Литература

1. Страуструп Б. Язык программирования С++. – М.: Бином, 2011. – 1136 с.
2. Страуструп Б. Программирование. Принципы и практика с использованием С++. – М.: Вильямс, 2018. – 1328 с.
3. Подбельский В.В. Стандартный Си++ : учебное пособие. - М.: Финансы и статистика, 2014.- 687 с.
4. Джосаттис Н.М. Стандартная библиотека С++: справочное руководство, 2-е изд. – М.: ИД Вильямс, 2017. – 1136 с.
5. Уильямс Э. Параллельное программирование на С++ в действии. Практика разработки многопоточных программ.- М.: ДМК Пресс, 2016.-672 с.
6. Мэйерс С. Эффективное использование С++. 55 верных советов улучшить структуру и код ваших программ. – М.: ДМК Пресс, 2014. – 300 с.
7. Быков А.Ю. Решение задач на языках программирования Си и Си++: методические указания к выполнению лабораторных работ. – М.: МГТУ им. Н.Э. Баумана, 2017. - 244с. Режим доступа: https://bmstu.press/catalog/item/4673

Тема № 10. Основы объектно-ориентированного программирования (ООП) в Си++

Введение. Историческая справка

Язык Си++ появился в 80-х годах первоначально назывался «Си с классами», название Си++ появилось в 1983 г. (в Си существует операция ++ (инкремент – увеличение на 1). Один из создателей Бьёрн Страуструп.

Стандарты языка:

- ISO/IEC 14882:1998 “Standard for the C++ Programming Language”
- ISO/IEC 14882:2003 (исправления)
- C++ Technical Report 1 (TR1) 2005 г. ISO/IEC TR 19768, библиотека расширений C++ (проект документа)
- ISO/IEC 14882:2011 (C++11)
- C++ ISO/IEC JTC1 («*International Standard ISO/IEC 14882:2014(E) Programming Language C++*») (C++ 14)
- ISO/IEC 14882:2017 (C++ 2017)
- ISO/IEC 14882:2020 (C++ 2020)

Некоторые новшества в языке

1. При определении переменной не обязательно указывать тип, если он определяется по инициализатору, указывается тип auto.

auto a=10; auto x = 10.5;

2. C++11 вводит ключевое слово constexpr, которое позволяет пользователю гарантировать, что или функция или конструктор объекта возвращает константу времени

компиляции.

```
constexpr int GiveFive() { return 5; }
int some_value[GiveFive() + 7];
```

3. C++11 позволяет вызывать одни конструкторы класса из других

4. Цикл for по набору значений (range base loop):

```
int m[] = { 1, 4, 10, 20, 8 };
for (int k : m) // Для каждого k в массиве m
cout << endl<< k;  // Печатается весь массив
double X[] = { 1.3, 4.5, 5.6, 5.7 };
for (double y : X) cout << endl << y;
```

5. Новый способ задания недействительного указателя (C++11):

```
int *p=nullptr;
```

6. Правосторонняя ссылка (разрешено использовать только справа от операции присваивания, слева нельзя) *T &&* (C++ 11).

7. Введено пространство имен - это группа имен, в которой имена не совпадают.

```
namespace  name
    {
      // объявления и определения имен
   int a;
    }
```

За пределами пространства имен обращаемся к именам или *name::a=10;*

Пространства имен могут быть вложенными, в этом случае обращаемся к объекту (переменной, функции, классу и др.): *name1::name2::name3::a=10*;

Или используем директиву:

```
using  namespace  name;
```

Далее просто по имени *a=10;*

8. Заголовочные файлы без расширения h.

stdio.h – старый cstdio – новый

9. Целые константы в двоичной форме (C++ 14): префиксы 0b или 0B `int a = 0b11100011;`

10.1. Понятие об ООП

Объектно-ориентированное программирование

Суть – вводится новый тип – класс, включающий данные (поля класса) и методы (функции) для обработки этих данных. Переменные этого типа называются объектами.

Основные принципы ООП:

- Инкапсуляция (объединение в одном типе данных и методов)
- Наследование классов (создание новых классов на базе существующих)
- Полиморфизм (переопределение методов в классах- наследниках)

10.2. Классы и объекты в Си++

Классы

```
<Ключевое слово> <Имя_класса>
{
      <список компонент>
};
```

Ключевые слова:

- struct
- class
- union

Класс это структура, в которую введены методы для обработки полей.

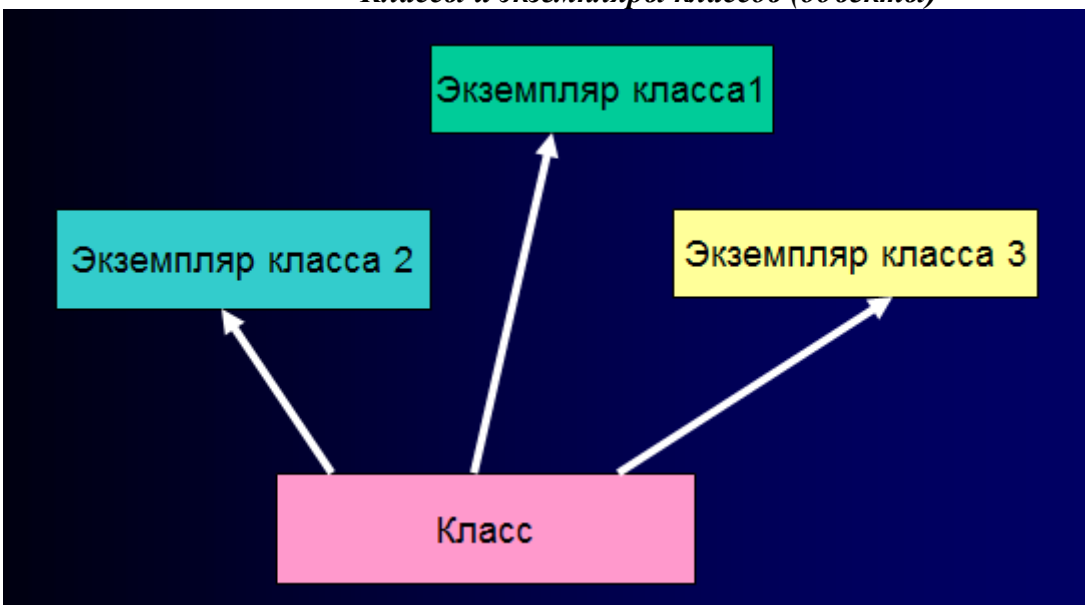
Объекты (переменные класса)

<Имя_класса> <Имя_объекта1>,…<Имя_объекта_N>;

Обращение к полям и методам класса внутри методов класса просто по имени, а за пределами класса через имя объекта и операцию «.» или через имя указателя на объект и операцию «->». Каждый объект класса имеет в оперативной памяти свои копии полей класса.

struct A { int i; void print() { printf("i=%d", i); } };
*A a1; A *pA=&a1;*
a1.i=10; a1.print(); pA->i=10; pA->print();
a1.A::i=10; a1.A::print();
pA->A::i=10; pA->A::print();

Классы и экземпляры классов (объекты)



10.3. Доступность компонент класса

Свойство доступности определяет возможность доступа к полям и методам за пределами класса (через имя объекта или через указатель на объект или в производном классе).

Существуют три статуса доступа:

- public (полностью доступны за пределами класса);
- private (не доступны за пределами класса можно обращаться к компонентам только в методах своего класса);
- protected (доступны только в своем классе и в производных классах).

По умолчанию, если класс определен с ключевым словом struct, то все компоненты имеют статус доступа public. Если с ключевым словом union, тоже public, но все поля

каждого объекта располагаются в памяти, начиная с одного адреса. Если класс определен с ключевым словом class, то все поля и методы по умолчанию имеют статус доступа private.

Статус доступа можно изменить с помощью соответствующих модификаторов.

```
struct A
{
  …… // Статус доступа public
private:
……. // Статус доступа private
protected:
…….// Статус доступа protected
};
class B
{
  …… // Статус доступа private
public:
……. // Статус доступа public
protected:
…….// Статус доступа protected
};
```

10.4. Основные элементы класса

10.4.1. Компонентные данные и функции (методы) класса

```
struct Complex
{
      double real,       image; // Поля класса
      void define(double re=0.0, double im=0.0) // Определение метода
        // внутри класса ,   метод  будет  подставляемым
      {
        real=re; image=im;// Обращение к полям внутри метода
      }
      void print(); // Описание метода
};
void Complex::print() // Определение метода за пределами класса
{
        printf("\nreal=%f  image=%f", real, image);
}
```

Возможна перегрузка методов.

Чаще всего класс объявляется в 2-х файлах:

- заголовочный файл с расширением .h, содержит описание класса с заголовками методов;
- файл реализации с расширением .cpp, содержит определения методов за пределами класса.

При необходимости использовать класс в каком-либо файле проекта, к этому файлу требуется подключить заголовочный файл с описанием класса.

В последних допустимо объявление полей класса с инициализацией (в ранних версиях это было запрещено).

Инициализация выполняется до выполнения конструктора. Функция класса может

иметь модификатор *const*, это означает, что она не может изменять поля класса, кроме полей с модификатором *mutable*

```
class point
{
	int x=-10;
	mutable int y=-10;

public:

void print() const;

……

};

void point::print() const // Определение метода за пределами класса с модификатором const
{
	cout<<"\nx="<<x<<" y="<<y;
	y = 10; // Было бы запрещено, если бы y был без mutable
}
```

10.4.2. Конструктор класса

Конструктор класса специальный блок операторов (инструкций), вызываемый при создании объекта. Назначение: присвоение нач. значений полям, выделение памяти, открытие файлов, сетевых соединений и т.п. Имя конструктора совпадает с именем класса, конструктор не имеет возвращаемого значения. Возможна перегрузка конструкторов. Конструктор может определяться как внутри класса, так и за пределами.

Формат определения конструктора внутри класса:
Имя_класса(Список_формальных_параметров)
{ Операторы_тела конструктора }

По умолчанию класс всегда имеет конструктор копирования вида A(const A& a) {… } (A – имя класса), создающий копию объекта (происходит копирование полей), и, если нет ни одного явного конструктора, то по умолчанию создается конструктор без параметров. Эти конструкторы можно переопределять. Конструктор копирования по умолчанию можно удалить, для этого в классе заголовок конструктора объявляется с ключевым словом `delete` без тела, например, A(const A& a)=`delete`;

Если нет явного конструктора копирования, то конструктор копирования по умолчанию будет удален при наличии явного конструктора перемещения (см. ниже) или оператора копирования с перемещением.

Примеры вызовов конструкторов:
A a1; A pA=new A;* // Вызываются конструкторы без параметров
*A a2(3, 4); A * pA2=new A(3, 4);*
// Вызываются конструкторы с 2-мя параметрами

Для конструктора с одним параметром можно использовать форму:
A a1=5; A a2=a1; вместо *A a1(5); A a2(a1);*

Ключевое слово `explicit` у конструктора с одним параметром запрещает это преобразование, допустима явная форма: *A a1(5); A a2(a1);*
Пример:

```
class point
{
	int x, y;
public:
	point(int, int); // Конструктор с 2-мя параметрами
	explicit point(int); // Конструктор с 1-м параметром
	point() : x(0), y(0)  // Можно инициализировать поля так
	{ // Конструктор без параметров
		// x=0; y=0;	// Или так
		cout << endl << "point()";
	}
	void print();
	~point()
	{
		printf("\nDestructor");
	}
};
point::point(int a, int b) // Определение конструктора за пределами класса
	: x(a), y(b) // Можно так
{
//	x=a; y=b; // Или так
}

point::point(int a) // Определение конструктора за пределами класса
{
	x=a; y=0;
}
void point::print() // Определение метода за пределами класса
{
	cout<<"\nx="<<x<<" y="<<y;
}
int main(int argc, char* argv[])
{
	point *pP=new point[10]; // Массив указателей на объеты, создается 10 объектов,
вызываются конструкторы без параметров
	point p1(12, 13);
	p1.print();
	point p2; // Вызов конструктора без параметров
	point p3=100; // point p3(100); Вызов конструктора с 1-м параметром Ошибка, надо
			point p3(100);   так как есть explicit у конструктора
	p2.print();
	p3.print();
	point p4=p1; // point p4(p1);  Вызов конструктора копирования
	p4.print();
	delete [] pP;
//	free(pP); Так не верно
	return 0;
}
```

10.4.3. Деструктор класса

Деструктор — специальный блок операторов (инструкций), служащий для деинициализации объекта (освобождение памяти, закрытие файлов и т.п.).

Вызывается автоматически при удаление объекта, например, оператором delete или при выходе из блока, в котором существует объект. Не имеет возвращаемого значения и параметров. Может определяться как внутри класса, так и за пределами.

~имя_класса()
{
  тело_деструктора
}

10.4.4. Конструктор перемещения (move constructor)

В отличие от конструктора копирования (этот конструктор «работает», как правило, быстрее), исходный объект уже не нужен, но его нужно привести в форму, подходящую для вызова деструктора, чтобы «не испортить» новый объект.

Пример:

Vec(Vec&& v) // Параметр - правосторонняя ссылка
// : p(v.p), len(v.len) ***Можно инициализировать так***
{
p = v.p;
len = v.len;
// Присвойте данным-членам исходного объекта значения по умолчанию.
// Это не позволяет деструктору многократно освобождать память
v.p = nullptr;
v.len = 0;
}

Вызов конструктора: *Vec V2=std::move(V1);*

Пример:

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

class Massiv
{
public:
    int *p=nullptr; // Изначально "пустой" объект
    int n=0;
    Massiv(int *pp, int nn);
    Massiv(const Massiv &M);
    // Конструктор перемещения
    Massiv(Massiv&& M);  // Параметр - правосторонняя ссылка
    ~Massiv(); // Описание деструктора
    void print() const;
};
Massiv::~Massiv()
{
    if (p != nullptr) delete[]p;
    std::cout << "Destryctor\n";
}
Massiv::Massiv(const Massiv & M)
{ // Новый конструктор копирования
    std::cout <<"Constr Copy\n";
    n = M.n;
    p = new int[n];
    for (int i = 0; i<n; i++)
```

```
        p[i] = M.p[i];
}

// Конструктор перемещения (вариант: делаем все явно)
/*Massiv::Massiv(Massiv&& M)  // Параметр - правосторонняя ссылка
    : p(M.p), n(M.n)    // Можно инициализировать так
{
    std::cout << "Constr Move\n";
    //p = M.p;  // Или инициализировать так
    //n = M.n;
    // Присвойте данным-членам исходного объекта значения по умолчанию. Это не
позволяет деструктору многократно освобождать память
    M.p = nullptr;
    M.n = 0;
}*/

// Конструктор перемещения (более короткий вариант)
Massiv::Massiv(Massiv&& M)  // Параметр - правосторонняя ссылка
{
    std::cout << "Constr Move\n";
    std::swap(p, M.p);
    std::swap(n, M.n);
}

Massiv::Massiv(int *pp, int nn)
{
    std::cout << "Constr\n";
    n = nn;
    p = new int[n];
    for (int i = 0; i<n; i++) p[i] = pp[i];
}

void Massiv::print() const
{
    if (p!=nullptr)
    for (int i = 0; i < n; i++)
        std::cout << p[i] << " ";
    else std::cout<<"Null Object";
    std::cout << "\n";
}

int main()
{
    int m[] = { 1, 2, 3, 4, 5, 6, 7 };
    Massiv M1(m, 7);
    M1.print();
    Massiv M2 = M1; // Massiv M2(M1); Вызывается конструктор копирования
    M2.print();
    M1.p[0] = 1000;
    M1.print();
    M2.print();
```

```
    {
        Massiv M3 = std::move(M1); // Вызов конструктора перемещения
        M3.print();
    } // При выходе из блока возникла бы проблема потерянной ссылки (указателя)
или утечки памяти, если бы не было деструктора
    Massiv *pM = new Massiv(M1);// Конструктор копирования
    delete pM; // Автоматически вызывается деструктор
    return 0;
}
```

Если нет явного конструктора перемещения и копирования, то конструктор перемещения создается по умолчанию. Если есть явный конструктор копирования, то конструктор перемещения не создается, вместо него используется этот явный конструктор копирования. Конструктор перемещения также можно явно удалить, для этого в классе заголовок конструктора объявляется с ключевым словом `delete` без тела, например,
A(A&& a)=`delete`;

10.5. Указатель this

Иногда при вызове нестатического метода требуется обращаться к адресу объекта, для которого вызывается метод. Но когда программист пишет код метода класса, объектов еще не существует. Для этих целей любой нестатический метод и конструкторы (деструктор) неявно (по умолчанию) получает указатель this – указатель на объект класса, для которого вызывается метод.

Первый случай использования this

```
class point {
        int x,y;
        public:
        point(int x, int y)
        {
            this->x=x;   this->y=y;
        }
        .....
};
```

Второй случай использования this

```
class List // Линейный односвязный список
{
        int x; // инф. поле
        List *pNext; // Указатель на след. элемент
public:
        List(int x)
        {
                this->x=x;
        }
        void add(List *&pF) // Добавить элемент в начало списка
        {
```

