Практические вопросы

1. Структура программы на языках C и C++. Функции C и C++. Перегрузка функций в C++. Параметры функций по умолчанию.

Программа состоит из набора файлов, который включает в себя объявление определений.

Существуют абстракции по:

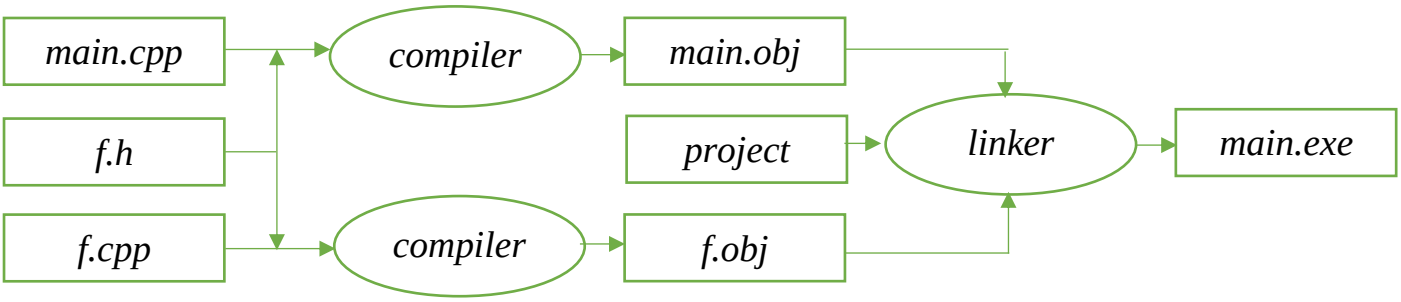
- Действию:
  - Функции - можем объявлять и определять
- Данным:
  - Типы данных - можем объявлять и определять
  - Переменные - можем объявлять и определять
  - Константы

Выполнение программы как правило начинается с ф-ции *"main"*, она обязательно должна присутствовать. Существуют приоритеты выполнения ф-ций.

Функции могут быть вложенными. Любая ф-ция может вызвать любую другую ф-цию, в том числе и саму себя (кроме main).

Для использования файлов реализации, все что мы хотим использовать необходимо сначала определить, либо объявить в том месте, где хотим использовать. Для удобства создаются заголовочные файлы, в кот. выносятся определения.

Процесс сборки:



В заголовочном файле мы можем определять константы времени компиляции, типы данных. Не можем переменные.

Значение из ф-ции может быть возвращено непосредственно, либо через аргумент.

Перегрузка функций – возможность создавать функции с одними и теми же названиями. Можно определить функцию, которую необходимо вызвать по кол-ву параметров, или по типу данных аргументов.

```cpp
void func1(int& x) { cout << "func1(int&)" << endl; }
void func1(const int& x) { cout << "func1(const int&)" << endl; }

void func2(int x) { cout << "func2(int)" << endl; }
void func2(int& x) { cout << "func2(int&)" << endl; }

void func3(const int& x) { cout << "func3(const int&)" << endl; }
void func3(int&& x) { cout << "func3(int&&)" << endl; }

void func4(int& x) { cout << "func4(int&)" << endl; }
void func4(int&& x) { cout << "func4(int&&)" << endl; }

void func5(int x) { cout << "func5(int)" << endl; }
void func5(int&& x) { cout << "func5(int&&)" << endl; }

void func6(int x) {}
void func6(const int& x) {}


    int i = 0;          const int ci = 0;       int& lv = i;
    const int& clv = ci;        int&& rv = i + 1;

    func1(i);           // int&
    func1(ci);          // const int&
    func1(lv);          // int&
    func1(clv);         // const int&
    func1(rv);          // int&
    func1(i + 1);       // const int&
    cout << endl;

    // func2(i);        // Error!
    func2(ci);          // int
    // func2(lv);       // Error!
    func2(clv);         // int
    // func2(rv);       // Error!
    func2(i + 1);       // int
    cout << endl;

    func3(i);           // const int&
    func3(ci);          // const int&
    func3(lv);          // const int&
    func3(clv);         // const int&
    func3(rv);          // const int&
    func3(i + 1);       // int&&
    cout << endl;

    func4(i);           // int&
    // func4(ci);       // Error!
    func4(lv);          // int&
    // func4(clv);      // Error!
    func4(rv);          // int&
    func4(i + 1);       // int&&
    cout << endl;

    func5(i);           // int
    func5(ci);          // int
    func5(lv);          // int
    func5(clv);         // int
    func5(rv);          // int
    // func5(i + 1);    // Error!

    // func6(i);        // Error!
    // func6(ci);       // Error!
    // func6(lv);       // Error!
    // func6(clv);      // Error!
    // func6(rv);       // Error!
    // func6(i + 1);    // Error!
```

В C++ один или несколько аргументов функции могут задаваться по умолчанию. Для каждого параметра значение по умолчанию можно указать не более одного раза, но каждое последующее объявление функции, а также определение функции может назначать параметрам значения по умолчанию. Параметры по умолчанию должны быть последними в списке параметров.

```
// Пример: функция сортировки по возрастанию массива вещественных чисел
void sort(double *arr, int count, int key = 0);

// key показывает направление сортировки (0 - по возрастанию, 1 - по убыванию).
// По умолчанию сортировка будет по возрастанию
// (key = 0, если в функцию передаются только два параметра - указатель на массив
// и количество элементов в массиве).
```

2. Ссылки. lvalue и rvalue ссылки. Передача параметров в функции по ссылке. Автоматическое выведение типа.

Ссылка – тот же указатель, но всегда инициализированный (не может указывать на null). Но указатель – тип данных, а ссылка нет.

```
int i;
int &ai = i;
ai = 2; // i = 2;
// int &  - левая ссылка
// int && - правая ссылка

int i = 0;
const int c = 0;

int &lv1 = i;
const int &clv1 = c;
const int& clv2 = i + 1; // создается новая ячейка памяти

int &&rv1 = i;           // Error
int &&rv2 = i + 1;
++rv2;                   // можем изменять содержимое данной области

int &&rv3 = rv2;         // Error
int &lv2 = rv2;
int &&rv4 = i + 0;       // ссылка на копию i
int &&rv5 = (int) i;     // ссылка на копию i

// получить правую ссылку из любого выражения:

int &&rv6 = std::move(i);
```

Ссылки используются для передачи параметра в функцию. На низком уровне ссылки хоть ничем не отличается от указателя, но идет контроль.

```
void swap(double &d1, double &d2)
{
    double temp = d1;
   d1 = d2;
   d2 = temp;
}

swap(arr[i], arr[j]);
```

Начиная с C++11 ключевое слово *auto* при инициализации переменной может использоваться вместо типа переменной, чтобы сообщить компилятору, что он должен присвоить тип переменной исходя из инициализируемого значения. Это называется выводом типа (или «автоматическим определением типа данных компилятором»). Например:

```
auto x = 4.0; // 4.0 - это литерал типа double, поэтому и x должен быть типа double
auto y = 3 + 4; // выражение 3 + 4 обрабатывается как целочисленное, поэтому и
переменная y должна быть типа int
```

Это работает даже с возвращаемыми значениями функций:

```
int subtract(int a, int b) {
    return a - b;
}

int main()
{
    auto result = subtract(4, 3); // функция subtract() возвращает значение типа int и,
следовательно, переменная result также должна быть типа int
    return 0;
}
```

Переменные, объявленные без инициализации, не могут использовать эту особенность (поскольку нет инициализируемого значения, и компилятор не может знать, какой тип данных присвоить переменной).

В C++14 функционал ключевого слова *auto* был расширен до автоматического определения типа возвращаемого значения функции. Например:

```
auto subtract(int a, int b)
{
    return a - b;
}
```

Так как выражение $a - b$ является типа *int*, то компилятор делает вывод, что и функция должна быть типа *int*.

3. Классы и объекты в C++. Определение класса с помощью class, struct, union. Ограничение доступа к членам класса в C++. Члены класса и объекта. Методы класса и объекта. Константные члены класса. Схемы наследования.

Уровни доступа к членам класса:

- private - нет доступа извне
- protected - к ним имеют доступ потомки класса
- public - методы / функции

Эти методы контролируют целостность объекта.

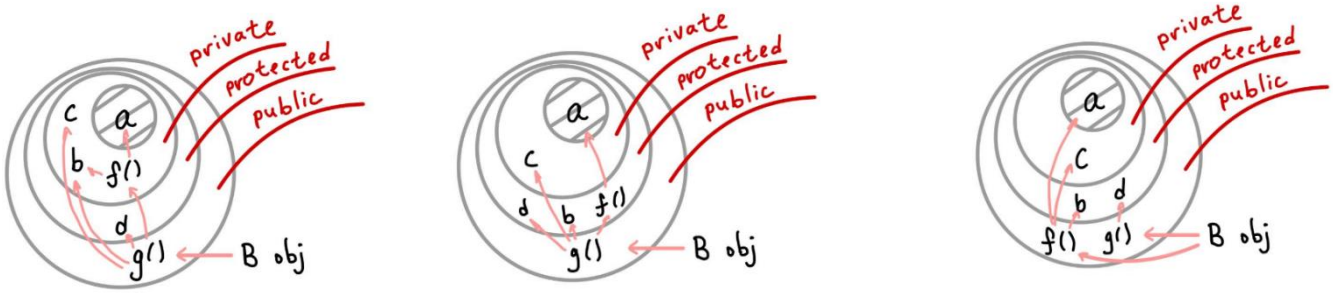
У *struct* по умолчанию уровень доступа для членов *public*, а у *class* – *private*. У *union* – *public*. В *union* нет уровня protected, так как *union* не может быть ни базовым классом, ни производным.

```
class <имя> [: <список баз>]
{
private:
    <члены>
protected:
    <члены>
public:
    <члены>
};
```

Схемы наследования.

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

class B: private A      // public / protected
{
private:
    int c;
protected:
    int d;
public:
    int g();            // все члены базового класса получат уровень доступа private
};
```

private *protected* *public*



Для того, чтобы восстановить уровень доступа можно написать:

```
public:
    using A::f;
```

Члены класса не относятся к конкретному объекту, они общие для всех объектов данного класса. Перед ними ставится ключевое слово *static*. Память под них не относится к объекту (при вызове *sizeof* они не будут учитываться).

Метод можно определить вне класса.

```
class A {
private:
   int a;
   const int cb;
   static int sc;
   static const int scd = 1;

public:
   int f();
};

int A::f() { return this->a; }
```

Методы могут быть константными, они не могут менять члены класса. Чтобы они могли менять член объекта, перед его определением необходимо написать ключевое слово *mutable*.

С членами класса можно работать, не создав ни одного объекта данного класса. Для этого необходимо создать методы класса.

