Designing Robust APIs

How to Write C++ Code that's Safe, Extensible, Efficient & Easy to Use

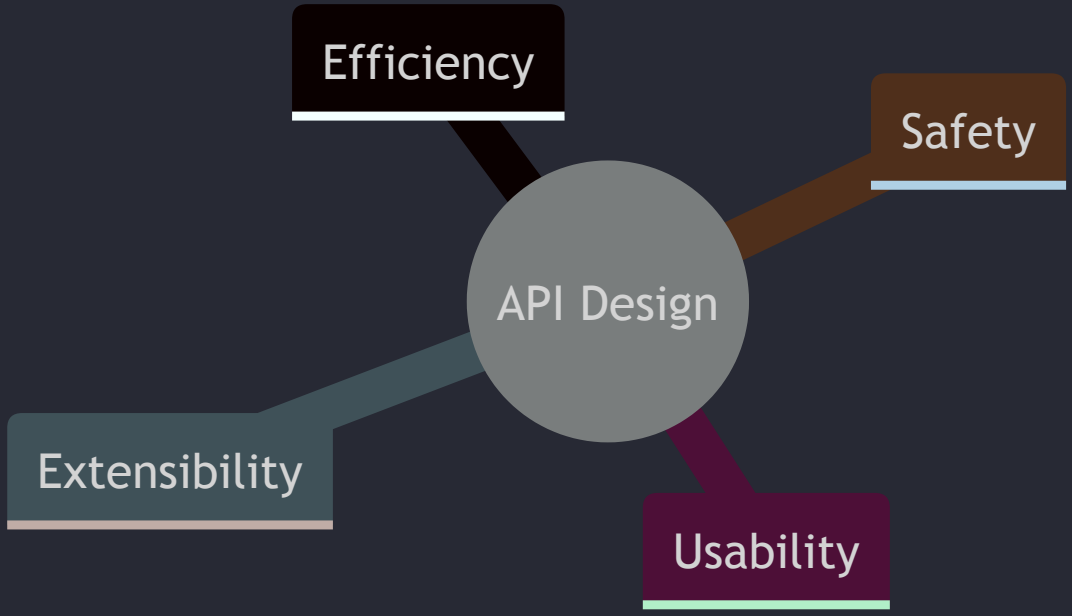
Обо мне

- Пишу на C++ больше 15 лет.
- Основал WG21 Russia в 2016 вместе с @apolukhin.
- В 2016-2019 представлял предложения от РФ в комитете.
- Руководил разработкой поискового движка в Яндексе.
- Руководил инфраструктурой, поиском и ML в Озоне.

Для кого этот доклад?

- Для тех, кто пишет библиотечный код.
- Для тех, чей код так или иначе будет долго жить или широко использоваться.
- Для тех, кто хочет писать код, которым приятно пользоваться.

```
1   class Buffer {
2   public:
3       Buffer(const char *data, size_t size);
4
5       Buffer(const Buffer &buffer)
6           : Data_(nullptr)
7           , Len_(0)
8           , Pos_(0)
9       {
10          *this = buffer;
11      }
12
13      // ...
14  }
```



Хороший API находит баланс между всеми этими аспектами.

Почему это важно?

The Wonderfully Terrible World of C and C++ Encoding APIs:

Feature Set 👇 vs. Library 👉	boost.text	utf8cpp	Standard C	Standard C++	Windows API
Handles Legacy Encodings	❌	❌	😐	😐	✅
Handles UTF Encodings	✅	✅	😐	😐	✅
Bounded and Safe Conversion API	❌	❌	😐	✅	✅
Assumed Valid Conversion API	✅	✅	❌	❌	❌
Unbounded Conversion API	✅	✅	❌	❌	✅
Counting API	❌	😐	❌	❌	✅
Validation API	❌	😐	❌	❌	❌
Bulk Conversions	✅	✅	😐	😐	✅
Single Conversions	✅	✅	✅	✅	❌
Custom Error Handling	❌	✅	✅	✅	❌
Updates Input Range (How Much Read™)	✅	❌	✅	✅	❌
Updates Output Range (How Much Written™)	✅	✅	✅	✅	❌

- *Standard C: it's trash.*
- *Standard C++: provides next-to-nothing of its own that is not sourced from C, and when it does it somehow makes it worse. Also trash.*

Почему это важно?

Exploiting aCropalypse: Recovering Truncated PNGs:

> *SimonTime — 2023-01-02 15:28*
>
> *so basically the pixel 7 pro, when you crop and save a screenshot, overwrites the image with the new version, but leaves the rest of the original file in its place*
>
> *Retr0id — 2023-01-02 15:28*
>
> *ohhhhhhh wow*
>
> *SimonTime — 2023-01-02 15:28*
>
> *so if you were to take a screenshot of an app which shows your address on screen, then crop it, if you could recover the information somehow that's a big deal*
>
> *...*
>
> *IMHO, the takeaway here is that API footguns should be treated as security vulnerabilities.*

См. Back to Basics: C++ API Design - CppCon 2022 by Jason Turner .

Rule #1:

