Лекция 06 — Lvalue-ссылки

Содержание

"https://www.youtube.com/watch?v=dQw4w9WgXcQ"
— Бьёрн Страуструп

Ссылки

Мотивация

Вернёмся к поднимавшемуся ранее вопросу реализации функции `swap`.
Вспомним, как мы сделали это с использованием *указателей*:

```
void swap(int* pi, int* pj) {
    int temp = *pi;
    *pi = *pj;
    *pj = temp;
}
```

К этому подходу нет, в сущности, никаких вопросов, кроме одного: *что произойдёт, если какой-либо из указателей окажется нулевым?*

Внимательный читатель знает, что это вызовет ошибку сегментации, однако даже если бы мы не имели представления о механизмах разыменования указателей, было бы достаточно ясно, что ничем хорошим это не закончится.
Нетрудно прийти к, например, следующим вариантам решения этой проблемы:

1. Использовать `assert`

> Этот подход хорош тем, что мы всегда узнаем, если функция получит некорректный набор входных данных, однако нам далеко не всегда хочется, чтобы при этом происходило завершение работы программы.

2. Изменить логику работы функции

> Подразумевается, что мы изменяем внутреннюю логику функции так, чтобы обрабатывать невалидные наборы входных данных отдельно от валидных. Программа при этом не завершается и критической ошибки не возникает, однако поведение функции значительно усложняется, причём не всегда очевидным образом, что порой может привести к ещё более печальным последствиям, чем ошибка сегментации.

3. Проверять валидность входных данных перед вызовом функции

> Этот подход лишён минусов предыдущих, однако, конечно, имеет свои собственные. Помимо того, что это усложняет и загрязняет код, мы сталкиваемся и с важной идеологической проблемой: проверка контракта на функцию происходит за её пределами. С точки зрения «хорошего тона» написания кода это большой недостаток.

4. Использовать коды возврата

> Этот подход совмещает все преимущества предыдущих, при этом обладая минимальным количеством собственных недостатков. Он не прерывает исполнение программы, позволяет точно определять место возникновения ошибки и моментально её обработать, при этом не вынося лишнюю логику за пределы функции, хотя всё ещё делает код программы более громоздким.

Каждое из этих решений обладает рядом преимуществ и недостатков, которые ограничивают область их применения. Помимо этого, все из них неизбежно усложняют код. Мы же, в свою очередь, хотим иметь в своём арсенале настолько универсальный и лаконичный инструмент, насколько это возможно. Именно в этот момент мы впервые сталкиваемся с таким понятием, как **ссылка**.

Концепция ссылки

Здесь будут рассматриваться только так называемые lvalue-ссылки — ровно те объекты, которые будут описаны далее. Хотя на первый взгляд может показаться, что rvalue-ссылки логично было бы рассмотреть в совокупности, на самом деле они имеют совершенно иную область применения, являясь важной частью семантики перемещения, поэтому будут отдельно рассмотрены позже.

Итак,

> **Info**
>
> **Ссылка** — это *псевдоним* объекта.

Это не столько строгое определение, сколько способ, которым нужно воспринимать ссылки. Иному читателю такая формулировка может показаться чересчур абстрактной, поэтому, вероятно, она требует пояснения.

Семантически ссылка в действительности не является ничем более сложным, чем другое название для существующей переменной. Она не представляет собой ни новый объект, ни копию исходного; ссылка — это *непосредственно сам объект*, на который она ссылается.
В C++ не существует синтаксиса, который позволял бы оперировать ссылками отдельно от объектов, к которым они привязаны. Эта «нематериальность» ссылки позволяет нам уже сейчас сформулировать ряд особенностей, отличающих её от указателя:

- о ссылке можно думать как о константной сущности (хотя формально она таковой и не является);
- ссылка *обязана* быть определена в момент объявления, поскольку не может ни на что не указывать;
- нельзя завести указатель на ссылку (но в определенных ситуациях имеет место «ссылка на ссылку» — см. раздел Reference collapsing в статье Reference declaration на cppreference.com).

С точки же зрения C++, ссылка на объект типа `T` — это переменная[1] ссылочного типа `T&`.
Когда компилятор встречает такую переменную, у него есть две опции:

1. Воспринять её как псевдоним и устранить, если это возможно:

```
// Исходный код
int x = 238;
int& y = x;
y++;

// Код после компиляции
int x = 238;
x++;
```

Сопутствующие этому процессу оптимизации могут включать в себя, например, инлайнинг[2] функций, удаление временных неиспользуемых ссылок и проч.

2. Воспринять её как *указатель*:

```
// Исходный код
void print(int& var) {
    std::cout << var;
}

int main() {
    int x;
    print(x);
}

// Код после компиляции
void print(int* var) {
    std::cout << *var;
}

int main() {
    int x;
    print(&x);
}
```

Поскольку ранее мы отметили, что ссылка как таковая не является объектом, можно было бы полагать, что она не занимает никакой памяти, однако только что была приведена причина, по которой это не отражает действительности. Убедиться в том, что ссылка может занимать память, легко:

```
struct s {
    char& ref;
};

int main() {
    std::cout << sizeof(s); // 8, а не 1!
}
```

> ⚠ **Warning**
>
> Память, которую может занимать ссылка, не превышает размера указателя.

