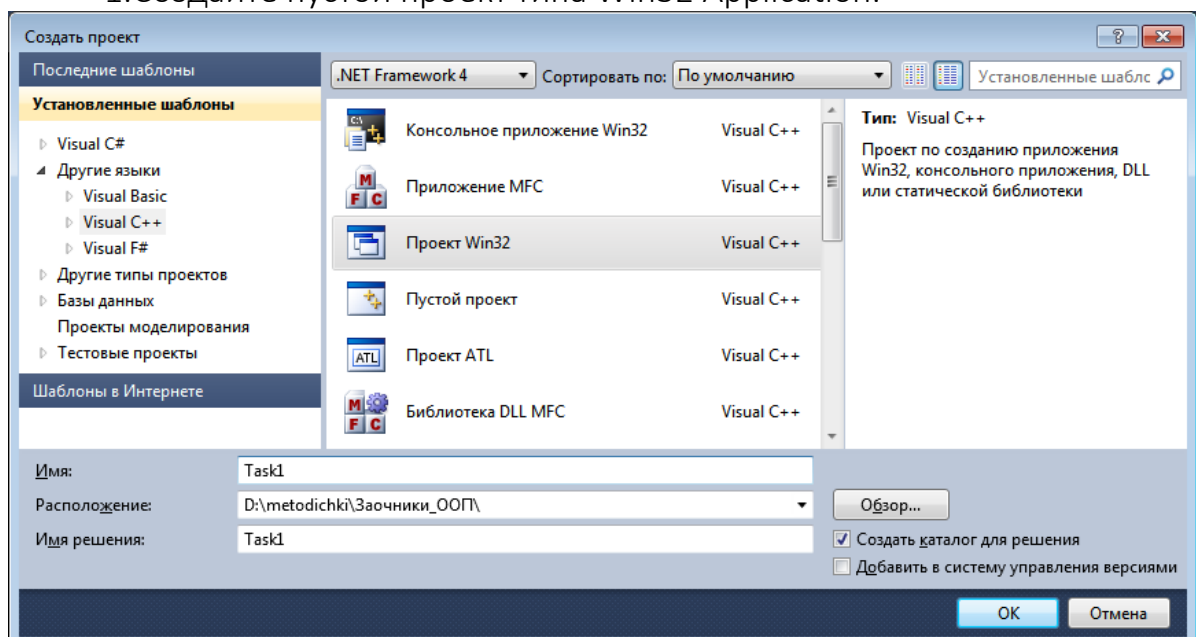


При закрытии программы появляется диалоговое окно, следующего вида:



Задания и порядок выполнения

1.Создайте пустой проект типа Win32 Application.



2. Ознакомьтесь с автоматически сгенерированным текстом приложения.

// Task1.cpp: определяет точку входа для приложения.

//

#include "stdafx.h"

#include "Task1.h"

#define MAX_LOADSTRING 100

// Глобальные переменные:

HINSTANCE hInst;

// текущий экземпляр

TCHAR szTitle[MAX_LOADSTRING];

// Текст строки заголовка

TCHAR szWindowClass[MAX_LOADSTRING];

// имя класса главного окна

// Отправить объявления функций, включенных в этот модуль кода:

ATOM MyRegisterClass(HINSTANCE hInstance);

BOOL InitInstance(HINSTANCE, int);

LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);

INT_PTR CALLBACK About(HWND, UINT, WPARAM, LPARAM);

int APIENTRY _tWinMain(HINSTANCE hInstance,

HINSTANCE hPrevInstance,

LPTSTR lpCmdLine,

int nCmdShow)

{

UNREFERENCED_PARAMETER(hPrevInstance);

UNREFERENCED_PARAMETER(lpCmdLine);

// TODO: разместите код здесь.

MSG msg;

HACCEL hAccelTable;

// Инициализация глобальных строк

LoadString(hInstance, IDS_APP_TITLE, szTitle, MAX_LOADSTRING);

LoadString(hInstance, IDC_TASK1, szWindowClass, MAX_LOADSTRING);

MyRegisterClass(hInstance);

// Выполнить инициализацию приложения:

if (!InitInstance (hInstance, nCmdShow))

{

return FALSE;

}

hAccelTable = LoadAccelerators(hInstance, MAKEINTRESOURCE(IDC_TASK1));

// Цикл основного сообщения:

while (GetMessage(&msg, NULL, 0, 0))

```

    {
        if (!TranslateAccelerator(msg.hwnd, hAccelTable, &msg))
        {
            TranslateMessage(&msg);
            DispatchMessage(&msg);
        }
    }

    return (int) msg.wParam;
}

//
// ФУНКЦИЯ: MyRegisterClass()
//
// НАЗНАЧЕНИЕ: регистрирует класс окна.
//
// КОММЕНТАРИИ:
//
// Эта функция и ее использование необходимы только в случае, если нужно, чтобы
// данный код
// был совместим с системами Win32, не имеющими функции RegisterClassEx'
// которая была добавлена в Windows 95. Вызов этой функции важен для того,
// чтобы приложение получило "качественные" мелкие значки и установило связь
// с ними.
//
ATOM MyRegisterClass(HINSTANCE hInstance)
{
    WNDCLASSEX wcex;

    wcex.cbSize = sizeof(WNDCLASSEX);

    wcex.style          = CS_HREDRAW | CS_VREDRAW;
    wcex.lpfnWndProc    = WndProc;
    wcex.cbClsExtra     = 0;
    wcex.cbWndExtra     = 0;
    wcex.hInstance      = hInstance;
    wcex.hIcon          = LoadIcon(hInstance, MAKEINTRESOURCE(IDI_TASK1));
    wcex.hCursor        = LoadCursor(NULL, IDC_ARROW);
    wcex.hbrBackground  = (HBRUSH)(COLOR_WINDOW+1);
    wcex.lpszMenuName   = MAKEINTRESOURCE(IDC_TASK1);
    wcex.lpszClassName  = szWindowClass;
    wcex.hIconSm        = LoadIcon(wcex.hInstance, MAKEINTRESOURCE(IDI_SMALL));

    return RegisterClassEx(&wcex);
}

//

```

```

// ФУНКЦИЯ: InitInstance(HINSTANCE, int)
//
// НАЗНАЧЕНИЕ: сохраняет обработку экземпляра и создает главное окно.
//
// КОММЕНТАРИИ:
//
// В данной функции дескриптор экземпляра сохраняется в глобальной переменной, а
также
// создается и выводится на экран главное окно программы.
//
BOOL InitInstance(HINSTANCE hInstance, int nCmdShow)
{
    HWND hWnd;

    hInst = hInstance; // Сохранить дескриптор экземпляра в глобальной переменной

    hWnd = CreateWindow(szWindowClass, szTitle, WS_OVERLAPPEDWINDOW,
        CW_USEDEFAULT, 0, CW_USEDEFAULT, 0, NULL, NULL, hInstance, NULL);

    if (!hWnd)
    {
        return FALSE;
    }

    ShowWindow(hWnd, nCmdShow);
    UpdateWindow(hWnd);

    return TRUE;
}

//
// ФУНКЦИЯ: WndProc(HWND, UINT, WPARAM, LPARAM)
//
// НАЗНАЧЕНИЕ: обрабатывает сообщения в главном окне.
//
// WM_COMMAND - обработка меню приложения
// WM_PAINT-Закрасить главное окно
// WM_DESTROY - ввести сообщение о выходе и вернуться.
//
//
LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)
{
    int wmId, wmEvent;
    PAINTSTRUCT ps;
    HDC hdc;

    switch (message)
    {
        case WM_COMMAND:

```

```

        wmlId = LOWORD(wParam);
        wmEvent = HIWORD(wParam);
        // Разобрать выбор в меню:
        switch (wmlId)
        {
        case IDM_ABOUT:
            DialogBox(hInst, MAKEINTRESOURCE(IDD_ABOUTBOX), hWnd, About);
            break;
        case IDM_EXIT:
            DestroyWindow(hWnd);
            break;
        default:
            return DefWindowProc(hWnd, message, wParam, lParam);
        }
        break;
    case WM_PAINT:
        hdc = BeginPaint(hWnd, &ps);
        // TODO: добавьте любой код отрисовки...
        EndPaint(hWnd, &ps);
        break;
    case WM_DESTROY:
        PostQuitMessage(0);
        break;
    default:
        return DefWindowProc(hWnd, message, wParam, lParam);
    }
    return 0;
}

```

// Обработчик сообщений для окна "О программе".

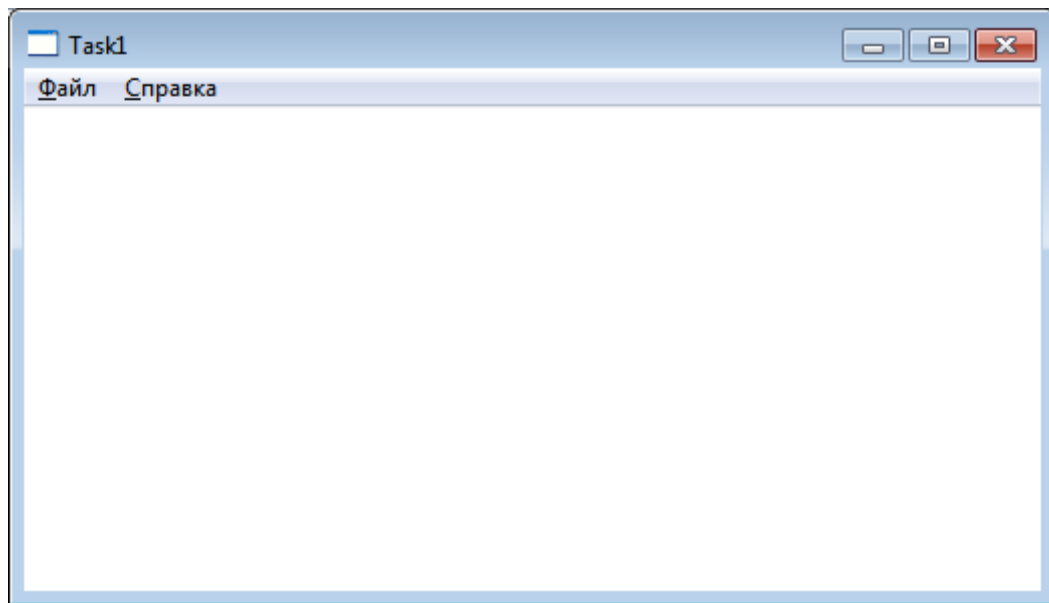
```

INT_PTR CALLBACK About(HWND hDlg, UINT message, WPARAM wParam, LPARAM lParam)
{
    UNREFERENCED_PARAMETER(lParam);
    switch (message)
    {
    case WM_INITDIALOG:
        return (INT_PTR)TRUE;

    case WM_COMMAND:
        if (LOWORD(wParam) == IDOK || LOWORD(wParam) == IDCANCEL)
        {
            EndDialog(hDlg, LOWORD(wParam));
            return (INT_PTR)TRUE;
        }
        break;
    }
    return (INT_PTR)FALSE;
}

```

3. Откомпилируйте проект. Получите результат.



4. Напишите обработчики следующих сообщений:

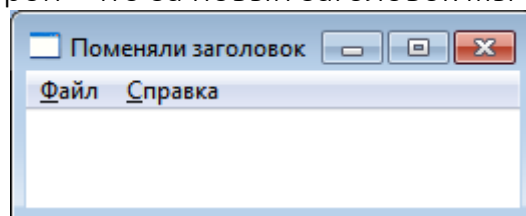
а) реакция на щелчок правой кнопки мыши - меняется заголовок

```
case WM_LBUTTONDOWN:
```

```
SetWindowText(hWnd, L"New title");
```

```
break;
```

Здесь мы вызываем API-функцию SetWindowText. Она меняет заголовок окна. У этой функции два параметра. Первый указывает на то, для какого окна мы будем это делать. Второй - что за новый заголовок мы установим.



б) реакция на ввод смвола - меняется размер и позиция окна

```
case WM_KEYDOWN:
```

```
{
```

```
    int new_x=10;
```

```
    int new_y =10;
```

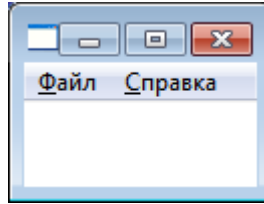
```
    int Width =100;
```

```
    int Height =100;
```

```
    MoveWindow(hWnd, new_x, new_y, Width, Height,TRUE);
```

```
}  
break;
```

Функция MoveWindow изменяет позицию и размеры окна



5. Можно пользоваться вспомогательными функциями.

- Изменить координаты окна hwnd на экране и его расположение по отношению к другим окнам.

```
BOOL SetWindowPos( HWND hwnd,  
int y, // новая координата верхнего края  
int cx, // новая ширина  
int cy, // новая высота  
UINT uFlags); // флажок позиционирования
```

- Двойной щелчок мыши WM_LBUTTONDOWNBLCLK. Сообщение относится только к окнам созданным со стилем CS_DBLCLKS. Указывается при регистрации класса окна.

Работа с некоторыми Win API функциями(информация о системе)

Некоторые Win API функции:

1) GetLogicalDrives

Функция GetLogicalDrives возвращает число-битовую маску в которой хранятся все доступные диски.

```
DWORD GetLogicalDrives(VOID);
```

Параметры:

Эта функция не имеет параметров.

Возвращаемое значение:

Если функция вызвана правильно, то она возвращает число-битовую маску в которой хранятся все доступные диски (если 0 бит равен 1, то диск "А:" присутствует, и т.д.)

Если функция вызвана не правильно, то она возвращает 0.

Пример:

```
int n;  
char dd[4];  
DWORD dr = GetLogicalDrives();  
for( int i = 0; i < 26; i++ )
```



```

{
    n = ((dr>>i)&0x00000001);
    if( n == 1 )
    {
        dd[0] = char(65+i); dd[1] = ':'; dd[2] = '\\'; dd[3] = 0;
        cout << "Available disk drives : " << dd << endl;
    }
}

```

2) GetDriveType

Функция GetDriveType возвращает тип диска (removable, fixed, CD-ROM, RAM disk, или network drive).

UINT GetDriveType(LPCTSTR lpRootPathName);

Параметры:

lpRootPathName

[in] Указатель на не нулевую строку в которой хранится имя главной директории на диске. Обратный слэш должен присутствовать! Если lpRootPathName равно NULL, то функция использует текущую директорию.

Возвращаемое значение:

Функция возвращает тип диска. Могут быть следующие значения:

Значение	Описание
DRIVE_UNKNOWN	Неизвестный тип.
DRIVE_NO_ROOT_DIR	Не правильный путь.
DRIVE_REMOVABLE	Съёмный диск.
DRIVE_FIXED	Фиксированный диск.
DRIVE_REMOTE	Удалённый или network диск.
DRIVE_CDROM	CD-ROM диск.
DRIVE_RAMDISK	RAM диск.

Пример:

```

int d;

d = GetDriveType( "c:\\");
if( d == DRIVE_UNKNOWN ) cout << " UNKNOWN" << endl;
if( d == DRIVE_NO_ROOT_DIR ) cout << " DRIVE NO ROOT DIR" << endl;
if( d == DRIVE_REMOVABLE ) cout << " REMOVABLE" << endl;
if( d == DRIVE_FIXED ) cout << " FIXED" << endl;
if( d == DRIVE_REMOTE ) cout << " REMOTE" << endl;
if( d == DRIVE_CDROM ) cout << " CDROM" << endl;
if( d == DRIVE_RAMDISK ) cout << " RAMDISK" << endl;

```

3) GetVolumeInformation

Функция GetVolumeInformation возвращает информацию о файловой системе и дисках(директориях).

```
BOOL GetVolumeInformation(
    LPCTSTR lpRootPathName,      // имя диска(директории)      [in]
    LPTSTR lpVolumeNameBuffer,    // название диска          [out]
    DWORD nVolumeNameSize,        // длина буфера названия диска [in]
    LPDWORD lpVolumeSerialNumber, // серийный номер диска     [out]
    LPDWORD lpMaximumComponentLength, // максимальная длина фыйла [out]
    LPDWORD lpFileSystemFlags,     // опции файловой системы   [out]
    LPTSTR lpFileSystemNameBuffer, // имя файловой системы     [out]
    DWORD nFileSystemNameSize      // длина буфера имени файл. сист. [in]
);
```

Возвращаемое значение:

Если функция вызвана правильно, то она возвращает не нулевое значение(TRUE).

Если функция вызвана не правильно, то она возвращает 0(FALSE).

Пример:

```
char VolumeNameBuffer[100];
char FileSystemNameBuffer[100];
unsigned long VolumeSerialNumber;
BOOL GetVolumeInformationFlag = GetVolumeInformationA(
    "c:\\",
    VolumeNameBuffer,
    100,
    &VolumeSerialNumber,
    NULL, //&MaximumComponentLength,
    NULL, //&FileSystemFlags,
    FileSystemNameBuffer,
    100
);
if(GetVolumeInformationFlag != 0)
{
    cout << "    Volume Name is " << VolumeNameBuffer << endl;
    cout << "    Volume Serial Number is " << VolumeSerialNumber << endl;
    cout << "    File System is " << FileSystemNameBuffer << endl;
}
else cout << " Not Present (GetVolumeInformation)" << endl;
```

4) GetDiskFreeSpaceEx

Функция GetDiskFreeSpaceEx выдаёт информацию о доступном месте на диске.

```

        BOOL GetDiskFreeSpaceEx(
            LPCTSTR lpDirectoryName,          // имя диска(директории)          [in]
            PULARGE_INTEGER lpFreeBytesAvailable, // доступно для
использования(байт) [out]
            PULARGE_INTEGER lpTotalNumberOfBytes, // максимальный объём( в
байтах ) [out]
            PULARGE_INTEGER lpTotalNumberOfFreeBytes // свободно на диске( в
байтах ) [out]
        );

```

Возвращаемое значение:

Если функция вызвана правильно, то она возвращает не нулевое значение(TRUE).

Если функция вызвана не правильно, то она возвращает 0(FALSE).

Пример:

```

DWORD FreeBytesAvailable;
DWORD TotalNumberOfBytes;
DWORD TotalNumberOfFreeBytes;

```

```

        BOOL GetDiskFreeSpaceFlag = GetDiskFreeSpaceEx(
            "c:\\",          // directory name
            (PULARGE_INTEGER)&FreeBytesAvailable, // bytes available to caller
            (PULARGE_INTEGER)&TotalNumberOfBytes, // bytes on disk
            (PULARGE_INTEGER)&TotalNumberOfFreeBytes // free bytes on disk
        );

```

```

        if(GetDiskFreeSpaceFlag != 0)
        {
            cout << " Total Number Of Free Bytes = " << (unsigned
long)TotalNumberOfFreeBytes
                << " ( " << double(unsigned long(TotalNumberOfFreeBytes))/1024/1000
                << " Mb )" << endl;
            cout << " Total Number Of Bytes = " << (unsigned long)TotalNumberOfBytes
                << " ( " << double(unsigned long(TotalNumberOfBytes))/1024/1000
                << " Mb )" << endl;
        }
        else cout << " Not Present (GetDiskFreeSpace)" << endl;

```

5) GlobalMemoryStatus

Функция GlobalMemoryStatus возвращает информацию о используемой системой памяти.

```

VOID GlobalMemoryStatus(

```

```

    LPMEMORYSTATUS lpBuffer // указатель на структуру MEMORYSTATUS
);
typedef struct _MEMORYSTATUS {
    DWORD dwLength;          // длина структуры в байтах
    DWORD dwMemoryLoad;      // загрузка памяти в процентах
    SIZE_T dwTotalPhys;      // максимальное количество физической
памяти в байтах
    SIZE_T dwAvailPhys;      // свободное количество физической памяти в
байтах
    SIZE_T dwTotalPageFile;  // макс. кол. памяти для программ в байтах
    SIZE_T dwAvailPageFile;  // свободное кол. памяти для программ в
байтах
    SIZE_T dwTotalVirtual;   // максимальное количество виртуальной
памяти в байтах
    SIZE_T dwAvailVirtual;   // свободное количество виртуальной
памяти в байтах
} MEMORYSTATUS, *LMEMORYSTATUS;

```

Возвращаемое значение:

Эта функция не возвращает параметров

Пример:

```

// The MemoryStatus structure is 32 bytes long.
// It should be 32.
// 78 percent of memory is in use.
// There are 65076 total Kbytes of physical memory.
// There are 13756 free Kbytes of physical memory.
// There are 150960 total Kbytes of paging file.
// There are 87816 free Kbytes of paging file.
// There are 1fff80 total Kbytes of virtual memory.
// There are 1fe770 free Kbytes of virtual memory.

