Структуры и классы — конспект темы

Введение

Типы без дополнительной логики называют **структурами**, а типы с методами — **классами**. Все встроенные типы C++ кроме стандартных `int`, `char`, `double`, `bool` — это классы, реализованные на C++.

Функция — законченное действие с ясным входом и выходом, а класс и структура — это объекты. У объекта может быть дополнительная логика в виде методов.

Недостатки пар

Недостатки контейнера `pair`:

1. Непонятность. Например, без комментария неясно, что хранится в `pair<int, int>`. Но комментарии ненадёжны.
2. Непрозрачность.
3. Трудоёмкость изменений.

Структура

Структура — тип данных, в котором можно объединить переменные разных типов. В C++ структура обозначается ключевым словом `struct`. Новый тип данных объявляется вне функции `main`:

```
#include <iostream>
#include <string>
#include <vector>

using namespace std;

struct DocumentContent {
    int id;
    vector<string> words;
};

int main() {
    DocumentContent document_content = {1, {"cat"s, "likes"s, "milk"s}};
```

```
    cout << document_content.id << " "s << document_content.words.size() << endl;
}
```

Вывод программы:

```
1 3
```

Чтобы объявить структуру, нужны:

1. Ключевое слово `struct`.
2. Название нового типа. Обычно его пишут с большой буквы, а при наличии нескольких слов в нём — `ВотТак`.
3. Типы и названия полей в фигурных скобках — в том же формате, что объявление переменных: `int id;` и `vector<string> words;`.
4. Точка с запятой после фигурных скобок.

```
struct DocumentContent {
    int id;
    vector<string> words;
}
// ошибка в этой строке: «error: expected ';' after struct definition»

int main() {

}
```

Значения полей можно указать внутри фигурных скобок сразу при объявлении переменной-структуры:

```
// Документ с id=3 и текстом "найден пушистый кот с зелёными глазами"
DocumentContent document_content = {
    // Значение поля id
    3,
    // внутри этих фигурных скобок перечислены элементы вектора words
    {"found"s, "a"s, "furry"s, "cat"s, "with"s, "green"s, "eyes"s}
};
```

Так `DocumentContent` можно сделать константным — `const DocumentContent`. Можно передать в функцию, вернуть из функции. Можно создать вектор из `DocumentContent`:

```
// Вектор, содержащий два элемента типа DocumentContent
vector<DocumentContent> documents = {
    // белый попугай ищет хозяина
    {1, {"a"s, "white"s, "parrot"s, "is"s, "looking"s, "for"s, "an"s, "owner"s}},
    // найдена коричневая морская свинка породы тедди
    {2, {"found"s, "a"s, "brown"s, "teddy", "guinea"s, "pig"s}},
};
```

Фигурные скобки позволяют создать структуру «на лету»:

```
vector<DocumentContent> documents;

int id = ReadLineWithNumber();
vector<string> words = SplitIntoWords(ReadLine());

// Создаём структуру DocumentContent и инициализируем её поля значениями переменных id и words
// Созданная структура добавляется в вектор
documents.push_back({id, words});
```

К полям обращаются только по их названию.

Сортировка вектора структур

В этом коде везде вместо пар структура, поэтому исправлять порядок полей в конце функции не нужно. Зато надо сортировать структуры в определённом порядке:

```
vector<Document> FindTopDocuments(const vector<DocumentContent>& documents,
                                  const set<string>& stop_words, const string& raw_query) {
    const set<string> query_words = ParseQuery(raw_query, stop_words);
    auto matched_documents = FindAllDocuments(documents, query_words);

    // Сортировка по убыванию релевантности
    // Раньше здесь было sort и reverse

    if (matched_documents.size() > MAX_RESULT_DOCUMENT_COUNT) {
        matched_documents.resize(MAX_RESULT_DOCUMENT_COUNT);
    }

    // Исправление порядка полей в парах

    return matched_documents;
}
```

В коде ниже — функция-компаратор, которая принимает два документа по константной ссылке. Первый документ называется `lhs` (левый операнд), а второй `rhs` (правый операнд):

```
bool HasDocumentLessId(const Document& lhs, const Document& rhs) {
    return lhs.id < rhs.id;
}
```

Функция возвращает `true`, если `lhs` меньше `rhs`.