```
public:
    static int g();
```

Статические методы не принимают указатель *this*. Для доступа к методам могут использоваться операторы: «.», «->», «::», «.*», «->*».

4. Создание и уничтожение объектов в C++. Конструкторы и деструкторы. Раздел инициализации конструкторов. Способы создания объектов. Явный и неявный вызов конструкторов. Приведение типа.

После создания объекта сразу должна пройти его инициализация, для этого мы используем специальные методы – конструкторы. Его вызов может быть как явным, так и неявным. По умолчанию создается конструктор без параметров, конструктор копирования, и конструктор переноса. Конструктор – метод, который не имеет типа возврата, его имя совпадает с именем класса:

```
<имя класса>::<имя конструктора>([параметры])[:<раздел инициализации>]
{
}
```

Пример:

```
class A
{
private:
   int a;
   const int cb = 2;
   static int sc;
   static const int scd = 1;

public:
    A(int i);
};
```

```
A::A(int i): a(i) //, cb(i), sc(i), scd(i)
{
    a = i;
    // cb = i;      // Error
    // sc = i;      // Не относится к классу в общем
    // scd = i;     // Error
}
```

Порядок, который мы написали в разделе инициализации не влияет на порядок создания объектов.

Для уничтожения объектов используются деструкторы. Он порождается по умолчанию. Как и оператор присваивания. Для создания/удаления объектов используются операторы языка *new*, *delete*. Они возвращают типизированный указатель (в отличие от *void* в C). Рассмотрим следующий класс:

```
class Array {
private:
    double* arr;
    int count;

public:
    Array() = default;
    Array(int cnt) : count(cnt) { arr = new double[count] {}; }
    Array(const Array& ar);
    Array(Array&& ar) noexcept;
    ~Array();

    bool equals(Array ar);

    static Array minus(const Array& ar);
};

Array::Array(const Array& ar) : count(ar.count) {
    arr = new double[count];

    for (int i = 0; i < count; ++i)
        arr[i] = ar.arr[i];
}

Array::Array(Array&& ar) noexcept : count(ar.count) {
    arr = ar.arr;
    ar.arr = nullptr;
}

Array::~Array() { delete[] arr; }

bool Array::equals(Array ar) {
    if (count != ar.count)
        return false;

    int i;
    for (i = 0; i < count && arr[i] == ar.arr[i]; ++i);

    return i == count;
}

Array Array::minus(const Array& ar)
{
    Array temp(ar);

    for (int i = 0; i < temp.count; ++i)
        temp.arr[i] *= -1;

    return temp;
}
```

Если перед именем конструктора поставить слово *explicit*, это будет гарантировать, что конструктор неявно не вызовется. Все конструкторы с 1 параметром должны быть с этим модификатором, это позволяет избежать случайного приведения типов.

Явно можно вызвать конструктор следующими способами:

```
<тип> <идентификатор> [(<параметры>)];            // A obj();
<тип> <идентификатор> {[параметры]};
<тип> *<идентификатор> = new <тип>{(параметры)};
<тип> <идентификатор> = <тип>{(параметры)};
```

Неявно:

```
<тип> <идентификатор> = <значение>;
<тип> <идентификатор> = {(параметры)};
```

Любой конструктор - оператор приведения типа.

5. Конструкторы копирования и переноса. Модификатор explicit. Удаление конструктора и default конструктор. Делегирующие и унаследованные конструкторы.

Обычный конструктор используется для той или иной инициализации объекта, он не должен вызываться для копирования уже существующего объекта, так как такой вызов изменит содержимое объекта (передаст объект в начальном состоянии) а мы хотим передать функции текущее состояние объекта, то есть нужен конструктор копирования.

Когда мы передаем в функцию какой-то объект по ссылке, конструктор не вызывается. Когда мы возвращаем объект из какой-либо функции (не по ссылке), если этот объект создается внутри этой функции, тоже вызывается конструктор копирования.

Конструктор копирования должен принимать в качестве параметра объект того же класса. Причем параметр лучше принимать по ссылке, потому что при передаче по значению компилятор будет создавать копию объекта. А для создания копия объекта будет вызываться конструктор копирования, что приведет бесконечной рекурсии.

```
class Person
{
    std::string name;
    unsigned age;
public:
    Person(std::string p_name, unsigned p_age)
    {
        name = p_name;
        age = p_age;
    }
    Person(const Person &p)
    {
        name = p.name;
        age = p.age + 1;     // для примера
    }
};
int main()
{
    Person tom{"Tom", 38};
    Person tomas{tom};     // создаем объект tomas на основе объекта tom
}
```

Конструктор переноса и оператор переноса был добавлен в C++ 11. Основная идея применения этих двух конструкций состоит в том, чтобы ускорить выполнение программы путем избегания копирования данных при начальной инициализации и присвоении так называемых *rvalue*-ссылок.

Конструктор переноса и оператор переноса целесообразно объявлять в классах, содержащих большие массивы данных.

Если в классе не реализован конструктор переноса, то его вызов заменяется конструктором копирования.

Общая форма объявления конструктора переноса в классе:

```
ClassName(ClassName&& rObj) noexcept
{
...
}
```

rObj – ссылка на ссылку на временный экземпляр класса, значение которого будет скопировано в текущий экземпляр.

В приведенной выше общей форме используется ключевое слово *noexcept*. Этот спецификатор указывает, что наш конструктор переноса не генерирует исключение или аварийно завершает свою работу. Компилятор рекомендует использовать слово noexcept для конструктора переноса. В конструкторе переноса не происходит никаких операций с памятью, а происходит простое присвоение указателя.

Если перед именем конструктора поставить слово *explicit*, это будет гарантировать, что конструктор неявно не вызовется. Все конструкторы с 1 параметром должны быть с этим модификатором, это позволяет избежать случайного приведения типов.

Ключевое слово *default* введено в C++ 11. Его использование указывает компилятору самостоятельно генерировать (использовать) соответствующую функцию класса, если таковая не объявлена в классе.

Общая форма использования ключевого слова default:

```
class ClassName
{
    // ...
    ClassName(parameters) = default;
    // ...
};
```

Ключевое слово *delete* используется в случаях, когда нужно запретить использование какого-либо конструктора, тогда его вызов приведет к ошибке.

Общая форма объявления конструктора класса с ключевым словом *delete*:

```
class ClassName
{
    // ...
    ClassName(parameters) = delete;
    // ...
};
```

6. Наследование в C++. Построение иерархии классов. Выделение общей части группы классов. Расщепление классов.

Наследование представляет один из ключевых аспектов объектно-ориентированного программирования, который позволяет наследовать функциональность одного класса (базового класса) в другом - производном классе.

В C ++ есть несколько типов наследования:

- публичный (public) — публичные (public) и защищенные (protected) данные наследуются без изменения уровня доступа к ним;
- защищенный (protected) — все унаследованные данные становятся защищенными;
- приватный (private) — все унаследованные данные становятся приватными.

В C ++ конструкторы и деструкторы не наследуются. Однако они вызываются, когда дочерний класс инициализирует свой объект. Конструкторы вызываются один за другим иерархически, начиная с базового класса и заканчивая последним производным классом. Деструкторы вызываются в обратном порядке.

Иерархия классов определяет взаимоотношения между ними. Т.е. при ее построении мы описываем само наследование, указывая есть ли расширение функционала, взаимосвязь не прямо наследуемых классов и т.п. В конечном итоге у нас получается общая структура всех классов.

Базовый класс выделяют в следующих случаях:

- Общая схема исполнения разных объектов.
- В объектах один и тот же набор методов.
- Имеются два разных класса с разными методами. Если у методов похожая реализация, то выделяем базовый класс.
- У нас имеется два разных класса. Если в дальнейшем они будут участвовать вместе, лучше сделать для них базовый класс уже на этом этапе разработки. Это нужно сделать, чтобы в дальнейшем нам было легко модифицировать программу.

Разбиваем класс в следующих случаях:

- Если один объект исполняет разные роли.
- Два множества методов используются в разной манере.
- Методы между собой никак не связаны.
- Одна сущность, но используется в разных частях программы.
- На классах возможно множественное наследование.

7. Множественное наследование. Прямая и косвенная базы. Виртуальное наследование. Понятие доминирования. Порядок создания и уничтожения объектов. Проблемы множественного наследования. Неоднозначности при множественном наследовании.

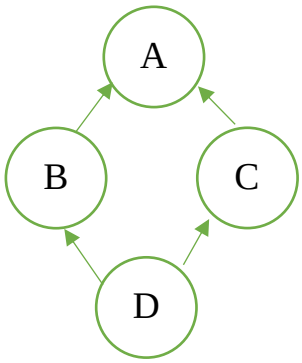
В C++ реализовано множественное наследование. Оно хорошо тем, что позволяет нам четко разделять наследуемые объекты, но на деле от него пытаются избавляться, в большинстве языков его уже нет.

Прямая база – база, непосредственно от которой мы породили нашу сущность.

Косвенная база – база, которая входит в нашу сущность, но от нее мы не порождались. Или же прямая база от прямой базы.

Виртуальное наследование.

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



Если убрать первый *virtual*, то класс A будет рассматриваться как подобъект класса D. И подобъект будет создаваться для класса D, потом будет создан подобъект класса B, а для подобъекта класса C создастся подобъект класса A, и потом создастся объект класса C и D.

Если мы не напишем второй *virtual*, то подобъект класса A не будет создаваться для класса D.

Если мы напишем два *virtual*, то подобъект класса A создастся только для класса D.

Конструкторы виртуальных баз вызываются в первую очередь.

Методы, определяемые в производных классах, доминируют над методами базовых классов. То есть они их подменяют. Для решения этой проблемы можно использовать *using*.

Проблемы множественного наследования:

- При ромбовидной иерархии как выше, что-то может отрабатывать два раза.

```
class A
{
public:
    void f() { cout << "Executing f from A;" << endl; }
};

class B : virtual public A
{
public:
    void f()
```

```
    {
        A::f();
        cout << "Executing f from B;" << endl;
    }
};

class C : virtual public A
{
public:
    void f()
    {
        A::f();
        cout << "Executing f from C;" << endl;
    }
};

class D : virtual public C, virtual public B
{
public:
    void f()
    {
        C::f();
        B::f();
        cout << "Executing f from D;" << endl;
    }
};

int main()
{
    D obj;

    obj.f();
}
```

Ее решение:

```
class A
{
protected:
    void _f() { cout << "Executing f from A;" << endl; }
public:
    void f() { this->_f(); }
};

class B : virtual public A
{
protected:
    void _f() { cout << "Executing f from B;" << endl; }
public:
    void f()
    {
        A::_f();
        this->_f();
    }
};

class C : virtual public A
{
protected:
    void _f() { cout << "Executing f from C;" << endl; }
public:
    void f()
    {
        A::_f();
        this->_f();
    }
```

```
};

class D : virtual public C, virtual public B
{
protected:
    void _f() { cout << "Executing f from D;" << endl; }
public:
    void f()
    {
        A::_f(); C::_f(); B::_f();
        this->_f();
    }
};

int main()
{
    D obj;

    obj.f();
}
```

- Неоднозначность

```
class A
{
public:
    int a;
    int (*b)();
    int f();
    int f(int);
    int g();
};

class B
{
    int a;
    int b;
public:
    int f();
    int g;
    int h();
    int h(int);
};

class C: public A, public B {};

class D
{
public:
    static void fun(C& obj)
    {
        // obj.a = 1;       // Error!!!
        // obj.b();            // Error!!!
        // obj.f();            // Error!!!
        // obj.f(1);        // Error!!!
        // obj.g = 1;       // Error!!!
        obj.h(); obj.h(1);  // Ok!
    }
};

int main()
{
    C obj;
    D::fun(obj);
}
```

- Подмена методов

```
class A
{
public:
    void f1() { cout << "Executing f1 from A;" << endl; }
    void f2() { cout << "Executing f2 from A;" << endl; }
};

class B
{
public:
    void f1() { cout << "Executing f1 from B;" << endl; }
    void f3() { cout << "Executing f3 from B;" << endl; }
};

class C : private A, public B {};

class D
{
public:
    void g1(A& obj)
    {
        obj.f1(); obj.f2();
    }
    void g2(B& obj)
    {
        obj.f1(); obj.f3();
    }
};


int main()
{
    C obj;
    D d;

