solarwinds

How to cook std::system_error

Yury Efimochev

Who am I?

SolarWinds Backup

Principal developer

yury.efimochev@solarwinds.com

efimyury@gmail.com

Example domain

```
struct Item;

class IDataSource
{
public:
    virtual IStreamPtr GetContent(Item const& entity) const = 0;
};

class IStorage
{
public:
    virtual void Place(Item const& entity, IStreamPtr stream) = 0;
};
```

Naive example

```
void Backup(IDataSource const& source, Items const& items, IStorage& storage)
{
    for (auto const& item : items)
    {
        storage.Place(item, source.GetContent(item);
    }
}
```

Example domain

```
struct Item;

class IDataSource
{
public:
    virtual IStreamPtr GetContent(Item const& entity) const = 0;
};

class IStorage
{
public:
    virtual void Place(Item const& entity, IStreamPtr stream) = 0;
};

class Exception : public std::exception { /*...*/ };
```

Error handling

```
void Backup(IDataSource const& source, Items const& items, IStorage& storage)
{
    for (auto const& item : items)
    {
        try
        {
            storage.Place(item, source.GetContent(item));
        }
        catch (Exception const&)
        {
        }
    }
}
```

Error handling

```
class Exception : public std::exception
{
    // ...
    Error GetError() const;
    // ...
};
```

@solarwinds

Error handling

```
void Backup(IDataSource const& source, Items const& items, IStorage& storage)
{
    for (auto const& item : items)
    {
        try
        {
            storage.Place(item, source.GetContent(item));
        }
        catch (Exception const& e)
        {
            Error const error = e.GetError();
            std::cerr << "Backup error." <<
                error << ": " << GetErrorText(error) << std::endl;
        }
    }
}
```

Error identification

@solarwinds

God enum

```
// Error.h
enum class Error
{
    // ...
    InvalidArgument,
    // ...
    Http_AccessDenied,
    // ...
    FileSystem_NotEnoughSpace,
    // ...
    Database_TableIsLocked,
    // ...
    HyperV_ResourceNotFound,
    // ...
};

char const* GetErrorText(Error const error);
```

- High coupling
- Cross-module reuse
- Re-compilation

Plain old int

```
using Error = int;

// DataSourceError.h
enum class DataSourceError
{
    EntityNotFound = 2000,
    // ...
};

// StorageError.h
enum class StorageError
{
    NoSpaceLeft = 3000,
    // ...
};

// char const* GetErrorText(Error const error);
```

- Range synchronization
- GetErrorText?

Plain old string

```
using Error = char const*;

// DataSourceError.h
namespace DataSourceError
{
Error EntityNotFound = "DataSource:EntityNotFound";
// ...
};

// StorageError.h
namespace StorageError
{
Error NoSpaceLeft = "StorageError:NoSpaceLeft";
// ...
};

// char const* GetErrorText(Error error);
```

- Localization
- GetErrorText?

<system_error>

std::error_code

std::error_category

std::error_code

```
class error_code
{
private:
    int val;
    error_category const* cat;
    //...
};

class error_category
{
public:
    virtual char const* name() const = 0;
    virtual std::string message(int ev) const = 0;
    // ...
};
```

std::error_code

```
enum class StorageError
{
    NoSpaceLeft,
    AccessDenied,
    IOError,
    TemporaryUnavailable,
    // ...
};
```

std::error_code

```
std::error_category const& GetStorageErrorCategory()
{
    class Category : public std::error_category
    {
        char const* name() const override { return "Storage"; }
        std::string message(int errorValue) const override
        {
            // ...
        }
    };

    static const Category s_category;
    return s_category;
}
```

std::error_code

```
std::error_code const error(
    static_cast<int>(StorageError::NoSpaceLeft),
    GetStorageErrorCategory());
```

std::error_code

```
template<>
struct std::is_error_code_enum<StorageError> : public std::true_type {};

std::error_code make_error_code(StorageError e)
{
    return { static_cast<int>(e), GetStorageErrorCategory() };
}
```

std::error_code

```
std::error_code const error = StorageError::NoSpaceLeft;
```

std::error_code

```
void Backup(IDataSource const& source, Items const& items, IStorage& storage)
{
    for (auto const& item : items)
    {
        try
        {
            storage.Place(item, source.GetContent(item));
        }
        catch (Exception const& e)
        {
            std::error_code const error = e.GetError();
            std::cout << "Entity backup failed." <<
                error << " " << std::quoted(error.message()) << std::endl;
        }
    }
}
```

Output example

```
...
Entity backup failed. Storage:1 “No space left on device”
Entity backup failed. DataSource:42 “Access denied”
...
```