Проектируйте API так, чтобы его нельзя было использовать неправильно

- Программа должна или работать корректно, или завершаться с ошибкой.
- Не должно существовать последовательности вызовов, которая приводит вашу программу в некорректное состояние.
- Чем меньше у вашего API способов завершиться с ошибкой — тем лучше. Зачем обрабатывать ошибки, если можно спроектировать API, в котором их нет?

```cpp
struct CsvStats {
    DateTime startTime;
    DateTime endTime;
    // ...
};

class CsvDb {
public:
    explicit CsvDb(std::string_view path);

    std::string path(std::string_view tableName);

    CsvStats stats(std::string_view tableName);
    void setStats(std::string_view tableName, CsvStats stats);
};
```

```cpp
struct CsvStats {
    DateTime startTime;
    DateTime endTime;   // Some per-table aggregates: start & end time,
    // ...              // maybe per-column min, max & mean values.
};

class CsvDb {
public:
    explicit CsvDb(std::string_view path);

    std::string path(std::string_view tableName);

    CsvStats stats(std::string_view tableName);
    void setStats(std::string_view tableName, CsvStats stats);
};
```

```cpp
struct CsvStats {
    DateTime startTime;
    DateTime endTime;
    // ...
};

class CsvDb {
public:
    explicit CsvDb(std::string_view path); // Path do data folder.

    std::string path(std::string_view tableName);

    CsvStats stats(std::string_view tableName);
    void setStats(std::string_view tableName, CsvStats stats);
};
```

```cpp
struct CsvStats {
    DateTime startTime;
    DateTime endTime;
    // ...
};

class CsvDb {
public:
    explicit CsvDb(std::string_view path);

    std::string path(std::string_view tableName); // Returns path to table CSV for reading
                                                  // OR writing / replacing.
    CsvStats stats(std::string_view tableName);
    void setStats(std::string_view tableName, CsvStats stats);
};
```

```cpp
struct CsvStats {
    DateTime startTime;
    DateTime endTime;
    // ...
};

class CsvDb {
public:
    explicit CsvDb(std::string_view path);

    std::string path(std::string_view tableName);

    CsvStats stats(std::string_view tableName);                 // Getter & setter for stats.
    void setStats(std::string_view tableName, CsvStats stats); // Stats are stored in db.info,
};                                                              // and flushed on setStats.
```

```cpp
struct CsvStats {
    DateTime startTime;
    DateTime endTime;
    // ...
};

class CsvDb {
public:
    explicit CsvDb(std::string_view path);

    std::string path(std::string_view tableName);

    CsvStats stats(std::string_view tableName);
    void setStats(std::string_view tableName, CsvStats stats); // Is this safe?
};
```

```cpp
struct CsvStats {
    DateTime startTime;
    DateTime endTime;
    // ...
};

class CsvDb {
public:
    explicit CsvDb(std::string_view path);

    std::string path(std::string_view tableName);

    CsvStats stats(std::string_view tableName);
    void replace(std::string_view tableName, std::string_view newPath,
                 CsvStats newStats); // Safer!
};
```

```cpp
struct CsvStats {
    DateTime startTime;
    DateTime endTime;
    // ...
};

class CsvDb {
public:
    explicit CsvDb(std::string_view path);

    CsvReader open(std::string_view tableName);

    CsvStats stats(std::string_view tableName);
    CsvWriter replace(std::string_view tableName); // WAY safer!
};
```

```cpp
struct CsvStats {
    DateTime startTime;
    DateTime endTime;
    // ...
};

class CsvDb {
public:
    explicit CsvDb(std::filesystem::path path); // Use std::filesystem::path for paths!

    CsvReader open(std::string_view tableName);

    CsvStats stats(std::string_view tableName);
    CsvWriter replace(std::string_view tableName);
};
```

VS

```cpp
struct CsvStats {
    DateTime startTime;
    DateTime endTime;
    // ...
};

class CsvDb {
public:
    explicit CsvDb(std::string_view path);

    std::string path(std::string_view tableName);

    CsvStats stats(std::string_view tableName);
    void setStats(std::string_view tableName,
                  CsvStats stats);
};
```

```cpp
struct CsvStats {
    DateTime startTime;
    DateTime endTime;
    // ...
};

class CsvDb {
public:
    explicit CsvDb(std::filesystem::path path);

    CsvReader open(std::string_view tableName);
    CsvWriter replace(std::string_view tableName);

    CsvStats stats(std::string_view tableName);

};
```

```
class Window {
public:
    void setTitle(const std::string &title);
    std::string title() const;

    void resize(const Size &size);
    Size size() const;

    void setPosition(const Point &point);
    Point position();

    // ...

    void initOpenGL();
    void bindContext();
    void swapBuffers();
}
```

```cpp
class Window {
public:
    void setTitle(const std::string &title);    //
    std::string title() const;                  //
                                                //
    void resize(const Size &size);              // Window properties ...
    Size size() const;                          //
                                                //
    void setPosition(const Point &point);       //
    Point position();                           //

    // ...

    void initOpenGL();
    void bindContext();
    void swapBuffers();
}
```

```
class Window {
public:
    void setTitle(const std::string &title);
    std::string title() const;

    void resize(const Size &size);
    Size size() const;

    void setPosition(const Point &point);
    Point position();

    // ...

    void initOpenGL(); // Initializes OpenGL context for this Window.
    void bindContext();
    void swapBuffers();
}
```

```cpp
class Window {
public:
    void setTitle(const std::string &title);
    std::string title() const;

    void resize(const Size &size);
    Size size() const;

    void setPosition(const Point &point);
    Point position();

    // ...

    void initOpenGL();
    void bindContext(); // Binds this Window's context for the current thread.
    void swapBuffers();
}
```

```cpp
class Window {
public:
    void setTitle(const std::string &title);
    std::string title() const;

    void resize(const Size &size);
    Size size() const;

    void setPosition(const Point &point);
    Point position();

    // ...

    void initOpenGL();
    void bindContext();
    void swapBuffers(); // Swaps buffers & starts next frame.
}
```

```
class Window {
public:
    void setTitle(const std::string &title);
    std::string title() const;

    void resize(const Size &size);
    Size size() const;

    void setPosition(const Point &point);
    Point position();

    // ...

    void initOpenGL();
    void bindContext(); // OOPS: assert(_glInitialized); inside.
    void swapBuffers(); // Can only be called after a call to initOpenGL().
}
```

```
class Window {
public:
    void setTitle(const std::string &title);
    std::string title() const;

    void resize(const Size &size);
    Size size() const;

    void setPosition(const Point &point);
    Point position();

    // ...

    std::unique_ptr<OpenGLContext> createOpenGLContext();
};

class OpenGLContext {
public:
    void bind();
    void swapBuffers();

    // ...
}
```

Rule #2:

Divide & Conquer: Дробите!

- Помните про S in SOLID.