    // obj.f1();  Error! Множественное определение
    // d.g1(obj); Error! Нет приведения к баз. классу при наследовании по схеме private
    d.g2(obj);
}
```

Исправить можно так:

```
class A
{
public:
    void f1() { cout << "Executing f1 from A;" << endl; }
    void f2() { cout << "Executing f2 from A;" << endl; }
};

class B
{
public:
    void f1() { cout << "Executing f1 from B;" << endl; }
    void f3() { cout << "Executing f3 from B;" << endl; }
};

class C : public A, public B {};

class D
{
public:
    void g1(A& obj)
    {
        obj.f1(); obj.f2();
```

```
    }
    void g2(B& obj)
    {
        obj.f1(); obj.f3();
    }
};


int main()
{
    C obj;
    D d;

    d.g1(obj);
    d.g2(obj);
}
```

8. Полиморфизм в C++. Виртуальные методы. Чисто виртуальные методы. Виртуальные и чисто виртуальные деструкторы. Понятие абстрактного класса. Ошибки, возникающие при работе с указателем или ссылкой на базовый класс. Дружественные связи.

Полиморфизм – возможность подменять одно другим, не изменяя написанный код. Возможность обработки разных типов данных, с помощью "одной и той же" функции, или метода.

Если в классе определен хотя бы один метод с модификатором *virtual*, то данный класс считается полиморфом и при создании объектов этих классов в них добавляется указатель на виртуальную таблицу, то есть на те методы, которые использует класс.

Ключевое слово *override* дает гарантию того, что метод является полиморфом и подменяет метод базового класса.

```
class A
{
public:
    virtual void f() { cout << "Executing f from A;" << endl; }
};

class B : public A
{
public:
    void f() override { cout << "Executing f from B;" << endl; }
};

class C
{
public:
    static void g(A& obj) { obj.f(); }
};

int main()
{
    B obj;
    C::g(obj);
}
```

Базовый класс всегда должен быть абстрактным. Его объекты создавать нельзя. Для того, чтобы создать абстрактный класс, в нем должен быть хотя бы один чисто виртуальный метод. Он задается следующим образом:

```
public:
    virtual void f() = 0;
```

Таким образом, производные классы должны подменить этот метод. Если они не реализуют этот метод, они тоже будут абстрактными.

Базовый класс всегда должен содержать виртуальный деструктор. Чисто виртуальный деструктор:

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

A::~A() = default;

class B : public A
{
public:
    ~B() override { cout << "Class B destructor called;" << endl; }
};

int main()
{
    A* pobj = new B;
    delete pobj;
}
```

Ошибки, возникающие при работе с указателями на базовый класс:

- Подобъект может находится по другому адресу самого объекта внутри объекта. Если это произошло, то следующий код может выдавать непредсказуемый ответ:

```
A *pa = 0;
B *pb = 0;
if (pa == pb)
```

- Может случайно вызваться конструктор копирования:

```
void f(A obj);

B obj;
A &alias = obj;
f(alias);
```

Поэтому для полиморфного класса необходимо запретить/определить копирование с explicit.

- Нельзя работать с массивами объектов, вместо этого необходимо создавать структуры с указателями на объекты.

```
A &index(A *arr, int i)
{
    return arr[i];
}

B *ar - new B[10];
A obj= index(B, 2);
```

Дружественные связи. Мы можем дать объектам одного классам доступ ко всем членам объектов другого класса. Сделать это позволяет модификатор friend, при этом доступ является односторонним. В современных языках от этого избавились, т.к. это нарушает целостность объекта. Пример с наследованием:

```
class C; // forward объявление

class A
{
private:
    void f1() { cout << "Executing f1;" << endl; }

    friend C;
};

class B : public A
{
private:
    void f2() { cout << "Executing f2;" << endl; }
};

class C
{
public:
    static void g1(A& obj) { obj.f1(); }
    static void g2(B& obj)
    {
        obj.f1();
//    obj.f2(); // Error!!! Имеет доступ только к членам A
    }
};

class D : public C
{
public:
// static void g2(A& obj) ( obj.f1(); } // Error!!! Дружба не наследуется
};



int main()
{
    A aobj;

    C::g1(aobj);

    B bobj;

    C::g1(bobj);
    C::g2(bobj);
}
```

Дружба и виртуальные методы:

```
class C; // forward объявление

class A
{
protected:
    virtual ~A() = default;
    virtual void f() { cout << "Executing f from A;" << endl; }

    friend C;
};
```

```
class B : public A
{
protected:
    void f() override { cout << "Executing f from B;" << endl; }
};

class C
{
public:
    static void g(A& obj) { obj.f(); }
};

int main()
{
    B bobj;

    C::g(bobj);      // Executing f from B;
}
```

9. Обработка исключительных ситуаций в C++. Решение проблем структурного программирования. Блоки try и catch. Блоки try и catch методов и конструкторов. Безопасный код относительно исключений. Обертывание исключения в exception_ptr. Задачи, которые может решать исключение. Проблемы с динамической памятью при обработке исключительных ситуаций.

Обработка исключительных ситуаций выглядит следующим образом:

```
try
{
    ->throw <объект>;
}
catch(<тип> &<объект>) {}
```

- Инструкция try - заворачиваем в неё блок кода, в котором может произойти ошибка, и берём его под контроль.
- Если в блоке try возникает исключительная ситуация, мы можем перейти на обработчик catch. Обработчики идут непосредственно после блока try.
- Генерируем исключительную ситуацию, используя инструкцию throw.

Здесь нет приведения типов, т.е. жесткая типизация, за исключением перевода ссылки с базового класса на производный, и указателя с базового на производный. Обработчиков может быть несколько. Если на этом уровне ни один из *catch* не перехватил этот объект, то это передается на более высокий уровень. Ошибка может никем не перехватиться, в этом случае программа "падает".

Мы можем перехватить любые исключительные ситуации, используя catch с 3 точками. Этот обработчик должен быть в конце списка, иначе он перехватит любую ситуацию, и другие обработчики ниже не будут работать.

При пробросе все статические переменные уничтожаются, однако динамические не будут освобождены.

Недостатки обработки ошибок в структурном программировании:

- Если где-то возникает ошибка в коде, мы вынуждены "протащить" ее через все уровни абстракции/иерархии до того места, пока мы не сможем обработать эту ошибку.
- Весь код насыщен непрерывными проверками. Обработка ошибки совмещена вместе с кодом.

Теперь, при ОО подходе мы получили из недостатков структурного следующие плюсы:

- Мы не "протаскиваем ошибку"
- Вся обработка сводится в одно место

Мы можем обрабатывать исключительные ситуации в методах и конструкторах класса. Исключения могут быть пойманы и вне.

Безопасный относительно исключений код должен представлять одну из трех гарантий:

1. Функции, предоставляющие базовую гарантию, обещают, что если исключение будет возбуждено, то все в программе остается в корректном состоянии. Никакие объекты или структуры данных не повреждены, и все объекты находятся в непротиворечивом состоянии. Однако точное состояние программы может быть непредсказуемо.
2. Функции, предоставляющие строгую гарантию, обещают, что если исключение будет возбуждено, то состояние программы не изменится. Вызов такой функции является атомарным; если он завершился успешно, то все запланированные действия выполнены до конца, если же нет, то программа останется в таком состоянии, как будто функция никогда не вызывалась.
   Работать с функциями, представляющими такую гарантию, проще, чем с функциями, которые дают только базовую гарантию, потому что после их вызова может быть только два состояния программы: то, которое ожидается в результате ее успешного завершения, и то, которое было до ее вызова. Напротив, если исключение возникает в функции, представляющей только базовую гарантию, то программа может оказаться в любом корректном состоянии.
3. Функции, предоставляющие гарантию отсутствия исключений, обещают никогда не возбуждать исключений, потому что всегда делают то, что должны делать. Все операции над встроенными типами (например, целыми, указателями и т. п.) обеспечивают такую гарантию. Это основной строительный блок безопасного относительно исключений кода. Разумно предположить, что функции с пустой спецификацией исключений не возбуждают их, но это не всегда так.

std::exception_ptr:

Этот тип позволяет в себе хранить исключение абсолютно любого типа. Его поведение сходно с *std::shared_ptr*: его можно копировать, передавать в качестве параметра, при этом само исключение не копируется. Основное предназначение *exception_ptr* — это передача исключений в качестве параметров функции, возможна передача исключений между потоками. Таким образом, объекты данного типа позволяют сделать обработку ошибок более гибкой.

Проблемы с динамической памятью.

В C++, перейдя на обработчик, мы не можем вернуться в место возникновения ошибки (все временные объекты будут уничтожены). Предположим, у класса *A* есть метод *f()*. Если мы динамически выделили память:

```
try {
A* obj = new A; // Динамически выделили память под объект
obj->f();       // Если внутри функции f() произошла ошибка и мы вышли на обработчик,
delete obj;     // происходит утечка памяти
}
```

Если при вызове метода *f()* возникает исключительная ситуация и мы выходим на какой-то из обработчиков, объект *obj* не удаляется. Происходит утечка памяти.

Два варианта решения проблемы:

```
void A::f() noexcept // Первый способ
void A::f() throw()  // Второй способ
```

Модификатор *noexcept* «дает обещание» компилятору не обрабатывать исключение, тогда вызывается специальный метод *terminate,* задача которого очистить стек. Метод *terminate()* приводит к тому, что будут вызываться все деструкторы только временных объектов в порядке, обратном их созданию.

Со *throw* результат непредсказуем, это старый синтаксис, его лучше не использовать.

- Если пишем *noexcept* без параметров аналогичен *noexcept(True)* - это говорит о том, что данный метод не должен обрабатывать исключительную ситуацию.
- Если пишем *noexcept(False)* или *throw(...)*, то этот метод может обрабатывать все исключительные ситуации, как и в случае если ничего не пишем.

Любой деструктор по умолчанию не бросает исключения, т.к. может выйти бесконечный вызов деструктора. В блоке *catch* можно писать *throw*, тогда исключения необходимо ловить в «старшем» блоке.

std::exception содержит виртуальный метод *what*, и на основе него мы можем создавать свои классы. От него есть производный класс ошибок *std::bad_alloc*. При создании своих производный классов мы придерживаемся такого подхода, что отдельный класс обрабатывает только одну ситуацию. Даже стандартные ошибки мы будет перекладывать на себя, создавая свои.

