1. Структура программы на языках C и C++

Программа на C/C++ состоит из одного или нескольких файлов, образующих проект. Файлы содержат в себе описания данных и функций, использующихся в программе, функция — особый участок кода программы, решающий часть задачи.

Главная функция программы — функция main, это функция точки входа в программу, она представляет собой решение всей задачи и с неё начинается выполнение кода. Функция main не может вызывать саму себя, и должна обязательно присутствовать в коде программы.

Одной из особенностей программ на языках C и C++ является раздельная компиляция, заключающаяся в том, что каждый файл программы должен быть скомпилирован отдельно и после этого программа собирается из объектных файлов.

Для того, чтобы собрать программу на C/C++ необходимо создать заголовочные файлы, которые содержат константы (const), объявления переменных, типы объявленных функций (нельзя определять в заголовочных файлах, поскольку невозможно будет заменить файл с реализацией).

Для того, чтобы избежать многократного объявления заголовочных файлов используются макросы:

```
#pragma once
```

либо:

```
#ifndef FILE_H
#define FILE_H
...
#endif FILE_H
```

Если у группы файлов есть один заголовочный файл, то она объединяется в библиотеку.

Ссылка (кличка) — механизм передачи параметров в функцию, работающий по принципу неявного указателя.

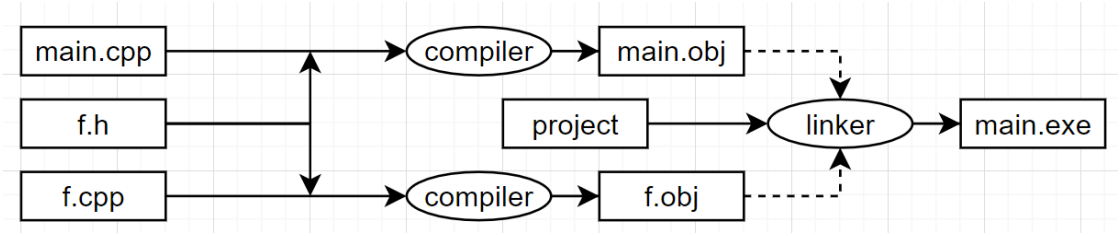
Перегрузка функции — это особенность языка C++, позволяющая создавать семейство функций с одинаковыми именами, но имеющие разные списки входных параметров и разные реализации, соответственно списку параметров.

Пример программы на C++:

```
#include f.h

int main(int argc, char* argv[])
{
	return 0;
}
```

Компиляция программы:



Отличия C++ от C:
- Добавлены классы и шаблоны
- Возможна перегрузка функций
- Новые операторы new и delete для работы с памятью
- Обработка исключений через throw/catch

2. Классы и объекты в C++. Ограничение доступа к членам класса в C++. Члены класса и объекта. Методы. Схемы наследования.

Класс — сложный тип данных, содержащий в себе набор данных (полей, атрибутов, членов класса) и функций для работы с ними (методов).

Объект класса — сущность в программе, у которой есть свойства и поведения, является экземпляром класса.

Механизмы описания класса:
С помощью struct:

```
#include <iostream>

using namespace std;

struct StructClass
{
	double a;
	void b()
	{
		cout << "b";
	}
};

int main()
{
	StructClass s_c;
	s_c.a = 1.0;
	cout << s_c.a << endl;
	s_c.b();
	return 0;
}
```

Вывод:
1
b

Задание класса с помощью структуры отличается тем, что у структуры по умолчанию поля и функции-члены задаются как публичные, а у класса, как собственные (private).

```
struct StructClass
{
private:
    double a;
public:
    void b()
    {
        cout << "b";
    }
};

int main()
{
    StructClass s_c;
    s_c.a = 1.0;
    cout << s_c.a << endl;
    s_c.b();
    return 0;
}
```

union – представляет собой структуру данных, члены которой расположены по одному и тому же адресу. Поэтому размер объединения (union) равен размеру его наибольшего члена. Данный тип структуры данных не может быть базовым и производным классом. Однако по умолчанию в union, также как и в struct все члены изначально открыты, что нарушает принцип инкапсуляции (нет доступа к данным извне) что отличает эти типы задания класса от обычного типа данных вида class, в котором по умолчанию доступ к данным и методам закрыт.

class – сложная структура данных, содержащая в себе набор данных и действий над этими данными.

Задание класса:

```
class <имя класса> : [:<список базовых классов>]
{
	<методы и поля данных>
};
```

Описание класса всегда заканчивается точкой с запятой, как и описание структуры.

Классы описываются в заголовочных файлах .hpp, а методы классов определяются в .cpp файлах.

Пример класса:

```
class A
{
private:
	int a;
protected:
	int b;
public:
	int f();
};
```

Ограничение доступа к членам класса в C++:
Всего существует три уровня доступа в C++:
1. private – приватный, или внутренний метод или поле данных класса C++, если поле имеет данный тип доступа, то доступ к нему извне получить нельзя, за исключением дружественных функций и классов.
2. protected – защищённый, к методам и данным с данным типом доступа могут иметь доступ только методы данного класса и методы производных классов.
3. public – общий или публичный, данный тип доступа позволяет получать доступ извне к полям или методам данного класса.