```

```

#define DIV 1024
#define WIDTH 7
char *divisor = "K";

```

```

MEMORYSTATUS stat;
GlobalMemoryStatus (&stat);
printf ("The MemoryStatus structure is %ld bytes long.\n",
    stat.dwLength);
printf ("It should be %d.\n", sizeof (stat));
printf ("%ld percent of memory is in use.\n",

```

```

stat.dwMemoryLoad);
printf ("There are %*ld total %sbytes of physical memory.\n",
    WIDTH, stat.dwTotalPhys/DIV, divisor);
printf ("There are %*ld free %sbytes of physical memory.\n",
    WIDTH, stat.dwAvailPhys/DIV, divisor);
printf ("There are %*ld total %sbytes of paging file.\n",
    WIDTH, stat.dwTotalPageFile/DIV, divisor);
printf ("There are %*ld free %sbytes of paging file.\n",
    WIDTH, stat.dwAvailPageFile/DIV, divisor);
printf ("There are %*lx total %sbytes of virtual memory.\n",
    WIDTH, stat.dwTotalVirtual/DIV, divisor);
printf ("There are %*lx free %sbytes of virtual memory.\n",
    WIDTH, stat.dwAvailVirtual/DIV, divisor);

```

6) GetComputerName, GetUserNameA

Функция GetComputerName возвращает NetBIOS имя локального компьютера.

```

BOOL GetComputerName(
    LPTSTR lpBuffer,
    // имя локального компьютера( длина буфера равна
    MAX_COMPUTERNAME_LENGTH + 1 ) [out]
    LPDWORD lpnSize
    // размер буфера ( лучше поставить MAX_COMPUTERNAME_LENGTH + 1 )
    [out/in]
);

```

Функция GetUserName возвращает имя текущего юзера.

```

BOOL GetUserName(
    LPTSTR lpBuffer, // имя юзера( длина буфера равна UNLEN + 1 ) [out]
    LPDWORD nSize    // размер буфера ( лучше поставить UNLEN + 1 ) [out/in]
);

```

Возвращаемые значения:

Если функции вызваны правильно, то они возвращают не нулевое значение(TRUE).

Если функции вызваны не правильно, то они возвращают 0(FALSE).

Пример:

```

char ComputerName[MAX_COMPUTERNAME_LENGTH + 1];
unsigned long len_ComputerName = MAX_COMPUTERNAME_LENGTH + 1;
char UserName[UNLEN + 1];
unsigned long len_UserName = UNLEN + 1;

```

```

BOOL comp = GetComputerName(

```

```

ComputerName,
&len_ComputerName
);

if( comp != 0 ) { cout << "Computer Name is " << ComputerName << endl; }
else cout << "Computer Name is NOT FOUND !!! " << endl;
comp = GetUserNameA (
UserName,
&len_UserName
);
if( comp != 0 ) { cout << "User Name is " << UserName << endl; }
else cout << "User Name is NOT FOUND !!! " << endl;

```

7) GetSystemDirectory, GetTempPath, GetWindowsDirectory, GetCurrentDirectory
 Функция GetSystemDirectory возвращает путь к системной директории.

```

UINT GetSystemDirectory(
    LPTSTR lpBuffer, // буфер для системной директории [out]
    UINT uSize      // размер буфера [in]
);

```

Возвращаемое значение:

Эта функция возвращает размер буфера для системной директории не включая нулевого

значения в конце, если она вызвана правильно.

Если функция вызвана не правильно, то она возвращает 0.

Функция GetTempPath возвращает путь к директории, отведённой для временных файлов.

```

DWORD GetTempPath(
    DWORD nBufferLength, // размер буфера [in]
    LPTSTR lpBuffer      // буфер для временной директории [out]
);

```

Возвращаемое значение:

Эта функция возвращает размер буфера для системной директории не включая нулевого

значения в конце, если она вызвана правильно.

Если функция вызвана не правильно, то она возвращает 0.

Функция GetWindowsDirectory возвращает путь к Windows директории.

```

UINT GetWindowsDirectory(
    LPTSTR lpBuffer, // буфер для Windows директории [out]

```

```
UINT uSize    // размер буфера    [in]
);
```

Возвращаемое значение:

Эта функция возвращает размер буфера для системной директории не включая нулевого

значения в конце, если она вызвана правильно.

Если функция вызвана не правильно, то она возвращает 0.

Функция GetCurrentDirectory возвращает путь к текущей директории.

```
DWORD GetCurrentDirectory(
    DWORD nBufferLength, // размер буфера    [in]
    LPTSTR lpBuffer      // буфер для текущей директории [out]
);
```

Возвращаемое значение:

Эта функция возвращает размер буфера для системной директории не включая нулевого

значения в конце, если она вызвана правильно.

Если функция вызвана не правильно, то она возвращает 0.

Пример:

```
char path[100];
```

```
GetSystemDirectory( path, 100 );
cout << "System Directory is " << path << endl;
GetTempPath( 100, path );
cout << "Temp path is " << path << endl;
GetWindowsDirectory( path, 100 );
cout << "Windows directory is " << path << endl;
GetCurrentDirectory( 100, path );
cout << "Current directory is " << path << endl;
```

Контекст устройства

Для того, чтобы что-то нарисовать (вывести) необходимо получить контекст устройства (DC). Во всех случаях, кроме обработки сообщения WM_PAINT, для этой цели используется функция **GetDC()**.

```
HDC GetDC(HWND hWnd);
```

где *hWnd* - handle окна.

Для получения контекста устройства для обработчика сообщения **WM_PAINT** используется функция **BeginPaint()**

```
HDC BeginPaint(  
    HWND hwnd,      // дескриптор окна  
    LPPAINTSTRUCT lpPaint // информация  
);
```

За исключением самого первого сообщения **WM_PAINT**, посылаемого окну при вызове **UpdateWindow()**, эти сообщения будут посылаться в следующих случаях:

- при изменении размеров окна
- если рабочая область была скрыта меню или окном диалога, которое в данный момент закрывается;
- при использовании функции **ScrollWindow()**;
- при принудительной генерации сообщения **WM_PAINT** вызовом функций **InvalidateRect()** или **InvalidateRgn()**.

Вывод текста

Для вывода текста в области окна можно использовать функции **TextOut()** и **DrawText()**.

```
BOOL TextOut(  
    HDC hdc,      // дескриптор DC  
    int nXStart,  // x-координата начальной точки  
    int nYStart,  // y- координата начальной точки  
    LPCTSTR lpString, // строка символов  
    int cbString  // число символов  
);
```

```
int DrawText(  
    HDC hdc,      // handle to DC  
    LPCTSTR lpString, // text to draw  
    int nCount,   // text length  
    LPRECT lpRect, // formatting dimensions  
    UINT uFormat  // text-drawing options  
);
```

Для того, чтобы установить шрифт надо:

- 1) Создать структуру типа **LOGFONT**.


```
typedef struct tagLOGFONT {
    LONG lfHeight;
    LONG lfWidth;
    LONG lfEscapement;
    LONG lfOrientation;
    LONG lfWeight;
    BYTE lfItalic;
    BYTE lfUnderline;
    BYTE lfStrikeOut;
    BYTE lfCharSet;
    BYTE lfOutPrecision;
    BYTE lfClipPrecision;
    BYTE lfQuality;
    BYTE lfPitchAndFamily;
    TCHAR lfFaceName[LF_FACESIZE];
} LOGFONT;
```

Например,

```
LOGFONT logfont;
logfont.lfHeight = 50;
logfont.lfWidth = 10;
//....
strcpy(logfont.lfFaceName, "Arial");
//...
```

2) Создать шрифт вызовом функции **CreateFontIndirect()**:

```
HFONT CreateFontIndirect(CONST LOGFONT *lpf);
```

Например,

```
HFONT newfont=CreateFontIndirect(&logfont);
```

3) Установить шрифт в контексте устройства вызовом функции **SelectObject()**:

```
HGDIOBJ SelectObject(
    HDC hdc,          // handle to DC
    HGDIOBJ hgdiobj  // handle to object
```

```
);
```

Например,

```
SelectObject(hdc,(HGDIOBJ) newfont);
```

Если нужно установить цвет букв, отличный от по умолчанию, то вызываем функцию **SetTextColor()**:

```
COLORREF SetTextColor(  
    HDC hdc,      // handle to DC  
    COLORREF crColor // text color  
);
```

Например,

```
SetTextColor(hdc, RGB(255, 0,0));
```

Рисование линий и фигур

Для рисования линий и геометрических фигур используются различные функции API.

Например, эллипс:

```
BOOL Ellipse(  
    HDC hdc,      // handle to DC  
    int nLeftRect, // x-coord of upper-left corner of rectangle  
    int nTopRect,  // y-coord of upper-left corner of rectangle  
    int nRightRect, // x-coord of lower-right corner of rectangle  
    int nBottomRect // y-coord of lower-right corner of rectangle  
);
```

Прямоугольник:

```
BOOL Rectangle(  
    HDC hdc,      // handle to DC  
    int nLeftRect, // x-coord of upper-left corner of rectangle  
    int nTopRect,  // y-coord of upper-left corner of rectangle  
    int nRightRect, // x-coord of lower-right corner of rectangle  
    int nBottomRect // y-coord of lower-right corner of rectangle  
);
```

Цвет заливки замкнутой фигуры устанавливается следующим образом:

1) Создается кисть нужного цвета:

```
long color=RGB(125,125,225);  
HBRUSH hNew;  
hNew=CreateSolidBrush(color);
```

2) Устанавливается объект-кисть в контексте устройства:

```
SelectObject(hdc,hNew);
```

№ 14 Библиотека классов и обработка сообщений MFC

При разработке программ использовать библиотеку классов MFC

1,8,15,22	<p>Написать приложение, рисующее в окне различные графические фигуры(окружность, эллипс, прямоугольник, квадрат, сектор, сегмент) в ответ на нажатия клавиш «1», «2», «3», «4», «5», «6» соответственно. Каждую фигуру закрашивать своим цветом.</p> <p>Добавить обработку нажатия на левую клавишу мыши, при котором необходимо стереть и обновить все окно. При нажатии на правую клавишу мыши - вывести 5 раз с разными координатами (можно с одинаковым приращением) строку текста (с определенным размером, цветом, начертанием символов) через каждые 3 сек.</p>
2,9,16,23	<p>Написать приложение, выводящее в центре окна текст различным шрифтом(меняется имя шрифта, размер, цвет, начертание символов) в ответ на нажатия клавиш «1», «2», ... и т.д. (всего 9 различных стилей).</p> <p>Добавить обработку нажатия на правую клавишу мыши, при котором будут рисоваться окружности с разным цветом, радиусом и координатами. Двойное нажатие на правую клавишу мыши прекращает рисование и очищает окно.</p>
3,10,17,24	<p>Написать приложение, рисующее в окне различные графические фигуры(окружность, эллипс, прямоугольник, квадрат, сектор, сегмент) с поясняющей надписью (название фигуры). Фигуры выводятся по сигналу от таймера с заданной частотой. По каждому следующему сигналу таймера изображение на экране(фигуры) меняется.</p> <p>Добавить обработку двойного нажатия на левую клавишу мыши увеличивает окно и продолжает вывод фигур в соответствии с алгоритмом. При достижении максимального размера окна оно уменьшается.</p>
4,11,18,25	<p>Написать приложение, рисующее в окне различные графические фигуры(окружность, эллипс, прямоугольник, квадрат, сектор, сегмент) в ответ на нажатия клавиши мыши по</p>

	<p>циклу.</p> <p>Добавить обработку нажатия на левую клавиши мыши, при котором будет выводиться текущее время (менять шрифт, размер и цвет текста) <code>GetCurrentTime()</code>.</p>
5,12,19,26	<p>Написать приложение, выводящее в центре окна текст различным шрифтом(меняется имя шрифта, размер, цвет, начертание символов) в ответ на нажатия клавиши мыши (всего 9 различных стилей).</p> <p>Добавить обработку нажатия на клавишу клавиатуры, при которой выводятся прямоугольники слева направо, сверху вниз.</p>
6,13,20	<p>Написать приложение, рисующее в окне окружности разного радиуса и цвета в ответ на нажатия клавиш «1», «2», «3», «4», «5», «6» соответственно.</p> <p>Добавить обработку нажатия на левую клавишу мыши, при которой выводится текст 10 раз с увеличивающимся размером через каждые 2 сек.</p>
7,14,21	<p>Написать приложение, рисующее в ответ на нажатия клавиш «1», «2», «3», «4», «5», «6» соответствующие цифры с различным размером и цветом.</p> <p>Добавить обработку одинарного и двойного нажатия на правую и левую клавиши мыши, при которых выводятся эллипс, прямоугольник, квадрат, сектор соответственно.</p>

Теория

Имена, используемые в MFC

Библиотека MFC содержит большое количество классов, структур, констант и т.д. Для того, чтобы текст MFC-приложений был более легким для понимания, принято применять ряд соглашений для используемых имен и комментариев.

Названия всех классов и шаблонов классов библиотеки MFC начинаются с заглавной буквы C. При наследовании классов от классов MFC можно давать им любые имена. Рекомендуется начинать их названия с заглавной буквы C (class). Это сделает исходный текст приложения более ясным для понимания.

Чтобы отличить элементы данных, входящих в класс, от простых переменных, их имена принято начинать с префикса m_ (member – элемент или член класса). Названия методов классов, как правило, специально не выделяются, но обычно их начинают с заглавной буквы.

Библиотека MFC включает в себя, помимо классов, набор служебных функций или глобальных. Названия этих функций начинаются с символов Afx, например AfxGetApp. Символы AFX являются сокращением от словосочетания Application FrameworkX, означающих основу приложения, его внутреннее устройство. Например, очень часто используется функция AfxMessageBox(), отображающая заранее определенное окно сообщения. Но есть и член-функция MessageBox(). Таким образом, часто глобальные функции перекрываются функциями-членами.

Символы AFX встречаются не только в названии функций MFC. Многие константы, макрокоманды и другие символы начинаются с этих символов. В общем случае AFX является признаком, по которому можно определить принадлежность того или иного объекта (функция, переменная, ключевое слово или символ) к библиотеке MFC.

Функции-члены в MFC

Большинство функций, вызываемых в MFC-программе, являются членами одного из классов, определенных в библиотеке. Большинство функций API доступны через функции-члены MFC. Тем не менее, всегда можно обращаться к функциям API напрямую. Иногда это бывает необходимым, но все же в большинстве случаев удобнее использовать функции-члены MFC.

Иерархия классов MFC

Библиотека MFC содержит большую иерархию классов, написанных на C++. Структура иерархии приведена на рис. В ее вершине находится класс CObject, который содержит различные функции, используемые во время выполнения программы и предназначенные, в частности, для предоставления информации о текущем типе во время выполнения, для диагностики, и для сериализации. На вершине иерархии находится единственный базовый класс CObject. Все остальные классы MFC можно условно разделить в зависимости от их отношения к CObject на производные и непроизводные. Для того чтобы

составить представления о структуре и возможностях библиотеки приведем краткое описание классов.

Самый базовый класс библиотеки MFC (класс CObject). Методы и элементы данных класса CObject представляют наиболее общие свойства наследованных из него классов MFC. Основное назначение класса CObject: хранение информации о классе времени выполнения; поддержка сериализации и диагностики объекта.

Единственной переменной этого класса является статическая переменная ClassObject типа CRuntimeClass, которая хранит информацию об объекте времени выполнения, ассоциированным с классом CObject.

Сериализация

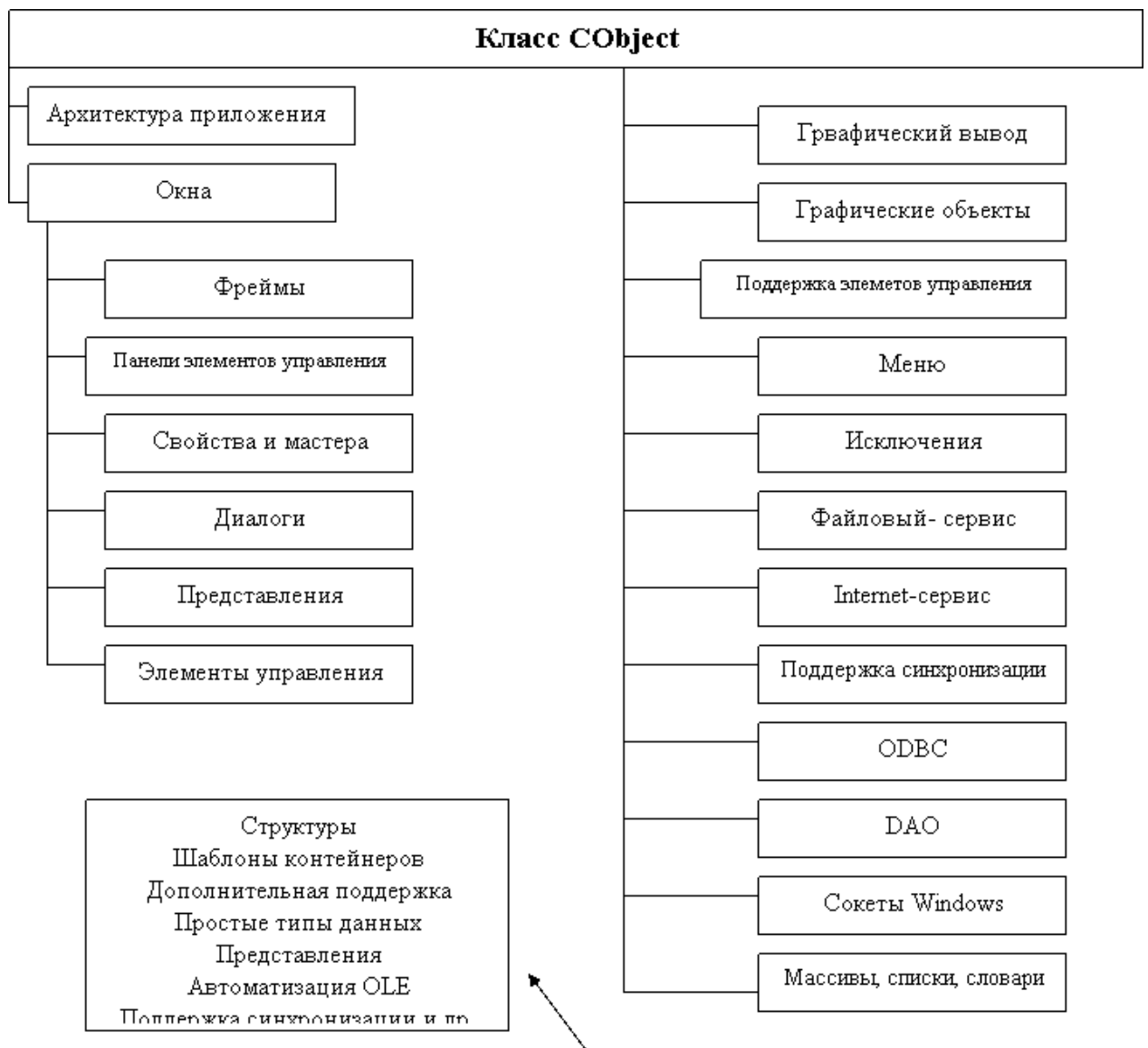
Сериализация - это механизм, позволяющий преобразовать текущее состояние объекта в последовательный поток байт, который обычно затем записывается на диск, и восстановить состояние объекта из последовательного потока, обычно при чтении с диска. Это позволяет сохранять текущее состояние приложения на диске, и восстанавливать его при последующем запуске. Этот метод можно разделить функционально на две составляющие:

Наличие определенной виртуальной функции Serialize позволяет унифицировать процесс сохранения/восстановления объектов. Единственное что нужно – переопределить функцию.

С другой стороны, сериализация поддерживает механизм динамического создания объектов неизвестного заранее типа. Например приложение должно сохранять и восстанавливать некоторое количество объектов различного типа.

Диагностика

Каждый класс, производный от CObject, может по запросу проверить свое внутреннее состояние и выдать диагностическую информацию. Это интенсивно используется в MFC при отладке.



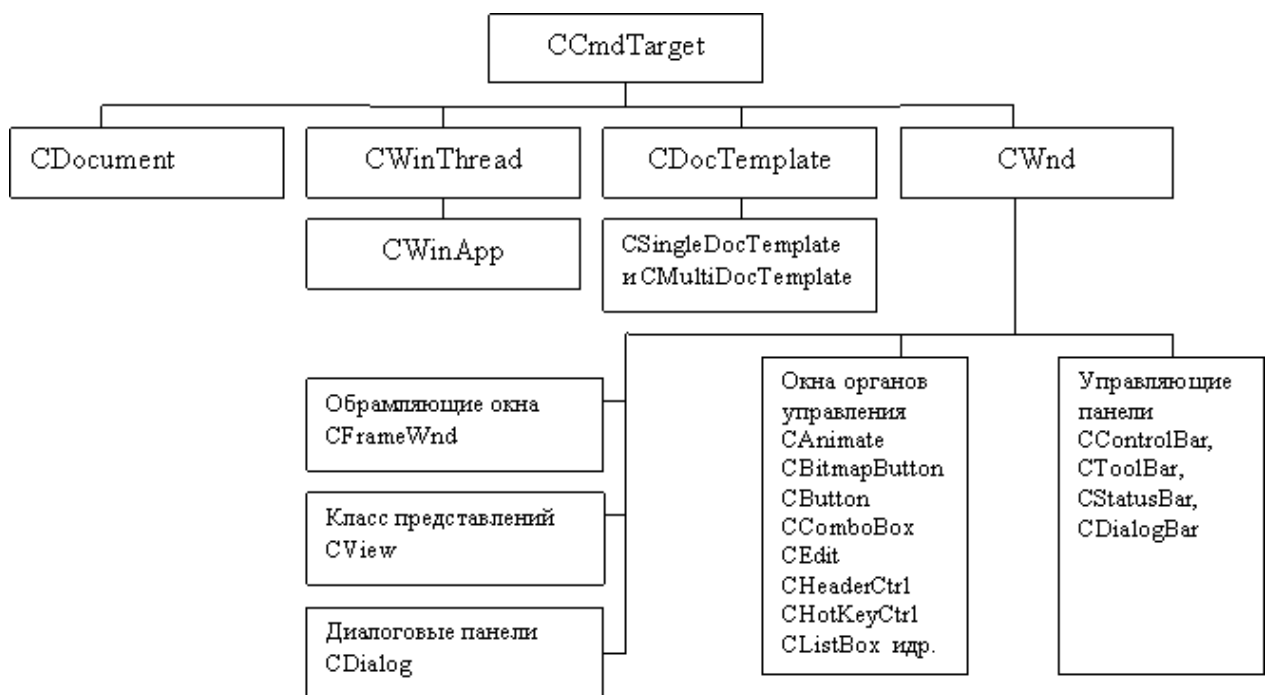
Классы, определяющие архитектуру приложения

Эти классы порождаются непосредственно от CObject. Класс CCmdTarget предназначен для обработки сообщений. Класс CFile предназначен для работы с файлами. Класс CDC обеспечивает поддержку контекстов устройств. Об контекстах устройств мы будем говорить несколько позднее. В этот класс включены практически все функции графики GDI. CGDIObject является базовым классом для различных DGI-объектов, таких как перья, кисти, шрифты и другие. Класс CMenu предназначен для манипуляций с меню. От класса CCmdTarget порождается очень важный класс CWnd. Он является базовым для создания всех типов окон, включая масштабируемые ("обычные") и диалоговые, а также различные элементы управления. Наиболее широко используемым производным классом является CFrameWnd. Как Вы увидите в

дальнейшем, в большинстве программ главное окно создается с помощью именно этого класса.

