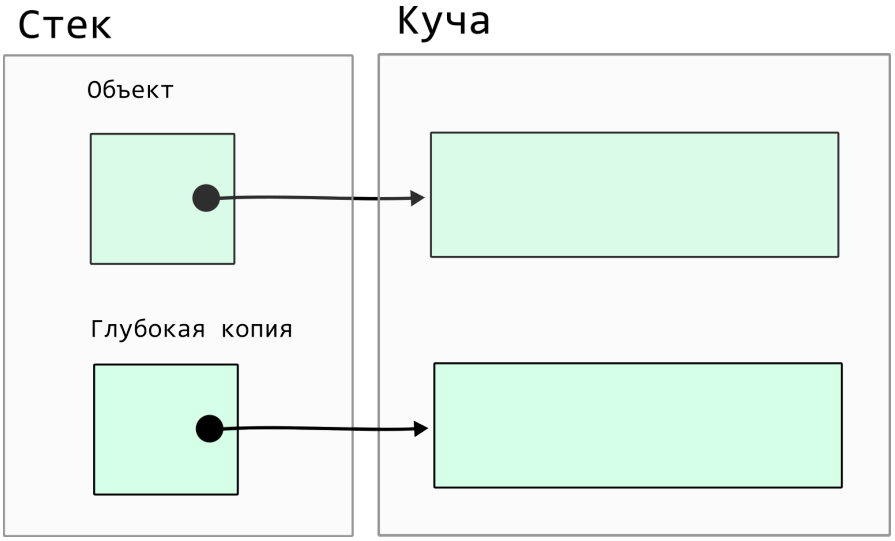
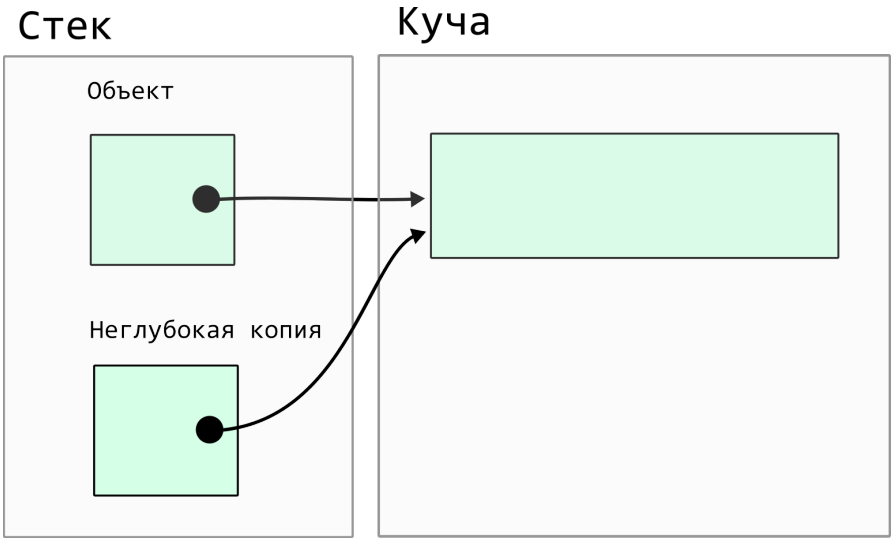
Семинар #16: Move-семантика.

Часть 1: Глубокое и поверхностное копирование

Во многих языках программирования существуют 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.



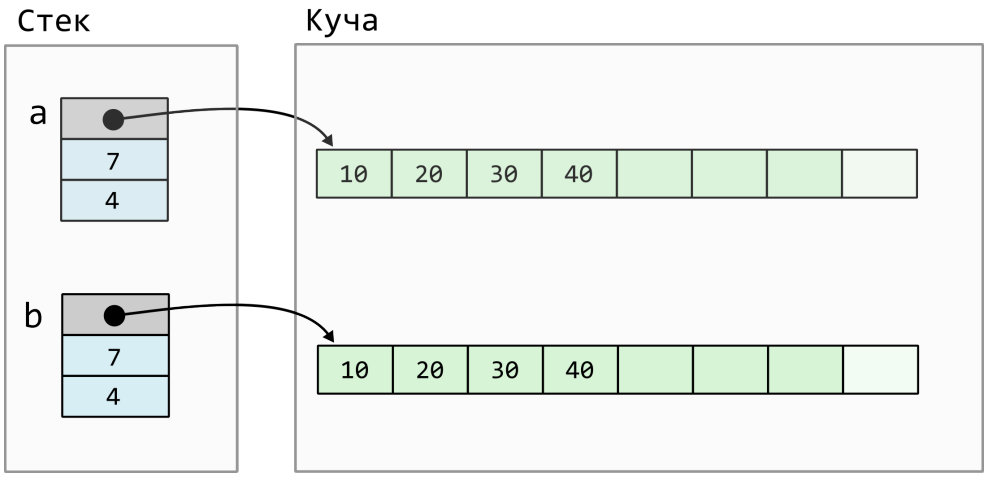
Поверхностное копирование имеет множество недостатков, связанных с безопастностью работы программы. Более того, иногда нам нужна полная копия объекта и поверхностное копирование просто не подойдёт. Но есть и большое преимущество такого копирования – оно значительно более эффективно.