Методы — функции данного класса.
Члены класса располагаются по наследованию в следующем порядке:
1) private
2) protected
3) public
Если создаём библиотеку, то в обратном порядке.

Пример наследования:

```
class A
{
private:
	int a;
	int fa();
protected:
	int fb();
	int b;
public:
	int c;
	int fc();
};

class B : public A
{
private:
	int d;
	int fd();
protected:
	int e;
	int fe();
public:
	int f;
	int ff();
};
```

Вместо public может быть любой из трёх видов наследования.
При изменении вида наследования изменяется доступ к методам и данным исходного класса в зависимости от того, какой тип доступа имело данное поле.

Вид наследования	Объявление компонентов в базовом классе	Видимость компонентов в производном классе
private	private	не доступны
	protected	private
	public	private
protected	private	не доступны
	protected	protected
	public	protected
public	private	не доступны
	protected	protected
	public	public

Статические члены класса:
static может быть как метод, так и поле данных.
Особенности статических членов класса:
- данный член класса является общим для всех экземпляров (объектов) данного класса, то есть является членом класса, а не членом объекта.
- инициализация статического члена происходит с помощью оператора расширения области видимости:

```
class A
{
public:
	int a;
	static int b;
};

int main(void)
{
	A::b = 0;
	A::a = 0;
	return 0;
}
```

- к данному полю класса нельзя применить указатель this
- вызывается для класса
- можно вызвать без создания объекта

Константные члены класса:
Определяются с помощью модификатора const.
- не меняются в течении жизни объекта.
- инициализируются в конструкторе.
- может быть создан константный объект, имеющий константные методы.
- для константного объекта возможен вызов только константных методов.

Ссылка:

Ссылка это тип данных, использующийся для передачи каких-либо данных в функцию без указателя.
- позволяет использовать несколько имён для одного объекта.
- используется для изменения и передачи больших значений.

3. Создание и уничтожение объектов в C++. Конструкторы и деструкторы. Виды конструкторов и способы создания объектов.

Для каждого класса в C++ выделяется метод, называемый конструктор. Этот метод вызывается при создании объекта данного класса, и инициализирует его определённым образом. У данного метода отсутствует тип возврата и его можно перегружать. Однако при этом конструктор не наследуется. Если конструктор имеет вид доступа private, то невозможно создание производных классов.

В каких случаях вызывается конструктор:
1) При определении для статических и внутренних объектов. Выполняется до функции main().
2) При определении локальных объектов.
3) При выполнении оператора new
4) Для временных объектов.

Конструктор у класса должен существовать всегда, если конструктор не задан явно, то по автоматически создаются два конструктора:
1. Конструктор по умолчанию (default)
2. Конструктор копирования — принимает ссылку на константный объект, вызывается при:
1) При инициализации одного объекта другим.
2) При передаче по значению параметров.
3) При возврате по значению.

Примеры конструкторов:
```
#include <iostream>

using namespace std;

class Summator
{
private:
	double _a, _b;
public:
	Summator() { _a = 0, _b = 0; };// стандартный конструктор
	Summator(double a) { _a = a; _b = 0; };// конструктор с инициализатором для первого
параметра
	Summator(double a, double b) { _a = a; _b = b; };// конструктор с инициализатором для
второго параметра
	Summator(Summator& sum) { _a = sum._a; _b = sum._b; };// конструктор копирования
```

```
	void get();
};

void Summator::get()
{
	cout << _a << ' ' << _b << endl << endl;
};

int main(void)
{
	//Summator s();//функция без параметров, возращающая объект класса Summator
	// по какой-то причине оно не копилируется, может быть из-за изменений в стандарте
	Summator s_1;//пример стандартной инициализации
	Summator s_2_1(1.0);//пример конструктора с одним параметром
	Summator s_2_2 = 1.0;
	Summator s_3_1(1.0, 2.0);//пример конструктора с двумя параметрами
	Summator s_3_2;
	s_3_2 = Summator (1.0, 2.0);//дефолтный конструктор копирования с двумя параметрами
	Summator s_4_1(s_3_1);// конструктор копирования при создании нового объекта
	Summator s_4_2;
	s_4_2 = s_3_2; //конструктор копирования через оператор присваивания

	//Summator s_5;
	//s_5= s();

	s_1.get();
	s_2_1.get();
	s_2_2.get();
	s_3_1.get();
	s_3_2.get();
	s_4_1.get();
	s_4_2.get();
	//s_5.get();

	return 0;
}
```

Результат работы программы:

Конструкторы класса не могут быть volatile (квалификатор типа, который используется для объявления о том, что объект может быть изменен в программе аппаратным обеспечением), static и const.

Деструктор — функция-член класса (метод класса), которая вызывается автоматически, когда объект выходит из области действия, или явно уничтожается с помощью вызова метода delete. Деструктор имеет то же имя, что и класс, которому он принадлежит, перед именем деструктора при объявлении ставится знак тильды (~). Например для предыдущего класса деструктор будет ~Summator(). Если деструктор не определён, то компилятор будет предоставлять его по умолчанию, и для многих классов этого достаточно. Пользовательский деструктор необходимо определять, если класс хранит дескрипторы для системных ресурсов, которые необходимо освободить, или указатели на память.

