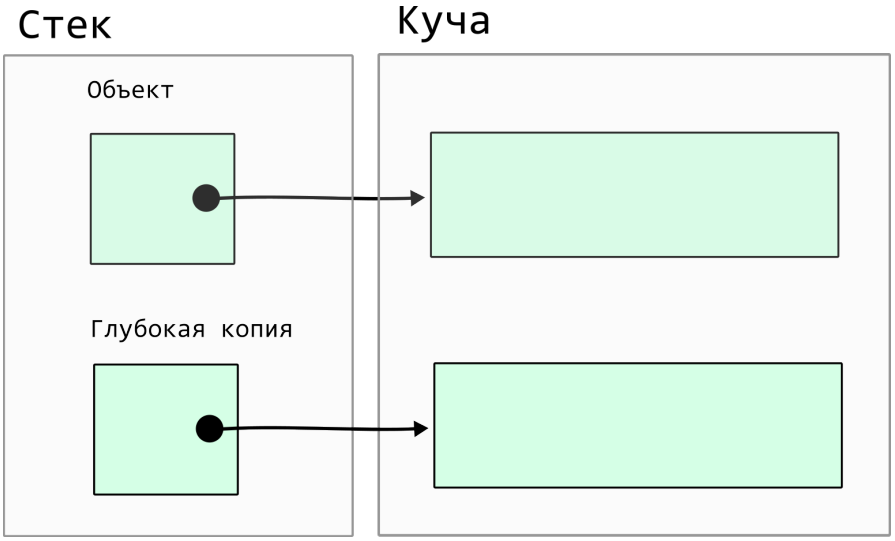
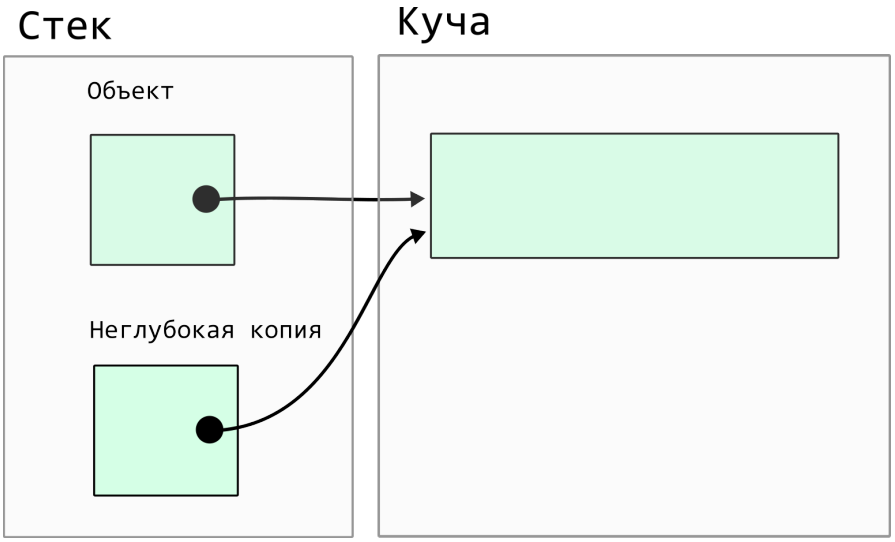
Семинар #16: Move-семантика.

Часть 1: Глубокое и поверхностное копирование

Во многих языках программирования существует понятия глубокого и поверхностного копирования (deep copy и shallow copy). Под глубоким копированием понимается рекурсивное копирование объекта и всех ресурсов, связанных с ним (например, памяти, выделенной в куче). Как правило, `operator=` в языке C++ перегружается таким образом, чтобы проводить глубокое копирование.



При поверхностном копировании происходит только побайтовое копирование полей объекта. В том числе копируются все указатели, которые продолжают указывать на ту же область памяти в куче. Так, например, работает присваивание структур в языке C.



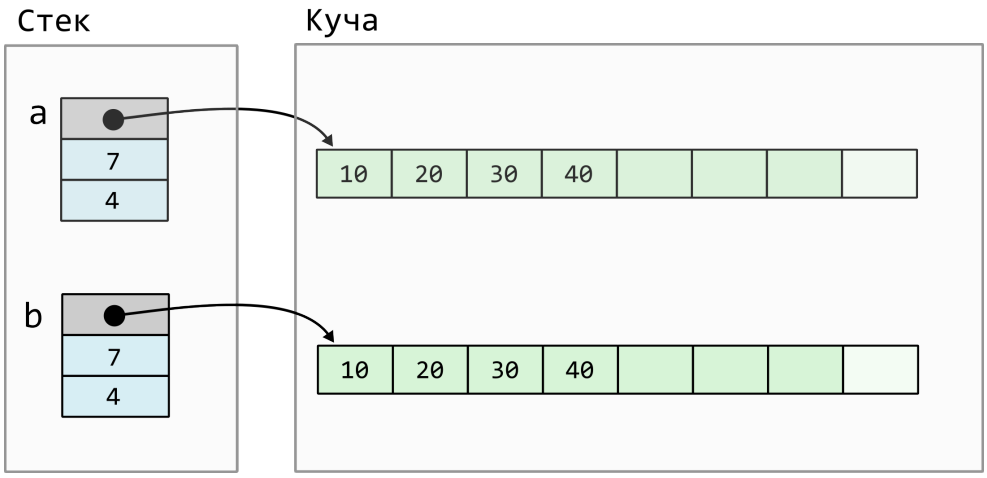
Поверхностное копирование имеет множество недостатков, связанных с безопастностью работы программы. Более того, иногда нам нужна полная копия объекта и поверхностное копирование просто не подойдёт. Но есть и большое преимущество такого копирования – оно значительно более эффективно.