Классификация ссылок

Константные ссылки

Общие представления

Если при попытке объявить ссылку на константный объект опустить квалификатор `const`, мы неизбежно столкнёмся с ошибкой компиляции:

```
const int x = 238;
int& ref = x; // CE!
```

Такое поведение совершенно предсказуемо, поскольку ссылка на объект типа `const T` по определению должна иметь ссылочный тип `const T&`.

Более интересна обратная ситуация: объявление константной ссылки на неконстантный объект:

```
int x = 238;
const int& ref = x; // OK!
```

Это не вызовет ошибки и даже будет работать достаточно естественным образом. Через константную ссылку нельзя модифицировать объект, на которой она ссылается; это означает, что для встроенных типов через такую ссылку запрещено присваивание, инкремент, декремент, а для пользовательских типов запрещен вызов любого неконстантного метода. При этом мы, конечно, сохраняем за собой возможность изменять объект напрямую.

Lifetime of a temporary

Хотя все временные объекты обычно уничтожаются сразу, как только был вычислен результат содержащего их выражения, мы внезапно можем обнаружить, что следующий код

```
int main() {
    const int& ref = 5;
    std::cout << ref;
}
```

компилируется и даже работает приблизительно так, как этого можно ожидать.

Дело в том, что из этого правила существует ряд исключений. В частности, как можно видеть, продолжительность жизни временного объекта продлевается до

продолжительности жизни ссылки на него, если таковая объявлена. Приведенный выше код с точностью до оптимизаций эквивалентен следующему:

```
int main() {
    const int x = 5;
    const int& ref = x;
    std::cout << ref;
}
```

Несмотря на то, что в современном C++ продление жизни временных объектов — это задача rvalue-ссылок, константные lvalue-ссылки также позволяют её решить. Это сделано из соображений обратной совместимости: до C+11 rvalue-ссылки вообще не существовали, и механизм продления продолжительности жизни реализовывался именно так.

Битые ссылки (Dangling references)

Несмотря на то, что ссылка всегда указывает на существующий объект в момент инициализации, мы не застрахованы от ситуации, при которой она остаётся доступной уже после завершения жизни этого объекта:

```
int& foo() {
    int x = 238;
    int& wtf = x;
    return wtf;
}

int main() {
    int& ref = foo();
    std::cout << ref; // 238
}
```

В этом примере `ref` является копией ссылки `wtf` на целочисленную переменную `x`, объявленную в функции `foo()`. На момент инициализации `wtf` переменная `x` была записана в стек и, очевидно, существовала, поэтому все определения в теле `foo()` валидны.
Как только `foo()` прекращает свою работу, стек сворачивается, и память, выделенная под переменную `x`, деаллоцируется.[3] Несмотря на это, благодаря `ref` мы всё ещё обладаем к ней прямым доступом, хотя семантически это нарушает главную гарантию, которую должна предоставлять ссылка.

Это поведение называется *dangling* и, разумеется, является неопределенным. В ряде случаев компилятор гарантирует отсутствие битых ссылок, однако в общем случае мы вынуждены самостоятельно следить, чтобы они не появлялись.

Ссылки на сложные объекты

Так же, как мы можем объявить указатель на массив, мы можем объявить соответствующую ссылку:

```
int arr[238];
int (&ref)[238] = arr;
```

Такие ссылки (равно как и указатели) позволяют нам работать с массивом целиком, избегая array-to-pointer конверсий. Это может быть полезно преимущественно когда массив выступает аргументом или возвращаемым значением функции, однако в общем случае таких ситуаций рекомендуется по возможности избегать, поскольку C++ предлагает более удобные и безопасные способы реализации такой бизнес-логики.

Аналогичным образом объявляются и ссылки на функцию:

```
void print() {
    std::cout << "Hello, World!\n";
}

int main() {
    void (&ref)() = print;
}
```

Впрочем, у них нет никаких преимуществ перед указателями на функцию, поэтому на практике они почти не используются.

Источники

1. Стандарт ISO C++20 (https://isocpp.org/files/papers/N4860.pdf)
2. dm_frox — Ссылки и ссылочные типы в C++ (https://habr.com/ru/articles/646005/#id-5-1)
3. cppreference.com — Reference initialization (https://en.cppreference.com/w/cpp/language/reference_initialization.html)
4. cppreference.com — Reference declaration (https://en.cppreference.com/w/cpp/language/reference.html)
5. C++ FAQ: References (https://isocpp.org/wiki/faq/references)

1. Согласно стандарту ISO C++20 ($\S 6.1$ — $\mathrm{Preamble}$), переменные появляются только в результате объявления объектов или ссылок на нестатические данные. ↩
2. Инлайнинг — способ оптимизации, при котором вызов функции заменяется непосредственно её телом. ↩
3. Практически говоря, в ряде ситуаций положение может спасти инлайнинг, однако здесь мы условимся, что его не происходит. ↩