В C++ реализуется неявный вызов деструктора. Деструктор не принимает никаких параметров, и нет типа возврата. Деструкторы вызываются в обратном порядке, для локальных статических объектов вызывается деструктор до уничтожения глобальных статических объектов, но после выполнения программы. Временные объекты уничтожаются, когда в них отпадает надобность. Деструктор не перегружается.

Деструктор не может быть const, volatile, static, но может быть virtual (объявляет функцию, которая определяется в базовом классе, а любой порожденный класс может ее переопределить).

4. Наследование в C++ и построение иерархии классов. Множественное наследование. Понятие доминирования. Порядок создания и уничтожения объектов. Неоднозначности при множественном наследовании.

Новые классы могут быть производными от существующих классов с помощью механизма наследования. Наследование — механика языка C++, которая позволяет создавать производные классы (классы наследники), взяв за основу все методы и элементы базового класса (класса родителя).

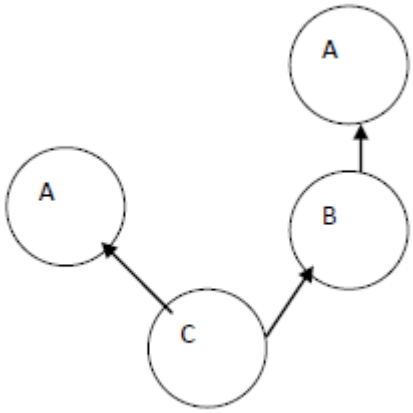
Основные моменты наследования — расширение класса и выделение общей части из разных классов.

Причины необходимости выделения общей части:
1) Общая схема использования.
2) Сходство между наборами операций
3) Сходство реализации

Расщепление классов:
1) Два подмножества операций в классе используются в разной манере
2) Методы класса имеют не связную реализацию.

3) Класс оперирует очевидным образом в двух несвязных обсуждениях проекта.

Прямая база — непосредственная база класса, прямая база может входить в класс только один раз.
Косвенная база — база класса, которая может входить в класс несколько раз.

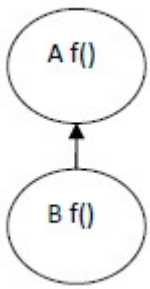


Для приведённого выше рисунка B – прямая база для C, а A – и косвенная и прямая.

Если у класса больше одного предка, то наследование называется множественным.

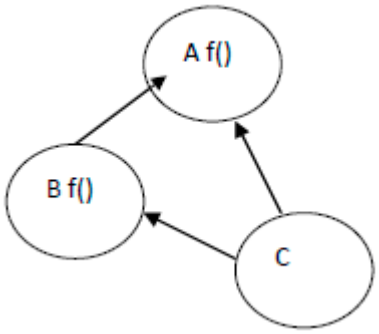
Преимущества и недостатки множественного наследования:
+ Гибкая схема наследования
+ Уменьшение иерархии наследования
- Неоднозначности при определении прямой и косвенной базы

Доминирование — перекрытие методов базового класса методами производного. Доминирование не имеет ничего общего с механизмом работы виртуальных функций, поскольку при наличии virtual используется таблица виртуальных функций.



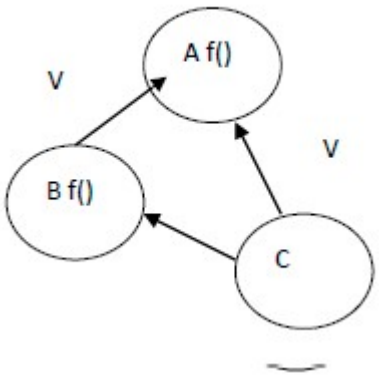
В приведённом выше примере метод f() производного класса B доминирует над методом f() в доминантном классе A.

Пример неоднозначности при наследовании:



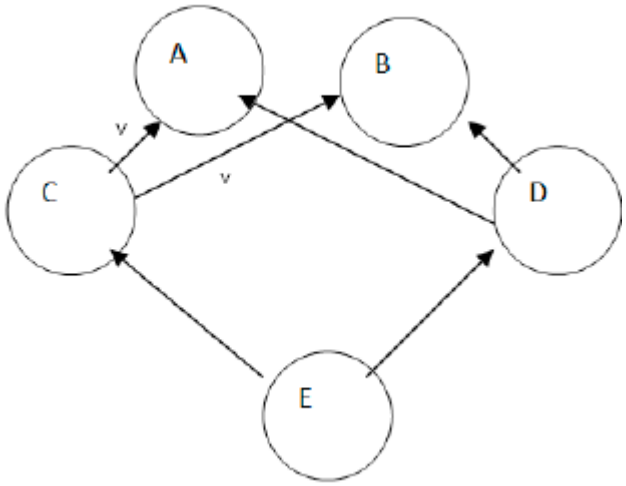
В данном случае, так как C наследуется и от A и от B получается неоднозначно, какой именно метод f() вызывать, метод f() класса B, или метод f() класса A.