От класса `CCmdTarget`, через класс `CWinThread`, порождается, наверное, единственный из наиболее важных классов, обращение к которому в MFC-программах происходит напрямую: `CWinApp`. Это один из фундаментальных классов, поскольку предназначен для создания самого приложения. В каждой программе имеется один и только один объект этого класса. Как только он будет создан, приложение начнет выполняться.

Основа структуры приложения (класс `CCmdTarget`)



Подзадачи приложения (классы `CWinThread` и `CWinApp`)

От класса `CCmdTarget` наследуется класс `CWinThread`, представляющий подзадачи приложения. Простые приложения, которые будут рассматриваться дальше, имеют только одну подзадачу. Эта подзадача, называемая главной, представляется классом `CWinApp`, наследованным от класса `CWinThread`.

Документ приложения (класс `CDocument`)

Большинство приложений работают с данными или документами, хранимыми на диске в отдельных файлах. Класс CDocument, наследованный от базового класса CCmdTarget, служит для представления документов приложения.

Шаблон документов (классы CDocTemplate, CSingleDocTemplate и CMultiDocTemplate)

Еще один важный класс, наследуемый от CCmdTarget, называется CDocTemplate. От этого класса наследуются два класса: CSingleDocTemplate и CMultiDocTemplate. Все эти классы предназначены для синхронизации и управления основными объектами, представляющими приложение, - окнами, документами и используемыми ими ресурсами.

Окна (класс CWnd)

Практически все приложения имеют пользовательский интерфейс, построенный на основе окон. Это может быть диалоговая панель, одно окно или несколько окон, связанных вместе. Основные свойства окон представлены классом CWnd, наследованным от класса CCmdTarget.

Программисты очень редко создают объекты класса CWnd. Класс CWnd сам является базовым классом для большого количества классов, представляющих разнообразные окна. Перечислим классы, наследованные от базового класса CWnd.

Обрамляющие окна (класс CFrameWnd). Класс CFrameWnd представляет окна, выступающие в роли обрамляющих окон, в том числе главные окна приложений. От этого класса также наследуются классы CMDIChildWnd и CMDIFrameWnd, используемые для отображения окон многооконного интерфейса MDI. Класс CMDIFrameWnd представляет главное окно приложения MDI, а класс CMDIChildWnd - его дочерние окна MDI. Класс CMiniFrameWnd применяется для отображения окон уменьшенного размера. Такие окна обычно используются для отображения в них панели управления.

Окна органов управления

Для работы с органами управления (кнопки, полосы прокрутки, редакторы текста и т.д.) в библиотеке MFC предусмотрены специальные классы, наследованные непосредственно от класса CWnd. Перечислим эти классы:

- CAnimate - используется для отображения видеоинформации.
- CBitmapButton - кнопка с рисунком.
- CButton - кнопка.
- CComboBox - список с окном редактирования.
- CEdit - поле редактирования.
- CHeaderCtrl - заголовок для таблицы.
- CHotKeyCtrl - предназначен для ввода комбинации клавиш акселераторов.
- CListBox - список.
- CListCtrl - может использоваться для отображения списка пиктограмм.
- CProgressCtrl - линейный индикатор.
- CPropertySheet - блокнот. Может состоять из нескольких страниц.
- CRichEditControl - окно редактирования, в котором можно редактировать форматированный текст.
- CScrollBar - полоса просмотра.
- CSliderCtrl - движок.
- CSpinButtonCtrl - обычно используется для увеличения или уменьшения значения какого-либо параметра.
- CStatic - статический орган управления.
- CTabCtrl - набор "закладок".
- CToolBarCtrl - панель управления.
- CToolTipCtrl - маленькое окно, содержащее строку текста.
- CTreeCtrl - орган управления, который позволяет просматривать иерархические структуры данных.

Управляющие панели (классы CControlBar, CToolBar, CStatusBar, CDialogBar)

Класс CControlBar и классы, наследуемые от него, предназначены для создания управляющих панелей. Такие панели могут содержать различные органы управления и отображаются, как правило, в верхней или нижней части главного окна приложения.

Так, класс CToolBar предназначен для создания панели управления. Эта панель обычно содержит ряд кнопок, дублирующих действие меню приложения.

Класс CStatusBar управляет панелью состояния. Панель состояния отображается в виде полосы в нижней части окна. В ней приложение может отображать всевозможную информацию, например, краткую подсказку о выбранной строке меню.

Большие возможности представляет управляющая панель, созданная на основе класса CDialogBar. Такая панель использует обычный шаблон диалоговой панели, которую можно разработать в редакторе ресурсов Visual C++.

Окна просмотра (класс CView и классы, наследованные от него)

Большой интерес представляют класс CView и классы, наследуемые от него. Эти классы представляют окно просмотра документов приложения. Именно окно просмотра используется для вывода на экран документа, с которым работает приложение. Через это окно пользователь может изменять документ.

Разрабатывая приложение, программисты наследуют собственные классы просмотра документов либо от базового класса CView, либо от одного из нескольких порожденных классов, определенных в библиотеке MFC.

Классы, наследованные от CCtrlView, используют для отображения готовые органы управления. Например, класс CEditView использует орган управления edit (редактор).

Класс CScrollView представляет окно просмотра, которое имеет полосы свертки. В классе определены специальные методы, управляющие полосами просмотра.

Класс CFormView позволяет создать окно просмотра документа, основанное на диалоговой панели. От этого класса наследуется еще два класса CRecordView и CDaoRecordView. Эти классы используются для просмотра записей баз данных.

Диалоговые панели (класс CDialog и классы, наследованные от него)

От базового класса наследуются классы, управляющие диалоговыми панелями. Если необходимо создать диалоговую панель, можно наследовать класс от CDialog.

Вместе с диалоговыми панелями обычно используется класс CDataExchange. Класс CDataExchange обеспечивает работу процедур обмена данными DDX (Dialog Data Exchange) и проверки данных DDV (Dialog Data Validation), используемых для диалоговых панелей. В отличие от CDialog класс CDataExchange не наследуется от какого-либо другого класса.

От класса CDialog наследуется ряд классов, представляющих собой стандартные диалоговые панели для выбора шрифта, цвета, вывода документа на печать, поиска в документе определенной последовательности

символов, а также поиска и замены одной последовательности символов другой последовательностью.

Чтобы создать стандартный диалог, можно просто определить объект соответствующего класса. Дальнейшее управление такой панелью осуществляется методами класса.

Массивы, списки, словари

В состав MFC включен целый набор классов, предназначенных для хранения информации в массивах, списках и словарях. Все эти классы наследованы от базового класса CObject.

Несмотря на то, что в языке C определено понятие массива, классы MFC обеспечивают более широкие возможности. Например, можно динамически изменять размер массива, определенного с помощью соответствующего класса.

Для представления массивов предназначены следующие классы:

- CByteArray - байты.
- CDWordArray - двойные слова.
- CObArray - указатели на объекты класса CObject.
- CPtrArray - указатели типа void.
- CStringArray - объекты класса CString.
- CUIntArray - элементы класса unsigned integer или UINT.
- CWordArray - слова.

Для решения многих задач применяются такие структуры хранения данных, как списки. MFC включает ряд классов, наследованных от базового класса CObject, которые представляют программисту готовое для создания собственных списков. В этих классах определены все методы, необходимые при работе со списками, - добавление нового элемента, вставка нового элемента, определение следующего или предыдущего элемента в списке, удаление элемента и т.д.

Перечислим классы списков, которые позволяют построить списки из элементов любых типов любых классов:

CObList - указатели на объекты класса CObject.

CPtrList - указатели типа void.

CStringList - объекты класса CString.

В библиотеке MFC определена еще одна группа классов, позволяющая создавать словари. Словарь представляет собой таблицу из двух колонок, устанавливающих соответствие двух величин. Первая величина представляет ключевое значение и записывается в первую колонку, а вторая - связанное с ней значение, хранящееся во второй колонке. Словарь позволяет добавлять в

него пары связанных величин и осуществлять выборку значений по ключевому слову.

Для работы со словарями используются классы:

- CMapPtrToPtr - ключевое слово - указатель типа void, связанное с ним значение - указатель типа void.
- CMapPtrToWord - ключевое слово - указатель типа void, связанное с ним значение - слово.
- CMapStringToOb - ключевое слово - объекты класса CString, связанное с ним значение - указатель на объекты класса CObject.
- CMapStringToPtr - ключевое слово - объекты класса CString, связанное с ним значение - указатель типа void.
- CMapStringToString - ключевое слово - объекты класса CString, связанное с ним значение - на объекты класса CObject.
- CMapWordToOb - ключевое слово - слово, связанное с ним значение - указатель на объекты класса CObject.
- CMapWordToPtr- ключевое слово - слово, связанное с ним значение - указатель типа void.

Файловая система (класс CFile)

Библиотека MFC включает класс для работы с файловой системой компьютера. Он называется CFile и также наследуется от базового класса CObject. Непосредственно от класса CFile наследуется еще несколько классов - CMemFile, CStdioFile, CSocketFile.

При работе с файловой системой может потребоваться получить различную информацию о некотором файле, например, дату создания, размер и т.д. Для хранения этих данных предназначен специальный класс CFileStatus. Класс CFileStatus - один из немногих классов, которые не наследуются от базового класса CObject.

Контекст отображения (класс CDC)

Для отображения информации в окне или на любом другом устройстве приложение должно получать так называемый контекст отображения. Основные свойства контекста отображения определены в классе CDC. От него наследуется 4 различных класса, представляющие контекст отображения различных устройств.

Дадим краткое описание классов, наследованных от CDC:

CClientDC - контекст отображения, связанный с внутренней областью окна (client area). Для получения контекста конструктор класса вызывает функцию программного интерфейса GetDC, а деструктор - функцию ReleaseDC.

CMetaFileDC - класс предназначен для работы с метафайлами.

CPaintDC - конструктор класса CPaintDC для получения контекста отображения вызывает метод CWnd::BeginPaint, деструктор - метод CWnd::EndPaint. Объекты данного класса можно использовать только при обработке сообщения WM_PAINT. Это сообщение обычно обрабатывает метод OnPaint.

CWindowDC - контекст отображения, связанный со всем окном. Для получения контекста конструктор класса вызывает функцию программного интерфейса GetWindowDC, а деструктор - функцию ReleaseDC.

Объекты графического интерфейса (класс CGdiObject)

Для отображения информации используются различные объекты графического интерфейса - GDI-объекты. Для каждого из этих объектов библиотека MFC содержит описывающий его класс, наследованный от базового класса CGdiObject.

Для работы с GDI-объектами используются классы:

- CBitmap - растровое изображение bitmap.
- CBrush - кисть.
- CFont - шрифт.
- CPalette - палитра цветов.
- CPen - -перо.
- CRgn - область внутри окна.

Меню (класс CMenu)

Практически каждое приложение имеет собственное меню. Оно, как правило, отображается в верхней части главного окна приложения. Для управления меню в состав MFC включен специальный класс CMenu, наследованный непосредственно от базового класса CObject.

Для управления меню и панелями используется также класс CCmdUI. Этот класс не наследуется от базового класса CObject.

Объекты класса CCmdUI создаются, когда пользователь выбирает строку меню или нажимает кнопки панели управления. Методы класса CCmdUI позволяют управлять строками меню и кнопками панели управления. Например, существует метод, который делает строку меню неактивной.

Другие классы

В MFC включено несколько классов, обеспечивающих поддержку приложений, работающих с базами данных. Это такие классы, как CDataBase, CRecordSet, CDaoDataBase, CDaoRecordSet, CDaoQueryDef, CDaoTableDef, CDaoWorkSpace, CLongBinary, CFieldExchange и CDaoFieldExchange.

Библиотека MFC позволяет создавать многозадачные приложения. Для синхронизации отдельных задач приложения предусмотрен ряд специальных классов. Все они наследуются от класса CSyncObject, представляющий собой абстрактный класс.

В некоторых случаях требуется, чтобы участок программного кода мог выполняться только одной задачей. Такой участок называют критической секцией кода. Для создания и управления критическими секциями предназначены объекты класса CCriticalSection.

Объекты класса CEvent представляют событие. При помощи событий одна задача приложения может передать сообщение другой.

Объекты класса CMutex позволяют в данный момент предоставить ресурс в пользование одной только задаче. Остальным задачам доступ к ресурсу запрещается.

Объекты класса CSemaphore представляют собой семафоры. Семафоры позволяют ограничить количество задач, которые имеют доступ к какому-либо ресурсу.

Для программистов, занимающихся сетевыми коммуникациями, в состав библиотеки MFC включены классы CAsyncSocket и наследованный от него класс CSocket. Эти классы облегчают задачу программирования сетевых приложений.

Кроме уже описанных классов библиотека MFC включает большое количество классов, предназначенных для организации технологии OLE.

Классы, не имеющие базового класса

Кроме классов, наследованных от базового класса CObject, библиотека MFC включает ряд самостоятельных классов. У них нет общего базового класса, и имеют различное назначение.

Несколько классов, которые не наследуются от базового класса CObject, уже упоминались. К ним относятся классы CCmdUI, CFileStatus, CDataExchange, CFieldExchange и CDaoFieldExchange.

Простые классы. Библиотека MFC содержит классы, соответствующие объектам типа простых геометрических фигур, текстовых строк и объектам, определяющим дату и время:

- CPoint - объекты класса описывают точку.
- CRect - объекты класса описывают прямоугольник.
- CSize - объекты класса определяют размер прямоугольника.
- CString - объекты класса представляют собой текстовые строки переменной длины.

- CTime - объекты класса служат для хранения даты и времени. Большое количество методов класса позволяет выполнять над объектами класса различные преобразования.

- CTimeSpan - объекты класса определяют период времени.

Архивный класс (класс CArchive). Класс CArchive используется для сохранения и восстановления состояния объектов в файлах на диске. Перед использованием объекта класса CArchive он должен быть привязан к файлу - объекта класса CFile.

Информация о классе объекта (структура CRuntimeClass). Во многих случаях бывает необходимо уже во время работы приложения получать информацию о классе и его базовом классе. Для этого любой класс, наследованный от базового класса CObject, связан со структурой CRuntimeClass. Она позволяет определить имя класса объекта, размер объекта в байтах, указатель на конструктор класса, не имеющий аргументов, и деструктор класса. Можно также узнать подобную информацию о базовом классе и некоторые дополнительные сведения.

Отладка приложения (классы CDumpObject, CMemoryState). В отладочной версии приложения можно использовать класс CDumpContext. Он позволяет выдавать состояние различных объектов в текстовом виде.

Класс CMemoryState позволяет локализовать проблемы, связанные с динамическим выделением оперативной памяти. Такие проблемы обычно возникают, когда пользователь выделяет память, применяя оператор new, а затем забывает вернуть эту память операционной системе.

Печать документа (класс CPrintInfo). Класс CPrintInfo предназначен для управления печатью документов на принтере. Когда пользователь отправляет документ на печать или выполняет предварительный просмотр документа перед печатью, создается объект класса CPrintInfo. Он содержит различную информацию о том, какие страницы документа печатаются, и т.д.

Каркас MFC-программы

В простейшем случае программа, написанная с помощью MFC, содержит два класса, порожденные от классов иерархии библиотеки: класс,

предназначенный для создания приложения, и класс, предназначенный для создания окна. Другими словами, для создания минимальной программы необходимо породить один класс от CWinApp, а другой - от CFrameWnd. Эти два класса обязательны для любой программы.

Класс приложение

В классе приложения необходимо:

- Переопределить виртуальную функцию InitInstance
- Создать внутри нее объект оконного класса

Например, стандартный шаблон класса приложение может выглядеть так :

```
#include <afxwin.h>
class CMyApp: public CWinApp
{ public:
    virtual BOOL InitInstance()
    {
        m_pMainWnd = new CMyWindow;
        m_pMainWnd -> ShowWindow(SW_SHOWNORMAL);
        return TRUE;
    }
};
CMyApp app;
virtual BOOL CWinApp::InitInstance();
```

Это виртуальная функция, которая вызывается каждый раз при запуске программы. В ней должны производиться все действия, связанные с инициализацией приложения. Функция должна возвращать TRUE при успешном завершении и FALSE в противном случае. В нашем случае, в функции сначала создается объект класса CMyApp, и указатель на него запоминается в переменной m_pMainWnd. Эта переменная является членом класса CWinThread. Она имеет тип CWnd* и используется почти во всех MFC-программах, потому что содержит указатель на главное окно. В последующих двух строчках через нее вызываются функции-члены окна. Когда окно создано, вызывается функция с прототипом:

```
BOOL CWnd::ShowWindow(int How);
```

Параметр определяет, каким образом окно будет показано на экране.

Строка CMyApp app – это объявление глобального app класса приложение. Каждое приложение, созданное с помощью MFC, должно содержать только один объект класса, производный от CWinApp. В конструкторе базового класса CWinApp, который вызывается при создании объекта app запускается функция WinMain, которая, в свою очередь, запускает цикл ожидания и обработки сообщений, условно называемый MessagePump.

Оконный класс

Оконный класс удобно наследовать от класса CFrameWnd. Самая простая реализация оконного класса, в которой создается пустое окно с атрибутами по умолчанию, но “своим” заголовком будет:

```
class CMyWindow : public CFrameWnd{
public:
    CMyWindow() // конструктор
    {
        Create(NULL, "Простой диалог"); // окно-рамка по умолчанию.
    }
};
```

Этот код необходимо вставить до определения класса CMyApp. Объект класса, производного от CFrameWnd, создается как окно-рамка. Термин “Окно-рамка” используется для обозначения тех элементов интерфейса окна, которые позволяют управлять его обликом: обрамление, заголовок, системное меню, кнопки минимизации и максимизации, восстановления окна. Вся остальная область окна называется клиентской областью. Именно в этой области располагаются элементы: меню, строки статуса, панель инструментов и другие отображаемые элементы. Существует три способа создания окна-рамки: явное конструирование путём вызова метода Create(как в нашем примере), загрузка окна, атрибуты которого заданы в файле ресурсов путём вызова метода LoadFrame, неявное конструирование на основе шаблона-документа.

Для простейшей программы необходимо создать два объекта MFC:

1. объект класса CWinApp;
2. объект класса CWnd.

Вспомним, что при разработке простейшего приложения с использованием функции API, мы создавали две функции: WinMain и оконную процедуру WndProc. Существует аналогия между этой структурой и структурой простейшего приложения на основе MFC. Аналогом WinMain является класс приложения CWinApp, а аналогом оконной процедуры – класс окна CWnd. Мы должны создать свой класс, производный от CWinApp, например, CFirstApp.

CObject→CCmdTarget→CWinThread→CWinApp

Всю работу, которую выполняет функция WinMain, выполняет теперь класс CWinApp. Нам остаётся только:

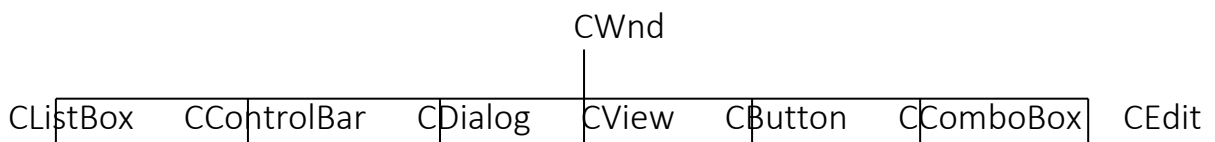
- переопределить в классе CFirstApp виртуальную функцию InitInstance;

- создать внутри этой функции объект CWnd – аналог оконной процедуры.

