4.32 Идиома SFINAE. Простейший пример выбора из двух функций с использованием SFINAE (когда более подходящую функцию пришлось отбросить). Структура enable_if, ее реализация и пример правильного использования.

Идиома SFINAE.

SFINAE - правило вывода шаблонных версий функций.
Substituon Failure Is Not An Error - неудачная шаблонная подстановка не является ошибкой компиляции.

В данном примере мы не смогли подставить T = int, так как у данного типа нет метода size(). Вместо CE компилятор отбрасывает верхнюю версию и идёт искать дальше подходящую, это и есть SFINAE:

```
    template <typename T>
    auto f(const T&) -> decltype(T().size()) {
        std::cout << 1;
    }

    void f(...) {
        std::cout << 2;
    }

    int main() {
        f(5); //2 is printed
    }
```

Работает для сигнатуры, не работает для тела функции.

Структура enable_if

Структура, которая позволяет выбирать перегрузки функций в зависимости от compile-time проверяемых условий

Реализация enable_if

```
    template <bool B, typename T = void>
        struct enable_if{};

    template <typename T>
    struct enable_if<true, T>{
        using type = T;
    };

    template <bool B, typename T = void>
    struct enable_if_t{
        using type = typename enable_if<B, T>::type;
    };
```

Использование enable_if

```
    template<typename  T, typename = std::enable_if_t<std::is_class_v<T>>>
    void g(const T&) {
            std::cout << 1;
    }
```

```
5
6       template<typename T, typename = std::enable_if_t<!std::is_class_v<
7       std::remove_reference_t<T>>>>
8       void g(T&&) {
9               std::cout << 2;
10      }
```

4.33 Константные выражения, ключевое слово constexpr. Отличие constexpr от const. Контексты, в которых требуются константные выражения. constexpr-функции и ограничения на их содержимое. Особенности throw в constexpr-функциях, особенности создания объектов в constexpr-функциях. Ключевое слово static_assert и его применение. Пример ситуации, когда static_assert неприменим.

Некоторые контексты требуют, чтобы переменная была известна на этапе компиляции.

Отличие constexpr от const

Если переменная является константной, это ещё не значит, что она compile-time вычисляемая:

```
int x;
cin >> x;
const int y = x;
```

Контексты, в которых требуются константные выражения

Constexrt - тип выражения, который является compile-time вычислимым. Такие выражения можно делать шаблонными параметрами:

```
constexpr int y = x;
std::array<int, y> arr;
```

Второй контекст, в котором это может потребоваться - инициализация статической константы: static const int y = constexpr-type expr.

Constexpr-функции и ограничения на их содержимое

Рассмотрим пример, который не скомпилируется:

```
/*constexpr*/ int factorial(int n) {
    if (n==0) return 1;
    return n * factorial(n-1);
}

int main() {
    constexpr int y = factorial(5);
    return 0;
}
```

Вызов функции должен быть константным выражением. Чтобы сделать его таковым, раскомментируем constexpr.

Constexpr-функции - такие, которые могут быть исполнены в compile-time.

Ограничения на содержимое:

1. Не виртуальна (до 20 стандарта)
2. Возвращаемый тип - литеральный
3. Каждый параметр - литеральный тип

4. Для конструктора (после 20 стандарта): класс не должен иметь виртуальный базовый класс

5. Тело функции не должно быть function-try-block

Литерал — это элемент программы, который непосредственно представляет значение. Целочисленные литералы начинаются с цифры и не имеют дробных частей или экспонент. Литералы с плавающей запятой задают значения, которые должны иметь дробную часть. Эти значения содержат десятичные разделители (.) и могут содержать экспоненты и далее, например, логические литералы true & false.

Особенности throw в constexpr функциях, особенности создания объектов в constexpr функциях

В constexpr функциях нельзя бросать исключения.

В общем случае, нельзя создавать объекты на этапе компиляции, но если конструктор помечен constexpr, то можно.

Если функция помечена constexpr это означает, что она **может** быть вычислена на этапе компиляции, но если мы вызываем её в неподходящем для этого контексте, она будет вызвана уже в runtime.

Ключевое слово static_assert и его применение

Проверяет программное утверждение во время компиляции. Если указанное константное выражение имеет значение false , компилятор отображает указанное сообщение, если оно предоставлено, и компиляция завершается ошибкой; в противном случае объявление не оказывает никакого влияния.

```
static_assert( constant-expression, string-literal );
static_assert( constant-expression ); // C++17
```

Компилятор проверяет static_assert объявление на наличие синтаксических ошибок при обнаружении объявления. Компилятор вычисляет параметр константного выражения немедленно, если он не зависит от параметра шаблона. В противном случае компилятор вычисляет параметр константного выражения при создании экземпляра шаблона. Таким образом, компилятор может вывести одно диагностическое сообщение, когда встретит объявление, а второе — когда будет создавать экземпляр шаблона.

Пример с областью видимости класса:

```
template <class T>
void swap(T& a, T& b) noexcept
{
    static_assert(std::is_copy_constructible<T>::value,
                  "Swap requires copying");
    static_assert(std::is_nothrow_copy_constructible<T>::value
               && std::is_nothrow_copy_assignable<T>::value,
                  "Swap requires nothrow copy/assign");
    auto c = b;
    b = a;
    a = c;
}
```

Пример ситуации, когда static_assert неприменим.

В целом, static_assert проверяет ровно то, что можно отловить на этапе компиляции. Если вы пытаетесь поймать что-то, что происходит в runtime, согласно изложенному выше, может возникнуть CE.

```
int main() {
    int x;
    cin >> x;
    static_assert(x==0);
    return 0;
}
```