3.3. Структуры. Классы

Структуры

3.3.1. Зачем нужны структуры?

Ядром ООП является создание программистом собственных типов данных. Для начала следует обсудить вопрос, зачем вообще такое может понадобиться.

Допустим, программа должна работать с видеолекциями, в том числе с их названиями и длительностями (в секундах). Можно написать такую функцию, которая будет работать с данными характеристиками видеолекции:

```
void PrintLecture(const string& title,
                  int duration) {
  cout << "Title: "    << title <<
       ", duration: " << duration << "\n";
}
```

Эта функция выводит на экран информацию о видеолекции, принимая в качестве параметров ее название и продолжительность.

Если нужно вывести информацию о курсе, то есть о серии видеолекций, можно написать функцию PrintCourse. Эта функция должна принять на вход набор видеолекций, но поскольку информация о них хранится в виде характеристик, функция принимает в качестве параметров вектор названий и вектор длительностей видеолекций:

```
PrintCourse(const vector<string>& titles,
            const vector<int>& durations) {
  int i = 0;
  while (i < titles.size()) {
    PrintLecture(titles[i], durations[i]);
    ++i;
  }
}
```

Может возникнуть необходимость хранить и обрабатывать дополнительно имя лекторов, которые читают лекции. Код постепенно разбухает. В функцию PrintLecture нужно передавать еще один параметр:

```
void PrintLecture(const string& title,
                  int duration,
                  const string& author) {
```

```
  cout << "Title: "    << title <<
        ", duration: " << duration <<
        ", author: "   << author << "\n";
}
```

Функцию PrintCourse также нужно модифицировать:

```
void PrintCourse(const vector<string>& titles,
                 const vector<int>& durations,
                 const vector<string>& authors) {
  int i = 0;
  while (i < titles.size()) {
    PrintLecture(titles[i],
                 durations[i],
                 authors[i]);
    ++i;
  }
}
```

Основные недостатки представленного подхода:

- Хочется работать с объектами (лекциями), а не отдельно с каждой из составляющих характеристик (название, продолжительность, имя лектора). Другими словами, в коде неправильно выражается намерение: вместо того, чтобы передать в качестве параметра лекцию, передается название, продолжительность и имя автора.
- При добавлении или удалении характеристики нужно менять заголовки функций, а также все их вызовы.
- Отсутствует единый список характеристик. Не существует единого места, где указаны все характеристики объекта.

3.3.2. Структуры

Для создания нового типа данных используется ключевое слово struct. После него идет название нового типа данных, а затем в фигурных скобках перечисляются поля.

```
struct Lecture {  // Составной тип из 3 полей
  string title;
  int duration;
  string author;
};
```

Синтаксис объявления полей похож на синтаксис объявления переменных.

Обратиться к определенному полю объекта можно написав после имени переменной точку, после которой записывается название требуемого поля. Теперь можно переписать функции, чтобы они использовали новый тип данных:

```
void PrintLecture(const Lecture& lecture) {
  cout << "Title: "    << lecture.title <<
        ", duration: " << lecture.duration <<
        ", author: "   << lecture.author << "\n";
}
```

Здесь лекция передается по ссылке, чтобы избежать копирования.

В функции PrintCourse все еще понятнее: она будет принимать то, что и задумывалось изначально — набор видеолекций, в виде вектора из элементов типа Lecture:

```
void PrintCourse(
    const vector<Lecture>& lectures) {
  for (Lecture lecture : lectures) {
    PrintLecture(lecture);
  }
}
```

Особо отметим, что хоть и был определен пользовательский тип данных, можно создавать контейнеры, элементы которого будут иметь такой тип. Более того, итерирование в данном случае уже можно производить с помощью цикла range-based for, а не while.

Код стал более понятным, более компактным, лучше поддерживаемым. Если нужно добавить новую характеристику видеолекции, достаточно поправить определение структуры, а менять заголовки и вызовы функций не потребуется. Разве что может понадобится добавление вывода нового поля в функцию PrintLecture, что вполне ожидаемо.

3.3.3. Создание структур

Существует несколько способов создания переменной пользовательского типа с определенными значениями полей. Самый простой из них — объявить переменную желаемого типа, а после — указать значения каждого поля вручную. Например:

```
Lecture lecture1;
lecture1.title = "OOP";
lecture1.duration = 5400;
lecture1.author = "Anton";
```

Проблема такого способа заключается в том, что название переменной постоянно повторяется, а также такой код занимает целых 4 строчки даже в таком простом примере.