CObject→CCommandTarget→CWnd→CFrameWnd

На самом деле функция WinMain ОС не вызывается. Вместо этого происходит обращение к стартовой функции из библиотеки времени выполнения C/C++. Компилятор VC++ знает, что она называется _WinMainCRTStartup и отвечает за выполнение таких операций, как:

- Поиск указателя на полную командную строку нового процесса.
- Поиск указателя на переменные окружения нового процесса.
- Инициализация глобальных переменных из библиотеки времени выполнения языка C, доступ к которым обеспечивается включением файла STDLIB.H.
- Инициализация блока памяти (heap), используемого функциями выделения памяти и процедурами низкоуровневого ввода/вывода языка C.
- Вызов функции WinMain библиотеки MFC.



Рассмотрим некоторые функции – члены класса CWinApp.

Конструктор CWinApp (LPCTSTR lpszAppName = NULL);

Параметр lpszAppName – указатель на строку, которая содержит имя приложения, используемого системой Windows. Если этот аргумент отсутствует или равен NULL, то для формирования имени приложения конструктор использует строку из файла ресурсов с идентификатором AFX_IDS_APP_TITLE, а при её отсутствии – имя исполняемого файла.

Функция virtual BOOL InitInstance();

Эта функция выполняет инициализацию каждого нового экземпляра приложения. В классе CWinApp эта функция ничего не делает и Вы должны переопределить её в производном классе. В этой функции создается объект “главное окно” приложения, загружаются стандартные настройки из INI-файла или из реестра Windows, создается новый или открывается существующий документ, регистрируются шаблоны документов, создаются сами документы, их представления (вид) и ассоциированные с ними окна.

virtual BOOL ExitInstance();

Функция вызывается только из функции Run для завершения работы приложения.

virtual int Run();

Функция запускает цикл обработки сообщений.

CFrameWnd – класс «окно - рамка».

Класс CFrameWnd служит для создания перекрывающихся или всплывающих окон и поддерживает однодокументный интерфейс Windows (SDI). Объект этого класса координирует взаимодействие приложения с документами и его представлением. Он отражается на экране в виде тех элементов интерфейса, которые позволяют управлять обликом окна: обрамление, строка заголовка, системное меню, кнопки минимизации, максимизации, восстановления. Это окно отвечает за управление размещением своих дочерних окон и других элементов рабочей (клиентской) области. Кроме того, это окно переадресует команды своими представлениями и может отвечать на извещения от элементов управления окна.

Существует три способа создания объекта CFrameWnd:

1. Явное конструирование путем вызова метода Create;
2. Загрузка окна, атрибуты которого заданы в файле ресурсов (rc - файле) путем вызова метода LoadFrame;
3. Неявное конструирование на основе шаблона документа.

Мы воспользовались первым способом и задали нуль в качестве первого аргумента функции Create. Это означает, что используется стиль оконного класса, заданный по умолчанию. Если вы хотите повлиять на процесс регистрации оконного класса, а тем самым и на стиль окна, то перед созданием окна следует вызвать функцию AfxRegisterWndClass и передать ей в качестве параметра нужные атрибуты окна.

Пример

//simpwin.h

```
#include <afxwin.h>
// Класс основного окна приложения
class CMainWin: public CFrameWnd {
public:
    CMainWin();
    // Декларирование карты сообщений
    DECLARE_MESSAGE_MAP()
};
// Класс приложения. Должен существовать только
// один экземпляр этого класса.
// Член-функция InitInstance() вызывается при запуске
// приложения.
class CApp: public CWinApp {
public:
    BOOL InitInstance();
};
```

```

/* simpwin.cpp*/

#include <afxwin.h>
#include <string.h>
#include "SIMPWIN.H"
// Создание одного и только одного экземпляра
// приложения
CApp App;
// Реализация
BOOL CApp::InitInstance()
{
    // Создание главного окна приложения и его
    // отображение.
    // Член CApp::m_pMainWnd - это указатель на объект
    // главного окна.
    m_pMainWnd = new CMainWin;
    m_pMainWnd->ShowWindow(SW_RESTORE);
    m_pMainWnd->UpdateWindow();
    // Сигнализируем MFC об успешной инициализации
    // приложения.
    return TRUE;
}

CMainWin::CMainWin()
{
    // Создание окна с заголовком. Используется
    // встроенный в MFC
    // класс окна, поэтому первый параметр 0.
    this->Create(0, L"Простейшее приложение на MFC");
}
// Реализация карты сообщений
BEGIN_MESSAGE_MAP(CMainWin /*класс окна*/, CFrameWnd
/*класс-предок*/)
    END_MESSAGE_MAP()

```

В примере была использована функция Create(), которая на самом деле имеет много параметров. Ее прототип таков:

```

BOOL CFrameWnd::Create(
    LPCSTR ClassName,      // Имя Windows-класса окна
    LPCSTR Title,          // Заголовок
    DWORD Style = WS_OVERLAPPEDWINDOW, // Стил
    const RECT &XYSIZE = rectDefault, // Область
    CWnd *Parent = 0,      //Окно-предок
    LPCSTR MenuName = 0,    //Имя ресурса меню
    DWORD ExStyle = 0,      //Расширенные стили
    CCreateContext *Context = 0 // Доп. данные
);

```

Первый параметр, `ClassName`, определяет имя класса окна для оконной подсистемы Windows. Обычно его не нужно явно задавать, так как MFC выполняет всю необходимую черновую работу. В данных методических указаниях мы не будем использовать своих классов окон. Параметр `Style` задает стиль окна. По умолчанию создается стандартное перекрываемое окно. Можно задать свой стиль, объединив с помощью операции "или" несколько констант.

Приложение без главного окна

```
//Файл first.cpp
#include <afxwin.h> // Включаемый файл для MFC

// Класс CFirstApp - главный класс приложения.
// Наследуется от базового класса CWinApp.
class CFirstApp:public CWinApp
{
public:
    // Переопределение метода InitInstance,
    // предназначенного для инициализации приложения.
    virtual BOOL InitInstance();
};

// Создание объекта приложения класса CFirstApp.
CFirstApp theApp;

// Метод InitInstance
// Переопределение виртуального метода InitInstance класса CWinApp.
// Он вызывается каждый раз при запуске приложения.
BOOL CFirstApp::InitInstance()
{
    AfxMessageBox("First MFC-application");
    return FALSE;
}
```

Приложение с главным окном

```
//Файл start.h
#include <afxwin.h>
class CStartApp: public CWinApp
{
public:
    virtual BOOL InitInstance();
};
```

```

//Файл startm.h
#include <afxwin.h>
// Класс CMainWindow - представляет главное окно приложения.
class CMainWindow : public CFrameWnd
{
public:
    CMainWindow();
};
//Файл start.cpp
#include <afxwin.h>
#include "start.h"
#include "startm.h"
BOOL CStartApp::InitInstance()
{
    // Создание объекта класса CMainWindow
    m_pMainWnd= new CMainWindow();
    // Отображение окна на экране.
    // Параметр m_nCmdShow определяет режим отображения окна.
    m_pMainWnd->ShowWindow(m_nCmdShow);
    // Обновление содержимого окна.
    m_pMainWnd->UpdateWindow();
    return TRUE;
}
CStartApp theApp;
//Файл startm.cpp
#include <afxwin.h>
#include "startm.h"
// Конструктор класса CMainWindow
CMainWindow::CMainWindow()
{
    // Создание окна приложения
    Create(NULL,"Hello");
}

```

Обработка окном сообщений

Работа операционной системы Windows основана на обработке сообщений. Когда пользователь работает с устройствами ввода/вывода (например, клавиатурой или мышью), драйверы этих устройств создают сообщения, описывающие его действия.

Сообщения сначала попадают в системную очередь сообщений операционной системы. Из нее сообщения передаются приложениям, которым они предназначены, и записываются в очередь приложений. Каждое приложение имеет собственную очередь сообщений.

Приложение в цикле, который называется циклом обработки сообщений, получает сообщения из очереди приложения и направляет их соответствующей функции окна, которая и выполняет обработку сообщения. Цикл обработки сообщений в традиционной Windows-программе обычно состоял из оператора `while`, в котором циклически вызывались функции `GetMessage` и `DispatchMessage`. Для более сложных приложений цикл обработки сообщений содержал вызовы других функций (`TranslateMessage`, `TranslateAccelerator`). Они обеспечивали предварительную обработку сообщений.

Каждое окно приложения имеет собственную функцию окна. В процессе обработки сообщения операционная система вызывает функцию окна и передает ей структуру, описывающую очередное сообщение. Функция обработки сообщения проверяет, какое именно сообщение поступило для обработки, и выполняет соответствующие действия.

Если при создании приложения используется библиотека классов MFC, то за обработку сообщений отвечают классы. Любой класс, наследованный от базового класса `CCmdTarget`, может обрабатывать сообщения. Чтобы класс смог обрабатывать сообщения, необходимо, чтобы он имел таблицу сообщений класса. В этой таблице для каждого сообщения указан метод класса, предназначенный для его обработки.

Группы сообщений

Сообщения, которые могут обрабатываться приложением, построенным с использованием библиотеки классов MFC, делятся на 3 группы.

Оконные сообщения

Эта группа включает сообщения, предназначенные для обработки функцией окна. Практически все сообщения, идентификаторы которых начинаются префиксом `WM_`, за исключением `WM_COMMAND`, относятся к этой группе.

Оконные сообщения предназначены для обработки объектами, представляющими окна. Это могут быть практически любые объекты класса `CWnd` или классов, наследованных от него (`CFrameWnd`, `CMDIFrameWnd`, `CMDIChildWnd`, `CView`, `CDialog`). Характерной чертой этих классов является то, что они включают идентификатор окна.

Большинство этих сообщений имеют параметры, детально характеризующие сообщение.

Сообщения от органов управления

Эта группа включает в себя сообщения `WM_COMMAND` от дочерних окон (включая окна стандартных классов), передаваемых их родительскому окну.

Сообщения от органов управления обрабатываются точно таким же образом, что и оконные сообщения.

Исключение составляет сообщение WM_COMMAND с кодом извещения BN_CLICKED. Это сообщение передается кнопкой, когда пользователь на нее нажимает. Обработка сообщений с кодом извещения BN_CLICKED от органов управления происходит аналогично обработке командных сообщений.

Командные сообщения

Это сообщения WM_COMMAND от меню, кнопок панели управления и клавиш акселераторов. В отличие от оконных сообщений и сообщений от органов управления командные сообщения могут быть обработаны более широким спектром объектов. Эти сообщения обрабатывают не только объекты, представляющие окна, но также объекты классов, представляющих приложение, документы или шаблон документов.

Характерной особенностью командных сообщений является идентификатор. Идентификатор командного сообщения определяет объект, который вырабатывает (посылает) данное сообщение.

Таблица сообщений

В библиотеке классов MFC для обработки сообщений используется специальный механизм, который имеет название Message Map - таблица сообщений.

Таблица сообщений состоит из набора специальных макрокоманд, ограниченных макрокомандами BEGIN_MESSAGE_MAP и END_MESSAGE_MAP. Между ними расположены макрокоманды, отвечающие за обработку отдельных сообщений:

```
BEGIN_MESSAGE_MAP(ИмяКласса,ИмяБазовогоКласса)  
    // макросы  
END_MESSAGE_MAP()
```

Макрокоманда BEGIN_MESSAGE_MAP представляет собой заголовок таблицы сообщений. Она имеет два параметра. Первый параметр содержит имя класса таблицы сообщений. Второй - указывает его базовый класс.

Если в таблице сообщений класса отсутствует обработчик для сообщения, оно передается для обработки базовому классу, указанному вторым параметром макрокоманды BEGIN_MESSAGE_MAP. Если таблица сообщений базового класса также не содержит обработчик этого сообщения, оно передается следующему базовому классу и т.д. Если ни один из базовых

классов не может обработать сообщение, выполняется обработка по умолчанию, зависящая от типа сообщения:

- стандартные сообщения Windows обрабатываются функцией обработки по умолчанию;
- командные сообщения передаются по цепочке следующему объекту, который может обработать командное сообщение.

Можно определить таблицу сообщений класса вручную, однако более удобно воспользоваться для этой цели средствами ClassWizard. ClassWizard не только позволяет в удобной форме выбрать сообщения, которые должен обрабатывать класс. Он включает в состав класса соответствующие методы-обработчики. Программисту останется только вставить в них необходимый код. К сожалению, использовать все возможности ClassWizard можно только в том случае, если приложение создано с применением средств автоматизированного программирования MFC AppWizard.

Рассмотрим макрокоманды, отвечающие за обработку различных типов сообщений.

Макрокоманда `ON_WM_<name>`. Обрабатывает стандартные сообщения операционной системы Windows. Вместо `<name>` указывается имя сообщения без префикса `WM_`. Например:

```
ON_WM_CREATE()
```

Для обработки сообщений, определенных в таблице макрокомандой `On_WM_<name>`, вызываются одноименные методы. Имя метода обработчика соответствует названию сообщения, без учета префикса `WM_`. В классе `CWnd` определены обработчики для стандартных сообщений. Эти обработчики будут использоваться по умолчанию.

Макрокоманды `ON_WM_<name>` не имеют параметров. Однако методы, которые вызываются для обработки соответствующих сообщений, имеют параметры, количество и назначение которых зависит от обрабатываемого сообщения.

Если определить обработчик стандартного сообщения Window в своем классе, то он будет вызываться вместо обработчика, определенного в классе `CWnd` (или другом базовом классе). В любом случае можно вызвать метод-обработчик базового класса из своего метода-обработчика.

Макрокоманда `ON_REGISTERED_MESSAGE`. Эта макрокоманда обслуживает сообщения операционной системы Windows, зарегистрированные с помощью функции `RegisterWindowMessage`. Параметра `nMessageVariable` этой макрокоманды указывает идентификатор сообщения, для которого будет вызываться метод `memberFxn`:

```
ON_REGISTERED_MESSAGE(nMessageVariable, memberFxn)
```

Макрокоманда ON_MESSAGE. Данная макрокоманда обрабатывает сообщения, определенные пользователем. Идентификатор сообщения (его имя) указывается параметром message. Метод, который вызывается для обработки сообщения, указывается параметром memberFxn:

ON_MESSAGE(message,memberFxn)

Макрокоманда ON_COMMAND. Эти макрокоманды предназначены для обработки командных сообщений. Командные сообщения поступают от меню, кнопок панели управления и клавиш акселераторов. Характерной особенностью командных сообщений является то, что с ними связан идентификатор сообщения.

Макрокоманда ON_COMMAND имеет два параметра. Первый параметр соответствует идентификатору командного сообщения, а второй - имени метода, предназначенного для обработки этого сообщения. Таблица сообщений должна содержать не больше одной макрокоманды для командного сообщения:

ON_COMMAND(id,memberFxn)

Обычно командные сообщения не имеют обработчиков, используемых по умолчанию. Существует только небольшая группа стандартных командных сообщений, имеющих методы-обработчики, вызываемые по умолчанию. Эти сообщения соответствуют стандартным строкам меню приложения. Так например, если строке Open меню File присвоить идентификатор ID_FILE_OPEN, то для его обработки будет вызван метод OnFileOpen, определенный в классе CWinApp.

Макрокоманда ON_COMMAND_RANGE. Макрокоманда ON_COMMAND ставит в соответствие одному командному сообщению один метод-обработчик. В некоторых случаях более удобно, когда один и тот же метод-обработчик применяется для обработки сразу нескольких командных сообщений с различными идентификаторами. Для этого предназначена макрокоманда ON_COMMAND_RANGE.

Она назначает один метод-обработчик для обработки нескольких командных сообщений, интервалы которых лежат в интервале от id1 до id2:

ON_COMMAND_RANGE(id1,id2,memberFxn)

Макрокоманда ON_UPDATE_COMMAND_UI. Данная макрокоманда обрабатывает сообщения, предназначенные обновления пользовательского

интерфейса, например меню, панелей управления, и позволяет менять их состояние.

Параметр `id` указывает идентификатор сообщения, а параметр `memberFxn` - имя метода для его обработки:

```
ON_UPDATE_COMMAND_UI(id,memberFxn)
```

Методы, предназначенные для обработки данного класса сообщений, должны быть определены с ключевым словом `afx_msg` и иметь один параметр - указатель на объект класса `CCmdUI`. Для удобства имена методов, предназначенных для обновления пользовательского интерфейса, начинаются с `OnUpdate`:

```
afx_msg void OnUpdate<имя_обработчика>(CCmdUI* pCmdUI);
```

В качестве параметра `pCmdUI` методу передается указатель на объект класса `CCmdUI`. В нем содержится информация об объекте пользовательского интерфейса, который нужно обновить, - строке меню или кнопке панели управления. Класс `CCmdUI` также включает методы, позволяющие изменить внешний вид представленного им объекта пользовательского интерфейса.

Сообщения, предназначенные для обновления пользовательского интерфейса, передаются, когда пользователь открывает меню приложения, а также во время цикла ожидания приложения, когда очередь сообщений приложения становится пустой.

При этом посылаются несколько сообщений, по одному для каждой строке меню. С помощью макрокоманд `ON_UPDATE_COMMAND_UI` можно определить методы-обработчики, ответственные за обновление внешнего вида каждой строки меню и соответствующие ей кнопки на панели управления. Эти методы могут изменять состояние строки меню - отображать ее серым цветом, запрещать ее выбор, отображать около нее символ "галочка" и т.д.

Если не определить метод для обновления данной строки меню или кнопки панели управления, то выполняется обработка по умолчанию. Выполняется поиск обработчика соответствующего командного сообщения, и, если он не обнаружен, выбор строки запрещается.

Макрокоманда `ON_UPDATE_COMMAND_UI_RANGE`. Эта макрокоманда обеспечивает обработку сообщений, предназначенных для обновления пользовательского интерфейса, идентификаторы которых лежат в интервале от `id1` до `id2`. Параметр `memberFxn` указывает метод, используемый для обработки:

```
ON_UPDATE_COMMAND_UI_RANGE(id1,id2,memberFxn)
```

Макрокоманда ON_<name>. Данные макрокоманды предназначены для обработки сообщений от органов управления. Такие сообщения могут передаваться органами управления диалоговой панели. Сообщения от органов управления не имеют обработчиков, используемых по умолчанию. При необходимости их нужно определить самостоятельно.

Все макрокоманды ON_<name> имеют два параметра. В первом параметре id указывается идентификатор органа управления. Сообщения от этого органа управления будут обрабатываться методом memberFxn. Например:

```
ON_BN_CLICKED(id,memberFxn)
```

Макрокоманда ON_CONTROL_RANGE. Эта макрокоманда обрабатывает сообщения от органов управления, идентификаторы которых находятся в интервале от id1 до id2. Параметр wNotifyCode содержит код извещения. Метод-обработчик указывается параметром memberFxn:

```
ON_CONTROL_RANGE(wNotifyCode,id1,id2,memberFxn)
```

Приложение, обрабатывающее сообщения

Предыдущие два рассматриваемых приложения, фактически никак не могли взаимодействовать с пользователем. Они не имели ни меню, ни панели управления. И, самое главное, они не содержали обработчиков сообщений.

Рассмотрим теперь приложение, которое имеет меню и содержит обработчики сообщений, передаваемых приложению, когда пользователь открывает меню и выбирает из него строки. Пусть меню приложения состоит из одного пункта Test. Можно выбрать одну из следующих команд - Веер или Exit.

Файл ресурсов, в который включается описание меню, можно построить либо непосредственным созданием нового файла ресурсов, либо при помощи средств AppWizard. В любом случае при создании меню нужно определить название меню или строки меню. Каждый элемент меню должен иметь уникальный идентификатор, однозначно его определяющий:

```
//Файл resource.h
#define IDR_MENU          101
#define ID_TEST_BEEP      40001
#define ID_TEST_EXIT      40002
//Файл resource.rc
```

```

#include "resource.h"
IDR_MENU MENU DISCARDABLE
BEGIN
    POPUP "Test"
    BEGIN
        MENUITEM "Beep",    ID_TEST_BEEP
        MENUITEM SEPARATOR
        MENUITEM "Exit",    ID_TEST_EXIT
    END
END
END

```

Файлы, в которых находятся определение классов приложения и главного окна, представлены ниже:

```

//Файл menu.h
#include <afxwin.h>
class CMenuApp: public CWinApp
{
public:
    virtual BOOL InitInstance();
};

//Файл menu.cpp
#include <afxwin.h>
#include "menu.h"
#include "menu.h"
BOOL CMenuApp::InitInstance()
{
    m_pMainWnd= new CMainWindow();
    m_pMainWnd->ShowWindow(m_nCmdShow);
    m_pMainWnd->UpdateWindow();
    return TRUE;
}

CMenuApp theApp;

//Файл menu.h
#include <afxwin.h>
class CMainWindow : public CFrameWnd
{
public:
    CMainWindow();
    afx_msg void TestBeep();
    afx_msg void TestExit();
    // макрокоманда необходима, так как класс обрабатывает сообщения
    DECLARE_MESSAGE_MAP()
};

//Файл menu.cpp
#include <afxwin.h>

```

```

#include "menu.h"
#include "resource.h"
// Таблица сообщений класса
BEGIN_MESSAGE_MAP(CMainWindow,CFrameWnd)
    ON_COMMAND(ID_TEST_BEEP,TestBeep)
    ON_COMMAND(ID_TEST_EXIT,TestExit)
END_MESSAGE_MAP()
CMainWindow::CMainWindow()
{
    Create(NULL,"Hello",WS_OVERLAPPEDWINDOW,rectDefault,
        NULL,MAKEINTRESOURCE(IDR_MENU));
}
void CMainWindow::TestBeep()// Метод TestBeep - обрабатывает команду меню
{
    MessageBeep(0);
}
void CMainWindow::TestExit()// Метод TestExit - обрабатывает команду меню
{
    DestroyWindow();
}

```

Чтобы объекты класса могли обрабатывать сообщения, в определении класса необходимо поместить макрокоманду DECLARE_MESSAGE_MAP. По принятым соглашениям эта макрокоманда должна записываться в секцию public.