  "The Single-responsibility principle: There should never be more than one reason for a class to change."
- Количество возможных взаимодействий (и багов) внутри класса растет как квадрат от размера класса. Меньше классы — меньше проблем!
- Мелкие классы легче читать и осознавать.
- Аналогично работает и с библиотеками. Классы в "помойке классов" начинают зависеть друг от друга, снова квадрат зависимостей и баги. Дробите на более мелкие библиотеки!
- Эта же логика применима к функциям.

```cpp
void parseStrings(const Buffer &buffer, std::vector<std::string> *result) { // Parses a buffer of null-terminated
    size_t pos = 0;                                                          // strings.
    while (pos < buffer.size()) {
        const char *nextPos = static_cast<const char *>(memchr(buffer.data() + pos, '\0', buffer.size() - pos));
        size_t size = (nextPos ? nextPos - buffer.data() : buffer.size()) - pos;
        std::string str = std::string(buffer.data() + pos, size);

        if (str.size() >= 2 && str.front() == '"' && str.back() == '"')
            str = str.substr(1, str.size() - 2);

        result->push_back(std::move(str));
        pos += size;
    }
}
```

```cpp
void parseStrings(const Buffer &buffer, std::vector<std::string> *result) {
    size_t pos = 0;
    while (pos < buffer.size()) { // Find next null terminator ↓
        const char *nextPos = static_cast<const char *>(memchr(buffer.data() + pos, '\0', buffer.size() - pos));
        size_t size = (nextPos ? nextPos - buffer.data() : buffer.size()) - pos;
        std::string str = std::string(buffer.data() + pos, size);

        if (str.size() >= 2 && str.front() == '"' && str.back() == '"')
            str = str.substr(1, str.size() - 2);

        result->push_back(std::move(str));
        pos += size;
    }
}
```

```cpp
void parseStrings(const Buffer &buffer, std::vector<std::string> *result) {
    size_t pos = 0;
    while (pos < buffer.size()) {
        const char *nextPos = static_cast<const char *>(memchr(buffer.data() + pos, '\0', buffer.size() - pos));
        size_t size = (nextPos ? nextPos - buffer.data() : buffer.size()) - pos;
        std::string str = std::string(buffer.data() + pos, size); // Extract next string.

        if (str.size() >= 2 && str.front() == '"' && str.back() == '"')
            str = str.substr(1, str.size() - 2);

        result->push_back(std::move(str));
        pos += size;
    }
}
```

```
void parseStrings(const Buffer &buffer, std::vector<std::string> *result) {
    size_t pos = 0;
    while (pos < buffer.size()) {
        const char *nextPos = static_cast<const char *>(memchr(buffer.data() + pos, '\0', buffer.size() - pos));
        size_t size = (nextPos ? nextPos - buffer.data() : buffer.size()) - pos;
        std::string str = std::string(buffer.data() + pos, size);

        if (str.size() >= 2 && str.front() == '"' && str.back() == '"')
            str = str.substr(1, str.size() - 2); // Unquote a string if it's quoted.

        result->push_back(std::move(str));
        pos += size;
    }
}
```

```cpp
void parseStrings(const Buffer &buffer, std::vector<std::string> *result) {
    size_t pos = 0;
    while (pos < buffer.size()) {
        const char *nextPos = static_cast<const char *>(memchr(buffer.data() + pos, '\0', buffer.size() - pos));
        size_t size = (nextPos ? nextPos - buffer.data() : buffer.size()) - pos;
        std::string str = std::string(buffer.data() + pos, size);

        if (str.size() >= 2 && str.front() == '"' && str.back() == '"')
            str = str.substr(1, str.size() - 2);

        result->push_back(std::move(str)); // Write results &
        pos += size;                       // prepare to parse the next string.
    }
}
```

```
void parseStrings(const Buffer &buffer, std::vector<std::string> *result) {
    size_t pos = 0;
    while (pos < buffer.size()) {
        const char *nextPos = static_cast<const char *>(memchr(buffer.data() + pos, '\0', buffer.size() - pos));
        size_t size = (nextPos ? nextPos - buffer.data() : buffer.size()) - pos;
        std::string str = std::string(buffer.data() + pos, size);

        if (str.size() >= 2 && str.front() == '"' && str.back() == '"')
            str = str.substr(1, str.size() - 2);

        result->push_back(std::move(str));
        pos += size; // BUG: should be pos += size + 1.
    }
}
```

```cpp
void parseStrings(const Buffer &buffer, std::vector<std::string> *result) {
    size_t pos = 0;
    while (pos < buffer.size()) {
        const char *nextPos = static_cast<const char *>(memchr(buffer.data() + pos, '\0', buffer.size() - pos));
        size_t size = (nextPos ? nextPos - buffer.data() : buffer.size()) - pos;
        std::string str = std::string(buffer.data() + pos, size);

        if (str.size() >= 2 && str.front() == '"' && str.back() == '"')
            str = str.substr(1, str.size() - 2);

        result->push_back(std::move(str));
        pos += size + 1;
    }
}
```

VS

```cpp
std::vector<std::string> parseStrings(const Buffer &buffer, std::vector<std::string> *result) {
    MemoryInput input(buffer);

    std::string line;
    while (input.readLine(&line, '\0'))
        result->push_back(unquote(line));
}
```

```cpp
void parseStrings(const Buffer &buffer, std::vector<std::string> *result) {
    size_t pos = 0;
    while (pos < buffer.size()) {
        const char *nextPos = static_cast<const char *>(memchr(buffer.data() + pos, '\0', buffer.size() - pos));
        size_t size = (nextPos ? nextPos - buffer.data() : buffer.size()) - pos;
        std::string str = std::string(buffer.data() + pos, size);

        if (str.size() >= 2 && str.front() == '"' && str.back() == '"')
            str = str.substr(1, str.size() - 2);

        result->push_back(std::move(str));
        pos += size + 1;
    }
}
```

VS

```cpp
std::vector<std::string> parseStrings(const Buffer &buffer, std::vector<std::string> *result) {
    MemoryInput input(buffer);

    std::string line;
    while (input.readLine(&line, '\0'))
        result->push_back(unquote(line));
}
```

```cpp
void parseStrings(const Buffer &buffer, std::vector<std::string> *result) {
    size_t pos = 0;
    while (pos < buffer.size()) {
        const char *nextPos = static_cast<const char *>(memchr(buffer.data() + pos, '\0', buffer.size() - pos));
        size_t size = (nextPos ? nextPos - buffer.data() : buffer.size()) - pos;
        std::string str = std::string(buffer.data() + pos, size);

        if (str.size() >= 2 && str.front() == '"' && str.back() == '"')
            str = str.substr(1, str.size() - 2);

        result->push_back(std::move(str));
        pos += size + 1;
    }
}
```

VS

```cpp
std::vector<std::string> parseStrings(const Buffer &buffer, std::vector<std::string> *result) {
    MemoryInput input(buffer);

    std::string line;
    while (input.readLine(&line, '\0'))
        result->push_back(unquote(line));
}
```

Rule #2:

Divide & Conquer: Дробите!