```
                pNext=pF;
                pF=this; // Текущий элемент будет первым
        }
……….
};
```

Пример:

```
#include <iostream>
using namespace std;
class List // Линейный односвязный список
{
        int x; // инф. поле
        List *pNext; // Указатель на след. элемент
public:
        List(int x)
        {
                this->x=x;
        }
        void add(List *&pF) // Добавить элемент в начало списка
        {
                pNext=pF;
                pF=this; // Текущий элемент будет первым
        }
        static void print(List *pF)
        {
                List *pi=pF;
                while(pi)
                {
                        cout<<endl<<pi->x;
                        pi=pi->pNext;
                }
        }
};

int main(int argc, char* argv[])
{
        List *pF=0, *p;
        for(int i=0; i<10; i++)
        {
                p=new List(i+1);
                p->add(pF);
        }

        List::print(pF);
        return 0;
}
```

10.6. «Друзья» классов

Существует способ доступа к защищенным или собственным полям и методам класса за пределами класса. С помощью «друзей» класса (дружественных функций или дружественных классов). Они должны быть описаны внутри класса с модификатором friend.

10.6.1. Дружественные функции класса

Особенности ДФ:

– ДФ не получает ук-ль this, т.к. она не принадлежит классу;

- дружественная функция получает доступ к объекту через формальный параметр, объект класса передается по указателю или по ссылке;
- место размещение прототипа ДФ внутри класса безразлично, на нее не распространяется действие модификаторов доступа;
- ДФ может быть компонентной функцией другого класса;
- ДФ может быть дружественной по отношению к нескольким классам.

10.6.2. Дружественные классы

class C1 { friend class C2; ... };
class C2 { .... }

Все функции класса C2 являются дружественными по отношению к классу C1.

Пример:

```
#include <iostream>
using namespace std;
class point
{
	int x, y;
	public:
		point(int a, int b)
		{
			x=a; y=b;
		}
	friend void print(point &p); // Заголовок ДФ внутри класса
//	friend void A::fun(point &p); // Если ДФ принадлежит классу A
};

void print(point &p) // Дружественная функция
{
	cout<<endl<<"x="<< p.x<<" y="<<p.y; // Имеет доступ к закрытым полям (private)
}
int main(int argc, char* argv[])
{
	point p1(3, 5);
	print(p1); // Вызов ДФ
	return 0;
}
```

10.7. Перегрузка стандартных операций

Перегрузка операций в языке Си++ это возможность распространения действия стандартных операций на операнды, для которых эти операции первоначально не предназначались. Это возможно, если хотя бы один из операндов является объектом класса, для этого создает специальная, так называемая, оператор- функция.

Формат оператор- функции

<тип_возвращаемого_значения> operator <знак_операции> (спецификация_параметров)
{
операторы_тела_функции
}

Два способа перегрузки:

- оператор-функция определяется как функция непринадлежащая классу, при необходимости может быть дружественной;
- оператор-функция определяется как функция класса.

Особенности перегрузки операций:

- можно перегружать только стандартные операции, например, нельзя перегрузить **;
- не допускают перегрузки ‘.’, ‘.*’, ‘?:’, ‘::’, ‘sizeof’, ‘#’, ‘##’;
- сохраняется арность операций;
- бинарная операция перегружается либо как функция, не принадлежащая классу с двумя параметрами, один обязательно объект (ссылка на объект) класса, или как функция класса с одним параметром, первым операндом операции выступает объект класса, для которого вызывается функция;
- бинарные операции ‘=‘, ‘[]’, ‘->’ должны обязательно определяться как компонентные функции класса;
- унарная операция перегружается либо как функция, не принадлежащая классу с одним параметром - объектом (ссылкой на объект) класса, или как функция класса без параметров, операндом операции выступает объект класса, для которого вызывается функция.

Примечание 1. Исключение составляет перегрузка операций ++ и --, чтобы отличить постфиксную форму от префиксной, вводится формальный параметр типа int.

Пример объявления операторов-функций внутри класса:

```
Vector & operator++() // Префиксная форма операции ++

Vector operator++(int)  // Постфиксная форма операции ++ всегда есть формальный (фиктивный)

                        // параметр типа int
```

Пример объявления операторов-функций за пределами класса:

```
Vector & operator++(Vector & V) // Префиксная форма операции ++

Vector operator++(Vector & V, int)  // Постфиксная форма операции ++ всегда есть формальный

                                    // (фиктивный) параметр типа int
```

В постфиксной форме вначале с помощью конструктора копирования создаем новый объект, в котором сохраняется начальное (старое) состояние текущего объекта, затем текущий объект изменяется, и возвращается созданный объект.

Примечание 2. По аналогии с конструкторами копирования и перемещения по умолчанию существуют в классе операторы присваивания с копированием и присваивания с перемещением. Правила создания и использования аналогичные. Эти операторы можно переопределять или удалять, например,

```
A & operator=(const A & Ob) = delete;

A & operator=(A && Ob) = delete;
```

Пример:

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

```
{
    int *p=nullptr;
    int n=0;
public:
    Vector() // "Пустой" объект
    {
        n = 0;
        p = nullptr;
    }
    ~Vector() { if (p != nullptr) delete[] p;
       cout <<  "Destructor"<<endl;
    } // Деструктор
    Vector(int *p, int n) // Создаем вектор на основе обычного массива
    {
        this->n = n;
        this->p = new int[n];
        for (int i = 0; i<n; i++) this->p[i] = p[i];
    }
    Vector(int n): n(n) // Выделяем память без инициализации
    {
        p=new int[n];
    }

    void print() const // Функция печати
    {
        if (p!=nullptr) for (int i = 0; i<n; i++)
            cout << p[i] << ' ';
        else cout << "Empty";
        cout << endl;
    }
    // Перегрузка операции обращение по индексу
    int & operator[](int index) // ссылка позволяет изменять значение элемента
    {
        return p[index]; // Возвращаем элемент по индексу
    }
    int operator[](int index) const // Изменять значение нельзя
    {
        return p[index]; // Возвращаем элемент по индексу
    }
    Vector & operator++() // Префиксная форма операции ++
    {
        for (int i = 0; i < n; i++) ++p[i];
        return *this;
    }
    Vector operator++(int)  // Постфиксная форма операции ++ всегда есть
формальный (фиктивный) параметр типа int
    {
        Vector temp (*this); // Создаем новый объект (старое состояние текущего
объекта)
        for (int i = 0; i < n; i++) ++p[i]; // Инкремент текущего объекта
        return temp; // Возвращаем старое состояние объекта
```

```
    }
    Vector(const Vector & V) // Конструктор копирования
    {
        // Задаем новый объект
        n = V.n;
        p = new int[n];
        for (int i = 0; i<n; i++) p[i] = V.p[i];
        cout << "Constructor Copy"<<endl;
    }
    Vector(Vector && V) // Конструктор перемещения
    {
        // Короткая форма
        swap(p, V.p);
        swap(n, V.n);
        cout << "Constructor Move"<<endl;
    }
    Vector & operator=(const Vector & V2) // Оператор присваивания с
копированием, параметр второй операнд, первый операнд - объект текущего класса
    {
        if (this!=&V2) // Запрет копирования объекта самого в себя
        {
            if (p != nullptr) delete[] p; // Очищает (удаляем) старый
объект,если он есть
            // Задаем новый объект
            n = V2.n;
            p = new int[n];
            for (int i = 0; i<n; i++) p[i] = V2.p[i];
        }
        cout << "operator= copy"<<endl;
        return *this; // Возвращаем текущий объект
    }
    Vector & operator=(Vector && V2) // Операторор присваивания с перемещением,
Параметр второй операнд, первый операнд - объект текущего класса
    {
        if (this!=&V2) // Запрет перемещения объекта самого в себя
        {
        /*  // Все делаем явно
        if (p != nullptr) delete[] p; // Очищает (удаляем) старый объект,если он
есть
        // Задаем новый объект
        n = V2.n;
        p = V2.p;
        V2.n = 0; V2.p = nullptr;*/
            // Короткая форма
            swap(p, V2.p);
            swap(n, V2.n);
        }
        cout << "operator= move"<<endl;
        return *this; // Возвращаем текущий объект
    }
    // Умножение вектора на число
```

```
    // на выходе новый вектор
    Vector operator*(int x) const
    {
        Vector V(n);
        for (int i = 0; i<n; i++)
            V[i] = p[i] * x;
        return V;
    }
    friend Vector operator*(int x, const Vector& v2);
    friend double operator*(const double *p1, const Vector &Ob2); // Чтобы
получить доступ к закрытым полям класса
//      friend Vector & operator++(Vector & V); // Префиксная форма операции ++
//      friend Vector operator++(Vector & V, int);  // Постфиксная форма
операции ++ всегда есть формальный (фиктивный) параметр типа int
};
// Умножение числа на вектор, так как 1 операнд не объект класса, то оператор-
функция не принадлежит классу
Vector operator*(int x, const Vector& v2)
{
    Vector V(v2.n);
    for (int i = 0; i<v2.n; i++)
        V[i] = x*v2[i];
     return V;
}
// Скалярное произведение векторов
// Оператор-функция обязательно не принадлежит классу, так как первый операнд не
объект класса
double operator*(const double *p1, const Vector &Ob2)
{
    double sum = 0;
    for (int i = 0; i<Ob2.n; i++)
        sum += p1[i] * Ob2.p[i];
    return sum;#include <iostream>
using namespace std;
class Vector
{
    int *p=nullptr;
    int n=0;
public:
    Vector() // "Пустой" объект
    {
        n = 0;
        p = nullptr;
    }
    ~Vector() { if (p != nullptr) delete[] p;
       cout <<  "Destructor"<<endl;
    } // Деструктор
    Vector(int *p, int n) // Создаем вектор на основе обычного массива
    {
        this->n = n;
        this->p = new int[n];
```

```
        for (int i = 0; i<n; i++) this->p[i] = p[i];
    }
    void print() const // Функция печати
    {
        if (p!=nullptr) for (int i = 0; i<n; i++)
            cout << p[i] << ' ';
        else cout << "Empty";
        cout << endl;
    }
    // Перегрузка операции обращение по индексу
    int & operator[](int index) // ссылка позволяет изменять значение элемента
    {
        return p[index]; // Возвращаем элемент по индексу
    }
    Vector & operator++() // Префиксная форма операции ++
    {
        for (int i = 0; i < n; i++) ++p[i];
        return *this;
    }
    Vector operator++(int)  // Постфиксная форма операции ++ всегда есть
формальный (фиктивный) параметр типа int
    {
        Vector temp (*this); // Создаем новый объект (старое состояние текущего
объекта)
        for (int i = 0; i < n; i++) ++p[i]; // Инкремент текущего объекта
        return temp; // Возвращаем старое состояние объекта
    }
    Vector(const Vector & V) // Конструктор копирования
    {
        // Задаем новый объект
        n = V.n;
        p = new int[n];
        for (int i = 0; i<n; i++) p[i] = V.p[i];
        cout << "Constructor Copy"<<endl;
    }
    Vector(Vector && V) // Конструктор перемещения
    {
        // Короткая форма
        swap(p, V.p);
        swap(n, V.n);
        cout << "Constructor Move"<<endl;
    }
    Vector & operator=(const Vector & V2) // Оператор присваивания с
копированием, параметр второй операнд, первый операнд - объект текущего класса
    {
        if (this!=&V2) // Запрет копирования объекта самого в себя
        {
            if (p != nullptr) delete[] p; // Очищает (удаляем) старый
объект,если он есть
            // Задаем новый объект
            n = V2.n;
```

```
            p = new int[n];
            for (int i = 0; i<n; i++) p[i] = V2.p[i];
        }
        cout << "operator= copy"<<endl;
        return *this; // Возвращаем текущий объект
    }
    Vector & operator=(Vector && V2) // Операторор присваивания с перемещением,
Параметр второй операнд, первый операнд - объект текущего класса
    {
        if (this!=&V2) // Запрет перемещения объекта самого в себя
        {
        /*  // Все делаем явно
        if (p != nullptr) delete[] p; // Очищает (удаляем) старый объект,если он
есть
        // Задаем новый объект
        n = V2.n;
        p = V2.p;
        V2.n = 0; V2.p = nullptr;*/
            // Короткая форма
            swap(p, V2.p);
            swap(n, V2.n);
        }
        cout << "operator= move"<<endl;
        return *this; // Возвращаем текущий объект
    }
    // Умножение вектора на число
    // на выходе новый вектор
    Vector& operator*(int x)
    {
        int *pM = new int[n];
        for (int i = 0; i<n; i++)
            pM[i] = p[i] * x;
        Vector *pV = new Vector(pM, n);
        delete[] pM;
        return *pV;
    }
    friend Vector& operator*(int x, Vector& v2);
    friend double operator*(double *p1, Vector &Ob2); // Чтобы получить доступ к
закрытым полям класса
//      friend Vector & operator++(Vector & V); // Префиксная форма операции ++
//      friend Vector operator++(Vector & V, int);  // Постфиксная форма
операции ++ всегда есть формальный (фиктивный) параметр типа int
};
// Умножение числа на вектор, так как 1 операнд не объект класса, то оператор-
функция не принадлежит классу
Vector& operator*(int x, Vector& v2)
{
    int *pM = new int[v2.n];
    for (int i = 0; i<v2.n; i++)
        pM[i] = x*v2[i];
    Vector *pV = new Vector(pM, v2.n);
```

```
    delete[] pM;
    return *pV;
}
// Скалярное произведение векторов
// Оператор-функция обязательно не принадлежит классу, так как первый операнд не
объект класса
double operator*(double *p1, Vector &Ob2)
{
    double sum = 0;
    for (int i = 0; i<Ob2.n; i++)
        sum += p1[i] * Ob2.p[i];
    return sum;
}
// Пример перегрузки операторов ++ вне класса
/*Vector & operator++(Vector & V) // Префиксная форма операции ++
{
    for (int i = 0; i < V.n; i++) ++V.p[i];
    return V;
}
Vector operator++(Vector & V, int)  // Постфиксная форма операции ++ всегда есть
формальный (фиктивный) параметр типа int
{
    Vector temp(V); // Создаем новый объект (старое состояние текущего объекта)
    for (int i = 0; i < V.n; i++) ++V.p[i]; // Инкремент текущего объекта
    return temp; // Возвращаем старое состояние объекта
}*/
int main()
{
    int m1[] = { 1, 2, 3, 4, 5 };
    Vector V1(m1, 5); // Создаем объект на основе обычного массива
    V1.print();
    V1[0] = 100;
    V1.print();
    Vector V2;
    V2 = 2 * V1; // V2=operator*(2, V1);
    V2.print();
    Vector V3;
    //V3 = move(V1++); // Чтобы не создавать лишние объекты используем оператор
перемещения
    V3 = V1++; // Здесь тоже вызывается оператор = с перемещением Чтобы не
создавать лишние объекты
    V1.print();
    V3.print();
    return 1;
}
}
// Пример перегрузки операторов ++ вне класса
/*Vector & operator++(Vector & V) // Префиксная форма операции ++
{
    for (int i = 0; i < V.n; i++) ++V.p[i];
    return V;
```

```
}
Vector operator++(Vector & V, int)  // Постфиксная форма операции ++ всегда есть
формальный (фиктивный) параметр типа int
{
    Vector temp(V); // Создаем новый объект (старое состояние текущего объекта)
    for (int i = 0; i < V.n; i++) ++V.p[i]; // Инкремент текущего объекта
    return temp; // Возвращаем старое состояние объекта
}*/
int main()
{
    int m1[] = { 1, 2, 3, 4, 5 };
    Vector V1(m1, 5); // Создаем объект на основе обычного массива
    V1.print();
    V1[0] = 100;
    V1.print();
    Vector V2;
    V2 = 2 * V1; // V2=operator*(2, V1);
    V2.print();
    Vector V3;
    //V3 = move(V1++); // Чтобы не создавать лишние объекты используем оператор
перемещения
    V3 = V1++; // Здесь тоже вызывается оператор = с перемещением Чтобы не
создавать лишние объекты
    V1.print();
    V3.print();
    return 1;
}
```

Правило трёх (также известное как «Закон Большой Тройки» или «Большая Тройка») - правило в C++, гласящее, что если класс или структура определяет один из следующих методов, то они должны явным образом определить все три метода:

- Деструктор
- Конструктор копирования
- Оператор присваивания копированием

С выходом одиннадцатого стандарта правило расширилось и теперь называется правило пяти. Теперь при реализации конструктора необходимо реализовать:

- Деструктор
- Конструктор копирования
- Оператор присваивания копированием
- Конструктор перемещения
- Оператор присваивания перемещением

10.8. Статические компоненты класса

Статические поля

Память под обычные поля (нестатические) выделяется при создание объекта, каждый

объект класса имеет свои копии обычных полей.

Статические поля объявляются в классе с модификатором static, память под статические поля выделяется при определении класса, и не выделяется при создание объектов. Статические поля существуют в единственном экземпляре независимо от того, сколько объектов создано, т.е. это поля класса, они общие для всех объектов. К статическим полям за пределами класса можно обращаться через имя объекта, как к обычным полям, но чаще к ним обращаются через имя класса (это можно делать даже когда объекты класса не созданы). Статические поля обязательно требуют инициализации (за пределами класса).

Статические методы

На статические поля распространяются модификаторы статуса доступа. Для доступа к собственным (private) или защищенным (protected) статическим полям за пределами класса служат открытые статические методы, которые определяются с модификатором static.

К статическим методам за пределами класса обращаются как к статическим полям через имя объекта или чаще через имя класса. Внутри статических методов можно обращаться только к статическим полям или вызывать другие статические методы. К обычным полям и методам обращаться нельзя, так как при вызове статического метода объекты могут быть еще не созданы (статический метод относится ко всему классу).

Пример:

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

class A
{

	static int N;// Счетчик созданных объектов
public:
	A()
	{
		N++;
	}
	~A() { N--; }
	static int getN();

};

 int A::getN()
	{
		return N;
	}

int A::N=0; // Обязательная инициализация статич поля

int main(int argc, char* argv[])
{
	cout << "N=" << A::getN();
	A a1, a2, a3, a4;
	cout<<"\nN="<< a1.getN();
	{
		A a5;
		cout << "\nN="<< a5.getN();
	}
	cout << "\nN="<< A::getN();
	return 0;
```

}

10.9. Указатели на компоненты класса

Операции ‘.*’ и ‘->*’ предназначены для работы с указателями на компоненты класса. До использования указателя его необходимо соответствующим образом настроить.