Виртуальное наследование - один из вариантов наследования, который нужен для решения некоторых проблем, порождаемых наличием возможности множественного наследования (особенно «ромбовидного наследования»), путём разрешения неоднозначности того, методы которого из суперклассов (непосредственных классов-предков) необходимо использовать. Оно применяется в тех случаях, когда множественное наследование вместо предполагаемой полной композиции свойств классов-предков приводит к ограничению доступных наследуемых свойств вследствие неоднозначности. Базовый класс, наследуемый множественно, определяется виртуальным с помощью ключевого слова virtual.
Важной особенностью виртуального наследования является то, что конструирование виртуальных базовых классов производит конструктор того класса, объект которого мы создаём, а не конструкторы тех классов, которые производят виртуальное наследование.
При виртуальном наследовании неоднозначность устраняется за счёт того, что виртуальные функции базового класса фактически становятся общими для классов-наследников, и следовательно без явного переопределения вызывается функция базового класса:



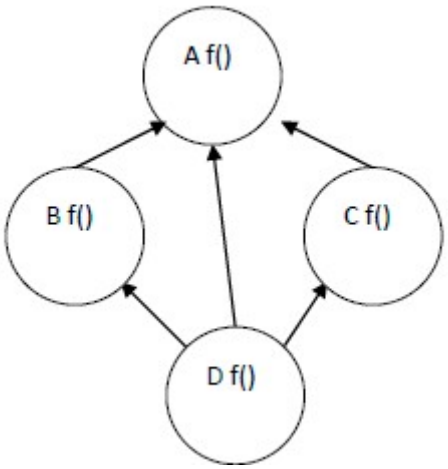
В данном случае будет вызвана функция f() класса B, т.к. класс A наследуется от классов B и C виртуально.

Ещё один пример виртуального наследования:



В данном случае классы A и B являются базовыми. И для того, чтобы исключить неопределённости при наследовании E от C и D мы используем виртуальное наследование.

```
class A {};
class B {};
class C:virtual public A,
virtual public B{};
class D:virtual public A,
virtual public B{};
class E:public C, public D{};
```



В случае выше возникает неопределённость относительно того, какой метод f() использовать:

```
void D::f()
{
A::_f();
B::_f();
C::_f();
_f();
}
```

Данная проблема решается с помощью:

- Вызова метода конкретного суперкласса.

(D_obj.A::f())

- Обращения к объекту подкласса как объекту определённого суперкласса.

(static_cast<A&>(D_obj).f())
- Переопределения проблемный метода в последнем дочернем классе.

```
void f() override;
```

Пример:

```
class A
{
public:
	int a;
	int (*b)();
	int f();
	int f(int);
	int g();
};
class B
{
private:
	int a;
	int b;
public:
	int f();
	int g;
	int h();
	int h(int);
};
class C:public A, public B{};
void f (C *pc)
{
	pc->a = 1; //Error! Проверка на неоднозначность происходит
	pc->b (); //Error! до проверки на степень доступа.
	pc->f (); //Error!
	pc->f (1); //Error!
	pc->g = 1; //Error!
	pc->h(); pc->h(1); // OK!
}
```

Решение проблемы — переопределить неоднозначные методы в классе C.

Порядок создания и уничтожения объектов:

1. Инициализация виртуальных базовых классов — инициализируются в том порядке, в котором они находятся в дереве наследования, если бы мы обходили его «слева направо», то есть вне зависимости от того, как они будут перечислены в списке инициализации. Конструктор каждого класса будет вызван единожды.
2. Инициализация прямых базовых классов — происходит в той же последовательности, в которой они указаны в списке наследования класса, вне зависимости от того, как они будут перечислены в списке инициализации.
3. Инициализация полей объектов:
	1) Константные поля, поля-ссылки и оператор копирования.
	2) Инициализация статических полей.
	3) Порядок инициализации полей — поля инициализируются строго в том порядке, в котором они объявлены в классе вне зависимости от порядка в списке инициализации.

4. Выполнение тела конструктора.

5. Полиморфизм в C++. Понятие абстрактного класса. Дружественные связи.

Полиморфизм — использование объектов с одинаковым интерфейсом без информации о типе и внутренней структуре объекта (один интерфейс, множество реализаций).

Виртуальные методы — методы, которые предполагаемо будут переопределены в производных класса. Ключевое слово — virtual. Гарантируется, что будет выбрана верная функция.

Виртуальный метод может быть дружественным (friend) и inline.

Пример:

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

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

class C: public B{
public:
	C(){}
	void f();
};

C c; //A::f()
c.g(); //C::f()
```

Чисто виртуальная функция не имеет тела, если класс имеет хотя бы одну чисто виртуальную функцию, то он называется абстрактным.