Часть 2: Копирование и Перемещение в C++

Копирование

Под копированием в языке C++ понимается глубокое копирование. В примере ниже вектор `a` копируется в вектор `b` с помощью конструктора копирования.

```
std::vector<int> a {10, 20, 30, 40};
a.reserve(7);
std::vector<int> b = a;
```



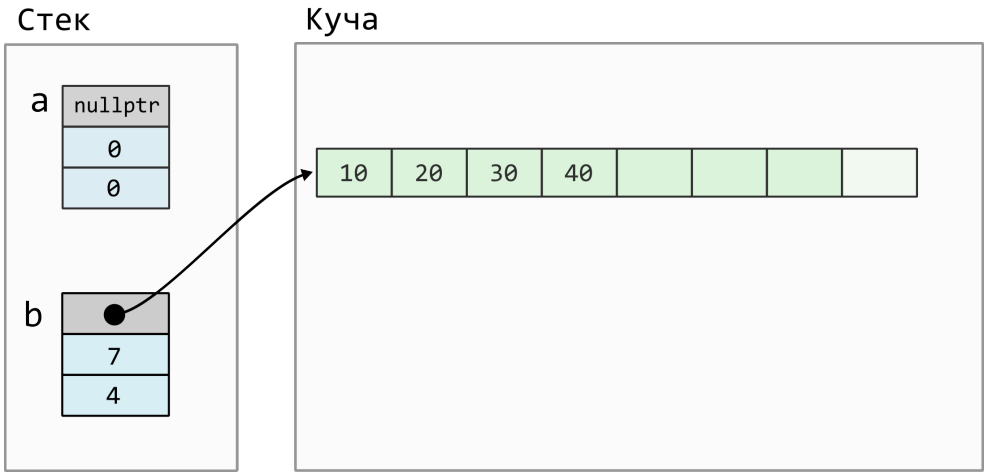
Перемещение

Под перемещением в C++ понимается операция, состоящая из двух частей:

1. Поверхностное копирование
2. Изменение объекта, из которого производилась копия. Объект должен перестать владеть ресурсом, но должен находится в корректном состоянии.

Перемещение проводится с помощью специальной стандартной функции `std::move`. В примере ниже проводится перемещение вектора `a` в вектор `b`. При этом вектор `a` после перемещения не содержит указатель на память в куче. Тем не менее вектор `a` продолжает находиться в корректном состоянии. В него, например, можно скопировать или переместить другой вектор.

```
std::vector<int> a {10, 20, 30, 40};
a.reserve(7);
std::vector<int> b = std::move(a);
```



Польза перемещения

Перемещение очень полезно тогда, когда объект из которого производится перемещение перестаёт быть нужным после перемещения. В этих случаях перемещение позволяет существенно ускорить программу.

Перемещение временных объектов

Рассмотрим следующий пример. Есть две строки `s1` и `s2`, которые потенциально могут быть очень длинными. Мы хотим конкатенировать эти строки и сохранить результат в другой строке `s`.

```
s = s1 + s2;
```

В этом случае будут произведены следующие операции:

1. Конкатенирование строки с помощью метода `operator+` строки. При этом создаётся некоторый временный объект (типа `std::string`), в котором будет хранится результат конкатенации.
2. Перемещение этого временного объекта в строку `s`.

Без перемещения, на втором шаге нам бы пришлось копировать временный объект в строку `s`, что было бы намного менее эффективно. Аналогично, перемещение ускоряет программу, когда мы передаём временные объект в функции по значению.

```
void func(std::string s) {...};
// ...
func(s1 + s2); // Тут используется
```

Ускорение некоторых операций с помощью перемещения

Перемещение может быть полезно не только для временных объектов. Перемещая обычные невременные объекты можно ускорить многие алгоритмы. Рассмотрим, например, задачу обмена значений двух строк, которые потенциально могут быть очень длинными:

```
std::swap(s1, s2);
```

Функция `swap` просто перемещает объекты и реализована следующим образом:

```
template<typename T>
void swap(T& a, T& b)
{
    T temp = std::move(a);
    a = std::move(b);
    b = std::move(temp);
}
```

Без перемещения объекты пришлось бы многократно копировать внутри функции `swap`, что было бы очень неэффективно для объектов, владеющих памятью в куче. С перемещением `swap` работает намного быстрее для объектов, выделяющих память в куче. Соответственно, будут работать намного быстрее все алгоритмы, использующие `swap` (например, алгоритмы сортировки).

Возвращаемое значение функции

Перемещение может помочь при возврате объекта из функции. Но в этом случае нет необходимости использовать `std::move`, так как перемещение происходит автоматически. Более того, использование `std::move` может не дать компилятору использовать RVO(Return Value Optimization), что может привести к более медленному коду.

Часть 3: `lvalue` **и** `rvalue`

Для объекта можно написать конструктор копирования и оператор присваивания, которые должны производить глубокое копирование объекта. По аналогии с копированием, для объекта можно создать конструктор перемещения и оператор присваивания перемещением.