Error classification

Backup example

```
void Backup(IDataSource const& source, Items const& items, IStorage& storage)
{
    for (auto const& item : items)
    {
        try
        {
            storage.Place(item, source.GetContent(item));
        }
        catch (Exception const& e) { /* Log */ }
    }
}
```

Critical errors

```
enum class DataSourceError
{
    IOError,
    AccessDenied,
    EntityNotFound,
    // ...
};

enum class StorageError
{
    IOError, // Critical
    NoSpaceLeft, // Critical
    TemporaryUnavailable,
    // ...
};
```

Exception hierarchy

```
class Exception : public std::exception {};

class CriticalException : public Exception {};

class DataSource::IOErrorException : public Exception {};
class DataSource::AccessDeniedException : public Exception {};
class DataSource::EntityNotFoundException : public Exception {};

class Storage::IOErrorException : public CriticalException {};
class Storage::NoSpaceLeftException : public CriticalException {};
class Storage::TemporaryUnavailableException : public Exception {};
```

Exception hierarchy

```
void Backup(IDataSource const& source, Items const& items, IStorage& storage)
{
    for (auto const& item : items)
    {
        try
        {
            storage.Place(item, source.GetContent(item));
        }
        catch (CriticalException const&) { throw; }
        catch (Exception const& e) { /* Log */ }
    }
}
```

Exception hierarchy

One person's fatal error is another person's common case.

-Anonymized

Restore

```
void Restore(IStorage const& storage, Items const& items, IDataSource& source)
{
    for (auto const& item : items)
    {
        try
        {
            source.Place(item, storage.GetContent(item));
        }
        catch (CriticalException const&) { throw; }
        catch (Exception const& e) { /* Log */ }
    }
}
```

Exception hierarchy

```
enum class DataSourceError
{
    IOError, // Critical for restore
    AccessDenied, // Critical for restore
    EntityNotFound,
    // ...
};

enum class StorageError
{
    IOError, // Critical for backup
    NoSpaceLeft, // Critical for backup
    TemporaryUnavailable,
    // ...
};
```

Exception filter

```
void Backup(IDataSource const& source, Items const& items, IStorage& storage)
{
    for (auto const& item : items)
    {
        try
        {
            storage.Place(item, source.GetContent(item));
        }
        catch (Storage::IOErrorException cosnt&) { throw; }
        catch (Storage::NoSpaceLeftException cosnt&) { throw; }
        catch (Exception const& e) { /* Log */ }
    }
}
```

Exception filter

```
bool IsCriticalBackupException()
{
    try
    {
        throw;
    }
    catch (Storage::IOErrorException const&) { return true; }
    catch (Storage::NoSpaceLeftException const&) { return true; }
    catch (...) { return false; }
}
```

Exception filter

```
void Backup(IDataSource const& source, Items const& items, IStorage& storage)
{
    for (auto const& item : items)
    {
        try
        {
            storage.Place(item, source.GetContent(item));
        }
        catch (Exception const& e)
        {
            if (IsCriticalBackupException()) { throw; }

            // Log
        }
    }
}
```

std::error_code

```
void Backup(IDataSource const& source, Items const& items, IStorage& storage)
{
    for (auto const& item : items)
    {
        try
        {
            storage.Place(item, source.GetContent(item));
        }
        catch (std::system_error const& e)
        {
            if (e.code() == Storage::IOError ||
                e.code() == Storage::NoSpaceLeft) { throw; }

            // Log
        }
    }
}
```

std::error_condition

Samurai without a sword is the same as a samurai with a sword but without a sword.

std::error_condition vs std::error_code

```
class error_code
{
private:
    int val;
    error_category const* cat;
    //...
};

class error_condition
{
private:
    int val;
    error_category const* cat;
    //...
};
```

std::error_condition vs std::error_code

```
bool operator==(std::error_code const& left, std::error_code const& right)
{
    return
        left.category() == right.category() &&
        left.value() == right.value();
}

bool operator==(std::error_code const& error, std::error_condition const& condition)
{
    return
        condition.category().equivalent(error, condition.value()) ||
        error.category().equivalent(error.value(), condition);
}
```

std::error_category

```
namespace std
{

class error_category
{
public:
   // ...
   virtual bool equivalent(
      int code, error_condition const& condition) const noexcept = 0;
   virtual bool equivalent(
      error_code const& code, int condition) const noexcept = 0;
   // ...
};

}
```

std::error_condition

```
enum class BackupError
{
    Critical,
    Retryable,
    // ...
};

std::error_category const& BackupErrorCategory();

std::error_condition make_error_condition(BackupError e)
{
    return { static_cast<int>(e), BackupErrorCategory() };
}

template<>
struct std::is_error_condition_enum<BackupError> : public std::true_type {};
```

std::error_condition

```
bool BackupErrorCategory::equivalent(
    std::error_code const& code, int value) const noexcept override
{
    auto const condition = static_cast<BackupError>(value);

    switch (condition)
    {
    case BackupError::Critical:
        return
            code == StorageError::IOError ||
            code == StorageError::AccessDenied;
    // ...
    }

    return false;
}
```

std::error_condition

```
void Backup(IDataSource const& source, Items const& items, IStorage& storage)
{
    for (auto const& item : items)
    {
        try
        {
            storage.Place(item, source.GetContent(item));
        }
        catch (std::system_error const& e)
        {
            if (e.code() == BackupError::Critical) { throw; }

            // Log
        }
    }
}
```