- Как только видите сложный код — думайте, каких абстракций вам не хватает!
- Дробить — это еще и про слои абстракции!
- Если ваш код все еще сложный — дробите дальше!

```cpp
template<class T>
class QFuture {
public:
    QFuture(const QFuture &other);

    const_iterator begin() const;
    const_iterator end() const;
    QList<T> results() const;

    void cancel();
    T takeResult();
    T result() const;

    auto then(Function &&function);
    auto then(QThreadPool *pool, Function &&function);
    auto then(QObject *context, Function &&function);

    bool isCanceled() const;
    bool isFinished() const;
    bool isRunning() const;
    bool isStarted() const;
    bool isValid() const;

    // ...
};
```

```cpp
template<class T>
class QFuture {
public:
    QFuture(const QFuture &other); // QFuture is ref-counted.

    const_iterator begin() const;
    const_iterator end() const;
    QList<T> results() const;

    void cancel();
    T takeResult();
    T result() const;

    auto then(Function &&function);
    auto then(QThreadPool *pool, Function &&function);
    auto then(QObject *context, Function &&function);

    bool isCanceled() const;
    bool isFinished() const;
    bool isRunning() const;
    bool isStarted() const;
    bool isValid() const;

    // ...
};
```

```cpp
template<class T>
class QFuture {
public:
    QFuture(const QFuture &other);

    const_iterator begin() const; //
    const_iterator end() const;   // Async sequence interface.
    QList<T> results() const;     //

    void cancel();
    T takeResult();
    T result() const;

    auto then(Function &&function);
    auto then(QThreadPool *pool, Function &&function);
    auto then(QObject *context, Function &&function);

    bool isCanceled() const;
    bool isFinished() const;
    bool isRunning() const;
    bool isStarted() const;
    bool isValid() const;

    // ...
}
```

```cpp
template<class T>
class QFuture {
public:
    QFuture(const QFuture &other);

    const_iterator begin() const;
    const_iterator end() const;
    QList<T> results() const;

    void cancel(); // Built-in cancellation support! ^_^
    T takeResult();
    T result() const;

    auto then(Function &&function);
    auto then(QThreadPool *pool, Function &&function);
    auto then(QObject *context, Function &&function);

    bool isCanceled() const;
    bool isFinished() const;
    bool isRunning() const;
    bool isStarted() const;
    bool isValid() const;

    // ...
}
```

```cpp
template<class T>
class QFuture {
public:
    QFuture(const QFuture &other);

    const_iterator begin() const;
    const_iterator end() const;
    QList<T> results() const;

    void cancel();
    T takeResult(); // Waits for completion & moves out the result. Support for move-only types.
    T result() const;

    auto then(Function &&function);
    auto then(QThreadPool *pool, Function &&function);
    auto then(QObject *context, Function &&function);

    bool isCanceled() const;
    bool isFinished() const;
    bool isRunning() const;
    bool isStarted() const;
    bool isValid() const;

    // ...
}
```

```
template<class T>
class QFuture {
public:
    QFuture(const QFuture &other);

    const_iterator begin() const;
    const_iterator end() const;
    QList<T> results() const;

    void cancel();
    T takeResult();
    T result() const; // Waits for completion & returns a copy of result.

    auto then(Function &&function);
    auto then(QThreadPool *pool, Function &&function);
    auto then(QObject *context, Function &&function);

    bool isCanceled() const;
    bool isFinished() const;
    bool isRunning() const;
    bool isStarted() const;
    bool isValid() const;

    // ...
}
```

```cpp
template<class T>
class QFuture {
public:
    QFuture(const QFuture &other);

    const_iterator begin() const;
    const_iterator end() const;
    QList<T> results() const;

    void cancel();
    T takeResult();
    T result() const;

    auto then(Function &&function);                    //
    auto then(QThreadPool *pool, Function &&function); // Continuations.
    auto then(QObject *context, Function &&function);  //

    bool isCanceled() const;
    bool isFinished() const;
    bool isRunning() const;
    bool isStarted() const;
    bool isValid() const;

    // ...
}
```

```cpp
template<class T>
class QFuture {
public:
    QFuture(const QFuture &other);

    const_iterator begin() const;
    const_iterator end() const;
    QList<T> results() const;

    void cancel();
    T takeResult();
    T result() const;

    auto then(Function &&function); // Where is this one run? It depends ...
    auto then(QThreadPool *pool, Function &&function);
    auto then(QObject *context, Function &&function);

    bool isCanceled() const;
    bool isFinished() const;
    bool isRunning() const;
    bool isStarted() const;
    bool isValid() const;

    // ...
}
```

```cpp
template<class T>
class QFuture {
public:
    QFuture(const QFuture &other);

    const_iterator begin() const;
    const_iterator end() const;
    QList<T> results() const;

    void cancel();
    T takeResult();
    T result() const;

    auto then(Function &&function);
    auto then(QThreadPool *pool, Function &&function);
    auto then(QObject *context, Function &&function); // Unlike QObject::connect, this WILL CRASH
                                                      // if you delete the context object.
    bool isCanceled() const;
    bool isFinished() const;
    bool isRunning() const;
    bool isStarted() const;
    bool isValid() const;

    // ...
}
```

```cpp
template<class T>
class QFuture {
public:
    QFuture(const QFuture &other);

    const_iterator begin() const;
    const_iterator end() const;
    QList<T> results() const;

    void cancel();
    T takeResult();
    T result() const;

    auto then(Function &&function);
    auto then(QThreadPool *pool, Function &&function);
    auto then(QObject *context, Function &&function);

    bool isCanceled() const; //
    bool isFinished() const; //
    bool isRunning() const;  // State observers (not all of them).
    bool isStarted() const;  //
    bool isValid() const;    //

    // ...
}
```

```cpp
template<class T>
class QFuture {
public:
    QFuture(const QFuture &other);

    const_iterator begin() const;
    const_iterator end() const;
    QList<T> results() const;

    void cancel();
    T takeResult();
    T result() const;

    auto then(Function &&function);
    auto then(QThreadPool *pool, Function &&function);
    auto then(QObject *context, Function &&function);

    bool isCanceled() const;
    bool isFinished() const;
    bool isRunning() const;  // How are these two different?
    bool isStarted() const;  // Gotta read the sources to figure out ...
    bool isValid() const;

    // ...
}
```

```cpp
template<class T>
class QFuture {
public:
    QFuture(const QFuture &other);

    const_iterator begin() const;
    const_iterator end() const;
    QList<T> results() const;

    void cancel();
    T takeResult();
    T result() const;

    auto then(Function &&function);
    auto then(QThreadPool *pool, Function &&function);
    auto then(QObject *context, Function &&function);

    bool isCanceled() const; //
    bool isFinished() const; // Why do we even have all these state observers?
    bool isRunning() const;  // Why not a single State state() function?
    bool isStarted() const;  //
    bool isValid() const;    //

    // ...
}
```

```cpp
template<class T>
class QWhatExactlyIsThisMonstrosity /* ¯\_(ツ)_/¯ */ {
public:
    QWhatExactlyIsThisMonstrosity(const QWhatExactlyIsThisMonstrosity &other); // Shared future?

    const_iterator begin() const;                       //
    const_iterator end() const;                         // Async sequence?
    QList<T> results() const;                           //

    void cancel();
    T takeResult();                                     // Not so shared future?
    T result() const;                                   // Shared future?

    auto then(Function &&function);
    auto then(QThreadPool *pool, Function &&function);
    auto then(QObject *context, Function &&function);   // (┘°□°)┘ ⌇ ┻┻

    bool isCanceled() const;
    bool isFinished() const;
    bool isRunning() const;
    bool isStarted() const;                             // (┘◉▃◉)┘ ‿ ┻┻
    bool isValid() const;

    // P.S.: Don't even try to read the sources. They'll give you nightmares.
}
```

Rule #3:

Тратьте время на придумывание хороших имен!

"There are only two hard things in Computer Science: cache invalidation and naming things." — Phil Karlton

“Clean code reads like well-written prose.” — Robert C. Martin

- Если что-то не получается назвать нормально — значит вы хотите странного, переделывайте свой дизайн.
- Не поддавайтесь соблазну назвать класс `HandlerHelper`. Все, что заканчивается на `Helper` — это признание поражения.
- Думайте над тем, в чем вообще концептуальная суть ваших абстракций.