Кроме этого, необходимо также определить таблицу сообщений. Таблица начинается макрокомандой BEGIN_MESSAGE_MAP и заканчивается макрокомандой END_MESSAGE_MAP. Между этими макрокомандами расположены строки таблицы сообщений, определяющие сообщения, подлежащие обработке данным классом, и методы, которые выполняют такую обработку.

Приложение может содержать несколько классов, обладающих собственными таблицами сообщений. Чтобы однозначно определить класс, к которому относится таблица сообщений, имя этого класса записывается в первый параметр макрокоманды BEGIN_MESSAGE_MAP.

Приложение menu обрабатывает только две команды от меню приложения. Для обработки этих команд используют методы, представленные в определении класса CMainFrame.

Приложению может поступать гораздо больше сообщений и команд, чем указано в таблице сообщений класса CMainFrame. Необработанные сообщения передаются для обработки базовому классу - классу CFrameWnd. Класс, который будет обрабатывать сообщения, не указанные в таблице сообщений, указывается во втором параметре макрокоманды BEGIN_MESSAGE_MAP.

Замечание. Если класс приложения тоже обрабатывает сообщения (т.е. имеет таблицу сообщений), и некоторые из сообщений обрабатываются как окном, так и приложением, то нужно понять, какова очередность обработки сообщений тем или иным объектом. Те команды, которые не имеют обработчика в таблице сообщений класса окна, передаются для обработки в класс приложения. Если же команда может быть обработана и в классе окна, и в классе приложения, она обрабатывается только один раз в классе окна. Обработчик класса приложения в этом случае не вызывается.

Некоторые методы класса CWnd

Создание и уничтожение Windows-окон

CWnd();

Создает объект класса CWnd, обеспечивающий доступ к Windows-окну. При этом само Windows-окно не создается. Далее можно либо создать новое Windows-окно и закрепить его за данным оконным объектом, либо закрепить уже имеющееся Windows-окно. Первое достигается методами CreateEx и Create, второе - методом Attach.

virtual BOOL DestroyWindow();

Уничтожает Windows-окно, закрепленное за объектом класса CWnd. Если окно уничтожено успешно, возвращается ненулевое значение, в противном случае - 0. После выполнения этого метода оконный объект уже не имеет закрепленного за ним Windows-окна. Сам оконный объект при этом не уничтожается. Метод DestroyWindow посылает соответствующие сообщения, чтобы уничтожить окно и связанные с ним ресурсы. Оно также уничтожает дочерние окна и, если требуется, информирует родительское окно.

Методы инициализации

Методы создания Windows-окон обсуждаться не будут, так как при работе со многими классами окна (к которым относятся и элементы управления) создаются каркасом приложения. Рассмотрим только методы для прикрепления и открепления Windows-окон.

BOOL Attach(HWND hWndNew);

Закрепляет Windows-окно, заданное определителем hWndNew за оконным объектом. При успешном выполнении возвращает ненулевое значение, в противном случае - 0.

HWND Detach();

Открепляет Windows-окно, закрепленное за данным оконным объектом, и возвращает определитель открепленного окна.

Методы управления состоянием окна

Перечислим некоторые методы управления состоянием окна.

BOOL IsWindowEnabled() const;

Если окно в выключенном состоянии, возвращает нулевое значение, в противном случае - 0.

BOOL EnableWindow(BOOL bEnable=TRUE);

Если значение параметра TRUE, окно переводится в включенное состояние, FALSE - в выключенное. Метод возвращает ненулевое значение, если в момент вызова окно находилось в выключенном состоянии. Если окно было во включенном состоянии или произошла ошибка, возвращается 0.

CWnd* SetActiveWindow();

Переводит окно в активное состояние. Возвращает указатель на оконный объект, обеспечивающий доступ к окну, активному в момент вызова этого метода (указатель может быть временным и не должен запоминаться для дальнейшего использования).

static CWnd* PASCAL GetActiveWindow();

Возвращает указатель на оконный объект, обеспечивающий доступ к окну, активному в момент вызова этого метода. Если в момент вызова активных окон нет, возвращается NULL. Этот указатель может быть временным и не должен запоминаться для дальнейшего использования.

CWnd* SetCapture();

Переводит окно в состояние захвата мыши. Возвращает указатель на оконный объект, обеспечивающий доступ к окну, которым мышь была захвачена в момент вызова этого метода. Если в момент вызова мышь не захвачена, возвращает NULL. Этот указатель может быть временным и не должен запоминаться для дальнейшего использования. Чтобы освободить

мышь, используется API-функция ReleaseCapture (параметров не имеет). При успешном ее выполнении возвращается TRUE, иначе - FALSE.

`static CWnd* PASCAL GetCapture();`

Возвращает указатель, задающий окно, захватившее мышь. Если такого окна нет, возвращает NULL.

`BOOL ModifyStyle(DWORD dwRemove, DWORD dwAdd, UINT nFlags=0);`

Изменяет стиль окна. Параметр dwRemove задает набор элементов стиля, которые должны быть изъяты из стиля окна. Параметр dwAdd - набор элементов стиля, которые должны быть добавлены к стилю окна. Возвращает ненулевое значение, если стиль был успешно изменен, в противном случае - 0.

Если параметр nFlags не равен 0, то после изменения стиля вызывается API-функция SetWindowPos, которая перерисовывает окно, используя набор флагов, полученный комбинацией значения:

- SWP_NOSIZE - сохранять текущий размер;
- SWP_NOMOVE - сохранять текущую позицию;
- SWP_NOZORDER - сохранять текущий Z-порядок;
- SWP_NOACTIVE - не делать окно активным.

Методы управления размером и положением окна

`void MoveWindow(int x, int y, int nWidth, int nHeight, BOOL bRepaint=TRUE);`

`void MoveWindow(LPCRECT lpRect, BOOL bRepaint=TRUE);`

Эти методы изменяют положение и размеры окна. Положение левого верхнего угла окна задантс координатами x,y, а размеры шириной nWidth и высотой nHeight. Параметр bRepaint определяет, будет ли инициироваться перерисовка. Если он равен TRUE, окну будет послано сообщение WM_PAINT, в противном случае сообщение не посылается, и перерисовка не производится. Эти действия применяются как к клиентской, так и неклиентской области окна, а также к частям родительского окна, открывшимся при перемещении.

Новое положение и размеры окна можно задать и с помощью структуры типа RECT или объекта класса CRect, передав в качестве параметра ссылку на структуру или объект класса.

Для окна, у которого нет родителя, координаты указываются относительно левого верхнего угла экрана, а для имеющего такового - относительно верхнего левого угла родительского окна.

`BOOL SetWindowPos(const CWnd* pWndInsertAfter, int x, int y, int cx, int cy, UINT nFlags);`

Изменяет положение, размеры и место окна в Z-упорядочении (порядке изображения окон в слоях изображения). Параметры x, y, cx, cy задают новое положение левой стороны, новое положение верхней стороны, новую длину и новую высоту соответственно.

Параметр pWndInsertAfter определяет окно, за которым нужно поместить исходное окно в Z-упорядочении. Этот параметр может быть либо указателем на объект класса CWnd, либо одним из следующих значений:

- `wndBottom` поместить окно в конец Z-упорядочения, т.е. позади всех окон на экране;
- `wndTop` поместить окно в начало Z-упорядочения, т.е. впереди всех окон на экране;
- `wndTopMost` поместить окно на ближайшее место, делающее окно неперекрытым. Окно перемещается на неперекрытое место, даже если оно неактивно;
- `wndNoTopMost` поместить окно на ближайшее место позади всех неперекрытых окон. Окно, перекрытое в момент вызова этого метода, не перемещается.

Параметр nFlags задает режим изменения размера и положения окна и может быть следующей комбинацией флагов:

- `SWP_DRAWFRAME` изображать вокруг окна рамку (определенную при его создании);
- `SWP_HIDEWINDOW` скрыть окно;
- `SWP_NOACTIVE` не делать окно активным. Если этот флаг не установлен, окно делается активным и помещается впереди, либо на место в Z-упорядочении, определяемое параметром pWndInsertAfter;
- `SWP_NOMOVE` сохранить текущее положение окна, проигнорировав параметры x и y;
- `SWP_NOREDRAW` не перерисовывать измененное окно. Если этот флаг установлен, то после выполнения функции окно с новыми установками на экране не появится, а старое изображение окна не будет стерто с родительского окна;
- `SWP_NOSIZE` сохранить текущий размер окна, проигнорировав параметры cx и cy;
- `SWP_NOZORDER` сохранить текущее Z-упорядочение, проигнорировав параметр pWndInsertAfter;
- `SWP_SHOWWINDOW` показать окно (сделать окно видимым и перерисовать).

Если окно не является дочерним, координаты указываются относительно левого верхнего угла экрана, иначе координаты указываются относительно верхнего левого угла клиентской области родительского окна. Приложение не может активизировать неактивное окно, не поместив его в начало Z-упорядочения. Приложение не может изменить место активного окна в Z-упорядочении произвольным образом. Перекрытые окна могут быть родительскими по отношению к неперекрытым, но не наоборот.

`void GetWindowRect(LPRECT lpRect) const;`

Копирует параметры прямоугольника, ограничивающего окно, в структуру типа RECT или объект класса CRect, заданные параметром lpRect. Этот прямоугольник включает все - и клиентскую и системную часть окна. Параметры даются относительно левого верхнего угла экрана. При вызове метода значением фактического параметра может быть либо ссылка на (не константную) структуру типа RECT либо объект класса CRect.

`void GetClientRect(LPRECT lpRect) const;`

Копирует параметры прямоугольника, ограничивающего клиентскую часть окна, в структуру типа RECT или объект класса CRect, определенные параметром lpRect. Параметры даются относительно левого верхнего угла клиентской области окна, поэтому левая и верхняя составляющие будут равны нулю, а правая и нижняя - ширине и длине клиентской области. При вызове метода параметр lpRect может быть либо указателем на структуру типа RECT, либо переменной класса CRect.

Методы взаимодействия Windows-окон

`BOOL UpdateData(BOOL bSaveAndValidate=TRUE);`

Выполняет обмен данными между объектом класса, производного от CWnd, и частным случаем Windows-окна - диалоговым окном. Если параметр равен FALSE, данные будут пересылаться от объекта и обновлять содержимое элементов управления диалогового окна. В противном случае данные будут считываться из элементов управления диалогового окна и обновлять переменные оконного объекта. При выполнении этого метода происходит вызов виртуального метода DoDataExchange класса CWnd. Эту функцию нужно переопределить, чтобы она выполнила весь необходимый обмен. Как правило, в качестве объектов, обменивающихся данными с диалоговым окном, выступают объекты пользовательских классов, производных от CDialog

или CFormView. Для них имеется возможность создания переменных, связанных с элементом управления. В этом случае работу по созданию переопределенной функции DoDataExchange выполняет ClassWizard.

Методы управления текстом окна

`void SetWindowText(LPCTSTR lpszString);`

Устанавливает текст заголовка окна. Если окно - элемент управления, устанавливает текст в этом элементе. При вызове функции параметр должен быть либо указателем на строку символов, оканчивающуюся нулевым символом, либо переменной типа CString.

`int GetWindowText(LPSTR lpszStringBuf, int nMaxCount) const;`

`void GetWindowText(CString& rString) const;`

Копирует текст из заголовка окна. Если окно - элемент управления, копирует текст из этого элемента. При вызове первого варианта функции параметр lpszStringBuf должен быть указателем на буфер, в который будет скопирован текст, а параметр nMaxCount - выражением, задающим размер буфера (максимальное число символов, которое разрешается скопировать в буфер). Функция возвращает число скопированных символов, не включающее нуль-символ. При вызове второго варианта параметр rString должен быть переменной типа CString.

`int GetWindowTextLength() const;`

Возвращает длину текста заголовка окна, а если окно является элементом управления - длину текста этого элемента. Длина не учитывает нуль-символ.

`CFont* GetFont() const;`

Получает текущий шрифт данного окна.

`void SetFont(CFont* pFont, BOOL bRedraw=TRUE);`

Устанавливает текущий шрифт окна. Параметр pFont должен задавать новое значение для текущего шрифта. Если параметр bRedraw равен TRUE, окно после установки нового шрифта перерисовывается.

Некоторые методы класса CButton

UINT GetState() **const**;

Возвращает описание набора текущих состояний кнопки. Чтобы выделить из этого описания значения конкретных типов состояния, можно использовать маски:

- 0x0003 - выделяет собственное состояние кнопки. Применимо только к флажку или переключателю. Если результат побитового умножения дает 0, значит кнопка находится в невыбранном состоянии, 1 - в выбранном, 2 - в неопределенном.

- 0x0004 - выделяет состояние первого типа. Ненулевой вариант означает, что кнопка "нажата", нулевой - кнопка свободна.

- 0x0008 - выделяет положение фокуса. Ненулевой вариант - кнопка в фокусе клавиатуры.

int GetCheck() **const**;

Возвращает собственное состояние флажка или переключателя. Возвращаемое значение может принимать одно из значений: 0 - кнопка не выбрана; 1 - кнопка выбрана; 2 - кнопка в неопределенном состоянии. Если кнопка не является ни переключателем, ни флажком, возвращается 0.

void SetCheck(int nCheck);

Устанавливает собственное состояние флажка или переключателя. Значения задаются из набора: 0 - невыбранное; 1 - выбранное; 2 - неопределенное. Значение 2 применимо только к флажку со свойством 3State.

UINT GetButtonStyle() **const**;

Возвращает стиль кнопки.

void SetButtonStyle(UINT nStyle, **BOOL** bRedraw=TRUE);

Устанавливает стиль кнопки. Если параметр bRedraw равен TRUE, кнопка перерисовывается.

HICON GetIcon() **const**;

Возвращает дескриптор пиктограммы, сопоставленной кнопке. Если у кнопки нет сопоставленной пиктограммы, возвращает NULL.

```
HICON SetIcon(HICON hIcon);
```

Сопоставляет кнопке пиктограмму. Значением параметра при вызове должен быть дескриптор пиктограммы.

Пиктограмма автоматически помещается на поверхность кнопки и сдвигается в ее центр. Если поверхность кнопки меньше пиктограммы, она обрезается со всех сторон до размеров кнопки. Положение пиктограммы может быть выровнено и не по центру. Для этого нужно, чтобы кнопка имела одно из следующих свойств: BS_LEFT, BS_RIGHT, BS_CENTER, BS_TOP, BS_BOTTOM, BS_VCENTER

Данный метод устанавливает для кнопки только одну пиктограмму, которая будет наравне с текстом присутствовать при любом ее состоянии. Не надо путать ее с растровым изображением у растровой кнопки.

```
HBITMAP GetBitmap() const;
```

Возвращает дескриптор растрового изображения, сопоставленного кнопке. Если такового не существует, то возвращается NULL.

```
HBITMAP SetBitmap(HBITMAP hBitmap);
```

Сопоставляет кнопке растровое изображение. Значением параметра должен быть дескриптор растрового изображения. Правила размещения растрового изображения такие же, как и у значка.

```
HCURSOR GetCursor();
```

Возвращает дескриптор курсора, сопоставленного кнопке методом SetCursor. Если у кнопки нет сопоставленного курсора, то возвращается NULL.

```
HCURSOR SetCursor(HCURSOR hCursot);
```

Сопоставляет кнопке курсор, изображение которого будет помещено на поверхность кнопки аналогично значку и растровому изображению.

Некоторые методы класса CEdit

Окна редактирования могут работать в режимах однострочного и многострочного редакторов. Приведем сначала методы, общие для обоих режимов, а затем методы для многострочного редактора.

Общие методы

DWORD GetSel() const;

void GetSel(int& nStartChar, int& nEndChar) const;

Получает первую и последнюю позиции выделенного текста. Для значения типа DWORD младшее слово содержит позицию первого, старшее - последнего символа.

void SetSel(DWORD dwSelection, BOOL bNoScroll=FALSE);

void SetSel(int nStartChar, int nEndChar, BOOL bNoScroll=FALSE);

Устанавливает новое выделение текста, задавая первый и последний выделенный символ. Значение FALSE параметра bNoScroll должно отключать перемещение курсора в область видимости.

void ReplaceSel(LPCTSTR lpszNewText);

Заменяет выделенный текст на строку, передаваемую в параметре lpszNewText.

void Clear();

Удаляет выделенный текст.

void Copy();

Копирует выделенный текст в буфер.

void Cut();

Переносит (копирует и удаляет) выделенный текст в буфер обмена.

void Paste();

Вставляет текст из буфера обмена, начиная с позиции, в которой находится курсор.

BOOL Undo();

Отмена последней операции, выполненной редактором. Если редактор однострочный, возвращается всегда неотрицательное значение, иначе неотрицательное значение возвращается лишь в случае успешной замены.

BOOL CanUndo() **const**;

Определяет, можно ли отменить последнюю операцию редактора.

void EmptyUndoBuffer();

Сбрасывает флаг undo, сигнализирующий о возможности отмены последней операции редактора, и тем самым делает невозможным отмену. Этот флаг сбрасывается автоматически при выполнении методов SetWindowText и SetHandle.

BOOL GetModify() **const**;

Возвращает неотрицательное значение, если содержимое окна редактирования не модифицировалось. Информация о модификации поддерживается в специальном флаге, обнуляемом при создании окна редактирования и при вызове метода:

void SetModify(**BOOL** bModified=TRUE);

Устанавливает или сбрасывает флаг модификации (см. предыдущий метод). Флаг сбрасывается при вызове метода с параметром FALSE и устанавливается при модификации содержимого окна редактирования или при вызове SetModify с параметром TRUE.

BOOL SetReadOnly(**BOOL** bReadOnly=TRUE);

Устанавливает режим просмотра (bReadOnly=TRUE) или редактирования (bReadOnly=FALSE).

TCHAR GetPasswordChar() **const**;

Возвращает символ, который при выводе пароля будет появляться на экране вместо символов, набираемых пользователем. Если такой символ не определен, возвращается 0. Устанавливается этот символ методом (по умолчанию используется "*"):

```
void SetPasswordChar(TCHAR ch);
```

```
void LimitText(int nChars=0);
```

Устанавливает максимальную длину в байтах текста, который может ввести пользователь. Если значение параметра равно 0, длина текста устанавливается равной UINT_MAX.

Методы работы с многострочным редактором

```
void LineScroll(int nLines, int nChars=0);
```

Прокручивает текст в области редактирования. Параметр nLines задает число строк для вертикальной прокрутки. Окно редактирования не прокручивает текст дальше последней строки. При положительном значении параметра область редактирования сдвигается вдоль текста к последней строке, при отрицательной - к первой.

Параметр nChars задает число символов для горизонтальной прокрутки. Окно редактирования прокручивает текст вправо, даже если строки закончились. В этом случае в области редактирования появляются пробелы. При положительном значении параметра область редактирования сдвигается вдоль к концу строки, при отрицательном - к началу.

```
int GetFirstVisibleLine() const;
```

Возвращает номер первой видимой строки.

```
int GetLineCount() const;
```

Возвращает число строк текста, находящегося в буфере редактирования. Если текст не вводился, возвращает 1.

```
int GetLine(int nIndex, LPTSTR lpszBuffer) const;
```

```
int GetLine(int nIndex, LPTSTR lpszBuffer, int nMaxLength) const;
```

Копирует строку с номером, равным значению параметра nIndex, в буфер, заданный параметром lpszBuffer. Первое слово в буфере должно задавать его размер. При вызове второго варианта метода значение параметра nMaxLength копируется в первое слово буфера.

Метод возвращает число в действительности скопированных байтов. Если номер строки больше или равен числу строк в буфере окна редактирования, возвращает 0. Текст копируется без каких-либо изменений, нуль-символ не добавляется.

```
int LineIndex(int nIndex=-1) const;
```

Возвращает номер первого символа в строке. Неотрицательное значение параметра принимается в качестве номера строки. Значение -1 задает текущую строку. Если номер строки больше или равен числу строк в буфере окна редактирования (строки нумеруются с 0), возвращается 0.

Некоторые методы класса CListBox

```
void ResetContent();
```

Очищает содержимое списка, делая его пустым.

```
int AddString( LPCSTR lpszItem);
```

Добавляет строку lpszItem в список и сортирует его, если при создании включено свойство Sort. В противном случае элемент добавляется в конец списка.

```
int DeleteString( UINT nIndex);
```

Удаляет из списка элемент с индексом nIndex. Индексация элементов начинается с 0.

```
int GetCurSel() const;
```

Получает индекс элемента, выбранного пользователем.

```
int SetCurSel( int nSelect);
```

Отмечает элемент с индексом nSelect как выбранный элемент списка. Если значение параметра равно -1, список не будет содержать отмеченных элементов.

```
int GetText( int nIndex, LPSTR lpszBuffer) const;
```

```
void GetText( int nIndex, CString& rString) const;
```

Копирует элемент с индексом nIndex в буфер.

```
int SetTopIndex( int nIndex);
```

Организует прокрутку списка в окне так, чтобы элемент с индексом nIndex был видимым.

```
int FindString( int nStartAfter, LPCSTR lpszItem) const;
```

Организует поиск в списке и возвращает в качестве результата индекс элемента списка, префикс которого совпадает со строкой lpszItem. Результат не зависит от регистра, в котором набирались символы сравниваемых строк. Параметр nStartAfter задает начало поиска, но поиск идет по всему списку. Он начинается от элемента, следующего за nStartAfter, до конца списка и затем продолжается от начала списка до элемента с индексом nStartAfter. В качестве результата выдается первый найденный элемент, удовлетворяющий условиям поиска. Если такого нет, результат получает значение LB_ERR.

```
int FindStringExact( int nIndexStart, LPCSTR lpszFind) const;
```

Этот метод отличается от предыдущего тем, что теперь не префикс элемента должен совпадать со строкой lpszFind, а сам элемент. Поиск по-прежнему не чувствителен к регистру, в котором набираются символы.