```
class A {
public:
	virtual void f() = 0;
};
```

Класс, который не реализует чисто виртуальный метод, однако имеет его в своём базовом классе также называется абстрактным.

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

Дружба — дружба позволяет дружественному классу получать доступ ко всем членам методов других классов. Для того, чтобы указать на дружественный класс, указываем, что есть «друг».

Пример:

```
class CA
{
	friend class CB;
};
```

```
class CB{};
```
Схема является зависимой, то есть при изменении CA приходится менять CB, однако вполне можно вместо дружественного класса ввести «дружественное» отношение к методу:
```
class CA
{
	friend class CB;
	friend int CB::f(CA&);
};

class CB
{
public:
	int f(CA&);
};
```
Для улучшения качества кода необходимо, чтобы дружественных отношений было как можно меньше, так как дружба не наследуется (сын друга не друг), не транзитивна (друг моего друга не друг).

Особенности полиморфизма в C++:
- Если базовый и производный классы имеют общий открытый интерфейс, говорят, что производный класс представляет собой подкласс базового.
- Отношение между классом и подклассом, позволяющее указателю или ссылке на базовый класс без вмешательства программиста адресовать объект производного класса, возникает в C++ благодаря поддержке полиморфизма.
- Полиморфизм позволяет предложить такую реализацию ядра объектно-ориентированного приложения, которая не будет зависеть от конкретных используемых подклассов.
- Динамическая идентификация типов времени выполнения обеспечивает специальную поддержку полиморфизма и позволяет программе узнать реальный производный тип объекта, адресуемого по ссылке или указателю на базовый класс. Поддержка RTTI в C++ реализована двумя операциями:
1) dynamic_cast – поддерживает преобразование типов времени выполнения.
2) typeid – идентифицирует реальный тип выражения.
Операции RTTI – это события времени выполнения для классов с виртуальными функциями и события времени компиляции для остальных типов. Исследования RTTI-информации полезно при решении задач системного программирования.
dynamic_cast:
- Встроенная унарная операция dynamic_cast языка C++ позволяет
1. Безопасно трансформировать указатель на базовый класс в указатель на производный класс, с возвратом nullptr при невозможность выполнения трансформации.

2. Преобразовывать леводопустимые выражения (выражения, ссылающиеся на переменные), ссылающиеся на базовый класс, в ссылка на производный класс (при ошибке возбуждается bad_cast)

Пример для преобразования указателей:

```
Alpha *alpha = new Beta;
if(Beta *beta = dynamic_cast<Beta*>(alpha)) {
// успешно…
}
else {
// неуспешно…
}
```

Пример для преобразования ссылок:

```
#include <typeinfo> // для std::bad_cast
void foo(Alpha &alpha) {
// …
try {
Beta &beta = dynamic_cast<Beta&>(alpha))
}
catch(std::bad_cast) { /* … */ }
}
```

Операция typeid:

1. Позволяет установить фактический тип выражения-операнда.
2. Может использоваться с выражениями и именами любых типов (включая выражения встроенных типов и константы).

Если оператор typeid принадлежит типу класса с одной и более виртуальными функциями (но не указателю на него), результат typeid не может не совпадать с типом самого выражения.

Операция typeid имеет тип (возвращает значение типа) type_info и требует подключение заголовочного файла <typeinfo>.

Реализация класса type_info зависит от компилятора, но в общем случае позволяет получить результат в виде константной С-строки, присваивать объекты type_info друг другу (оператор =), а также сравнивать их на равенство и неравенство (оператор == и оператор !=).

6. Перегрузка операторов в С++.

Перегрузка оператора — изменение действия, которое будет выполнять оператор при работе с данным типом/объектом класса. Перегрузка подразумевает под собой создание функции, которая содержит слово operator и символ перегружаемого оператора. Функция оператора может быть определена как член класса, либо вне класса.

Всего существует два основных способа перегрузки операторов: глобальные функции, дружественные для класса, или подставляемые функции самого класса.

Пример:

```
class Integer
{
	//унарный +
	friend const Integer& operator+(const Integer& i);
	bool operator==(const Integer& left, const Integer& right) {
		return left.value == right.value;
}
```

Особенности перегрузки операторов:

- Нельзя перегрузить операторы «.», «::»(оператор расширения области видимости), «,», «?:»(тернарный оператор — сокращённый способ ветвления (условние) ? выражение : другое_выражение;), sizeof, typeid, ".*" (выбор члена через указатель на член).
- При перегрузке не меняется сущность, приоритет и порядок выполнения.
- При перегрузке не меняется смысл (по крайней мере это желательно к выполнению):

1) Операторы =, (), [], →, →* перегружаются только как члены класса.
2) Оператор = не наследуется и возвращает ссылку или void.
3) Перегружать () можно только один раз.
4) [] - ассоциативные массивы (выбор по критерию).
5) →, →* - возвращает указатель или ссылку на объект.
6) <знак>= - лучше как член класса.
7) Унарные операторы (имеют только один операнд, например ++, –, ! и тд.) - лучше как член класса.
8) Бинарные, если объект — член класса, если создание объекта — выполняется перегрузка.
9) ++a // a.operator++() - префиксный инкремент как унарный оператор.
10) a++ // a.operator++(0) – инфиксный инкремент как бинарный оператор.
11) В случае переноса и копирования:

```
class A
{
public:
	A(const A&); // Копирование
	A(A&&); // Перенос
	A& operator =(const A&); // Копирование
	A& operator =(A&&);
private:
	int *buf;
};
A& A::operator =(const A& obj)
{
	delete buf;
	buf = new int[sizeof(obj)];
	copy(buff, obj.buf);
	return *this;
}
A& A::operator =(A&& obj)
{
	delete buf;
	buf = obj.buf;
	obj.buf = nullptr;
```

```
        return *this;
}
```

12) new и delete – только как члены класса:

```
class A
{
public:
void* operator new(size_t size); // new A
void* operator new(size_t size, void *p); // new(buff) A
void* operator new [](size_t size); // new A[n]
void operator delete(void *p); // delete pobj
void operator delete[](void *p); // delete []p
};
```

Пример к (5):

```
class A{
public:
        void f(){}
};
class B{
public:
        A* operator ->(){
}
};
B obj;
obj->f();
obj.operator ->()->f(); //то же самое, что и выше
```

Пример к (8):

```
complex complex:: operator +(const complex &c1, const complex &c2)
{
        complex c(c1.ze + c2.ze, c1.im + c2.im);
        return c;
}
c1 = c2 + c3;
```

7. Шаблоны функций и классов в C++. Специализация шаблонов частичная и полная.

Шаблоны функций — инструкции, согласно которым создаются локальные версии шаблонированной функции для определённого набора параметров и типов данных.

Пример шаблонной функции, выводящей элементы массива вне зависимости от их типа:

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

using namespace std;

template <typename T>
void printArray(const T* array, size_t count)
{
        for (size_t ix = 0; ix < count; ix++)
                cout << array[ix] << " ";
        cout << endl;
}
```

Шаблоны классов — инструкции, согласно которым создаются локальные версии шаблонного класса для определённого набора параметров и типов данных.
Любой шаблон начинается со слова template, после ключевого слова template следуют угловые скобки < > в которых перечисляется список параметров шаблона. Каждому параметру должно предшествовать зарезервированное слово class или typename:
template <class T>
template <typename T>
template <typename T1, typename T2>
typename – в шаблоне используется встроенный тип данных, такой как int, double, float, char и тд.
class – в качестве параметров будут использоваться пользовательские типы данных.

В общем случае шаблонные классы позволяют получать классы, отличающиеся типом только в отдельных местах.

```
template <typename T>
class A
{
	T* u;
public:
	void f();
};
```

Также у классов могут быть шаблонные методы:

```
template <typename T>
void A(t)::f()
{
}
```

Помимо этого шаблоны могут иметь специализацию для отдельного типа данных:

```
template<>
class A<float>
{
}
```

Частичная специализация шаблонов — механизм языка C++, предназначенный для специализации обобщённый шаблонных классов под конкретные задачи или под конкретное множество своих параметризованных типов данных. Частичная специализация шаблона класса определяется как некая конфигурация параметров первичного класса, которая служит аргументом специализированной реализации. Примером такой специализации может служить любой класс-контейнер, реализованный для хранения объектов указателей.

```
// первичный класс
template <typename T>
class Vector
```

```
{ /* ... реализация класса-контейнера для объектов типа T... */ };
// частичная специализация первичного класса для хранения указателей
template <typename T>
class Vector<T*>
{ /* ... реализация класса-контейнера для указателей на объекты типа T... */ };
```

```
template<typename T1, typename T2>
class A{…}

template<typename T>
class A<T,T>{…}

template<typename T>
class A<T,int>{…}

template<typename T1, typename T2>
class A<T1*,T2*>{…}

A<int,float>a1;
A<float,float>a2;
A<float,int>a3;
A<int*,float*>a4;
A<int,int>a5; // ошибка
A<int*,float*>a6; // ошибка
```

Также можно указывать значения шаблонных параметров по умолчанию в конце списка параметров:
template <typename T=int>

Параметры-значения: только константные внешние объекты:
template <char *const name>
class A {}

extern char const st[] = “name”
A<st>obj;

8. Обработка исключительных ситуаций в C++. Пространства имён.

Исключение — проблема, возникающая во времени выполнения программы. Исключение C++ - ответ на исключительное обстоятельство, которое возникает во время работы программы, к примеру попытка деления на ноль.

Исключения обеспечивают способ передачи контроля из одной части в программы в другую. Обработка исключений C++ построена - на трёх командах:

try – блок try идентифицирует блок кода, для которого будут активированы определённые исключения. За ним следует один, или несколько блоков catch.

catch – программа выхватывает исключение с обработчиком исключений в месте в программе, где проблема, возникшая в коде должна быть обработана. Ключевое слово catch указывает на отлов исключения.

throw – программа выдаёт исключение, когда возникает проблема. Это делается с использованием ключевого слова throw.

Выброс ошибки в коде:

```
if(...) throw <имя>();
```

Отлов ошибки:

```
try
{
...
}
catch (<имя> & идентификатор)
{
...
}
```

В качестве выбрасываемого объекта передаём объект класса, отвечающий за определённый тип ошибки, но унаследованный от базового класса ошибки.

В общем случае идея механизма отлова ошибки состоит в том, чтобы передавать управление как обработчик некоторой ошибки. Проблема состоит в том, что нужно возвращаться в точку возникновения ситуации.
Каждый класс должен сам отвечать за свою исключительную ситуацию.
Классы, которые отвечают за обработку исключительных ситуаций должны быть родственными. И тогда если мы создаём новый класс, то передаём указатель в класс-обработчик:

```
try
{
        throw <выражение>; создание объекта класса
}

catch (<параметр>)
{
        …
}
```

Каким образов нельзя обрабатывать ошибки:

```
try
{
        A* pobj;
        pobj->f();
        delete pobj;
}
```

Естественный порядок функционирования программ нарушается возникающими нештатными ситуациями, в большинстве случаев связанных с ошибками времени выполнения (в некоторых случаях с необходимостью внезапно переключить контекст приложения).
В языке C++ такие нештатные ситуации называются исключительными (иначе говоря — исключениями).
Примеры данных ситуаций:

1. Нехватка оперативной памяти.
2. Попытка доступа к элементу коллекции по некорректному индексу.
3. Попытка недопустимого преобразования динамических типов и прочее.