Исключения решают следующие задачи:

- Исключают необходимость прокидывания ошибок через много уровней, все делает обработчик
- Разделяют логику программы от обработки ошибок, выносы обработчики отдельно
- Позволяют легко модифицировать и развивать ПО
- Буквально контролируют работу программы: создание объектов и выполнение действий над ними

Пример обработки исключительных ситуаций:

```
class ExceptionArray : public std::exception
{
protected:
    static const size_t sizebuff = 128;
    char errormsg[sizebuff]{};

public:
    ExceptionArray() noexcept = default;
    ExceptionArray(const char* msg) noexcept
    {
        strcpy_s(errormsg, sizebuff, msg);
    }
    ~ExceptionArray() override {}

    const char* what() const noexcept override { return errormsg; }
};

class ErrorIndex : public ExceptionArray
{
private:
    const char* errIndexMsg = "Error Index";
    int ind;

public:
    ErrorIndex(const char* msg, int index) noexcept : ind(index)
    {
        sprintf_s(errormsg, sizebuff, "%s %s: %4d!", msg, errIndexMsg, ind);
    }
    ~ErrorIndex() override {}

    const char* what() const noexcept override { return errormsg; }
};

int main()
{
    try
    {
        throw(ErrorIndex("Index!!", -1));
    }
    catch (const ExceptionArray& error)
    {
        cout << error.what() << endl;
    }
    catch (std::exception& error)
    {
        cout << error.what() << endl;
    }
    catch (...)
    {
        cout << "All errors!" << endl;
    }
}
```

10. Перегрузка операторов в C++. Операторы .*, ->*. Правила перегрузки операторов. Перегрузка операторов =, () и [ ]. Перегрузка операторов ->, * и –>*.

Перегрузка операторов – задание новой операции на основе существующего оператора. Это позволяет не заботится об используемом литерале, приоритете, арности и т.д. Есть операторы, которые нельзя перегружать, к ним относятся «.», «.*», «::», «?:» (тернарный), «sizeof», «typeid».

Посмотрим на очень интересный пример с функциями:

```
void f();      // Определили функцию f()
void (*pf)(); // Определили указатель на функцию
pf = f;          // Этот указатель инициализируем адресом функции(так как имя любой
функции - это ее адрес в памяти)
pf();          // Через указатель на функцию вызываем функцию
```

Оператор () - оператор разыменования - вызов функции по адресу. Так же вызвать функцию f() можно и таким образом: (*pf)().

Что касается методов класса:

```
void A::f();      // Метод класса A
void (A::*pf)(); // Указатель на метод класса A

// Хотелось бы проинициализировать этот указатель.
// pf = A::f; - Если мы таким образом напишем, мы получим не адрес этого метода
// Метод не находится в классе. Он вызывается по указателю, и чтобы получить этот
адрес,
// было принято решение добавить вот такой синтаксис
pf = &A::f;  // Вычисление адреса метода.

A obj;
(obj.*pf)(); // Чтобы вызвать метод через указатель, используется оператор .*
// Этот указатель имеет более низкий приоритет, чем (),
// поэтому, чтобы использовать (),
// надо повысить его приоритет, взяв obj.*pf в круглые скобки.

A* p = &obj;
(p->*pf)();  // Оператор ->* используется для указателя на объект
// В метод, на который указывает этот указатель, будет передаваться
// указатель на объект.
```

Таким образом, мы разделяем вызов функции и вызов метода. Если мы вызываем метод класса через указатель для объекта, используется оператор .*, а если работаем с указателем на объект, используется оператор ->.

Правила перегрузки операторов:

1. Операторы, которые можно перегрузить только как члены классов:
   - Оператор = - оператор присваивания (бинарный)
   - Оператор () - функтуатор (бинарный)
   - Оператор [] - индексация (бинарный)
   - Оператор -> - унарный
   - Оператор ->* - бинарный, так как принимает указатель на метод и объект, метод которого вызываем
2. Бинарные операторы можно перегружать как члены класса или как внешние функции-операторы. Это зависит от ситуации. Конечно, надо отдавать предпочтение члену класса. Если мы перегружаем бинарный оператор, как член класса, он принимает 1 параметр (второй параметр он принимает неявно - *this).
3. Унарные операторы перегружаем как члены класса.
4. Нельзя перегружать операторы «.», «.*», «::», «?:» (тернарный), «sizeof», «typeid»

```
class Array
{
    ...
public:
    bool operator==(const Array& arr) const;
    ...
};

// Первый вариант - перегрузка оператора, как члена класса
bool Array::operator==(const Array& arr) const
{...}

// Второй вариант - перегрузка оператора, как внешней функции
bool Array::operator!=(const Array& arr1, const Array& arr2) const
{...}
```

Перегрузка оператора ():

Этот оператор можно реализовать только как член класса. Он может иметь любое число параметров любого типа, тип возвращаемого значения также произвольный. Классы, с перегруженным оператором (), называются функциональными, их экземпляры называются функциональными объектами или функторами. Именно с помощью таких классов и объектов в C++ реализуется парадигма функционального программирования. Функциональные классы и объекты, используемые в стандартной библиотеке, в зависимости от назначения имеют свои названия: предикаты, компараторы, хеш-функции, аккумуляторы, удалители. В зависимости от контекста использования, стандартная библиотека предъявляет определенные требования к функциональным классам. Экземпляры этих классов должны быть копируемыми по значению, не модифицировать аргументы, не иметь побочных эффектов и изменяемое состояние (чистые функции), соответственно реализация перегрузки оператора () обычно является константным членом класса. Есть исключение — алгоритм std::for_each(), для него функциональный объект может модифицировать аргумент и иметь изменяемое состояние.

Перегрузка оператора []:

Этот бинарный оператор, который обычно называют индексатором, может быть реализован только, как функция-член, которая должна иметь ровно один параметр. Тип этого параметра произвольный, соответственно, перегрузок может быть несколько, для разных типов параметра. Индексатор обычно перегружается для «массивоподобных» типов, а также для других контейнеров, например ассоциативных массивов. Возвращаемое значение обычно является ссылкой на элемент контейнера. Также может быть возврат по значению, но следует иметь в виду, что при этом для получения адреса элемента нельзя будет использовать выражения &x[i], допустимые для встроенного индексатора. Такое выражение не будет компилироваться, если возвращаемый тип встроенный, и будет давать адрес временного объекта для пользовательского возвращаемого типа.

Индексатор часто перегружают в двух вариантах — константном и неконстантном.

```
T& operator[](int ind);
const T& operator[](int ind) const;
```

Первая версия позволяет модифицировать элемент, вторая только прочитать и она будет выбрана для константных экземпляров и в константных функциях-членах.

Перегрузка оператора =:

Если мы динамически выделяем память под члены класса, мы обязаны явно определить оператор присваивания или запретить. Дело в том, что для любого типа неявно определяется оператор присваивания, который побайтно копирует данные. Может выйти так, что два объекта указывают на одну область памяти. В большинстве случаев это не нужно.

Разница оператора присваивания с копированием - создает копию объекта, а оператор присваивания с переносом - захватывает временный объект.

Копирование - выделяем новую память и копируем из одной области в другую, а при переносе захватываем область того объекта, который получаем. Параметр обнуляем, чтобы при деструкторе не произошло удаления области памяти, которой мы захватили.

Перегрузка операторов -> и *:

Оператор -> перегружается как член класса, он унарный, принимающий один параметр и должен возвращать либо указатель, либо ссылку на объект. Совместно с указателем -> еще перегружается оператор *. Он помогает делать примерно то же самое - возвращает ссылку на объект, и по ссылке мы уже вызываем метод. Мы можем перегружать эти операторы как для константных, так и для не константных объектов.

```
class A
{
public:
    void f() const { cout << "Executing f from A;" << endl; }
};

class B
{
private:
    A* pobj;

public:
    B(A* p) : pobj(p) {}

    A* operator->() noexcept { return pobj; }
    const A* operator->() const noexcept { return pobj; }
    A& operator*() noexcept { return *pobj; }
    const A& operator*() const noexcept { return *pobj; }
};

void main()
{
    A a;

    B b1(&a);
    b1->f();

    const B b2(&a);
    (*b2).f();
}
```

Перегрузка оператора ->*:

Оператор ->* перегружается как член класса и является бинарным (*this и указатель на метод)

Класс Pointer по существу скрывает связь объекта который выбирает связку и указателя на метод. Создаем объект и вызывая перегруженный оператор, он возвращает нам объект, который отвечает за связь и этот объект имеет перегруженный оператор круглые скобочки. Он уже вызывает указатель.

```
class Callee
{
private:
    int index;

public:
    Callee(int i = 0) : index(i) {}
    int inc(int d) { return index += d; }
};

class Caller
{
public:
    using FnPtr = int (Callee::*)(int);

private:
    Callee* pobj;
    FnPtr ptr;

public:
    Caller(Callee* p, FnPtr pf) : pobj(p), ptr(pf) {}
    int operator ()(int d) { return (pobj->*ptr)(d); }
};

class Pointer
{
private:
    Callee* pce;

public:
    Pointer(int i) { pce = new Callee(i); }
    ~Pointer() { delete pce; }

    Caller operator->*(Caller::FnPtr pf) { return Caller(pce, pf); }
};

int main()
{
    Caller::FnPtr pn = &Callee::inc;

    Pointer pt(1);

    cout << "Result: " << (pt->*pn)(2) << endl;
}
```

11. Перегрузка унарных и бинарных операторов. Проблемы с перегрузкой операторов &&, ||, „ &. Перегрузка операторов ++, --. Перегрузка операторов приведения типов. Тривалентный оператор spaceship.

Перегрузка операторов – задание новой операции на основе существующего оператора. Это позволяет не заботится об используемом литерале, приоритете, арности и т.д. Есть операторы, которые нельзя перегружать, к ним относятся «.», «.*», «::», «?:» (тернарный), «sizeof», «typeid».

Унарные операторы перегружаем как члены класса.

Пример перегрузки бинарных операторов:

```
class Complex
{
private:
   double re, im;

public:
   Complex(double r = 0., double i = 0.) : re(r), im(i) {}

   Complex operator-() const { return Complex(-re, -im); }
   Complex operator-(const Complex& c) const { return Complex(re + c.re, im + c.im); }
   friend Complex operator+(const Complex& c1, const Complex& c2);

   friend ostream& operator<<(ostream& os, const Complex& c);
};

Complex operator+(const Complex& c1, const Complex& c2) {
   return Complex(c1.re + c2.re, c1.im + c2.im);
}

ostream& operator<<(ostream& os, const Complex& c) {
   return os << c.re << " + " << c.im << "i";
}

int main() {
   Complex c1(1., 1.), c2(1., 2.), c3(2., 1.);

   Complex c4 = c1 + c2;
   cout << c4 << endl;

   Complex c5 = 5 + c3;
   cout << c5 << endl;

   // Complex c6 = 6 - c3; Error!!!

   Complex c7 = -c1;
   cout << c7 << endl;
}
```

Не рекомендуется перегружать следующие три бинарных оператора: , (запятая), &&, ||. Дело в том, что для них стандарт предусматривает порядок вычисления операндов (слева направо), а для последних двух еще и так называемую семантику быстрых вычислений (short-circuit evaluation), но для перегруженных операторов это уже не гарантируется или просто бессмысленно, что может оказаться весьма неприятной неожиданностью для программиста. (Семантика быстрых вычислений заключается в том, для оператора && второй операнд не вычисляется, если первый равен false, а для оператора || второй операнд не вычисляется, если первый равен true.)

