Лекция 5. Память программы на C++. Семантика перемещения

1. Память программы (процесса)
   - Стек
   - Куча
   - Сегмент BSS данных (Block Started by Symbol)
   - Сегмент данных (Data Segment)
   - Сегмент данных для чтения (RODATA Segment)
   - Сегмент кода программы (Text Segment)
   - Сегмент локальных переменных потока (Thread Local Segment)
2. Продолжительность хранения (*Storage duration*)
   - Static storage duration
   - Automatic storage duration
   - Dynamic storage duration
   - Thread storage duration
3. Связывание (*linkage*)
   - Внешнее связывание (*external linkage*)
   - Внутренее связывание (*internal linkage*)
   - Отсутствие связывания (*no linkage*)
   - Модульное связывания (*module linkage*) C++20
4. Ключевое слово `static`
   - Вне класса. Глобальная статическая переменная
   - Вне класса. Локальная статическая переменная
   - Вне класса. Глобальная статическая функция
   - Внутри класса. Статическое поле
   - Внутри класса. Статический метод
5. Initialization Order Fiasco
6. Ключевое слово `inline`
7. Работа с динамической памятью
   - Операторы `new` и `delete`
   - Операторы `new[]` и `delete[]`
   - Переопределение операторов `new` и `delete`
8. Ключевое слово `noexcept`
9. Категория выражений (value categories)
   - rvalue-ссылка
   - Связывание ссылок
   - Перегрузка функции по ссылке
   - Квалификация метода по виду ссылки
   - Temporary materialization (C++17)
10. Семантика перемещения

- Функция `std::move`
- Правило нуля (*Rule of Zero*)
- Правило пяти (*Rule of Five*)
- Правила для специальных методов

11. Copy Elision
    - Return Value Optimization (RVO)
    - Named Return Value Optimization (NRVO)

Память программы (процесса)

Программа имеет виртуальное адресное пространство. При запуске программа не полностью загружается в оперативную память. Операционная система (ОС) настраивает таблицы страниц и выделяет области памяти для программы. Таблицы страниц служат для отображения виртуальных адресов программы на C++ в физические адреса оперативной памяти.

В языке C++ упрощенно можно выделить следующие сегменты памяти:

- **автоматический** - представлен стеком, служит для автоматического выделения и освобождения памяти под локальные переменные, аргументы функции.
- **статический** - для глобальных и статических переменных, существует всю жизнь программы
- **динамический** - представлен кучей, позволяет получать память в использование во время выполнения программы и требует управления выделением и освобождением памяти
- **потоковый** - для специально помеченных локальных переменных потока, существует всю жизнь потока

Память процесса можно разделить на следующие сегменты:

- Stack (Стек)
- TLS (Thread local segment)
- Heap (Куча)
- BSS (Block Started by Symbol)
- Data Segment
- RODATA Segment (read only)
- Text Segment (Code)

Стек

Стек имеет ограниченный размер, который зависит от ОС и её настроек. По умолчанию размер составляет 1-8 Мб (Windows, Linux). Стек организован по принципу LIFO

Выделяемая память на стеке **НЕ** инициализируется по умолчанию.

При вызове функции на стеке формируется стек фрейм, в котором размещаются параметры функции, локальные переменные, адрес возврата, сохраненные регистры.

Компилятор **НЕ** гарантирует, что переменные на стеке будут лежать в порядке объявления.

При выходе из области видимости вызываются деструкторы объектов в порядке обратном созданию

При выходе из функции указатель текущего фрейма стека (stack pointer) просто сдвигается, тем самым память автоматически считается свободной.

Переполнение стека приводит к ошибке stack overflow -> segmentation fault.

К переполнению может приводить как размещение слишком больших массивов, так и глубокий рекурсивный вызов функий.

Куча

Динамическая память (куча) представляет собой участок памяти, требующий ручного управления. Динамическая память выделяется во время выполнения программы и имеет продолжительность хранения, пока не будет освобождена.

Куча медленнее стека поскольку при выделении может происходить запрос к менеджеру памяти ОС (системный вызов) во время выполнения программы.

Сегмент BSS данных (Block Started by Symbol)

Данный сегмент хранит глобальные и статические переменные, которые не имеют явной инициализации или инициализированы нулем.

Загрузчик ОС при запуске программы выделяет память под эти переменные и зануляет её, поэтому они не занимают места в бинарном файле, что позволяет экономить место.

