Рефакторинг — конспект темы

Конструкторы для начинающих

У класса `ReadingManager` было две константы:

```
static const int MAX_USER_COUNT_ = 100'000;
static const int MAX_PAGE_COUNT_ = 1'000;
```

Допустим, вы сделали эти значения вариативными и хотите передавать их в качестве аргументов конструктора:

```
ReadingManager(int max_user_count, int max_page_count)
        : user_page_counts_(max_user_count + 1, -1),
          page_achieved_by_count_(max_page_count + 1, 0)
{
}
```

Метод `Cheer` использует множитель `cheer_factor`, который считает результат работы этого метода:

```
ReadingManager(int max_user_count, int max_page_count, double cheer_factor)
        : user_page_counts_(max_user_count + 1, -1),
          page_achieved_by_count_(max_page_count + 1, 0),
          cheer_factor_(cheer_factor)
{
}
```

Теперь вызов конструктора выглядит так:

```
ReadingManager manager(20000, 1000, 2);
```

Такой код сложно понять сходу. К тому же, вы рискуете ошибиться и передать аргументы в неверном порядке.

Чтобы помочь себе и читателю, можно добавить комментарии:

```
ReadingManager manager(
   /* max_user_count */ 20000,
   /* max_page_count */ 1000,
   /* cheer_factor */   2);
```

Конструкторы для продвинутых

Рефакторить конструкторы можно так, чтобы не зависеть от порядка передачи аргументов — создавать методы-сеттеры:

```
class ReadingManager {
public:
    ReadingManager();
```

```
    void SetMaxUserCount(int max_user_count) {
        max_user_count_ = max_user_count;
    }

    void SetMaxPageCount(int max_page_count) {
        max_page_count_ = max_page_count;
    }

    void SetCheerFactor(double cheer_factor) {
        cheer_factor_ = cheer_factor;
    }
    ...
private:
   int max_user_count_ = 0;
   int max_page_count_ = 0;
   double cheer_factor_ = 0;
};
```

Теперь подготовка объекта класса `ReadingManager` будет выглядеть так:

```
ReadingManager manager;
manager.SetMaxUserCount(20000);
manager.SetMaxPageCount(1000);
manager.SetCheerFactor(2);
```

Но возникнут 2 проблемы:

- Теперь после вызова конструктора объект не сконструирован до конца, использование его методов вызывает неопределённое поведение.
- Нужно переписывать другие методы класса.

Конструкторы для мастеров

Три подхода к рефакторингу конструктора с длинным списком аргументов:

1. Использовать иператор неявного преобразования вместо метода Build

Можно использовать оператор приведения типа, чтобы избавиться от необходимости вызывать метод `Build`:

```
class ReadingManagerBuilder {
public:
    ReadingManagerBuilder& SetMaxUserCount(int max_user_count);
    ReadingManagerBuilder& SetMaxPageCount(int max_page_count);
    ReadingManagerBuilder& SetCheerFactor(double cheer_factor);

    operator ReadingManager() const {
        // throw exception if not valid
        return {max_user_count_, max_page_count_, cheer_factor_};
    }

private:
    int max_user_count_;
    int max_page_count_;
    double cheer_factor_;
};
```

Указанный оператор автоматически вызовется при попытке преобразовать `ReadingManagerBuilder` к `ReadingManager`:

```
ReadingManager manager =
    ReadingManagerBuilder().
    SetMaxUserCount(10000).
    SetMaxPageCount(500).
    SetCheerFactor(2);
```

Код стал более компактным. Но в процессе преобразования может выброситься исключение.

2. Принимать в конструкторе ReadingManager структуру, созданную с помощью Set-методов

Объединим параметры конструктора `ReadingManager` в структуру:

```
class ReadingManager {
public:
    ReadingManager(const ReadingManagerParams& params);
    // ...

private:
    // ...
};
```

Валидность набора параметров будет проверять сам конструктор.

Структуру `ReadingManagerParams` можно почти полностью списать с `ReadingManagerBuilder`:

```
struct ReadingManagerParams {
    int max_user_count;
    int max_page_count;
    double cheer_factor;

    ReadingManagerParams& SetMaxUserCount(int max_user_count) {
        this->max_user_count = max_user_count;
        return *this;
    }

    ReadingManagerParams& SetMaxPageCount(int max_page_count) {
        this->max_page_count = max_page_count;
        return *this;
    }

    ReadingManagerParams& SetCheerFactor(double cheer_factor) {
        this->cheer_factor = cheer_factor;
        return *this;
    }
};
```

Создание `ReadingManager` будет выглядеть следующим образом:

```
ReadingManager manager(
    ReadingManagerParams().
    SetMaxUserCount(10000).
    SetMaxPageCount(500).
    SetCheerFactor(2)
);
```

Оставлять ли поля структуры публичными или делать её классом — дело вкуса.