Также не рекомендуется перегружать унарный оператор & (взятие адреса). Тип с перегруженным оператором & опасно использовать с шаблонами, так как они могут использовать стандартную семантику этого оператора. Правда в C++11 появилась стандартная функция (точнее шаблон функции) std::addressof(), которая умеет получать адрес без оператора & и правильно написанные шаблоны должны использовать именно эту функцию вместо встроенного оператора.

Перегрузка унарных операторов, в том числе ++ и --:

```
class Integer
{
private:
    int value;
public:
    Integer(int i): value(i) {}
    //унарный +
    friend const Integer& operator+(const Integer& i);
    //унарный -
    friend const Integer operator-(const Integer& i);
    //префиксный инкремент
    friend const Integer& operator++(Integer& i);
    //постфиксный инкремент
    friend const Integer operator++(Integer& i, int);
    //префиксный декремент
    friend const Integer& operator--(Integer& i);
    //постфиксный декремент
    friend const Integer operator--(Integer& i, int);
};

//унарный плюс ничего не делает.
const Integer& operator+(const Integer& i) {
    return i.value;
}

const Integer operator-(const Integer& i) {
    return Integer(-i.value);
}

//префиксная версия возвращает значение после инкремента
const Integer& operator++(Integer& i) {
    i.value++;
    return i;
}

//постфиксная версия возвращает значение до инкремента
const Integer operator++(Integer& i, int) {
    Integer oldValue(i.value);
    i.value++;
    return oldValue;
}

//префиксная версия возвращает значение после декремента
const Integer& operator--(Integer& i) {
    i.value--;
    return i;
}

//постфиксная версия возвращает значение до декремента
const Integer operator--(Integer& i, int) {
    Integer oldValue(i.value);
    i.value--;
    return oldValue;
}
```

Можно реализовать оператор приведения типа с автоматическим выведением типа:

```
class A
{
private:
    int val;

public:
    A(int i) : val(i) {}

    operator auto() const& { return val; }
    operator auto()&& { return val; }
    operator auto* () const { return &val; }
};

int main()
{
    A obj{ 10 };

    int v1 = obj;               // operator auto() const&
    double v2 = obj;            // operator auto() const&
    const double& al = obj;     // operator auto() const&
    int v3 = std::move(obj);    // operator auto()&&
    const int* p = obj;         // operator auto*() const
}
```

Оператор space ship <=>:

Используется для сравнения двух объектов:

```
(a <=> b) < 0  //true if a < b
(a <=> b) > 0  //true if a > b
(a <=> b) == 0 //true if a is equal/equivalent to b
```

Данный оператор был добавлен в C++20 и может быть определен компилятором с помощью *default*, что сильно упрощает задачу во многих случаях, когда для класса необходимо написать операторы сравнения. Если не использовать space ship, придется отдельно перегружать <, >, <=, >=, ==. Рассмотрим такой пример:

```
# define _CRT_SECURE_NO_WARNINGS
#include <iostream>
#include <compare>
#include <string.h>

using namespace std;

class MyInt {
public:
    constexpr MyInt(int val) : value{ val } { }
    auto operator<=>(const MyInt&) const = default;

private:
    int value;
};

class MyDouble {
public:
    constexpr MyDouble(double val) : value{ val } { }
    auto operator<=>(const MyDouble&) const = default;

private:
    double value;
};
```

```
class MyString {
public:
    constexpr MyString(const char* val) : value{ val } { }
    auto operator<=>(const MyString&) const = default;
private:
    const char* value;
};

int main() {
    MyInt i1{ 1 }, i2{ 2 };
    cout << (i1 < i2) << endl;

    MyDouble d1{ -0. }, d2{ 0. };
    cout << (d1 != d2) << (1. < d2) << (d1 < 2.) << endl;

    char st[5];
    strcpy(st, "Ok!!");
    MyString s1{ "Ok!" }, s2{ st };
    cout <<  (s1 < s2) << ("Ok!!" == s2) << endl; // сравнение адресов
}
```

Еще варианты перегрузки:

```
class MyInt {
private:
    int value;

public:
    MyInt(int val = 0) : value(val) {}

    //strong_ordering operator <=>(const MyInt& rhs) const
    //{
    //    return value <=> rhs.value;
    //}

    //strong_ordering operator <=>(const MyInt& rhs) const
    //{
    //    return value == rhs.value ? strong_ordering::equal :
    //           value < rhs.value ? strong_ordering::less :
    //                               strong_ordering::greater;
    //}

    //weak_ordering operator <=>(const MyInt& rhs) const
    //{
    //    return value == rhs.value ? weak_ordering::equivalent :
    //           value < rhs.value ? weak_ordering::less :
    //                               weak_ordering::greater;
    //}

    partial_ordering operator <=>(const MyInt& rhs) const {
        return value == rhs.value ? partial_ordering::equivalent :
               value < rhs.value ? partial_ordering::less :
               value > rhs.value ? partial_ordering::greater :
               partial_ordering::unordered;
    }

    bool operator ==(const MyInt&) const = default;
};

int main() {
    MyInt a{ 1 }, b{ 2 }, c{ 3 }, d{ 1 };
    cout << "a < b: " << (a < b) << ", c > b: " << (c >= b) << endl;
    cout << "a < b: " << (a < b) << ", c > b: " << (c > b) << ", a != b: " << (a != b)
<< endl;
    cout << "a < 5: " << (a < 5) << ", 1 < c: " << (1 < c) << endl;
}
```

12. Шаблоны функций, методов классов и классов в C++. Недостатки шаблонов. Параметры шаблонов. Параметры типы и параметры значения. Шаблоны функций и методов классов. Подстановка параметров в шаблон. Выведение типов параметров шаблона. Явное указание значений типов параметров шаблона при вызове функции. Срезание ссылок и модификатора const.

В языке Си приходилось создавать подобные функции, работающие с разными типами данных. По существу, был copy-past кода под разные типы данных. Можно решать эту проблему с помощью макросов с параметрами, но это крайне опасная вещь. На этапе препроцесирования происходит подстановка макроса в код, на этапе компиляции идет проверка подставленного. Если у нас ошибка в макросе, то определить её крайне сложно.

```
#define print(a) printf("%d \n", a)
#define min(a,b) (a < b ? a : b)
```

В языке C++ эта проблема подобного копирования кода решается с помощью шаблонов. Мы можем определить шаблон. Это может быть функция, класс, метод класса и также мы можем определить шаблон имени типа. Во время компиляции будет подстановка значения параметров шаблона с проверкой шаблона.

Шаблон не является ни классом, ни функцией. Функция или класс генерируется на основе шаблона во время использования. Компилятор встречает вызов функции, смотрит, может ли использоваться шаблон, и, если может, создаёт по шаблону функцию или класс.

Параметрами шаблона могут быть:

- Типы. Параметрами типа могут быть простые типы языка си, производные типы языка си и классы.
- Параметры значений. Параметры значения - только константные параметры целого типа или указатели с внешним связыванием.

Шаблоны можно определять:

- функций
- типов
- классов
- методов класса

Синтаксис шаблона в общем виде:

```
template<[параметры шаблона]>
функция | класс | тип
```

Программа, использующая шаблоны, содержит код для каждого порожденного типа, что может увеличить размер исполняемого файла. Кроме того, с одними типами данных шаблоны могут работать не так эффективно, как с другими. В этом случае имеет смысл использовать специализацию шаблона.

Возможно два варианта создания функции по шаблону:

- Функция принимает параметры шаблона, например, void freeArray(Type* arr);
- Явное указание параметров функции, например, Type* initArray(int count);

Пример шаблона функции и вызова функции:

```
template<typename Type>
void swap(Type&, Type&);
...
swap<double>(ar[i], ar[j]);
```

В обычном классе можно определить шаблонный метод.

Правило вызова. Компилятор рассматривает, может ли он вызвать перегруженную функцию. Если он находит перегруженную функцию, он ее вызывает. Если он не нашел ни одной перегруженной функции, он рассматривает специализации и подбирает подходящую. Если не подошла ни одна специализация, компилятор использует шаблон. То есть, шаблонная функция используется только в том случае, если для вызова функции компилятор не смог подобрать подходящую перегруженную функцию или функцию со специализацией.

Выведение типов параметров шаблона — это процесс, при котором компилятор определяет типы аргументов, передаваемых в параметры шаблона, на основе типов аргументов, передаваемых при вызове шаблона.

Когда используется шаблон функции или шаблон класса и передаются аргументы, компилятор анализирует их и определяет типы параметров шаблона, если они не были явно указаны. Это позволяет использовать шаблоны с разными типами данных без явного указания типов.

Можно явно указывать значения типов параметров шаблона:

```
template <typename T>
void print(T value) {
    std::cout << value << std::endl;
}

print<int>(10);     // Явное указание типа параметра шаблона как int
print<double>(3.14);  // Явное указание типа параметра шаблона как double
print<const char*>("Hello");  // Явное указание типа параметра шаблона как const char*
```

Срезание ссылок — это механизм, который определяет, как типы ссылок комбинируются при объявлении или использовании шаблонов с параметрами ссылочного типа.

- T& & становится T&
- T& && становится T&
- T&& & становится T&
- T&& && становится T&&

