MUHAMMAD AL-XORAZMIY
NOMIDAGI TATU FARG'ONA FILIALI

FERGANA BRANCH OF TUIT
NAMED AFTER MUHAMMAD AL-KHORAZMI

"AL-FARG'ONIY AVLODLARI"

ELEKTRON ILMIY JURNALI | ELECTRONIC SCIENTIFIC JOURNAL

TA'LIMDAGI ILMIY, OMMABOP VA ILMIY TADQIQOT ISHLARI

2-SON 1(2)
2023-YIL

TATU, FARG'ONA
O'ZBEKISTON

ISSN 2181-4252

O’ZBEKISTON RESPUBLIKASI RAQAMLI TEXNOLOGIYALAR VAZIRLIGI

MUHAMMAD AL-XORAZMIY NOMIDAGI TOSHKENT AXBOROT TEXNOLOGIYALARI UNIVERSITETI FARG’ONA FILIALI

Muassis: Muhammad al-Xorazmiy nomidagi Toshkent axborot texnologiyalari universiteti Farg‘ona filiali.
Chop etish tili: O‘zbek, ingliz, rus.
Jurnal texnika fanlariga ixtisoslashgan bo‘lib, barcha shu sohadagi matematika, fizika, axborot texnologiyalari yo‘nalishida maqolalar chop etib boradi.

Учредитель: Ферганский филиал Ташкентского университета информационных технологий имени Мухаммада ал-Хоразми.
Язык издания: узбекский, английский, русский.
Журнал специализируется на технических науках и публикует статьи в области математики, физики и информационных технологий.

Founder: Fergana branch of the Tashkent University of Information Technologies named after Muhammad al-Khorazmi.
Language of publication: Uzbek, English, Russian.
The magazine specializes in technical sciences and publishes articles in the field of mathematics, physics, and information technology.

2023 yil, Tom 1, №2
Vol.1, Iss.2, 2023 y

ELEKTRON ILMIY JURNALI **ELECTRONIC SCIENTIFIC JOURNAL**

«Al-Farg‘oniy avlodlari» («The descendants of al-Fargani», «Potomki al-Fergani») O’zbekiston Respublikasi Prezidenti administratsiyasi huzuridagi Axborot va ommaviy kommunikatsiyalar agentligida 2022-yil 21 dekabrda 054493-son bilan ro’yxatdan o’tgan.

Tahririyat manzili:
151100, Farg‘ona sh., Aeroport ko‘chasi 17-uy, 201A-xona
Tel: (+99899) 998-01-42
e-mail: info@al-fargoniy.uz
Qo‘lyozmalar taqrizlanmaydi va qaytarilmaydi.

FARG’ONA - 2023 YIL

TAHRIR HAY'ATI

Jurnal quyidagi bazalarda indekslanadi:

MUNDARIJA | ОГЛАВЛЕНИЕ | TABLE OF CONTENTS

АЛЬТЕРНАТИВНЫЕ ПОДХОДЫ К ОБРАБОТКЕ ОШИБОК: СРАВНЕНИЕ EXCEPTIONS И STD::EXPECTED В C++

Эрматова Зарина Кахрамоновна,
Ассистент Ферганского филиала
Ташкентского университета информационных технологий
имени Мухаммада ал-Хоразми

Аннотация: В данной статье хочу немного рассказать о небольшом исследовании реализации expected, в которой используется стирание типа ошибки. Посмотрев на новый тип из грядущего стандарта под названием std::expected я пришел к интересному на мой взгляд мнению, что можно немного переосмыслить его суть и сделать несколько ближе к исключениям.

Ключевые слова: c++, ошибка, exceptions, исчключение, пространства имен, программирование, библиотека, MathError, type erasure

Введение. Данное введение представляет собой обзор библиотеки std::expected в языке программирования C++. Библиотека std::expected предоставляет механизм для работы с результатами операций, позволяя управлять ошибками и исключительными ситуациями в C++ коде.

std::expected представляет альтернативу классическим механизмам обработки ошибок, таким как исключения или возвращение специальных значений. Она предлагает элегантное решение, которое позволяет разработчикам явно указывать, что функция может вернуть успешный результат или ошибку, а также явно обрабатывать эти случаи.