Более короткий способ создания структур с требуемыми значениями полей: при инициализации после знака равно записать в фигурных скобках желаемые значения полей в том же порядке, в котором они были объявлены:

```
Lecture lecture2 = {"OOP", 5400, "Anton"};
```

Более того, такой способ годится даже для вызова функций без создания промежуточных переменных:

```
PrintLecture({"OOP", 5400, "Anton"});
```

Точно также, с помощью фигурных скобок можно вернуть объект из функции:

```
Lecture GetCurrentLecture() {
  return {"OOP", 5400, "Anton"};
}

Lecture current_lecture = GetCurrentLecture();
```

3.3.4. Вложенные структуры

Поле некоторого пользовательского типа может иметь тип, который также является пользовательским. Другими словами, можно создавать вложенные структуры.

Например, если название лекции представляет собой не одну, а три строки (название специализации, курса и название недели), можно создать структуру LectureTitle:

```
struct LectureTitle {
  string specialization;
  string course;
  string week;
};

struct DetailedLecture {
  LectureTitle title;
  int duration;
};
```

Новый тип можно использовать везде, где можно было использовать встроенные типы языка C++. В том числе указывать как тип поля при создании других типов.

Создать вложенную структуру можно используя уже известный синтаксис:

```
LectureTitle title = {"C++", "White belt", "OOP"};
DetailedLecture lecture1 = {title, 5400};
```

Этот код можно записать короче и без использования временной переменной:

```
DetailedLecture lecture2 = {
    {"C++", "White belt", "OOP"},
    5400
};
```

Обращаться к внутренним полям можно ожидаемым образом:

```
cout << lecture2.title.specialization << "\n";
// Выведет «C++»
```

3.3.5. Область видимости типа

Использовать тип можно только после его объявления. Поэтому поменять местами объявления DetailedLecture и LectureTitle не получится: будет ошибка компиляции.

```
struct DetailedLecture {
  LectureTitle title;  // Не компилируется:
  int duration;        // тип LectureTitle
};                     // пока неизвестен

struct LectureTitle {
  string specialization;
  string course;
  string week;
};
```

Классы

3.3.6. Приватная секция

Пусть требуется написать программу, которая работает с маршрутами между городами. Каждый маршрут будет представлять собой название

двух городов, где маршрут начинается и где маршрут заканчивается. Объявим структуру:

```
struct Route {
  string source;
  string destination;
};
```

Кроме того, пусть дана функция для расчета длины пути.

```
int ComputeDistance(
    const string& source,
    const string& destination);
```

Эта функция уже написана кем-то и ее реализация может быть достаточно тяжела: функция может в ходе исполнения обращаться к базе данных и запрашивать данные оттуда.

В любом случае, в программе иногда возникает необходимость вычислить длину маршрута. Каждый раз вычислять длину затратно, поэтому ее нужно где-то хранить. Можно создать еще одно поле в существующей структуре.

```
struct Route {
  string source;
  string destination;
  int length;
};
```

Теперь, в принципе, можно написать программу, которая будет делать то, что требуется, и она может отлично работать. Однако поле length доступно публично, то есть нельзя быть уверенным, что length — это расстояние между source и destination:

- Можно случайно изменить значение переменной length
- Можно изменить один из городов и забыть обновить значение length

Хочется минимизировать количество возможных ошибок при написании кода. Для этого нужно запретить прямой, то есть публичный, доступ к полям.

Таким образом можно объявить приватную секцию:

```
struct Route {
 private:
  string source;
  string destination;
  int length;
};
```

Теперь к данным полям нет доступа снаружи класса:

```
Route route;
route.source = "Moscow";
    // Раньше компилировалось, теперь нет
cout << route.length;
    // Так тоже нельзя: запрещён любой доступ
```

Теперь структура абсолютно бесполезна, потому что в публичном доступе ничего нет. Для того, чтобы обратиться к приватным полям, нужно использовать методы.

3.3.7. Методы

Можно дописать методы к структуре, чтобы она стала более функциональной:

```
struct Route {
 public:
  string GetSource() { return source; }
  string GetDestination() { return destination; }
  int GetLength() { return length; }

 private:
  string source;
  string destination;
  int length;
};
```

Методы очень похожи на функции, но привязаны к конкретному классу. И когда эти методы вызываются, они будут работать в контексте какого-то конкретного объекта.