Выведение типов, срезание ссылок и const:

```
# define V_1

# ifdef V_1
template <typename T>
T f(T v) { return v; }

# elif defined(V_2)
template <typename T>
T f(T& v) { return v; }

# elif defined(V_3)
template <typename T>
T f(const T& v) { return v; }

# elif defined(V_4)
template <typename T>
T f(T&& v) { return v; }

# elif defined(V_5)
template <typename T>
T& f(T&& v) { return v; }

# elif defined(V_6)
template <typename T>
T&& f(T&& v) { return std::forward<T>(v); }

# elif defined(V_7)
auto f(auto v) { return v; }

# elif defined(V_8)
auto f(auto& v) { return v; }

# elif defined(V_9)
auto f(const auto& v) { return v; }

# elif defined(V_10)
auto&& f(auto&& v) { return v; }

# elif defined(V_11)
auto&& f(auto&& v) { return std::forward<decltype(v)>(v); }

# elif defined(V_12)
decltype(auto) f(auto&& v) { return std::forward<decltype(v)>(v); }

# elif defined(V_13)
template <typename T>
auto f(T&& v) -> decltype(v) { return std::forward<T>(v); }

# endif


int main()
{
    int i;
    int& a = i;
    const int& b = 0;

    decltype(auto) r1 = f(i);
    // 1. T f(T v) --->   int f<int>(int)
    // 2. T f(T& v)    --->   int f<int>(int&)
    // 3. T f(const T& v)  --->   int f<int>(const int&)
    // 4. T f(T&& v)   --->   int& f<int&>(int&)
    // 5. T& f(T&& v)  --->   int& f<int&>(int&)
    // 6. T&& f(T&& v) { return std::forward<T>(v); }  --->   int& f<int&>(int&)
    // 7. auto f(auto v)   --->   int f<int>(int)
```

```
    // 8. auto f(auto& v)  --->   int f<int>(int&)
    // 9. auto f(const auto& v)    --->   int f<int>(const int&)
    // 10. auto&& f(auto&& v)  --->   int& f<int&>(int&)
    // 11. auto&& f(auto&& v) { return std::forward<decltype(v)>(v); }
    //     --->   int& f<int&>(int&)
    // 12. decltype(auto) f(auto&& v) { return std::forward<decltype(v)>(v); }
    //     --->   int& f<int&>(int&)
    // 13. auto f(T&& v) -> decltype(v) { return std::forward<T>(v); }
    // --->   int& f<int&>(int&)

    decltype(auto) r2 = f(a);
    // 1. T f(T v) --->   int f<int>(int)
    // 2. T f(T& v)    --->   int f<int>(int&)
    // 3. T f(const T& v)  --->   int f<int>(const int&)
    // 4. T f(T&& v)   --->   int& f<int&>(int&)
    // 5. T& f(T&& v)  --->   int& f<int&>(int&)
    // 6. T&& f(T&& v) { return std::forward<T>(v); }  --->   int& f<int&>(int&)
    // 7. auto f(auto v)   --->   int f<int>(int)
    // 8. auto f(auto& v)  --->   int f<int>(int&)
    // 9. auto f(const auto& v)    --->   int f<int>(const int&)
    // 10. auto&& f(auto&& v)  --->   int& f<int&>(int&)
    // 11. auto&& f(auto&& v) { return std::forward<decltype(v)>(v); }
    //     --->   int& f<int&>(int&)
    // 12. decltype(auto) f(auto&& v) { return std::forward<decltype(v)>(v); }
    //     --->   int& f<int&>(int&)
    // 13. auto f(T&& v) -> decltype(v) { return std::forward<T>(v); }
    // --->   int& f<int&>(int&)

    decltype(auto) r3 = f(b);
    // 1. T f(T v) --->   int f<int>(int)
    // 2. T f(T& v)    --->   const int f<const int>(const int&)
    // 3. T f(const T& v)  --->   int f<int>(const int&)
    // 4. T f(T&& v)   --->   const int& f<const int&>(const int&)
    // 5. T& f(T&& v)  --->   const int& f<const int&>(const int&)
    // 6. T&& f(T&& v) { return std::forward<T>(v); }
    //     --->   const int& f<const int&>(const int&)
    // 7. auto f(auto v)   --->   int f<int>(int)
    // 8. auto f(auto& v)  --->   int f<const int>(const int&)
    // 9. auto f(const auto& v)    --->   int f<int>(const int&)
    // 10. auto&& f(auto&& v)  --->   const int& f<const int&>(const int&)
    // 11. auto&& f(auto&& v) { return std::forward<decltype(v)>(v); }
    //     --->   const int& f<const int&>(const int&)
    // 12. decltype(auto) f(auto&& v) { return std::forward<decltype(v)>(v); }
    //     --->   const int& f<const int&>(const int&)
    // 13. auto f(T&& v) -> decltype(v) { return std::forward<T>(v); }
    // --->   const int& f<const int&>(const int&)

    decltype(auto) r4 = f(std::move(a));
    // 1. T f(T v) --->   int f<int>(int)
    // 2. T f(T& v)    --->   Error!
    // 3. T f(const T& v)  --->   int f<int>(const int&)
    // 4. T f(T&& v)   --->   int f<int>(int)
    // 5. T& f(T&& v)  --->   int& f<int>(int&&)
    // 6. T&& f(T&& v) { return std::forward<T>(v); }  --->   int&& f<int>(int&&)
    // 7. auto f(auto v)   --->   int f<int>(int)
    // 8. auto f(auto& v)  --->   Error!
    // 9. auto f(const auto& v)    --->   int f<int>(const int&)
    // 10. auto&& f(auto&& v)  --->   int f<int>(int&&)
    // 11. auto&& f(auto&& v) { return std::forward<decltype(v)>(v); }
    //     --->   int&& f<int>(int&&)
    // 12. decltype(auto) f(auto&& v) { return std::forward<decltype(v)>(v); }
    //     --->   int&& f<int>(int&&)
    // 13. auto f(T&& v) -> decltype(v) { return std::forward<T>(v); }
    // --->   int&& f<int>(int&&)
}
```

13. Неявные шаблоны. Протаскивание типа передаваемого параметра через шаблон (шаблон std::forward). Определение типа с помощью decltype. decltype(auto).

В языке Си приходилось создавать подобные функции, работающие с разными типами данных. По существу, был copy-past кода под разные типы данных. Можно решать эту проблему с помощью макросов с параметрами, но это крайне опасная вещь. На этапе препроцесирования происходит подстановка макроса в код, на этапе компиляции идет проверка подставленного. Если у нас ошибка в макросе, то определить её крайне сложно.

```
#define print(a) printf("%d \n", a)
#define min(a,b) (a < b ? a : b)
```

В языке C++ эта проблема подобного копирования кода решается с помощью шаблонов. Мы можем определить шаблон. Это может быть функция, класс, метод класса и также мы можем определить шаблон имени типа. Во время компиляции будет подстановка значения параметров шаблона с проверкой шаблона.

Шаблон не является ни классом, ни функцией. Функция или класс генерируется на основе шаблона во время использования. Компилятор встречает вызов функции, смотрит, может ли использоваться шаблон, и, если может, создаёт по шаблону функцию или класс.

Неявные шаблоны – функции, в которых вместо типов используется *auto*. Происходит то же самое, что и при обычных шаблонах, но встречаются некоторые нюансы.

```
auto f(auto v)
```

Но если мы хотим вернуть тип v, может произойти срезание ссылок (если бы мы передавали, например, *&&v*). Так же невозможно определить функцию, например, в заголовочном файле, т.к. ее тип не известен.

Срезание ссылок — это механизм, который определяет, как типы ссылок комбинируются при объявлении или использовании шаблонов с параметрами ссылочного типа.

- T& & становится T&
- T& && становится T&
- T&& & становится T&
- T&& && становится T&&

Decltype определяет тип выражения. С помощью него можно избежать срезания ссылок.

Decltype(auto) позволяет определить функцию с неизвестным возвращаемым типом, и, главное, избежать срезания ссылок.

```
decltype(auto) f(auto &&v)
{
    return std::forwart<decltype(v)>(v);
}
```

Использование *forward* для идеальной передачи (*lvalue-copy, rvalue-move*):

```
class A
{
public:
    A() = default;
    A(const A&) { cout << "Copy constructor" << endl; }
    A(A&&) noexcept { cout << "Move constructor" << endl; }
};

template <typename Func, typename Arg>
decltype(auto) call(Func&& func, Arg&& arg)
{
//    return func(arg);
    return forward<Func>(func)(forward<Arg>(arg));
}

A f(A a) { cout << "f called" << endl; return a; }

int main()
{
    A obj{};

    auto r1 = call(f, obj);
    cout << endl;
    auto r2 = call(f, move(obj));
}
```

14. Специализация шаблонов функций. Шаблоны типов. Шаблоны классов. Полная или частичная специализация шаблонов классов. Параметры шаблонов, задаваемых по умолчанию. Шаблоны с переменным числом параметров. Пространства имен.

В языке Си приходилось создавать подобные функции, работающие с разными типами данных. По существу, был copy-past кода под разные типы данных. Можно решать эту проблему с помощью макросов с параметрами, но это крайне опасная вещь. На этапе препроцесирования происходит подстановка макроса в код, на этапе компиляции идет проверка подставленного. Если у нас ошибка в макросе, то определить её крайне сложно.

```
#define print(a) printf("%d \n", a)
#define min(a,b) (a < b ? a : b)
```

В языке C++ эта проблема подобного копирования кода решается с помощью шаблонов. Мы можем определить шаблон. Это может быть функция, класс, метод класса и также мы можем определить шаблон имени типа. Во время компиляции будет подстановка значения параметров шаблона с проверкой шаблона.

Шаблон не является ни классом, ни функцией. Функция или класс генерируется на основе шаблона во время использования. Компилятор встречает вызов функции, смотрит, может ли использоваться шаблон, и, если может, создаёт по шаблону функцию или класс.

Шаблонный тип. В языке Си для задания имени типа использовался typedef. Это аналог определения какой-либо переменной, только вместо этого - имя типа.

Например, определим имя для типа, принимающего указатель на функцию, принимающую int и возвращающую void:

```
typedef void (*Tpf)(int);
```

В языке C++ шаблон на основе typedef мы определить не можем. Добавляется еще одна конструкция - using, выполняющая ту же функцию, что и typedef. После using записываем имя типа, а далее записывается так называемый абстрактный описатель (определение переменной без ее имени).

```
using Tpf = void (*)(int);
```

Две вышеприведенные записи с typedef и using эквивалентны, но для typedef мы не можем определить шаблон. Пример шаблонного типа:

```
template<typename T>
using cmp_t = int(*) (const T&, const T&);
//...
cmp_t<double> v;
```

Определение шаблона класса:

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

Методы шаблона класса также являются шаблонами. В принципе, у нас может быть нешаблонный класс, но с шаблонными методами, то есть в обычном классе можно определить шаблонный метод. Шаблонные методы поддерживают шаблон класса. Методы будут создаваться только для того типа, которого класс. Не будет проблемы срезания ссылок.

Так же, как и у любой функции, у шаблона метода может быть специализация.

Для вызовов методов действует такое же правило, как и для функций: сначала для созданного класса, если есть специализация выбирается специализация, если нет специализации, создается метод по шаблону.

Полная специализация.

```
template <typename T>
class A {...};

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

Специализация является тоже шаблоном. По специализации так же создается класс, если нужно. У нас создаются классы по специализации. Специализация может иметь отличные параметры от шаблонов, вернее тело специализации может быть другим: другие члены, данные, методы, отличные от самого шаблоны.

Пример:

```
template <typename Type>
class A
{
public:
    A() { cout << "constructor of template A;" << endl; }
    void f() { cout << "metod f of template A;" << endl; }
};

template <>
void A<int>::f() { cout << "specialization of metod f of template A;" << endl; }

template <>
class A<float>
{
public:
    A() { cout << "specialization constructor template A;" << endl; }
    void f() { cout << "metod f specialization template A;" << endl; }
    void g() { cout << "metod g specialization template A;" << endl; }
};

int main()
{
    A<double> obj1;
    obj1.f();

    A<float> obj2;
    obj2.f();
    obj2.g();

    A<int> obj3;
    obj3.f();
}
```

Частичная специализация - когда мы указываем не все значения параметра шаблона. При частичной специализации шаблонов возможна неоднозначность при вызове. Здесь аналогичный подход с функциями: сначала идет выбор специализации, если невозможно создать класс по специализации, создается класс по шаблону.

```
template <typename T1, typename T2 = double>
class A
{
public:
    A() { cout << "constructor of template A<T1, T2>;" << endl; }
};

// Specialization #1
template <typename T>
class A<T, T>
{
public:
    A() { cout << "constructor of template A<T, T>;" << endl; }
};

// Specialization #2
template <typename T>
class A<T, int>
{
public:
    A() { cout << "constructor of template A<T, int>;" << endl; }
};
```

```cpp
// Specialization #3
template <typename T1, typename T2>
class A<T1*, T2*> {
public:
    A() { cout << "constructor of template A<T1*, T2*>;" << endl; }
};


int main()
{
    A<int> a0;           // Template
    A<int, float> a1;      // Template
    A<float, float> a2;        // Specialization #1
    A<float, int> a3;      // Specialization #2
    A<int*, float*> a4;        // Specialization #3

    // A<int, int> a5;        // Error!!!
    // A<int*, int*> a6;   // Error!!!
}
```

Так же, как при определении функций, параметры шаблона могут быть по умолчанию. В случае выше сам шаблон имеет один параметр по умолчанию - double. Если мы передаем только один параметр, будет вызываться этот шаблон (а второй параметр по умолчанию типа double):

```cpp
template <typename T1, typename T2 = double>
class A {
public:
    A() { cout << "constructor of template A<T1, T2>;" << endl; }
};
```

Так же как функции, шаблоны мы можем создавать с переменным числом параметров. Это могут быть шаблоны функций, методов и классов.