Диагностика объектов.

Диагностический сервис помогает при отладке приложений. Для диагностики объектов в MFC существуют макросы, позволяющие печатать отладочную информации во время выполнения приложения, отслеживать распределение памяти, содержимое дампа (dump) объекта.

Макросы

ASSERT (booleanExpression) - прерывает выполнение программы, если выражение booleanExpression=FALSE, печатая при этом сообщение об ошибке.

ASSERT_KINDOF (className, pObject) - проверяет, является ли pObject объектом класса className. className — имя класса, производного от класса CObject. Этот макрос работает только, если в области объявления класса используется один из следующих макросов: DECLARE_DYNAMIC или DECLARE_SERIAL.

ASSERT_VALID (pObject) - проверяет внутреннее состояние объекта при помощи сравнения pObject с NULL, после чего вызывает его функцию-член AssertValid. При возникновении ошибки при одной из двух проверок, выводится сообщение об ошибке.

TRACE (exp) - выводит на экран форматированную строку exp, аналогично функции printf(), например: TRACE ("Hello %s. Year %d", "world", 2000) выведет на экран строку "Hello world. Year 2000."

TRACE0, ..., TRACE3 - аналогичны TRACE. Отличие состоит в том что эти макросы позволяют вывести форматированную строку с числом аргументов от 1 до 3 соответственно.

Вышеперечисленные макросы работают только в отладочной (debug) версии приложения.

VERIFY (booleanExpression) — макрос, аналогичный макросу ASSERT, только для рабочей (release) версии приложения.

Графические объекты

Библиотека MFC обеспечивает разработчиков всеми необходимыми классами, которые инкапсулируют соответствующие графические объекты Windows. Кроме того, библиотека имеет в своем составе дополнительные классы, значительно облегчающие решение ряда задач(классы CPoint, CSize, CRect, CRectTracker) или унифицирующие работу с графическими объектами как таковыми(CGdiObject).

Классы графических объектов

Класс CGdiObject

Базовый класс для всех классов, обеспечивающий интерфейс с графическими объектами Windows.

Класс CPen

Инкапсулирует объект Windows «карандаш», который может быть выбран в контекст устройства и использоваться для определения типа и цвета линий или графических фигур.

Класс CBrush

Инкапсулирует объект Windows «кисть», который может быть выбран в контекст устройства и использоваться для определения типа и цвета заливки внутренних областей замкнутых фигур.

Класс CFont

Инкапсулирует объект Windows «шрифт», который может быть выбран в контекст устройства и использоваться при операциях вывода текстовой информации.

Создать объект класса CPen или CBrush можно двумя способами:

1) Конструктор используется как для создания собственно объекта, так и для его инициализации.

Пример

```
CPen pen(PS_DOT,1,RGB(255,0,0));  
CBrush brush(HS_CROSS,RGB(0,255,0));
```

2) Конструктор используется только для создания объекта, а для его инициализации дополнительно вызывается функции (эта функции имеет префикс Create)

Пример

```
CPen pen;  
pen.CreatePen(PS_SO:ID,2,RGB(200,150,50));  
CBrush brush;  
brush.CreateSolidBrush(RGB(0,0,255));
```

Чтобы настроить параметры рисования с помощью соответствующего графического объекта, необходимо выполнить следующие действия

1) Создать графический объект.

2) Заменить в контексте устройства текущий графический объект вновь созданным, сохранив при этом указатель на «старый» объект.

3) Закончив операции рисования, восстановить «старый» графический объект в контексте устройства при помощи сохраненного указателя.

4) Обеспечить удаление созданного объекта при выходе из области видимости, что происходит автоматически для объектов, созданных в стеке приложения.

Установка графических объектов

Установка объектов рисования выполняется функцией SelectObject()

```
CPen* SelectObject( CPen* pPen );  
CBrush* SelectObject( CBrush* pBrush );
```

```
virtual CFont* SelectObject( CFont* pFont );
```

Кроме графических объектов, созданных в приложении, можно использовать и предопределенные системные. Для установки системных графических объектов используется функция SelectStockObject()

```
virtual CGdiObject* SelectStockObject( int nIndex );
```

Параметр nIndex задает тип создаваемого объекта.

Отображение графических фигур

Для отображения графических фигур можно использовать следующие функции-члены класса CDC:

Прямоугольник(квадрат)

```
BOOL Rectangle( int x1, int y1, int x2, int y2 );
```

```
BOOL Rectangle( LPCRECT lpRect );
```

Эллипс(окружность)

```
BOOL Ellipse( int x1, int y1, int x2, int y2 );
```

```
BOOL Ellipse( LPCRECT lpRect );
```

Сегмент

```
BOOL Chord( int x1, int y1, int x2, int y2, int x3, int y3, int x4, int y4 );
```

```
BOOL Chord( LPCRECT lpRect, POINT ptStart, POINT ptEnd );
```

Сектор

```
BOOL Pie( int x1, int y1, int x2, int y2, int x3, int y3, int x4, int y4 );
```

```
BOOL Pie( LPCRECT lpRect, POINT ptStart, POINT ptEnd );
```

Фигуры рисуются установленным «карандашом» CPen и закрашиваются кистью CBrush.

Вывод текста

Для вывода текста можно использовать функции-члены класса CDC- DrawText() и TextOut() , инкапсулирующие соответствующие функции Windows


```

int DrawText( LPCTSTR lpszString, int nCount, LPRECT lpRect, UINT nFormat );
int DrawText( const CString& str, LPRECT lpRect, UINT nFormat );
virtual BOOL TextOut( int x, int y, LPCTSTR lpszString, int nCount );
BOOL TextOut( int x, int y, const CString& str );

```

Цвет символов и фона текста устанавливаются следующими функциями

```

CDC::virtual COLORREF SetTextColor( COLORREF crColor );
CDC::virtual COLORREF SetBkColor( COLORREF crColor );

```

Для создания шрифта для вывода текста необходимо создать объект класса TFont, проинициализировать его и установить в контексте устройства.

```

CFont::BOOL CreateFontIndirect(const LOGFONT* lpLogFont );

```

Пример

```

CFont font;
LOGFONT lf;
memset(&lf, 0, sizeof(LOGFONT));
lf.lfHeight = 12;
strcpy(lf.lfFaceName, "Arial");
VERIFY(font.CreateFontIndirect(&lf));
CClientDC dc(this);
CFont* def_font = dc.SelectObject(&font);
dc.TextOut(5, 5, "Hello", 5);
dc.SelectObject(def_font);
font.DeleteObject();

```

```

CFont::BOOL CreateFont( int nHeight, int nWidth, int nEscapement, int
nOrientation, int nWeight, BYTE bItalic, BYTE bUnderline, BYTE cStrikeOut, BYTE
nCharSet, BYTE nOutPrecision, BYTE nClipPrecision, BYTE nQuality, BYTE
nPitchAndFamily, LPCTSTR lpszFacename );

```

Пример

```

CFont font;
VERIFY(font.CreateFont(
    12,           // nHeight
    0,           // nWidth

```

```

0,          // nEscapement
0,          // nOrientation
FW_NORMAL,  // nWeight
FALSE,      // bItalic
FALSE,      // bUnderline
0,          // cStrikeOut
ANSI_CHARSET, // nCharSet
OUT_DEFAULT_PRECIS, // nOutPrecision
CLIP_DEFAULT_PRECIS, // nClipPrecision
DEFAULT_QUALITY, // nQuality
DEFAULT_PITCH | FF_SWISS, // nPitchAndFamily
"Arial")); // lpszFacename
CClientDC dc(this);
CFont* def_font = dc.SelectObject(&font);
dc.TextOut(5, 5, "Hello", 5);
dc.SelectObject(def_font);
font.DeleteObject();
BOOL CreatePointFont( int nPointSize, LPCTSTR lpszFaceName, CDC* pDC = NULL );

```

Пример

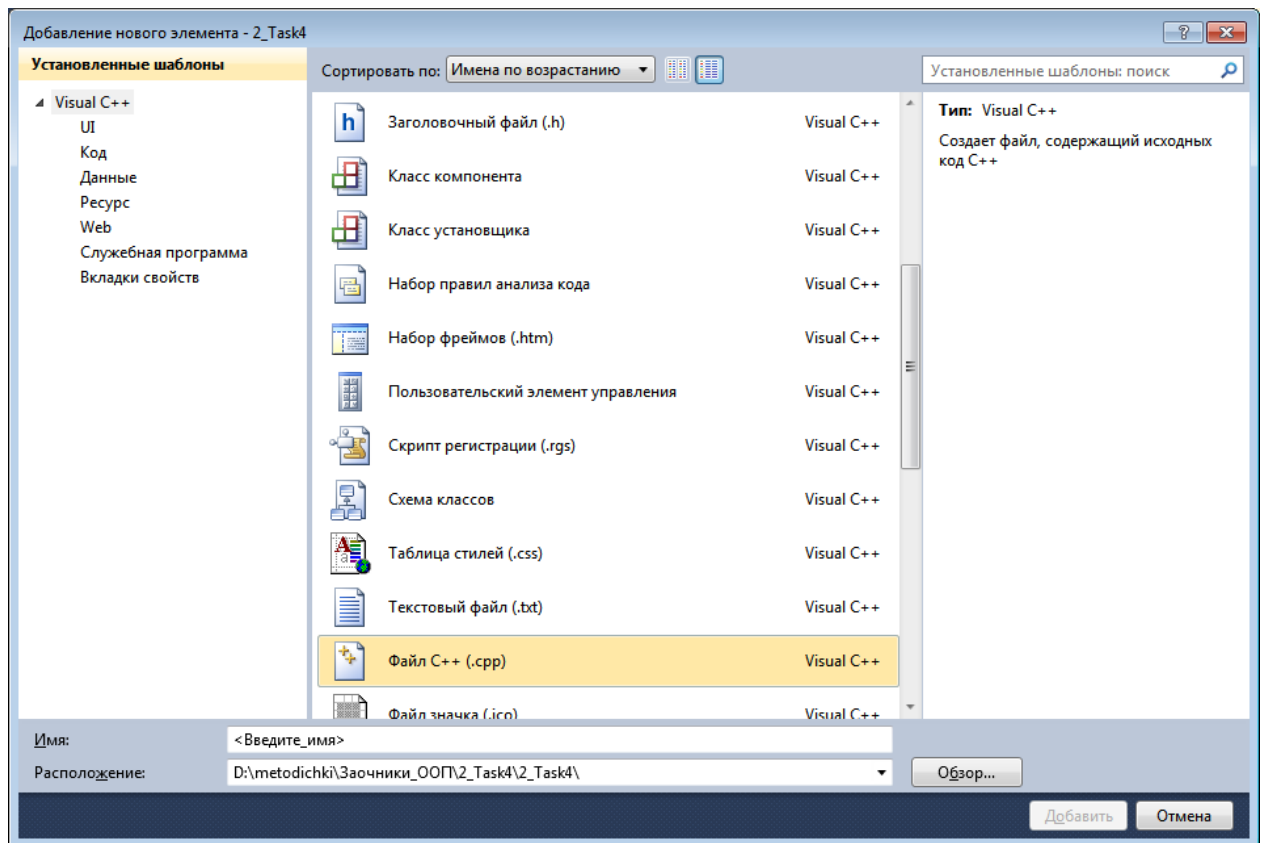
```

CClientDC dc(this);
CFont font;
VERIFY(font.CreatePointFont(120, "Arial", &dc));
CFont* def_font = dc.SelectObject(&font);
dc.TextOut(5, 5, "Hello", 5);
dc.SelectObject(def_font);
font.DeleteObject();

```

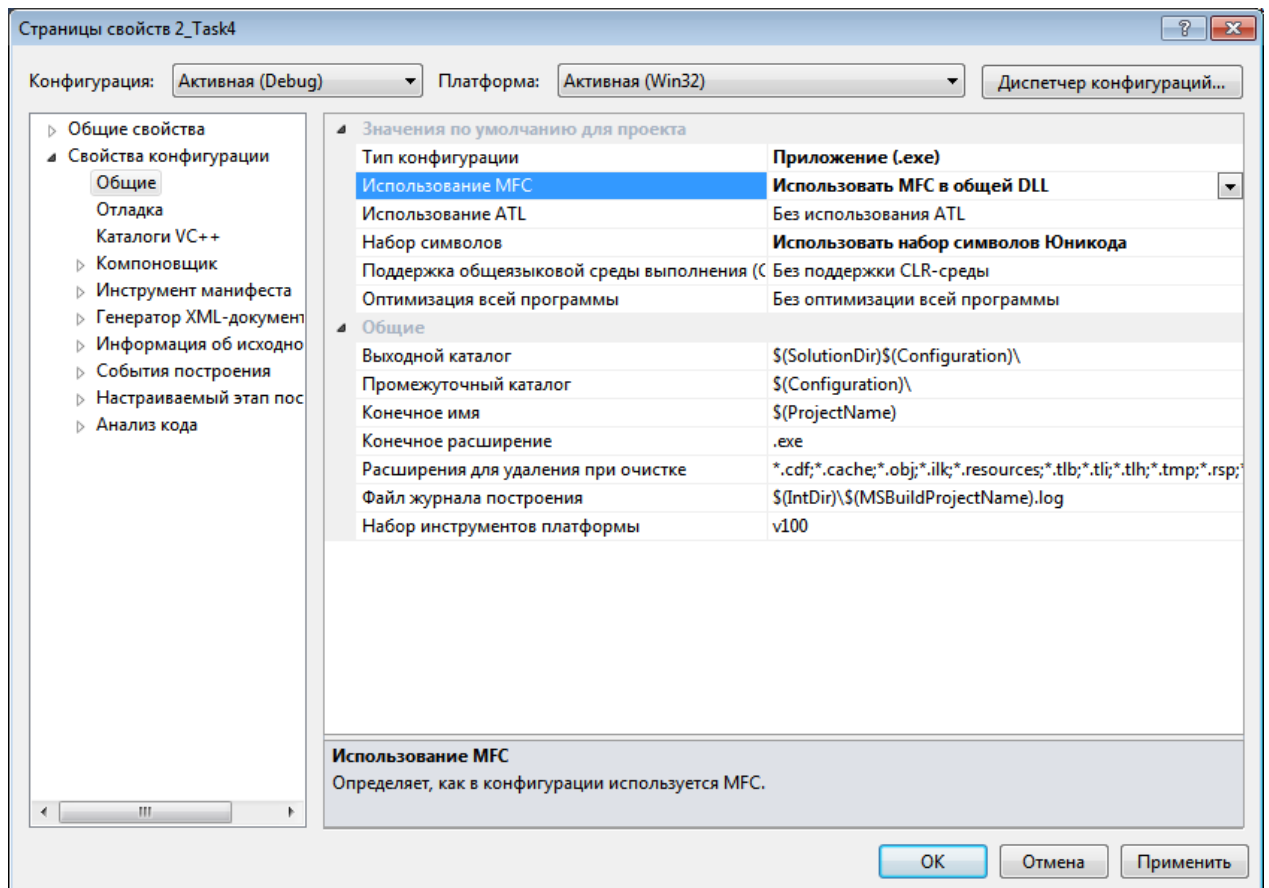
Методические указания

1. Загружаем Visual Studio и выбираем Файл->Создать ->Проект.
2. Выбираем тип проекта **Win32**. Этот тип - приложение под Windows, но без использования MFC. MFC же мы подсоединим позже. Выберите пункт пустой проект.
3. Добавим в проект файлы.



4. Укажем явно, что мы хотим использовать MFC. Для этого заходим в Свойства

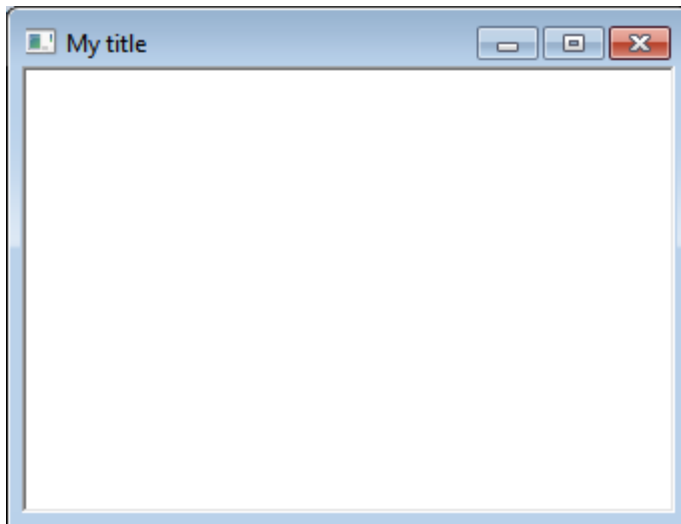
Выбираем **Использовать MFC в общей DLL**. Теперь наш проект будет использовать MFC.



5. Вставьте в файл вашего приложения следующий код:

```
#include <afxwin.h>
class CMyMainWnd : public CFrameWnd{
public:
    CMyMainWnd(){ // конструктор
        Create(NULL, L"My title");
    }
};
class CMyApp : public CWinApp{
public:
    CMyApp(){}; // конструктор
    virtual BOOL InitInstance(){
        m_pMainWnd=new CMyMainWnd();
        m_pMainWnd->ShowWindow(SW_SHOW);
        return TRUE;
    }
};
CMyApp theApp;
```

6. Запустите программу на выполнение
Должно появиться окно с заголовком "My title".



7. Разберем код.

Мы создали два класса - **CMyMainWnd** и **CMyApp**. Первый из них задаёт главное окно нашего приложения. Второй - само приложение. В конце нашего кода в строке

```
CMyApp theApp;
```

мы создаём экземпляр нашего приложения.

В классе главного окна ничего кроме конструктора нет. В конструкторе мы вызываем метод `Create`, который наш класс окна наследует от родительского класса.

В классе **CMyApp** переопределяется функция `InitInstance` родительского класса. В ней в строке

```
m_pMainWnd= new CMyMainWnd();
```

динамически создается новый экземпляр нашего главного окна. В следующей строке

```
m_pMainWnd->ShowWindow(SW_SHOW);
```

наше созданное окно показывается на экране.

8. Сделаем так, чтобы программа обращала внимание на наши действия. Например, чтобы при щелчке мышкой появлялся **MessageBox**.

Для этого в наш класс окна вставьте следующий строчки(они выделены):

```
class CMyMainWnd : public CFrameWnd{
public:
    CMyMainWnd(){ // конструктор
        Create(NULL,L"My title");
    }

    afx_msg void OnLButtonDown(UINT, CPoint);
    DECLARE_MESSAGE_MAP()
```

```
};
```

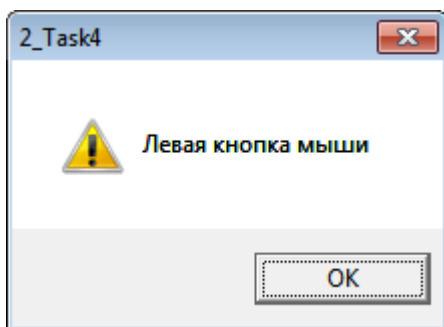
```
    После определения класса запишите  
BEGIN_MESSAGE_MAP(CMyMainWnd, CFrameWnd)  
ON_WM_LBUTTONDOWN()  
END_MESSAGE_MAP()
```

И, наконец, в конце файла добавьте строки

```
CMyApp theApp;
```

```
void CMyMainWnd::OnLButtonDown(UINT, CPoint){  
    AfxMessageBox(L"Левая кнопка мыши");  
}
```

9.Откомпилируйте и выполните приложение. При нажатии левой кнопки мыши в окне появляется MessageBox с надписью "Левая кнопка мыши".



10.Разберем код.

Для того, чтобы наш класс обращал внимание на наши действия, мы должны сделать следующие действия.

Первое. Мы должны вставить в конец нашего класса макрос **DECLARE_MESSAGE_MAP()**. Это достаточно сделать один раз. Этот макрос в классе и означает, что этот класс будет реагировать на некоторые сообщения.

Второе. Мы должны где-то после класса добавить два макроса **BEGIN_MESSAGE_MAP(..., ...)** и **END_MESSAGE_MAP()**. Это тоже достаточно сделать только один раз. Это так называемая карта сообщений. В первый макрос первым параметром вы должны вставить имя вашего класса, вторым - имя родительского класса. Первый параметр показывает, для какого класса мы пишем нашу карту сообщений, а второй - кто должен обрабатывать сообщение, которое наш класс обработать не может.

