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

ОТВЕТ: Структура программы на языках C и C++. Функции C и C++. Перегрузка функций в C++. Ссылки.

Структура программы:

- Набор файлов, содержимым которых может являться различные объявления и определения (функций, переменных и т.п.), директивы препроцессора.
- Программа на C/C++ состоит из одной или более подпрограмм.
- Функция в языке C/C++ должна иметь свое имя, возвращаемый тип, типы принимаемых параметров и тело функции. Функцию можно сначала объявить (не описывать тело), а лишь потом определить (повторить объявление, но уже с описанием тела функции). Объявлений может быть сколько угодно, определение только одно.
- Главная функция – `main`. Должна обязательно быть, не может вызывать саму себя.
- Файлы компилируются раздельно. Механизм раздельной компиляции заключается в том, чтобы получить исполняемый файл в два этапа (этап компиляции и линковки).
    - На этапе компиляции каждый файл с исходным кодом компилируется независимо, в результате чего из этого файла получается объектный модуль.
    - На этапе линковки объектные модули соединяются в один исполняемый файл с помощью линковщика(компоновщика). Кроме того, на этом этапе могут подключаться библиотеки, если таковые используются.
    - Чтобы нормально собрать программу, создают заголовочные файлы, где содержатся константы и объявления переменных и функций (без определения).
- Так же в программах, написанных на языке C/C++ помимо файлов с исходным кодом используются заголовочные файлы, которые содержат объявление функций (которые будут позже определены в файле с исходным кодом), типов, констант, переменных. Для защиты от множественного включения заголовочного файла используется include-guard (связка макросов `#ifndef` и `#define`) или макрос `#pragma once` (но он в отличие от include guard может некорректно работать с символическими ссылками):
    - тык
    - `#include "f.h"`
    - `int main(int argc, char* argv[]) {`
    - `    return 0;`
    - `}`
    - `// во избежание многократного объявления:`
    - `#pragma once - нежелательный вариант`
    - `// или так:`
    - `#ifndef G_H`
    - `#define G_H`
    - `//...`
    - `#endif // тут мы можем выбирать, защищать весь файл или только часть`

Отличия C++ от C:

- Классы и шаблоны
- Перегрузка функций
- Операторы `new` и `delete`
- Обработка исключений через `throw/catch`

Ссылки

Ссылка (alias) – механизм передачи параметров в функцию и возврата из функции. По сути это ещё одно имя того же самого данного.

Ссылка - не тип данных!

- Пример кода
- `int i;`
- `int& ai = i;`
- `ai = 5; //то же самое, что и i = 5;`

Но такое использование ссылок - абсурд. Ссылки надо использовать для того, чтобы избавиться от указателей.

О возврате. Нельзя возвращать ссылку на локальную переменную. Возврат по ссылке нужен для формирования левого выражения = , чтобы присвоить переменной нужное значение.

Перегрузка функций

Перегрузка функций – множество функций с одним и тем же именем (список параметров разный)

На перегрузку функции влияет:

- Разное количество принимаемых параметров.
- Разный тип переменных.
- `const` значение функции/метода. Если объект константный, то будет вызываться const метод, иначе - не `const`.

Тип возвращаемой переменной не имеет значения.

Параметры функций по умолчанию в C++: В C++ вы можете определить функцию с параметрами по умолчанию, что позволяет вызывать функцию, опуская один или несколько последних аргументов. Это удобно, когда функция должна выполняться с часто используемым набором параметров.

Пример:

```cpp
void display(int a, int b = 10, int c = 20) {
    std::cout << "a: " << a << ", b: " << b << ", c: " << c << std::endl;
}

int main() {
    display(1);        // Вывод: a: 1, b: 10, c: 20
    display(1, 2);     // Вывод: a: 1, b: 2, c: 20
    display(1, 2, 3);  // Вывод: a: 1, b: 2, c: 3
    return 0;
}
```

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

Ссылки в C++

1. **Ссылки (References)**:
   - Ссылка — это алиас (псевдоним) для уже существующего объекта.
   - Для создания ссылки используется символ `&`, например: `int a = 5; int& b = a;` — `b` теперь является ссылкой на `a`.
2. **lvalue и rvalue ссылки**:
   - **lvalue** (левый операнд): Это объект, который имеет адрес в памяти, т.е. его можно взять по ссылке. Пример: переменные, элементы массива.
   - **rvalue** (правый операнд): Это временные значения, которые не имеют постоянного адреса в памяти. Например, литералы или результат выражений (например, `x + y`).

   В C++:

   - **lvalue ссылка**: `int&` — ссылка на lvalue.
   - **rvalue ссылка**: `int&&` — ссылка на rvalue, введена в C++11, используется для оптимизации через перемещение (move semantics).

   Пример:

   ```
   cpp
   КопироватьРедактировать
   int a = 5;    // lvalue
   int&& b = 10; // rvalue (временное значение)
   ```

3. **Передача параметров в функции по ссылке**:

- **По обычной ссылке (lvalue)**: Это позволяет функции изменять исходные данные.

```
void f(int& x) {
    x = 10; // Изменяет значение переменной, переданной в
функцию.
}
```

- **По rvalue ссылке (перемещение)**: Это позволяет принимать временные объекты, что полезно для оптимизации.

```
void f(int&& x) {
    // x может быть перемещён в другие ресурсы
}
```

4. **Автоматическое выведение типа**:
   - В C++11 и выше можно использовать **автоматическое выведение типа** с помощью `auto` для переменных и ссылок.
   - Пример:

```
int a = 5;
auto& b = a; // b - lvalue ссылка на int
auto&& c = 10; // c - rvalue ссылка на 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

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

Класс – тип данных, представляющий совокупность атрибутов объекта (переменные члены класса) и возможных операций над этими объектами (методов класса). Описывается в заголовочных файлах.

Механизм описания класса:

- `struct`
- `class`
- `union` – не может быть базовым классом, и не может быть производным (поэтому нет уровня доступа `protected`). Нужен только для совместимости с Си. Его использование нежелательно, т.к. нет контроля со стороны компилятора.

По умолчанию в структуре и объединении члены классов открыты, в классе закрыты, чем достигается принцип инкапсуляции (нет доступа к данным извне)

Другое отличие структуры от класса: схема наследования по умолчанию у класса `private`, а у структуры - `public`.

```
class<имя класса>[ : <список базовых классов>] {
private:  //доступ к данным есть только внутри самого класса
   int a;

protected:  //доступ есть внутри класса и во всех его наследниках
   int b;

public:  //доступны для внешнего кода
   int f();
};  //класс заканчивается ; как и структура

//В библиотечных классах лучше располагать члены в обратном порядке (начиная с public)
//Классы описываются в заголовочных файлах .h. Методы же определяются в .cpp
```

Размещение полей класса публичными – плохой тон, поскольку это противоречит принципу инкапсуляции (скрывание реализации, предоставление пользователю только интерфейса для работы).

Поля:

- Члены объекта
- Члены класса (`static`)

Методы:

- Методы объекта
- Методы класса (`static`)
- тык
- `class A`
- `private:`
- `    int a;                // Член объекта`
- `    const int ca;         // Константный член объекта`
- `    static int sa;        // Член класса`
- `    static const int sca; // Константный член класса`
-
- `public:`
- `    int f();              // Метод объекта`
- `    int f() const;         // Константный метод объекта`
- `    static int sg();      // Метод класса`
- `    A(int ia) :`
- `     // Объект еще не создан`
- `         a(ia),  // Так можно - объекта еще нет`
- `         ca(ia), // Так можно - объекта еще нет`
- `         sa(ia), // Ошибка - sa не член объекта`
- `         csa(ia) // Ошибка - cas не член объекта`
- `     {`
- `     // Объект создан`
- `         a = ia;   // Так можно`
- `         ca = ia;  // Константу изменить нельзя - объект уже создан`
- `         sa = ia;  // Так можно`
- `         csa = ia; // Константу изменить нельзя - объект уже создан`
- `     }`
- `};`
-
- `// static поля класса можно проинициализировать здесь`
- `int A::sa = 0;`
- `const int A::csa = 0;`

Константный метод объекта обязуется не изменять поля объекта.

Метод класса вызывается имя_класса::имя_метода. Метод объекта вызывается через сам объект. Методы объекта/класса имеют доступ ко всем элементам класса. Методы класса не получают `this`. Методы класса могут работать только со статическими членами.

Метод класса не может быть константным, т.к. не принимает указатель на объект (`this`)

- тык
- `int A::f() // :: - оператор доступа к контексту`
- `            // Используется для определения метода вне класса`
- `{`
- `    return this->a; // this – ключевое слово объекта, его можно не писать`
- `    // return a; - равносильно`
- `}`
-
- `int A::sg()`
- `{`
- `    return sa;`
- `}`

3 схемы наследования:

1. `private` (схема по умолчанию): все члены базового класса получают уровень доступа `private`, причём из унаследованного класса нет доступа к `private` полям базового класса.
2. `protected`: члены базового класса `public` получают уровень доступа `protected`
3. `public`: все члены базового класса наследуются со своим уровнем доступа.

- тык
- `class A`
- `{`
- `private:`
- `    int a;`
- `protected:`
- `    int b;`
- `public:`
- `    int f();`
- `};`
-
- `class B: {private(по умолчанию)/protected/public} A`
- `{`
- `private:`
- `    int c;`
- `protected:`
- `    int d;`
- `public:`
- `    int g();`
- `};`

1. Вариант с `private`: произошла полная подмена интерфейса (`f()`-> `g()`)
2. Вариант с `protected`: интерфейс тоже полностью подменяется, но у производных классов есть доступ к полям базового класса (`b` и `f()` перешли в уровень `protected`, все остальное без изменений).
3. Вариант с `public`: все члены со своим доступом, произошло расширение интерфейса.

Нужна возможность подмены интерфейса при наследовании (чтобы модифицировать программу, не изменяя уже написанный код), а наследование private и protected такой возможности не дают.

Модификатор `final` запрещает наследование от этого класса.

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

ОТВЕТ: Создание и уничтожение объектов в C++. Конструкторы и деструкторы. Виды конструкторов. Раздел инициализации конструкторов. Способы создания объектов.

В любой момент времени все члены данного объекта должны быть определены.

Конструктор - метод, который инициализирует объект класса

Когда используется конструктор:

- Когда определяем объект
- Когда идёт приведение типов
- Когда передаем параметры в метод по значению или возвращаем по значению
- Когда динамически выделяем память, используя оператор new (явный вызов).

Свойства:

- отсутствует тип возврата
- можно перегружать
- не наследуется
- если private, то невозможно создание производных классов
- не может быть volatile, static, const, virtual
- может быть constexpr (вычисляется на этапе компиляции), explicit (вызывается только явно)
- если в классе не определяется явно ни одного конструктора, то создаются конструктор по умолчанию (без параметров) и конструктор копирования
- если мы указали хотя бы один конструктор, то конструктор по умолчанию не создаётся
- конструктор копирования создаётся всегда.
- тык
- <имя класса>::<имя класса(конструктора)>([<Список параметров>])[:<раздел инициализации>] //отвечает за создание
- {
- тело конструктора // объект уже создан
- }

Конструктор копирования вызывается:

1. При инициализации одного объекта другим;
2. При передаче параметров по значению;
3. При возврате по значению.

Желательно у конструкторов с одним параметром (копирования и преобразования) ставить модификатор explicit.

Делегирующие конструкторы: можно использовать один конструктор в другом.

- тык
- class A
- {
- public:
- A(int i);
- A(double d) : A(1) {}
- };

В конструкторе можно использовать конструкторы базового класса:

- тык
- class B : class A
- {
- public:
- B(int i);
- using A::A; // Если бы этого не было, то при B(5.5) вызвался бы
- // не A(double d), а B(int i)
- // с неявным приведением типа
- };

Мы обязаны инициализировать в разделе инициализации:

- Базовые классы (если в базовых классах есть конструкторы по умолчанию, их можно не инициализировать, но если их нет, то мы обязаны явно вызвать конструкторы баз, они будут отрабатывать в том порядке, в котором мы их наследуем)
- Члены данных, если они являются объектами
- Константные члены объектов

При создании и во время работы объект может захватывать ресурсы, связи, может быть связан с другими объектами. Когда мы выходим из области видимости, объект уничтожается. В этом случае надо освободить ресурсы и оставить связи непротиворечивыми.

Деструктор - метод, который уничтожает объект класса.

Основной метод вызова деструктора - неявный. Начиная с C++17 можно явно вызывать деструктор полным квалификационным методом (с указанием имени класса, объекта)

Свойства:

- не может быть const, volatile, static
- может быть virtual
- не принимает параметров
- отсутствует тип возврата
- если явно не определить - создаётся по умолчанию
- не перегружается
- деструкторы вызываются в порядке, обратном порядку создания (по стеку)
- тык
- ~<имя класса>::<имя класса(деструктора)>() //отвечает за уничтожение
- {
- тело деструктора
- }

Порядок вызова: (1) ···

Aa Объект	Деструктор	Конструктор	
Глобальный	После функции main в порядке обратном определению	До функции main в порядке определения	
Локальный (статический)	Вызываются по стеку в порядке, обратном созданию, после функции main, но до уничтожения глобальных статических объектов	В порядке вызова при первой передачи управления в блок	
Локальный автоматический	после выхода из области видимости в порядке обратном созданию		

В любом классе нужно явно определить:

- Конструктор копирования
- Конструктор переноса
- Деструктор
- Оператор присваивания

Явный и неявный вызов конструкторов:

- **Явный вызов конструктора происходит, когда конструктор вызывается напрямую для создания объекта, например, через оператор new или когда объект создается статически с инициализацией.**
- **Неявный вызов конструктора случается, когда объект создается автоматически, например, при передаче объекта функции по значению или возвращении объекта из функции. Это также включает вызов конструктора по умолчанию при создании массива объектов.**

Приведение типа:

- **В C++ приведение типов может быть выполнено с использованием статического приведения (static_cast), динамического приведения (dynamic_cast), приведения констант (const_cast), и reinterpret_cast (reinterpret_cast). Каждый из этих методов используется в разных сценариях, в зависимости от требований безопасности и типа конверсии.**
- **Особенно важно использование dynamic_cast для безопасного приведения типов в иерархии наследования, особенно когда в игру вступают виртуальные функции.**

Дополнительные сведения о конструкторах и деструкторах:

- **Конструкторы и деструкторы могут быть defaulted или deleted с использованием соответствующих ключевых слов = default; и = delete; в современном C++. Это позволяет разработчикам явно указать, должны ли эти методы генерироваться компилятором по умолчанию или быть запрещенными.**
- **Использование ключевого слова explicit для конструкторов предотвращает неявные преобразования типов, что может улучшить типобезопасность кода и уменьшить вероятность ошибок из-за неожиданных преобразований.**

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

   Конструкторы копирования и переноса:
   1. **Конструктор копирования создает новый объект как копию существующего объекта. Синтаксис: ClassName(const ClassName& other). Если конструктор копирования не определен явно, компилятор может сгенерировать его автоматически.**
   2. **Конструктор переноса позволяет перенести ресурсы из одного объекта в другой, а не копировать их. Это эффективно для управления ресурсами в современном C++. Синтаксис: ClassName(ClassName&& other) noexcept. Конструктор переноса также может быть сгенерирован компилятором, если не определен явно.**

   Модификатор explicit: Модификатор explicit используется в объявлении конструкторов для предотвращения неявных преобразований типов. Это важно для избежания неожиданных ошибок и повышения ясности кода. Пример: explicit ClassName(int x).

   Удаление конструктора и default конструктор:
   - **Удаленный конструктор (= delete;) используется для запрета определенных способов создания объектов. Например, запретить копирование объекта: ClassName(const ClassName&) = delete;.**
   - **Конструктор по умолчанию (= default;) явно указывает, что компилятор должен сгенерировать конструктор по умолчанию, если другие конструкторы определены, но требуется также иметь конструктор по умолчанию.**

   Делегирующие и унаследованные конструкторы:
   - **Делегирующие конструкторы позволяют одному конструктору вызывать другой в том же классе, чтобы избежать дублирования кода. Пример: ClassName(int x) : ClassName(x, "default") {}.**
   - **Унаследованные конструкторы позволяют подклассу наследовать конструкторы своего базового класса, используя синтаксис using BaseClassName::BaseClassName;.**

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

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

Наследование – создание нового класса (понятия) на основе старого.

Расширение и выделение общей части из разных классов.

Причины выделения общей части у разных классов:

1. Общая схема использования объектов.
2. Сходство между наборами методов.
3. Сходство реализации методов.

Причины расщепления классов:

1. Один объект исполняет разные роли.
2. Два подмножества операций в классе используются в разной манере (один объект исполняет несколько ролей).
3. Методы класса (их реализации) между собой никак не связаны.
4. Одна сущность используется в разных, не связанных между собой частях программы.

За счёт наследования мы можем развивать нашу программу.

Полиморфизм неразрывно связан с наследованием.

Формировать иерархию наследования нужно в случаях: Выделение общей базы:

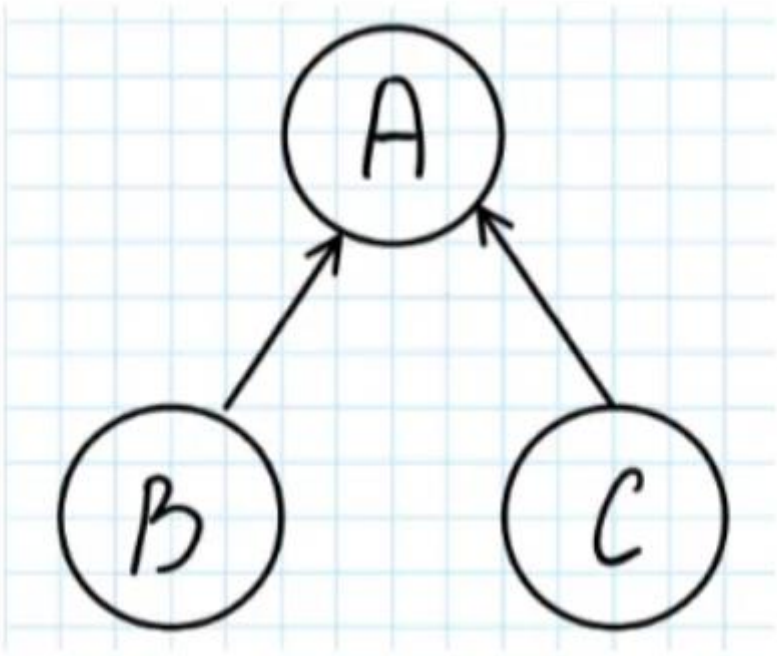
1. Если у двух (разных) понятий общая схема использования, мы обязаны объединить общим базовым классом, который определяет эту схему (создает интерфейс).
2. Разная схема использования, но общий набор методов (или несколько общих методов)
3. Разные понятия могут иметь разный набор методов, но общую реализацию (частично общую)

Выделенное понятие должно выполнять только одну роль.

Если на одну роль возлагают несколько ответственностей, то надо расщеплять класс.

Для данной роли необходимо использовать только ограниченный набор методов (использовать их в определённой манере).

На начальном этапе может быть одно понятие, с которым работаем в разных частях обсуждениях проекта. В дальнейшем это может быть не одна роль, а несколько ролей. Если мы обсуждаем одно и тоже понятие в несвязанных между собой частях проекта, необходимо изначально разбивать это понятие на два.



3 схемы наследования:

1. `private` (схема по умолчанию): все члены базового класса получают уровень доступа `private`, причём из унаследованного класса нет доступа к `private` полям базового класса.
2. `protected`: члены базового класса `public` получают уровень доступа `protected`
3. `public`: все члены базового класса наследуются со своим уровнем доступа.

- тык
- `class A`
- `{`
- `private:`
- `    int a;`
- `protected:`
- `    int b;`
- `public:`
- `    int f();`
- `};`
-
- `class B: {private(по умолчанию)/protected/public} A`
- `{`
- `private:`
- `    int c;`
- `protected:`
- `    int d;`
- `public:`
- `    int g();`
- `};`

1. Вариант с `private`: произошла полная подмена интерфейса (`f()`-> `g()`)
2. Вариант с `protected`: интерфейс тоже полностью подменяется, но у производных классов есть доступ к полям базового класса (`b` и `f()` перешли в уровень `protected`, все остальное без изменений).
3. Вариант с `public`: все члены со своим доступом, произошло расширение интерфейса.

Нужна возможность подмены интерфейса при наследовании (чтобы модифицировать программу, не изменяя уже написанный код), а наследование private и protected такой возможности не дают.

Модификатор `final` запрещает наследование от этого класса.

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

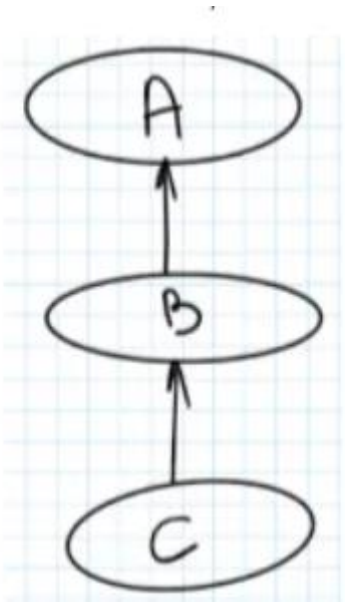
ОТВЕТ: Множественное наследование. Прямая и косвенная базы. Виртуальное наследование. Понятие доминирования. Порядок создания и уничтожения объектов. Проблемы множественного наследования. Неоднозначности при множественном наследовании.

ООП использует рекурсивный дизайн – постепенное разворачивание программы от базовых классов к более специализированным. C++ один из немногих языков с множественным наследованием. Оно может упростить граф наследования, но также создает пучок проблем для программиста: возникает неоднозначность, которую бывает тяжело контролировать.

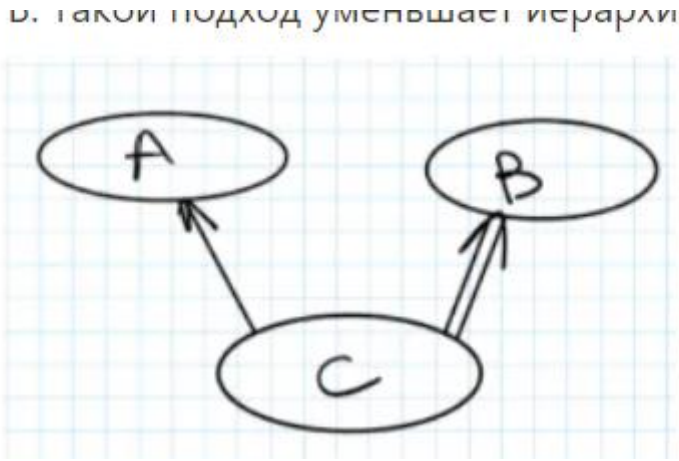
Преимущества множественного наследования:

Какие преимущества множественного наследования? Пойдем методом от противного - попробуем избавиться от множественного наследования, и представим, что было бы.

1. Представим такую ситуацию: выстраиваем вертикальную иерархию, класс C наследуется от класса B, а B наследуется от класса A. В этом случае, в класс A мы должны вынести много того, что к понятию класса A не относится. Не совсем логично.



В случае со множественным наследованием (рис. ниже), мы четко разделяем понятия A и B. Такой подход уменьшает иерархию.

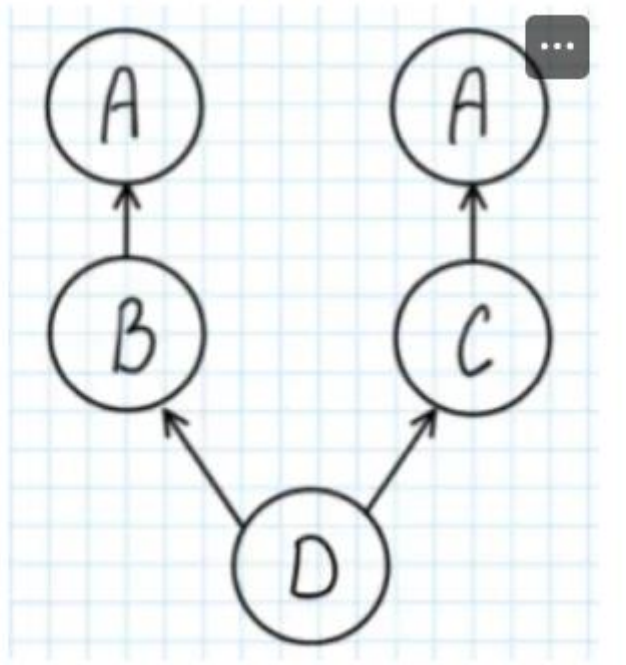


2. Второй момент (опять не используем множественное наследование). Мы можем не выносить что-то в базовый класс, а один из классов включить как подобъект, то есть не использовать наследование. Пусть С - производная от класса А и включает подобъект класса В. Тоже возникнет проблема - не будем иметь доступа к защищенным полям класса Б (нам придется делать это через методы класса Б) + придется протаскивать интерфейс класса Б для класса С.

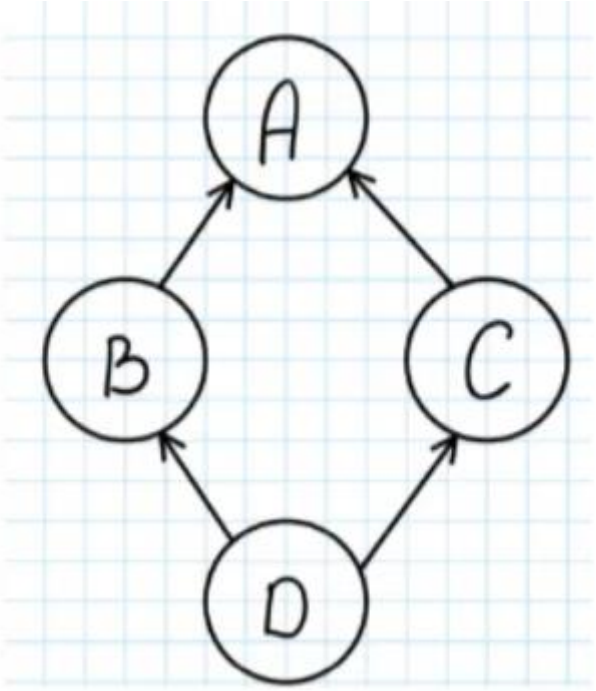
Прямая база - класс, от которого мы наследуемся. При наследовании может входить только 1 раз.

Косвенная база - прямая база прямой базы. При наследовании - сколько угодно раз.

Может возникнуть ситуация, когда в наш класс косвенная база входит два раза:



Надо свести всё к такой схеме наследования:



В большинстве случаев нам необходимо, чтобы базовый класс входил в производный только один раз. Рассмотрим пример ниже, где базовый класс войдёт в производный 2 раза:

- Пример

```
class A
{
public:
    A (char* s) { cout << "Creature A" << s << ";" << endl; }
};

class B : public A
{
public:
    B() : A (" from B") { cout << "Creature B;" << endl; }
};

class C : public B, public A // В класс C подобъект класса A будет
{                            // входить два раза.
public:
    C() : A(" from C") { cout << "Creature C;" << endl; }
};

void main()
{
    C obj;
}
```

Вызовется конструктор класса C. Из C вызовется конструктор класса B (так как класс B наследуется раньше класса A). Вызовется конструктор класса A. Создастся объект класса A. Создастся объект класса B. Из C вызовется конструктор класса A. Создастся объект класса A. Создастся объект класса C.

- На экран в результате работы программы будет выведено следующее:
- `Creature A from B;`
- `Creature B;`
- `Creature A from C;`
- `Creature C;`

Проблема решается с помощью виртуального наследования.

Виртуальное наследование

В большинстве случаев необходимо, чтобы базовый класс входил в производный только один раз. Для этого используется виртуальное наследование.

При виртуальном наследовании меняется порядок создания объекта: если в списке наследования есть виртуальное наследование (виртуальные базы), они отрабатывают в первую очередь слева направо, а потом всё остальные базы.

Чтобы сделать родительский (базовый) класс общим, используется ключевое слово `virtual` в строке объявления дочернего класса.

Пример. Базовый класс входит в производный один раз

Исправим предыдущий пример, добавив виртуальное наследование. Теперь, когда для класса C будет создаваться объект класса A, будет включаться механизм виртуального наследования. Когда будет создаваться подобъект класса C, для него не будет создан объект класса A.

- тык
- `class A`
- `{`
- `public:`
- `    A(char* s) { cout << "Creature A" << s << ";" << endl; }`
- `};`
-
- `class B : virtual public A`
- `{`
- `public:`
- `    B() : A(" from B") { cout << "Creature B;" << endl; }`
- `};`
-
- `class C : public B, virtual public A`
- `{`
- `public:`
- `    C() : A(" from C") { cout << "Creature C;" << endl; }`
- `};`
-
- `int main()`
- `{`

- `    C obj;`
- `}`
- На экран в результате работы программы будет выведено следующее:
- `Creature A from C;`
- `Creature B;`
- `Creature C;`

Порядок создания

- конструкторы виртуальных базовых классов выполняются до конструкторов не виртуальных базовых классов, независимо от того, как эти классы заданы в списке порождения;
- если класс имеет несколько виртуальных базовых классов, то конструкторы этих классов вызываются в порядке объявления виртуальных базовых классов в списке порождения;

Проблемы виртуального наследования

Рассмотрим следующий пример:

- Пример
- `class A {};`
- `class B : virtual public A{}; // Здесь virtual наследование`
- `class C : public A {};        // Здесь не virtual наследование`
- `class D : public B, public C{};`

Порядок создания объекта класса D: сначала вызывается конструктор класса B. Для него вызывается конструктор класса A, будет выполняться механизм виртуальности. Создастся подобъект класса A, отработает конструктор класса B. Для C уже не будет выполняться A.

Но если поменять порядок наследования для класса D:

- тык
- `class D : public C, public B{};`

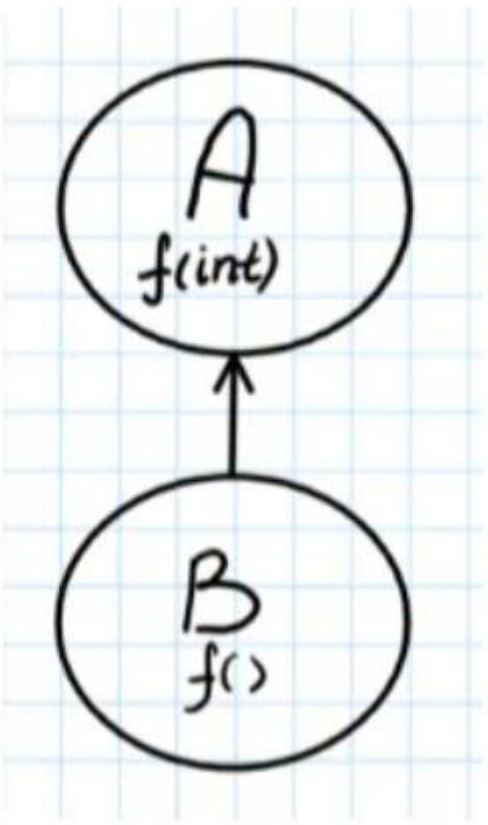
Смена последовательности наследование приводит к тому, что класс A будет включен два раза, что не должно происходить при включении механизма виртуальности.

ВАЖНО! Используя множественное наследование, надо стараться виртуально наследоваться по всем ветвям, чтобы не зависеть от порядка наследования.

Ниже представлена правильная версия:

- Пример
- `class A {};`
- `class B : virtual public A{};   // Здесь virtual наследование`
- `class C : virtual public A {};  // Тут теперь тоже virtual наследование`
- `class D : public B, public C{};`

Доминирование



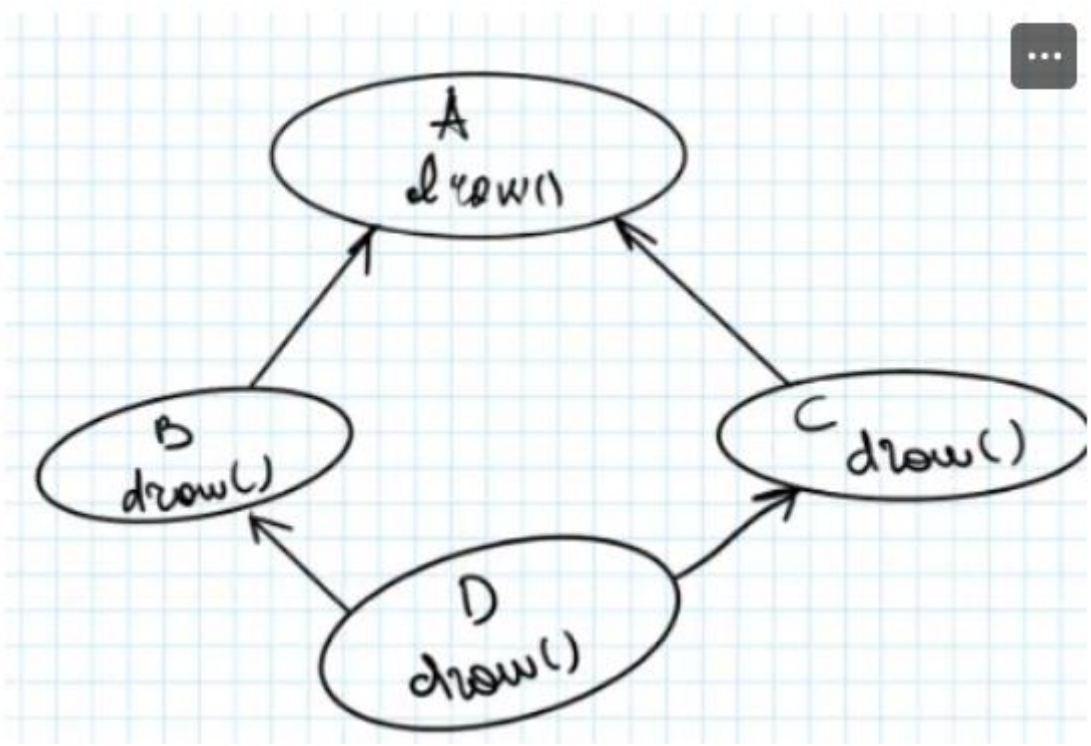
При одинаковых именах, доминировать будет метод производного класса (B) над базовым (A)

- тык
- `B obj;`
- `obj.f();  // вызовется этот`
- `obj.f(1); // error, тк метод базового класса подменили методом`
- `           // производного`

Проблемы, возникающие с множественным наследованием

1. Множественный вызов методов

   Рассмотрим следующую схему:



Предположим, есть объект класса A с методом `draw()`, который умеет себя нарисовать. Производные от него классы - B и C, тоже имеют метод `draw()` и тоже могут себя нарисовать, а так же они могут нарисовать подобъект базового класса. То есть, при рисовании объектов класса B (аналогично для C) вызывается метод `draw()` класса A.

Когда мы создаем объект класса D, в котором мы должны отрисовать объект класса B и объект класса C, `draw()` класса A вызывается два раза. Это называется проблема множественного вызова базового класса.

- Пример. Множественный вызов методов

```
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;
    }
};

void main()
{
    D obj;
    obj.f();
}
```

Метод `f()` класса A срабатывает дважды.

- Программа выведет на экран:

```
Executing f from A;
Executing f from C;
Executing f from A;
Executing f from B;
Executing f from D;
```

Красивых способов борьбы с этой проблемой, к сожалению, нет.

Пример. Решение проблемы множественного вызова методов

Идея: разделить метод на две части. Часть, которая относится непосредственно к самому классу, и часть, которая относится ко всему объекту.

В классе A мы разделили `f()` на две части, причем, то что относится к самому классу A - его собственное, мы делаем его защищённым, доступным только для методов класса A и производных классов. В производных классах мы тоже разделяем - метод, относящийся непосредственно к самому классу и ко всему объекту.

- тык

```
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();
    }
};

void main()
{
    D obj;

    obj.f();
}
```

Программа выведет на экран:

```
Executing f from A;
Executing f from C;
Executing f from B;
Executing f from D;
```

Решение не самое красивое, но других, к сожалению, нет. В современных ЯП избавились от множественного наследования. И нам в C++ желательно отказаться, хотя оно иногда помогает реализовать некоторые паттерны.

2. Неоднозначность при множественном наследовании.

тык

```
class A
{
public:
    int a;
    int (*b)(); // Если что, это указатель на функцию :)
    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); // Только для тех методов, которые идут по одной ветви - всё корректно.
    }
};

void main()
{
    C obj;

    D::fun(obj);
}
```

Есть два класса – класс A и класс B. Класс C – производная от классов A и B. В классе C мы получаем доступ к членам объекта класса C. Здесь играет следующее правило проверки на неоднозначность: проверка на неоднозначность происходит до проверки на перегрузку, на тип и до проверки на уровень доступа.

Решение проблемы (тоже некрасивое решение): Объединяем два класса, и в производном классе полностью подменяем то, что находится в базовых классах. Получается громоздкий производный класс (ведь мы всё, что относится к базовым классам - подменяем). Когда мы объединяем два класса одним классом, есть еще один недостаток.(см. пример ниже).

Неоднозначности при множественном наследовании

Также есть еще один недостаток – когда мы программируем, для объекта какого-либо класса мы должны выделить цель - для чего мы создаем это понятие. Объекты данного класса должны выполнять определенную задачу, и она должна быть только одна.

Правило: У одного объекта не должно быть несколько обязанностей. Когда мы объединяем два понятия, это чаще всего приводит к тому, что новое сформированное

понятие имеет несколько обязанностей. Такого быть не должно. Если один объект выступает в разных ролях, мы не должны объединять интерфейс, должны его разносить.

Исключением является ситуация, когда мы объединяем два разных понятия, формируя интерфейс одной обязанности. В этом случае используется следующая схема. У нас есть два класса, и мы формируем новое понятие, используя интерфейс только одного класса. В данном случае идёт наследование только по схеме `public` только от класса B, от класса A по схеме private. Таким образом, для объектов класса C интерфейс класса A невидим.

- тык
- ```
  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();
      }
  };

  void main()
  {
      C obj;
      D d;

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

Но здесь здесь есть проблема – проверка на неоднозначность происходит до проверки на схему наследования. Поэтому метод `f()` для объектов класса C мы вызвать не сможем - это неоднозначность, хотя наследуем по разной схеме.
  ```

Что нужно сделать: Нужно в классе C подменить те методы, которые идут по ветви `public`.

При такой схеме (когда в одном случае мы поддерживаем только один интерфейс) множественное наследование можно использовать.

ВАЖНО! Если нет общей базы (общая база задает интерфейсные методы для производных классов), то подмены должны осуществляться только по одной ветке.

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.

ОТВЕТ: Полиморфизм в C++. Виртуальные методы. Виртуальные деструкторы. Чисто виртуальные методы. Понятие абстрактного класса. Ошибки возникающие при работе с указателем на базовый класс. Дружественные связи.

Полиморфизм - использование объектов с одинаковым интерфейсом без информации о типе и внутренней структуре объекта. (“Один интерфейс, множество реализаций.”)

По сути полиморфизм ~~ безразличие (не важно, кто предоставляет функционал)

Виртуальные методы

Идея виртуальных методов - когда создаются объекты, создавать в памяти специальные таблицы (виртуальные), которые для данного объекта содержат адреса методов, которые надо вызывать. Соответственно объект должен иметь указатель на эту таблицу. Всегда это использовать не удобно.

Минусы:

1. увеличивается объём памяти. (В памяти хранятся таблицы для каждого объекта).
2. Увеличивается время вызова метода. (В таблице нужно найти метод, получить его адрес и по нему вызвать метод).

Плюсы:

1. Лёгкость подмены одного понятия на другое

Полиморфный класс - класс, в котором есть хотя бы один виртуальный метод (тогда в объекты этого класса добавляется указатель на таблицу виртуальных методов).

Правило полиморфных классов При наследовании нельзя ни сужать, ни расширять интерфейс базового класса.

Чисто виртуальный метод - метод, не имеющий тела.

- Пример
- `virtual Product CreateProduct() = 0;`

Абстрактный класс - класс, имеющий хотя бы один чисто виртуальный метод. Объекты абстрактного класса создавать нельзя.

Если производные классы не будут подменять чисто виртуальные методы, они будут тоже абстрактными.

Для абстрактного класса все методы, которые могут быть подменены в производном, определяем с модификатором `virtual`.

Для полиморфного класса мы всегда должны определять деструктор.

- Пример

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

Тут возникает проблема: объект уничтожается в обратном порядке (относительно порядка создания). Поэтому реализовать этот деструктор мы обязаны. (звучит, как костыль)

- Пример

```
A::~A() = default; // хотя бы так
```

Проблемы с виртуальными методами.

Правило: мы не должны вызывать виртуальные методы в конструкторах и деструкторах.

- Пример

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

    virtual void f()
    {
        cout << "Executing f from A" << endl;
    }
};

class B : public A
{
public:
    B()
    {
        // прикол в том, что класса C ещё нет
        // поэтому вызовется метод класса A
        this->f();
    }
    virtual ~B()
    {
        cout<<"Class B destructor called;"<<endl;
        // прикол в том, что класса C уже нет
        // поэтому вызовется метод класса A
        this->f();
    }

    void g() { this->f(); }

```

```
};

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

    virtual void f() override
    {
        cout<<"Executing f from C;"<<endl;
    }
};

void main()
{
    C obj;
    // this в методе B::g() будет = obj;
    // т.е отработает штатно
    obj.g();
}
```

Желательно, чтобы виртуальным был только интерфейс.

Неявное преобразование

ВАЖНО! В языке C++ происходит неявное преобразование от указателя объекта производного класса к указателю на объект базового класса. То же самое касается ссылок.

- Пример

```
A *p = new B;   // Неявное преобразование (пример с указателями)
B obj;
A& alias = obj; // Неявное преобразование (пример со ссылкой)
...
delete p;
```

Класс A – абстрактный, класс B – не абстрактный. Мы можем работать с классом B, вызывая метод.

Вступает правило: для класса A мы все методы, которые могут быть подменены в классе B, должны определить с модификатором `virtual`, чтобы один объект можно было подменить другим.

Мы работаем с указателем на A. Мы подменили один объект другим, но правую часть мы вызываем конкретно. Напрашивается виртуальный конструктор, но конструктор - это не метод объекта, а метод класса, конструктор не может быть виртуальным. Получается проблема с подменой (спойлер: решается с помощью порождающих паттернов).

Ошибки возникающие при работе с указателем на базовый класс

Возникает еще одна проблема – вызывается деструктор, а деструктор должен вызываться для объекта класса B. Но деструктор вызывается для объекта B, поэтому деструктор может быть виртуальным. Соответственно, когда мы определяем какой-либо класс, в любом случае для базового полиморфного класса мы должны определить деструктор. Если мы не можем его определить, то мы делаем деструктор чисто виртуальным.

- Пример

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

Но возникает проблема – у нас создается объект какого-то производного класса, по цепочке отрабатывает конструктор, в обратном порядке отрабатывают деструкторы. А мы удалили этот деструктор! Поэтому реализовать этот деструктор мы обязаны. Реализовать как пустой.

- Пример

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

Задав виртуальный деструктор, каждый производный класс определяет для себя этот деструктор.

Итоги: если у нас полиморфный базовый класс, мы должны подменяемые методы определить, как виртуальные, а так же должны объявить и определить виртуальный деструктор. В дальнейшем мы будем говорить, что базовые классы ВСЕГДА должны быть абстрактными. Как раз механизм с виртуальным деструктором дает нам возможность формировать такой базовый класс. Если мы определили чисто виртуальный деструктор в классе, то этот класс тоже является абстрактным, хотя, казалось бы, есть его реализация.

- Пример. Виртуальный деструктор

```
class A // Абстрактый класс, несмотря на наличие реализации деструктора
{
public:
    virtual ~A() = 0; // Чисто виртуальный деструктор

};

A::~A() {} // Реализация деструктора

class B : public A
{
public:
```

```
    virtual ~B() { cout<<"Class B destructor called;"<<endl; }
};

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

Дружба

Дружба - это плохо. Дружба даёт доступ ко всем членам класса методов других классов. В классе для объектов указываем, что есть «друг». Схема получается очень зависимая.

Она приводит к тому, что если нужно вносить изменения написанный код (фу позор!!!).

Необходимо, чтобы дружественных отношений было как можно меньше.

- Пример из лекции

```
class C
{
friend void f(C& ac); // друг-функция
friend A::f();        // друг-метод
friend B;             // друг-класс (самый плохой вариант)
};
```

Свойства дружбы:

1. Дружба не наследуется (сын друга - не друг)
    - тык

```
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
};
```

2. Дружба не транзитивна (друг моего друга мне не друг).

   Есть исключение: при полиморфизме мы получаем доступ к защищённым полям производных классов.

```
тык
class C; // forward-объявление. C - это класс

class A // полиморфный, так как есть виртуальный метод
{
    protected:
        virtual void f();
        friend class C;
};

class B: public A
{
    protected:
        virtual void f() override;
};

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

B obj;

C::g(obj);
```

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.**

ОТВЕТ: *Обработка исключительных ситуаций в C++. Проблемы с динамической памятью при обработке исключительных ситуаций.

Обработка исключительных ситуаций в C++.

Главный недостаток структурного программирования – если на каком-то нижнем уровне возникает ошибка, её необходимо протащить наверх, где её обработают спустя N уровней абстракции. Обработка ошибки совмещена вместе с логикой программы.

Идея - ПРОКИДЫВАТЬ ошибку СРАЗУ туда, где её можно обработать.

- Пример
- `try`
- `{`
- `  // код, который может упасть`
- `  // где-то произошёл throw <объект>`
- `}`
- `catch (<тип1>& <объект>) // если объект приводится к тип1, то обработает этот обработчик`
- `{`
- `}`
- `catch (<тип2>& <объект>) // если предыдущие обработчики не могут обработать,`
- `{                          // то передаётся исключение след. обработчику`
- `}`
- `catch (...) // перехватит любую исключительную ситуацию (порядок обработчиков важен)`
- `{}`

Если ошибка никем не перехватилась, то программа падает.

Задачи обработчика:

1. Выдать сообщение (пользователю или в лог)
    1. Информация об ошибке (в лог файл пишем)
    2. Время, когда она произошла
    3. Где произошла
    4. Данные, которые привели к этой ошибке
2. Обработать ситуацию (по возможности)
3. Если ошибка критическая - корректно завершить программу (например, корректно закрыть связь с бд)

Плюсы:

- Не протаскиваем ошибку

- Всю обработку исключительной ситуации вынесли в одно место (отделили от логики задачи)
- Можно легко развивать ПО и модифицировать

Проблемы с динамической памятью при обработке исключительных ситуаций:

- Страшный код:
- `try`
- `{`
- `A *pobj = new A;`
- `pobj->f();`
- `delete pobj;`
- `}`
- `catch(...)`
- `{}`

в методе f была выброшена ошибка.

Что происходит:

1. Возникает искл ситуация
2. Удаляются все временные объекты, которые создавались в блоке `try`
3. Выходим на обработчик

Возникает утечка дин памяти.

Бросать исключение из деструкторов не стоит. Непредсказуемый результат. Если же в нём возникает исключение, то в нём мы должны её обработать.

Решение проблемы с помощью `noexcept` или `throw()`:

Можно запретить методу обрабатывать исключительную ситуацию.

Два варианта решения проблемы:

- тык
- `void A::f() noexcept // Первый способ`
- `void A::f() throw()  // Второй способ`
- С **`noexcept`** при возникновении исключительной ситуации вызывается функция `terminate()`. Функция `terminate()` приводит к тому, что будут вызываться все деструкторы только временных объектов в порядке, обратном их созданию.
- С **`throw()`** результат непредсказуем, это старый синтаксис, который лучше не использовать.

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.

Определение исключительных ситуаций, которые метод может обрабатывать

Можно указать, какую исключительную ситуацию метод может обрабатывать.

- `void A::f() throw(<тип>)`

Этот метод может обрабатывать все исключительные ситуации с этим типом и производными от него.

Ловля ошибки в разделе инициализации.

- тык
- `Array::Array(int q) try: mas(new double[q]), cnt(q)`
- `{}`
- `catch(const std::bad_alloc& exc)`
- `{ cout<<exc.what()<<endl; }`

Идея exception:

У нас есть базовый класс `std::exception`. Этот базовый класс нам представляет виртуальный метод `what()`, возвращающий строку `message`. Стандартные ошибки являются производными от этого класса. Производные классы могут подменять метод `what()`.

Например, есть стандартная ошибка `bad_alloc` - ошибка, связанная с выделением памяти.

И да, мы свои классы можем тоже порождать от этого базового класса! Пусть у нас есть класс Array, мы для него хотим создать объекты, которые отвечают за определенные ситуации. Назовём класс *ErrorArray*. От него уже будем порождать конкретные ошибки: некорректный индекс - *ErrorIndex, ErrorAlloc* (перехватываем `bad_alloc` на себя).

Любую ошибку с нашим классом мы можем перехватить. На уровне класса exp мы можем перехватить любую ошибку, связанную с нашим массивом.

Для нашего ПО таких уровней перехвата ошибок может быть много. Удобно модифицировать. Модифицируя наш класс, мы добавим новую ошибку, но она будет перехватываться на уровне базового класса, отвечающего за ошибки, связанные с объектом нашего класса.

- Пример кода
- `class ExceptionArray : public std::exception // Базовый класс для отлова ошибок, связанных с классом Array`
- `{`
- `protected:`
- `  char* errormsg;`
- `public:`
- `  ExceptionArray(const char* msg) // В конструкторе получаем строку сообщения`
- `  {`
- `  int Len = strlen(msg) + 1;  // Определяем длину`
- `  this->errormsg = new char[Len];  // Выделяем память`
- `  strcpy_s(this->errormsg, Len, msg); // Копируем в нашу строку`
- `  }`
- `  virtual ~ExceptionArray() { delete[]errormsg; }`

```
  virtual const char* what() const noexcept override { return this->errormsg; } // Подменяем метод what
};

class ErrorIndex : public ExceptionArray // Уже конкретная ошибка, связанная с классом Array
{
private:
  const char* errIndexMsg = "Error Index";
  int ind;
public:
  ErrorIndex(const char* msg, int index) : ExceptionArray(msg), ind(index) {}
  virtual ~ErrorIndex() {}

    // Здесь идет просто формирование строки, которая выдает описание ошибки.
  virtual const char* what() const noexcept override
  {
          int Len = strlen(errormsg) + strlen(errIndexMsg) + 8;
          char* buff = new char[Len + 1];
          sprintf_s(buff, Len, "%s %s: %4d", errormsg, errIndexMsg, ind);
          char* temp = errormsg;
          delete[]temp;
          const_cast<ErrorIndex*>(this)->errormsg = buff;
          return errormsg;
  }
};

int main()
{
  try
  {
          throw(ErrorIndex("Index!!", -1)); // Генерим исключительную ситуацию, создаём объект
  }                                         // и отлавливаем объект ниже
  catch (ExceptionArray& error)
  {
          cout << error.what() << endl;
  }
  catch (std::exception& error)
  {
          cout << error.what() << endl;
  }
  catch (...)
  {
  }

  return 0;
}
```

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

ОТВЕТ: *. Перегрузка операторов в C++. Правила перегрузки операторов. Операторы ., ->. Перегрузка унарных и бинарных операторов. Перегрузка оператора = : копирование, перенос. Перегрузка операторов ->, , []. Перегрузка операторов ++, --. Операторы приведения типов.

Перегрузка операторов в C++.

Шесть операторов, которые перегружать нельзя

1. `.` - доступ к члену объекта
2. `.*` - см ниже
3. `::` - доступ к контексту
4. `?` - тернарный оператор (не смогли придумать, как перегрузить)
5. `sizeof` - размер объекта
6. `typeid` - id типа объекта

Указатель на метод:

- `auto pf = &A::f();`
- `A obj;`
- `(obj.*pf)(); // .* имеет ниже приоритет чем ()`

`->*` аналогичен `.*`, только работаем с указателем на объект.

Правила перегрузки операторов:

1. `=`, `()`, `[]`, `->`, `->*` перегружаются только как члены класса
2. Бинарные операторы можно перегружать или как члены класса (более предпочтительно), или как внешние функции
3. Унарные перегружаем как члены класса.
4. `&&`, `||`, `,` , `&` особенность операторов в том, что второй операнд может не вычисляться, но при перегрузке он будет вычислен обязательно

Перегрузка оператора =: копирование, перенос:

- тык
- `Array& Array::operator=(const Array& arr)`
- `{`
- `  if( this == &arr ) return *this;`
- `  if ( this->count != arr.count)`
- `  {`
- `    delete []this->mas;`
- `    this->count = arr.count;`
- `    this->mass = new double[this->count];`
- `  }`
- `  memcpy(this->mass, arr.mass, this->count * sizeof(double)); // плохая функция, но лень цикл писать`
- `  return *this;`

```
}

Array& Array::operator=(Array&& arr)
{
  if( this == &arr ) return *this;
  if ( this->count != arr.count)
  {
    delete []this->mas;
    this->count = arr.count;
  }
  this->mass = arr.mas;
  arr.mas = nullptr;
  return *this;
}
```

Перегрузка оператора ->*. Функтор.

ТЫК

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

class Caller {
public:
   typedef int (Callee::*FnPtr)(int);  // указан тип
private:
   Callee* pobj;
   FnPtr ptr;

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

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);
   }  // принимающий указатель на метод
};

void main() {
   Caller::FnPtr pn = &Callee::inc;
   Pointer pt(1);
   cout << "Result: " << (pt->*pn)(2)
        << endl;  // (pt.operator->*(pn)).operator()(2)
```

- }

Перегрузка операторов -> и *.

- тык
- 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->() { return pobj; }
- const A* operator->() const { return pobj; }
- A& operator*() { return *pobj; }
- const A& operator*() const { return *pobj; }
- };
-
- void main()
- {
- A a;
-
- B b1(&a);
- b1->f();
-
- const B b2(&a);
- (*b2).f();
- }

Перегрузка оператора []:

- тык
- double& Array::operator[](const Index& index)
- {
- if (index < 0 || index >= cnt) throw std::out_of_range("Error: class Array operator [];");
- return mas[index];
- }

На всякий случай (дополнительно): перегрузка оператора ,:

При перегрузке оператора , к вниманию принимается последний справа аргумент. Все другие аргументы игнорируются.

- class A:
- {
- private:

```
    int a;
public:
    A(int i) { this->a = i; }
    int get() { return this->a; }

    A operator,(A &other)
    {
            A tmp(other.get());
            return tmp;
    }
};

int main()
{
    A a(1), b(2);
    A c = (a, b);
    std::cout << c.get() << std::endl; // Выведется 2

    return 0;
}
```

Операторы инкремент (++) и декремент (--) и оператор приведения типа

Идея: отделить постфиксную от префиксной записи.

Решение: унарный - префиксный, бинарный - постфиксный операторы.

Замечание: если мы бинарный не перегружаем, то и для постфиксного, и для префиксного будет вызываться перегруженный унарный оператор. Иначе мы четко разделяем их.

- тык
- ```
  class Index {
  private:
      int ind;

  public:
      Index(int i = 0) : ind(i) {}

      Index& operator++()  // ++obj
      {
          ++ind;
          return *this;
      }

      Index operator++(int)  // obj++
      {
          Index it(*this);
          ++ind;
          return it;
      }

      // оператор приведения типа, может неявно приводится к int
      operator int() const { return ind; }
  };
  ```

С минусом все аналогично.
  ```

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

Перегрузка унарных и бинарных операторов в C++ предоставляет возможность изменить стандартное поведение операторов для работы с объектами пользовательских типов. Это позволяет объектам вести себя похоже на встроенные типы данных. Вот подробное объяснение различных аспектов перегрузки операторов:

Перегрузка унарных операторов

Унарные операторы, такие как ++ (инкремент), -- (декремент), и ! (логическое отрицание), работают только с одним операндом. Перегрузка этих операторов позволяет изменить их поведение для классов или структур. Например, перегрузка оператора ++ может быть использована для увеличения значений некоторых внутренних полей объекта.

Пример:

```
class Counter {
public:
    int value;
    Counter(int value) : value(value) {}
    // Префиксный инкремент
    Counter& operator++() {
        ++value;
        return *this;
    }
    // Постфиксный инкремент
    Counter operator++(int) {
        Counter temp = *this;
        ++*this;
        return temp;
    }
};
```

Перегрузка бинарных операторов

Бинарные операторы, такие как +, -, *, и /, требуют двух операндов. Перегрузка этих операторов позволяет определить правила их взаимодействия между двумя объектами класса.

Примеры:

```
class Point {
public:
    int x, y;
    Point(int x, int y) : x(x), y(y) {}
    Point operator+(const Point& other) {
```

```
        return Point(x + other.x, y + other.y);
    }
};
```

Проблемы с перегрузкой операторов &&, ||, „ &

- **&& и ||: При перегрузке этих операторов теряется возможность "short-circuiting", то есть ленивое вычисление, когда второй операнд не вычисляется, если результат уже ясен из первого операнда.**
- **, (запятая): Этот оператор часто используется для создания последовательности операций, и его перегрузка может привести к неожиданному и непрозрачному поведению.**
- **& (адрес): Перегрузка этого оператора может сделать код запутанным, поскольку он изменяет стандартное поведение получения адреса переменной.**

Перегрузка операторов приведения типов

Операторы приведения типов позволяют объекту одного типа преобразовываться в другой тип. Это делается для обеспечения совместимости объекта с различными типами данных или API.

```
class Fraction {
public:
    int numerator, denominator;
    Fraction(int num, int den) : numerator(num), denominator(den) {}
    // Приведение к типу double
    operator double() const {
        return (double)numerator / denominator;
    }
};
```

Тривалентный оператор spaceship (<=>)

Оператор <=>, введенный в C++20, известен как оператор "spaceship", потому что его символ похож на космический корабль. Этот оператор возвращает стандартное упорядочение (меньше, равно, больше) между двумя объектами и используется для упрощения перегрузки операторов сравнения.

```
class MyClass {
public:
    int value;
    auto operator<=>(const MyClass&) const = default;  // Сгенерировать
автоматически
};
```

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.

ОТВЕТ: Шаблоны функций и классов в C++. Параметры шаблонов. Специализация шаблонов частичная и полная. Параметры шаблона задаваемые по умолчанию.

Шаблоны в C++

Шаблоны в C++ представляют собой мощный инструмент для создания обобщенных функций, методов классов и классов, что позволяет программистам использовать функции и классы с любыми типами данных. Это способствует повторному использованию кода и увеличивает его гибкость и масштабируемость.

Основные понятия шаблонов

1. **Шаблоны функций и методов классов**: Шаблоны позволяют функциям и методам работать с различными типами данных без переписывания кода для каждого типа. Например, функция для нахождения максимального значения из двух может быть написана как шаблон, чтобы работать с любым типом данных.

2. **Шаблоны классов**: Аналогично функциям, классы можно параметризировать типами данных, позволяя создавать экземпляры класса для различных типов. Например, класс std::vector в стандартной библиотеке C++ является шаблоном, который можно использовать для создания списка любого типа данных.

Параметры шаблонов

- **Параметры типа**: Обычно указываются с ключевым словом typename или class, и они определяют типы, которые будут использоваться в шаблоне.

- **Параметры значения**: Шаблоны могут также принимать параметры-значения, например, целые числа, которые могут определять размер массива или другие статические значения.

Специализация шаблонов

- **Полная специализация**: Определяет поведение шаблона для конкретного типа.

- **Частичная специализация**: Позволяет специализировать часть параметров шаблона, оставляя другие параметры обобщенными.

Недостатки шаблонов

1. **Сложность отладки**: Сообщения об ошибках, связанных с шаблонами, могут быть трудными для понимания.

2. **Раздувание кода**: Каждая инстанциация шаблона с новым типом данных приводит к генерации нового блока кода, что может увеличить размер исполняемого файла.

3. **Увеличение времени компиляции**: Шаблоны увеличивают время компиляции, так как требуют более сложной обработки на этапе компиляции.

```
template <typename T>
T max(T x, T y) {
    return x > y ? x : y;
}

int main() {
    // Использование функции max для int
    std::cout << max<int>(3, 7) << std::endl;  // Вывод: 7

    // Использование функции max для double
    std::cout << max<double>(3.5, 2.5) << std::endl;  // Вывод: 3.5

    // Автоматическое вывод типов
    std::cout << max(3.5, 2.5) << std::endl;  // Вывод: 3.5
}
```

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

Неявные шаблоны

Неявные шаблоны подразумевают, что компилятор автоматически выводит параметры шаблона на основе типов переданных аргументов. Это удобно, так как позволяет избежать явного указания типов при вызове шаблонных функций или методов.

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

int main() {
    print(42);          // Компилятор автоматически выводит тип T как int
    print("Hello");     // Компилятор выводит тип T как const char*
}
```

Компилятор анализирует тип переданного аргумента и автоматически подставляет соответствующий тип в шаблон.

Протаскивание типа через шаблон (std::forward)

std::forward используется для сохранения категории значения (lvalue или rvalue) при передаче аргументов в шаблоны. Это особенно важно для универсальных ссылок, чтобы избежать лишнего копирования объектов.

Пример:

```
#include <iostream>
#include <utility>

template <typename T>
void process(T&& arg) {
    doSomething(std::forward<T>(arg));  // Сохраняем категорию значения
}

void doSomething(const int& x) { std::cout << "Lvalue: " << x << std::endl; }
void doSomething(int&& x) { std::cout << "Rvalue: " << x << std::endl; }

int main() {
    int a = 10;
    process(a);          // Передаём lvalue
    process(20);         // Передаём rvalue
}
```

Как работает std::forward:

- **Если аргумент является lvalue, то std::forward передаст его как lvalue.**
- **Если аргумент является rvalue, то он передаст его как rvalue.**

Определение типа с помощью decltype

decltype используется для определения типа выражения во время компиляции. Это особенно полезно при работе с шаблонами, где типы могут быть сложными для определения вручную.

Пример:

```
int a = 5;
decltype(a) b = 10;  // Тип b будет int
```

decltype(auto)

decltype(auto) позволяет использовать тип выражения в месте объявления переменной. Это полезно, если тип выражения сложен или включает ссылки.

```
int x = 42;
decltype(auto) y = (x);  // y будет int&
decltype(auto) z = x;    // z будет int
```

Специализация шаблонов функций

Специализация шаблонов позволяет определить специфическое поведение функции для конкретного типа данных.

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

// Полная специализация для типа int
```

```cpp
template <>
void print<int>(int value) {
    std::cout << "Integer: " << value << std::endl;
}

int main() {
    print(42);          // Вызовет специализированную версию
    print(3.14);        // Вызовет общую версию
}
```

Обобщение

1. **Неявные шаблоны позволяют компилятору автоматически выводить типы.**
2. **std::forward используется для сохранения категории значения (универсальные ссылки).**
3. **decltype и decltype(auto)` помогают определить тип выражения, особенно в шаблонах.**
4. **Специализация шаблонов функций дает возможность настроить поведение для конкретных типов.**

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

ОТВЕТ: Шаблонный тип:

В языке С использовался `typedef`.

В C++ - `using`

- Синтаксис:
- `typedef void (*func)(int); // C`
- `using func = void (*)(int); // C++`

```
using ИмяТипа = АбстрактныйОписатель (определение переменной без её имени)
```

Для using можно определять шаблон:

- Код шаблонного `using`
- `template <typename Type>`
- `using func = int(*)(Type, Type)`

Специализация

Любую функцию можно перегружать или делать специализацию функции

Специализация - определение функции с конкретными значениями параметров

Полная - для всех параметров

Неполная - не для всех

Специализация тоже является шаблоном

- Специализация синтаксис:

```
// Шаблон функции
template <typename Type>
void swap(Type& val1, Type& val2) {}

// Полная специализация
template<> // Показываем, что функция - шаблон
void swap<float>(float& val1, float& val2) {}
```

Приоритет для компилятора (в порядке уменьшения):

1. Перегруженная функция
2. Специализации
3. Шаблон

- Пример 11. Правило вызова функций.

```
template <typename Type>
void swap(Type& val1, Type& val2)
{
  Type temp = val1; val1 = val2; val2 = temp;
}

template<>
void swap<float>(float& val1, float& val2)
{
  float temp = val1; val1 = val2; val2 = temp;
}

void swap(float& val1, float& val2)
{
  float temp = val1; val1 = val2; val2 = temp;
}

void swap(int& val1, int& val2)
{
  int temp = val1; val1 = val2; val2 = temp;
}

void main()
{
  const int N = 2;
  int a1[N];
  float a2[N];
  double a3[N];

  swap(a1[0], a1[1]);     // swap(int&, int&)
  swap<int>(a1[0], a1[1]);   // swap<int>(int&, int&)
  swap(a2[0], a2[1]);     // swap(float&, float&)
  swap<float>(a2[0], a2[1]); // swap<>(float&, float&)
  swap(a3[0], a3[1]);     // swap<double>(double&, double&)
}
```

Определение типа возвращаемого значения шаблона: На вызов и создание по шаблону конкретной функции влияют только параметры, которые мы передаем. Возвращаемый тип можно определять по какому-либо выражению.

Ставится `auto` и `decltype`

- Пример 12. Определение типа возвращаемого значения для шаблона функции.
- `template <typename T, typename U>`
- `auto sum(const T& elem1, const U& elem2) -> decltype(elem1 + elem2)// decltype можно не писать вроде`
- `{`
- `  return elem1 + elem2;`
- `}`
-
- `int main()`
- `{`
- `  auto s = sum(1, 1.2); // будет тип double, так как int + double = double`
-
- `  cout << "Result: " << s << endl;`
- `}`

Классы

Методы шаблонного класса - также шаблоны. В обычном классе могут быть шаблонные методы

- тык
- `template <Type>`
- `Class A:`
- `{`
- `Type var;`
- `void f();`
- `}`
-
- `template <Type>`
- `void A<Type>::f()`
- `{`
- `}`
- Параметры значения:
- `template <typename Type, const char* string>`
- `Class A:`
- `{}`
-
-
- `const char* str = "asd";`
- `A<str> object;  // ошибка. нет внешнего связывания. константная строка может инициализироваться неконстантной`
-
- `extern char S[] = "asd";`
- `A<S> obj2; // ok, так как есть внешнее связывание`
-
- `template <int b>`
- `Class A:`
- `{}`

Под каждый параметр создастся отдельный класс, надо быть аккуратней, они не взаимозаменяемы.

Полная специализация

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

Частичная

- Код
- ```
  // Второй параметр - по умолчанию.
  // То есть если компилятор генерит класс по шаблону, а в параметры передаётся
  // только один параметр, второй будет приниматься за double.
  template <typename T1, typename T2 = double>
  class A {};
  
  // Частичная специализация, когда два параметра равны
  template <typename T>
  class A<T, T> {};
  
  // Частичная специализация, второй параметр - int
  template <typename T>
  class A<T, int> {};
  ```

Параметры шаблона задаваемые по умолчанию

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):

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

Шаблоны с переменным числом параметров.

Может быть заменен initializer_list. Нужно ограничивать, как и рекурсию (в примере - функция терминатор с одним параметром) Ну тут добавить нечего, вот пример, вот такой вот синтаксис.

- Пример 5. Шаблон функции с переменным числом параметров.
- ```
  template <typename Type>
  ```

```
Type sum(Type value)
{
  return value;
}

template <typename Type, typename ...Args> // сначала три точки при объявлении template
Type sum(Type value, Args... args)         // три точки после типа при использовании
{
  return value + sum(args...);
}

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

  return 0;
}
```

Пространства имен

- Пример кода

```
namespace myNamespace
{
   class A {...};
}

// Доступ извне в пространство имен происходит через доступ к контексту
myNamespace::A obj();

//Или включить всё пространство:
using namespace myNamespace;
A obj();
```

Имя пространства имен может быть опущено (анонимное пространство имен). Тогда к нему нельзя получить доступ извне

Можно углубляться, то есть `ns1::ns2::ns3`

Надо стараться избегать злоупотребления.

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

С появлением C++20 в язык были введены концепты и требования (requires), которые позволяют наложить ограничения на параметры шаблонов, делая их использование более безопасным и понятным. Это помогает избегать сложных и запутанных сообщений об ошибках компилятора, характерных для шаблонов в ранних версиях C++.

Ограничения на шаблоны

Ограничения на шаблоны — это условия, которым должны соответствовать типы, используемые в качестве аргументов шаблонов. Ограничения помогают компилятору определить, какие типы допустимы, и предотвращают ошибки на этапе компиляции.

Проблемы до C++20:

- **Компилятор выдавал сложные для понимания сообщения об ошибках, если шаблон использовался с неподходящими типами.**
- **Ограничения приходилось проверять вручную, что увеличивало количество кода и сложность.**

Концепты и requires

Концепты — это способ наложения ограничений на параметры шаблонов. Они позволяют явно указать, какие свойства и операции должны поддерживать типы, передаваемые в шаблон.

requires — ключевое слово, используемое для определения условий, которым должен удовлетворять тип. Это позволяет писать более выразительные и читаемые шаблоны.

```
#include <concepts>
#include <iostream>

template <typename T>
requires std::integral<T>  // Ограничение: T должен быть целым числом
T add(T a, T b) {
    return a + b;
}

int main() {
```

```
    std::cout << add(3, 5) << std::endl;  // OK
    // std::cout << add(3.5, 5.5) << std::endl;  // Ошибка: тип double не
удовлетворяет std::integral
}
```

Типы ограничений

1. **Ограничения на основе концептов: Концепты представляют собой заранее определенные шаблоны ограничений, такие как:**
    - **std::integral — для целочисленных типов.**
    - **std::floating_point — для чисел с плавающей точкой.**
    - **std::same_as<T> — для проверки, что два типа одинаковы.**
2. **Пользовательские концепты: Разработчики могут создавать свои собственные концепты для описания специфических требований.**

```
3. template <typename T>
4. concept Addable = requires(T a, T b) {
5.     { a + b } -> std::same_as<T>;  // Требование: поддержка операции
   сложения
6. };
7.
8. template <Addable T>
9. T add(T a, T b) {
10.          return a + b;
11.      }
```

Преимущества концептов и ограничений

- **Улучшение читаемости кода: Концепты делают код более понятным, так как ограничения описываются явно.**
- **Улучшение сообщений об ошибках: Компилятор может сообщить, какое именно ограничение не выполнено, вместо генерации сложных сообщений.**
- **Повышение безопасности: Позволяет избегать некорректных операций на этапе компиляции.**

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.

Проблемы с динамическим выделением и освобождением памяти

Динамическое управление памятью в C++ сопряжено с рядом сложностей, включая:

1. **Утечки памяти: Если память, выделенная через new, не освобождается с помощью delete, это приводит к утечкам, особенно в сложных программах с большим количеством объектов.**
2. **Двойное освобождение памяти: Попытка дважды освободить одну и ту же память (delete для одного и того же указателя) вызывает неопределенное поведение.**
3. **Dangling указатели: Указатели, ссылающиеся на память, которая уже была освобождена, могут привести к ошибкам при их использовании.**
4. **Сложности в многопоточности: В условиях многопоточности ошибки управления памятью (например, одновременное освобождение одной памяти) могут быть еще более критичными.**

Шаблон Holder

Шаблон Holder — это простая структура, которая используется для управления ресурсами (например, динамически выделенной памятью). Он автоматически освобождает ресурс при уничтожении объекта Holder, предотвращая утечки памяти.

Пример простого Holder:

```
template <typename T>
class Holder {
private:
    T* ptr;
public:
    explicit Holder(T* p = nullptr) : ptr(p) {}
    ~Holder() { delete ptr; }

    T* get() const { return ptr; }
    T& operator*() const { return *ptr; }
    T* operator->() const { return ptr; }

    // Запрещаем копирование
    Holder(const Holder&) = delete;
    Holder& operator=(const Holder&) = delete;

    // Разрешаем перемещение
    Holder(Holder&& other) noexcept : ptr(other.ptr) { other.ptr = nullptr; }
    Holder& operator=(Holder&& other) noexcept {
        if (this != &other) {
            delete ptr;
            ptr = other.ptr;
            other.ptr = nullptr;
        }
        return *this;
    }
};
```

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

Проблемы с динамическим выделением и освобождением памяти

Динамическое управление памятью в C++ сопряжено с рядом сложностей, включая:

Утечки памяти: Если память, выделенная через new, не освобождается с помощью delete, это приводит к утечкам, особенно в сложных программах с большим количеством объектов.

Двойное освобождение памяти: Попытка дважды освободить одну и ту же память (delete для одного и того же указателя) вызывает неопределенное поведение.

Dangling указатели: Указатели, ссылающиеся на память, которая уже была освобождена, могут привести к ошибкам при их использовании.

Сложности в многопоточности: В условиях многопоточности ошибки управления памятью (например, одновременное освобождение одной памяти) могут быть еще более критичными.

Шаблон Holder

Шаблон Holder — это простая структура, которая используется для управления ресурсами (например, динамически выделенной памятью). Он автоматически освобождает ресурс при уничтожении объекта Holder, предотвращая утечки памяти.

Пример простого Holder:

```
template <typename T>
class Holder {
private:
    T* ptr;
```

```
public:
    explicit Holder(T* p = nullptr) : ptr(p) {}
    ~Holder() { delete ptr; }

    T* get() const { return ptr; }
    T& operator*() const { return *ptr; }
    T* operator->() const { return ptr; }

    // Запрещаем копирование
    Holder(const Holder&) = delete;
    Holder& operator=(const Holder&) = delete;

    // Разрешаем перемещение
    Holder(Holder&& other) noexcept : ptr(other.ptr) { other.ptr = nullptr; }
    Holder& operator=(Holder&& other) noexcept {
        if (this != &other) {
            delete ptr;
            ptr = other.ptr;
            other.ptr = nullptr;
        }
        return *this;
    }
};
```

«Умные указатели» в C++

Современный C++ предоставляет более мощные инструменты для управления динамической памятью в виде умных указателей, которые решают вышеуказанные проблемы.

unique_ptr:

Уникальный владелец объекта.

Обеспечивает автоматическое освобождение памяти, когда указатель выходит из области видимости.

Невозможно копировать, но можно перемещать.

```
std::unique_ptr<int> ptr = std::make_unique<int>(42);
```

shared_ptr:

Разделяет владение объектом между несколькими указателями.

Использует счетчик ссылок для управления временем жизни объекта. Объект уничтожается, когда счетчик ссылок достигает нуля.

```
std::shared_ptr<int> sp1 = std::make_shared<int>(42);
std::shared_ptr<int> sp2 = sp1;  // sp1 и sp2 разделяют владение
```

weak_ptr:

Создает слабую ссылку на объект, управляемый shared_ptr.

Не увеличивает счетчик ссылок, предотвращая циклические зависимости.

```
std::shared_ptr<int> sp = std::make_shared<int>(42);
std::weak_ptr<int> wp = sp;  // wp не влияет на счетчик ссылок
```

Связь между shared_ptr и weak_ptr

- **shared_ptr владеет объектом и управляет его временем жизни через счетчик ссылок.**

- **weak_ptr создается на основе shared_ptr и не влияет на счетчик ссылок, что предотвращает циклические зависимости.**
- **Пример предотвращения цикла:**

```
struct Node {
    std::shared_ptr<Node> next;
    std::weak_ptr<Node> prev;  // Используем weak_ptr, чтобы избежать цикла
};
```

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.

ОТВЕТ: Приведение типа в C++: static_cast, dynamic_cast, const_cast, reinterpret_cast. «Умные указатели» в C++: unique_ptr, shared_ptr, weak_ptr. Контейнерные классы и итераторы. Работа с итераторами. Цикл for для работы с контейнерными объектами.

Приведение типа в C++: static_cast, dynamic_cast, const_cast, reinterpret_cast.

В С использовалась конструкция (новыйТип) ДанноеСтарогоТипа. Такое приведение небезопасно. Компилятор делает не задумываясь, не предупреждая нас возможно ли или не возможно.

В C++ указатель(или ссылка) на производный класс по умолчанию приводится к указателю на базовый. Если нужно обратное преобразование - проблема. В связи с этим появляется два оператора преобразования.

Один выполняется на этапе компиляции (static_cast), второй на этапе выполнения (dynamic_cast).

Для использование всех этих указателей надо

```
#include <memory>
```

static_cast

static_cast используется для приведения родственных классов, находящихся на одной ветви. Так же используется для стандартных типов, для которых определен механизм явного приведения (но смысла нет).

- тык
- `Class Base;`
- `Class FirstBranch: Base;`
- `Class FirstBranchContinue: FirstBranch;`

-
- `class SecondBranch: Base;`

В таком раскладе static_cast позволит привести друг к другу Base, FirstBranch, FirstBranchContinue и Base, SecondBranch

FirstBranch и SecondBranch хоть и родственные, но статик кастом не приводятся.

- Синтаксис:
- `Type* a = static_cast<Type*>(otherPtr)`
- Пример
- `Base* ptr = new FirstBranch;`
- `FirstBranch* ptr2 = static_cast<FirstBranch*>(ptr)`
- `FirstBranchContinue* ptr3 = static_cast<FirstBranchContinue*>(ptr2) // ну тут бан, UB. Но привидение произойдет`

Проверяется на этапе компиляции.

dynamic_cast

dynamic_cast приводит к типу, только если указатель реально указывает на объект этого типа. Иначе возвращает `nullptr`, в случае работы с ссылкой, возникает исключение `bad_cast`. Требование: классы, к которым он приводит, должны быть полиморфны(virtual на методе или virtual деструктор).

- Синтаксис такой же:
- `Type* a = dynamic_cast<Type*>(otherPtr).`
- Пример
- `Base* ptr = new FirstBranch;`
- `FirstBranch* ptr2 = dynamic_cast<FirstBranch*>(ptr)`
- `FirstBranchContinue* ptr3 = dynamic_cast<FirstBranchContinue*>(ptr2) // не приведется, вернет nullptr. Если работаем со ссылкой - возникает исключение bad_cast.`

const_cast

const_cast - убирает модификатор const. Некоторые компиляторы запрещают убирать const, если изначально объект был константным.

- Пример
- `class B`
- `{`
- `   f() // неконстантный метод`
- `}`
- `const B obj;`
- `const B* p = &obj;`
- `const_cast<B*>(p)->f(); // снимаем константность и вызываем неконстантный метод`

reinterpret_cast

reinterpret_cast - эквивалентен приведению в C

- тык
- `class A {};`
- `A* pA = new A;`
- `char* pByte = reinterpret_cast<char*>(pA); // Бандитизм, но reinterpret_cast позволяет`

«Умные указатели» в C++: unique_ptr, shared_ptr, weak_ptr.

Проблема - утечка памяти. Потеря указателя на объект. Не удаление объекта. Приводит к Утечке памяти

Проблема - висящий указатель. Указатель на объект, которого нет. Приводит к крашу системы.

Допустим на один объект держат ссылки разные объекты. Проблема контроля за валидностью.

Важно упомянуть

Важно сказать про make_shared и подобные

- Скрывает new
- Делает более безопасным
- На cppreference написано, что `std::shared_ptr<T>(new T(args...))` делает как минимум две аллокации (один для объекта T и один для блока управления общим указателем), а make_shared обычно выполняет только одно выделение (стандарт рекомендует, но не требует этого; все известные реализации делают это)

unique_ptr

unique_ptr - жестко держит один объект и не дает во владение другим. Отсутствует конструктор копирования и оператор присваивания. Может рассматриваться как временный объект, поэтому определяется конструктор переноса.

Есть метод `get()`, отдающий указатель. Если удалить из под одного указателя память, то получится висящий unique_ptr

Есть метод `release()`. Перестает владеть объектом и отдает указатель.

Метод `reset()` - сброс владения или замена на другой объект.

Метод `swap(unique_ptr& other_ptr)` - обменивает значения под указателями.

`unique_ptr` - занимает мало места в памяти, хранит только указатель.

Простейшая оболочка над указателем. Основная проблема, которую он решает - обработка исключительных ситуаций. Он освобождает память при выходе из области видимости.

shared_ptr

shared_ptr - используется, если несколько объектов указывают на один объект.

Держит в себе помимо указателя кол-во ссылок (shared_ptr'ов) на объект (совместное владение). Как только кол-во ссылок = 0, объект освобождается. // на Украине последний уезжающий выключает свет

Методы для `shared_ptr`:

`get()`, как у `unique_ptr`

`reset()` - уменьшает счетчик shared'ов и становится nullptr

`use_count()` - кол-во shared'ов

`unique()` - true, если `shared_ptr` один

weak_ptr

weak_ptr(слабое владение) идет в паре с `shared_ptr`. Он не отвечает за освобождение памяти. Можно только проверить - есть объект или нет. Для использования `weak_ptr` нужно сделать из `weak_ptr` - `shared_ptr` (метод `lock()`);

Методы для `weak_ptr`:

`reset()`,

`use_count()` - кол-во weak'ов,

`expired()` - есть объект или нет.

`lock()` - сделать из `weak_trp` - `shared_ptr`.

Они связаны, так как оба имеют доступ к счетчику ссылок на объект. ПОВОРОТ. Счетчиков два - один для weak_ptr'ов, второй для shared_ptr'ов. Объект удаляется, когда счётчик shared_ptr'ов становится равен 0.

`shared_ptr` и `weak_ptr` образованы от общей базы, которая держит счетчик shared'ов и weak'ов

для shared_ptr и unique_ptr есть операторы `*`, `->`, `bool`, `[]`. для `weak_ptr` - нет

В примерах к лекции 8 реализован каждый из указателей.

- Таблица по умным указателям(скролить по горизонтали)

Контейнерные классы и итераторы. Работа с итераторами. Цикл for для работы с контейнерными объектами.

Общий механизм работы итераторов.

Итератор нужен для того, чтобы унифицировать работу с контейнерным классом.

Есть стандартный шаблон iterator, у которого есть специализации под разные виды работы с итераторами, у каждой есть свой тег:

1. Итератор ввода - <input_iterator_tag> (можем менять то, на что он указывает)
2. Итератор вывода - <output_iterator_tag> (не можем менять то, на что он указывает)
3. Однонаправленный итератор на чтение и запись - <forward_iterator_tag> (`*`, `->`, `bool`, `++`(префиксный и постфиксный инкремент), `!=`, `==`)
4. Двунаправленный итератор на чтение и запись - <bidirectional_iterator_tag> (`*`, `->`, `bool`, `++`(префиксный и постфиксный инкремент), `--`, `!=`, `==`)
5. Итератор произвольного доступа - <random_access_iterator_tag> (`*`, `->`, `bool`, `++`(префиксный и постфиксный инкремент), `--`, `!=`, `==`, `>`, `<`, `>=`, `<=`, `[]`, `+=`, `-=`, `+`, `-` (сложение и вычитание с целым числом))

Теги итераторов отличаются набором операций и оптимизациями под свою задачу.

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

С помощью итератора просматриваем содержание контейнера.

Мы должны иметь возможность сравнивать итераторы и получать доступ к данным, на которые он указывает (операторы `*`, `->`). В каждом итераторе надо минимум реализовать сравнение итераторов (`!=`), доступ к данным (`*`, `->`, `bool`) и увеличение итератора (`++`).

У контейнерного класса должны быть методы `begin()` и `end()`, возвращающие итератор на начало и конец (или за них), для универсального просмотра контейнерного класса.

Если они реализованы, то можно применять цикл `foreach` (Тассов это называет так, а вообще правильно ”Range-based for loop”), который перебирает все элементы контейнера от `begin()` до `end()`:

- Синтаксис
- `for (<тип>& <имя> : <container>)`
- `{}`
- Пример разложения
- `// Это:`
- `for (auto elem : obj)`
- `  cout << elem;`
-
- `// Разложится в это:`
- `for (Iterator <Type> It = obj.begin(); It != obj.end(); ++It)`
- `{`
- `  auto elem = *It;`
- `  cout << elem;`
- `}`

В контейнерном классе должны быть методы `begin()` и `end()` , для итератора - `!=`, `++`, `*`

Также могут быть реализованы в контейнере методы `begin() const`, `end() const` и `cbegin()`, `cend()`

возвращающие константный итератор для константного объекта и константный для неконстантного соответственно