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;
```



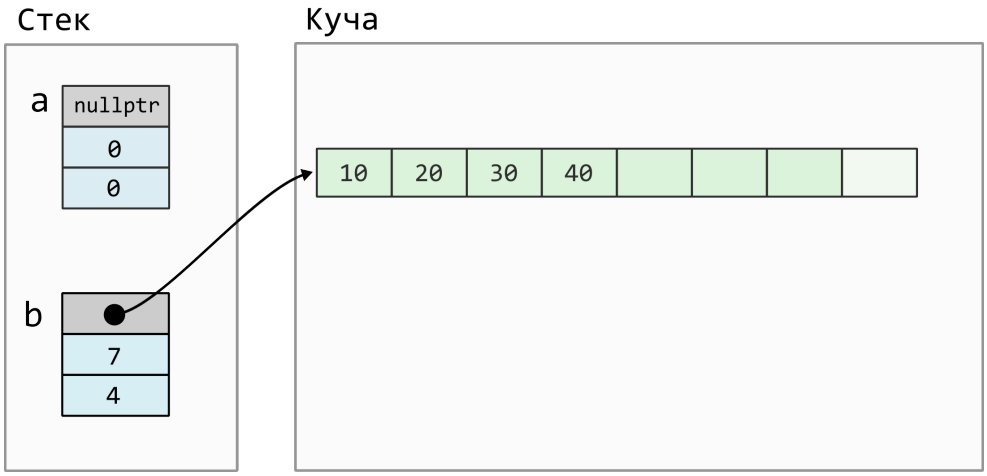
Перемещение

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 обычно 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

Пусть есть функция func, принимающая объект некоторого типа по значению. В качестве типа объекта в этом примере возьмём std::string, но вообще это может быть любой тип.

```
void func(std::string s) {...}
```

Предположим, что мы передаём на вход этой функции некоторое выражение. Что лучше использовать при передаче в этом случае: копирование или перемещение? На самом деле это зависит от выражения, которое приходит на вход функции.

```
std::string s1 = get1();
std::string s2 = get2();

func(s1);      // В этом случае лучше использовать копирование, так как s1 нам ещё нужна
func(s1 + s2); // В этом случае лучше использовать перемещение, так как s1 + s2 нам не нужен
```

В общем случае нам бы хотелось, чтобы те выражения, у которых есть имя или известный адрес, передавались копированием. Так как они могут быть использованы после вызова функции. Такие выражения называются lvalue-выражениями. Остальные выражения, то есть те, которые мы хотим перемещать, называются rvalue-выражениями.

В коде ниже представлены примеры lvalue и rvalue выражений.

```
#include <string>
#include <math>

int main()
{
    int a = 123;
    int array[5] = {1, 2, 3, 4, 5};
    std::string s = "Cat";

    // Примеры lvalue выражений:
    a   array    s   array[i]

    // Примеры rvalue выражений:
    a+1    -a    2*array[i]    s+"!"    sqrt(5)
}
```

На самом деле, приведённое выше определение lvalue не полностью верно и не учитывает некоторые случаи. Более точное определение понятий lvalue и rvalue приведено в стандарте, оно слишком громоздко, чтобы описать его здесь.

При передаче в функцию по значению компилятор автоматически определяет является выражение lvalue или rvalue и, либо копирует его, либо перемещает. Но иногда бывают ситуации, когда нам хочется переметить lvalue выражение. В этом случае нужно просто использовать стандартную функцию std::move. Эта функция сама по себе ничего не передвигает, а просто превращает lvalue в rvalue.

```
func(s1);             // Произойдёт копирование
func(s1 + s2);        // Произойдёт перемещение временного объекта s1 + s2
func(std::move(s1));  // Произойдёт перемещение s1
// Тут s1 стал пустой строкой, так как мы его переместили
```

Часть 4: rvalue-ссылки

В предыдущей часте мы рассмотрели передачу в функцию по значению. Но чаще используется передача по ссылке. Для того, чтобы можно было различать категорию выражения при передаче по ссылке в язык были введены rvalue-ссылки. Такие ссылки очень похожи на обычные ссылки (которые теперь называются lvalue-ссылками). Основное отличие таких ссылок в том, что они инициализируются только rvalue-выражениями.

```
int a = 123;
// rvalue ссылку можно инициализировать так:
int&& r1 = 10;
int&& r2 = a + 1;
// Следующее не будет работать так как a это lvalue:
int&& r3 = a;
```

С помощью rvalue ссылок и перегрузки функций можно различить категорию выражения, приходящую на вход функции. В примере ниже вызовется соответствующий вариант перегрузки в зависимости от категории выражения.

```
#include <iostream>
#include <string>
using std::cout, std::endl;

void func(std::string& s)
{
    cout << "Pass by lvalue reference" << endl;
}
void func(std::string&& s)
{
    cout << "Pass by rvalue reference" << endl;
}

int main()
{
    std::string s1 = "Cat";
    std::string s2 = "Dog";

    func(s1);               // Передадим по lvalue ссылке
    func(s1 + s2);          // Передадим по rvalue ссылке
    func(s1.substr(0, 2));  // Передадим по rvalue ссылке
}
```

Часть 5: Конструктор перемещения и оператор присваивания перемещения

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.