Третье. В классе пишем метод для обработки конкретного сообщения. Для стандартных сообщений имена методов стандартны. Например, для

сообщения **WM_ONLBUTTONDOWN** имя метода-обработчика **OnLButtonDown**. Перед названием метода не забудем написать `afx_msg`. В нашем примере это `afx_msg void OnLButtonDown(UINT, CPoint);`
Четвёртое. В карту сообщений пишем макрос для нашего сообщения. В нашем примере это строка **ON_WM_LBUTTONDOWN()**. Его имя - это **ON_** плюс имя сообщения.

```
BEGIN_MESSAGE_MAP(CMyMainWnd, CFrameWnd)
ON_WM_LBUTTONDOWN()
END_MESSAGE_MAP()
```

Пятое. Записываем код метода. Здесь мы для примера написали

```
void CMyMainWnd::OnLButtonDown(UINT, CPoint){
    AfxMessageBox("Левая кнопка мыши");
}
```

Функции с префиксом `Afx` определены в MFC как глобальные

11. Нарисуем что-нибудь в окне.

Когда окну надо что-либо перерисовать, оно получает сообщение **WM_PAINT**. Для рисования нам надо написать обработчик для этого события.

Вносим объявление функции-обработчика события **WM_PAINT** в класс окна:

```
class CMyMainWnd : public CFrameWnd{
public:
    CMyMainWnd(){ // конструктор
        Create(NULL, L"My title");
    }

    afx_msg void OnLButtonDown(UINT, CPoint);
    afx_msg void OnPaint();
    DECLARE_MESSAGE_MAP()

};
```

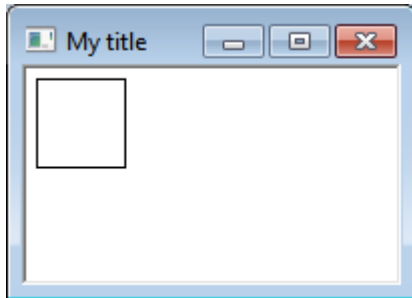
Затем добавляем макрос в карту сообщений:

```
BEGIN_MESSAGE_MAP(CMyMainWnd, CFrameWnd)
ON_WM_LBUTTONDOWN()
ON_WM_PAINT()
END_MESSAGE_MAP()
```

И, наконец, пишем реализацию нашей функции:

```
void CMyMainWnd::OnPaint(){
    CPaintDC* pDC=new CPaintDC(this);
    pDC->Rectangle(5,5,50,50);
}
```

Откомпилируем и выполним программу. В левом углу должен появиться квадратик.



12. Добавим в программу обработку таймера, т.е. события **WM_TIMER**.

Для этого, во-первых, создадим таймер. Для этого в конструктор класса CMyMainWnd добавим следующий код:

```
CMyMainWnd(){ // конструктор
    Create(NULL,L"My title");
    SetTimer(1, 1000, NULL);
}
```

Во-вторых, напишем обработчик события WM_TIMER.

Для этого вы должны сделать три действия - добавить соответствующий метод в класс, написать его реализацию и добавить соответствующий макрос в карту сообщений.

Добавим метод в класс:

```
...
afx_msg void OnPaint();
afx_msg void OnTimer(UINT);
DECLARE_MESSAGE_MAP()
```

Напишем реализацию этого метода:

```
void CMyMainWnd::OnTimer(UINT){
    MessageBeep(-1);
}
```

В-третьих, добавим макрос:


```

...
BEGIN_MESSAGE_MAP(CMyMainWnd, CFrameWnd)
ON_WM_LBUTTONDOWN()
ON_WM_PAINT()
ON_WM_TIMER()
END_MESSAGE_MAP()

```

13.Выполним программу. Каждую секунду должен издаваться звук beep.

14.Так как таймер системный ресурс, в конце программы его надо удалить. Для этого вносим класс код для деструктора:

```

~CMyMainWnd(){
    KillTimer(1);
}

```

15.Заставим программу работать одновременно с двумя таймерами. Добавляем в программу ещё один таймер и сразу пишем код в деструкторе класса для уничтожения нового таймера:

```

CMyMainWnd(){ // конструктор
    Create(NULL,L"My title");
    SetTimer(1, 1000, NULL);
    SetTimer(2, 3000, NULL);
}
~CMyMainWnd(){
KillTimer(1);
    KillTimer(2);
}

```

Идентификатор нового таймера 2 , и он тикает раз в три секунды.

16.Отдельный обработчик для второго таймера писать не надо, а надо изменить обработчик для таймера следующим образом:

```

void CMyMainWnd::OnTimer(UINT nIDEvent){
    if(nIDEvent==1)
        MessageBeep(-1);
    else
        SetWindowText(L"Title");
}

```

У метода OnTimer есть параметр типа UINT. Это есть идентификатор таймера, для которого мы обрабатываем сообщение WM_TIMER. Если сообщение поступило от первого таймера, то издаём сигнал, а если от второго, то меняем заголовок окна на "Title".

17.Выполним программу. Звук раздаётся раз в секунду, и через три секунды заголовок окна меняется

18. Измените интервал у таймера, т. е. с делайте, чтобы сначала он тикал с одной частотой, а затем с другой. Частота должна меняться по щелчку правой кнопки мыши.

Принцип здесь простой - сначала надо убить старый таймер, а затем создать новый с таким же идентификатором.

Для этого напишете обработчик события **WM_RBUTTONDOWN**, откомпилируйте и выполните программу.

Пример

```
#include <afxwin.h>
#include <cstring>
class CMainWin: public CFrameWnd
{
public:
    CMainWin();
    afx_msg void OnChar(UINT ch, UINT, UINT);
    afx_msg void OnPaint();
    afx_msg void OnLButtonDown(UINT flags, CPoint Loc);
    afx_msg void OnRButtonDown(UINT flags, CPoint Loc);
    char str[50];
    int nMouseX, nMouseY, nOldMouseX, nOldMouseY;
    char pszMouseStr[50];
DECLARE_MESSAGE_MAP()
};
class CApp: public CWinApp
{
public:
    BOOL InitInstance();
};
BOOL CApp::InitInstance()
{
    m_pMainWnd = new CMainWin;
    m_pMainWnd->ShowWindow(SW_RESTORE);
    m_pMainWnd->UpdateWindow();
    return TRUE;
}
CMainWin::CMainWin()
{
    this->Create(0, "Обработка сообщений");
    strcpy(str, "");
    strcpy(pszMouseStr, "");
    nMouseX = nMouseY = nOldMouseX = nOldMouseY = 0;
}

BEGIN_MESSAGE_MAP
```

```

(CMainWin /* класс */, CFrameWnd /* базовый класс */)
ON_WM_CHAR()
ON_WM_PAINT()
ON_WM_LBUTTONDOWN()
ON_WM_RBUTTONDOWN()
END_MESSAGE_MAP()

afx_msg void CMainWin::OnChar(UINT ch, UINT, UINT)
{
    sprintf(str, "%c", ch);
    this->InvalidateRect(0);
}

afx_msg void CMainWin::OnPaint()
{
    CPaintDC dc(this);
    dc.TextOut(nOldMouseX, nOldMouseY, " ", 30);
    dc.TextOut(nMouseX, nMouseY, pszMouseStr);
    dc.TextOut(1, 1, " ");
    dc.TextOut(1, 1, str);
}

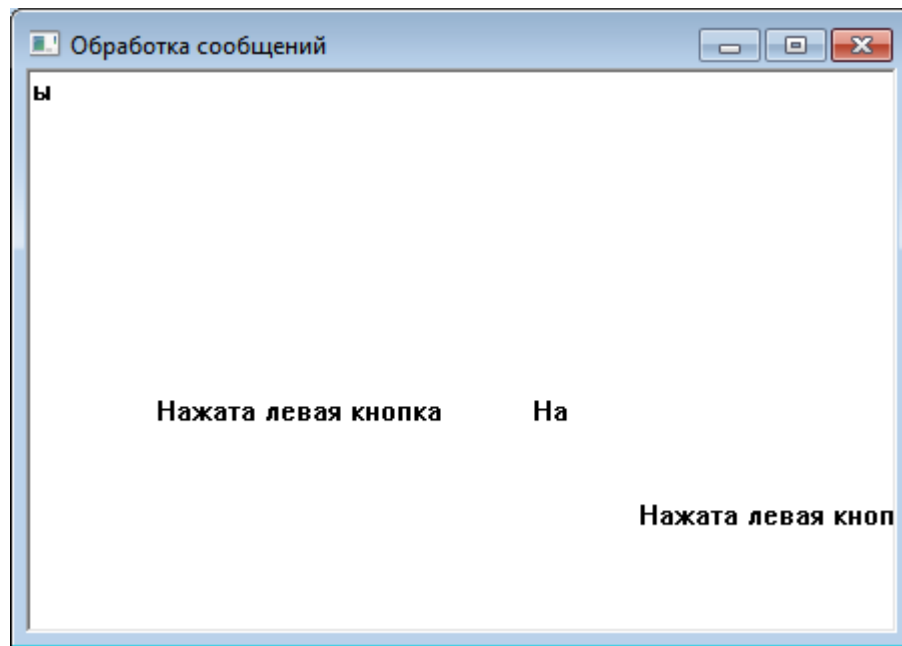
afx_msg void CMainWin::OnLButtonDown
                      (UINT, CPoint loc)
{
    nOldMouseX = nMouseX;
    nOldMouseY = nMouseY;
    strcpy(pszMouseStr, "Нажата левая кнопка");
    nMouseX = loc.x; nMouseY = loc.y;
    this->InvalidateRect(0);
}

afx_msg void CMainWin::OnRButtonDown
                      (UINT, CPoint loc)
{
    nOldMouseX = nMouseX;
    nOldMouseY = nMouseY;
    strcpy(pszMouseStr, "Нажата правая кнопка");
    nMouseX = loc.x; nMouseY = loc.y;
    this->InvalidateRect(0);
}

CApp App;

```

Результат выполнения



№ 15 Библиотеки динамической компоновки. Экспортирование функций и классов

1. Создать Проект, в котором в одно диалоговое окно выводится строка – результат функции из DLL в другое - выводится строка – результат функции из класса DLL (функции могут совпадать, тогда при вывод дать пояснительный текст, какая из функций работает).
2. Реализовать два варианта подключения – явное (динамическая загрузка LoadLibrary) и неявное (подключение через статическую библиотеку).

Вариант	Функция
1,7,13, 19,25	Определить сумму свободного места на логических дисках
2,8,14, 20,26	Определить логические диски компьютера
3,9,15,21	Определить имя локального компьютера
4,10,16,22	Определить свободное количество физической памяти в байтах
5,11,17,23	Определить загрузку памяти в процентах
6,12,18,24	Определить MAC адрес компьютера

Для справки:

GetLogicalDrives

Функция GetLogicalDrives возвращает число-битовую маску в которой хранятся все доступные диски.

DWORD GetLogicalDrives(VOID);

Параметры:

Эта функция не имеет параметров.

Возвращаемое значение:

Если функция вызвана правильно, то она возвращает число-битовую маску в которой

хранятся все доступные диски (если 0 бит равен 1, то диск "А:" присутствует, и т.д.)

Если функция вызвана не правильно, то она возвращает 0.

GetDriveType

Функция GetDriveType возвращает тип диска (removable, fixed, CD-ROM, RAM disk, или network drive).

```
UINT GetDriveType(LPCTSTR lpRootPathName);
```

Параметры:

lpRootPathName

[in] Указатель на не нулевую строку в которой хранится имя главной директории на диске. Обратный слэш должен присутствовать! Если lpRootPathName равно NULL, то функция использует текущую директорию.

Возвращаемое значение:

Функция возвращает тип диска. Могут быть следующие значения:

Значение	Описание
DRIVE_UNKNOWN	Не известный тип.
DRIVE_NO_ROOT_DIR	Не правильный путь.
DRIVE_REMOVABLE	Съёмный диск.
DRIVE_FIXED	Фиксированный диск.
DRIVE_REMOTE	Удалённый или network диск.
DRIVE_CDROM	CD-ROM диск.
DRIVE_RAMDISK	RAM диск.

GetVolumeInformation

Функция GetVolumeInformation возвращает информацию о файловой системе и дисках(директориях).

```
BOOL GetVolumeInformation(
LPCTSTR lpRootPathName, // имя диска(директории) [in]
LPTSTR lpVolumeNameBuffer, // название диска [out]
DWORD nVolumeNameSize, // длина буфера названия диска [in]
LPDWORD lpVolumeSerialNumber, // серийный номер диска [out]
LPDWORD lpMaximumComponentLength, // максимальная длина файла [out]
LPDWORD lpFileSystemFlags, // опции файловой системы [out]
LPTSTR lpFileSystemNameBuffer, // имя файловой системы [out]
DWORD nFileSystemNameSize // длина буфера имени файл. сист. [in]
);
```

Возвращаемое значение:

Если функция вызвана правильно, то она возвращает не нулевое значение(TRUE).

Если функция вызвана не правильно, то она возвращает 0(FALSE).

GetDiskFreeSpaceEx

Функция GetDiskFreeSpaceEx выдаёт информацию о доступном месте на диске.

```
BOOL GetDiskFreeSpaceEx(  
LPCTSTR lpDirectoryName, // имя диска(директории) [in]  
PULARGE_INTEGER lpFreeBytesAvailable, // доступно для использования(байт)  
[out]  
PULARGE_INTEGER lpTotalNumberOfBytes, // максимальный объём( в байтах )  
[out]  
PULARGE_INTEGER lpTotalNumberOfFreeBytes // свободно на диске( в байтах )  
[out]  
);
```

Возвращаемое значение:

Если функция вызвана правильно, то она возвращает не нулевое значение(TRUE).

Если функция вызвана не правильно, то она возвращает 0(FALSE).

GlobalMemoryStatus

Функция GlobalMemoryStatus возвращает информацию о используемой системой памяти.

```
VOID GlobalMemoryStatus(  
LPMEMORYSTATUS lpBuffer // указатель на структуру MEMORYSTATUS  
);
```

```
typedef struct _MEMORYSTATUS {  
DWORD dwLength; // длина структуры в байтах  
DWORD dwMemoryLoad; // загрузка памяти в процентах  
SIZE_T dwTotalPhys; // максимальное количество физической памяти в байтах  
SIZE_T dwAvailPhys; // свободное количество физической памяти в байтах  
SIZE_T dwTotalPageFile; // макс. кол. памяти для программ в байтах  
SIZE_T dwAvailPageFile; // свободное кол. памяти для программ в байтах  
SIZE_T dwTotalVirtual; // максимальное количество виртуальной памяти в  
байтах
```

```
SIZE_T dwAvailVirtual; // свободное количество виртуальной памяти в байтах  
} MEMORYSTATUS, *LPMEMORYSTATUS;
```

Возвращаемое значение:

Эта функция не возвращает параметров

GetComputerName, GetUserNameA

Функция GetComputerName возвращает NetBIOS имя локального компьютера.

```
BOOL GetComputerName(  
LPTSTR lpBuffer, // имя локального компьютера( длина буфера равна  
MAX_COMPUTERNAME_LENGTH + 1 ) [out]  
LPDWORD lpnSize // размер буфера ( лучше поставить  
MAX_COMPUTERNAME_LENGTH + 1 ) [out/in]  
);
```

Функция GetUserName возвращает имя текущего узера.

```
BOOL GetUserName(  
LPTSTR lpBuffer, // имя юзера( длина буфера равна UNLEN + 1 ) [out]  
LPDWORD nSize // размер буфера ( лучше поставить UNLEN + 1 ) [out/in]  
);
```

Возвращаемые значения:

Если функции вызваны правильно, то они возвращают не нулевое значение(TRUE).

Если функции вызваны не правильно, то они возвращают 0(FALSE).

GetSystemDirectory, GetTempPath, GetWindowsDirectory, GetCurrentDirectory

Функция GetSystemDirectory возвращает путь к системной директории.

```
UINT GetSystemDirectory(  
LPTSTR lpBuffer, // буфер для системной директории [out]  
UINT uSize // размер буфера [in]  
);
```

Возвращаемое значение:

Эта функция возвращает размер буфера для системной директории не включая нулевого

значения в конце, если она вызвана правильно.
Если функция вызвана не правильно, то она возвращает 0.

Теория

Введение

Динамически подключаемые библиотеки (dynamic-link libraries, DLL) — краеугольный камень операционной системы Windows, начиная с самой первой ее версии. В DLL содержатся все функции Windows API. Три самые важные DLL: Kernel32.dll (управление памятью, процессами и потоками), User32.dll (поддержка пользовательского интерфейса, в том числе функции, связанные с созданием окон и передачей сообщений) и GDI32.dll (графика и вывод текста). В Windows есть и другие DLL, функции которых предназначены для более специализированных задач. Например, в AdvAPI32.dll содержатся функции для защиты объектов, работы с реестром и регистрации событий, в ComDlg32.dll ~ стандартные диалоговые окна (вроде File Open и File Save), а ComCtl32.dll поддерживает стандартные элементы управления. Вот лишь некоторые из причин, по которым нужно применять DLL:

- Расширение функциональности приложения. DLL можно загружать в адресное пространство процесса динамически, что позволяет приложению, определив, какие действия от него требуются, подгружать нужный код.
- Возможность использования разных языков программирования. У Вас есть выбор, на каком языке писать ту или иную часть приложения. Так, пользовательский интерфейс приложения Вы скорее всего будете создавать на Microsoft Visual Basic, но прикладную логику лучше всего реализовать на C++. Программа на Visual Basic может загружать DLL, написанные на C++, Коболе, Фортране и др.
- Более простое управление проектом. Если в процессе разработки программного продукта отдельные его модули создаются разными группами, то при использовании DLL таким проектом управлять гораздо проще. Однако конечная версия приложения должна включать как можно меньше файлов.
- Экономия памяти. Если одну и ту же DLL использует несколько приложений, в оперативной памяти может храниться только один ее экземпляр, доступный этим приложениям. Пример — DLL-версия библиотеки C/C++. Ею пользуются многие приложения. Если всех их скомпоновать со статически подключаемой версией этой библиотеки, то код таких функций, как `sprintf`, `strcpy`, `malloc` и др., будет многократно дублироваться в памяти. Но если они компонируются с DLL-версией библиотеки C/C++, в памяти будет присутствовать лишь одна копия кода этих функций, что позволит гораздо эффективнее использовать оперативную память.

- Разделение ресурсов. DLL могут содержать такие ресурсы, как шаблоны диалоговых окон, строки, значки и битовые карты (растровые изображения). Эти ресурсы доступны любым программам

- Упрощение локализации. DLL нередко применяются для локализации приложений. Например, приложение, содержащее только код без всяких компонентов пользовательского интерфейса, может загружать DLL с компонентами локализованного интерфейса

- Решение проблем, связанных с особенностями различных платформ. В разных версиях Windows содержатся разные наборы функций. Зачастую разработчикам нужны новые функции, существующие в той версии системы, которой они пользуются. Если Ваша версия Windows не поддерживает эти функции, Вам не удастся запустить такое приложение: загрузчик попросту откажется его запускать.

Согласование интерфейсов

При использовании собственных библиотек или библиотек независимых разработчиков придется обратить внимание на согласование вызова функции с ее прототипом.

Если бы мир был совершенен, то программистам не пришлось бы беспокоиться о согласовании интерфейсов функций при подключении библиотек и все они были бы одинаковыми. Однако мир далек от совершенства, и многие большие программы написаны с помощью различных библиотек без C++.

По умолчанию в Visual C++ интерфейсы функций согласуются по правилам C++. Это значит, что параметры заносятся в стек справа налево, вызывающая программа отвечает за их удаление из стека при выходе из функции и расширении ее имени. Расширение имен (name mangling) позволяет редактору связей различать перегруженные функции, т.е. функции с одинаковыми именами, но разными списками аргументов. Однако в старой библиотеке C функции с расширенными именами отсутствуют.

Хотя все остальные правила вызова функции в C идентичны правилам вызова функции в C++, в библиотеках C имена функций не расширяются. К ним только добавляется впереди символ подчеркивания (_).

Если необходимо подключить библиотеку на C к приложению на C++, все функции из этой библиотеки придется объявить как внешние в формате C:

```
extern "C" int MyOldCFunction(int myParam);
```

Объявления функций библиотеки обычно помещаются в файле заголовка этой библиотеки, хотя заголовки большинства библиотек C не

рассчитаны на применение в проектах на C++. В этом случае необходимо создать копию файла заголовка и включить в нее модификатор `extern "C"` к объявлению всех используемых функций библиотеки. Модификатор `extern "C"` можно применить и к целому блоку, к которому с помощью директивы `#include` подключен файл старого заголовка C. Таким образом, вместо модификации каждой функции в отдельности можно обойтись всего тремя строками:

```
extern "C"
{
    #include "MyCLib.h"
}
```

В программах для старых версий Windows использовались также соглашения о вызове функций языка PASCAL для функций Windows API. В новых программах следует использовать модификатор `winapi`, преобразуемый в `_stdcall`. Хотя это и не стандартный интерфейс функций C или C++, но именно он используется для обращений к функциям Windows API. Однако обычно все это уже учтено в стандартных заголовках Windows.

Загрузка неявно подключаемой DLL

При запуске приложение пытается найти все файлы DLL, неявно подключенные к приложению, и поместить их в область оперативной памяти, занимаемую данным процессом. Поиск файлов DLL операционной системой осуществляется в следующей последовательности:

- Каталог, в котором находится EXE-файл.
- Текущий каталог процесса.
- Системный каталог Windows.

