Основы разработки на C++. Пространства имён и указатель this

Оглавление

Пространства имён

2.1 Знакомство с учебным примером

Поговорим о пространствах имён в языке C++. Начнем с знакомства с учебным примером, на котором мы будем разбирать, что такое пространства имён, и как ими, собственно, пользоваться. Давайте представим, что мы с вами пишем программу управления личными финансами. И мы хотим собирать в одном месте все наши расходы и как-то потом их обрабатывать. И у нас уже есть какой-то код, с которого мы начинаем работу.

```
#include <algorithm>
#include <iostream>
#include <fstream>
#include <vector>

using namespace std;
```

У нас есть структура `Spending` — собственно, трата, которая состоит из двух полей. Это категория — собственно, что это за трата: продукты, транспорт, одежда или что-то еще, и amount — количество, сколько денег мы на эту категорию потратили.

```
struct Spending {
    string category;
    int amount;
};
```

И вот эти вот самые траты мы как-то в нашей программе умеем обрабатывать. Например, у нас есть функция `CalculateTotalSpendings`, которая по вектору расходов, по вектору трат считает, собственно, сколько всего денег мы потратили. Ну, она делает это предельно просто: она просто проходит по вектору и находит сумму полей `amount`. И возвращает, сколько в сумме денег мы потратили.

```
int CalculateTotalSpendings(const vector<Spending>& spendings) {
    int result = 0;
    for (const Spending& s : spendings) {
        result += s.amount;
    }
    return result;
}
```

Также у нас есть другая функция обработки расходов — это функция `MostExpensiveCategory`. Она находит самую дорогую категорию расходов в наших тратах. Она тоже устроена достаточно просто, она использует стандартный алгоритм `max_element` и проходит по вектору расходов и находит ту категорию, у которой расход, собственно, вот это поле `amount`, максимальный.

```
int MostExpensiveCategory(const vector<Spending>& spendings) {
    auto compare_by_amount = [](const Spending& lhs, const Spending& rhs) {
        return lhs.amount, rhs.amount;
    };
    rerurn max_element(begin(spendings), end(spendings), compare_by_amount)->category;
}
```

И у нас есть функция `main`, в которой представлен пример использования структуры `Spending` и функций. Мы создали какой-то вектор, заполнили его. Вот у нас сказано, что на продукты мы потратили 2500 рублей, на транспорт — 1150, ну и так далее. И мы этот вектор передаем в функцию `CalculateTotalSpendings` и `MostExpensiveCategory`.

```
int main() {
    const vector<Spending> spendings = {
        {"food", 2500},
        {"transport", 1150},
        {"restaurants", 5780},
        {"clothes", 7500},
        {"travel", 23740},
        {"sport", 12000}
    };
    cout << "Total " << CalculateTotalSpendings(spendings) << endl;
    cout << "Most expensive " << MostExpensiveCategory(spendings) << endl;
}
```

Давайте, как обычно, скомпилируем нашу программу, убедимся, что наш стартовый код компилируется, запустим и видим, что суммарно мы потратили 52670 рублей, и самой дорогой категорией расходов оказались путешествия. Ну и если мы посмотрим в наш вектор, то убедимся, что да, наша функция правильно работает, действительно на путешествия мы потратили денег больше всего. Но сейчас мы можем добавлять расходы, только указывая их вот в этом векторе в функции `main` и перекомпилируя программу. Естественно, если мы разрабатываем программу для массового потребителя — это будет неудобно.

И мы решили, что, чтобы наша программа была удобна пользователям, мы хотим добавить в нее возможность загрузки расходов из формата XML и из формата JSON. Естественно, мы не хотим самостоятельно писать свой XML парсер, потому что их уже много написано. Поэтому мы

хотим воспользоваться готовой библиотекой. Ну и то же самое для JSON, мы тоже хотим взять готовый код и просто им воспользоваться. Для этого **в проект нужно добавить соответствующие библиотеки для работы с данными форматами**.

Вот так вот выглядит XML документ, и здесь те же самые расходы, которые у меня в функции `main`, но описанные в формате XML.

```
<july>
    <spend amount = "2500" category = "food"></spend>
    <spend amount = "1150" category = "transport"></spend>
    <spend amount = "5780" category = "restaurants"></spend>
    <spend amount = "7500" category = "clothes"></spend>
    <spend amount = "23740" category = "travel"></spend>
    <spend amount = "12000" category = "sport"></spend>
</july>
```

Собственно, что мы хотим сделать в коде? Мы хотим написать функцию, которая возвращает вектор структур `Spending`, она называется `LoadFromXml`, принимает на вход поток ввода. Ещё одну функцию мы хотим написать для формата JSON с точно таким же интерфейсом. При этом мы хотим воспользоваться нашими готовыми библиотеками, чтобы, собственно, не писать парсеры самим.

```
vector<Spending> LoadFromXml(istream& input) {

}

vector<Spending> LoadFromJson(istream& input) {

}
```

Для того, чтобы вы как следует познакомились с кодом наших готовых библиотек, **написание вот этих функций** — `LoadFromXml` и `LoadFromJson` — **мы вынесли в тренировочные задачи, чтобы вы сами эти функции написали**, сами познакомились с кодом библиотек, потому что так вам будет проще дальше понимать, как мы будем применять и изучать пространства имён.

2.2 Проблема пересечения имён двух разных библиотек

Итак, вы решили две тренировочные задачи, в которых по отдельности добавили в нашу программу управления личными финансами поддержку форматов JSON и XML. Теперь давайте попробуем добавить одновременно поддержку обоих этих форматов в нашу программу. Ведь каким-то пользователям нашего приложения будет удобно загружать свои расходы из XML, а каким-то из JSON. И, конечно, хорошо бы, чтобы наша программа умела одновременно работать с обоими форматами.