Чтобы отсортировать документы по возрастанию `id`, нужно вызвать функцию `sort` и передать в неё новый аргумент — функцию-компаратор:

```
// Теперь sort будет использовать функцию HasDocumentLessId,
// чтобы узнать, как нужно упорядочивать документы
sort(documents.begin(), documents.end(), HasDocumentLessId);
```

Зачем нужны классы

Когда проектируете код, важно думать о масштабируемости. Чтобы при масштабировании объём кода не вышел из-под контроля, нужно объединить несколько объектов по смыслу и объявить новый тип данных — `class`:

```
struct DocumentContent {
    int id = 0;
    vector<string> words;
};

// Объявляем класс SearchServer с полями documents_ и stop_words_
class SearchServer {
    DocumentContent documents_;
    set<string> stop_words_;
};
```

В программе появился класс `SearchServer`, внутри которого содержится вектор документов и множество стоп-слов.

Отличия **классов** и **структур**:

- у `struct` поля по умолчанию открытые, а у `class` — приватные;
- наследование у `struct` тоже открытое по умолчанию, а у `class` — приватное.

Публичные поля структуры уязвимы — кто угодно может изменить содержимое поля в обход задуманной автором логики.

С классом можно сделать только то, что позволяют его методы. **Метод** — специальная функция, которая выполняет операцию над объектом класса. В этом идея **инкапсуляции**.

Методы классов

По умолчанию все поля класса приватные. В структурах поля по умолчанию публичные, но это не мешает добавить туда приватные поля и методы.

Для внешнего пользователя класса объект должен выглядеть не как простой набор полей (вектор и множество), а как объект, с которым можно взаимодействовать через методы.

В этом коде метод `AddDocument` заменил одноимённую функцию:

```
class SearchServer {
    void AddDocument(int document_id, const string& document) {
        const vector<string> words = SplitIntoWordsNoStop(document, stop_words_);
        documents_.push_back({document_id, words});
    }

    struct DocumentContent {
        int id = 0;
        vector<string> words;
    };

    vector<DocumentContent> documents_;
    set<string> stop_words_;
};
```

Метод выглядит как функция, но находится в классе. Он больше не принимает параметры `documents` и `stop_words`.

Метод запускается через точку в контексте конкретного объекта — `server.AddDocument(id, document)`. При этом у него:

- два явных параметра — `id` и содержимое документа,
- один неявный параметр — сам объект `server` типа `SearchServer`.

Чтобы обратиться к полям неявного параметра, упомяните их по названию, без точки: `documents_.push_back({document_id, words});`.

`push_back` — такой же метод для вектора, как `AddDocument` — метод для класса `SearchServer`.

Константные методы

Методы, не меняющие объект, в контексте которого вызваны, нужно помечать константными. Для этого используют ключевое слово `const`:

```
class SearchServer {
public:
    void SetStopWords(const string& text) { /* ... */ }

    void AddDocument(int document_id, const string& document) { /* ... */ }

    vector<Document> FindTopDocuments(const string& raw_query) const /* ВНИМАНИЕ СЮДА! */ {
        const set<string> query_words = ParseQuery(raw_query);
        auto matched_documents = FindAllDocuments(query_words);

        sort(matched_documents.begin(), matched_documents.end(), HasDocumentGreaterRelevance);
        if (matched_documents.size() > MAX_RESULT_DOCUMENT_COUNT) {
            matched_documents.resize(MAX_RESULT_DOCUMENT_COUNT);
        }
        return matched_documents;   }

private:
    struct DocumentContent {
        int id = 0;
        vector<string> words;
    };

    vector<DocumentContent> documents_;
    set<string> stop_words_;

    vector<Document> FindAllDocuments(const string& query) { /* ... */ }

    /* Прочие методы */
};
```

Вот что делает `const` в заголовке метода `FindTopDocuments`:

1. Поля класса при обращении к ним в теле метода стали константными. Изменить их нельзя.
2. Из этого метода нельзя вызывать неконстантные методы.
3. Константный метод `FindTopDocuments` теперь можно вызвать для константного объекта.

Статические методы не должны помечаться как константные.