Архитектурной особенностью механизма обработки исключительных ситуаций в языке C++ является принципиальная независимость (несвязность) фрагментов программы, где исключение возбуждается и где оно обрабатывается. Обработка исключительных ситуаций носит невозвратный характер.

Носителями информации об аномальной ситуации (исключении) в C++ являются объекты заранее выбранных на то роль типов (пользовательских или базовых, например char*). Такие объекты называются объектами — исключениями. Жизненный цикл объектов-исключений начинается с возбуждения исключительной ситуации посредством оператора throw:

```
throw "Illegal cast"; // char *
throw IllegalCast(); // class IllegalCast
enum EPrgStatus {OK, BADINDEX, ILLEGALCAST};
throw ILLEGALCAST; // enum EPrgStatus
```

Исключение, для обработки которого не найден catch-блок, инициирует запуск функции terminate(), передающей управление функции abort(), которая аварийно завершает программу.
Стандартная библиотека языка C++ содержит собственную иерархию классов исключений, являющихся прямыми или косвенными потомками базового класса exception. Потомки класса exception условно представляют две категории ошибок: логические ошибки и ошибки времени исполнения.

Пространство имён — это декларативная область, в рамках которой определяются различные идентификаторы (имена типов, функций, переменных , и т.п.). Пространства имён используются для организации кода в виде логических групп с целью избежания конфликтов имён, которые могут возникнуть, особенно в таких случаях, когда база кода включает несколько библиотек. Все идентификаторы в пределах пространства имён доступны друг другу без уточнения. Идентификаторы за пределами пространства имён могут обращаться к членам с помощью полного имени для каждого идентификатора, например std::vector<std::string> vec;, или с помощью объявления using для одного идентификатора (using std::string) или директивы using для всех идентификаторов в пространстве имён (using namespace std;). Код в файлах заголовков всегда должен содержать полное имя в пространстве имён.

Конфликт имён — возникает, когда два одинаковых идентификатора находятся в одной области видимости и компилятор не может понять, какой из этих двух следует использовать в конкретной ситуации.

Для того, чтобы избежать конфликты имён необходимо объявить в заголовочном файле пространство имён.
Пример.
У нас есть два файла:
boo.h
```
// Функция doOperation() выполняет операцию сложения своих параметров
int doOperation(int a, int b)
{
    return a + b;
}
```
doo.h
```
// Функция doOperation() выполняет операцию вычитания своих параметров
int doOperation(int a, int b)
{
    return a - b;
}
```
и main.cpp
```
#include <iostream>
#include "boo.h"
#include "doo.h"

int main()
{
    std::cout << doOperation(5, 4); // какая версия doOperation() выполнится здесь?
    return 0;
}
```
Если boo.h и foo.h скомпилировать отдельно, то ошибок не будет, если же их соединить в одной программе, то это приведёт к конфликту имён.

Пространство имён в C++ определяет область кода, в которой гарантируется уникальность всех идентификаторов. По умолчанию, глобальные переменные и обычные функции определены в глобальном пространстве имён.
Для избежания конфликта в примере выше объявим пространство имён в каждом заголовочном файле следующим образом:
```
// Функция doOperation() выполняет операцию сложения своих параметров
namespace Boo
{
    int doOperation(int a, int b)
    {
        return a + b;
    }
}

// Функция doOperation() выполняет операцию вычитания своих параметров
namespace Doo
{
    int doOperation(int a, int b)
    {
        return a - b;
    }
}
```
Однако теперь при выполнении main.exe программа выдаст ошибку, поскольку мы не указали, в каком именно пространстве имён выполняется данная

функция. Для того, чтобы уточнить этот момент используется оператор разрешения области видимости (::).

```
int main()
{
    std::cout << Doo::doOperation(5, 4);
    return 0;
}
```

Либо можно сделать таким образом:

```
using namespace Doo;

int main()
{
    std::cout << doOperation(5, 4);
    return 0;
}
```

Также оператор можно использовать без префикса (::doOperation), так мы ссылаемся на глобальное пространство имён.

Если у пространств имён одинаковые имена, то при включении заголовочных файлов всё, что находится внутри блоков имён с одинаковыми именами, считается частью только этого блока.

Одни пространства имён могут быть вложены в другие пространства имён.

```
namespace Boo
{
    namespace Doo
    {
        int doOperation(int a, int b)
        {
            return a - b;
        }
    }
}

int main()
{
    std::cout << Boo::Doo::doOperation(5, 4);
    return 0;
}
```

Для облегчения работы с такими конструкциями существуют псевдонимы, к примеру можно сделать следующим образом:

```
namespace Foo = Boo::Doo;

int main()
{
    std::cout << Foo::doOperation(5, 4);
    return 0;
}
```

9. Умные указатели в C++: unique_ptr, shared_ptr, weak_ptr. Использование weak_ptr на примере паттерна итератор.