  "Что такое future?"
- Пользуйтесь thesaurus.com и ChatGPT.

```cpp
template<class T>
class QFuturePart1 {
public:
    void cancel();
    const_iterator begin() const;
    const_iterator end() const;
    QList<T> results() const;
    // ...
}

template<class T>
class QFuturePart2 {
public:
    QFuture(const QFuture &other);
    void cancel();
    T result() const;
    // ...
}

template<class T>
class QFuturePart3 {
public:
    QFuture(QFuture &&other);
    void cancel();
    T takeResult();
    // ...
}
```

```cpp
template<class T>
class QGenerator {
public:
    void cancel();
    const_iterator begin() const;
    const_iterator end() const;
    QList<T> results() const;
    // ...
}

template<class T>
class QSharedFuture {
public:
    QFuture(const QFuture &other);
    void cancel();
    T result() const;
    // ...
}

template<class T>
class QUniqueFuture {
public:
    QFuture(QFuture &&other);
    void cancel();
    T takeResult();
    // ...
}
```

Rule #2 + Rule #3:

Дробление и нейминг — это про абстракции, а не про классы!

- Ваш код — это перевод абстракций в вашей голове на C++.
- Если в голове нет порядка, то в коде его тоже не будет.

Что такое future?

- Это абстракция над вычислением.
- Но это и абстракция над контекстом, в котором это вычисление производится!

```
template<class T>
class QSharedFuture {
public:
    auto then(Function &&function); // Where will it run?
                                    // Can I pass in a function that does a lot of work?
    // ...                          // Can we do better?
};
```

```
1
2   Task<TwitterPosts> fetchTwitterPosts(Network &network, const TwitterFetchOptions &opts) {
3       return network
4           .request(makeNetworkRequest(opts))
5           .then([](std::string_view jsonData) {
6               TwitterPosts result;
7               deserialize(Json::parse(jsonData), &result);
8               return result;
9           });
10  }
11
12  void myAwesomeFunction(Network &network) {
13      TwitterPosts posts = fetchTwitterPosts(network, TwitterFetchOptions("@stroustrup", 20))
14          .run(globalThreadPool())
15          .join();
16      std::print("{}", posts);
17  }
```

```cpp
// Returns a Task that will fetch twitter posts with the provided options when run.
Task<TwitterPosts> fetchTwitterPosts(Network &network, const TwitterFetchOptions &opts) {
    return network
        .request(makeNetworkRequest(opts))
        .then([](std::string_view jsonData) {
            TwitterPosts result;
            deserialize(Json::parse(jsonData), &result);
            return result;
        });
}

void myAwesomeFunction(Network &network) {
    TwitterPosts posts = fetchTwitterPosts(network, TwitterFetchOptions("@stroustrup", 20))
        .run(globalThreadPool())
        .join();
    std::print("{}", posts);
}
```

```
Task<TwitterPosts> fetchTwitterPosts(Network &network, const TwitterFetchOptions &opts) {
    return network
        .request(makeNetworkRequest(opts)) // Network::request() also returns a Task!
        .then([](std::string_view jsonData) {
            TwitterPosts result;
            deserialize(Json::parse(jsonData), &result);
            return result;
        });
}

void myAwesomeFunction(Network &network) {
    TwitterPosts posts = fetchTwitterPosts(network, TwitterFetchOptions("@stroustrup", 20))
        .run(globalThreadPool())
        .join();
    std::print("{}", posts);
}
```

```
Task<TwitterPosts> fetchTwitterPosts(Network &network, const TwitterFetchOptions &opts) {
    return network
        .request(makeNetworkRequest(opts))
        .then([](std::string_view jsonData) { // Tasks are composable!
            TwitterPosts result;
            deserialize(Json::parse(jsonData), &result);
            return result;
        });
}

void myAwesomeFunction(Network &network) {
    TwitterPosts posts = fetchTwitterPosts(network, TwitterFetchOptions("@stroustrup", 20))
        .run(globalThreadPool())
        .join();
    std::print("{}", posts);
}
```

```
Task<TwitterPosts> fetchTwitterPosts(Network &network, const TwitterFetchOptions &opts) {
    return network
        .request(makeNetworkRequest(opts))
        .then([](std::string_view jsonData) {
            TwitterPosts result;
            deserialize(Json::parse(jsonData), &result);
            return result;
        });
}

void myAwesomeFunction(Network &network) { // Fetch Bjarne's 20 latests posts ↓
    TwitterPosts posts = fetchTwitterPosts(network, TwitterFetchOptions("@stroustrup", 20))
        .run(globalThreadPool())
        .join();
    std::print("{}", posts);
}
```

```
1
2  Task<TwitterPosts> fetchTwitterPosts(Network &network, const TwitterFetchOptions &opts) {
3      return network
4          .request(makeNetworkRequest(opts))
5          .then([](std::string_view jsonData) {
6              TwitterPosts result;
7              deserialize(Json::parse(jsonData), &result);
8              return result;
9          });
10 }
11
12 void myAwesomeFunction(Network &network) {
13     TwitterPosts posts = fetchTwitterPosts(network, TwitterFetchOptions("@stroustrup", 20))
14         .run(globalThreadPool()) // Run in the global thread pool. Can do run(thisThread()) instead!
15         .join();
16     std::print("{}", posts);
17 }
```

```
1
2  Task<TwitterPosts> fetchTwitterPosts(Network &network, const TwitterFetchOptions &opts) {
3      return network
4          .request(makeNetworkRequest(opts))
5          .then([](std::string_view jsonData) {
6              TwitterPosts result;
7              deserialize(Json::parse(jsonData), &result);
8              return result;
9          });
10 }
11
12 void myAwesomeFunction(Network &network) {
13     TwitterPosts posts = fetchTwitterPosts(network, TwitterFetchOptions("@stroustrup", 20))
14         .run(globalThreadPool())
15         .join(); // Wait for completion.
16     std::print("{}", posts);
17 }
```

```
Task<TwitterPosts> fetchTwitterPosts(Network &network, const TwitterFetchOptions &opts) {
    return network
        .request(makeNetworkRequest(opts))
        .then([](std::string_view jsonData) {
            TwitterPosts result;
            deserialize(Json::parse(jsonData), &result);
            return result;
        });
}

void myAwesomeFunction(Network &network) {
    TwitterPosts posts = fetchTwitterPosts(network, TwitterFetchOptions("@stroustrup", 20))
        .run(globalThreadPool())
        .join();
    std::print("{}", posts); // Print it out.
}
```

```
1
2   Task<TwitterPosts> fetchTwitterPosts(Network &network, const TwitterFetchOptions &opts) {
3       return network
4           .request(makeNetworkRequest(opts))
5           .then([](std::string_view jsonData) {
6               TwitterPosts result;
7               deserialize(Json::parse(jsonData), &result);
8               return result;
9           });
10  }
11
12  void myAwesomeFunction(Network &network) {
13      TwitterPosts posts = fetchTwitterPosts(network, TwitterFetchOptions("@stroustrup", 20))
14          .run(globalThreadPool())
15          .join();
16      std::print("{}", posts);
17  }
```

- Task и контекст выполнения ортогональны!

См. Working with Asynchrony Generically: A Tour of C++ Executors - CppCon 21 by Eric Niebler.

Rule #4:

Создавайте ортогональные и взаимозаменяемые абстракции

- Дробите абстракции на взаимозаменяемые кусочки.
  ⇒ Реализуйте N+M вариантов вашего кода вместо N×M.
- Вдохновляйтесь STL. Ортогональность данных и алгоритмов можно успешно распространить на очень многие предметные области!
- Если у вас есть два схожих концепта — думайте, должны ли они быть взаимозаменяемы. Выбирайте подходящий под вашу задачу тип полиморфизма:
  - Виртуальные функции для динамического полиморфизма.
  - Overload sets для статического полиморфизма.

```cpp
// Static polymorphism.

struct TwitterPost {
    std::string author;
    std::string text;
};

struct TwitterUser {
    // ...
};

void deserialize(const Json &json, TwitterPost *value) {
    value->author = json["author"];
    value->text = json["text"];
}

void deserialize(const Json &json, TwitterUser *value) {
    // ...
}
```

VS

```cpp
// Static polymorphism.

struct TwitterPost {
    std::string author;
    std::string text;
};

struct TwitterUser {
    // ...
};

void deserialize(const Json &json, TwitterPost *value) {
    value->author = json["author"];
    value->text = json["text"];
}

void deserialize(const Json &json, TwitterUser *value) {
    // ...
}
```

```cpp
// Dynamic polymorphism.

class Deserializable {
public:
    virtual void deserialize(const Json &json) = 0;

protected:
    ~Deserializable() = default;
};

struct TwitterPost : Deserializable {
    std::string author;
    std::string text;

    virtual void deserialize(const Json &json) override {
        author = json["author"];
        text = json["text"];
    }
};

struct TwitterUser : Deserializable {
    // ...

    virtual void deserialize(const Json &json) override {
        // ...
    }
};
```