Функции `LoadFromXml` и `LoadFromJson` реализованы с помощью библиотек, которые мы вам выдавали.

```
vector<Spending> LoadFromXml(istream& input) {
    Document doc = Load(input);
    vector<Spending> result;
    for (const Node& node : doc.GetRoot().Children()) {
        result.push_back(node.AttributeValue<string>("category"),
            node.AttributeValue<int>("amount"));
    }
    return result;
}

vector<Spending> LoadFromJson(istream& input) {
    Document doc = Load(input);
    vector<Spending> result;
    for (const Node& node : doc.GetRoot().AsArray()) {
        result.push_back(node.AsMap().at("category").AsString(),
            node.node.AsMap().at("amount").AsInt());
    }
    return result;
}
```

Пример использования: создаём поток чтения из файла `spendings.json`, в котором расходы описаны в формате JSON, загружаем оттуда вектор расходов и дальше обрабатываем его с помощью наших функций подсчёта суммарного количества расходов и поиска самого большого расхода.

```
int main() {
    ifstream json_input("spendings.json");
    const auto spendings = LoadFromJson(json_input);

    cout << "Total " << CalculateTotalSpendings(spendings) << endl;
    cout << "Most expensive " << MostExpensiveCategory(spendings) << endl;
}
```

Запускаем компиляцию, и у нас, ожидаемо, ничего не компилируется. Компилятор говорит, что "Document was not declared in this scope". Логично, мы не подключили в главный файл библиотеки для работы с XML и JSON. Поэтому давайте это сделаем. Подключаем `xml.h` и `json.h` и снова запускаем компиляцию программы.

```
#include "xml.h"
#include "json.h"
```

Она снова не компилируется, но ошибка компиляции теперь другая. Давайте посмотрим на неё внимательно. Компилятор пишет: **«redefinition of class Document»**. И давайте мы на неё нажмём. Компилятор ругается, что класс `Document` в файле `json.h` определён заново. Давайте посмотрим, где же находится первое определение.

Мы можем нажать на **«previous definition of class Document»** и ожидаемо попасть в файл `xml.h`, в котором у нас тоже есть класс `Document`.

И, собственно, в чём проблема, почему наша программа не компилируется? Дело в том, что у нас есть два файла, две библиотеки: `xml.h` и `json.h`, и **в обеих этих библиотеках есть**

классы и функции с одинаковыми именами. И там, и там есть классы `Node` и `Document` и есть функция `Load`. И компилятор, собирая наш проект воедино, не может выбрать: **он видит две разные реализации одного и того же имени, и происходит нарушение правила одного определения**. Далее мы посмотрим, как решить эту проблему.

2.3 Знакомство с пространствами имён

Итак, мы столкнулись с проблемой использования двух библиотек, так как в обеих библиотеках есть классы с одинаковыми названиям `Node` и `Document`. И из-за этого, когда мы собирали проект, у нас нарушалось правило одного определения и проект не собирался.

- `xml.h`

```
class Node {...};
class Document {...};
Document Load(istream& ...);
```

- `json.h`

```
class Node {...};
class Document {...};
Document Load(istream& ...);
```

Давайте подумаем, как эту проблему можно решить.

Казалось бы, если имена одинаковые, давайте их сделаем разными и проблема исчезнет. Например, мы могли бы в файле `xml.h` классы `Node`, `Document` и функцию `Load` переименовать, например в `XmlNode`, `XmlDocument` и `LoadXml`. И, в принципе, это нормальное решение, все будет работать, проблема наша исчезнет. И, например, таким образом поступили в библиотеке Qt, которая широко применяется во многих проектах на C++. Но мы с вами не будем делать так, мы пойдем другим путем и воспользуемся сущностью, которая специально существует в C++ для решения вот такой проблемы пересечения имён. Эта сущность — **пространство имён**.

Сделаем следующее: **обернем каждую библиотеку в свое пространство имён**. Начнем с библиотеки `xml`. Вначале перед классом `Node` мы напишем `namespace Xml` и поставим открывающую скобку. Мы только что создали новое пространство имён, которое назвали `Xml`. И с помощью фигурной скобки мы его открыли. Все, что будет внутри этого пространства имён, внутри этой фигурной скобки, будет относиться к пространству имён `Xml`. Поэтому мы переходим к концу нашего заголовочного файла и закрываем фигурную скобку.

```
...
#include <unordered_map>

using namespace std;

namespace Xml {

class Node {...};

class Document {...};

Document Load(istream& input);
...

}
```

Теперь классы `Node`, `Document` и функция `Load` находятся в пространстве имён `Xml`.

Но пока что мы обернули в пространство имён только их объявления. Давайте перейдем в `cpp`-файл и точно так же обернем реализации методов классов и реализацию функции `Load`.

Теперь то же самое давайте сделаем для библиотеки `json`. Точно так же объявим пространство имён, назовем его `Json`, и обернем классы `Node`, `Document` и функцию `Load` в это пространство.

```
...
#include <unordered_map>

using namespace std;

namespace Json {

class Node {...};

class Document {...};

Document Load(istream& input);
...

}
```

Перейдем в `cpp`-файл и обернем в пространство имён все реализации.

Теперь давайте снова соберем наш проект. Запускаем компиляцию. Компиляция завершилась неудачно. Но важно, что ошибка компиляции теперь другая. Теперь компилятор говорит нам: «Document was not declared in this scope». Он говорит это два раза для каждой из функций `LoadFromXml` и `LoadFromJson`.

То есть, если раньше компилятор понимал, что такое `Document`, но видел два его определения, то теперь он не понимает, что это за имя такое `Document`, он его не видит. И чтобы наша программа начала компилироваться, нам нужно указать полное имя класса `Document`. Для этого мы напишем `Xml::Document` и аналогично для всех остальных классов и функций (и также для `Json`).

```
vector<Spending> LoadFromXml(istream& input) {
    Xml::Document doc = Xml::Load(input);
    vector<Spending> result;
    for (const Xml::Node& node : doc.GetRoot().Children()) {
        result.push_back(node.AttributeValue<string>("category"),
            node.AttributeValue<int>("amount"));
    }
    return result;
}

vector<Spending> LoadFromJson(istream& input) {
    Json::Document doc = Json::Load(input);
    vector<Spending> result;
    for (const Json::Node& node : doc.GetRoot().AsArray()) {
        result.push_back(node.AsMap().at("category").AsString(),
            node.node.AsMap().at("amount").AsInt());
    }
    return result;
}
```

Запустим компиляцию. Наша программа скомпилировалась и корректно работает.

Таким образом, **мы обернули каждую из библиотек в свое пространство имён**. А в том месте, **где мы эти библиотеки использовали, мы указали полные имена классов и функций**. И теперь у компилятора не возникает неоднозначности. Когда он видит `Json::Document`, он понимает, что это класс `Document` из пространства имён `Json`. И это совсем не тот же самый `Document`, который находится в пространстве имён `Xml`.

2.4 Особенности синтаксиса пространств имён

Давайте подробнее рассмотрим синтаксис, который применяется при работе с пространствами имён. Начнём с простейшего вопроса. Собственно, **как создать свое пространство имён**? Мы это уже сделали в предыдущем примере, и все довольно просто. Мы **пишем ключевое слово `namespace` и за ним указываем имя этого пространства имён. А дальше в фигурных скобках, собственно, в блоке кода, мы перечисляем содержимое этого пространства имён**.

Теперь рассмотрим, как же определять, как же создавать определение, реализацию элементов пространства имён. Здесь есть два варианта синтаксиса.

- пишем имя нашего пространства имён `namespace`, и дальше, в фигурных скобках, реализуем функцию (в данном случае функцию `Load`)

```
namespace Json {
    Document Load(isream& input) {
        ...
    }
}
```

- другой вариант: это не оборачивать реализацию в пространстве имён, а просто указать полные имена используемых объектов.

```
Json::Document Json::Load(isream& input) {
    ...
}
```

Проверим второй способ: возьмем реализацию функции `Load` в пространстве имён `Xml` и вынесем ее в самый конец файла за границы пространства имён `Json`.

```
namespace Xml {
    ...
}

Document Load(isream& input) {
    return Document{LoadNode(input)};
}
```

Запустим компиляцию и увидим, что у нас не компилируется, потому что компилятор не знает, что такое `Document`.

Теперь мы укажем полные имена используемых классов и функций.

```
namespace Xml {
    ...
}

Xml::Document Xml::Load(isream& input) {
    return Xml::Document{Xml::LoadNode(input)};
}
```

Скомпилируем, и у нас компилируется. То есть вот этот второй вариант создания работает.

Но при этом, я думаю, вам очевиден недостаток такого синтаксиса. Смотрите, в этих двух строчках мы написали `Xml::` **четыре раза**. Такой способ часто приводит к загромождению кода.

Теперь давайте отметим важную вещь: **пространства имён расширяемы**, то есть мы можем в разных файлах объявлять одно и то же пространство имён, и все, что мы в этих разных файлах туда поместим, попадет в это пространство имён.

Продемонстрируем это в нашем проекте. Мы создадим новый заголовочный файл. Давайте назовем его `xml_load.h`, а в нем мы подключим файл `xml.h`, объявим пространство имён `Xml` и перенесем функцию `Load` в этот файл. При этом реализацию мы оставим в `xml.ccp`. Только чтобы компилятор знал, какую функцию мы реализуем, мы здесь тоже подключим наш `xml_load.h`.

```
#pragma once;

#include "xml.h"

namespace Xml {
    Document Load(isream& input);
}
```

В главной программе мы тоже подключим `xml_load.h`. Запустим компиляцию. Компиляция прошла успешно.

Теперь давайте поговорим об обращении к элементам пространств имён. Ранее мы показывали, что для того чтобы обратиться к элементу пространства имён какого-то конкретного, **нужно написать имя этого пространства имён, два двоеточия и, собственно, имя класса или функции**. Так вот, это нужно делать, **когда мы обращаемся снаружи**, то есть мы, например, в функции `main` пишем какой-то код и обращаемся к элементам библиотеки `Xml`.

Если же мы обращаемся изнутри пространства имён, как в примере в реализации функции `Load`, — у нас функция `Load` реализована внутри пространства имён `Xml`, поэтому мы можем просто писать `Document`, можем просто вызывать функцию `LoadNode`, которая также находится в этом пространстве имён, — нам **не обязательно** указывать полное имя, потому что **компилятор будет искать это имя**, в данном случае `Document`, **в первую очередь внутри пространства имён**.

Если же мы обращаемся снаружи к каким-то элементам, то нужно указывать полное имя, потому что без него компилятор не будет заглядывать в те пространства имён, которые есть в вашей программе.

2.5 Using-декларация

Давайте рассмотрим юнит-тест `TestDocument` из заготовки решения задачи «Библиотека работы с INI-файлами», которую вы должны были решить ранее. В этом тесте мы объявляем переменную `section`, которая имеет тип указатель на `Ini::Section`.

```
Ini::Section* section = &doc.AddSection("one");
```

При этом ниже, ближе к концу теста, у нас еще объявлены три константы, которые также имеют тип `Ini::Section`, то есть имя `Ini::Section` мы в нашем тесте используем 4 раза. Это юнит-тест на библиотеку работы с INI-файлами, то есть понятно, что он будет обращаться к содержимому пространству имён `Ini`, а не к какому-то другому пространству имён, и поэтому использование полного имени `Section` может быть излишним, оно может затруднять написание кода. Сейчас у нас, конечно, короткое имя у пространства имён, а если оно будет длинное, то код и читать может быть довольно сложно.

В C++ есть возможность **не указывать полное имя объекта из пространства имён** с помощью так называемой **using-декларации**, то есть мы можем написать `using Ini::Section`, и дальше в коде мы можем не использовать префикс `Ini::` при обращении к имени `Section`.

```
using Ini::Section;

Section* section = &doc.AddSection("one");
```

То есть, мы командой `using Ini::Section` сказали компилятору, что когда ты видишь имя `Section`, это значит, что мы имеем в виду вот это полное имя `Ini::Section`. Надо отметить, что **декларация распространяется естественно только на то имя, которое в ней указано**.

Второй важный момент состоит в том, что **using-декларация действует только внутри того блока кода, в котором она находится**, то есть сейчас `using`-декларация находится внутри тела нашей функции, внутри фигурных скобок, которые задают тело функции.

2.6 Директива using namespace

Давайте рассмотрим другой юнит-тест из заготовки решения задачи «Библиотека работы с INI-файлами» — `TestLoadIni`. В нем у нас есть обращение к `Ini::Document`, к `Ini::Load`, и к `Ini::Section`. При этом к `Ini::Section` мы обращаемся даже дважды. На самом деле мы в нашем юнит-тесте обращаемся ко всем именам, которые у нас есть в пространстве имён `Ini`. Поэтому нам хочется писать краткие имена. Мы уже знаем как это сделать. Мы для этого можем воспользоваться `using`-декларацией и написать

```
using Ini::Document;
using Ini::Section;
using Ini::Load;
```

Теперь мы можем спокойно удалить префиксы, запустить компиляцию и увидеть, что всё компилируется. Но на самом деле мы не так много выиграли. Сейчас у нас только три имени, но если бы их было 30, то нам вряд ли было бы удобно писать 30 using-деклараций. Нам нужно какое-то другое средство, которое позволит сказать: «Я хочу использовать все имена из данного пространства имен». И конечно же, в языке C++ такое средство есть, и оно вам хорошо знакомо. Это директива `using namespace`. Мы можем написать `using namespace Ini`, убрать другие `using`-декларации, и наша программа продолжит компилироваться. То есть мы одной этой строчкой сказали компилятору: «Я хочу, чтобы при поиске имён ты бы ещё заглядывал в пространство имён `Ini` и искал там». Как и `using`-декларация, директива `using namespace` действует только в том блоке кода, в котором объявлена.

2.7 Глобальное пространство имён

Если **функция или класс не помещены ни в какое пространство имён**, то говорят, что они находятся в **глобальном пространстве имён**. Мы говорили, что `using`-декларация и директива `using namespace` действуют внутри того блока кода, в котором они объявлены. Однако, их можно использовать и в глобальном пространстве имён, то есть, например, в нашем `cpp`-файле мы

можем написать `using Ini::Document`, и тогда везде в cpp-файле мы можем уже не указывать перед `Document` пространство имён. Или же мы можем прямо здесь написать `using namespace Ini` и вообще не использовать префикс `Ini` в нашем файле. Можно проверить, что при подобном имзенени программа компилируется и всё работает. То есть мы за счет использования `using namespace` в глобальном пространстве имён сократили наш код и избавили себя от необходимости каждый раз указывать префикс.

Но с такими вещами нужно быть осторожными. Пусть, например, есть у вас имя `Document`, и оно видно компилятору. Компилятор понимает, что такое `Document`. **Когда мы помещаем это имя в пространство имён, мы ограничиваем его видимость**, мы говорим, что теперь это имя можно видеть только через специальное окошко. В виде этого окошка выступает префикс `Ini::`. Когда же мы в глобальном пространстве имён вводим `using`-декларация или директиву `using namespace`, мы убираем это окошко и снова имя `Document` становится видно компилятору отовсюду, то есть мы добиваемся эффекта обратного тому, который мы создаем с помощью пространства имён.

А при этом **мы используем пространство имён именно для того, чтобы избежать конфликтов в глобальном пространстве имён**, соответственно, злоупотребляя `using`-декларациями и директивой `using namespace`, мы можем свести на нет все наши усилия от заворачивания классов в функции в пространстве имён. И отсюда следует очень важная, практическая рекомендация: **минимизируйте область действия `using`-деклараций и директивы `using namespace`**, то есть если вы используете их в широкой области видимости, то вы повышаете риск возникновения конфликтов имён. Когда же мы используем `using`-декларацию и директивы `using namespace` только в маленьких блоках кода, в маленьких функциях или даже внутри какого-то блока, который является частью функций, тогда вероятность того, что у нас там возникнут какие-то конфликты имён, очень низкая.

2.8 Using namespace в заголовочных файлах

Использование `using namespace` в заголовочных файлах вызывает проблемы. Давайте вернёмся к примеру, с которого мы начинали, а именно к программе управления личными финансами, которая умеет загружать список расходов из форматов JSON и XML. И давайте для начала мы зайдём в функцию `LoadFromXml` и перепишем её так, как мы научились делать: воспользуемся директивой `using namespace` и не будем писать префикс `Xml::` в этой функции.

```
vector<Spending> LoadFromXml(istream& input) {
    using namespace Xml;

    Document doc = Load(input);
    vector<Spending> result;
    for (const Node& node : doc.GetRoot().Children()) {
        result.push_back(node.AttributeValue<string>("category"),
            node.AttributeValue<int>("amount"));
    }
    return result;
}
```

Давайте запустим компиляцию, убедимся что всё хорошо. Здесь мы можем прекрасно использовать `using namespace Xml`, у нас маленькая функция.

Теперь давайте представим, что мы в своей программе решили не только загружать расходы из формата JSON, но и сохранять их в этот формат. Ну, мы развиваем наше приложение, хотим, чтобы у нас было больше пользователей и добавляем новые функции. При этом наша библиотека по работе с форматом JSON умеет только загружать (это не наша библиотека, мы её откуда-то взяли), и она умеет только загружать данные из формата JSON, поэтому мы решили, что сохранение в JSON мы напишем сами. Давайте это сделаем.

Для этого мы добавим в наш проект заголовочный файл, назовём его `json_utils.h`. Затем, мы перенесём в него структуру `Spending`, чтобы нам было удобнее, и добавим следующий набор функций.

```
#pragma once

#include <string>
#include <vector>
#include <ostream>

#include "json.h"

using namespace std;

struct Spending {
    string category;
    int amount;
};

Json::Node ToJson(const Spending& s);
Json::Document ToJson(const vector<Spending>& spendings);
ostream& operator <<(ostream& os, const Json::Node& node);
ostream& operator <<(ostream& os, const Json::Document& document);
```

Отлично, теперь давайте мы этот файл подключим в главном файле, в `main.cpp`, и запустим компиляцию. Программа компилируется.

Тут вы могли задуматься, почему мы только объявили эти функции, но не реализовали, и при этом наша программа компилируется? Всё нормально, потому что мы эти функции пока нигде не вызываем, поэтому компилятору достаточно видеть их объявление, он их не вызывает, и определения ему не нужны.

Итак, пока у нас всё хорошо, у нас программа компилируется. Но мы посмотрели вот в этот заголовочный файл и подумали: это же файл про `Json`, поэтому, может быть, нам не стоит много раз использовать префикс `Json::`. Мы же здесь только про `Json` пишем, поэтому давайте воспользуемся директивой `using namespace Json` и не будем писать полные имена содержимого этого пространства имён.

```
#pragma once
```

```
#include <string>
#include <vector>
#include <ostream>

#include "json.h"

using namespace std;

struct Spending {
    string category;
    int amount;
};

using namespace Json;

Node ToJson(const Spending& s);
Document ToJson(const vector<Spending>& spendings);
ostream& operator <<(ostream& os, const Node& node);
ostream& operator <<(ostream& os, const Document& document);
```

Вроде бы всё хорошо, но давайте запустим компиляцию. Мы запустили компиляцию и видим, что **программа-то у нас больше не компилируется**, при этом давайте посмотрим на сообщение компилятора, он пишет: «reference to Document is ambiguous». Где это происходит? Это происходит в функции `LoadFromXml`

```
#include "xml.h"
#include "json.h"
#include "json_utils.h"

...

vector<Spending> LoadFromXml(istream& input) {
    using namespace Xml;

    Document doc = Load(input);
    ...
}
```

Теперь наш компилятор не понимает, какому объекту соответствует имя `Document`, потому что у нас в функции используется `using namespace Xml`, а из заголовочного файла `json_utils.h` нам прилетела ещё директива `using namespace Json`. Получается, что вот в этой функции, в точке вызова функции `Load` компилятор видит оба имени `Document`: он видит и `Json::Document`, и `Xml::Document`, и соответственно, у него возникает неоднозначность, и наша программа не компилируется.

Конфликт возник из-за того, что мы воспользовались директивой `using namespace` в заголовочном файле. Чем она плоха? Тем, что мы не знаем, в какие другие файлы будет подключен наш

заголовочный файл, соответственно мы не знаем, какие имена будут видны в тех файлах. А мы берём и в своём заголовочном файле приносим в тот файл все имена из пространства имён, для которого мы воспользовались директивой `using namespace`. Поэтому использование директивы `using namespace` в заголовочных файлах может приводить к неожиданным конфликтам имён, и поэтому **в общем случае нельзя использовать `using namespace` в заголовочных файлах**.

2.9 Пространство имён std

Настало время наконец-то обратить внимание на ту строчку, которую мы писали в наших программах с самого начала — `using namespace std`. **В пространстве имён `std` находится все стандартная библиотека**: все алгоритмы, все контейнеры находятся в этом пространстве имён, то есть `vector`, `deque`, `list`, `map`, алгоритмы, примитивы работы с многопоточностью, `async`, `future`, `mutex` — все они находятся в пространстве имён `std`.

Давайте посмотрим: каково это писать программы без директивы `using namespace std`. Уберем ее. И для примера напишем программу `hello user`, которая спрашивает имя пользователя и здоровается с ним. Она будет выглядеть вот так.

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

int main() {
    std::string name;
    std::cout << "Enter your name" << std::endl;
    std::cin >> name;
    std::cout << "Hello, " << name << std::endl;
    return 0;
}
```

Она у нас компилируется и работает.

Или другой пример: давайте объявим с вами `map`, который отображает строку в вектор строк. Как это будет выглядеть?

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

int main() {
    std::map<std::string, std::vector<std::string>> m;
    return 0;
}
```

Эти два коротких примера демонстрируют нам два важных момента.

- все контейнеры и алгоритмы, которые мы с вами изучали имеют полные имена, которые начинаются с `std::`. И теперь, когда вы будете читать документацию, например на `std::vector`, вы будете понимать, что же это за такое `std::`, потому что в документации обычно для элементов стандартной библиотеки используют их полные имена `std::vector`, `std::string` и так далее.

- использование префикса `std::` в коде, который интенсивно использует стандартную библиотеку, существенно этот код раздувает. Во втором примере мы не просто так объявили этот `map` — видите, в одной строчке в объявлении одной переменной префикс `std::` встречается 4 раза. И, конечно, это может существенно раздувать код, и затруднять его чтение, и, иногда, даже и написание.

Давайте мы с вами сформулируем **рекомендации по использованию пространства имён** `std`. Во-первых, мы сохраняем рекомендацию о том, что **не надо использовать** `using namespace`, **в данном случае** `using namespace std` **в заголовочных файлах**, потому что это может приводить к конфликтам имён. В нашем курсе этих конфликтов не возникает из-за используемого стиля именования функций и классов. Так как в пространстве имён `std` все функции и классы начинаются с маленькой буквы, а мы функции и классы называем с большой буквы. Но в общем случае в других проектах за пределами нашей специализации может использоваться стиль, когда функции и классы именуются с маленькой буквы, и тогда они могут конфликтовать с тем, что есть в пространстве имён `std`.

В `cpp`-файлах ситуация другая. Как мы с вами видели, часто мы в `cpp`-файлах можем интенсивно использовать стандартную библиотеку. И поэтому в них все-таки практичным является использование директивы `using namespace std` даже в глобальном пространстве имён. **В отдельных функциях использование** `using`**-декларации или** `using namespace std` **— в принципе, хорошая практичная рекомендация, но только для** `cpp`**-файлов**.

Давайте перейдем к нашей программе по работе с личными финансами. И давайте перепишем ее в соответствии с рекомендациями, которые мы только что написали. `using namespace Json`, который нельзя использовать, вернем назад, чтобы наша программа компилировалась.

```
#pragma once

#include <string>
#include <vector>
#include <ostream>

#include "json.h"

using namespace std;

struct Spending {
    string category;
    int amount;
};

Json::Node ToJson(const Spending& s);
```

```
Json::Document ToJson(const vector<Spending>& spendings);
ostream& operator <<(ostream& os, const Json::Node& node);
ostream& operator <<(ostream& os, const Json::Document& document);
```

Теперь давайте перейдем к `json.h` и поправим его, потому что мы в нем используем `using namespace std`, а мы только что договорились, что не будем использовать эту директиву в заголовочных файлах. И в этом заголовочном файле нам нужно расставить префикс `std` везде, где мы используем элементы стандартной библиотеки.

Конечно, **код** наш именно **в заголовочном файле** от этого немного **раздувается**. Но **зато мы страхуем себя от неприятных проблем**. При этом в `cpp`-файле мы не будем расставлять префикс `std`, а спокойно напишем `using namespace std`, потому что `cpp`-файл никуда не будет включаться и мы можем более безопасно использовать здесь директиву `using namespace`. Запустим компиляцию, и у нас все хорошо. Осталось сделать то же самое для `xml.h`.

Таким образом, директива `using namespace std` существенно сокращает код, который интенсивно используют стандартную библиотеку, но ее нельзя применять в заголовочных файлах и с осторожностью надо применять в `cpp`-файлах.

2.10 Структурирование кода с использованием пространств имён

Все это время мы говорили, что пространства имён в C++ используются только для избежания конфликтов имён, но на самом деле они применяются и для других целей. Посмотрим, как с помощью пространства имён можно улучшить структурирование кода.

Давайте вспомним финальную задачу желтого пояса. В ней вам надо было написать базу данных, которая работала с датами и событиями, при этом вам на вход поступали определенные условия и по этим условиям нужно было отбирать записи в базе данных. На самом деле, если вы не проходили желтый пояс и не решали эту задачу, то ничего страшного, сейчас все станет понятно. Итак, нам на вход поступали условия и мы эти условия разбирали и строили из них абстрактное синтаксическое дерево, которое потом использовали для того, чтобы вычислять условия для каждой записи в базе данных. Абстрактное синтаксическое дерево состояло из некоторых узлов, и каждый узел был экземпляром какого-то класса.

Приведем пример того, в какое дерево выстраивается условие из примера. У нас имеется класс `LogicalOperationNode`, с двумя потомками — `DateComparisonNode` и `EventComparisonNode`, которые, собственно, выполняют вычисление нашего условия.

Давайте посмотрим фрагмент решения финальной задачи желтого пояса, который отвечает как раз за работу с этими условиями. Посмотрим в функцию `main`, она написана специально для нашего примера и работает только с условиями. Она считывает со стандартного ввода строчку условия, разбирает ее с помощью функции `ParseCondition` (она у нас была в финальной задаче желтого пояса) создает абстрактное дерево, которое у нас будет храниться в переменной `condition`, и сейчас, для примера, вычисляет условия для конкретной записи: дата — 31 августа 2018 года и событие — Video.

```
void TestAll();

int main() {
    TestAll();
    string contition_str;
    getline(cin, contition_str);

    istringstream in(contition_str);
    auto condition = ParseCondition(in);

    cout << condition->Evaluate(Date(2018, 8, 31), "Video");
    return 0;
}
```

Давайте мы с вами посмотрим на файл `node.h`. Этот файл содержит классы, которые используются для представления узлов абстрактного синтаксического дерева, при этом этот файл можно открыть в специальном окне, который называется **outline**: в этом окне отображаются все функции, классы и типы, объявленные в файле. И давайте обратим внимание вот на какую особенность: в именах классов в файле `node.h` присутствует слово `Node`, то есть у нас есть базовый класс `Node`, у него есть потомки: `EmptyNode`, `DataComparisonNode`, `EventComparisonNode` и `LogicalOperationNode`. Во всех этих классах используется слово `Node`. Используется и используется — ничего страшного.

Давайте перейдем в `cpp`-файл и посмотрим на него. В нем мы видим что это самое слово `Node`, оно, например, вот в этой строчке используется дважды: для имени класса и еще раз имя класса, потому что это конструктор.

```
...
DateComparisonNode::DateComparisonNode(Comparison comparison, const Date& value) :
    comparison_(comparison), value_(value) {
}
...
```

И вот здесь в конструкторе точно так же.

```
...
EventComparisonNode::EventComparisonNode(Comparison comparison, const string& value) :
    comparison_(comparison), value_(value) {
}
...
```

Давайте вообще ради любопытства посчитаем, сколько раз в этом файла встречается строчка `Node` — во всем файле слово `Node` встречается 12 раз. Ещё давайте заглянем в `node_test`. Этот файл с unit-тестами, который мы запускали в нашей программе, и здесь тоже мы используем эти классы, мы создаем их экземпляры и здесь тоже очень много встречается этот суффикс `Node`.

И понимаете какое дело: возможна ситуация, когда вот это частое использование суффикса `Node` загромождает наш код.

Вы конечно можете сказать, ну подумаешь, что там четыре буквы, но на самом деле этот пример основан на реальной практике, на реальной задаче. Так вот там у классов общий суффикс

имел в длину 14 символов, и часто использование этого суффикса действительно перегружало код и затрудняло его чтение и понимание.

Давайте в нашем учебном примере попробуем избавиться от частого использования общего суффикса в именах классов. У нас есть наши классы в файле `node.h`:

- `Node`;
- `EmptyNode`;
- `DataComparisonNode`;
- `EventComparisonNode`;
- `LogicalOperationNode`;

Мы удалим в их именах общий суффикс `Node`. Дальше мы заведем пространство имён `Nodes` и поместим в него все эти классы. Обратите внимание, что базовый класс мы в пространство имён не помещаем (на самом деле его можно помещать, можно не помещать — это дело вкуса; мы в нашем примере оставим его в глобальном пространстве имён). Давайте осуществим вот это преобразование с нашим проектом.

После того, как мы заменили все имена классов во всех файлах проекта, удалив суффикс `Node` из их имён, давайте сделаем следующий шаг: обернем все классы потомки и их реализации в пространство имён `Nodes`.

Мы выполнили наше преобразование, однако, теперь нам нужно поменять окружающий код, потому что там используются еще имена без учета пространства имён. Мы это сделаем достаточно просто, мы просто запустим компиляцию и будем идти по ошибкам компиляции. Например, первое место, где компилятор не знает, что такое `DateComparison` — это файл `condition_parse.cpp`, в котором реализована это функция `ParseCondition`, разбирающая условия. В этом файле мы укажем полное имя класса, потому что этот файл, так сказать, является клиентом нашего набора узлов, и здесь мы используем их полные имена. Когда мы начинали работу, у нас было `DateComparisonNode`, а теперь будет `Nodes::DateComparison`, так что здесь код изменился совсем чуть-чуть.

И так далее.

В итоге мы просто уменьшаем размеры некоторых файлов в байтах и упрощаем его чтение, потому что нам надо меньше читать дублирующиеся суффиксы. Также, именование классов типа `Nodes::DateComparison` в коде, который их использует, проще понимается, потому что эти «::» подчеркивают структуру.

Мы посмотрели, как с помощью пространства имён можно улучшать структурирование кода. Мы показали, что объединение общих по смыслу классов и функций в пространстве имён подчеркивает логическую структуру кода, уменьшает размер некоторых файлов и может упрощать чтение имён классов, если они длинные.

2.11 Рекомендации по использованию пространств имён

Давайте подведем итоги и сформулируем рекомендации по применению пространства имён в ваших проектах.

- **Пространство имён имеет смысл применять только в больших проектах**, потому что если у вас маленькая программа, которая состоит из одного, двух, максимум трех файлов, то конечно вряд ли у вас возникнет конфликт имён. Когда же у вас большой проект на десятки, сотни, а то и тысячи файлов, то вероятность возникновения конфликтов имён там довольно высока, и поэтому здесь уже возникает смысл использовать пространство имён.

- Пусть вы создали какую-то библиотеку, то есть какой-то обособленный набор функций и классов, который решает не одну какую-то конкретную специфическую задачу, а какой-то набор задач; пусть вы хотите поделиться ею с миром. Тогда **обязательно оберните функции и классы в вашей библиотеке в пространство имён**, для того, чтобы у других пользователей не возникало конфликтов имён, как это было в наших библиотеках по работе с форматами XML и JSON.

- `using`-декларации и директивы `using namespace` позволяют уменьшить объем кода, но при этом повышают вероятность возникновения конфликтов имён, поэтому ими надо пользоваться с осторожностью и стараться минимизировать область их действия (область действия `using`-декларации и директивы `using namespace` — это тот блок кода, в котором они объявлены). **Не надо использовать `using namespace` в заголовочных файлах**. А в `cpp`-файлах, при определенных условиях, это директива отлично работает и упрощает написание кода, поэтому ее **можно использовать в `cpp`-файлах, но с осторожностью, чтобы не возникали неожиданные конфликты имён**.

- `using namespace std` — это специальное пространство имён, в котором находится вся стандартная библиотека; и мы видели, как использование префикса перед каждым членом стандартной библиотеки может раздувать код, поэтому **используйте `using namespace std` в коде, который интенсивно работает со стандартной библиотекой, но если это не заголовочный файл**.

- можно **пробовать объединять общие по смыслу функции и классы в специальное пространство имён**, чтобы подчеркнуть логическую структуру вашей программы, сократить размер отдельных файлов, и в отдельных случаях упростить чтение и понимание вашего кода.

Указатель this

3.1 Присваивание объекта самому себе

Давайте начнем с примера. В «Красном поясе по C++» мы реализовывали шаблон `SimpleVector` — это такая сильно упрощенная реализация стандартного вектора. Мы с вами реализовали набор его конструкторов, деструктор, а также некоторые методы. И сейчас давайте рассмотрим довольно простой, на первый взгляд, способ применения шаблона `SimpleVector`.

```
template <typename T>
ostream& operator << (ostream& os, const SimpleVector<T>& rhs) {
    os << "Size = " << rhs.Size() << " Items:";
    for (const auto& x : rhs) {
        os << ' ' << x;
    }
    return os;
}

int main() {
    SimpleVector<int> source(5);
    for (size_t i = 0; i < source.Size(); ++i) {
        source[i] = i;
    }

    cout << source << endl;
    source = source;
    cout << source << endl;

    return 0;
}
```

Запускаем нашу программу и смотрим, что она вывела. Смотрите, какая интересная вещь: первый вывод ожидаемый, адекватный — размер вектора 5, элементы 0 1 2 3 4. Ровно те элементы, которые мы в него записали в цикле. А вот после присваивания самому себе почему-то наш вектор стал хранить какие-то странные значения. Размер у него не изменился, так и остался 5, а вот элементы почему-то вообще какие-то другие. То есть явно наш оператор копирующего присваивания работает неправильно в том случае, когда мы присваиваем объект самому себе.

Но у вас может возникнуть вопрос: а вообще есть ли смысл в присваивании объекта самому себе? Какая-то странная операция. Но на самом деле такое бывает. Например, у нас есть два

указателя, и оба они указывают на один и тот же объект. И мы эти два указателя разыменовываем и присваиваем один объект другому. Вот в такой ситуации, когда у нас есть только указатели и мы не знаем, на что они на самом деле указывают, может на самом деле произойти присваивание самому себе. Поэтому оно должно работать корректно.

И давайте посмотрим, как сейчас у нас реализован оператор копирующего присваивания в шаблоне `SimpleVector`. Устроен он довольно просто и, вообще говоря, ожидаемо. Мы первым делом освобождаем свои данные, которыми мы владеем, затем выделяем необходимое количество памяти, чтобы сохранить все элементы вектора `other`, копируем его размер, его емкость, и затем, с помощью алгоритма `copy` мы копируем к себе данные другого вектора.

```
template <typename T>
void SimpleVector<T>::operator=(const SimpleVector<T>& other) {
    delete[] data;
    data = new T[other.capacity];
    size = other.size;
    capacity = other.capacity;
    copy(other.begin(), other.end(), begin());
}
```

Вроде бы нормальная реализация, но из нее сразу очевидно, почему у нас некорректно работает присваивание себе. Потому что мы первым же делом удаляем собственные данные. А потом в алгоритме `copy` мы из этой освобожденной памяти читаем. Нам, вообще говоря, везет, что наша программа корректно завершается. Она могла бы падать с ошибкой. Таким образом, нам надо придумать, как поменять реализацию нашего оператора копирующего присваивания, чтобы он корректно обрабатывал присваивание самому себе. И кажется, самый лучший способ это сделать — это проверить, что объект `other`, который нам приходит в качестве параметра, не совпадает с объектом, которому выполняется присваивание. Потому что если он совпадает, то, вообще говоря, делать ничего не надо: мы присваиваем самому себе, свои данные можно не трогать. И у нас возникает вопрос: а как внутри метода класса `SimpleVector`, проверить, что объект `other` — это тот же самый объект, которому выполняется присваивание? В этом нам поможет указатель `this`.

3.2 Знакомство с this

`this` — это специальный указатель, который внутри методов класса указывает на текущий объект этого класса.

Давайте вернёмся к реализации `SimpleVector`, зайдём в его конструктор, который принимает размер, и в нём напишем

```
template <typename T>
SimpleVector<T>::SimpleVector(size_t size) : data(new T[size]), size(size), capacity(size) {
    cout << this << endl;
}
```

Объявим две переменные — `source` и `source2` — обе переменные типа `SimpleVector`. И выведем адреса этих переменных на консоль.

```
int main() {
    SimpleVector<int> source(5);
    SimpleVector<int> source2(5);

    cout << &source << ' ' << &source2 << endl;
}
```

Скомпилируем наш код. И запустим его. Мы видим, что указатель `this` действительно хранит в себе адрес текущего объекта.

Как же теперь нам применить указатель `this`, чтобы решить нашу проблему и ничего не делать, если мы выполняем присваивание самому себе? Очень просто. Пойдём в оператор присваивания и напишем

```
template <typename T>
void SimpleVector<T>::operator=(const SimpleVector<T>& other) {
    if (this != &other) {
        delete[] data;
        data = new T[other.capacity];
        size = other.size;
        capacity = other.capacity;
        copy(other.begin(), other.end(), begin());
    }
}
```

Итак, мы видим, что всё теперь работает. Теперь наш код работает правильно, и после присваивания вектора самому себе он не меняется и всё так же выводит те элементы, которые мы в него записали.

И давайте сделаем ещё одну вещь. Мы заглянем в оператор перемещающего присваивания. С ним же, на самом деле, может быть та же самая проблема в строчке

```
    source = move(source);
```

Поэтому давайте зайдём в оператор перемещающего присваивания и сделаем такое же условие

```
template <typename T>
void SimpleVector<T>::operator=(SimpleVector<T>&& other) {
    if (this != &other) {
        delete[] data;
        data = new T[other.capacity];
        size = other.size;
        capacity = other.capacity;

        other.data = nullptr;
        other.size = other.capacity = 0;
    }
}
```

Всё у нас хорошо работает, всё компилируется. Отлично. Таким образом, с помощью указателя `this` мы решили проблему присваивания самому себе в классе `SimpleVector`.

3.3 Ссылка на себя

Давайте рассмотрим такой пример. Допустим, у нас есть много переменных типа `int`. Вот давайте объявим их: переменные `a, b, c, d, e`. И мы хотим вот эти все переменные проинициализировать нулём. Мы это можем сделать так:

```
int main() {
    int a, b, c, d, e;
    a = b = c = d = e = 0;
    return 0;
}
```

Это скомпилируется.

А теперь давайте проверим, можно ли сделать то же самое для нашего шаблона `SimpleVector`. Объявляем эти же переменные, делаем у них тип `SimpleVector` и в присваивании убираем присваивание нуля. И мы поменяли наш код так, что мы переменным `a, b, c, d` присваиваем переменную `e`.

```
int main() {
    SimpleVector<int> a, b, c, d, e;
    a = b = c = d = e;
    return 0;
}
```

Запускаем компиляцию, и у нас не компилируется. При этом смотрим, что нам пишет компилятор. Он говорит, что у него нет оператора присваивания для типов `SimpleVector` от `int` и `void`. Таким образом, у нас не получилось сделать вот такое цепное присваивание.