Определение метода похоже на определение функции, но производится внутри класса. Нужно сначала записать возвращаемый тип, затем название метода, а после, в фигурных скобках, тело метода.

Теперь созданные методы можно использовать следующим образом:

```
Route route;

route.GetSource() = "Moscow";
    // Бесполезно, поле не изменится

cout << route.GetLength();
    // Так теперь можно: доступ на чтение
```

```
int destination_name_length =
    route.GetDestination().length();
    // И так можно
```

Отличия методов от функций:

- Методы вызваются в контексте конкретного объекта.
- Методы имеют доступ к приватным полям (и приватным методам) объекта. К ним можно обращаться просто по названию поля.

На самом деле, структура с добавленными приватной, публичной секциями и методами — это формально уже не структура, а класс. Поэтому вместо ключевого слова struct лучше использовать class:

```
class Route {  // class вместо struct
 public:
  string GetSource() { return source; }
  string GetDestination() { return destination; }
  int GetLength() { return length; }

 private:
  string source;
  string destination;
  int length;
};
```

Программа будет работать точно так же, как если бы это не делать. Но это увеличит читаемость кода, так как существует следующая договоренность:

Структура (struct) — набор публичных полей. Используется, если не нужно контролировать консистентность. Типичный пример структуры:

```
struct Point {
 double x;
 double y;
};
```

Класс (class) скрывает данные и предоставляет определенный интерфейс доступа к ним. Используется, если поля связаны друг с другом и эту связь нужно контролировать. Пример класса — класс Route, описанный выше.

3.3.8. Контроль консистентности

В обсуждаемом примере поля класса Route были сделаны приватными, чтобы использование класса было более безопасным. Планируется, что класс сам при необходимости будет, например, обновлять длину маршрута. Чтобы предоставить способ для изменения полей, нужно написать еще несколько публичных методов:

SetSource — позволяет изменить начало маршрута.

SetDestination — позволяет изменить пункт назначения.

В каждом из этих методов нужно не забыть обновить длину маршрута. Это лучше всего сделать с помощью метода UpdateLength, который будет доступен только внутри класса, то есть будет приватным методом.

В итоге код класса будет выглядеть следующим образом:

```
class Route {
 public:
  string GetSource() {
    return source;
  }
  string GetDestination() {
    return destination;
  }
  int GetLength() {
    return length;
  }
  void SetSource(const string& new_source) {
    source = new_source;
    UpdateLength();
  }
  void SetDestination(const string& new_destination) {
    destination = new_destination;
    UpdateLength();
  }

 private:
  void UpdateLength() {
    length = ComputeDistance(source, destination);
  }
  string source;
  string destination;
  int length;
};
```

Таким образом, создан полноценный класс, который можно использовать, например, так:

```
Route route;
route.SetSource("Moscow");
route.SetDestination("Dubna");
cout << "Route from " <<
    route.GetSource() << " to " <<
    route.GetDestination() << " is " <<
    route.GetLength() << " meters long";
```

Итак, смысловая связь между полями класса контролируется в методах.

3.3.9. Константные методы

Попробуем написать функцию, которая будет что-то делать с нашим классом. Например, функцию, которая распечатает информацию о маршруте:

```
void PrintRoute(const Route& route) {
  cout << route.GetSource() << " - " <<
     route.GetDestination() << endl;
}
```

Маршрут принимается по константной ссылке, чтобы лишний раз не копировать объект.

Создадим маршрут и вызовем эту функцию:

```
int main() {
  Route route;
  PrintRoute(route);
  return 0;
}
```

При попытке запуска кода появляется ошибка.

Дело в том, что в методе GetSource нигде явно не указано, что он не меняет объект. С другой стороны, в функцию PrintRoute объект route передается по константной ссылке, то есть функция PrintRoute не имеет право изменять этот объект. Поэтому компилятор не дает вызывать те методы, для которых не указано явно, что объект они не меняют.

Чтобы указать, что метод не меняет объект, нужно объявить метод константным. То есть дописать ключевое слово const:

```
class Route {
 public:
```

```
  string GetSource() const {
        return source;
  }
  string GetDestination() const {
    return destination;
  }
  int GetLength() const {
    return length;
  }
```

Также давайте добавим начало и конец маршрута, чтобы вывод был интереснее:

```
int main() {
  Route route;
  route.SetSource("Moscow");
  route.SetDestination("Vologda");
  PrintRoute(route); // Выведет Moscow - Vologda
  return 0;
}
```

Теперь все работает. Итак, константными следует объявлять все методы, которые не меняют объект.

Если попытаться объявить константным метод, который меняет объект, компилятор выдаст сообщение об ошибке. Сообщения об ошибках, как правило, понятны, но в случае ошибок с константностью — не всегда. Поэтому следует запомнить, что ошибка «passing ... discards qualifiers» значит, что имеет место проблема с константностью. Также нельзя вызывать не константные методы для объекта, переданного по константной ссылке.

Следующая функция переворачивает маршрут. Она принимает значение по не константной ссылке, потому что объект будет изменен.

```
void ReverseRoute(Route& route) {
  string old_source = route.GetSource();
  string old_destination = route.GetDestination();
  route.SetSource(old_destination);
  route.SetDestination(old_source);
}
```

Этот пример демонстрирует то, что по не константной ссылке можно вызывать как константные, так и не константные методы.

```
ReverseRoute(route);
PrintRoute(route);
```

3.3.10. Параметризованные конструкторы

Чтобы сделать классы более удобными в использовании, можно использовать так называемые конструкторы.

Допустим, нужно создать маршрут между конкретными городами. Можно сделать это, например, с помощью уже известного синтаксиса:

```
Route route;
route.SetSource("Zvenigorod");
route.SetDestination("Istra");
```

Недостаток такого способа: для такой простой задачи нужно написать три строчки кода. Избавиться от этого недостатка можно написав функцию, которая будет создавать маршрут:

```
Route BuildRoute(
    const string& source,
    const string& destination) {
  Route route;
  route.SetSource(source);
  route.SetDestination(destination);
  return route;
}

Route route = BuildRoute("Zvenigorod", "Istra");
```

Такое решение этой очень распространенной проблемы весьма искусственно и выглядит подозрительным. Действительно, имя BuildRoute не стандартизировано: может быть функция CreateTrain или MakeLecture. По названию класса становится невозможно понять, как называется та самая функция, которая создает объекты данного класса.

В C++ существует готовое решение для этой проблемы — конструкторы, которые уже встречались ранее для встроенных типов данных:

```
vector<string> names(5);  // Вектор из 5 пустых строк
string spaces(10, ' ');   // Строка из 10 пробелов
```

Хочется, чтобы и в случае пользовательского типа можно было сделать как-то похоже:

```
Route route("Zvenigorod", "Istra");  // Не умеем
```

Чтобы такой код работал, нужно написать конструктор.

Конструктор — это специальный метод класса без возвращаемого значения, название которого совпадает с названием класса.

Конструктор, который принимает названия двух городов, можно написать, вызывая в его теле методы SetSource и SetDestination:

```
class Route {
 public:
  Route(const string& new_source,
        const string& new_destination) {
    SetSource(new_source);
    SetDestination(new_destination);
  }
};

Route route("Zvenigorod", "Istra");
    // Теперь работает
cout << "Route from Zvenigorod to Istra " <<
    "has length " << route.GetLength() << "\n";
```

Строго говоря, в таком случае метод UpdateLength вызывается дважды, причем один раз еще до того, как значение конечного пункта маршрута не было установлено. Поэтому в конструкторе не стоит использовать методы, созданные для использования вне класса. Внутри конструктора можно просто проинициализировать поля нужными значениями непосредственно, а после этого вызывать метод UpdateLength всего один раз.

```
class Route {
 public:
  Route(const string& new_source,
        const string& new_destination) {
    source = new_source;
    destination = new_destination;
    UpdateLength();
  }
  // ...
};
```

3.3.11. Конструкторы по умолчанию, использование конструкторов

Если для класса был написан параметризованный конструктор, создание переменной без параметров уже не будет работать.

```
class Route {
 public:
  Route(const string& new_source,
        const string& new_destination) {
    source = new_source;
    destination = new_destination;
```

```
    UpdateLength();
  }
  // ...
};

Route route;  // Теперь не компилируется
```

Чтобы исправить это, нужно дописать так называемый конструктор по умолчанию.

```
class Route {
 public:
  Route() {}  // Раньше компилятор делал это сам
  Route(const string& new_source,
        const string& new_destination) {
    source = new_source;
    destination = new_destination;
    UpdateLength();
  }
};
```

Если по умолчанию не нужно как-то инициализировать поля, тело конструктора по умолчанию можно оставить пустым. Если для класса (никакой) конструктор не указан, компилятор создает пустой конструктор самостоятельно.