[123456789] Move - Examine [+-] monster or [*_] spot - [Z] Abort
>> You see... <<
A tunnel.
Yug-Saggath, the great blue wyrm.
Hostile. Somewhat experienced. Not
injured. [M]onster memory.
Ig skel St:39 Le:26 Wi:20 Dx:25 To:41 Ch:29 Ap:14 Ma:19 Pe:20 N=
DV/PV:105/44 H:584(587) P:84(136) Exp:34/7636445 MT Sp:126
Blessed Satiated Burdened Coward

```cpp
class Equipment {
public:
    virtual void onUse() = 0;
    //  ...
};
```

```cpp
class Equipment {
public:
    virtual void onUse() = 0;
    //  ...
};

class Weapon : public Equipment {
public:
    virtual void onAttack(Monster &monster) = 0;
    //  ...
};
```

```cpp
class Equipment {
public:
    virtual void onUse() = 0;
    // ...
};

class Weapon : public Equipment {
public:
    virtual void onAttack(Monster &monster) = 0;
    // ...
};

class VampiricSword : public Weapon {
public:
    virtual void onAttack(Monster &monster) override {
        Damage damage(DMG_PHYSICAL, rollDice(3, 5) + 5);
        monster.takeDamage(damage);
        Damage healing(DMG_DARKMAGIC, damage.amount / 2);
        owner().heal(healing);
    }
    // ...
};
```

```
class Equipment {
public:
    virtual void onUse() = 0;
    // ...
};

class Weapon : public Equipment {
public:
    virtual void onAttack(Monster &monster) = 0;
    // ...
};

class VampiricSword : public Weapon {
public:
    virtual void onAttack(Monster &monster) override {
        Damage damage(DMG_PHYSICAL, rollDice(3, 5) + 5);  // Create a damage instance for 3d5+5.
        monster.takeDamage(damage);
        Damage healing(DMG_DARKMAGIC, damage.amount / 2);
        owner().heal(healing);
    }
    // ...
};
```

```
class Equipment {
public:
    virtual void onUse() = 0;
    // ...
};

class Weapon : public Equipment {
public:
    virtual void onAttack(Monster &monster) = 0;
    // ...
};

class VampiricSword : public Weapon {
public:
    virtual void onAttack(Monster &monster) override {
        Damage damage(DMG_PHYSICAL, rollDice(3, 5) + 5);
        monster.takeDamage(damage);                    // Send it to monster, this call updates damage.
        Damage healing(DMG_DARKMAGIC, damage.amount / 2);
        owner().heal(healing);
    }
    // ...
};
```

```cpp
class Equipment {
public:
    virtual void onUse() = 0;
    // ...
};

class Weapon : public Equipment {
public:
    virtual void onAttack(Monster &monster) = 0;
    // ...
};

class VampiricSword : public Weapon {
public:
    virtual void onAttack(Monster &monster) override {
        Damage damage(DMG_PHYSICAL, rollDice(3, 5) + 5);
        monster.takeDamage(damage);
        Damage healing(DMG_DARKMAGIC, damage.amount / 2); // Heal the player for half the damage dealt.
        owner().heal(healing);                            // Also, it's DARKMAGIC!
    }
    // ...
};
```

```cpp
class Equipment {
public:
    virtual void onUse() = 0;
    // ...
};

class Weapon : public Equipment {
public:
    virtual void onAttack(Monster &monster) = 0;
    // ...
};

class VampiricSword : public Weapon {
public:
    virtual void onAttack(Monster &monster) override {
        Damage damage(DMG_PHYSICAL, rollDice(3, 5) + 5);
        monster.takeDamage(damage);
        Damage healing(DMG_DARKMAGIC, damage.amount / 2);
        owner().heal(healing);
    }
    // ...
};

class Shield : public Equipment {
public:
    virtual void onTakeDamage(Damage &damage) = 0;
    // ...
};
```

```cpp
class Equipment {
public:
    virtual void onUse() = 0;
    // ...
};

class Weapon : public Equipment {
public:
    virtual void onAttack(Monster &monster) = 0;
    // ...
};

class VampiricSword : public Weapon {
public:
    virtual void onAttack(Monster &monster) override {
        Damage damage(DMG_PHYSICAL, rollDice(3, 5) + 5);
        monster.takeDamage(damage);
        Damage healing(DMG_DARKMAGIC, damage.amount / 2);
        owner().heal(healing);
    }
    // ...
};

class Shield : public Equipment {
public:
    virtual void onTakeDamage(Damage &damage) = 0;
    // ...
};

class SpikedShield : public Weapon, public Shield { // Eeeeeeh?????????
    // ...
};
```

```
class Event { // Every interaction in the game is an event, or several events.
public:
    EventType type;
    // ...
}
```

```
class Event {
public:
    EventType type;
    // ...
}

class Behaviour { // Behaviours are composable pieces of event-handling logic.
public:
    virtual void process(Event *event) = 0;
    // ...
}
```

```cpp
class Event {
public:
    EventType type;
    // ...
}

class Behaviour {
public:
    virtual void process(Event *event) = 0;
    // ...
}

class Equipment { // No subclassing,
public:           // all logic is in behaviours ↓
    std::vector<std::unique_ptr<Behaviour>> behaviours;
    // ...
};
```

```cpp
// First event: sent to player's items to populate the damage rolls.
class AttackOutEvent : public Event {
public:
    std::vector<Damage> damageRolls;
    // ...
};

// Second event: sent to monster's items to apply armor & protection.
class AttackInEvent : public Event {
public:
    std::vector<Damage> damageRolls;
    // ...
};

// Third event: sent back to player's items to notify of success / failure.
class DamageEvent : public Event {
public:
    std::vector<Damage> damageRolls;
    // ...
};
```

```
class WeaponBehavior : public Behaviour {
    virtual void process(Event *event) override {
        if (event->type != ATTACK_OUT_EVENT)
            return;
        AttackOutEvent *e = static_cast<AttackOutEvent *>(event);

        e->damageRolls.push_back(Damage(
            owner(),
            DMG_PHYSICAL,
            owner().rollAttack()
        ));
    }
}

class ArmorBehavior : public Behaviour {
    virtual void process(Event *event) override {
        if (event->type != ATTACK_IN_EVENT)
            return;
        AttackInEvent *e = static_cast<AttackInEvent *>(event);

        for (Damage &damage : e->damageRolls)
            if (damage.type == DMG_PHYSICAL)
                damage.amount = std::max(0, damage.amount - owner().armorClass());
    }
};
```

```
class WeaponBehavior : public Behaviour {
    virtual void process(Event *event) override {
        if (event->type != ATTACK_OUT_EVENT) // Process only outgoing attack events.
            return;
        AttackOutEvent *e = static_cast<AttackOutEvent *>(event);

        e->damageRolls.push_back(Damage(
            owner(),
            DMG_PHYSICAL,
            owner().rollAttack()
        ));
    }
}

class ArmorBehavior : public Behaviour {
    virtual void process(Event *event) override {
        if (event->type != ATTACK_IN_EVENT)
            return;
        AttackInEvent *e = static_cast<AttackInEvent *>(event);

        for (Damage &damage : e->damageRolls)
            if (damage.type == DMG_PHYSICAL)
                damage.amount = std::max(0, damage.amount - owner().armorClass());
    }
};
```

```
class WeaponBehavior : public Behaviour {
    virtual void process(Event *event) override {
        if (event->type != ATTACK_OUT_EVENT)
            return;
        AttackOutEvent *e = static_cast<AttackOutEvent *>(event);

        e->damageRolls.push_back(Damage( // Record a damage roll
            owner(),
            DMG_PHYSICAL,
            owner().rollDamage()
        ));
    }
}

class ArmorBehavior : public Behaviour {
    virtual void process(Event *event) override {
        if (event->type != ATTACK_IN_EVENT)
            return;
        AttackInEvent *e = static_cast<AttackInEvent *>(event);

        for (Damage &damage : e->damageRolls)
            if (damage.type == DMG_PHYSICAL)
                damage.amount = std::max(0, damage.amount - owner().armorClass());
    }
};
```

```
class WeaponBehavior : public Behaviour {
    virtual void process(Event *event) override {
        if (event->type != ATTACK_OUT_EVENT)
            return;
        AttackOutEvent *e = static_cast<AttackOutEvent *>(event);

        e->damageRolls.push_back(Damage(
            owner(),
            DMG_PHYSICAL,
            owner().rollDamage()
        ));
    }
}

class ArmorBehavior : public Behaviour {
    virtual void process(Event *event) override {
        if (event->type != ATTACK_IN_EVENT) // Process only incoming attack events.
            return;
        AttackInEvent *e = static_cast<AttackInEvent *>(event);

        for (Damage &damage : e->damageRolls)
            if (damage.type == DMG_PHYSICAL)
                damage.amount = std::max(0, damage.amount - owner().armorClass());
    }
};
```

```
class WeaponBehavior : public Behaviour {
    virtual void process(Event *event) override {
        if (event->type != ATTACK_OUT_EVENT)
            return;
        AttackOutEvent *e = static_cast<AttackOutEvent *>(event);

        e->damageRolls.push_back(Damage(
            owner(),
            DMG_PHYSICAL,
            owner().rollDamage()
        ));
    }
}

class ArmorBehavior : public Behaviour {
    virtual void process(Event *event) override {
        if (event->type != ATTACK_IN_EVENT)
            return;
        AttackInEvent *e = static_cast<AttackInEvent *>(event);

        for (Damage &damage : e->damageRolls) // Apply protection against physical damage.
            if (damage.type == DMG_PHYSICAL)
                damage.amount = std::max(0, damage.amount - owner().armorClass());
    }
};
```

```
class VampiricBehavior : public Behaviour {
    virtual void process(Event *event) override {
        if (event->type != DAMAGE_EVENT)
            return;
        DamageEvent *e = static_cast<DamageEvent *>(event);

        int totalDamage = 0;
        for (const Damage &damage : e->damageRolls) {
            if (damage.source == owner() && damage.type == DMG_PHYSICAL)
                totalDamage += damage.amount;

        if (totalDamage <= 1)
            return;

        sendEvent(
            owner().owner(),
            SpellEvent(
                owner(),
                SPELL_VAMPIRIC_HEALING,
                totalDamage / 2
            )
        );
    }
}
```

```
class VampiricBehavior : public Behaviour {
    virtual void process(Event *event) override {
        if (event->type != DAMAGE_EVENT) // Handle only damage notification events.
            return;
        DamageEvent *e = static_cast<DamageEvent *>(event);

        int totalDamage = 0;
        for (const Damage &damage : e->damageRolls) {
            if (damage.source == owner() && damage.type == DMG_PHYSICAL)
                totalDamage += damage.amount;

        if (totalDamage <= 1)
            return;

        sendEvent(
            owner().owner(),
            SpellEvent(
                owner(),
                SPELL_VAMPIRIC_HEALING,
                totalDamage / 2
            )
        );
    }
}
```

```
class VampiricBehavior : public Behaviour {
    virtual void process(Event *event) override {
        if (event->type != DAMAGE_EVENT)
            return;
        DamageEvent *e = static_cast<DamageEvent *>(event);

        int totalDamage = 0; // Calculate total physical damage dealt by this weapon.
        for (const Damage &damage : e->damageRolls) {
            if (damage.source == owner() && damage.type == DMG_PHYSICAL)
                totalDamage += damage.amount;

        if (totalDamage <= 1)
            return;

        sendEvent(
            owner().owner(),
            SpellEvent(
                owner(),
                SPELL_VAMPIRIC_HEALING,
                totalDamage / 2
            )
        );
    }
}
```

```
class VampiricBehavior : public Behaviour {
    virtual void process(Event *event) override {
        if (event->type != DAMAGE_EVENT)
            return;
        DamageEvent *e = static_cast<DamageEvent *>(event);

        int totalDamage = 0;
        for (const Damage &damage : e->damageRolls) {
            if (damage.source == owner() && damage.type == DMG_PHYSICAL)
                totalDamage += damage.amount;

        if (totalDamage < 2) // Vampirism kicks in at totalDamage ⩾ 2.
            return;

        sendEvent(
            owner().owner(),
            SpellEvent(
                owner(),
                SPELL_VAMPIRIC_HEALING,
                totalDamage / 2
            )
        );
    }
}
```

```
class VampiricBehavior : public Behaviour {
    virtual void process(Event *event) override {
        if (event->type != DAMAGE_EVENT)
            return;
        DamageEvent *e = static_cast<DamageEvent *>(event);

        int totalDamage = 0;
        for (const Damage &damage : e->damageRolls) {
            if (damage.source == owner() && damage.type == DMG_PHYSICAL)
                totalDamage += damage.amount;

        if (totalDamage <= 1)
            return;

        sendEvent(              // Send vampiric healing event to the player
            owner().owner(),    // for half the damage dealt.
            SpellEvent(
                owner(),
                SPELL_VAMPIRIC_HEALING,
                totalDamage / 2
            )
        );
    }
}
```