10.9.1. Указатели на компонентные функции (методы)

Формат определения указателя на функцию:

<тип_возвр_значения> (<имя_класса>::<имя_указателя>)(<спецификация_форм_параметров>);*

Настройка указателя:

<имя указателя>= &<имя_класса>::<имя_функции>;

Вызов функции по указателю:

(<имя_объекта>.<имя_указателя>)(<параметры>);*
(<имя_ук_на_об>-><имя_указателя>)(<параметры>);*

10.9.2. Указатели на поля

Формат определения указателя на поле класса:

<тип_поля> (<имя_класса>::<имя_указателя>);*

Настройка указателя:

<имя указателя>= &<имя_класса>::<имя_поля>;

Обращение к полю по указателю:

<имя_объекта>.<имя_указателя>=10;*
<имя_ук_на_об>-><имя_указателя>=10;*

Пример:

```
#include <iostream>
using namespace std;
struct point
{
	int x, y;
	void printX()
	{
		cout<<endl<<"x="<<x;
	}
	void printY()
	{
		cout<<endl<<"y="<<y;
	}
};

int main(int argc, char* argv[])
{
	void (point::*pF)()=&point::printX; // Указатель на функцию класса
	int point::*p=&point::x; // Указатель на поле класса
	point p1;
//	p1.x=5;
//	p1.y=10;
	p1.*p=5; // Обращение к полю x через указатель
	p=&point::y; // Указатель настраиваем на поле y
	p1.*p=10; // Обращение к полю y через указатель
	(p1.*pF)(); // Вызываем функцию printX через указатель
```

```
	pF=&point::printY;// Указатель на функцию класса настраиваем на функцию printY
	(p1.*pF)(); // Вызываем функцию printY через указатель

	return 0;
}
```

10.10 Локальные классы

Локальный класс – класс, определенный внутри блока.

- локальный класс не может иметь статических данных;
- его компонентные функции могут быть только встроенными;
- внутри класса можно использовать любые внешние переменные, функции, которые можно использовать внутри блока, за исключением переменных класса памяти auto.

10.11. Шаблоны классов

Шаблон класса задает семейство классов, в которых могут быть использованы разные типы.

template <список_параметров_шаблона>
<определение_шаблона_класса>

Имя параметра шаблона – это имя неизвестного заранее типа. При создании объекта указывается имя конкретного типа и по шаблону генерируется класс.

Существует библиотека стандартных шаблонов (STL- Standard Template Library), вошла в стандарт C++ 11

В ней шаблонами задаются контейнеры – предназначенные для хранение набора объектов в памяти (объекты могут быть любых типов). Контейнеры: vector, list, set, stack, queue.

Пример:

```
#include <iostream>
using namespace std;
template <class T> // Шаблон класса вектор (массив)
// T - тип элементов вектора
class vektor
{
	T* data;// Указатель на массив
	int size;// Размерность
	public:
		vektor(int n=10);
		~vektor() { delete [] data; }
		T& operator[] (int i) // Оператор функция возвращает элемент по индексу
		{
			return data[i];
		}
		T getSum();  // Функция считает сумму элементов

};
// Внешнее определение конструктора
template <class T> // Перед любым определением метода за пределами класса
vektor<T>::vektor(int n)
{
	data=new T[n]; size=n;
}
```

```
// Внешнее определение функции
template <class T> // Перед любым определением метода за пределами класса
T vektor<T>::getSum()
{
	T sum=0;
	for(int i=0; i<size; i++)
		sum+=data[i];
	return sum;

}
int main()
{
	vektor<int> X(5); // Вектор элементов типа int
	X[1]=101;// Вызов оператор функции
	cout<<"x[1]="<< X[1];
	vektor<double> Y(10); // Создается класс вместо T подставляется double
	Y[0]=10.51;// Вызов оператор функции
	cout << "\nY[0]=" << Y[0];
	for(int i=0; i<10; i++) Y[i]=i;
	cout << "\nSum="<< Y.getSum();
	return 0;

}
```

Примечание. Вместо ключевого слова class можно перед параметром шаблона использовать ключевое слово typename

```
template <typename T> // Шаблон класса вектор (массив)
// T - тип элементов вектора
class vektor
{
  ….
};
```

Тема № 11. Наследование классов

11.1. Общие сведения о наследовании

Основная идея: на базе существующего класса (класс родитель или базовый класс), создается производный класс (класс- наследник, дочерний класс), который включает в себя поля и функции базового класса (наследует их), и содержит дополнительные поля (обладает новыми свойствами) и функции.

11.2. Определение производного класса

Примеры:

class S: X, Y, Z { ... } ;

class B: public A {....};

class D: public X, protected B
{....};

Перед именем базового класса может указываться статус доступа наследования или тип наследования доступа (одно из ключевых слов public, protected, private). Статус доступа наследования определяет статус доступа наследуемых полей и функций из базового класса внутри производного класса.

Производный класс может определяться с ключевым словом struct или class (с ключевым словом union производный класс не определяется). Если производный класс определен с ключевым словом class, то по умолчанию статус доступа наследования private.

Если производный класс определен с ключевым словом struct, то по умолчанию статус доступа наследования public.

11.3. Статусы доступа при наследовании классов

Тип наследования доступа	Доступность в базовом классе	Доступность компонент базового класса в производном
public	public	public
	protected	protected
	private	не доступны
protected	public	protected
	protected	protected
	private	не доступны
private	public	private
	protected	private
	private	не доступны

11.4. Особенности конструкторов при наследовании

Конструктор производного класса в первую очередь всегда должен вызывать конструктор базового класса. Если это действие не выполняется явно, то по умолчанию вызывается конструктор без параметров (если он есть, если его нет, будет ошибка). Если класс имеет несколько базовых, то конструкторы базовых классов должны вызываться в порядке перечисления этих классов в списке базовых.

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

class A
{
public:
	A()
	{
		cout<<endl<<"A1";
	}
	A(int x)
	{
		cout<<endl<<"A2";
	}
};

class B: public A
{
public:
	B(): A(4) // Явный вызов конструктора с 1-м параметром, если нет явного вызова, то вызывается конструктор без параметров, если его нет, будет ошибка
	{
		cout<<endl<<"B";
	}
};

class C
{
public:
	/*C()
	{
		cout<<endl<<"C1";
	}*/
	C(int x)
	{
		cout<<endl<<"C2";
	}
};

class D: public A, public C
{
public:
	D() : A(3), C(5) // Явный вызов конструкторов
	{
		cout<<endl<<"D";
	}
};
```

```
int main(int argc, char* argv[])
{
	B b1;
	D d1;
	system("pause");
	return 0;
}
```

11.5. Особенности деструкторов при наследовании

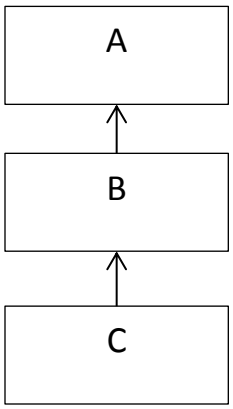
Деструктор производного класса всегда неявно по умолчанию после выполнения своего тела вызывает деструкторы базовых классов. Причем порядок разрушения объекта (вызовов деструкторов) обратен порядку создания (вызова конструкторов).

```
#include <iostream>
using namespace std;
class A
{
public:
	~A()
	{
		cout<<endl<<"A";

	}
};
class B
{
public:
	~B()
	{
		cout<<endl<<"B";
	}
};
class C: public A, public B
{
public:
	~C()
	{
		cout<<endl<<"C";
	}
};
int main(int argc, char* argv[])
{
	{
		C c1;
	} // При выходе их блока объект c1 уничтожается - вызывается деструктор
	return 0;
}
```

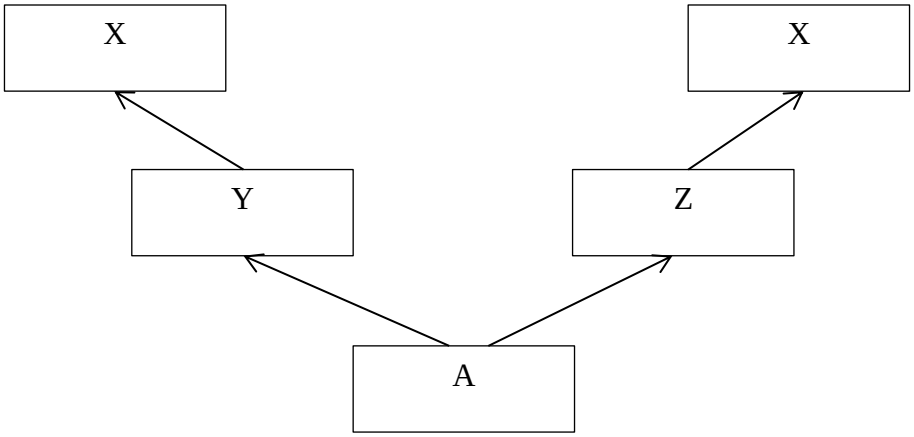
11.6. Множественное наследование и виртуальные базовые классы

Представление отношения наследования в виде диаграммы классов.



Множественное наследование – это когда класс имеет в качестве базовых более одного класса.

Пример диаграммы



Пример заголовков классов

class X { ... public: int k; void f() { ... } ... };
class Y: public X { ... } ;
class Z: public X { ... };
class A: public Y, public Z { ... };

При такой схеме наследования происходит дублирование полей и функций класса X в классе A.

A d;

Обращение к полям и функциям при дублировании за пределами класса:

d.A::Y::X::k=5; d.A::Y::X::f();
d.A::Z::X::k=10; d.A::Z::X::f();

или

d.Y::k=5; d.Y::f();
d.Z::k=10; d.Z::f();

Внутри класса A:
Y::X::k=5; Y::X::f();
Z::X::k=5; Z::X::f();
или
Y::k=5; Y::f();
Z::k=5; Z::f();

Для устранения возможность дублирования полей и методов, когда они наследуются из 1-го класса при множественном наследовании, введено понятие виртуального класса, в предыдущем примере класс X должен быть объявлен как виртуальный, для этого используется ключевое слово virtual.

Виртуальный класс — класс, который при множественном наследовании дополнительно включается в классы-потомки ссылкой на объект этого класса во избежания дублирования. (Для обычного наследования в Си++ в оперативной памяти в объекте производного класса вначале идут поля базового класса, а затем поля производного, при виртуальном наследовании дополнительно появляется ссылка в производном классе).

Термин «виртуальный» собственно относится не к базовому классу, а к способу наследования от него.

class X { ... public: int k; void f() { ... } ... };
class Y: virtual public X { ... } ;
class Z: virtual public X { ... };
class A: public Y, public Z { ... };

11.7. Переопределение функций. Виртуальные функции

Если в производном классе объявлена функция с именем, типом возвращаемого значения и количеством и типами параметров, такая же, как в базовом классе, то данная функция является переопределенной. (Не путать с перегрузкой функций). С помощью простых переопределенных функций реализуется механизм статического полиморфизма. (Полиморфизм – возможность функции в производном классе работать по-другому).

Суть статического связывания: когда указатель одного типа ссылается на объект другого типа при наследовании классов, то выбор переопределенного метода определяется типом указателя, а не типом объекта.

```
#include <iostream>
using namespace std;
// Пример статического связывания
class A
```

```
{
	int a;
public:
	void print()
	{
		cout<<endl<<"A";
	}
};
class B: public A
{
public:
	void print() // Переопределенная функция
	{
		cout<<endl<<"B";
	}
};
int main(int argc, char* argv[])
{

	B b;
	A *pA;
	pA=&b; // Преобразование типов выполняется по умолчанию (обратное не верно)
	pA->print(); // Статический полиморфизм. Вызов функции определяется типом указателя,
а не типом объекта
	cout<<endl<<"sizeof(A)="<<sizeof(A)<<" sizeof(B)="<<sizeof(B);
	return 0;
}
```

Суть динамического связывания: когда указатель одного типа ссылается на объект другого типа при наследовании классов, то выбор переопределенного метода определяется типом объекта, а не типом указателя, для этого переопределенный метод должен быть объявлен виртуальным в базовом классе.

Встретив у функции модификатор virtual компилятор создает для класса таблицу виртуальных функций, а в класс добавляется новое скрытое для программиста поле — указатель на эту таблицу.

Таблица виртуальных функций хранит в себе адреса всех виртуальных методов класса (по сути, это массив указателей), а также всех виртуальных методов базовых классов этого класса.

```
#include <iostream>
using namespace std;
// Пример динамического связывания
class A
{
	int a;
public:
	virtual void print() // Функция в базовом классе объявлена как виртуальная
	{
		cout<<endl<<"A";
	}
};
class B: public A
{
```

```
public:
	void print() // Переопределенная функция
	{
		cout<<endl<<"B";
	}
};
int main(int argc, char* argv[])
{

	B b;
	A *pA;
	pA=&b; // Преобразование типов выполняется по умолчанию (обратное не верно)
	pA->print(); // Динамический полиморфизм. Вызов функции определяется типом объекта,
а не типом указателя
	cout<<endl<<"sizeof(A)="<<sizeof(A)<<" sizeof(B)="<<sizeof(B);
// По размеру объекта видно, что есть дополнительное поле- указатель на таблицу вирт. функций
	return 0;
}
```

Имя указателя на таблицу отличается в зависимости от компилятора. К примеру, компилятор Visual Studio называет его __vfptr, а саму таблицу ‘vftable’. В литературе указатель на таблицу виртуальных функций принято называть VPTR, а саму таблицу VTABLE.

Каждый объект имеет дополнительное поле указатель на данную таблицу.

Часто внутри переопределенного метода требуется (или удобно) вызвать этот метод из базового класса, в следующем примере демонстрируется эта возможность.

```
#include <iostream>
#include <cstdlib>
using namespace std;
class point // Базовый класс - "Точка на плоскости"
{
	double x, y; // Координаты точки
public:
	point(double x, double y) // Конструктор для инициализации полей
	{
		this->x = x; this->y = y;
	}
	virtual void print() // Метод для печати полей (виртуальный)
	{
		cout << "\nx=" << x << " y=" << y; // Печатаем значения полей
	}
};
class point3d : public point // Производный класс - "Точка в пространстве"
{
	double z; // Новое поле - координата z
public:
	point3d(double x, double y, double z) : // Конструктор
		point(x, y) // Явный вызов конструктора базового класса
	{
		this->z = z;
	}
	void print() // Переопределенный метод print
	{
```

```
            point::print(); // Вызов в переопределенном методе метода базового класса
            cout << " z=" << z; // Допечатывает поле z
      }
};

int main(int argc, char* argv[])
{
      point p1(1, 2); // Создается объект с вызовом конструктора
      point3d p3(3, 4, 5); // Создается объект с вызовом конструктора
      point *pp; // Указатель типа базового класса
      pp = &p1; // Настраиваем на объект базового класса
      pp->print(); // Вызов метода через указатель
      pp = &p3; // Настриваем указатель на объект производного класса (преобразование типа
допустимо)
      pp->print(); // Вызов метода через указатель, вызывается метод класса point3d
      // Если метод print в классе point был объявлен без virtual, то вызывался бы метод
print класс point
      system("pause");
      return 0;
}
```

Виртуальный деструктор

Если в программе указатели базового класса могут ссылаться на объекты производных классов и при создании объектов производных классов дополнительно выделяется динамически память или используются ресурсы, которые нужно освободить, необходимо использовать виртуальный деструктор (объявляется явно).

Если этого не сделать, то возможны утечки памяти.