Подход 3. Принимать в конструкторе ReadingManager структуру, созданную с помощью designated initializers

При создании структуры можно указывать названия полей — designated initializers при условии, что:

- Они полностью поддерживаются в языке C.
- Они поддерживаются в компиляторах C++, но полулегально, в целях совместимости с C.
- Официально в C++ эта возможность появится в стандарте C++20.

Возможность явно указывать названия полей снижает вероятность ошибки и не требует создавать `Set`-методы:

```
ReadingManager manager(ReadingManagerParams{
    .max_user_count = 10000,
    .max_page_count = 500,
    .cheer_factor = 2}
);
```

Функции с длинным списком аргументов

Проблемы функцияй с большим количеством параметров:

1. Функцию трудно понимать и тестировать.
2. Когда целочисленных и булевых параметров много, они приводят к ошибкам в месте вызова и необходимости упоминать название аргумента рядом с его значением.

Аргументы можно снабжать комментариями, но это не защитит от ошибок:

```
const Query query = ParseQuery(text, /* country_id */ 7,
                               /* max_word_count */ 225,
                               /* max_word_length */ 23,
                               /* max_form_count_per_word */ 20,
                               /* allow_minus_words */ true,
                               /* keep_stop_words */ false,
                               /* case_sensitive */ false);
```

В случае функции `ParseQuery` можно было бы вынести в `ParsingParams` последние шесть параметров:

```
const Query query = ParseQuery( text, 225,
                                ParsingParams().
                                SetMaxWordCount(7).
                                SetMaxWordLength(23).
                                SetMaxFormCountPerWord(20).
                                SetAllowMinusWords(true));
```

Параметрам `keep_stop_words` и `case_sensitive` можно присвоить значения по умолчанию, которые они получат при отсутствии вызова соответствующих Set-методов. Это позволит не указывать их при инициализации `ParsingParams`.

Вот так будет выглядеть вызов функции при использовании designated initializers:

```
const Query query = ParseQuery(text, 225,
                                {.max_word_count = 7,
                                 .max_word_length = 23,
                                 .max_form_count_per_word = 20,
                                 .allow_minus_words = true,
                                 .keep_stop_words = false,
                                 .case_sensitive = false});
```

Если нужно распарсить поток запросов с одинаковыми настройками, рассмотрите вариант создания класса `QueryParser` с инициализацией по одной из рассмотренных методологий и методом `Parse`:

```
class QueryParser {
public:
    QueryParser(const ParsingParams& params);
    Query Parse(string_view text, int country_id);
};

// ...

QueryParser parser(ParsingParams{.max_word_count = 7,
                                 .max_word_length = 23,
                                 .max_form_count_per_word = 20,
                                 .allow_minus_words = true,
                                 .keep_stop_words = false,
                                 .case_sensitive = false});

// ...

const Query query = parser.Parse(text, 225);
```

Есть несложные способы усовершенствовать работу с булевыми и числовыми параметрами.

- **Использовать enum вместо bool.**
  В будущем вы можете захотеть добавить ещё один режим учёта стоп-слов: учитывать их только в том случае, если перед ними идёт символ +. Тогда можно добавить `enum`-тип `StopWordsMode` «на вырост», тем самым повысив читаемость вызова функции и избежав шанса перепутать параметры с другими аргументами типа `int` и `bool`:

```
enum class StopWordsMode {
    IGNORE,
    KEEP,
    // в будущем этот класс можно будет расширять
};

Query ParseQuery(string_view text, int country_id, int max_word_count, int max_word_length, int max_form_count_per_word,
                 bool allow_minus_words, StopWordsMode stop_words_mode, bool case_sensitive);

// ...

const Query query = ParseQuery(text, 225, 7, 23, 20, true, StopWordsMode::IGNORE, false);
```