Rule #4:

Создавайте ортогональные и взаимозаменяемые абстракции

На выходе имеем невероятную гибкость:

```
// With a little bit of DSL help:
auto VampiricSword       = WeaponBehavior() & VampiricBehavior();
auto SpikedShield        = WeaponBehavior() & ArmorBehavior();
auto CrownOfChaos        = ArmorBehavior() & CorruptingBehavior();
auto SwordOfFireballs    = WeaponBehavior() & RandomCastBehavior(SPELL_FIREBALL, 0.01);

auto RingOfIce           = ResistanceBehavior(DMG_WATER, 0.0) &
                           VulnerabilityBehavior(DMG_FIRE, 2.0) &
                           FreezingBehavior();
// ...
```

И теперь ваш гейм-дизайнер здоровается с вами за руку!

А еще мы на самом деле придумали часть Entity Component System (ECS). Рекомендую доклад Brian Bucklew про Caves of Qud.

```
// Fetch twitter posts asynchronously.
Task<TwitterPosts> fetchTwitterPosts(Network &network, const TwitterFetchOptions &opts) {
    return network
        .request(makeNetworkRequest(opts))
        .then([](std::string_view jsonData) {
            TwitterPosts result;
            deserialize(Json::parse(jsonData), &result);
            return result;
        });
}
```

```
// Fetch twitter posts asynchronously.
Task<TwitterPosts> fetchTwitterPosts(Network &network, const TwitterFetchOptions &opts) {
    return network
        .request(makeNetworkRequest(opts))
        .then([](std::string_view jsonData) {
            TwitterPosts result;
            deserialize(Json::parse(jsonData), &result);
            return result;
        });
}
```

VS

```
// Fetch twitter posts ... asynchronously?
TwitterPosts fetchTwitterPosts(Network &network, const TwitterFetchOptions &opts) {
    auto jsonData = network.request(makeNetworkRequest(opts));
    TwitterPosts result;
    deserialize(Json::parse(jsonData), &result);
    return result;
}
```

```cpp
// Fetch twitter posts asynchronously.
Task<TwitterPosts> fetchTwitterPosts(Network &network, const TwitterFetchOptions &opts) {
    return network
        .request(makeNetworkRequest(opts))
        .then([](std::string_view jsonData) {
            TwitterPosts result;
            deserialize(Json::parse(jsonData), &result);
            return result;
        });
}
```

VS

```cpp
// Fetch twitter posts asynchronously!
CoResult<TwitterPosts> fetchTwitterPosts(Network &network, const TwitterFetchOptions &opts) {
    auto jsonData = co_await network.request(makeNetworkRequest(opts));
    TwitterPosts result;
    deserialize(Json::parse(jsonData), &result);
    co_return result;
}
```

```
// Fetch twitter posts asynchronously.
Task<TwitterPosts> fetchTwitterPosts(Network &network, const TwitterFetchOptions &opts) {
    return network
        .request(makeNetworkRequest(opts))
        .then([](std::string_view jsonData) {
            TwitterPosts result;
            deserialize(Json::parse(jsonData), &result);
            return result;
        });
}
```

VS

```
// Calls allocate a coroutine frame.
CoResult<TwitterPosts> fetchTwitterPosts(Network &network, const TwitterFetchOptions &opts) {
    auto jsonData = co_await network.request(makeNetworkRequest(opts)); // Also allocates.
    TwitterPosts result;
    deserialize(Json::parse(jsonData), &result);
    co_return result;
}
```

```cpp
// Fetch twitter posts asynchronously.
Task<TwitterPosts> fetchTwitterPosts(Network &network, const TwitterFetchOptions &opts) {
    return network
        .request(makeNetworkRequest(opts))
        .then([](std::string_view jsonData) {
            TwitterPosts result;
            deserialize(Json::parse(jsonData), &result);
            return result;
        });
}
```

VS

```cpp
// Should be called with co_await, turning caller function into a coroutine...
CoResult<TwitterPosts> fetchTwitterPosts(Network &network, const TwitterFetchOptions &opts) {
    auto jsonData = co_await network.request(makeNetworkRequest(opts));
    TwitterPosts result;
    deserialize(Json::parse(jsonData), &result);
    co_return result;
}
```

Cм. What Color is Your Function by Bob Nystrom .

Rule #5:
Итерируйтесь!

- Начинайте с написания клиентского кода, пробуйте разные варианты API, оставляйте тот, что лучше подходит под ваши use cases.
- Иногда решения нет. Для приятной глазу асинхронности нужна поддержка stackful корутин из коробки. *Или можно сходить на доклад @apolukhin.*
- Иногда можно поправить core language (привет `mdspan`).
- Иногда у вас уже есть плохой API — не стесняйтесь писать адаптеры (привет `std::chrono`).
- Ваша цель — достаточно хороший API, соответствующий вашим требованиям. Идеальных API не бывает.

Cheat Sheet

1. Проектируйте API так, чтобы его нельзя было использовать неправильно.

   Все возможные способы использования вашего API должны или отрабатывать корректно, или завершаться ошибкой!

2. Divide & Conquer: Дробите!

   На классы, на модули, на функции, на слои абстракции.

3. Тратьте время на придумывание хороших имен!

   Если не получается придумать нормальное имя — значит вы придумали плохую абстракцию.

4. Создавайте ортогональные и взаимозаменяемые абстракции.

   Хорошее правило которое почти всегда верно — данные ортогональны логике. Вдохновляйтесь STL!

5. Итерируйтесь!

   И ждите статью на Хабре.

END

END