```
A *pA=new B();
delete pA;   /* Если деструктор не виртуальной, то вызывается деструктор класса
      A */
```

11.8. Абстрактные классы

Абстрактный класс это класс, который имеет в своем составе хотя бы одну чистую виртуальную функцию. ЧВФ не имеет тела и ничего не делает.

Чистая виртуальная функция:

virtual <тип_возвр_значения> <имя_функции>(<список_форм_парам>)=0;

virtual void fpure()=0;

Нельзя создать объект абстрактного класса (тип указателя на абстрактный класс может быть, этот указатель может содержать адрес объекта производного класса, не являющегося абстрактным). Абстрактный класс нужен, чтобы на его основе создавать обычные классы, в которых ЧВФ переопределяются (заменяются) на обычные функции.

Тема № 12. Использование объектов своих классов в контейнерах библиотеки STL

12.1. Требования к классам, объекты которых являются элементами последовательных контейнеров. Сортировка объектов контейнера

Пусть A, класс созданный программистом. Особенности, которые необходимо учитывать:

1. Форма записи `vector<A> vec(4);`

   Требует в классе A конструктор без параметров, при его отсутствии будет ошибка.
2. В коде `A a; vec.push_back(a);`

   В `push_back` вызывается конструктор копирования, если требуется «глубокое» копирование, например, память под данные в классе выделяется динамически (в «куче»), то этот конструктор необходимо переопределить.
3. В коде `vec.push_back(A());`

   Вызывается конструктор перемещения (перемещается в вектор созданный временный объект), если в классе перемещение отличается от копирования, например, данные находятся в «куче» (в динамической памяти), то следует учитывать, что перемещение работает быстрее, желательно задать конструктор перемещения (иначе, будет вызываться конструктор копирования). Кроме того, при добавлении новых объектов в контейнер-вектор, вектор периодически реаллоцируется (увеличивает размер выделенной памяти), при этом используется конструктор копирования.

Сортировка контейнера

При сортировке можно использовать функцию-предикат (см. тему 7, 1-ый семестр), которая передается в глобальную функцию (алгоритм) sort по указателю (для list используется метод класса sort) или передавать в качестве параметра лямбда-функцию (выражение).

Есть алгоритм sort с двумя параметрами: итератор на первый элемент внутри контейнера и на последний элемент внутри контейнера, между этими элементами выполняется сортировка элементов (по аналогии, в list есть метод sort без параметров). При использовании этого алгоритма нужно иметь правило сравнения двух объектов класса между собой, для этого необходимо выполнить перегрузку операции <. Ниже приведен пример подобной сортировки:

```
#include <iostream>
#include <vector>
#include <string>
#include <algorithm>
using namespace std;
class A
{
	string str;
	int value;
public:
	A(string str, int value) : str(str), value(value) {}
	bool operator<(const A & ob2) // Для сортировки надо уметь сравнивать 2 объекта
	{
```

```
                return str < ob2.str; // Сортировка по строковому полю
        }
        friend ostream& operator<<(ostream& out, const A & ob);
};

ostream& operator<<(ostream& out, const A & ob)
{
        out << "str=" << ob.str << " value=" << ob.value;
        return out;
}

int main()
{
        vector<A> vec;
        vec.push_back(A("World", 127));
        vec.push_back(A("Hello", 8));
        vec.push_back(A("Apple", 100));
        sort(vec.begin(), vec.end()); // Вызываем алгоритм для сортировки
        for (auto &pos : vec) cout << pos << endl;
        return 0;
}
```

12.2. Использование лямбда-функций в алгоритмах

В стандарте C++11 появились лямбда-выражения или лямбда-функции, позволяющие создавать определения подставляемых функций, которые можно использовать в качестве параметра или локального объекта.

Лямбда-выражение – это определение анонимного объекта-функции непосредственно в месте его вызова или передачи в функцию в качестве аргумента.

Минимальная лямбда-функция не имеет параметров и просто делает что-то. Например:

```
[]    {
std::cout << "hello lambda“<< std::endl;
}
```

Эту функцию можно вызывать непосредственно

```
[]   {
std::cout << "hello lambda" << std::endl; } ();
```

или передавать вызываемым объектам

```
auto I = [] {
std::cout << "hello lambda" << std::endl;
};
I(); // выводим на экран "hello lambda"
```

Лямбда-функция с параметрами

```
auto I2 = [] (std::string str){
        std::cout << str << std::endl;
};

I2("Hello World!!!!");
```

Захваты (доступ к внешней области видимости)

В инициаторе лямбда-функции (квадратных скобках перед лямбда-функцией) можно задать список захвата (capture) для доступа к данным из внешней области видимости, которые не передаются как аргументы.

– Символы [=] означают, что внешняя область видимости передается в лямбда-функцию по значению.
– Символы [&] означают, что внешняя область видимости передается в лямбда-функцию по ссылке.

Для каждого объекта в лямбда-функции необходимо указать режим доступа к нему: по значению или по ссылке. Это позволяет ограничивать доступ и смешивать разные режимы.

```
int x = 0;          int   y = 42;
auto qqq = [x, &y]    {
std::cout << "x:     " << x << std::endl;
std::cout << "y:     " << y << std::endl;
++y; // OK
};
x = y = 77;
qqq(); qqq();
std::cout << "final y:     " << y << std::endl;
```

y будет изменяться, x нет.

Пример:

```
#include <iostream>
#include <string>
int main(int argc, char* argv[])
{
	[] {
		std::cout << "hello lambda" << std::endl; } ();
		auto I = [] {
			std::cout << "hello lambda" << std::endl;
		};

		I();

		auto I2 = [](std::string str) {
			std::cout << str << std::endl;
		};

		I2("Hello World!!!!");
		int  x = 0;		int  y = 42;
		auto qqq = [x, &y] {
			std::cout << "x:   " << x << std::endl;
			std::cout << "y:   " << y << std::endl;
			++y; // OK
		//	++x;   Ошибка изменять нельзя
		};
		x = y = 77;
		qqq(); qqq();
		std::cout << "final y:   " << y << std::endl;
		return 0;
}
```

Использование лямбда-функций в STL

```
// Возводим все элементы в куб
transform(coll.begin(), coll.end(),
coll2.begin(),
[](int d)   // Лябда как функтор
```

{ return d*d*d; });
// Ищем первый элемент > x и < y
auto pos = find_if(coll.cbegin(), coll.cend(), // range
[=](int i) { // Лямбда функция вместо предиката
return i > x && i < y; });

Сортировка с использованием лямбда-функций

class Person { public: string firstname() const; string lastname() const; //...};
int main(){ deque<Person> coll;
//... Ввод элементов для дека
// sort
sort(coll.begin(),coll.end(), // range
[] (const Person& p1, const Person& p2) { // sort
return p1.lastname()<p2.lastname() || (p1.lastname()==p2.lastname() &&
p1.firstname()<p2.firstname()); }); //...}

Пример:

```
#include <iterator>
#include <algorithm>
#include <deque>
#include <vector>
#include <iostream>
using namespace std;

int main()
{
	deque<int> coll = { 1, 3, 19, 5, 13, 7, 11, 2, 17 };

	vector<int> coll2;
	coll2.resize(coll.size());
	// Возводим все элементы в куб
	transform(coll.begin(), coll.end(),
		coll2.begin(),
		[](int d)  // Лябда как функтор
	{
		return d*d*d;
	});

	copy(coll2.cbegin(), coll2.cend(),		// source
		ostream_iterator<int>(cout, " ")); // destination
	cout << endl;
	int x = 5;
	int y = 12;
	// Ищем первый элемент > x и < y
	auto pos = find_if(coll.cbegin(), coll.cend(),  // range
		[=](int i) {  // Лямбда функция вместо предиката, знак = означает все (x и y)
передается по значению
		return i > x && i < y;
	});
	cout << "first elem >5 and <12: " << *pos << endl;
}
```

12.3. Функциональные объекты (функторы)

Функциональный объект это то, что ведет себя как функция, можно вызывать с помощью пары скобок и передавать параметры.

Примеры:

– указатели на функции;

– объекты классов (функторы), содержащих перегруженную операцию operator() (в узком смысле имеется виду именно это понятие, особенность в том, что существует внутреннее состояние);
– лямбда-функции.

Стандартные функциональные объекты

Критерий сортировки с помощью операции < представляет собой стандартный функциональный объект less<>, с помощью операции > функциональный объект greater<>

```
set<int, less<int>> coll; // Сортировка с помощью <
set<int, greater<int>> coll; // Сортировка >
transform(coll.cbegin(), coll.cend(),   coll.begin(),
negate<int>());                 // Изменение знака элемента
```

Умножение элементов 2-х контейнеров:

```
transform(coll.cbegin(), coll.cend(),   // 1-й источник
coll.cbegin(),                  //   2-й источник
coll.begin(),               // Назначение
multiplies<int>());                 //   Умножение
```

Пример:

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

int main()
{
	deque<int> coll = { 1, 2, 3, 5, 7, 11, 13, 17, 19 };

	cout << "initialized: ";
	copy(coll.cbegin(), coll.cend(),
		ostream_iterator<int>(cout, " "));


	// negate all values in coll
	transform(coll.cbegin(), coll.cend(),
		coll.begin(),
		negate<int>());                 // Изменение знака элемента

	cout << endl<<"negated:     ";
	copy(coll.cbegin(), coll.cend(),
		ostream_iterator<int>(cout, " "));
	// square all values in coll
	transform(coll.cbegin(), coll.cend(),      // Первый источник
		coll.cbegin(),           // Второй источник
		coll.begin(),           // Назначение
		multiplies<int>());             // Умножение
	cout << endl<<"squared:     ";
	copy(coll.cbegin(), coll.cend(),
		ostream_iterator<int>(cout, " "));

	set<int, less<int>> coll2 = { 10, 30, 2, 6, 7, 8, 1 }; // Сортировка с помощью <
	cout << endl << "coll2:     ";
	copy(coll2.cbegin(), coll2.cend(),
		ostream_iterator<int>(cout, " "));
```

```
	set<int, greater<int>> coll3 = { 123, 34, 56, 37, 18, 2, 3, 5, 7 }; // Сортировка с помощью
>
	cout << endl << "coll3:     ";
	copy(coll3.cbegin(), coll3.cend(),
		ostream_iterator<int>(cout, " "));


}
```

Пример созданного функционального объекта:

```
#include <list>
#include <algorithm>
#include <iostream>
#include <iterator>

using namespace std;

// Функциональный объект добавляющий к элементу значение, полученное при инициализации
class AddValue {
private:
	int theValue;	// Значение
public:
	// Конструктор
	AddValue(int v) : theValue(v) {
	}

	// Вызов функции для элемента для которого добавляется значение
	void operator() (int& elem) const {
		elem += theValue;
	}
};

int main()
{
	list<int> coll;

	// Добавляем элементы в список
	for (int i = 1; i <= 9; ++i) {
		coll.push_back(i);
	}

	cout << "initialized:                ";
	copy(coll.cbegin(), coll.cend(),		// source
		ostream_iterator<int>(cout, " "));


	// Добавляем значение 10 к каждому элементу
	for_each(coll.begin(), coll.end(),	// диапазон
		AddValue(10));			// операция

	cout << endl << "after adding 10:            ";
	copy(coll.cbegin(), coll.cend(),		// source
		ostream_iterator<int>(cout, " "));

	// Добавляем значение первого элемента к каждому элементу
	for_each(coll.begin(), coll.end(),	// range
		AddValue(*coll.begin()));	// operation
	cout << endl << "after adding first element: ";
	copy(coll.cbegin(), coll.cend(),		// source
		ostream_iterator<int>(cout, " "));
```

```
}
```

12.4. Требования к классам, объекты которых являются элементами контейнеров set и map

Контейнеры ***set и map*** являются отсортированными, хранятся в памяти в виде двоичного дерева поиска (красно-черное дерево). Поэтому должен быть механизм сравнения любых двух элементов между собой, чтобы это обеспечить, для объектов класса должна быть перегружена операция < (меньше, по умолчанию). Пример представлен ниже.

```
#include <iostream>
#include <string>
#include <set>
using namespace std;
class A
{
	string str;
	int value;
public:
	A(string str, int value) : str(str), value(value) {}
	bool operator<(const A & ob2) const // Для сортировки надо уметь сравнивать 2 объекта
	{
		return str < ob2.str; // Сортировка по строковому полю
	}
	friend ostream& operator<<(ostream& out, const A & ob);
};

ostream& operator<<(ostream& out, const A & ob)
{
	out << "str=" << ob.str << " value=" << ob.value;
	return out;
}

int main()
{
	set<A> s;

	s.insert(A("World", 127));
	s.insert(A("Hello", 8));
	s.insert(A("Apple", 100));

	for (auto &pos : s) cout << pos << endl;
	return 0;
}
```

12.5. Требования к классам, объекты которых являются элементами контейнеров unordered_set и unordered_map

Объекты внутри этих контейнеров в памяти хранятся в виде хеш-таблицы, поэтому необходимо для объектов используемого класса задать хеш-функцию («хешер») и перегрузить

операцию == (равно) для объектов класса. Для стандартных типов для получения хеша рекомендуется использовать шаблонную структуру hash, объект шаблона hash является функтором. Например, hash<double> dhash; double x; dhash(x); // Получение хеша для переменной x типа double.

Если класс (структура) имеет несколько полей разных типов, то для получения комбинированного хеша, рекомендуется делать поступать так, как показано в примере ниже (задается класс-«хешер», объект класса является функтором):

```
#include <iostream>
#include <string>
#include <unordered_set>
using namespace std;
struct Person {
  string name; // Имя
  int age; //  Возраст
  double weight; // Вес
  bool operator==(const Person & ob2) const
  {
         return name == ob2.name && age == ob2.age && weight == ob2.weight;
  }
};

// Хешер для Person
struct PersonHasher
{
       // Хешеры для отдельных полей
       hash<string> shash;
       hash<int> ihash;
       hash <double> dhash;
       size_t operator()(const Person & ob) const
       {
              const size_t coef = 2'946'901; // 5171;// 11171;// 103141;// 2'946'901;
              // Получаем комбинированный хеш
              return (
                     coef * coef * shash(ob.name) +
                     coef * ihash(ob.age) +
                     dhash(ob.weight)
                     );
       }
};


int main()
{
       unordered_set<Person, PersonHasher> un_set;
       un_set.insert({ "Ivanov", 35, 73.5 });
       un_set.insert({ "Petrov", 30, 70.0 });
       un_set.insert({ "Sidorov", 29, 90.0 });
       return 1;
}
```

Получение «хорошего» хеша, который равномерно раскладывает ключи по диапазону хеш-таблицы с целью уменьшения коллизий (когда два или более разных элементов имеют один хеш) , отдельная задача. В качестве множителя coef рекомендуется использовать простые числа. В зависимости от числа используемых полей для получения общего хеша, рекомендуется использовать числа: `5171; 11171; 103141;` или `2'946'901;`

Тема № 13. Обработка особых (исключительных) ситуаций

13.1. Основы механизма обработки особых (исключительных) ситуаций

Определение особой (исключительной) ситуации:

Любая ситуация, достижимая в процессе выполнения программы, может быть объявлена программистом как особая или исключительная. Например, какая-то переменная приняла значения в заданном диапазоне (вышла из заданного диапазона), встретился конец файла, деление на 0, переполнение и т.д.

Следует отметить, что исключительной может быть объявлена синхронная ситуация, т.е. такая, которая может быть достигнута в процессе выполнения программы, прерывания от внешних устройств синхронными ситуациями не являются

Для реализации механизма обработки исключений в Си++ введены ключевые слова:

try (контролировать);
catch (ловить);
throw (генерировать).

Общий формат обработки исключительных ситуаций:

```
try // Контролируемый блок
{
……..
      if (условие_определяющее_искл_ситуацию)
      throw значение_исключения; // Генерация исключения
……..
}
// Один или несколько обработчиков исключений
catch(тип_исключения1 имя) { операторы }
……..
catch(тип_исключенияN имя) { операторы }
```

Механизм работы:

Как только выполняется оператор генерации исключения:

throw значение_исключения;

управление передается за пределы контролируемого блока в один из подходящих обработчиков исключений catch. Подходящий обработчик ищется по типу исключения. Обработчики исключений похожи на функцию с одним параметром, не возвращающую значение. После выполнения одного из обработчиков управление передается оператору, который первым следует за обработчиками, обратно в контролируемый блок управление не передается.

13.1.1. Пример обработки исключения внутри функции, в которой сгенерировано исключение

```
#include <iostream>
using namespace std;
double mydiv(double x, double y)
{
	// Обработка исключения внутри функции
	try // Контролируемый блок
	{
		if (y==0) throw '0'; // Генерация исключения 1		тип char
		if (x<0) throw "negative 1 param"; // Генерация исключения 2	тип char *
		if (y<0) throw "negative 2 param"; // Генерация исключения 3	тип char *
		return x/y;
	}
	catch(char * str) // Обработчик исключения 2 или 3 через параметр передается значение
исключения
	{
		cout<<"\nException: "<< str;

	}
	catch(char) // Обработчик исключения 1   для типа char
	{
		cout<<"\nException: x="<<x<<" y="<<y;
	}
	return 0;
}
void main()
{
	cout<<"\nRez=" << mydiv(1, -2);
	}
```

13.1.2. Пример обработки исключения вне функции, в которой сгенерировано исключение

```
#include <iostream>
using namespace std;
// В функции исключение генерируется но не обрабатывается
// обработка возложена на внешнее функцию
double mydiv(double x, double y) // throw(char *, double) Могло быть так до стандарта C++17, теперь
исключено
{

		if (y==0) throw "divide on 0";
		if (y<0) throw y;
		if (x<0) throw x;
		return x/y;
}

void main()
{
	double x=1, y=-10;

	try
	{

	    cout<<"\n%Rez="<<mydiv(x, y);
	}
	catch(char * str)
	{
```

```
		cout<<"\n"<<str;

	}
	catch(double param)
	{
		cout<<"\nparam="<<param;
	}
	cout << "\n\nEnd program";
}
```

Стандарт С11 содержит ключевое слово *noexcept (*аналог *throw()).* Означает, что функция не может генерировать не обработанное исключение:

void myfun() noexcept;

13.1.3. Пример генерации исключения как объекта класса

В этом случае исключение может быть более информативным, дополнительная информация сохраняется в полях класса.

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

struct data // Класс для задания исключения при создании в полях хранится дополнительная информация об исключении
{
	double x, y;
	char str[20];
	data(double a, double b, char *s)
	{
		x=a; y=b; strcpy(str, s);
	}
};
double mydiv(double x, double y)
{

		if (y==0) throw data(x, y, "Divide on 0"); // Создаем исключение как объект класса
		if (y<0 || x<0) throw data(x, y, "Negative parameter"); // Создаем исключение как объект класса
		return x/y;
}

void main()
{

	try
	{

	    cout<<"\nRez="<< mydiv(5, 0);
	}
	catch(data & d)
	{
		cout<<"\n"<<d.str<<" x="<<d.x<<" y="<< d.y;

	}
}
```

Основная идея:

Функция, сталкивающаяся с неразрешимой проблемой, формирует исключение в

надежде на то, что вызывающая ее (прямо или косвенно) функция сможет обработать исключение.

13.2. Формы обработчиков исключений

После блока try может быть несколько обработчиков исключений, существует три различные формы:

catch(тип_искл имя_искл) {.....}
catch(тип_исключения) {.....}
/* не предусматривает использование значения исключения, важен тип */
catch(...) {.....}
// Подходит для исключения любого типа
// обработчик должен быть последним

```
#include <iostream>
using namespace std;
class ZeroDivide {};
class Overflow {};
float mydiv(float x, float y)
{
	if (y==0) throw ZeroDivide();
	double z=x/y;
	if (z>1e+30) throw Overflow();
	if (z==0) throw 1; // Исключение типа int (Сработает универсальный обработчик)
	return z;
}

void main()
{
	try
	{
		cout<<"\nRez="<< mydiv(1e20, 1e-20);
	}
	catch(Overflow) // Важен тип исключения имя параметра отсутствует
	{
		cout<<"\nOverflow";
	}
	catch(ZeroDivide)
	{
		cout<<"\nZeroDivide"; // Важен тип исключения имя параметра отсутствует
	}
	catch(...) // Обработчик без типа подходит для любого исключения должен быть последним
	{
		cout<<"\nException";
	}
}
```

Сравнения по типам в обработчиках имеет более широкий смысл, например, обработчик catch(T t) {…..} подходит для обработки исключений типа: const T, T&, const T&, и типов производных классов от T.

13.3. Формы выражений генерации исключений

2 формы выражений генерации исключений:

throw выражение;

/* Исключение формируется как статический объект, значение которого определяется выражением генерации */

throw;

```
#include <iostream>
using namespace std;
double mydiv(double x, double y)
{
	try
	{
		if (y==0) throw "ZeroDivide"; // Тип исключения char *
		if (y<0) throw y; // Тип исключения double
		return x/y;
	}
	catch(char *str)
	{
		cout<<str;

	}
	catch(double y)
	{
		throw; // Ретрансляция исключения во внешний блок
	}
	return 0;
}

void main()
{
	try  // Внешний блок try
	{
		cout<<"\nRez="<< mydiv(9, -4);
	}
	catch(double a) // Сюда передается исключение из внутреннего блока
	{
		cout<<"\nException: a="<< a;
	}
}
```

Оператор «ретранслирует» уже существующее исключение в блок try верхнего уровня, используется внутри блока catch в случае вложенных блоков try.

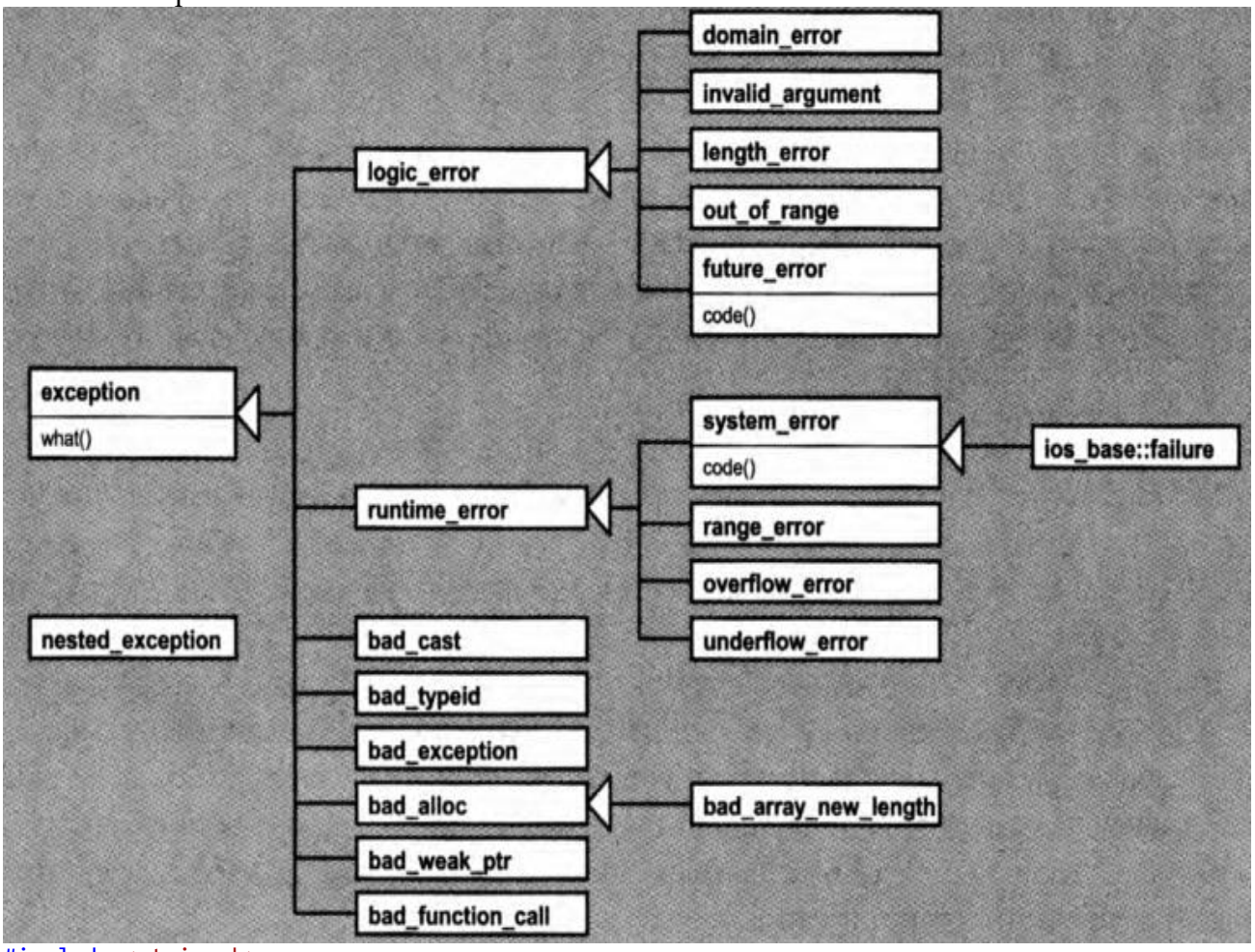
При вложении контролируемых блоков исключение, возникшее во внутреннем блоке, последовательно «просматривает» обработчики, переходя от внутреннего блока к внешнему, до тех пор, пока не будет найден подходящий обработчик. Если во всей совокупности обработчиков не будет найден подходящий, то выполняется аварийное завершение программы

13.4. Обработка исключений в стандартной библиотеке Си++

Все исключения являются производными от класса exception (пространство имен std, заголовочный файл exception). Класс содержит строку текста с описанием исключения (char *), есть конструктор для инициализации этой строки в производных классах от exception, в классе exception строка содержит «std::exception», чтобы получить указатель на строку, вызывается метод char * what();

Исключения можно разделить на 3 категории:

– языковая поддержка;
– логические ошибки;
– ошибки времени выполнения.




```
#include <string.h>
#include <iostream>
#include <exception>
using namespace std;

struct mydata: public exception // Класс для задания исключения при создании в полях хранится дополнительная информация об исключении
{
	double x, y;
	mydata(double a, double b) : exception()
	{
		x=a; y=b;
	}
};
double div(double x, double y)
{

	if (y == 0) throw exception();  // Исключение класса exception
	if (y<0 || x<0) throw mydata(x, y); // Исключение класса производного от exception
		return x/y;
}
```

```
int main()
{

	try
	{

	    cout<<"Rez="<< div(5., -0.5)<<endl;
	}
	catch(mydata & d) // Обработчик класса производного от exception
	{
		cout << endl << d.what()<< " x="<< d.x<<" y="<<d.y;

	}
	catch (exception & d) // Обработчик класса exception
	{
		cout << endl << d.what();

	}
	return 1;

}
```

Все функции в определении класса exception имеют пустую спецификацию noexcept (или ранее throw()).

Исключения языковой поддержки используются на уровне языка C++. Эти исключения генерируются при неудачных попытках выполнения некоторых операций. Некоторые классы:

- исключение класса bad_alloc генерируется при неудачном выполнении глобального оператора new (кроме версии new с запретом исключений);
- исключение класса bad_cast генерируется оператором dynamic_cast, если преобразование типа по ссылке во время выполнения завершается неудачей;
- исключение класса bad_typeid генерируется оператором typeid, предназначенным для идентификации типов по время выполнения, если аргументом typeid является ноль или null-указатель, генерируется исключение;
- исключение класса bad_exception предназначено для обработки непредвиденных исключений.

Логические ошибки обусловлены нарушением внутренней логики программы. Предполагается, что их можно найти и предотвратить еще до начала выполнения программы. Базовый класс для логических ошибок:

```
class logic_error : public exception
```

В стандартной библиотеке определены следующие классы для логических ошибок (заголовки классов):

```
class invalid_argument : public logic_error  // неверный аргумент

class out_of_range : public logic_error  // вне диапазона

class length_error : public logic_error  // неверная длина

class domain_error : public logic_error  // вне допустимой области
```

Ошибки времени выполнения связаны с событием, с самой программой не связанным. Предполагается, что их нельзя обнаружить, пока программа не начала работать. Базовый класс:

```
class runtime_error : public exception  // ошибка времени выполнения
```

Классы ошибок времени выполнения:

```
class range_error : public runtime_error  // ошибка диапазона
```

Функция может возбудить исключение range_error, чтобы сообщить об ошибке во внутренних вычислениях.

```
class overflow_error : public runtime_error  // переполнение

class underflow_error : public runtime_error { // потеря значимости
```

13.5. Структурная обработка исключений (факультативно)

Исключения в C++ это механизм обработки ошибок. Исключения представлены ключевыми словами try/catch. В мире Windows (для средств разработки Microsoft) существует еще один тип исключений: SE (Structured Exception) – структурированные исключения.

SEH (structured exception handling) введена в Windows для повышения надежность программного обеспечения. Компилятор генерирует специальный код на входах и выходах блоков исключений, создает таблицы вспомогательных структур данных для поддержки SEH.

С точки зрения программы, обработка обоих видов исключений – одинаковая. В компиляторе MS VS есть ключевые слова __try/__except/__finally. Они позволяют полностью выполнять обработку структурированных исключений способом, похожим на обработку C++ исключений.

13.5.1. Обработчики завершений

```
__try {
// Защищенный блок
….
}
__finally {
// Обработчик завершения
….
}
```

Блок __finally будет выполнен в любом случае. Как при нормальном, так и не нормальном завершении.

13.5.2. Фильтры и обработчики исключений

```
__try {
// Защищенный блок
…
}
```

__except(фильтр_исключений) { Синтаксис обработчика исключений:

```
// Обработчик исключений
…
}
```

(За __try всегда 1 блок либо finally, либо except)

Возможные значения фильтров определены в заголовочном файле Excpt.h:

EXCEPTION_EXECUTE_HANDLER (исключение во время выполнения программы)

при этом выполняется, так называемая «глобальная раскрутка» программы и выполняется код обработчика.

EXCEPTION_CONTINUE_EXECUTION –повторно проверяется фильтр. На первый взгляд, получим бесконечный процесс постоянно проверяется фильтр.

Для разрешения такой проблемы используют функции-фильтры, которые возвращают разные значения. Эта функция пишется в качестве выражения в блоке __except.

EXCEPTION_CONTINUE_SEARCH – система переходит к предыдущему блоку __try (верхнего уровня) и вычисляет его фильтр, код обработчика исключений не выполняется.

```
// MySEH_Simple.cpp : Defines the entry point for the console application.
//
#include <iostream>
#include <Excpt.h>
using namespace std;


int main(int argc, char* argv[])
{
	int x;
	int *p=0;

	while(1)
	{
	__try
	{
	*p=100; // Здесь будет ошибка (обращение по нулевому адресу), которую можно исправить
	cout << endl <<"*p=" << *p<<endl;
	break;
	}
	__except(EXCEPTION_EXECUTE_HANDLER)
	{
		printf("EXCEPTION_EXECUTE_HANDLER");
		p=&x; // Исправляем ошибку
	}
	}
	/*__finally
	{
		printf("\nFinally");
	}*/
	cout<<"End Program";
	return 0;
}
```

Тема № 14. Потоковая многозадачность

Дополнительная литература:

Уильямс Э. Параллельное программирование на C++ в действии. Практика разработки многопоточных программ.- М.: ДМК Пресс, 2016. – 672 с.

14.1. Понятие многозадачности

Два вида многозадачности:

– многозадачность, основанная на процессах;
– многозадачность, основанная на потоках.

Процесс можно определить как копию (экземпляр) выполняющейся программы. В данном случае копия – понятие статическое. Т.е. процесс– это объект, который не выполняется, а просто «владеет» выделенным ему адресным пространством, другими словами, процесс – структура в памяти.

В адресном пространстве процесса находятся не только код и данные, но и потоки (thread) – выполняющиеся объекты. При запуске процесса автоматически запускается поток (он называется главным). Главный поток может запускать другие «дочерние» потоки.

Потоки могут работать параллельно (одновременно) друг с другом с многопроцессорных системах (в однопроцессорных системах работают «как бы параллельно» за счет временного разделения) с учетом их приоритетов и имеют доступ к ресурсам процесса (приложения).

Понятие потока (thread) как части процесса не следует путать с понятием потока ввода-вывода (stream).

Потоки бывают интерфейсные (способны принимать и обрабатывать сообщения) и рабочие.

Потоки можно создавать с помощью, например, API функций Windows (платформенно-зависимые средства) или с помощью стандартной библиотеки классов Си++ (платформенно-независимые средства).

Поддержка многопоточности в языке Си++ определена стандартом 2011 г., до этого приходилось использовать платформенно-зависимые средства.

14.2. Многозадачность в стандартной библиотеке C++. Высокоуровневый интерфейс

Используется функция std::async() и шаблон класса std::future< >

Требуется вычислить:

func1()+func2()

Обеспечить параллельное вычисление.

Пример:

```
#include <iostream>
#include <future>
using namespace std;
// Требуется вычислить :
// func1() + func2()
// Обеспечить параллельное вычисление.

int func1()
{
```

```
	for (int i = 0; i < 5; i++)
	{
		cout << "func1 i=" << i << endl;
	}
	return 5;
}
int func2()
{
	for (int i = 0; i < 10; i++)
	{
		cout << "func2 i=" << i << endl;
	}
	return 10;
}

class X
{
public:
	void print(int n)
	{
		for (int i = 0; i < 10; i++)
			cout << "X n=" << n++ << endl;
	}
};
void main()
{
	future<int> rez1(async(func1)); // Асинхронное выполнение сейчас или позже
	int rez2 = func2(); // Запускаем синхронное выполнение
	int Rez = rez1.get() + rez2; // Вызов get - ожидание завершения func1
	cout << "Rez=" << Rez<<endl;
	// Передача в async указатель на функцию класса
	X x;
	future<void> a(async(&X::print, x, 11)); // Пытаемся вызвать x.print(11);
	a.get(); // Ждем завершения

}
```

Ниже представлен пример многопоточной программы, в которой умножается матрица на матрицу, потоки создаются с помощью класса thread. В этой же программе вычисляется скалярное произведение двух векторов, потоки создаются с помощью шаблона future. Потоки при создании помещаются в контейнер шаблона vector.