- **Объединить булевые параметры в маску.**

Поэтому булевы флаги можно объединить в один `enum ParsingFlag` и собрать из подручных средств такую конструкцию:

```
enum class ParsingFlag {
    // значения, являющиеся степенями двойки,
    // позволяют работать с масками без дополнительных
    // преобразований вида 1 << value
    ALLOW_MINUS_WORDS = 1 << 0,  // 001
    KEEP_STOP_WORDS = 1 << 1,    // 010
    CASE_SENSITIVE = 1 << 2      // 100
};

using ParsingFlags = uint64_t;

Query ParseQuery(string_view text, int country_id, int max_word_count, int max_word_length, int max_form_count_per_word,
                 ParsingFlags flags) {
    // ...
    if (flags & static_cast<ParsingFlags>(ParsingFlag::KEEP_STOP_WORDS)) {
        // обработка запроса с учётом флага KEEP_STOP_WORDS
    }
    // ...
}

// ...

const Query query = ParseQuery(text, 225, 7, 23, 20,
                               static_cast<ParsingFlags>(ParsingFlag::ALLOW_MINUS_WORDS)
                               | static_cast<ParsingFlags>(ParsingFlag::KEEP_STOP_WORDS));
                               // выключенные флаги не указываются в списке
```

Писать всюду `static_cast` неудобно, поэтому можно создать простой шаблон `Flags` и определить для `Flags<ParsingFlag>` и `ParsingFlag` операторы `&` и `|` так, чтобы следующий код работал, как предыдущий:

```
enum class ParsingFlag {
    ALLOW_MINUS_WORDS = 1 << 0,
    KEEP_STOP_WORDS = 1 << 1,
    CASE_SENSITIVE = 1 << 2
};

using ParsingFlags = Flags<ParsingFlag>;

Query ParseQuery(string_view text, int country_id, int max_word_count, int max_word_length, int max_form_count_per_word,
                 ParsingFlags flags) {
    // ...
    if (flags & ParsingFlag::KEEP_STOP_WORDS) {
        // обработка запроса с учётом флага KEEP_STOP_WORDS
    }
    // ...
}

// ...

const Query query = ParseQuery(text, 225, 7, 23, 20,
                               ParsingFlag::ALLOW_MINUS_WORDS | ParsingFlag::KEEP_STOP_WORDS);
```

Безопасности не будет без типизации. Поэтому определим для каждого числового параметра свой тип с explicit-конструктором:

```
struct CountryId {
    int value;
    explicit CountryId(int v)
        : value(v)
    {
    }
```

```
};

struct MaxWordCount {
    int value;
    explicit MaxWordCount(int v)
        : value(v)
    {
    }
};

struct MaxWordLength {
    int value;
    explicit MaxWordLength(int v)
        : value(v)
    {
    }
};

struct MaxFormCountPerWord {
    int value;
    explicit MaxFormCountPerWord(int v)
        : value(v)
    {
    }
};

Query ParseQuery(string_view text,
                 CountryId country_id,
                 MaxWordCount max_word_count,
                 MaxWordLength max_word_length,
                 MaxFormCountPerWord max_form_count_per_word,
                 bool allow_minus_words,
                 bool keep_stop_words,
                 bool case_sensitive);

// ...

const Query query = ParseQuery(text, CountryId(225),
                               MaxWordCount(7),
                               MaxWordLength(23),
                               MaxFormCountPerWord(20),
                               true, false, false);
```

То же можно проделать и для булевых параметров.

Определение каждой такой структуры довольно однотипно. Напрашивается макрос `DECLARE_INT_PARAM`:

```
#define DECLARE_INT_PARAM(Name) \
    struct Name {               \
        int value;              \
        explicit Name(int v)    \
            : value(v) {        \
        }                       \
    }

DECLARE_INT_PARAM(CountryId);
DECLARE_INT_PARAM(MaxWordCount);
DECLARE_INT_PARAM(MaxWordLength);
DECLARE_INT_PARAM(MaxFormCountPerWord);
```

Можно доработать этот макрос так, чтобы разрешить подобным типам автоматически приводиться к `int`:

```
#define DECLARE_INT_PARAM(Name) \
    struct Name {               \
```

```
        int value;              \
        explicit Name(int v)    \
            : value(v) {        \
        }                       \
        operator int() const {  \
            return value;       \
        }                       \
    }
```

Ещё один способ создавать подобные типы — определение специальных суффиксов. Они позволяют писать так:

```
const Query query = ParseQuery(text, CountryId(225),
                               7_words, 23_letters, 20_forms,
                               true, false, false);
```

Итоги: есть несколько способов бороться с громоздкостью и запутанностью списка параметров функции:

1. Сгруппировать параметры в структуры.
2. Сделать функцию методом класса, унеся глобальный контекст из параметров функции в поля класса.
3. Использовать `enum` для булевых параметров: свой для каждого или один общий — с маской.
4. Определять отдельные типы для числовых параметров, снабжая их explicit-конструкторами или другими способами явного создания при вызове функции.