Сегмент данных (Data Segment)

Данный сегмент хранит глобальные и статические переменные, которые явно инициализированы ненулевыми значениями. Переменные занимают место в бинарном файле

Сегмент данных для чтения (RODATA Segment)

Как следует из названия в данном сегменте лежат данные для чтения. Поэтому попытка изменения данных приводит к ошибкам компиляции или к UB, падению программы при

попытке изменить константу. Продолжительность хранения данного сегмента статическая - все время жизни программы

В данном сегменте лежат именованные константы, строковые литералы.
Строковые литералы могут лежать в единственном экземпляре.

Сегмент кода программы (Text Segment)

В данном сегменте располагается машинный код программы. В данном сегменте записаны функции, методы и точка входа в программу `main`. Как правило, данный сегмент предназначен только для чтения и является разделяемым (*shared*) между потоками.

Сегмент локальных переменных потока (Thread Local Segment)

В данном сегменте располагаются локальные переменные потока. Для таких переменных при объявлении используется ключевое слово `thread_local` перед типом. У каждого потока свои копии переменных, а их время жизни соответствует времени жизни потока.

Продолжительность хранения (*Storage duration*)

В языке C++ с сегментами памяти тесно связано понятие storage duration.

Продолжительность хранения (storage duration) - период, в течение которого зарезервирована память для объекта.

Продолжительность хранения это свойство объекта, которое определяет минимальный потенциальный срок жизни выделенной памяти под объект.

В языке C++ выделены следующие виды продолжительности хранения:

- static
- automatic
- dynamic
- thread

Static storage duration

Static storage duration - память выделяется и освобождается независимо от области видимости объекта и продолжает существовать все время выполнение процесса.

Соответствует сегментам памяти `.bss`, `.data`, `.rodata`

Глобальные, статические переменные, в том числе объявленные внутри `namespace` имеют static storage duration

Создается до входа в `main` (для локальных статических переменных при первом входе)

Порядок инициализации между единицами трансляции **НЕ** определен. Следовательно, некорректно инициализировать переменную со статическим storage duration, переменной с аналогичным storage duration из другой единицы трансляции (файла)

```
int global;
int global_init = 18;
static int static_value = 10;
extern int value;

namespace utils {
    const double IN_TO_CM = 2.54;
}

int& get_static() {
    static int instance;
    return &instance;
}
```

Automatic storage duration

Automatic storage duration - память выделяется при входе в область видимости и освобождается при выходе из области видимости, причем вызываются деструкторы.

Представлен стеком.

```
void function(int a, double b, const char* name) {
    std::string info(name);
    {
        double num = a * b;
        info += std::to_string(num);
    }
    std::cout << info << std::endl;
}
```

- переменные a, b, name, info, num - располагаются на стеке

Dynamic storage duration

Dynamic storage duration - управление памяти осуществляется вручную.

Представлен кучей.

```
int* ptr = new int(5);
int* arr = new int[5]{1, 2, 3};
delete ptr;
delete[] arr;
```

Thread storage duration

Thread storage duration - память выделяется при запуске потока и освобождается при его завершении.

У каждого потока своя копия

```
thread_local int value = 18;
```

Связывание (*linkage*)

Связывание определяет видимость идентификаторов между единицами трансляции. Связывание не относится к памяти напрямую, а затрагивает связывание имён на этапе линковки

Внешнее связывание (*external linkage*)

External linkage - внешнее связывание означает, что идентификатор (имя) виден в других единицах трансляции. По умолчанию для сущностей C++ в глобальной области видимости.

Внутренее связывание (*internal linkage*)

Internal linkage - внутреннее связывание, означает, что идентификатор (имя) виден только в текущей единице трансляции. Для `static` переменных, `const` переменных, имен шаблонных параметров.

Таким образом `const` в заголовочном файле, включаемый в разные `.cpp` файлы, представлен копией в каждой единице трансляции.

Для внешнего связывания `const` переменной используется ключевое слово `extern`

Для внешнего связывания константы в C++17 используется ключевое слово `inline constexpr`

Отсутствие связывания (*no linkage*)

No linkage - отсутствует связывание, означает, что идентификатор (имя) виден только в области видимости. Для параметров функций, имен шаблонных параметров, локальных переменных.