В основе std::expected лежит контейнер, который может содержать либо значение, либо объект ошибки. Такой подход обеспечивает прозрачную и безопасную передачу ошибок между функциями и позволяет исключить неопределенное поведение или необработанные исключения.

Использование std::expected упрощает кодирование обработки ошибок и повышает надежность программы. Разработчики могут явно указывать, что функция может вернуть ошибку, а вызывающий код может легко проверять и обрабатывать эту ошибку. Это способствует созданию более надежного и поддерживаемого кода в C++.

Теперь, когда был представлен обзор std::expected в языке программирования C++, давайте рассмотрим его основные возможности и примеры использования.[1]

Литературный обзор: В данной статье проведен обзор литературы, связанной с использованием библиотеки std::expected в C++. Были изучены научные публикации, документация и сообщества разработчиков, чтобы оценить преимущества и применение std::expected в различных сценариях разработки.

В литературе отмечается, что std::expected предоставляет строгую типизацию и контроль над ошибками в C++. Библиотека предлагает простой и эффективный способ возвращать результаты операций вместе с информацией об ошибке. Это позволяет более точно определить, какие функции могут вернуть ошибку, и обеспечивает явную обработку и передачу ошибок между компонентами программы.[2]

Методология: Для достижения целей исследования, проведенного в данной статье, был использован следующий методологический подход. Во-первых, был проведен анализ существующих решений для обработки ошибок в C++, таких как исключения, коды ошибок или возврат специальных значений. Это позволило оценить преимущества и недостатки каждого подхода и выявить потенциальные проблемы, с которыми может столкнуться разработчик.

Во-вторых, было изучено документацию по библиотеке std::expected, а также реализации и примеры использования в существующих проектах. Это помогло понять основные

https://doi.org/10.5281/zenodo.7979482

https://journals.indexcopernicus.com/search/article?articleId=3609148

https://al-fargoniy.uz/

концепции и возможности библиотеки, а также оценить ее производительность и надежность.

В-третьих, были разработаны и проведены эксперименты для оценки эффективности std::expected в сравнении с другими подходами обработки ошибок в C++. Эксперименты включали создание простых программных модулей с использованием std::expected и анализ их производительности, сложности кода и общего качества программы.

На основе полученных результатов исследования были сделаны выводы о применимости и эффективности std::expected в контексте разработки на C++. Были идентифицированы сильные стороны и ограничения библиотеки, а также предложены рекомендации по ее использованию в реальных проектах.

Таким образом, предложенная методология исследования позволила достичь целей статьи и изучить возможности и применение библиотеки std::expected в контексте разработки на языке C++.

Результаты. Данный тип задумывался как один из вариантов обработки ошибок. В отличие от исключений, он хорош тем, что даёт дополнительный выигрыш в производительности, благодаря отсутствию необходимости разворачивания стека, а также освобождает программиста от рутинных задач, таких как явное указание noexcept в API своего проекта. Он представляет собой своего рода золотую середину между исключениями (уже привычным механизмом в C++) и возвращаемыми кодами ошибки (как принято делать в языке C).

Данный тип в общем случае представляет собой контейнер в стиле std::optional, но с двумя параметрами шаблона: T (тип, значение которого содержится в контейнере) и E (тип ошибки, содержащейся в этом контейнере).[3]

```
std::expected<std::string, int> foo = "hello";
```

При использовании этого типа мы можем либо попробовать достать оттуда значение, либо запросить о том какое там значение ошибки. Обычно это выглядит следующим образом:

```
enum class MathError : unsigned char
{
    ZeroDivision,
    NegativeNotAllowed
};

std::expected<int, MathError> Bar(int a, int b)
{
  if (b == 0)
    return std::unexpected(MathError::ZeroDivision);
  if (a < 0 || b < 0)
    return
std::unexpected(MathError::NegativeNotAllowed);

  return a / b;
}

int main()
{
  std::expected<int, MathError> foo = Bar(1, 3);

  if (foo.has_value())
  {
    std::cout << *foo;
  }
  else if (foo.error() == MathError::ZeroDivision)
  {
    std::cout << "Divided by zero";
  } else if (foo.error() ==
MathError::NegativeNotAllowed)
  {
    std::cout << "Negative numbers not allowed";
  }
}
```