```
#include <iostream>
#include <thread>
#include <vector>
#include <future>
using namespace std;
//vector<vector<int>> Matr; <n x m> * <m x k> -> <n x k>
// Потоковая функция: умножает строку 1-ой матрицы на стобцец 2-ой матрицы, результат записывается
в параметр Rez
void str_in_col(vector<int> & str,  // Строка 1-ой матрицы
	vector<vector<int>>& Matr,  // 2-ая матрица целиком
	int j_col, // Номер столбца, на который умножаем
	int& rez) // Результат
{
	rez = 0;
	for (int i = 0; i < str.size(); ++i)
		rez += str[i] * Matr[i][j_col];
}
```

```
// Потоковая функция для умножения двух чисел
int proiz(int a, int b)
{
	return a * b;
}


int main()
{
	int n = 3, m = 4, k = 5;
	// Скалярное произведение двух векторов через future
	vector<int> V1{ 1, 2, 3, 4 }, V2 = { 1, 2, 3, 4 };
	vector<future<int>> VF; // Вектор future
	// Создаем потоки
	for (int i = 0; i < V1.size(); i++) VF.push_back(async(proiz, V1[i], V2[i]));
	int Rez = 0;
	// Получаем результаты
	for (auto & pos : VF) Rez += pos.get();
	cout << "Rez=" << Rez << endl;

	// Далее произведение матриц через thread
	vector<vector<int>> Matr1(n), Matr2(m), MatrRez(n);
	cout << "Matr1\n";
	for (int i = 0; i < n; i++)
	{
		Matr1[i].resize(m);
		MatrRez[i].resize(k);
		for (int j = 0; j < m; j++)
		{
			Matr1[i][j] = rand() % 10;
			cout << Matr1[i][j] << ' ';
		}
		cout << endl;
	}
	cout << "Matr2\n";
	for (int i = 0; i < m; i++)
	{
		Matr2[i].resize(k);
		for (int j = 0; j < k; j++)
		{
			Matr2[i][j] = rand() % 10;
			cout << Matr2[i][j] << ' ';
		}
		cout << endl;
	}
	vector<thread> Th; // Вектор для хранения потоков
	for (int i = 0; i < n; i++)
	{
		// Создаем потоки
		for (int j = 0; j < k; j++)
			Th.push_back(thread(str_in_col, ref(Matr1[i]), ref(Matr2), j,
ref(MatrRez[i][j])));
	}
	// Ждем завершения потоков
	for (auto & pos : Th)
		// if (pos.joinable())
			pos.join();
	cout << "MatrRez\n";
	for (auto & pos: MatrRez)
```

```
	{
		for (auto pos2 : pos)
			cout << pos2 << " ";
		cout << endl;
	}

}
```

14.3. Многозадачность в стандартной библиотеке C++. Низкоуровневый интерфейс

Для любого потока, включая главный, в заголовочном файле <thread> объявлено пространство имен std::this_thread, в котором предусмотрены потоковые глобальные функции:
this_thread::get_id(); // Получить идентиф. Потока
this_thread::sleep_for(dur); // Блокировать поток на период
this_thread::sleep_for(chrono::seconds(10)); // Пример
this_thread::sleep_until(tp); // Блокировать поток до момента времени
this_thread::sleep_until(chrono::system_clock::now()+
chrono::seconds(10));

Получить число потоков, которые могут выполняться одновременно на текущем процессоре, можно с помощью статической функции класса thread:
unsigned int thread::hardware_concurrency()

Поток создается как объект класса *thread (*пространство имен *std).*

```
void myfun(string str)   // Потоковая функция может иметь   параметры при необходимости
{
	for (int i = 0; i < 10; i++)   // 10 раз печатаем
		cout << endl << str.data();
}
 int main(int argc, char* argv[])
{
	thread th1(myfun, "Java"), th2(myfun, "C++");     // Создание потоков
// первый параметр - указатель на потоковую функцию,
// следующие параметры передаются в потоковую функцию при необходимости


	th1.join();   // Ждем завершение дочернего потока
	th2.join();    // Ждем завершение дочернего потока
	system("pause");   // Останавливаем программу до нажатия любой клавиши
	return 0;
}
```

Альтернативой метода join является вызов метода detach (отсоединить поток от родительского потока), например, th1.detach(); Отсоединенные потоки фактически выполняются в фоновом режиме, владение и управление ими передаются в библиотеку среды выполнения C++, которая гарантирует правильное высвобождение ресурсов, связанных с потоком, при выходе из него. Если не вызвать join или detach, то при удалении созданных потоков, например, при выходе из блока, в котором они созданы, будет выдано исключение.

14.4 Синхронизация

Проблема синхронизации может возникнуть, когда 2 или более потока пытаются «одновременно» получить доступ к одним и тем же данным (или одному объекту). Также часто один поток должен ожидать сигнала от другого потока, что некоторое событие произошло (например, подготовлены требуемые исходные данные) или потоки должны выполняться в определенном порядке.

Состояние гонки (race condition) — ошибка проектирования многопоточной системы, при которой работа системы зависит от того, в каком порядке выполняются части кода.

Для синхронизации можно использовать объект класса *mutex* .

```
mutex mut; // объект для синхронизации
void myfun(string str)   // Потоковая функция
{
      for (int i = 0; i < 10; i++)   // 10 раз печатаем
      {
        mut.lock();   // Блокируем объект
        cout << endl << str.data();
        mut.unlock();   // Снимаем блокировку
    }
}
```

Блокировки

std::lock_guard <std::mutex> lock(mut);
(Другой класс unique_lock<> позволяет иметь заблокированный и разблокированный mutex, lock_guard всегда блокирует mutex).

Когда объект перестает существовать блокировка снимается. Принцип:

Resource Acquisition Is Initialization (RAII) – «Получение ресурса есть инициализация»

Программная идиома объектно-ориентированного программирования, смысл которой заключается в том, что с помощью тех или иных программных механизмов получение некоторого ресурса неразрывно совмещается с инициализацией, а освобождение — с уничтожением объекта.

Пример:

```
#include <iostream>
#include <thread>
#include <stdlib.h>
#include <mutex>
using namespace std;
std::mutex mut;
void mythread(std::string str)
{
	for (int i = 0; i < 10; i++)
	{
		mut.lock();
		std::cout << std::endl << str.data();
		mut.unlock();
	}
}
void mythread2()
{
	for (int i = 0; i < 10; i++)
	{
	//	mut.lock();
```

```
            std::lock_guard <std::mutex> lock(mut); // Здесь снимать блокировку с mut не надо,
она автоматически снисмается когда объект перестает существовать
            std::cout << std::endl << "Goodbye Thread";
      //     mut.unlock();
      } // Объект lock перестает существовать при выходе из блока mut автоматически
разблокируется
}

void main()
{

      std::cout << "\nN=" << std::thread::hardware_concurrency(); // Число потоков могут
одновременно выполняться
      thread th1(mythread, "Java"), th2(mythread, "C++"); // первый параметр - указатель на
потоковую функцию,
      // следующие параметры передаются в потоковую функцию при необходимости
      std::thread::id myid = th1.get_id(); // Идентификатор потока
      mut.lock();
      std::cout << "\nid=" << myid;
      mut.unlock();

      thread th3(mythread2);
      // первый параметр - указатель на потоковую функцию,
      // следующие параметры передаются в потоковую функцию при необходимости

      th1.join();  // Ждем завершение дочернего потока
      th2.join();   // Ждем завершение дочернего потока
      th3.join();

      system("pause");
}
```

Условные переменные

Используется заголовочный файл <condition_variable>

Условная переменная это переменная, с помощью которой поток может активировать один или несколько других ожидающих потоков.

Для использования необходимо включить в программу файлы <mutex> и <condition_variable>. Объявить mutex и условную переменную:

mutex mut1;
condition_variable con_var;

Поток сигнализирующий о выполнении условия должен выполнить вызов:
con_var.notify_one(); или
con_var.notify_all();
Любой поток ожидающий выполнение условие должен выполнить вызов:
unique_lock<mutex> lock(mut1);
con_var.wait(lock);
Пример:

```
#include <future>
#include <mutex>
#include <condition_variable>
#include <iostream>
using namespace std;
mutex mut1;
condition_variable con_var;
bool flag = false; // Флаг для предотвращения ложных срабатываний
```

```
void thread1()
{
	// Имитируем подготовку исходных данных
	cout << "Start Thread1" << endl;
	for (int i = 0; i < 10; i++)
		cout << "Thread 1 i=" << i << endl;
	flag = true; // Данные готовы
	con_var.notify_one(); // Данные готовы (посылаем "сигнал")
}

void thread2()
{
	// Имитируем ожидание данных от первого потока
	cout << "Start Thread2" << endl;
	unique_lock<mutex> lock(mut1);
	while(!flag) con_var.wait(lock); // Ждем "сигнала"
	flag = false; // Условная переменная готова для следующего использования
	cout << "Thread 2 Data read" << endl;

}

void main()
{
	auto f1 = async(launch::async, thread1);
	auto f2 = async(launch::async, thread2);
	system("pause");
}
```

Атомарные операции

Не могут быть прерваны другим потоком, выполняются без перерывов. atomic – шаблон (заголовочный файл atomic).

Определены следующие классы для стандартных типов:

std::atomic_char	std::atomic<char>
std::atomic_schar	std::atomic<signed char>
std::atomic_uchar	std::atomic<unsigned char>
std::atomic_short	std::atomic<short>
std::atomic_ushort	std::atomic<unsigned short>
std::atomic_int	std::atomic<int>
std::atomic_uint	std::atomic<unsigned int>
std::atomic_long	std::atomic<long>
std::atomic_ulong	std::atomic<unsigned long>
std::atomic_llong	std::atomic<long long>
std::atomic_ullong	std::atomic<unsigned long long>

Создается атомарный объект (заголовочный файл <atomic>): основные операции store – присвоить новое значение и load – возвращает текущее значение, fetch_add – увеличить значение объекта на заданное значение. Ниже в примере несколько потоков работают с обычной переменной, увеличивая ее на 1 (10 потоков каждый по 10000 выполняет инкремент переменной). При этом, каждый раз получаем разный результат, как правило, не 100000.

Пример с обычной переменной:

```
#include <iostream>

#include <future>

#include <vector>

using namespace std;

int value=0;

void do_work()

{

	for(int i=0; i<10000; i++)

	++value; // Увеличиваем на 1

}


int main()

{

	{

		vector<future<void>> vecFuture;

		for(int i=0; i<10; i++) vecFuture.push_back(async(do_work));

	} // При выходе из блока дожидаемся завершения потоков

	std::cout << "Result : " << value << '\n';

}
```

Ниже в примере, выполняется тоже самое с атомарным объектом, результат будет при каждом старте один и тот же 100000.

Пример с атомарным объектом:

```
#include <iostream>

#include <future>

#include <vector>

#include <atomic>
```

```
using namespace std;

atomic<int> value=0;

void do_work()

{

	for(int i=0; i<10000; i++)

	value.fetch_add(1); // Увеличиваем на 1

}


int main()

{

	{

		vector<future<void>> vecFuture;

		for(int i=0; i<10; i++) vecFuture.push_back(async(do_work));

	} // При выходе из блока дожидаемся завершения потоков

	std::cout << "Result : " << value << '\n';

}
```

Далее материал факультативный, посвящен программирования многопоточных программ с использованием платформенно-зависимых средств ОС Windows.

14.5. Создание потока с помощью API – функций Windows

Основные функции:

```
HANDLE CreateThread(     // Создать поток
LPSECURITY_ATTRIBUTES lpThreadAttributes, // дескриптор защиты
	SIZE_T dwStackSize,                  // начальный размер стека
	LPTHREAD_START_ROUTINE lpStartAddress,     // функция потока
	LPVOID lpParameter,                  // параметр потока
	DWORD dwCreationFlags,                  // опции создания
	LPDWORD lpThreadId                   // идентификатор потока
);
```

DWORD SuspendThread(// Приостановить поток
HANDLE hThread // дескриптор (хэндел) потока
);

DWORD ResumeThread(// Возобновить поток
HANDLE hThread // дескриптор (хэндел) потока
);

Потоковая функция должна иметь следующий заголовок:
DWORD WINAPI PotokFun(LPVOID param)

Установить приоритет потока:
BOOL SetThreadPriority(
HANDLE *hThread*, // дескриптор потока
int *nPriority* // уровень приоритета потока
);

Возможные значения уровней приоритета:
THREAD_PRIORITY_LOWEST THREAD_PRIORITY_BELOW_NORMAL
THREAD_PRIORITY_NORMAL THREAD_PRIORITY_HIGHEST
THREAD_PRIORITY_ABOVE_NORMAL THREAD_PRIORITY_ERROR_RETURN
THREAD_PRIORITY_TIME_CRITICAL THREAD_PRIORITY_IDLE

Нормальное завершение потока - естественный выход из потоковой функции.

Поток можно завершить из вне с помощью вызова функции:
BOOL TerminateThread(
HANDLE *hThread*, // дескриптор потока
DWORD *dwExitCode* // код завершения для потока
);

Такое завершение потока небезопасное (нет гарантии, что внешние ресурсы, используемые потоком будут освобождены).

Пример:

```
#include <Windows.h>
#include <iostream>
using namespace std;
DWORD WINAPI MyThread1(LPVOID par) // Потоковая функция
{
	char Str[64];
	strcpy_s(Str, (char *)par); // Копируем строку

	for (int i = 0; i<10; i++)
	{

		cout << endl << Str;

		Sleep(10);
	}
```

```
	return 1;
}


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

	HANDLE h1 = CreateThread(0, 0, MyThread1, "C++", 0, 0); // Создаем 1-й поток

	HANDLE h2 = CreateThread(0, 0, MyThread1, "Java", 0, 0); // Создаем 2-й поток
	//	SetThreadPriority(h2, THREAD_PRIORITY_TIME_CRITICAL);

	for (int i = 0; i<10; i++)
		cout << endl << "Main Thread";

	system("pause");
	return 0;
}
```

14.6. Синхронизация потоков

Объекты синхронизации (Windows):

Синхронизирующий объект	Назначение	Используемые функции Win32 API
Взаимное исключение	Запрет доступа более чем одному потоку к общим ресурсам	CreateMutex WaitForSingleObject WaitForMultipleObjects ReleaseMutex CloseHandle
Критическая секция	Запрет доступа более чем одному потоку к одному фрагменту кода	InitializeCriticalSection EnterCriticalSection LeaveCriticalSection DeleteCriticalSection
Семафор	Ограничение числа потоков, имеющих одновременный доступ к общим ресурсам	CreateSemaphore WaitForSingleObject WaitForMultipleObjects ReleaseSemaphore CloseHandle
Событие	Позволяет потоку передавать сигналы другим потокам	CreateEvent SetEvent PulseEvent ResetEvent WaitForSingleObject WaitForMultipleObjects CloseHandle

Использование объектов:

1. *Взаимное исключение*

HANDLE HMutex; // Объявляется глобально и все потоки имеют доступ
HMutex=CreateMutex(
NULL, // атрибуты прав доступа по умолчанию

FALSE, // Взаимное исключение изначально свободно
NULL); // не присваивается имя взаимному исключению

Работа с синхронизированным объектом
WaitForSingleObject(HMutex, // Занять взаимное исключение
INFINITE); // ждите сколько нужно
Count++; // Работа с синхронизированным объектом
ReleaseMutex(HMutex); // Освободить взаимное исключение

Удалить взаимное исключение:
CloseHandle(HMutex);

Пример:

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

using namespace std;

HANDLE HMutex;  // Объект - взаимное исключение

DWORD WINAPI PotokFun(LPVOID param)   // Потоковая функция
{
	char *str = (char *)param;

	for (int i = 0; i<10; i++)
	{
		WaitForSingleObject(HMutex, // Занять взаимное исключение
			INFINITE); // ждите сколько нужно

		cout << "\n\"" << str; // Печатаем строку в кавычках
		//Sleep(10);
		cout << "\""; // Кавычка закрывается
		ReleaseMutex(HMutex); // Кавычка закрывается
	}

	return 1;
}



int main(int argc, char* argv[])
{
	DWORD id;
	HMutex = CreateMutex(
		0, // атрибуты прав доступа по умолчанию
		FALSE, // Взаимное исключение изначально свободно
		NULL); // не присваивается имя взаимному исключению

	CreateThread(0, 0, PotokFun, "C++", 0, &id);
	CreateThread(0, 0, PotokFun, "Java", 0, &id);

	system("pause");
	CloseHandle(HMutex);


	return 0;
}
```

2. *Семафор*

Работает по аналогии с взаимным исключение только доступ к объекту может получить не один поток, а их число определяется параметром MaximumCount (при занятии потоком семафора текущее значение счетчика уменьшается на 1, семафор полностью занят если значение счетчика равно 0).

```
HANDLE   CreateSemaphore(   // Создать семафор
  LPSECURITY_ATTRIBUTES lpAttr,   // Атрибуты доступа
   LONG InitialCount,     // Начальное значение семафора
  LONG MaximumCount,   // Макс. значение
 LPCTSTR lpName );     // Имя семафора
```

Возможная схема использования:

```
WaitForSingleObject(HSem, // Занять семафор
            INFINITE); // ждите сколько нужно
        Count++; // Работа с синхронизированным объектом
ReleaseSemaphore(HSem, 1, 0); // Освободить одно место семафора
```

Пример:

```
#include <windows.h>
#include <iostream>
using namespace std;
HANDLE hSem; // Объект - семафор
DWORD WINAPI PotokFun(LPVOID param)
{
	char *str = (char *)param;

	for (int i = 0; i<10; i++)
	{
		WaitForSingleObject(hSem, // Занять одно место в семафоре
			INFINITE); // ждите сколько нужно
		cout<<"\n\""<< str; // Печатаем строку в кавычках
	//	Sleep(10); // Задежка в мс
		cout<<"\""; // Кавычка закрывается
		ReleaseSemaphore(hSem, 1, 0); // Освободить одно место в семафоре
	}

	return 1;
}
int main(int argc, char* argv[])
{
	hSem = CreateSemaphore(0, 1, 1, 0);
	CreateThread(0, 0, PotokFun, "C++", 0, 0);
	CreateThread(0, 0, PotokFun, "Java", 0, 0);
	system("pause");
	CloseHandle(hSem);
	return 0;
}
```

3. Критическая секция - это фрагмент программы, защищенный от одновременного выполнения несколькими потоками. Критическую секцию в данный момент может выполнять только один поток.

InitializeCriticalSection - данная функция ***создает объект под названием критическая секция***. Параметры функции:

Указатель на структуру, CRITICAL_SECTION. Поля данной структуры используются только внутренними процедурами, их смысл безразличен.

EnterCriticalSection - ***войти в критическую секцию***. После выполнения этой функции данный поток становится владельцем данной секции. Следующий поток, вызвав данную функцию, будет находиться в состоянии ожидания. Параметр функции такой же, что и в предыдущей функции.

LeaveCriticalSection - ***покинуть критическую секцию***. После этого второй поток, который был остановлен функцией **EnterCriticalSection**, станет владельцем критической секции. Параметр функции **LeaveCriticalSection** такой же, как и у предыдущих функций.

DeleteCriticalSection - ***удалить объект "критическая секция"***. Параметр аналогичен предыдущим.

Пример:

```
#include <Windows.h>
#include <iostream>
using namespace std;
CRITICAL_SECTION cs;
DWORD WINAPI PotokFun(LPVOID param)
{
	char *str = (char *)param;

	EnterCriticalSection(&cs);
	for (int i = 0; i<10; i++)
	{

		cout<<"\n\""<< str; // Печатаем строку в кавычках
		Sleep(10); // Задежка в мс
		cout<<"\""; // Кавычка закрывается

	}
	LeaveCriticalSection(&cs);

	return 1;
}
int main(int argc, char* argv[])
{
	InitializeCriticalSection(&cs);
	CreateThread(0, 0, PotokFun, "C++", 0, 0);
	CreateThread(0, 0, PotokFun, "Java", 0, 0);
	system("pause");
	DeleteCriticalSection(&cs);
	return 0;
}
```

4. *Событие*

```
HANDLE hEvent;
hEvent=CreateEvent(0,
	false, // TRUE - событие со сбросом вручную FALSE — событие с автосбросом
	false, // событие вкл. (произошло) – 1,
			// ссобытие выкл. (не произошло) - 0.
	0);
	WaitForSingleObject(hEvent,
	INFINITE);
// Программа далее не выполняется
```

Когда событие произошло необходимо вызвать в другом потоке функцию

```
SetEvent(hEvent);
```

Для сброса события используется функция ResetEvent.

Пример:

```
#include <Windows.h>
#include <iostream>
using namespace std;
HANDLE hEvent;

DWORD WINAPI PotokFun(LPVOID param)
{
	char *str = (char *)param;
	for (int i = 0; i<20; i++)
	{

		cout << "\n\"" << str; // Печатаем строку в кавычках
		//Sleep(10);
		cout << "\""; // Кавычка закрывается

	}
	SetEvent(hEvent); // Событие произошло
	return 1;
}

int main(int argc, char* argv[])
{
	hEvent = CreateEvent(0,
		false, // TRUE событие со сбросом вручную FALSE – событие с автосбросом
		false, // свободное (TRUE) занятое (FALSE).
		0);
	CreateThread(0, 0, PotokFun, "C++", 0, 0);
	WaitForSingleObject(hEvent, // Ожидаем наступления события
		INFINITE);
	cout<<"\nEnd Main";
	system("pause");
	CloseHandle(hEvent);

	return 0;
}
```

Синхронизация разных процессов

С помощью объектов синхронизации можно организовать синхронизацию различных приложений (процессов). Например, доступ к объекту синхронизации, созданному в одном приложении, можно получить в другом приложении:

HANDLE OpenEvent(
DWORD dwDesiredAccess, // флаги доступа
BOOL bInheritHandle, // режим наследования
LPCTSTR lpName // имя события
);

Возможные значения флагов доступа:

- EVENT_ALL_ACCESS
- EVENT_MODIFY_STATE
- SYNCHRONIZE

14.7. Пул потоков (факультативно)

Чтобы не создавать и удалять потоки, можно создать множество потоков или пул (они

могут находится в состоянии ожидания). Пулу потоков передаются задания для решения, представляющие собой потоковые функции.

Основные функции:

Создать пул:
PTP_POOL WINAPI CreateThreadpool(PVOID reserved);

Установить минимальное и максимальное число потоков в пуле:
BOOL WINAPI SetThreadpoolThreadMinimum(PTP_POOL ptpp,
DWORD cthrdMin);
VOID WINAPI SetThreadpoolThreadMaximum(PTP_POOL ptpp,
DWORD cthrdMost);

Инициализировать среду:
void InitializeThreadpoolEnvironment(
PTP_CALLBACK_ENVIRON pcbe);

Связать пул потоков со средой:
void SetThreadpoolCallbackPool(
PTP_CALLBACK_ENVIRON pcbe,
PTP_POOL ptpp);

Создать работу для пула(связать с функцией):
PTP_WORK WINAPI CreateThreadpoolWork(PTP_WORK_CALLBACK pfnwk, PVOID Context,
PTP_CALLBACK_ENVIRON pcbe);

Передать работу на выполнение:
VOID WINAPI SubmitThreadpoolWork(PTP_WORK pwk);

Функция для закрытия объектов

- CloseThreadpoolWork
- DestroyThreadpoolEnvironment
- CloseThreadpool

Пример:

```
#include <Windows.h>
#include <stdio.h>
HANDLE HMutex; // Взаимное исключение для синхронизации
struct PARAM // Данные для выполнения задачи
{
	double x, y;
};
int Num1 = 1;
VOID CALLBACK sum(PTP_CALLBACK_INSTANCE Instance, // Функция для выполнения суммы (работа для пула потоков)
	PVOID p,
	PTP_WORK Work)

{
	PARAM *pPar = (PARAM *)p;// Копируем указатель на данные
```

```
	double rez = pPar->x + pPar->y;

	WaitForSingleObject(HMutex, // Занять взаимное исключение
		INFINITE); // ждите сколько нужно

	printf("\n\"Num=%d %.3f=%.2f+%.2f", Num1, rez, pPar->x, pPar->y);
	Sleep(10);
	pPar->x++; pPar->y++; Num1++;
	printf("\"");
	ReleaseMutex(HMutex);

}

VOID CALLBACK proiz(PTP_CALLBACK_INSTANCE Instance, // Функция для выполнения произведения
(работа для пула потоков)
	PVOID p,
	PTP_WORK Work)

{
	PARAM *pPar = (PARAM *)p;//  Копируем указатель на данные
	double rez = pPar->x*pPar->y;

	WaitForSingleObject(HMutex, // Занять взаимное исключение
		INFINITE); // ждите сколько нужно

	printf("\n\"Num=%d %.3f=%.2f*%.2f", Num1, rez, pPar->x, pPar->y);
	Sleep(10);
	pPar->x++; pPar->y++; Num1++;
	printf("\"");
	ReleaseMutex(HMutex);

}

VOID CALLBACK div(PTP_CALLBACK_INSTANCE Instance, // Функция для выполнения деления (работа для
пула потоков)
	PVOID p,
	PTP_WORK Work)

{
	PARAM *pPar = (PARAM *)p;// Копируем указатель на данные
	double rez = pPar->x / pPar->y;

	WaitForSingleObject(HMutex, // Занять взаимное исключение
		INFINITE); // ждите сколько нужно

	printf("\n\"Num=%d %.3f=%.2f/%.2f", Num1, rez, pPar->x, pPar->y);
	Sleep(10);
	pPar->x++; pPar->y++; Num1++;
	printf("\"");
	ReleaseMutex(HMutex);

}


VOID CALLBACK minus(PTP_CALLBACK_INSTANCE Instance, // Функция для выполнения вычитания (работа
для пула потоков)
	PVOID p,
	PTP_WORK Work)

{
```

```
	PARAM *pPar = (PARAM *)p;// Копируем указатель на данные
	double rez = pPar->x - pPar->y;

	WaitForSingleObject(HMutex, // Занять взаимное исключение
		INFINITE); // ждите сколько нужно

	printf("\n\"Num=%d %.3f=%.2f-%.2f", Num1, rez, pPar->x, pPar->y);
	Sleep(10);
	pPar->x++; pPar->y++; Num1++;
	printf("\"");
	ReleaseMutex(HMutex);

}
int main(int argc, char* argv[])
{
	HMutex = CreateMutex(
		0, // атрибуты прав доступа по умолчанию
		FALSE, // Взаимное исключение изначально свободно
		NULL); // не присваивается имя взаимному исключению
	PARAM Par;
	PTP_POOL pt = CreateThreadpool(NULL); // Создать пул

	if (pt == NULL)
		printf("Error!!!");
	else printf("Ok!");
	// Установить минимальное и максимальное число потоков  в пуле
	SetThreadpoolThreadMinimum(pt,
		2);
	SetThreadpoolThreadMaximum(pt,
		4);

	TP_CALLBACK_ENVIRON enf;
	InitializeThreadpoolEnvironment(&enf); // Инициализировать среду
	SetThreadpoolCallbackPool(&enf,
		pt);   // Связать пул потоков со средой
	Par.x = 1; // Данные
	Par.y = 1;
	// Создать работы для пула (связать с функцией обратного вызова)
	PTP_WORK work1 = CreateThreadpoolWork((PTP_WORK_CALLBACK)sum, // Сумма
		&Par,
		&enf);
	PTP_WORK work2 = CreateThreadpoolWork((PTP_WORK_CALLBACK)div, // Деление
		&Par,
		&enf);
	PTP_WORK work3 = CreateThreadpoolWork((PTP_WORK_CALLBACK)proiz, // Произведение
		&Par,
		&enf);
	PTP_WORK work4 = CreateThreadpoolWork((PTP_WORK_CALLBACK)minus, // Вычитание
		&Par,
		&enf);



	for (int i = 0; i<50; i++)  // Передать работу на выполнение в цикле
	{

		SubmitThreadpoolWork(work1);
		SubmitThreadpoolWork(work2);
		SubmitThreadpoolWork(work3);
		SubmitThreadpoolWork(work4);
```

```
        }
        system("pause");
        // Далее закрытие объектов
        CloseThreadpoolWork(work1);
        CloseThreadpoolWork(work2);
        CloseThreadpoolWork(work3);
        CloseThreadpoolWork(work4);
        DestroyThreadpoolEnvironment(&enf);
        CloseThreadpool(pt);
        CloseHandle(HMutex);

        return 0;
}
```

Тема 15. Умные (интеллектуальные) указатели, введение в управление памятью

Умные указатели (smart pointers) набор классов (шаблонов классов) библиотеки C++. Назначение – автоматизировать работу с динамической памятью, основная цель предотвращение утечки памяти.

Объекта класса типа умного указателя поддерживает основные операции применяемые к обычным (иногда используется термин «сырой» указатель) указателям, такие как, *, -> за счет перегрузки этих операций.

При в умных указателях применяется принцип *Resource Acquisition Is Initialization (RAII)* – «Получение ресурса есть инициализация», смысл которого заключается в том, что с помощью тех или иных программных механизмов получение некоторого ресурса неразрывно совмещается с инициализацией, а освобождение — с уничтожением объекта. В частности, для умного указателя как объекта класса автоматически вызывается деструктор, которой и должен освободить динамическую память, являющуюся ресурсом для указателя.

Для использования умных указателей подключается заголовочный файл <memory>

15.1. Указатель std::unique_ptr

std::unique_ptr простейший умный указатель, является шаблоном класса (перегруженным шаблоном, существует два шаблона: один для отдельного объекта, второй – для массива объектов). Указатель unique_ptr всегда полностью владеет объектом, т.е. на один объект указывает только один указатель, на один объект не могут указывать два или более указателей. Поэтому в шаблоне класса конструктор копирования и оператор присваивания с копированием удалены, конструктор перемещения и оператор присваивания с перемещением присутствуют.

Существует конструктор, который получает «сырой» указатель на объект класса. Пример использования unique_ptr

```
#include <iostream>
#include <memory>
using namespace std;
struct A
{
    int a, b;
    int *p=nullptr;
    A(int a, int b): a(a), b(b) { p= new int [100];}
    A() { a=b=0; }
    ~A()
    {
        delete [] p;
        cout<<"~A()"<<endl;
    }
};
ostream & operator<<(ostream & out, const A & ob)
{
    out<<"a="<<ob.a<<" b="<<ob.b;
    return out;
}

int main()
{
    unique_ptr<A> pA(new A(2, 4)); // Создаем умный указатель на A
```

```
    pA->a=101;
    cout<<*pA<<endl; // Печатаем объект
    return 0; // При выходе из функции unique_ptr в деструкторе вызывает
деструктор для объекта A
}
```

Использование конструктора, который получает «сырой» указатель на объект не безопасно. Ниже показан пример, когда это приводит к ошибке, объект A «удаляется» дважды при выходе из main.

```
int main()
{
    A *p=new A(2, 4);
    unique_ptr<A> pA1(p);
    unique_ptr<A> pA2(p); // Второй указатель указывает на тот же объект
    return 0;
}
```

Для создания указателя std::unique_ptr рекомендуется использовать глобальную шаблонную функцию std::make_unique, которая получает параметры как у конструктора объекта, на который указывает указатель.

```
 int main()
{
    unique_ptr<A> pA=make_unique<A>(1, 2);
    cout<<*pA<<endl;
    return 0;
}
```

Получить «сырой» указатель можно с помощью вызова метода get() для объекта unique_ptr.

Чтобы понять, как работает unique_ptr необходимо написать его код самому. Ниже представлен пример упрощенного шаблонного класса для указателя с основными функциями как у unique_ptr:

```
#include <iostream>
#include <memory>
using namespace std;
struct A
{
    int a, b;
    int *p=nullptr;
    A(int a, int b): a(a), b(b) { p= new int [100];}
    A() { a=b=0; }
    ~A()
    {
        delete [] p;
        cout<<"~A()"<<endl;
    }

};
ostream & operator<<(ostream & out, const A & ob)
```

```
{
    out<<"a="<<ob.a<<" b="<<ob.b;
    return out;
}

template<class T>
class MyUnique
{
    T * p=nullptr;
public:
    MyUnique(T *p): p(p) {}
    ~MyUnique() { delete p; }
    MyUnique(const MyUnique & )=delete;
    MyUnique(MyUnique && ob)
    {
        swap(p, ob.p);
    }
    MyUnique & operator=(const MyUnique &)=delete;
    MyUnique & operator=(MyUnique && ob)
    {
        swap(p, ob.p);
        return *this;
    }
    T * get() const
    {
       return p;
    }
    T & operator*()
    {
        return *p;
    }
    T * operator->()
    {
        return p;
    }
};

int main()
{
    MyUnique<A> myA(new A(3, 4));
    cout<<*myA<<endl;
    myA->a=111;
    myA->b=222;
    cout<<*myA<<endl;
    return 0;
}
```

15.2. Указатель std::shared_ptr

В отличии от unique_ptr разрешает копирование, таким образом, на один объект могут указывать два или более указателей. По аналогии с unique_ptr рекомендуется объект с указателем shared_ptr создавать с помощью глобальной функции std::make_shared (также передаются параметры конструктора объекта класса).

Пример создания объекта:

```
int main()
```

```
{
    shared_ptr<A> pA=make_shared<A>(1, 2);
    cout<<*pA<<endl;
    return 0;
}
```

shared_ptr должен каким-то образом знать, а существуют ли другие shared_ptr, которые указывают на тот же самый объект. Когда для shared_ptr вызывается деструктор, то он должен определить, удалять объект, на который он указывает, или нет. Если существуют другие shared_ptr, указывающие на этот же объект, то объект не удаляется, если других shared_ptr, указывающих на объект не существует, то объект удаляется. Для этого в динамической памяти (куче) создается небольшой объект ControlBlock. В этом контрольном блоке сохраняется счетчик ссылок, т.е. текущее количество shared_ptr указывающих на текущий объект. У себя shared_ptr хранит указатель на этот контрольный блок. Таким образом shared_ptr имеет два указателя: на сам объект и на контрольный блок. При создании копии указателя копируются два указателя и в контролируем блоке счетчик объектов увеличивается на 1. В деструкторе shared_ptr в контролируемом блоке счетчик объектов уменьшается на 1, если он стал равен 0, это значит - удаляется последний shared_ptr, указывающий на объект и объект уничтожается, также освобождается память, выделенная для ControlBlock.

При перемещении shared_ptr счетчик в ControlBlock не изменяется. Но в случае перегрузки оператора присваивания с копированием или перемещением. Если объект, в который копируем или перемещаем, отличается от копируемого или перемещаемого и не является пустым, то для этого объекта уменьшаем счетчик ссылок на 1, если счетчик равен 0, то объект удаляем, и удаляем счетчик ссылок для него.

Значение счетчика можно получить с помощью функции класса:

```
long use_count() const noexcept;
```

```
int main()
{
    shared_ptr<A> pA1=make_shared<A>(1, 2);
    shared_ptr<A> pA2=make_shared<A>(3, 4);
    shared_ptr<A> pA1_Copy=pA1;
    shared_ptr<A> pA2_Copy=pA2;
    cout<<pA1.use_count()<<' '<<pA2.use_count()<<endl; // Печатает 2 2
    pA2_Copy=pA1;
    cout<<pA1.use_count()<<' '<<pA2.use_count()<<endl; // Печатает 3 1
    return 0;
}
```

15.3. Указатель std::weak_ptr

Не владеющий (слабый) указатель. Можно проинициализировать указателем shared_ptr при этом значение счетчика ссылок не увеличится. С помощью метода weak_ptr::lock() получаем shared_ptr для объекта, если объект существует (существует хотя бы один shared_ptr для объекта). Если объект уже не существует (перестали существовать все shared_ptr на него), то lock() вернет «пустой» shared_ptr.

```
int main()
{
```

```
    weak_ptr<A> pA_W;
    {
        shared_ptr<A> pA=make_shared<A>(1, 2);
        pA_W=pA; // Инициализация weak_ptr
        cout<<pA_W.use_count()<<endl; // 1
        auto p=pA_W.lock(); // p имеет тип shared_ptr<A>
        if (p)
          cout<<*p<<endl; // Печатается объект
        else cout<<"Object deleted!!!"<<endl;
        cout<<pA_W.use_count()<<endl;  // 2
    } // При выходе из блока все shared_ptr<A>  уничтожаются
    auto p=pA_W.lock(); // Вернет пустой shared_ptr<A>
    if (p)
        cout<<*p<<endl;
    else cout<<"Object deleted!!!"<<endl; // Печатается Object deleted!!!
    cout<<pA_W.use_count()<<endl;  // 0
    return 0;
}
```

15.4. Перегрузка операторов new и delete

Основные стандартные формы

```
new T(/* аргументы конструктора */)
new T[/* длина массива */]
delete ptr;
delete[] ptr;
```

Дополнительные стандартные формы

При подключении заголовочного файла <new> становятся доступными еще 4 стандартные формы оператора new:

```
new(ptr) T(/* аргументы конструктора*/);
new(ptr) T[/* длина массива */];
new(std::nothrow) T(/* аргументы конструктора */);
new(std::nothrow) T[/* длина массива */];
```

Первые две из них называются размещающим оператором new (non-allocating placement new). Аргумент ptr — это указатель на область памяти, размер которой достаточен для размещения экземпляра или массива. Вторые два варианта называются не выбрасывающим исключений оператором new (nothrow new) и отличаются тем, что при невозможности удовлетворить запрос возвращают nullptr, а не выбрасывают исключение типа std::bad_alloc.

Функции выделения и освобождения памяти

Стандартные формы операторов new/delete используют следующие функции выделения и освобождения памяти (allocation and deallocation functions):

```
void* operator new(std::size_t size);
void operator delete(void* ptr);
void* operator new[](std::size_t size);
void operator delete[](void* ptr);
void* operator new(std::size_t size, void* ptr);
void* operator new[](std::size_t size, void* ptr);
```

```
void* operator new(std::size_t size, const std::nothrow_t& nth);
void* operator new[](std::size_t size, const std::nothrow_t& nth);
```

Эти функции определены в глобальном пространстве имен. Функции выделения памяти для размещающих операторов new ничего не делают и просто возвращают ptr.

Перегрузка стандартных форм операторов new/delete заключается в определении пользовательских функций выделения и освобождения памяти, сигнатуры которых совпадают со стандартными. Не рекомендует делать перегрузку в глобальном пространстве имен.

Что делает new для объекта:

- выделяет память под объект (если это не размещающий оператор new);

- вызывает конструктор объекта;

- возвращает указатель на выделенную память.

Что делает delete для объекта:

- получает указатель на память, которую надо очистить;
- вызывает деструктор объекта;
- освобождает память.

```
#include <iostream>
using namespace std;
struct T {
  T() { std::cout << "T::T()" << std::endl; }
  void* operator new(std::size_t size) {
    auto p = ::operator new(size);
    std::cout << "T::new(" << size << ") " << p << std::endl;
    return p;
 }
 void operator delete(void* p) {
    std::cout << "T::delete(" << p << ")" << std::endl;
    if (!p) return;
    ::operator delete(p);
  }
  ~T() { std::cout << "T::~T()" << std::endl; }
 };

int main()
{
    T * ptr = new T();
    delete ptr;
    return 0;
}
```

Размещение объектов в заранее выделенной памяти, например, в статической.

```
#include <iostream>
using namespace std;
char buf[1000];
struct T {
  T() { std::cout << "T::T()" << std::endl; }
  void* operator new(std::size_t size, void *ptr) {
```

```
    auto p = ::operator new(size, ptr);
    std::cout << "T::new(" << size << ") " << p << std::endl;
    return p;
 }
 ~T()
 {
     cout<<"T::~T()"<<endl;
 }
};

int main()
{
    char *data=buf;
    T * ptr1 = new(data) T();
    data+=sizeof(T);
    T * ptr2 = new(data) T();
    ptr1->~T();
    ptr2->~T();
    return 0;
}
```

Оператор new может иметь дополнительные параметры:

```
#include <iostream>

using namespace std;
struct T {
  T() { std::cout << "T::T()" << std::endl; }
  void* operator new(std::size_t size, string str) {
    auto p = ::operator new(size);
    std::cout << "T::new(" << size << ") " << p << std::endl;
    cout<<str<<endl;
    return p;
 }
 void operator delete(void* p) {
    std::cout << "T::delete(" << p << ")" << std::endl;
    if (!p) return;
    ::operator delete(p);
  }
 };
int main()
{
    T * ptr = new("Hello World") T();
    delete ptr;
    return 0;
}
```

15.5. Распределитель памяти (allocator)

В некоторых частях стандартной библиотеки C++ используются специальные объекты для выделения и освобождения памяти, которые называются распределителями памяти. В стандартной библиотеке определен распределитель памяти по умолчанию template< class T > struct allocator;

Все стандартные контейнеры используют распределители памяти:

```
template<
        class T,
        class Allocator = std::allocator<T>
> class vector;


template<
        class T,
        class Allocator = std::allocator<T>
> class list;
```

и другие контейнеры.

Стандартный распределитель можно заменить на свой. В C++ 2011 для создания своего распределителя требуется только определить конструктор, деструктор (их можно использовать по умолчания), allocate, deallocate.

Пример:

```
#include <iostream>

#include <string>
#include <vector>
template<class T>
struct DebugAllocator
{
    typedef T              value_type;

    DebugAllocator() = default;

    template<class U>
    DebugAllocator(const DebugAllocator<U>&) { }

    ~DebugAllocator() = default;

    T * allocate(std::size_t num)
    {
        std::cout << "DebugAllocator::alocate " << num << std::endl;
        return static_cast<T*>(::operator new(num * sizeof(T)));
    }

    void deallocate(T * p, std::size_t num)
    {
        std::cout << "DebugAllocator::dealocate " << num << std::endl;
        ::operator delete(p);
    }
};

int main()
```

```
{
  std::vector<int, DebugAllocator<int>> v = {1, 2, 3};
  v.push_back(10);
  v.push_back(10);
  v.push_back(10);
  v.push_back(10);
}
```

Вывод программы:

DebugAllocator::alocate 3

DebugAllocator::alocate 6

DebugAllocator::dealocate 3

DebugAllocator::alocate 12

DebugAllocator::dealocate 6

DebugAllocator::dealocate 12

Если в примере заменить vector на list, то вывод программы изменится на:

DebugAllocator::alocate 1

DebugAllocator::alocate 1

DebugAllocator::alocate 1

DebugAllocator::alocate 1

DebugAllocator::alocate 1

DebugAllocator::alocate 1

DebugAllocator::alocate 1

DebugAllocator::dealocate 1

DebugAllocator::dealocate 1

DebugAllocator::dealocate 1

DebugAllocator::dealocate 1

DebugAllocator::dealocate 1

DebugAllocator::dealocate 1

DebugAllocator::dealocate 1

Тема № 16. Преобразования и динамическая идентификация типов

16.1. Дополнительные операции преобразования типов

1. Операция приведения типов в стиле С

Операция может записываться в двух формах:

- **тип (выражение)**
- **(тип) выражение**

Считается устаревшей, не рекомендуется использовать. Так как различные ошибки, связанные с приведением типов не проверяются.

2. Операция const_cast

Операция служит для удаления модификатора const. Как правило, она используется при передаче в функцию константного указателя на место формального параметра, не имеющего модификатора const. Формат операции:

- **const_cast <тип> (выражение)**

Пример:

```
#include <iostream>
using namespace std;
void print(char *Str)
{
	cout<<endl<< Str;
}
int main(int argc, char* argv[])
{
	char const *S1 = "Hello World"; // Не разрешается менять содержимое строки
	char *const S2 = "Hello World"; // Не разрешается менять значение указателя
                                                        //S2=0;   Ошибка
	print(const_cast<char*>(S1));
	//     print(S1);  Ошибка
	print(S2);  // Допустимо
	return 0;
}
```

3. Операция dynamic_cast (ошибки выявляются на этапе выполнения программы)

Операция применяется для преобразования указателей или ссылок на объекты классов. Разрешено преобразовывать указатели без ограничений, но если преобразование не является повышающим, то результат 0 (nullptr). Для ссылок на объекты разрешено - только ссылку базового класса настраивать на объект производного класса (для ссылок остальные преобразования запрещены), при этом во время выполнения программы производится проверка допустимости преобразования.

Формат операции:

- **dynamic_cast <тип *> (выражение)**

После проверки допустимости преобразования в случае успешного выполнения операция формирует результат заданного типа, в противном случае для указателя результат равен нулю, а для ссылки порождается исключение std::bad_cast (требуется заголовочный файл #include <typeinfo>).

Преобразование из базового класса в производный называют понижающим (downcast), так как графически в иерархии наследования принято изображать производные классы ниже базовых. Приведение из производного класса в базовый называют повышающим (upcast), а приведение между производными классами одного базового или, наоборот, между базовыми классами одного производного — перекрестным (crosscast).

4. Операция static_cast (ошибки выявляются на этапе компиляции)

Операция static_cast используется для преобразования типа на этапе компиляции между:

- **целыми типами;**
- **целыми и вещественными типами;**
- **целыми и перечисляемыми типами;**
- **указателями и ссылками на объекты одной иерархии, при условии, что оно является понижающим или повышающим.**

Формат операции:

- **static_cast <тип> (выражение)**

Пример:

```
#include <iostream>
#include <typeinfo>
using namespace std;
class A
{
public:
	virtual	void print()
	{
		cout << endl << "A";
	}
};

class B : public A
{
public:
	void print()
	{
		cout << endl << "B";
	}
};

class C : public A
{
public:
	void print()
	{
		cout << endl << "B";
	}
};

class D {
public:
	virtual ~D() {}
};

int main()
{
	A a1;
	B b1;
	D d1;
	try
	{
```

```
            A &pA2 = dynamic_cast<A &>(b1); // Повышающее преобразование для ссылок
разрешено
            //B &pB=dynamic_cast<B &>(a1); // Понижающее преобразование для ссылок
запрещено возникает исключение
            B * pB1 = dynamic_cast<B *>(&a1); // Понижающее преобразование для указателей
формально разрешено, но результат 0
            cout << endl << "pB1=" << pB1; // 0
            C * pC1 = dynamic_cast<C *>(&b1); // Перекрестное преобразование для
указателей формально разрешено, но результат 0
            cout << endl << "pC1=" << pC1; // 0
            A * pA1 = dynamic_cast<A *>(&b1); // Повышающее преобразование для указателей
разрешено
            pA1->print();
            D *pD = dynamic_cast<D *>(&a1);    // Такое преобразование тоже формально
разрешено, но результат 0
            cout << endl << "pD=" << pD; // Результат 0
            //pD = static_cast<D *>(&a1);    // Запрещено
            int p1 = static_cast<int>(1.23545); //  Разрешено
            A * pA = static_cast<A *>(&b1); // Разрешено
            B * pB = static_cast<B *>(&a1); // Разрешено
            cout << endl << "pB=" << pB;
            pB->print();
            A &pA3 = static_cast<A &>(b1); // Повышающее преобразование для ссылок
разрешено
            B &pB3 = static_cast<B &>(a1); // Понижающее преобразование для ссылок
разрешено
            //C * pC=static_cast<C *>(&b1); // Запрещено
      }
      catch (const std::bad_cast & e)
      {
            std::cout << std::endl << e.what() << std::endl;
            return 1;
      }
      cout << "\nOk";
      return 1;

}
```

5. Операция reinterpret_cast

Операция reinterpret_cast применяется для преобразования не связанных между собой типов, например, указателей в целые или наоборот, а также указателей типа void* в конкретный тип. При этом внутреннее представление данных остается неизменным, а изменяется только точка зрения компилятора на данные. Разрешено все, контроль лежит на программисте.

Формат операции:

- **reinterpret_cast <тип> (выражение)**

16.2. Понятие о динамической идентификации типов

Механизм динамической идентификации типов (RTTI - Run-Time Type Identification) входит в стандарт Си++ и позволяет идентифицировать конкретные типы объектов во время выполнения программы, даже если известны только указатель или только ссылка на интересующий вас объект (классы должны быть виртуальными, т.е. содержать виртуальную функцию или виртуальный деструктор).

Для использования механизма необходимо подключить файл typeinfo.h (или typeinfo).

Механизм динамического определения (идентификации) типов позволяет проверить, является ли некоторый объект объектом заданного типа, а также сравнивать типы двух данных объектов. Для этого используется операция typeid, которая определяет тип аргумента и возвращает ссылку на объект типа const typeinfo (typeinfo является классом), описывающий этот тип. В качестве аргумента typeid можно также использовать имя некоторого типа. В этом случае typeid вернет ссылку на объект const typeinfo этого типа.

16.3. Формы операции typeid

typeid (выражение)
typeid (имя_типа)

16.4. Открытые методы класса typeinfo

int operator==(const typeinfo &) const;
int operator!=(const typeinfo &) const;
int before(const typeinfo &) const;
const **char** * name() const; (*возвращает условное имя типа, не совпадает с реальным именем*)

Функция before() возвращает истину, если вызываемый объект стоит выше в иерархии объектов, чем объект, используемый в качестве параметра. Функция before() предназначена большей частью для внутреннего использования. Возвращаемое ею значение не имеет ничего общего с иерархией классов или наследованием.
typeid(T1).before(**typeid**(T2));
Пример:

```
#include <iostream>
#include <typeinfo>

using namespace std;
class A
{
public:
    virtual     void print()
    {
        printf("\nA");
    }
};

class B : public A
{
public:
    void print()
    {
        printf("\nA");
    }
};

class D {
public:
```

```
    virtual ~D() {}
};

int main(int argc, char* argv[])
{
    A a1;
    B b1;
    D d1;
    cout<< typeid(6 * 5.6).name()<<endl;
    A *pA = &b1;
    double d = 4;
    short ii;
    if (typeid(b1) == typeid(B)) cout<< "b1 is B"<<endl;
    else cout<< "b1 is not B" << endl;
    if (typeid(b1) == typeid(A)) cout<< "b1 is A"<<endl;
    else cout<< "b1 is not A"<<endl;
    if (typeid(a1) == typeid(A)) cout<<"a1 is A"<<endl;
    else cout<<"a1 is not A" << endl;
    if (typeid(a1) == typeid(B)) printf("\na1 is B");
    else cout<<"a1 is not B"<<endl;
    cout<<"a1 is "<<typeid(a1).name()<<endl;
    cout<<"b1 is " << typeid(b1).name()<<endl;
    cout<<"d is " << typeid(d).name()<<endl;
    cout<<"*pA is " << typeid(*pA).name()<<endl;
    cout<<"pA is " << typeid(pA).name()<<endl;
    cout<<"A before B=" << typeid(A).before(typeid(B))<<endl;
    cout<<"B before A=" << typeid(B).before(typeid(A))<<endl;
    cout<<"B before d=" << typeid(B).before(typeid(d))<<endl;
    cout<<"d before ii=" << typeid(d).before(typeid(ii))<<endl;
    cout<<"ii before d=" << typeid(ii).before(typeid(d))<<endl;
    cout<<"d1 before a1="<< typeid(d1).before(typeid(a1))<<endl;
    cout<<"a1 before d1=" << typeid(a1).before(typeid(d1))<<endl;
    cout<<"type const 10=" << typeid(10ul).name()<<endl;
    return 0;
}
```