Migration tips

Migration tips

```
{
    try
    {
        // ...
    }
    catch (DataSource::NotFoundException const& e) { /* ... */ }
    catch (DataSource::AccessDeniedException const& e) { /* ... */ }
    catch (DataSource::IOErrorException const& e) { /* ... */ }
    catch (DataSource::TemporaryUnavailableException const& e) { /* ... */ }
    catch (Storage::NotFoundException const& e) { /* ... */ }
    catch (Storage::AccessDeniedException const& e) { /* ... */ }
    catch (Storage::IOErrorException const& e) { /* ... */ }
    catch (Storage::TemporaryUnavailableException const& e) { /* ... */ }
    catch (Exception const& e) { /* ... */ }
    catch (std::exception const& e) { /* ... */ }
}
```

Migration tips

```
{
    try
    {
        // ...
    }
    catch (DataSource::NotFoundException const& e) { /* ... */ }
    catch (DataSource::AccessDeniedException const& e) { /* ... */ }
    catch (DataSource::IOErrorException const& e) { /* ... */ }
    catch (DataSource::TemporaryUnavailableException const& e) { /* ... */ }
    catch (Exception const& e) { /* ... */ }
    catch (std::system_error const& e)
    {
        if (e.code() == BackupError::Critical) { throw; }
        if (e.code() == BackupError::Transient) { /*...*/ }
    }
    catch (std::exception const& e) { /* ... */ }
}
```

ExceptionFilter helper

```
class ExceptionFilter
{
    // ...
    using Handler = std::function<void()>;

    template<typename ... Exceptions> ExceptionFilter& Rethrow()
    template<typename ... Exceptions> ExceptionFilter& Ignore();

    template<typename ... Exceptions> ExceptionFilter& On(Handler handler);
    ExceptionFilter& On(std::error_condition, Handler handler);
    ExceptionFilter& On(std::error_code, Handler handler);

    ExceptionFilter& Finally(Handler handler);
    ExceptionFilter& Default(Handler handler);
    void Nevermind();
    // ...
};
```

ExceptionFilter helper

```
class Ok : public std::exception {};
class TooBad : public std::exception {};
class Fine : public std::exception {};
class Allright : public std::exception {};
class Good : public std::exception {};
class Well : public std::exception {};
```

ExceptionFilter helper

```
{
    auto fallback = [](){ /*...*/ };
    auto whatever = [](){ /*...*/ };

    try
    {
        // ...
    }
    catch (...)
    {
        ExceptionFilter().
            On<Good>(fallback).Rethrow<Fine>().
            Ignore<TooBad, Ok, Allright>().
            On<Well>(whatever).Nevermind();
    }
}
```

Migration tips

```
{
    try
    {
        // ...
    }
    catch (DataSource::NotFoundException const& e) { /* ... */ }
    catch (DataSource::AccessDeniedException const& e) { /* ... */ }
    catch (DataSource::IOErrorException const& e) { /* ... */ }
    catch (DataSource::TemporaryUnavailableException const& e) { /* ... */ }
    catch (Exception const& e) { /* ... */ }
    catch (std::system_error const& e)
    {
        if (e.code() == BackupError::Critical) { throw; }
        if (e.code() == BackupError::Transient) { /*...*/ }
    }
    catch (std::exception const& e) { /* ... */ }
}
```

Migration tips

```
{
    try
    {
        // ...
    }
    catch (std::exception const&)
    {
        ExceptionFilter().
            On<DataSource::NotFoundException>(/*...*/).
            On<DataSource::TemporaryUnavailableException>(/*...*/).
            Rethrow<DataSource::AccessDenied, DataSource::IOErrorException>().
            Rethrow(BackupError::Critical).
            On(BackupError::Transinet, /*...*/).
            Default(/*...*/);
    }
}
```

Summary

<system_error>

std::error_code
std::error_category
std::error_condition
std::system_error
std::errc std::generic_category

Provides standard approach for error classification and identification without constraining generation mechanism.

THANK YOU!

@solarwinds

Q&A