То есть мы знаем заранее какой тип ошибки у нас может быть, и в данном примере это MathError. А что если под Bar подразумевается довольно неоднозначная логика? Может быть арифметическая ошибка, а может системная. Первое, что приходит на ум - это сделать enum с разными значениями ошибки, таким образом мы привязываемся явно к этому перечислению.[4] Однако, можно ли "скрыть" этот тип и определять ошибку динамически во время выполнения программы?

Стирание типа ошибки. Стирание типа (type erasure) - паттерн в языке C++, который основан на использовании шаблонов и полиморфизма. Что же это дает нам? Он позволит сохранять ошибку какого угодно типа в объекте типа expected в любой момент времени, при этом сигнатура объявления переменной сокращается до одного шаблонного аргумента, например expected<int>.[5]

Общий интерфейс класса при этом выглядел бы примерно так:

```
template<typename T>
class Expected
```


https://doi.org/10.5281/zenodo.7979482

https://journals.indexcopernicus.com/search/article?articleId=3609148

```
{
public:

  template<typename E>
  Expected(Unexpected<E> Unexp)
  {
    SetError(Unexp.Error);
  }

  Expected(T Value)
  {
    StoredValue = Value;
  }

  bool HasError() const;

  template<typename E>
  void SetError(E&& Error);

  template<typename E>
  const E* GetError() const;

  bool HasValue() const;

  inline operator T() const;

protected:

  std::optional<T> StoredValue;

  // Сюда будет помещаться сама ошибка
  std::unique_ptr<ErrorHolderBase> StoredError;
};

// Структура, необходимая для передачи ошибки в
Expected
template<typename E>
struct Unexpected
{
  Unexpected(E InError)
  {
    Error = InError;
  }
  E Error;
};
```

Теперь к реализации с помощью стирания типа. И первое, что мы делаем это объявляем базовый класс хранителя ошибки.

```
struct ErrorHolderBase
{
  // Возвращает текст ошибки
  virtual std::string GetErrorText() const = 0;

  // Возвращает указатель на хранимую ошибку
  virtual void* GetErrorPtr() const = 0;

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

Возникает вопрос, а что нам даст указатель на ошибку стёртый до void*? Чтобы решить его мы можем использовать идентификаторы типов, как это реализовано в std::any, и тут можно пойти двумя путями: использовать RTTI с typeid, либо отказаться от RTTI и сделать самописный счётчик уникальных идентификаторов типов, чтобы уметь различать типы ошибок друг от друга. Второй вариант мне больше импонирует, так как я работаю в проекте, в котором, по соглашению, RTTI отключено (привет, UnrealEngine).[6] В общем буду пользоваться своим велосипедом и приведу в спойлере пример реализации такого счётчика:

```
struct TypeIdCnt
{
  template<typename>
  static uint32 GetUniqueId()
  {
    static const int32 TypeId = NewTypeId();
    return TypeId;
  }

private:
  static uint32 NewTypeId()
  {
    // thread-safe
    static std::atomic<uint32> CurrentId = 0;
    return CurrentId++;
  }
};

template<typename T>
static uint32 GetTypeId()
{
  return TypeIdCnt::GetUniqueId<T>();
}
```

Суть такова: каждый новый тип T создаёт новый инстанс функции, что увеличивает счётчик.

Мы будем сохранять так же и идентификатор типа в хранилище ошибки. Теперь код будет выглядеть вот так:

```
struct ErrorHolderBase
{
  // Возвращает текст ошибки
  virtual std::string GetErrorText() const = 0;

  // Возвращает указатель на хранимую ошибку
```


https://doi.org/10.5281/zenodo.7979482

https://journals.indexcopernicus.com/search/article?articleId=3609148

```
  virtual void* GetErrorPtr() const = 0;

  // Возвращает либо указатель на ошибку, либо
nullptr, если тип не соответствует
  virtual void* RetrieveError(uint32 ErrorTypeId)
const = 0;

  virtual ~ErrorHolderBase() {}

  std::set<uint32> Bases;
};

template<typename ErrorType>
struct ErrorHolder : ErrorHolderBase
{
  ErrorHolder(ErrorType InError)
  {
    Error = InError;
  }

  virtual std::string GetErrorText() const
  {
    // для каждого типа ошибки можно перегрузить
функцию error_to_str для получения текстового
представления
    return error_to_str(*Error)
  }

  virtual void* GetErrorPtr() const
  {
    return reinterpret_cast<void*>(&Error);
  }

  virtual void* RetrieveError(uint32 ErrorTypeId)
const
  {
    if (GetTypeId<ErrorType>() == ErrorTypeId)
      return GetErrorPtr();
    return nullptr;
  }

  ErrorType Error;
};
```

Мы можем получать ошибку из контейнера, зная её тип. Однако, это означает, что в контейнере может быть любой тип ошибки, а получить ошибку мы сможем только если предположим правильный тип (указывать в качестве шаблонного параметра). Это ограничивает возможность классификации ошибок по категориям.[7] Такой подход подходит для базовых типов, строк и других типов, которые не требуют категоризации ошибок. Хотелось бы добавить дополнительный метод под названием Catch, который эмулировал бы механизм исключений (в некоторой степени), позволяя извлекать ошибки из нового варианта expected по категориям (хранить наследника и ловить по родителю).[8] Пример кода может выглядеть следующим образом:

```
struct BaseError{};
struct MathError : BaseError{};
struct SystemError : BaseError{};

expected<int> ValueOrError = unexpected(MathError());

if (auto Error = ValueOrError.Catch<BaseError>())
{
  // ...
}
```

Чтобы решить данную задачу, мы можем сохранять идентификаторы классов ошибок непосредственно во всю их иерархию. Для этого мы воспользуемся паттерном "декоратор", который будет добавлять наследнику идентификатор его родителя. Таким образом мы можем получить идентификаторы всех типов в иерархии:

```
template<typename T>
struct DeriveError : T
{
  using T::T;

  // Данная функция собирает идентификаторы из всей
иерархии рекурсивно
  static std::set<uint32> GetBaseIds()
  {
    std::set<uint32> Bases = { GetTypeId<T>() };
      // Так же спрашиваем идентификаторы у родителя
    Bases.merge(T::GetBaseIds());
    return Bases;
  }

};
```

Так же, необходим какой-то базовый класс ошибки наподобие std::exception, который хранит как свой идентификатор, так и предоставляет некоторый интерфейс для получения информации об ошибке.

```
struct ErRuntimeError
{
  ErRuntimeError(const std::string& InMessage)
  {
    Message = InMessage;
  }

  static std::set<uint32> GetBaseIds()
  {
```


https://doi.org/10.5281/zenodo.7979482

https://journals.indexcopernicus.com/search/article?articleId=3609148

https://al-fargoniy.uz/

```
    return { GetTypeId<ErRuntimeError>() };
  }

  std::string What() const
  {
    if (Message.IsEmpty())
          return GetErrorType();
    return GetErrorType() + ": " + Message;
  }

  virtual std::string GetErrorType() const
  {
    return "RuntimeError";
  }

  virtual ~ErRuntimeError() = default;

protected:
  std::string Message;

};

// Перегрузка для получения текстового представления об ошибке
inline std::string error_to_str(const ErRuntimeError& Error)
{
  return Error.What();
}
```

Так же имеет смысл завести макрос, которым можно будет создавать новые типы ошибок, чтобы миновать boilerplate-кода:

```
#define DEFINE_RUNTIME_ERROR(Error, Parent) \
  struct Error : DeriveError<Parent> \
  { \
    using ParentType = DeriveError<Parent>; \
    using ParentType::ParentType; \
    virtual FString GetErrorType() const override \
    { \
      return #Error; \
    } \
  };
```

Теперь объявления ошибок будут выглядеть таким образом:

```
// Мат. ошибка
DEFINE_RUNTIME_ERROR(ErMathError, ErRuntimeError);
// Мат. ошибка - деление на ноль
DEFINE_RUNTIME_ERROR(ErZeroDivisionError, ErMathError);
// Ошибка значения
DEFINE_RUNTIME_ERROR(ErValueError, ErRuntimeError);
```

И сейчас, когда у нас есть иерархия ошибок с идентификаторами, мы можем написать свой вариант Catch для нового expected.[9] При получении ошибки, мы имеем полное право явно прикастовать void* к E*, так как CatchError обязан выдать указатель на ошибку, если переданный идентификатор существует в иерархии, либо вернуть nullptr.[10]

```
template<typename T>
template<typename E>
const E* Expected<T>::Catch()
{
  const int32 ErrorTypeId = GetTypeId<E>();
  return static_cast<E*>(StoredError->CatchError(ErrorTypeId));
}
```

А при установке ошибки мы делаем что-то вроде этого:

```
template<typename T>
template<typename E>
void Expected<T>::SetError(E&& Error)
{
  StoredError = std::make_unique<ErrorHolder<E>>(std::forward(Error));

  if constexpr (std::is_base_of_v<ErRuntimeError, E>)
  {
    std::set<uint32> Bases = E::GetBaseIds();
    Bases.add(GetTypeId<E>());
    StoredError->SetBases(Bases);
  }
}
```

Здесь создаётся само хранилище ошибки. А далее просто передаются идентификаторы типов всей высшей иерархии ошибки E (включая саму ошибку) в хранилище ошибки.[11]

Вместо вышеупомянутого в статье RetrieveError, теперь мы можем пользоваться методом CatchError нашего полиморфного хранилища ошибки, который прежде чем выдать указатель на ошибку, проверяет, есть ли такой идентификатор типа в сохраненном ранее списке иерархии, либо возвращает nullptr.[12]

```
void* ErrorHolderBase::CatchError(uint32 ErrorTypeId) const
{
  if (Bases.contains(ErrorTypeId))
    return GetErrorPtr();
  return nullptr;
```


https://doi.org/10.5281/zenodo.7979482

https://journals.indexcopernicus.com/search/article?articleId=3609148

https://al-fargoniy.uz/

```
}

void ErrorHolderBase::SetBases(const
std::set<uint32>& InBases)
{
  Bases = InBases;
}
```

Теперь мы можем проверять содержимое expected в стиле исключений C++:

```
Expected<int> Bar(int a, int b)
{
  if (b == 0)
    return Unexpected(ErZeroDivisionError("b is
Zero"));

  if (a < 0 || b < 0)
    return Unexpected(ErNegativeNotAllowed("a < 0 or
b < 0"));

  return a / b;
}

int main()
{
  Expected<int> foo = Bar(1, 3);