Модульное связывания (*module linkage*) C++20

В языке C++20 появились модули `module` и вместе с ними появился новый вид связывания

Module linkage (C++20) - модульное связывания означает, что идентификатор (имя) виден в пределах модуля и не виден извне.

Модуль может включать несколько единиц трансляции.

Ключевое слово `static`

Ключевое слово `static` применяется к идентификатору переменной или функции и имеет разные смыслы в зависимости от контекста использования.

Вне класса. Глобальная статическая переменная

Глобальная переменная, помеченная ключевым словом `static`, имеет статическую продолжительность хранения (static storage duration) и внутреннее связывание (internal linkage), то есть видна только в текущей единице трансляции. Что позволяет скрыть реализацию в .cpp файле и избежать конфликтов **ODR** (*One Definition Rule*)

```
static int value = 18;
```

НО если определить её в `.h` файле, то каждый включающий её `.cpp` файл будет иметь копию, видную только в пределах данного файла.

Вне класса. Локальная статическая переменная

Локальная статическая переменная (внутри функции) имеет *static storage duration* и область видимости внутри функции, а также инициализируется только **один раз** при первом вызове функции.

```
void f() {
    static int counter = 0;
    ++counter;
    std::cout << "f() call count = " << counter << std::endl;
}
```

- **НЕ** видна вне функции

Как правило используется в качестве кэша, счетчика, синглтона (singleton), при ленивой инициализации (lazy initialization)

Вне класса. Глобальная статическая функция

Глобальная функция, помеченная ключевым словом `static`, также имеет внутреннее связывание и не видна из других единиц трансляции (.cpp файлов)

```
static void inner_helper();
```

В современном C++ заменяют `static` в данном контексте на размещение глобальной функции или переменной в безымянном пространстве имен `namespace`

Внутри класса. Статическое поле

Переменная внутри класса, объявленная с ключевым словом `static`, имеет статическую продолжительность хранения (static storage duration) и внешнее связывание (external linkage), если нет inline.

Статическая переменная класса принадлежит классу, а не конкретному экземпляру.

Доступ к статическому полю принято осуществлять посредством имени класса и оператора `::`. Также доступно через экземпляр класса и оператор `.`

Статическое поле является объявлением. Требует определения только в одном `.cpp`, поэтому инициализация в теле класса вызывает ошибку **ODR violation**.

```
// .h
class C {
    static int counter; // declaration
    // static int counter = 0; // compile error
};
// .cpp
int C::counter = 0; // definition
```

Инициализация `static` поля внутри класса разрешена для интегральных констант.

НО для получения адреса константного статического поля необходимо определение **ВНЕ** класса, даже если значение проинициализировано.

```
// .h
class C {
    static const int multipliyer = 2; // declaration
    // static const double PI = 3.14; // error double is not integral
```

```
};
// .cpp
const int C::multipliyer; // definition
const int* ptr = &C::multipliyer; // work after definition
```

Инициализация `static` поля внутри класса разрешена для `constexpr` полей (C++11).

```
// .h
class C {
    static constexpr int multipliyer = 2; // declaration
    static constexpr double PI = 3.141592653589793; //ok
};
// .cpp
constexpr int C::multipliyer; // definition
const int* ptr = &C::multipliyer; // work after definition
```

- Но для получения адреса необходимо определение **ВНЕ** класса до C++17. Начиная с C++17 `constexpr` по умолчанию является `inline` и определение **ВНЕ** класса **НЕ** требуется

Инициализация `static` поля внутри класса разрешена для `inline static` полей (C++17). Является определением, определение **ВНЕ** класса **НЕ** требуется, разрешено в заголовке.

```
class C {
    inline static int multipliyer = 2;
    inline static std::string default_name = "name";
};
```

Внутри класса. Статический метод

Функция внутри класса, объявленная с ключевым словом `static` не привязана к какому-либо экземпляру (объекту) класса, а логически принадлежит всему классу. Следовательно, внутри функции нет указателя на объект `this`.

```
struct A {
    static int f();
};
```

Внутри функции можно обращаться только к `static` полям, к обычным нельзя

Вызвать функцию можно с указанием принадлежности классу, используя оператор `::`

```
int value = A::f();
```

Initialization Order Fiasco

Initialization Order Fiasco - Ошибка порядка статической инициализации, которая возникает, когда один объект со *static storage duration* использует для инициализации другой объект из другого файла с аналогичным *storage duration*, поскольку порядок инициализации между единицами трансляции не определен.