Если необходимо, чтобы по умолчанию поля были заполнены определенными значениями, это можно указать в конструкторе по умолчанию:

```
class Route {
 public:
  Route() {
    source = "Moscow";
    destination = "Saint Petersburg";
    UpdateLength();
  }
  // ...
};

Route route;  // Маршрут от Москвы до СПб
```

Если переменная объявляется без указания параметров, то используется конструктор по умолчанию:

```
Route route1;
    // По умолчанию: Москва - Петербург
```

Если после названия переменной в круглых скобках указаны некоторые параметры, то вызывается параметризованный конструктор:

```
Route route2("Zvenigorod", "Istra");
    // Параметризованный
```

Если маршрут по умолчанию нужно передать в функцию, которая принимает объект по константной ссылке:

```
void PrintRoute(const Route& route);
```

то в качестве объекта можно передать пустые фигурные скобки:

```
PrintRoute(Route()); // По умолчанию
PrintRoute({});      // Тип понятен из заголовка функции
```

Если же нужно передать произвольный объект, аргументы параметризованного конструктора можно перечислить в фигурных скобках без указания типа:

```
PrintRoute(Route("Zvenigorod", "Istra"));
PrintRoute({"Zvenigorod", "Istra"});
```

На самом деле, такой синтаксис можно использовать и для встроенных в язык функций и методов:

```
vector<Route> routes;
routes.push_back({"Zvenigorod", "Istra"});
```

А также, когда необходимо вернуть объект в результате работы функции:

```
Route GetRoute(bool is_empty) {
  if (is_empty) {
    return {};
  } else {
    return {"Zvenigorod", "Istra"};
  }
}
```

Компилятор уже видит, объект какого типа функция возвращает, поэтому в return параметры конструктора можно указать в фигурных скобках, или написать пустые фигурные скобки для использования конструктора по умолчанию.

Аналогично можно делать и в случае встроенных типов:

```
vector<int> GetNumbers(bool is_empty) {
  if (is_empty) {
    return {};
  } else {
    return {8, 6, 9, 6};
  }
}
```

3.3.12. Значения по умолчанию для полей структур

Как правило, конструкторы в структурах не нужны. Создавать объект можно и с помощью синтаксиса с фигурными скобками:

```
struct Lecture {
  string title;
  int duration;
};

Lecture lecture = {"ООР", 5400};
    // ОК, работало и без конструкторов
```

Но в некоторых случаях могло бы быть полезным использование конструктора по умолчанию для структур. Оказывается, что если нужен только конструктор по умолчанию, достаточно задать значения по умолчанию для полей:

```
struct Lecture {
  string title = "C++";
  int duration = 0;
};
```

Тогда при создании переменной без инициализации будут использоваться значения по умолчанию:

```
Lecture lecture;
cout << lecture.title << " " lecture,duration << "\n";
// Выведет <<C++ 0>>
```

При этом все еще доступен синтаксис с фигурными скобками:

```
Lecture lecture2 = {"ООР", 5400};
```

Также можно не указывать несколько последних полей:

```
Lecture lecture3 = {"ООР"};
```

В этом случае для них будут использоваться значения по умолчанию.

3.3.13. Деструкторы

Деструктор — специальный метод класса, который вызывается при уничтожении объекта. Его назначение — откат действий, сделанных в конструкторе и других методах: закрытие открытого файла и освобождение выделенной вручную памяти. Название деструктора состоит из символа тильды (~) и названия класса.

Также в деструкторе можно осуществлять любые другие действия, например, вывод информации. На практике писать деструктор самому нужно очень редко. Как правило, достаточно использовать деструктор, который генерируется компилятором.

Рассмотрим созданный ранее класс Route:

```
class Route {
public:
  Route() {
    source = "Moscow";
    destination = "Saint Petersburg";
    UpdateLength();
  }
  Route(const string& new_source,
        const string& new_destination) {
    source = new_source;
    destination = new_destination;
    UpdateLength();
  }
  string GetSource() const {
    return source;
  }
  string GetDestination() const {
    return destination;
  }
  int GetLength() const {
    return length;
  }
  void SetSource(const string& new_source) {
    source = new_source;
    UpdateLength();
  }
  void SetDestination(const string& new_destination) {
    destination = new_destination;
    UpdateLength();
  }

private:
  void UpdateLength() {
    length = ComputeDistance(source, destination);
  }
  string source;
  string destination;
```

```
  int length;
};
```

Для демонстрационных целей в качестве ComputeDistance можно использовать простую заглушку:

```
int ComputeDistance(const string& source,
                    const string& destination) {
  return source.length() - destination.length();
}
```

Реально же ComputeDistance может содержать запросы к базе данных, сложные вычисления и так далее, то есть может выполняться долго. Поэтому при написании программы имеет смысл минимизировать количество вызовов ComputeDistance.

Создадим лог вызовов функции ComputeDistance:

```
private:
  void UpdateLength() {
    length = ComputeDistance(source, destination);
    compute_distance_log.push_back(
        source + " - " + destination);
  }
  string source;
  string destination;
  int length;
  vector<string> compute_distance_log;
};
```

В деструкторе объекта теперь можно сделать так, чтобы этот лог выводился в печать перед уничтожением объекта.

```
class Route {
public:
  Route() {
    source = "Moscow";
    destination = "Saint Petersburg";
    UpdateLength();
  }
  Route(const string& new_source,
        const string& new_destination) {
    source = new_source;
    destination = new_destination;
    UpdateLength();
  }
```

```
  ~Route() {
    for (const string& entry : compute_distance_log) {
      cout << entry << "\n";
    }
  }
```

Теперь посмотрим, что выведет такой код:

```
Route route("Moscow", "Saint Petersburg");
route.SetSource("Vyborg");
route.SetDestination("Vologda");
```

```
Moscow — Saint Petersburg
Vyborg — Saint Petersburg
Vyborg — Vologda
```

3.3.14. Время жизни объекта

С помощью отладочной информации изучим то, как и когда уничтожаются объекты в разных ситуациях. Добавим отладочную печать во все конструкторы и деструкторы:

```
class Route {
public:
  Route() {
    source = "Moscow";
    destination = "Saint Petersburg";
    UpdateLength();
    cout << "Default constructed\n";
  }

  Route(const string& new_source,
        const string& new_destination) {
    source = new_source;
    destination = new_destination;
    UpdateLength();
    cout << "Constructed\n";
  }
  ~Route() {
    cout << "Destructed\n";
  }
  string GetSource() const {
    return source;
  }
```

```
    string GetDestination() const {
      return destination;
    }
    int GetLength() const {
      return length;
    }
    void SetSource(const string& new_source) {
      source = new_source;
      UpdateLength();
    }
    void SetDestination(const string& new_destination) {
      destination = new_destination;
      UpdateLength();
    }

  private:
    void UpdateLength() {
      length = ComputeDistance(source, destination);
    }
    string source;
    string destination;
    int length;
};
```

Выполним следующий код:

```
for (int i : {0, 1}) {
  cout << "Step " << i << ": " << 1 << "\n";
  Route route;
  cout << "Step " << i << ": " << 2 << "\n";
}
cout << "End\n";
```

Результат его выполнения:

```
Step 0: 1
Default constructed
Step 0: 2
Destructed
Step 1: 1
Default constructed
Step 1: 2
Destructed
End
```

На каждой итерации, как только объект выходит из своей зоны видимости, он уничтожается. При уничтожении объекта вызывается деструктор.

```
int main() {
  cout << 1 << "\n";
  Route first_route;
  if (false) {
    cout << 2 << "\n";
    return 0;
  }
  cout << 3 << "\n";
  Route second_route;
  cout << 4 << "\n";
  return 0;
}
```

```
1
Default constructed
3
Default constructed
4
Destructed
Destructed
```

Компилятор уничтожает объекты в обратном порядке относительно того, как они создавались. Объект, который был создан вторым, уничтожается первым.

Теперь отправим на выполнение такой код:

```
void Worthless(Route route) {
  cout << 2 << "\n";
}

int main() {
  cout << 1 << "\n";
  Worthless({});
  cout << 3 << "\n";
  return 0;
}
```

Результат будет следующий:

```
1
Default constructed
2
Destructed
3
```

```
Route GetRoute() {
  cout << 1 << "\n";
```

```
  return {};
}

int main() {
  Route route = GetRoute();
  cout << 2 << "\n";
  return 0;
}
```

```
1
Default constructed
2
Destructed
```

Если результат вызова функции не сохраняется, результат получается иной:

```
Route GetRoute() {
  cout << 1 << "\n";
  return {};
}

int main() {
  GetRoute();
  cout << 2 << "\n";
  return 0;
}
```

```
1
Default constructed
Destructed
2
```

Это связано с тем, что созданная в функции переменная не может быть использована после выполнения этой функции. Она никуда не была сохранена, поэтому она сразу же была уничтожена.