  if (foo.has_value())
  {
    std::cout << *foo;
  }
  else if (auto ZDError =
foo.Catch<ErZeroDivisionError>())
  {
    std::cout << ZDError->What();
  }
}
```

Заключение. Для чего может понадобиться такой вариант expected?

Мои причины использовать expected со стёртым типом ошибки следующие:

1. Более простая семантика объявления ожидаемых значений. Использование стёртого типа ошибки в expected позволяет сократить сигнатуру объявления переменной до одного шаблонного аргумента. Это улучшает читаемость и понимание кода.

2. Возможность устанавливать любой тип ошибки на лету. Если гипотетическая функция имеет возможность создать разного рода ошибки, то почему бы не взять ошибку оттуда, откуда она реально возникла (из другого expected) и передать её в текущий expected? Кстати, данный пункт очень красиво ложится на монадический подход применения expected с использованием сопрограмм, что, к слову, ещё сильнее приближает удобство пользования expected к исключениям. Подумываю написать статью так же и об этом.

3. Обработка ошибок по категориям. Использование стёртого типа ошибки позволяет обрабатывать ошибки по категориям, что приближает expected к механизму исключений. Это дает гибкость и удобство при обработке различных видов ошибок.

Причины, почему не стоит использовать стёртый тип ошибки:

1. Отсутствие возможности использовать в compile-time. Динамический полиморфизм не даёт возможность использовать expected во время компиляции.

2. Неопределенный тип ошибки затрудняет понимание источника ошибки. Однако на мой взгляд это легко можно решить с помощью дополнительных средств языка, например добавить в хранилище ошибки так же и информацию о месте в исходном коде, где возникла эта ошибка: std::source_location

3. Потребление большего объема памяти. Использование стёртого типа ошибки в expected требует хранения идентификаторов классов предков ошибки, что может привести к увеличению потребления памяти.

Список использованной литературы

1. Солиев, Б. Н. Перспективы развития электронной торговли и онлайн-курсов в Узбекистане на основе системы LMS / Б. Н. Солиев // Исследования молодых ученых : Материалы XV Международной научной конференции, Казань, 20–23 декабря 2020 года / Под редакцией И.Г. Ахметова [и др.]. – Казань: Общество с ограниченной ответственностью "Издательство Молодой ученый", 2020. – С. 1-3. – EDN NFKTEB.

2. Солиев Б. Н. и др. ТЕНДЕНЦИИ РАЗВИТИЕ ИНТЕРНЕТ-ЛОГИСТИКИ В ЭЛЕКТРОННОЙ КОММЕРЦИИ //Журнал Технических исследований. – 2022. – Т. 5. – №. 1.

3. На грани между exceptions и std::expected, https://habr.com/ru/articles/737408/

https://doi.org/10.5281/zenodo.7979482

https://journals.indexcopernicus.com/search/article?articleId=3609148

https://al-fargoniy.uz/

4. Soliev B. N., kizi Abdurasulova D. B., Yakubov M. S. USING GINJA TEMPLATES TO CREATE E-COMMERCE PLATFORMS //Publishing House “Baltija Publishing”. – 2023.

5. Nabijonovich S. B., Mahamatovich R. A. Prospects for the Development of Electronic Trade Processes Based on Local Characteristics //International Journal on Orange Technologies. – 2021. – Т. 3. – №. 3. – С. 305-309.

6. Soliev B. N., Abdurasulova D., Yakubov M. S. USING THE DJANGO FRAMEWORK FOR E-COMMERCE PROCESSES //Journal of Integrated Education and Research. – 2022. – Т. 1. – №. 6. – С. 229-233.

7. Солиев Б. Н. и др. ИЗУЧИТЬ ОПЫТ ДРУГИХ СТРАН ПО РАЗВИТИЮ ЭЛЕКТРОННОЙ КОММЕРЦИИ В УЗБЕКИСТАНЕ //Журнал Технических исследований. – 2022. – Т. 5. – №. 1.

8. Musayev, X., & Soliev, B. (2023). PUBLIC, PROTECTED, PRIVATE MEMBERS IN PYTHON. Потомки Аль-Фаргани, 1(1), 43–46. извлечено от https://al-fargoniy.uz/index.php/journal/article/view/17

9. Zulunov, R., & Soliev, B. (2023). IMPORTANCE OF PYTHON LANGUAGE IN DEVELOPMENT OF ARTIFICIAL INTELLIGENCE. Потомки Аль-Фаргани, 1(1), 7–12. извлечено от https://al-fargoniy.uz/index.php/journal/article/view/3

10. Солиев Б. Н. Проблемы моделирования электронных торговых процессов на основе местных характеристик //Исследования молодых ученых. – 2020. – С. 8-11.

11. Musayev X. S. H., Ermatova Z. Q., Abdurahimova M. I. Kotlin dasturlash tilida klasslar va ob’yektlar tushunchasi //Journal of Integrated Education and Research. – 2022. – Т. 1. – №. 6. – С. 126-130.

12. Musayev X.SH., Ermatova Z.Q. Kotlin dasturlash tilida korutinlar bilan ishlashni talabalarga o‘rgatish //Journal of Integrated Education and Research. – 2022. – Т. 1. – №. 6. – С. 119-125.

https://doi.org/10.5281/zenodo.7979482

https://journals.indexcopernicus.com/search/article?articleId=3609148