Следующий код вызывает **UB**:

```
// a.cpp
extern int y;
int x = y + 1;

// b.cpp
int y = 10;
```

- порядок инициализации внутри одной единцы трансляции сверху вниз

Ключевое слово `inline`

Ключевое слово `inline` означает, что можно иметь несколько определений, если они одинаковы.

Для переменных (C++17) это значит, что это одно логическое определение и можно размещать в заголовке.

Таким образом, использование `inline static` используется для вспомогательных функций, определяемых в заголовочном файле.

Работа с динамической памятью

При выполнении операции выделения памяти процесс обращается к ОС, ОС находит свободные физические кадры памяти и обновляет таблицу страниц, чтобы отобразить виртуальный адрес (возвращаемый в программе) на этот физический кадр. Если физическая память заканчивается, ОС перемещает старые страницы на диск (подкачка), помечая запись и загружает новые. Если запрошенная по адресу переменная не находится в таблице страниц, происходит *page fault* и ОС загружает нужную страницу обратно в память. Поэтому работа с динамической памятью медленнее, чем со стеком.

Выделенная процессу память будет считаться занятой, пока с помощью операции освобождения памяти она не будет явно освобождена и отдана ОС. Если динамическую память не освободить, то она будет освобождена только по факту завершения процесса

При работе с динамической памятью могут возникать следующие проблемы:

- Утечки памяти (memory leak) - когда выделенная память не освобождается и уже не используется. Такая память будет считаться ОС занятой до тех пор, пока не завершится

весь процесс (программа).

- Двойное удаление (double free) - когда производится вызов оператора освобождения памяти дважды для одного указателя, что приведет к ошибке и падению программы
- Висячий указатель (dangling pointer) - когда указатель, указывает на область памяти, которая уже была освобождена (удалена), но сам указатель не был обнулен, либо это другой указатель на ту же область памяти.

Операторы `new` и `delete`

Для выделения динамической памяти используется оператор `new`, а для освобождения оператор `delete`.

Синтаксис: `<type>* <ptr_name> = new <type>;`
Синтаксис с инициализацией:

- `<type>* <ptr_name> = new <type>(<value>);`
- `<type>* <ptr_name> = new <type>{<value>};`

```
int* ptr = new int;
int* ptr_init = new int(5);
delete ptr;
delete ptr_init;
```

Операторы `new[]` и `delete[]`

Также есть соответствующая версии операторов для работы массивами `new[]`, `delete[]`.
Обязательно необходимо освобождать память с использованием парного оператора.

Синтаксис: `<type>* <name> = new <type>[<size>];`
Синтаксис с инициализацией аналогичен созданию обычных массивов.

```
int* arr = new int[10];
delete[] arr;
```

Переопределение операторов `new` и `delete`

Глобальный оператор `new`:

```
void* operator new(std::size_t);
void operator delete(void*) noexcept;
```

Оператор `new` внутри класса:

```
struct Struct {
    static void* operator new(std::size_t);
    static void operator delete(void*) noexcept;
};
```

Для вызова глобальных операторов в таком случае можно воспользоваться оператором разрешения области видимости `::operator new` , `::operator delete`

```
static void* Struct::operator new(std::size_t bytes) {
    std::cerr << "allocate: " << bytes << " bytes\n";
    return ::operator new(bytes);
}

static void Struct::operator delete(void* ptr, std::size_t bytes) noexcept {
    std::cerr << "deallocate: " << bytes << " bytes\n";
    ::operator delete (ptr);
}
```

Оператор placement `new` переопределить нельзя:

```
void* operator new(std::size_t, void*) noexcept;
```

Ключевое слово `noexcept`

Ключевое слово `noexcept` является спецификатором функции (в том числе метода, оператора) и оператором, проверяющим на этапе компиляции может ли выражение выбрасывать исключение `noexcept(<expression>)` .

Спецификатор `noexcept` пишется после сигнатуры функции и перед `{` .
Спецификатор сообщает компилятору, что код внутри не выбрасывает исключений, и позволяет ему выполнить оптимизации (убрать код для раскрутки стека).

Если исключение всё же будет выброшено из помеченной функции, то вызовется `std::terminate` , который завершит процесс.

Хорошим тоном считается писать `noexcept` там, где действительно код не может выбросить исключение.

Правила использования:

- Деструкторы по умолчанию `noexcept`
- