Андерс Шау Кнаттен

ХОРОШО ЛИ ВЫ ЗНАЕТЕ C++



ДМК ПРЕСС
издательство

УДК 004.438C++(075)
ББК 32.73.2я7
К53

Андерс Шау Кнаттен
К53 Хорошо ли вы знаете C++? / пер. с англ. А. А. Слинкина. – М.: ДМК Пресс, 2024. – 138 с.: ил.

ISBN 978-5-93700-351-5

Эта книга содержит 25 тщательно отобранных задач, которые призваны проверить, насколько хорошо вы понимаете современные версии языка. Каждая задача сопровождается ответом и объяснениями. Ожидается, что читатель знает основы языка C++ и владеет навыками программирования.

Книга будет полезна как опытным специалистам, желающим углубленно изучить C++, так и приступающим к освоению языка.

УДК 004.438C++(075)
ББК 32.73.2я7

Все права защищены. Любая часть этой книги не может быть воспроизведена в какой бы то ни было форме и какими бы то ни было средствами без письменного разрешения владельцев авторских прав.

ISBN (анг.) 979-8-88865-051-6
ISBN (рус.) 978-5-93700-351-5

Copyright © 2024 The Pragmatic Programmers, LLC.
© Оформление, издание, перевод, ДМК Пресс, 2024

Оглавление

Предисловие от издательства 7

Вступительное слово Ольве Маудала 8

Благодарности 10

Предисловие 11

Задача 1. Сколько градусов? 17

Задача 2. Теория струн 19

Задача 3. Хакнем планету 25

Задача 4. На пути к глобализации 29

Задача 5. Деструктивные отношения 33

Задача 6. Кто первый? 37

Задача 7. Все хорошее должно когда-то заканчиваться 41

Задача 8. Переместится или нет? 47

Задача 9. Подсчет копий 53

Задача 10. Странное присваивание 59

Задача 11. Когда наступила смерть? 63

Задача 12. Фальстарт 69

Задача 13. Постоянная борьба 73

Задача 14. Аристотелева сумма частей 77

Задача 15. Назад из будущего 81

Задача 16. Перегруженный контейнер 85

Задача 17. Строгий указ 89

Задача 18. Освобождаем помещение 93

Задача 19. Маленькая сумма 97

Задача 20. Монстры на марше 105

Задача 21. Измерение некоторых символов 109

Задача 22. Космический корабль-призрак 113

Задача 23. Доброе начало полдела откачало 119

Задача 24. Специальная теория струн 123

Задача 25. Слабо типизированная, сильно озадачивающая 129

Предметный указатель 132

Задача 1

Сколько градусов?

how-many-degrees.cpp

```
#include <iostream>

struct Degrees
{
    Degrees() : degrees_(0)
    {
        std::cout << "Default constructed\n";
    }
    Degrees(double degrees) : degrees_(degrees)
    {
        std::cout << "Constructed with " << degrees_ << "\n";
    }
    double degrees_;
};
struct Position
{
    Position() : latitude_{1} { longitude_ = Degrees{2}; }
    Degrees latitude_;
    Degrees longitude_;
};

int main()
{
    Position position;
}
```

Угадайте результат



Попробуйте угадать результат, не переворачивая страницу.

Программа печатает следующий результат:

```
Constructed with 1
Default constructed
Constructed with 2
```

Обсуждение

В структуре `Position` два члена типа `Degrees`. Один инициализируется в списке инициализации членов, второй в теле конструктора. Тогда почему мы видим *три* сконструированных объекта `Degrees`?

Все члены класс инициализируются до входа в тело конструктора. Если вы явно инициализируете член, как мы поступили с `latitude_`, то именно эта инициализация и будет использована. Если нет, как в случае с `longitude_`, то член будет инициализирован по умолчанию, поэтому мы видим, что напечатана строка `Default constructed`. После того как все члены инициализированы, мы входим в тело конструктора, инициализируем временный объект `Degrees{2}`, а затем присваиваем его копированием ранее инициализированному по умолчанию члену `longitude_`.

Заметим, что вместо явной инициализации члена в списке инициализации членов можно было бы использовать инициализатор по умолчанию прямо в объявлении члена:

examples/default-member-initializer/default-member-initializer.cpp

```
struct Position
{
    Position() { longitude = Degrees{2}; }
    Degrees latitude{1}; // инициализатор члена по умолчанию
    Degrees longitude;
};
```

Использование инициализатора члена по умолчанию может быть полезной идеей, если конструкторов несколько. Тогда не нужно помнить о включении этого члена в каждый список инициализации членов.

Для дополнительного чтения

Конструкторы и списки инициализации членов
https://en.cppreference.com/w/cpp/language/constructor.

Задача 2

Теория струн

string-theory.cpp

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

void serialize(const void*) { std::cout << "const void*"; }
void serialize(const std::string&) { std::cout << "const string&"; }
int main()
{
    serialize("hello world");
}
```

Угадайте результат



Попробуйте угадать результат, не переворачивая страницу.

Программа печатает следующий результат:

```
const void*
```

Обсуждение

Почему передача строки функции `serialize` приводит к вызову перегруженного варианта, который принимает указатель на `void`, а не к вызову варианта, принимающего строку?

Когда мы вызываем функцию, имеющую несколько перегруженных вариантов, компилятор применяет процесс *разрешения перегрузки*, чтобы понять, какой вариант самый подходящий. Для этого компилятор пытается преобразовать каждый аргумент функции к типу соответствующего параметра для каждого перегруженного варианта. Одни преобразования считаются лучше других, а самое лучшее – когда аргумент уже имеет правильный тип.

Перегруженные варианты, в которых все аргументы можно успешно преобразовать, помещаются в множество *подходящих функций*. Затем компилятор должен определить, какой вариант выбрать из этого множества. Если некоторый перегруженный вариант имеет лучшее преобразование, чем другие, хотя бы для одного аргумента и не худшее для всех остальных, то этот вариант считается *лучшей подходящей функцией* и является результатом процесса разрешения перегрузки. Если ни один вариант не лучше всех остальных, то вызов считается недопустимым и не компилируется.

Рассмотрим пример:

```
serialize(int, int); // 1
serialize(float, int); // 2
```

При таких двух перегруженных вариантах предположим, что имеется вызов:

```
serialize(1, 2);
```

Оба перегруженных варианта `serialize` являются подходящими. Но для первого преобразование первого аргумента лучше (`int` → `int` лучше, чем `int` → `float`), а для второго – не хуже (`int` → `int` в обоих вариантах), поэтому он выбирается как лучшая подходящая функция.

Задача немного проще этого примера, потому что оба перегруженных варианта `serialize` имеют только один параметр. Первый принимает `const void *`, второй – `const std::string&`. Как выглядит преобразование для каждого варианта? `std::string` – класс из стандартной библиотеки. Обычно память для него выделяется из кучи (если только строка не очень мала), а строка может расти или модифицироваться каким-то другим способом во время выполнения.

Однако строка `"hello world"` не объект класса `std::string`, а простой строковый литерал. Строковые литералы – это массивы `char` в стиле C, которые включены в двоичный файл компоновщиком и не могут быть модифицированы во время выполнения. Строковый литерал имеет тип «массив из `n` элементов типа `const char`». Литерал `"hello world"` состоит из 11 символов плюс завершающий `\0`, т. е. имеет тип «массив из `12` элементов типа `const char`».

Поскольку аргумент `"hello world"` принадлежит не типу `const void*` и не типу `std::string`, а типу «массив из `12` элементов типа `const char`», то для обоих перегруженных вариантов необходимо преобразование. Если существует неявное преобразование типа фактического аргумента к типу формального параметра, то перегруженный вариант добавляется в множество подходящих функций. В противном случае вариант игнорируется.

Рассмотрим первый перегруженный вариант и выясним, можно ли неявно преобразовать «массив из `12` элементов типа `const char`» в `const void *`. Первым делом массив преобразуется в указатель. Любой ««массив из `N` элементов типа `T`» можно преобразовать в «указатель на `T`», указывающий на его первый элемент. Следовательно, наш «массив из `12` элементов типа `const char`» преобразуется в «указатель на `const char`».

Далее, любой «указатель на *cv* `T`» (где *cv* означает `const`, `volatile`, `const volatile` или отсутствие любого из этих модификаторов) можно преобразовать в «указатель на *cv* `void`». Стало быть, наш «указатель на `const char`» преобразуется в «указатель на `const void`», т. е. именно то, чего ожидает первый перегруженный вариант.

Заметим, что в этой последовательности преобразований не было ни конструкторов, ни функций преобразования. То есть это *последовательность стандартных преобразований*, а не *последовательность определенных пользователем преобразований*. Это окажется важно позже.

Теперь рассмотрим второй перегруженный вариант и выясним, можно ли неявно преобразовать «массив из `12` элементов типа `const`

char» в «ссылку на const std::string». В типе std::string имеется конструктор std::string(const char* s), которым мы можем воспользоваться. Сначала преобразуем «массив из 12 элементов типа const char» в «указатель на const char», как и раньше. Затем передадим его конструктору std::string и получим в ответ объект std::string, содержащий копию строкового литерала. Параметр типа const std::string& можно непосредственно связать с нашим аргументом типа std::string.

Заметим, что для этого нам пришлось использовать конструктор, т. е. мы имеем последовательность определенных пользователем преобразований, а не последовательность стандартных преобразований. Неважно, что std::string – тип из стандартной библиотеки, он все равно считается определенным пользователем. Правила одинаковы и для вас, и для стандартной библиотеки.

Итак, компилятор нашел допустимую последовательность преобразований нашего «массива из 12 элементов типа const char» в тип параметра каждого перегруженного варианта и должен решить, как последовательность лучше:

```
Последовательность преобразований для void serialize(const void*)

const char[12] → const char * → const void *   Стандартное преобразование,
                                               новые объекты не создаются

Последовательность преобразований для serialize(const std::string&)

const char[12] → const char * → std::string    Определенное пользователем
                                               преобразование, создается
                                               объект типа std::string!
```

Для вызова варианта с const void* нам нужна только последовательность стандартных преобразований (верхняя). А для вызова варианта с std::string нужна последовательность определенных пользователем преобразований (нижняя), в которой встречается создание нового временного объекта std::string. Последовательность стандартных преобразований всегда *лучше* последовательности определенных пользователем преобразований, поэтому вызывается первый перегруженный вариант и печатается const void*.

Для дополнительного чтения

Разрешение перегрузки
https://en.cppreference.com/w/cpp/language/overload_resolution.

std::string
https://en.cppreference.com/w/cpp/string/basic_string.

Строковый литерал
https://en.cppreference.com/w/cpp/language/string_literal.

Задача 3

Хакнем планету

hack-the-planet.cpp

```
#include <iostream>

int getUserId() { return 1337; }
void restrictedTask1()
{
    int id = getUserId();
    if (id == 1337) { std::cout << "did task 1\n"; }
}
void restrictedTask2()
{
    int id;
    if (id == 1337) { std::cout << "did task 2\n"; }
}
int main() {
    restrictedTask1();
    restrictedTask2();
}
```

Угадайте результат



Попробуйте угадать результат, не переворачивая страницу.

Поведение программы не определено! Но она может напечатать такой результат:

```
did task 1
did task 1
```

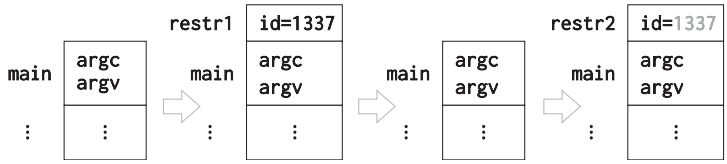
Обсуждение

Переменная `id` в функции `restrictedTask2` не была инициализирована, ее значение неизвестно. Использование ее значения и есть неопределенное поведение. Когда поведение программы не определено, может случиться все что угодно; стандарт C++ не дает никаких гарантий. Даже та часть программы, которая *предшествует* моменту, когда мы прочитали значение `id`, не определена!

Однако если вы запустите эту программу на своем компьютере, то она, скорее всего, напечатает оба сообщения, `did task 1` и `did task 2`, по крайней мере, если была откомпилирована без оптимизации. То есть значение `1337` волшебным образом телепортировалось из `restrictedTask1` в `restrictedTask2`! Как такое могло произойти?

В большинстве систем для размещения локальных переменных используется стек. В функции `restrictedTask1` имеется локальная переменная `id`, для которой она выделяет место в своем кадре стека. Так получилось, что в функции `restrictedTask2` столько же локальных переменных и точно таких же типов (одна типа `int`), поэтому структура ее кадра стека в точности такая же, как у `restrictedTask1`.

Пока мы находились в `main`, стек вырос до определенной точки – см. самую левую часть рисунка ниже. После этого мы вызываем `restrictedTask1`, стек растет, `restrictedTask1` резервирует в своем кадре стека место для `id` и инициализирует ее значением `1337`. Затем управление возвращается `main`, и стек снова сжимается.



Далее мы вызываем `restrictedTask2`, стек снова растет, и `restrictedTask2` резервирует в своем кадре стека место для `id`, но не инициализирует ее. Однако кадр стека `restrictedTask2` оказывается точно там же, где раньше был кадр стека `restrictedTask1`. Между вызовами функций стек не очищается, поэтому содержимое кадра