Если библиотека DLL не обнаружена, приложение выводит диалоговое окно с сообщением о ее отсутствии и путях, по которым осуществлялся поиск. Затем процесс отключается.

Если нужная библиотека найдена, она помещается в оперативную память процесса, где и остается до его окончания. Теперь приложение может обращаться к функциям, содержащимся в DLL.

Динамическая загрузка и выгрузка DLL

Вместо того, чтобы Windows выполняла динамическое связывание с DLL при первой загрузке приложения в оперативную память, можно связать программу с модулем библиотеки во время выполнения программы (при таком способе в процессе создания приложения не нужно использовать

библиотеку импорта). В частности, можно определить, какая из библиотек DLL доступна пользователю, или разрешить пользователю выбрать, какая из библиотек будет загружаться. Таким образом можно использовать разные DLL, в которых реализованы одни и те же функции, выполняющие различные действия. Например, приложение, предназначенное для независимой передачи данных, сможет в ходе выполнения принять решение, загружать ли DLL для протокола TCP/IP или для другого протокола.

Загрузка обычной DLL

Первое, что необходимо сделать при динамической загрузке DLL, - это поместить модуль библиотеки в память процесса. Данная операция выполняется с помощью функции `::LoadLibrary`, имеющей единственный аргумент имя загружаемого модуля. Соответствующий фрагмент программы должен выглядеть так:

```
HINSTANCE hMyDll;  
    if((hMyDll=::LoadLibrary("MyDLL"))==NULL) {  
/* не удалось загрузить DLL */  
    else {  
/* приложение имеет право пользоваться функциями DLL через hMyDll */  
    }
```

Стандартным расширением файла библиотеки Windows считает `.dll`, если не указать другое расширение. Если в имени файла указан и путь, то только он будет использоваться для поиска файла. В противном случае Windows будет искать файл по той же схеме, что и в случае неявно подключенных DLL, начиная с каталога, из которого загружается `exe`-файл, и продолжая в соответствии со значением `PATH`.

Когда Windows обнаружит файл, его полный путь будет сравнен с путем библиотек DLL, уже загруженных данным процессом. Если обнаружится тождество, вместо загрузки копии приложения возвратится дескриптор уже подключенной библиотеки.

Если файл обнаружен и библиотека успешно загрузилась, функция `::LoadLibrary` возвращает ее дескриптор, который используется для доступа к функциям библиотеки.

Перед тем, как использовать функции библиотеки, необходимо получить их адрес. Для этого сначала следует воспользоваться директивой `typedef` для определения типа указателя на функцию и определить переменную этого нового типа, например:

```
// тип PFN_MyFunction будет объявлять указатель на функцию,
```

```

        // принимающую указатель на символьный буфер и выдающую
значение типа int
        typedef int (WINAPI *PFN_MyFunction)(char *);

        PFN_MyFunction pfnMyFunction;

```

Затем следует получить дескриптор библиотеки, при помощи которого и определить адреса функций, например адрес функции с именем MyFunction:

```

        hMyDll=::LoadLibrary("MyDLL");

pfnMyFunction=(PFN_MyFunction)::GetProcAddress(hMyDll,"MyFunction");

        int iCode=(*pfnMyFunction)("Hello");

```

Адрес функции определяется при помощи функции ::GetProcAddress, ей следует передать имя библиотеки и имя функции. Последнее должно передаваться в том виде, в котором эксортируется из DLL.

Можно также сослаться на функцию по порядковому номеру, по которому она экспортируется (при этом для создания библиотеки должен использоваться def-файл, об этом будет рассказано далее):

```

pfnMyFunction=(PFN_MyFunction)::GetProcAddress(hMyDll,
                                                MAKEINTRESOURCE(1));

```

После завершения работы с библиотекой динамической компоновки, ее можно выгрузить из памяти процесса с помощью функции ::FreeLibrary:

```

        ::FreeLibrary(hMyDll);

```

Загрузка MFC-расширений динамических библиотек

При загрузке MFC-расширений для DLL (подробно о которых рассказывается далее) вместо функций LoadLibrary и FreeLibrary используются функции AfxLoadLibrary и AfxFreeLibrary. Последние почти идентичны функциям Win32 API. Они лишь гарантируют дополнительно, что структуры MFC, инициализированные расширением DLL, не были запорчены другими потоками.

Ресурсы DLL

Динамическая загрузка применима и к ресурсам DLL, используемым MFC для загрузки стандартных ресурсов приложения. Для этого сначала

необходимо вызвать функцию LoadLibrary и разместить DLL в памяти. Затем с помощью функции AfxSetResourceHandle нужно подготовить окно программы к приему ресурсов из вновь загруженной библиотеки. В противном случае ресурсы будут загружаться из файлов, подключенных к выполняемому файлу процесса. Такой подход удобен, если нужно использовать различные наборы ресурсов, например для разных языков.

Замечание. С помощью функции LoadLibrary можно также загружать в память исполняемые файлы (не запускать их на выполнение!). Дескриптор выполняемого модуля может затем использоваться при обращении к функциям FindResource и LoadResource для поиска и загрузки ресурсов приложения. Выгружают модули из памяти также при помощи функции FreeLibrary.

Пример обычной DLL и способов загрузки

Приведем исходный код динамически подключаемой библиотеки, которая называется MyDLL и содержит одну функцию MyFunction, которая просто выводит сообщение.

Сначала в заголовочном файле определяется макроконтстанта EXPORT. Использование этого ключевого слова при определении некоторой функции динамически подключаемой библиотеке позволяет сообщить компоновщику, что эта функция доступна для использования другими программами, в результате чего он заносит ее в библиотечку импорта. Кроме этого, такая функция, точно так же, как и оконная процедура, должна определяться с помощью константы CALLBACK:

```
-----MyDLL.h-----  
-----  
#define EXPORT extern "C"__declspec (dllexport)  
EXPORT int CALLBACK MyFunction(char *str);
```

Файл библиотеки также несколько отличается от обычных файлов на языке C для Windows. В нем вместо функции WinMain имеется функция DllMain. Эта функция используется для выполнения инициализации, о чем будет рассказано позже. Для того, чтобы библиотека осталась после ее загрузки в памяти, и можно было вызывать ее функции, необходимо, чтобы ее возвращаемым значением было TRUE:

```
----- MyDLL.c-----  
-----  
#include <windows.h>  
#include "MyDLL.h"
```

```

        int WINAPI DllMain(HINSTANCE hInstance, DWORD fdReason, PVOID
pvReserved)
        {
            return TRUE;
        }
    EXPORT int CALLBACK MyFunction(char *str)
    {
        MessageBox(NULL,str,"Function from DLL",MB_OK);
        return 1;
    }

```

После трансляции и компоновки этих файлов появятся два файла: MyDLL.dll (сама динамически подключаемая библиотека) и MyDLL.lib (ее библиотека импорта).

Пример неявного подключения DLL приложением

Приведем теперь исходный код простого приложения, которое использует функцию MyFunction из библиотеки MyDLL.dll:

```

#include <windows.h>
#include "MyDLL.h"

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE
hPrevInstance,
                LPSTR lpCmdLine, int nCmdShow)
{
    int iCode=MyFunction("Hello");
    return 0;
}

```

Эта программа выглядит как обычная программа для Windows, чем она в сущности и является. Тем не менее, следует обратить внимание, что в исходный ее текст помимо вызова функции MyFunction из DLL-библиотеки включен и заголовочный файл этой библиотеки MyDLL.h. Также необходимо на этапе компоновки приложения подключить к нему библиотеку импорта MyDLL.lib (процесс неявного подключения DLL к исполняемому модулю).

Чрезвычайно важно понимать, что сам код функции MyFunction не включается в файл MyApp.exe. Вместо этого там просто имеется ссылка на файл MyDLL.dll и ссылка на функцию MyFunction, которая находится в этом файле. Файл MyApp.exe требует запуска файла MyDLL.dll.

Заголовочный файл MyDLL.h включен в файл с исходным текстом программы MyApp.c точно так же, как туда включен файл windows.h. Включение библиотеки импорта MyDLL.lib для компоновки аналогично

включению туда всех библиотек импорта Windows. Когда программа MyApp.exe работает, она подключается к библиотеке MyDLL.dll точно так же, как ко всем стандартным динамически подключаемым библиотекам Windows.

Пример динамической загрузки DLL приложением

Приведем теперь полностью исходный код простого приложения, которое использует функцию MyFunction из библиотеки MyDLL.dll, используя динамическую загрузку библиотеки:

```
#include <windows.h>
typedef int (WINAPI *PFN_MyFunction)(char *);

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE
hPrevInstance,
                    LPSTR lpCmdLine, int nCmdShow)
{
    HINSTANCE hMyDll;
    if((hMyDll=LoadLibrary("MyDLL"))==NULL) return 1;

    PFN_MyFunction pfnMyFunction;

    pfnMyFunction=(PFN_MyFunction)GetProcAddress(hMyDll,"MyFunction");
    int iCode=(*pfnMyFunction)("Hello");

    FreeLibrary(hMyDll);
    return 0;
}
```

Создание DLL

Теперь, познакомившись с принципами работы библиотек DLL в приложениях, рассмотрим способы их создания. При разработке приложения функции, к которым обращается несколько процессов, желательно размещать в DLL. Это позволяет более рационально использовать память в Windows.

Проще всего создать новый проект DLL с помощью мастера AppWizard, который автоматически выполняет многие операции. Для простых DLL, таких как рассмотренные в этой главе, необходимо выбрать тип проекта Win32 Dynamic-Link Library. Новому проекту будут присвоены все необходимые

параметры для создания библиотеки DLL. Файлы исходных текстов придется добавлять к проекту вручную.

Если же планируется в полной мере использовать функциональные возможности MFC, такие как документы и представления, или намерены создать сервер автоматизации OLE, лучше выбрать тип проекта MFC AppWizard (dll). В этом случае, помимо присвоения проекту параметров для подключения динамических библиотек, мастер проделает некоторую дополнительную работу. В проект будут добавлены необходимые ссылки на библиотеки MFC и файлы исходных текстов, содержащие описание и реализацию в библиотеке DLL объекта класса приложения, производного от CWinApp.

Иногда удобно сначала создать проект типа MFC AppWizard (dll) в качестве тестового приложения, а затем в библиотеку DLL в виде его составной части. В результате DLL в случае необходимости будет создаваться автоматически.

Функция DllMain

Большинство библиотек DLL в просто коллекции практически независимых друг от друга функций, экспортируемых в приложения и используемых в них. Кроме функций, предназначенных для экспортирования, в каждой библиотеке DLL есть функция DllMain. Эта функция предназначена для инициализации и очистки DLL. Она пришла на смену функциям LibMain и WEP, применявшимся в предыдущих версиях Windows. Структура простейшей функции DllMain может выглядеть, например, так:

```
BOOL WINAPI DllMain (HANDLE hInst,DWORD dwReason, LPVOID
IpReserved)
{
    BOOL bAllWentWell=TRUE;
    switch (dwReason)
    {
        case DLL_PROCESS_ATTACH:
// Инициализация процесса.
            break;
        case DLL_THREAD_ATTACH: // Инициализация потока.
            break;
        case DLL_THREAD_DETACH: // Очистка структур потока.
            break;
        case DLL_PROCESS_DETACH: // Очистка структур
процесса.
            break;
    }
    if(bAllWentWell)        return TRUE;
```

```
        else                return FALSE;
    }
```

Функция DllMain вызывается в нескольких случаях. Причина ее вызова определяется параметром dwReason, который может принимать одно из следующих значений.

При первой загрузке библиотеки DLL процессом вызывается функция DllMain с dwReason, равным DLL_PROCESS_ATTACH. Каждый раз при создании процессом нового потока DllMainO вызывается с dwReason, равным DLL_THREAD_ATTACH (кроме первого потока, потому что в этом случае dwReason равен DLL_PROCESS_ATTACH).

По окончании работы процесса с DLL функция DllMain вызывается с параметром dwReason, равным DLL_PROCESS_DETACH. При уничтожении потока (кроме первого) dwReason будет равен DLL_THREAD_DETACH.

Все операции по инициализации и очистке для процессов и потоков, в которых нуждается DLL, необходимо выполнять на основании значения dwReason, как было показано в предыдущем примере. Инициализация процессов обычно ограничивается выделением ресурсов, совместно используемых потоками, в частности загрузкой разделяемых файлов и инициализацией библиотек. Инициализация потоков применяется для настройки режимов, свойственных только данному потоку, например для инициализации локальной памяти.

В состав DLL могут входить ресурсы, не принадлежащие вызывающему эту библиотеку приложению. Если функции DLL работают с ресурсами DLL, было бы, очевидно, полезно сохранить где-нибудь в укромном месте дескриптор hInst и использовать его при загрузке ресурсов из DLL. Указатель lpReserved зарезервирован для внутреннего использования Windows. Следовательно, приложение не должно претендовать на него. Можно лишь проверить его значение. Если библиотека DLL была загружена динамически, оно будет равно NULL. При статической загрузке этот указатель будет ненулевым.

В случае успешного завершения функция DllMain должна возвращать TRUE. В случае возникновения ошибки возвращается FALSE, и дальнейшие действия прекращаются.

Замечание. Если не написать собственной функции DllMain(), компилятор подключит стандартную версию, которая просто возвращает TRUE.

Экспортирование функций из DLL

Чтобы приложение могло обращаться к функциям динамической библиотеки, каждая из них должна занимать строку в таблице

экспортируемых функций DLL. Есть два способа занести функцию в эту таблицу на этапе компиляции.

Метод `__declspec (dllexport)`

Можно экспортировать функцию из DLL, поставив в начале ее описания модификатор `__declspec (dllexport)`. Кроме того, в состав MFC входит несколько макросов, определяющих `__declspec (dllexport)`, в том числе `AFX_CLASS_EXPORT`, `AFX_DATA_EXPORT` и `AFX_API_EXPORT`.

Метод `__declspec` применяется не так часто, как второй метод, работающий с файлами определения модуля (.def), и позволяет лучше управлять процессом экспортирования.

Файлы определения модуля

Синтаксис файлов с расширением .def в Visual C++ достаточно прямолинеен, главным образом потому, что сложные параметры, использовавшиеся в ранних версиях Windows, в Win32 более не применяются. Как станет ясно из следующего простого примера, .def-файл содержит имя и описание библиотеки, а также список экспортируемых функций:

```
MyDLL.def
LIBRARY      "MyDLL"
DESCRIPTION  'пример DLL-библиотеки

EXPORTS
    MyFunction @1
```

В строке экспорта функции можно указать ее порядковый номер, поставив перед ним символ @. Этот номер будет затем использоваться при обращении к `GetProcAddress ()`. На самом деле компилятор присваивает порядковые номера всем экспортируемым объектам. Однако способ, которым он это делает, отчасти непредсказуем, если не присвоить эти номера явно.

В строке экспорта можно использовать параметр `NONAME`. Он запрещает компилятору включать имя функции в таблицу экспортирования DLL:

```
MyFunction @1 NONAME
```

Иногда это позволяет сэкономить много места в файле DLL. Приложения, использующие библиотеку импортирования для неявного подключения DLL, не «замечают» разницы, поскольку при неявном подключении порядковые номера используются автоматически. Приложениям, загружающим библиотеки DLL динамически, потребуется передавать в GetProcAddress порядковый номер, а не имя функции.

При использовании вышеприведенного def-файла описания экспортируемых функций DLL-библиотеки может быть, например, не таким:

```
#define EXPORT extern "C" __declspec (dllexport)
EXPORT int CALLBACK MyFunction(char *str);
```

а таким:

```
extern "C" int CALLBACK MyFunction(char *str);
```

Экспортирование классов

Создание .def-файла для экспортирования даже простых классов из динамической библиотеки может оказаться довольно сложным делом. Понадобится явно экспортировать каждую функцию, которая может быть использована внешним приложением.

Если взглянуть на реализованный в классе файл распределения памяти, в нем можно заметить некоторые весьма необычные функции. Оказывается, здесь есть неявные конструкторы и деструкторы, функции, объявленные в макросах MFC, в частности _DECLARE_MESSAGE_MAP, а также функции, которые написанные программистом.

Хотя можно экспортировать каждую из этих функций в отдельности, есть более простой способ. Если в объявлении класса воспользоваться макромодификатором AFX_CLASS_EXPORT, компилятор сам позаботится об экспортировании необходимых функций, позволяющих приложению использовать класс, содержащийся в DLL.

Память DLL

В отличие от статических библиотек, которые, по существу, становятся частью кода приложения, библиотеки динамической компоновки в 16-разрядных версиях Windows работали с памятью несколько иначе. Под управлением Win 16 память DLL размещалась вне адресного пространства задачи. Размещение динамических библиотек в глобальной памяти обеспечивало возможность совместного использования их различными задачами.

В Win32 библиотека DLL располагается в области памяти загружающего ее процесса. Каждому процессу предоставляется отдельная копия "глобальной" памяти DLL, которая реинициализируется каждый раз, когда ее загружает новый процесс. Это означает, что динамическая библиотека не может использоваться совместно, в общей памяти, как это было в Win16.

И все же, выполнив ряд замысловатых манипуляций над сегментом данных DLL, можно создать общую область памяти для всех процессов, использующих данную библиотеку.

Допустим, имеется массив целых чисел, который должен использоваться всеми процессами, загружающими данную DLL. Это можно запрограммировать следующим образом:

```
#pragma data_seg(".myseg")
    int sharedInts[10] ;
    // другие переменные общего пользования
#pragma data_seg()
#pragma comment(lib, "msvcrt" "-SECTION:.myseg,rws");
```

Все переменные, объявленные между директивами #pragma data_seg(), размещаются в сегменте .myseg. Директива #pragma comment () не обычный комментарий. Она дает указание библиотеке выполняющей системы С пометить новый раздел как разрешенный для чтения, записи и совместного доступа.

Полная компиляция DLL

Если проект динамической библиотеки создан с помощью AppWizard и .def-файл модифицирован соответствующим образом этого достаточно. Если же файлы проекта создаются вручную или другими способами без помощи AppWizard, в командную строку редактора связей следует включить параметр /DLL. В результате вместо автономного выполняемого файла будет создана библиотека DLL.

Если в .def-файле есть строка LIBRARY, указывать явно параметр /DLL в командной строке редактора связей не нужно.

ЛИТЕРАТУРА

Библиотека БГТУ

1. Пацей, Н. В. Объектно-ориентированное программирование на C++/C#: учеб.-метод. пособие . В 2 ч., Ч. 1/ Н. В. Пацей – Минск.: БГТУ, 2014. – 191 с.

2. Смелов, В.В. Введение в объектно-ориентированное программирование на C++: учебн-метод. пособие – Минск.: БГТУ, 2009. – 94 с.

3. Пацей, Н. В. Технология разработки программного обеспечения: учебн-метод. пособие по курсовому проектированию / Н. В. Пацей, Д. В. Шиман, И.Г. Сухорукова – Минск.: БГТУ, 2011. – 130 с.

Электронные:

4. Прата, С. Язык программирования C++. Лекции и упражнения, 6-е изд. : Пер. с англ. / С. Прата – М. : ООО И.Д. Вильямс, 2012. – 1248 с.

5. Рихтер, Дж. CLR via C#. Программирование на платформе Microsoft .NET Framework 4.0 на языке C#. 3-е изд./ Дж. Рихтер – СПб.: Питер – 2012. – 928 с.

6. Шилдт, Г. C# 4.0: полное руководство.: Пер. с англ. – М. : ООО И.Д. Вильямс, 2011. – 1056 с.

7. Медведев, В.И. Особенности объектно-ориентированного программирования на C++/CLI, C# и Java. 2-е изд., испр. и доп. / В. И. Медведев – Казань: РИЦ «Школа», 2010. – 444 с.

8. Хортон, А. Visual C++ 2005. Базовый курс / А. Хортон. – М.: Вильямс, 2007. – 1152 с.

9. Харви, Д. Как программировать на C++ / Д. Харви, П. Дейтел. – М.: БИНОМ, 2007. – 1156 с.

10. Нейгел, К. C# 2005 для профессионалов.: Пер. с англ. – М.: Издательский дом "Вильямс", 2007 г.

11. Ноутон П. Java 2.: Пер. с англ. / П. Ноутон, Г. Шилдт - СПб.: БХВ-Петербург, 2007. – 1072 с..

12. Медведев, В.И. Программирование на C++, C++.NET/C# и .NET компоненты. 2-е издание. / В. И. Медведев: – Казань: Мастер Лайн, 2007. – 296 с.

13. Медведев, В.И. .NET компоненты, контейнеры и удалённые объекты / В. И. Медведев: – Казань: РИЦ «Школа», 2006. – 320 с.

14. Александреску, А. Современное проектирование на C++ (серия C++ in Depth) /А. Александреску. – М.: Издательский дом Вильямс, 2008. – 336с

15. Влссидес, Дж. Применение шаблонов проектирования. Дополнительные штрихи / Дж. Влссидес – М.: Издательский дом "Вильямс", 2003. – 144с

16. Гамма, Э., Хелм Р., Джонсон Р., Влссидес Дж. Приемы объектно-ориентированного проектирования. Паттерны проектирования. / Э. Гамма, Р. Хелм, Р. Джонсон , Дж. Влссидес. - СПб.: Питер, 2010. – 368с .