Умный указатель — это класс, предназначенный для управления динамически выделенной памятью и обеспечения освобождения (удаления) выделенной памяти при выходе объекта этого класса из области видимости.

В общем случае умный указатель это шаблон класса, который объявляется в стеке и инициализируется с помощью необработанного указателя, указывающего на размещённый в куче объект. После инициализации интеллектуальный указатель становится владельцем необработанного (raw) указателя. Это означает, что интеллектуальный указатель отвечает за удаление памяти, заданной необработанным указателем. Деструктор интеллектуального указателя содержит вызов для удаления, и поскольку сам умный указатель объявлен в стеке, его деструктор вызывается, как только интеллектуальный указатель оказывается вне области, даже если исключение создаётся где-либо в другой части стека.

Доступ к инкапсулированному (к данным и методам) указателю осуществляется стандартно с помощью операторов указателя → и *, которые перегружены в классе умного указателя для возврата инкапсулированного необработанного указателя.

Важные шаги использования интеллектуальных указателей:
1. Объявление умного указателя, как автоматической (глобальной) переменной. (не используется new, malloc)
2. В параметре типа указывается тип, на который указывает умный указатель.
3. Передаётся необработанный указатель на новый (new) объект в конструкторе умного указателя. (Некоторые служебные функции или конструкторы умных указателей делают это автоматически).
4. Используются операторы → и * для доступа к объекту.
5. Умный указатель удаляет объект автоматически.

Умные указатели предоставляют способ прямого доступа к необработанному указателю.

Все умные указатели находятся в заголовочном файле <memory>.

Существует три основных умных указателя:
unique_ptr – обеспечивает, чтобы у базового указателя был только один владелец. Используется как вариант по умолчанию, кроме случая, когда понятно, что точно потребуется shared_ptr. Может быть передан новому владельцу, но не может быть скопирован, или сделан общим. Заменяет auto_ptr, использование которого не рекомендуется.

shared_ptr – умный указатель с подсчитанными ссылками. Используется, когда необходимо присвоить один необработанный указатель нескольким владельцам, например, когда копия указателя возвращается из контейнера, но требуется сохранить оригинал. Необработанный указатель не будет удалён до тех пор, пока все владельцы shared_ptr не выйдут из области видимости, или не откажутся от владения.
weak_ptr – умный указатель для особых случаев использования, обычно работает в связке в shared_ptr. weak_ptr предоставляет доступ к объекту, который принадлежит одному, или нескольким экземплярам shared_ptr, но не участвует в подсчёте ссылок. Используется, когда необходимо отслеживать объект, но не требуется, чтобы он оставался в активном состоянии. Требуется в нескольких случаях для разрыва циклических ссылок между экземплярами shared_ptr. weak_ptr не считается владельцем объекта.

Пример использования weak_ptr в паттерне итератор:
Итератор — поведенческий паттерн проектирования, позволяющий последовательно обходить элементы составных объектов, не раскрывая их внутреннего представления.

```
#pragma once
#include "matrix_const_iterator.hpp"

// Weak ptr for data access
template <typename T>
using ptrWeak = std::weak_ptr<T>;

// Matrix class
template <typename T>
class Matrix;

// Iterator class derived from std::iterator
template <typename T>
class MatrixIterator : public MatrixConstIterator
{
public:
	// Iterator constructor
	MatrixIterator(const Matrix<T> &matrix, const size_t index = 0) :
		_index(index), _rows(matrix._rows), _cols(matrix._cols), _data(matrix._data)
{}
	// Copy constructor
	MatrixIterator(const MatrixIterator& refIterator) = default;

	// Pointer operations
	T& operator*();
	T& value();

	// Overloading reference operator
	T* operator->();

	// Increment operator and method
	MatrixIterator<T>& operator++() noexcept;
	MatrixIterator<T>& next() noexcept;
	MatrixIterator<T> operator++(int) noexcept;
```

```
        // Decrement operator and method
        MatrixIterator<T>& operator--() noexcept;
        MatrixIterator<T>& previous() noexcept;
        MatrixIterator<T> operator--(int) noexcept;

        // Assign operator
        MatrixIterator<T>& assign(const MatrixIterator<T>& refIterator)  noexcept;
        MatrixIterator<T>& operator=(const MatrixIterator<T>& retIterator) noexcept;

private:
        // Check interator validity
        void _iterValid();
        // Index
        size_t _index = 0;
        // Matrix rows
        size_t _rows = 0;
        // Matrix columns
        size_t _cols = 0;
        // Weak ptr for matrix data access
        ptrWeak<typename Matrix<T>::MatrixRow[]> _data;
};
```

В случае итератора, weak_ptr используется для того, чтобы хранить в объекте итератора указатель на данные, по которым он проходит. Когда итератор выходит из области видимости, так как указатель на данные является weak_ptr, то он просто уничтожается сам, но оставляет данные, если на них указывает какой-либо другой умный указатель, поскольку не владеет этими данными, а является наблюдателем относительно них.