```cpp
template <typename Type>
Type sum(Type value)
{
    return value;
}

template <typename Type, typename ...Args>
Type sum(Type value, Args... args)
{
    return value + sum(args...);
}

int main()
{
    cout << sum(1, 2, 3, 4, 5) << endl;
    return 0;
}
```

Программа растет, она становится большой. Во время разрастания программы может возникнуть проблема конфликта имён. Мы используем разные библиотеки, в одной библиотеке нужно что-то взять, в другой что-то взять. у некоторых библиотек может дублироваться функционал. Это типичная ситуация. Например, несколько реализаций функции с именем swap в различных библиотеках. Когда подключаем разные библиотеки, может возникнуть конфликт имен. Из какой библиотеки мы вызываем swap? Это так же может касаться имен классов, методов, которые мы используем.

В C++ мы можем задавать пространства имён.

Синтаксис такой:

```
namespace <имя>
{
<блок пространства имён>
}

// Доступ к пространству имён
<имя>::f(); // f() - член пространства имен
```

Или можно сделать так:

```
// или можно включить это пространство и использовать f
using namespace <имя>;
f(); // но таким образом можно вернуться к изначальной проблеме
```

В примере выше тоже можно натолкнуться на проблему конфликта имён, так как подключив несколько namespace, может возникнуть та же самая ситуация. Поэтому, когда у нас есть много пространств имён с пересекающимися параметрами, лучше использовать синтаксис :: - это поможет избежать конфликта.

Пространствами имён злоупотреблять не надо. Вложенных пространств имён надо избегать или сводить к минимуму.

Имя пространства имён может отсутствовать — это анонимные пространства имён. Их особенность в том, что из другого файла нельзя получить доступ к членам анонимного пространства имён. Грубо говоря, если у нас часть, и мы не хотим, чтобы она была видна из других частей, мы можем определить это, как анонимное пространство имен.

15. Ограничения, накладываемые на шаблоны. Требования к шаблонам (requires). Концепты. Типы ограничений. Варианты определения шаблонов функций и классов с концептами.

Ограничения шаблонов (как функций, так и классов) позволяют ограничить набор возможных типов, которые будут применяться параметрами шаблонов. Добавляя ограничения к параметрам шаблона, решаются следующие задачи:

- Из заголовка шаблона сразу видно, какие аргументы шаблона разрешены, а какие нет.
- Шаблон создается только в том случае, если аргументы шаблона удовлетворяют всем ограничениям.
- Любое нарушение ограничений шаблона приводит к сообщению об ошибке, которое гораздо ближе к первопричине проблемы, а именно к попытке использовать шаблон с неверными аргументами.

Начиная со стандарта C++20 в язык был добавлен оператор requires, который позволяет установить для параметров шаблонов ограничения.

```
template <параметры> requires ограничения
содержимое шаблона;
```

```
#include <iostream>

template <typename T> requires std::is_same<T, int>::value || std::is_same<T,
double>::value
T sum(T a, T b){ return a + b;}

int main()
{
    std::cout << sum(3, 4) << std::endl;
    std::cout << sum(12.5, 4.3) << std::endl;
    //std::cout << sum(5l, 7l) << std::endl;
}
```

Начиная со стандарта C++20 в язык C++ была добавлена такая функциональность как concepts (концепты). Концепты позволяют установить ограничения для параметров шаблонов (как шаблонов функций, так и шаблонов класса).

Концепт фактически представляет шаблон для именованного набора ограничений, где каждое ограничение предписывает одно или несколько требований для одного или нескольких параметров шаблона. В общем случае он имеет следующий вид:

```
template <параметры>
concept имя_концепта = ограничения;
```

Список параметров концепта содержит один или несколько параметров шаблона. Во время компиляции компилятор оценивает концепты, чтобы определить, удовлетворяет ли набор аргументов заданным ограничениям.

Простейший пример:

```
template <typename T>
concept size = sizeof(T) <= sizeof(int);
```

Концепты бывают 4 видов:

1. Простые
2. Составные
3. Типовые
4. Вложенные

Пример:

```
# include <iostream>
# include <vector>

using namespace std;

template <typename T>
concept HasBeginEnd = requires(T a)
{
    a.begin();
    a.end();
};

# define PRIM_1

# ifdef PRIM_1
template <typename T>
requires HasBeginEnd<T>
```

```
ostream& operator <<(ostream& out, const T& v)
{
    for (const auto& elem : v)
        out << elem << endl;

    return out;
}

# elif defined(PRIM_2)
template <HasBeginEnd T>
ostream& operator <<(ostream& out, const T& v)
{
    for (const auto& elem : v)
        out << elem << endl;

    return out;
}

# elif defined(PRIM_3)
ostream& operator <<(ostream& out, const HasBeginEnd  auto& v)
{
    for (const auto& elem : v)
        out << elem << endl;

    return out;
}

# endif

int main()
{
    vector<double> v{ 1., 2., 3., 4., 5. };
    cout << v;
}
```

Варианты использования концепта:

```
template <typename T>
concept Incrementable = requires(T t)
{
    {++t} noexcept;
    t++;
};

# define PRIM_1

# ifdef PRIM_1
template <typename T>
requires Incrementable<T>
auto inc(T& arg)
{
    return ++arg;
}

# elif defined(PRIM_2)
template <typename T>
auto inc(T& arg) requires Incrementable<T>
{
    return ++arg;
}

# elif defined(PRIM_3)
template <Incrementable T>
auto inc(T& arg)
{
    return ++arg;
}
```

```
# elif defined(PRIM_4)
auto inc(Incrementable auto& arg)
{
    return ++arg;
}

# elif defined(PRIM_5)
template <typename T>
requires requires(T t)
{
    {++t} noexcept;
    {t++};
}
auto inc(T& arg)
{
    return ++arg;
}

# endif

class A {};

int main() {
    int i = 0;

    cout << "i = " << inc(i) << endl;

    A obj{};
    //    cout << "obj = " << inc(obj) << endl;
}
```

16. Проблемы с динамическим выделением и освобождением памяти. Шаблон Holder. «Умные указатели» в C++: unique_ptr, shared_ptr, weak_ptr. Связь между shared_ptr и weak_ptr.

Существует проблема. Предположим, у нас есть класс A, в котором есть метод f(). Мы не знаем, что творится внутри f(), и, естественно, мы используем механизм обработки исключительных ситуаций. Внутри f() происходит исключительная ситуация, она приводит к тому, что мы перескакиваем на какой-то обработчик, неизвестно где находящийся. Это приводит к тому, что объект p не удаляется - происходит утечка памяти.

```
A* p = new A;
p->f();   // Внутри f() происходит исключительная ситуация
delete p; // Объект p не удаляется
```

Идея: обернуть объект в оболочку, которая статическая распределяет память. Эта оболочка будет отвечать за этот указатель. И соответственно, поскольку мы статически распределили, когда будет вызываться деструктор, в деструкторе мы будем освобождать память.

Шаблон Holder. Мы можем указатель p обернуть в объект-хранитель. Этот объект будет содержать указатель на объект A. Задача объекта: при выходе из области видимости объекта-хранителя будет вызываться деструктор obj, в котором мы можем уничтожить объект A.

```
Holder<A> obj(new A);
```

Для объекта хранителя достаточно определить три операции - * (получить значение по указателю), ->(обратиться к методу объекта, на который указывает указатель) и bool(проверить, указатель указывает на объект, nullptr он или нет). Чтобы можно было записать obj->f();. То есть эта оболочка должна быть "прозрачной". Её задача должна быть только вовремя освободить память, выделенную под объект. Мы работаем с объектом класса A через эту оболочку.

```
template <typename Type>
class Holder
{
private:
    Type* ptr{ nullptr };

public:
    Holder() = default;
    explicit Holder(Type* p) : ptr(p) {}
    Holder(Holder&& other) noexcept
    {
        ptr = other.ptr;
        other.ptr = nullptr;
    }
    ~Holder() { delete ptr; }

    Type* operator ->() noexcept { return ptr; }
    Type& operator *() noexcept { return *ptr; }
    operator bool() noexcept { return ptr != nullptr; }
    Type* release() noexcept
    {
        Type* work = ptr;
        ptr = nullptr;

        return work;
    }

    Holder(const Holder&) = delete;
    Holder& operator =(const Holder&) = delete;
};

class A
{
public:
    void f() { cout << "Function f of class A is called" << endl; }
};

int main()
{
    Holder<A> obj(new A{});

    obj->f();
}
```

Проблема висящего указателя. Этот хранитель решает ситуацию, связанную с обработкой исключительных ситуаций. Но предположим, что у нас есть один объект класса A и класс B держит указатель на объект класса A.

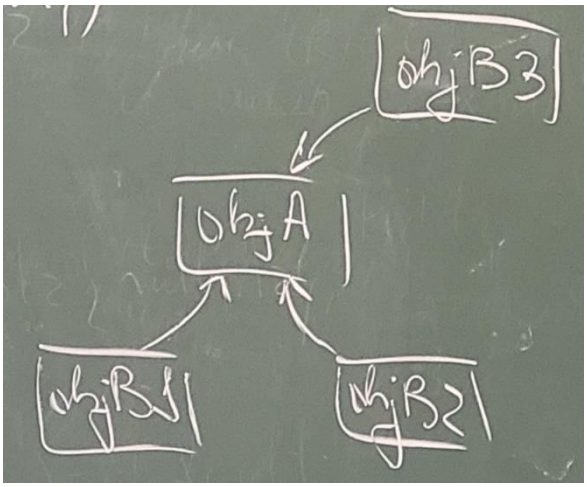
```
class A {...};

class B
{
    A* p;
}
```

Например, мы получили указатель p. Этот объект может быть удалён, и в этом случае возникает проблема: указатель, инициализированный каким-то адресом, будет указывать на удалённый объект. Можно рассматривать каждый объект, который держит указатель, как хранитель. То есть мы отдаём указатель на объект, а объект-хранитель считает, что этот объект его собственный, происходит захват.

В случае если хранитель отдаёт объект, нужно позаботиться о том, чтобы не образовался "висящий" указатель, то есть указатель на объект, которого нет.

Проблема с утечкой памяти не такая острая как проблема с висящим указателем. Утечка памяти приводит всего лишь к нехватке памяти, в то время как с висящим указателем мы можем случайно вызвать метод несуществующего объекта, что приведёт к падению системы.

Представим, что на один объект держат указатели несколько объектов. Как понять, какой из объектов должен удалять этот указатель? Если это отдавать на откуп программиста, то о надежности такого кода говорить нельзя, возможно ошибка. Допустим, мы выбрали один из объектов ответственным. Какая гарантия, что он не уничтожится раньше, чем другие два объекта?



Идея: последний, кто уезжает, выключает свет. То есть последний объект (класса B), который будет уничтожаться, он должен позаботиться об объекте класса A.

Умные указатели решают проблемы с утечкой памяти и с висящим указателем. Первоначально эту проблему пытались решить одним указателем, но скоро поняли, что одним указателем решить проблему невозможно.

Существует три вида умных указателей, каждый решает свою проблемы.

unique_ptr

Пример с Holder (ранее), по существу, представляет собой указатель unique_ptr. Жестко сберегает какой-то один объекта. Он хранит уникальную ссылку на объект и не позволяет другим указателям владеть этим объектом.

Применение unique_ptr:

```
class A
{
public:
    A() { cout << "Constructor" << endl; }
    ~A() { cout << "Destructor" << endl; }

    void f() { cout << "Function f" << endl; }
};

int main() {
    unique_ptr<A> obj1(new A{});
    unique_ptr<A> obj2 = make_unique<A>();
    unique_ptr<A> obj3(obj1.release()); // move(obj1)
```

```
    obj1 = move(obj3);

    if (!obj3) {
        A *p = obj1.release();

        obj2.reset(p);
        obj2->f();
    }
}
```

shared_ptr и weak_ptr

Если shared_ptr обеспечивает нас счетчиком, это так называемое совместное владение, то а паре с ним идет weak_ptr - слабое владение. Этот указатель не отвечает за освобождение памяти из-под объекта. Он может только проверить, есть объект или его нет. Эти два указателя связаны между собой.

У нас должен быть счетчик countS, определяющий, сколько объектов указывают на сберегаемый объект. Указателю weak_ptr тоже надо знать об этом счетчике.

Пусть есть какой-то базовый класс, от которого порождаем два класса: shared_ptr и weak_ptr. И тому и другому нужен указатель на object и нужен счетчик countS. Базовый класс содержит указатель на объект, и счетчик countS тоже должен быть доступен всем shared_ptr и weak_ptr, следовательно, счетчик countS мы тоже должны вынести, как объект.

Так как память object вынесли, по счетчику countS weak_ptr определяет, есть ли этот object или нет. Если счетчик равен нулю, то объекта нет. Когда создается новый shared_ptr на область памяти object, счетчик countS увеличивается. Удаляется shared_ptr - счетчик уменьшается. Если счетчик равен нулю - эта память должна быть освобождена.

А что будет отвечать за память счетчика countS, когда освободится память из-под object? Здесь встает необходимость считать не только количество объектов shared_ptr, но и количество объектов weak_ptr. То есть, по существу, у нас не один счетчик, а два. Счетчик weak_ptr нужен для того, что, если он станет равен нулю, и второй счетчик равен нулю, освободить эту память. Соответственно, общий класс для shared_ptr и weak_ptr может решать эту проблему. Он будет контролировать и память объектов, и область счетчика.

Memory	Владение	Операторы	Копия	Методы
unique_ptr	строгое	`*, ->, bool, []`	Нет	get, release, reset, swap
shared_ptr	совместное	`*, ->, bool, []`	Да	get, reset, use_count, unique (true, если счётчик shared равен 1, иначе false)
weak_ptr	слабое	Нет	Да	use_count, expired (возвращает признак, есть объект или его нет), reset, lock (возвращает shared_ptr, на основе weak мы создаём shared)

```cpp
class SomeClass {
public:
    void sayHello() {
        std::cout << "Hello!" << std::endl;
    }
    ~SomeClass() {
        std::cout << "~SomeClass" << std::endl;
    }
};

int main() {
    std::weak_ptr<SomeClass> wptr;
    {
        auto ptr = std::make_shared<SomeClass>();
        wptr = ptr;

        if(auto tptr = wptr.lock())
            tptr->sayHello();                          // !
        else
            std::cout << "lock() failed" << std::endl;
    }

    if(auto tptr = wptr.lock())
        tptr->sayHello();
    else
        std::cout << "lock() failed" << std::endl;     // !
}
```

17. Приведение типа в C++: static_cast, dynamic_cast, const_cast, reinterpret_cast. Контейнерные классы и итераторы. Требования к контейнерам и итераторам. Категории итераторов. Операции над итераторами. Цикл for для работы с контейнерными объектами.

В языке C++ решили использовать разные варианты приведения типов.

По умолчанию всегда указатель на производный класс приводится к указателю на базовый класс (или ссылка). А если нужно обратное преобразование?

Для обратного преобразования появились два оператора преобразования:

- Первый оператор преобразования выполняется на этапе компиляции - static_cast
- Второй оператор преобразования на этапе выполнения - dynamic_cast.

static_cast

На этапе компиляции выполняется оператор static_cast. Этот оператор используется для приведения родственных классов, находящихся по одной ветви наследования. Также используется для стандартных типов, для которых определён механизм явного приведения.

Рассмотрим следующую иерархию:

С помощью static_cast мы можем от указателя A привестись к указателю B или к указателю на класс C, или от A к D. Он не позволит нам привестись от класса указателя на B к D. Они родственные, но находятся по разным ветвям.

```cpp
A *pa = new B;               // У нас есть указатель
B *pb = static_cast<B*>(pa); // Приведение
```

Проблема — это выполняется на этапе компиляции. На этапе компиляции невозможно проверить, что это за объект, то есть приведение будет срабатывать, но указатель pa может не указывать на объект класса B. Мы можем написать такую строчку:

```
A *pa = new B;                // У нас есть указатель
C *pc = static_cast<C*>(pa); // Приведение
```

Что будет в таком случае – неизвестно.

Если приведение невозможно, будет выдаваться ошибка на этапе компиляции. Хотелось бы, чтобы это проверялось на этапе выполнения.

dynamic_cast

Оператор dynamic_cast делает проверку на этапе выполнения. Он приводит к типу, если реально указатель указывает на объект этого типа. Если нет - возвращает указатель на NULL. Приведение может быть не только указателем, но и ссылкой. Для dynamic_cast есть жесткое требование - базовый классы должен быть полиморфными (то есть либо virtual метод, либо virtual деструктор).

Пример:

```
pb = dynamic_cast<B*>(pa);
if (pb)
```

Можно проверить: если pb не равно нулю, приведение осуществилось, иначе не осуществилось. В данном случае всё будет нормально, так как pa указывает на объект класса B - приведение возможно. А приведение, например, к указателю на класс C невозможно из pa, так как pa указывает на объект другого класса.

Такое приведение типов удобно тем, что это удобно на этапе выполнение программ.

Если работаем со ссылкой в dynamic_cast - вместо NULL возникает исключение bad_cast, которое можно отловить. Исключения ловить неприятно, поэтому лучше работать с указателями.

const_cast

Мы работаем с модификатором const. Мы контролируем, что объект может быть константными, контролируем методы. Но есть проблема - мы не можем менять поля константных объектов. Иногда возникают ситуации, когда нам необходимо менять.

Предположим, у нас есть объект, который держит указатель на другой. Мы определили его, как константный, но этот указатель мы хотим отобрать от него. Чтобы отобрать, нам нужно это поле обнулить. А сделать это мы не можем, так как не можем обнулять поля константных объектов.

Чтобы убрать модификатор const, используется оператор const_cast. Есть компиляторы, которые не позволяют изменять константность объектов.

reinterpret_cast

Также существует оператор, эквивалентный С-му приведению - reinterpret_cast. Может приводить из любого типа.

```
class A {...};
A* p = new A;
char* pbyte = reinterpret_cast<char*>(p); // Можем выполнить такой бандитизм
```

Мы поставили указатель типа char на первый байт объекта класса А.

Это то же самое, как преобразование, которое было в языке Си. Небезопасное преобразование. Неизвестно, к чему это приведет.

Контейнерные классы и итераторы. Пусть у нас есть класс:

```
struct List
{
    Node *first, *last;
};
```

Идея: выделить текущий указатель в структуру и создавать по надобности:

```
struct Iterator
{
    Node *curent;
};
```

Стоит задача унификации работы с контейнерными данными. Можно идти по пути унификации интерфейса, чтобы разные контейнерные классы имели один и тот же интерфейс. Желательно, чтобы каждая сущность имела свой интерфейс.

Идея простая: унифицировать работу с разными контейнерами за счёт итератора. А у итератора будет унифицированный интерфейс.

Есть базовый шаблон - итератора, и уже конкретный итератор - специализация. 5 видов специализации. Специализация определяется тэгом. Тэг - простейшая структура.

```
struct output_iterator_tag {}; // итератор вывода
struct input_iterator_tag {}; // итератор ввода
struct forward_iterator_tag {}; // однонаправленный итератора на чтение-запись
struct bi_directional_iterator_tag {}; // двунаправленный итератор на чтение-запись
struct random_access_iterator_tag {}; // произвольный доступ
```

Первый допускает операторы - *, ++

Второй, третий и четвертый - *, ->, ++, !=, ==

Пятый – it - n, it + n, += , ==, it1 - it2, []

Раньше мы вынуждены были любой итератор порождает от этих базовых итераторов. В принципе, в современных стандартах можно свой итератор не порождать от этих стандартных итераторов.

В языке C++ появился цикл foreach:

```
for (auto elem: obj)
cout << elem;
// Цикл выше разворачивается в
for (Iterator<Type> It = obj.begin(); It != obj.end(); ++It) {
auto elem = *It;
cout << elem;
}
```

Мы обязаны в контейнерный класс добавить методы: begin, end, cbegin, cend (c - const), rbegin, rend, crbegin, crend. Также добавить операторы: !=, ++, * (++ - префиксный инкремент).

Контейнер может быть хранитель, но есть проблема. Мы на контейнеры можем создавать итераторы. По этой причине "голенький" указатель в контейнере хранить нельзя. То есть итератор должен хранить weak_ptr. Контейнер должен оборачивать данные в shared_ptr.

Существуют классы, которые содержат в себе другие классы. Например, множество, вектор и т. п. Такие классы называются контейнерными - они включают в себя другие классы.

Для работы с контейнерными классами, нам, во-первых, необходимо абстрагироваться от внутренней структуры организации контейнера - нас это не должно интересовать, а во-вторых, мы можем обрабатывать контейнер вне зависимости от типа объекта внутри контейнера. Для этого были придуманы классы-итераторы. Существуют стандартные итераторы, поэтому задача программиста - задавать общий механизм работы с итераторами. Классы, которые отвечают за просмотр содержимого в других объектах, называют итераторами.

Есть стандартный итератор - шаблон класса Iterator. У него есть специализации под разные виды работы с итераторами. Итераторы делятся на итераторы ввода (мы можем менять то, на что итератор указывает) и итераторы вывода (мы НЕ можем менять то, на что итератор указывает, то есть мы можем только читать, но не записывать).

Специализация ввода или вывода задаётся при наследовании пользовательского итератора от стандартного итератора.

Итератор может рассматривать контейнер как направленную последовательность, двунаправленную последовательность и последовательность произвольного доступа.

С помощью итератора можно просматривать содержимое контейнера.

Благодаря итераторам в C++ стало возможным создать оператор foreach. Для работы с этим оператором контейнерный класс должен содержать два метода: метод begin(), указывающий на начало последовательности, и метод end(), указывающий на конец последовательности. Конец — это не последний элемент, а ЗА последним элементом.

```
for (<тип>& <имя> : <объект>)
{
...
}
```

При реализации следует предусмотреть итераторы для работы как с константным контейнерным классом, так и с неконстантным.