Однако фундаментальной особенностью языка C++ является возможность создавать пользовательские типы таким образом, чтобы их можно было использовать точно так же, как и встроенные типы. Поэтому должна быть возможность реализовать вот такое вот цепное присваивание для нашего класса `SimpleVector`. И давайте разбираться, как это сделать.

Вернемся к примеру с переменными типа `int`. **Операция присваивания правоассоциативна**. Что это значит? Это значит, что когда у нас выполняется вот такое вот цепное присваивание, то сначала выполняется самое правое присваивание в команде, затем результат этого присваивания присваивается второй справа переменной и так далее.

```
int main() {
    int a, b, c, d, e;
    (a = (b = (c = (d = (e = 0)))));
    return 0;
}
```

Какой здесь самый важный момент? Мы сказали фразу: **результат присваивания**. То есть присваивание должно что-то возвращать. И оно должно возвращать нечто, что мы потом присвоим следующей переменной.

Давайте посмотрим, что возвращает оператор присваивания в нашем классе `SimpleVector`. Он возвращает `void`. Это тот самый `void`, который у нас был в ошибке компиляции в сообщении

компилятора, когда он говорил, что не может для нашего `SimpleVector` сделать цепное присваивание. Значит, нам отсюда, из оператора присваивания, нужно что-то возвращать, чтобы это что-то можно было присвоить следующему объекту. А что мы хотим присвоить? На самом деле себя. То есть тот объект, которому мы выполнили присваивание.

Для начала поменяем тип возвращаемого значения на ссылку на `SimpleVector`. А вернуть ссылку на себя довольно просто: у нас же есть указатель `this`, который указывает на текущий объект. А если у нас есть указатель, то получить ссылку довольно просто (нужно разыменовать этот указатель):

```
template <typename T>
SimpleVector<T>& SimpleVector<T>::operator=(SimpleVector<T>&& other) {
    if (this != &other) {
        delete[] data;
        data = new T[other.capacity];
        size = other.size;
        capacity = other.capacity;

        other.data = nullptr;
        other.size = other.capacity = 0;
    }
    return *this;
}
```

Также нужно не забыть поменять объявление метода (поменяв тип возвращаемого значения). Снова объявляем наши переменные, даём им тип `SimpleVector`.

```
int main() {
    SimpleVector<int> a, b, c, d, e;
    a = b = c = d = e;
    return 0;
}
```

Запускаем компиляцию. Код компилируется. Мы можем даже проверить, что он работает, например, вот таким образом. Мы объявим переменную `e` отдельно и сконструируем ее конструктором, который получает размер. А в конце выведем размер вектора `a`.

```
int main() {
    SimpleVector<int> e(5);
    SimpleVector<int> a, b, c, d;
    a = b = c = d = e;
    cout << a.Size() << endl;
    return 0;
}
```

Скомпилируем, запустим, видим, что вывелась 5.

И у нас в коде осталось еще одно присваивание — перемещающее. И с ним нужно сделать то же самое.

```
template <typename T>
```

```
SimpleVector<T>& SimpleVector<T>::operator=(SimpleVector<T>&& other) {
    if (this != &other) {
        delete[] data;
        data = new T[other.capacity];
        size = other.size;
        capacity = other.capacity;

        other.data = nullptr;
        other.size = other.capacity = 0;
    }
    return *this;
}
```

Запускаем компиляцию, у нас всё хорошо.

Вообще говоря, в стандартной библиотеке C++ принято, чтобы операторы присваивания возвращали ссылку на себя. Это делается для того, чтобы работало вот такое цепное присваивание. Мы рекомендуем вам следовать правилам, принятым в стандартной библиотеке, и из операторов присваивания всегда возвращать ссылку на себя.

3.4 this как неявный параметр методов класса

Давайте рассмотрим очень простой код на C++. У нас есть переменная `x`, присваиваем ей значение 3 и выводим. Потом присваиваем значение 8 и снова выводим.

```
int main() {
    int x;

    x = 3;
    cout << x << ' ';
    x = 8;
    cout << x << ' ';
}
```

Давайте запустим программу и увидим что всё работает. Что здесь важно? Что всякий раз, когда мы пишем «`x` равно чему-нибудь», мы записываем это значение в переменную `x`, которая объявлена в начале функции `main`.

Теперь давайте рассмотрим класс `ValueHolder`. В данном случае это структура, у этого класса есть поле `x` и метод `SetValue`, который принимает какое-то значение, записывает в поле `x` и выводит значение этого самого поля.

```
struct ValueHolder {
    int x;

    void SetValue(int value) {
        x = value;
        cout << x << ' ';
    }
};
```

И давайте мы воспользуемся классом `ValueHolder`.

```
int main() {
    ValueHolder a, b;
    a.SetValue(3);
    b.SetValue(5);
}
```

Компилируем, запускаем и в консоль вывелось 3 и 5. Ожидаемо. А как же компилятор догадывается в какую именно переменную, и в какую именно область памяти нужно записать значение при выполнении команды `x = value;`? Ведь в примере с целочисленными переменными все было понятно: есть переменные функция `main`, и мы туда все время записываем. Но здесь у нас есть только одна строчка кода.

Однако, в зависимости от того, для какой переменной `a` или `b` мы вызываем эту команду, значения записываются в разные области памяти. Как же это работает? Как компилятор догадывается, куда писать?

Дело в том, что под капотом указатель `this` является неявным параметром всех методов классов, то есть когда компилятор компилирует ваш код, то он генерирует функцию наподобие следующей.

```
void SetValue(ValueHolder* this_, int Value) {
    this_->x = value;
    cout << this_->x << ' ';
}
```

Таким образом когда вот здесь мы вызываем `SetValue` для `a` и `SetValue` для `b`, в эту функцию передаются разные указатели, и по этим указателей мы обращаемся к разным ячейкам памяти, хранящим поле `x`.

На самом деле, в некоторых языках программирования, например, в Python, даже требуется в методы явно передавать ссылку или указатель на текущий объект класса. Но в C++ этот указатель передается неявно.

Давайте вернемся к традиционному синтаксису для классов. Когда мы обращаемся к полю `x`, мы на самом деле можем написать вот так:

```
struct ValueHolder {
    int x;

    void SetValue(int value) {
```

```
        this->x = value;
        cout << this->x << ' ';
    }
};
```

Таким образом, **указатель `this` является неявным параметром всех методов класса**. И когда внутри метода мы обращаемся к какому-то полю, например, `field`, то на самом деле мы обращаемся к области памяти `this->field`.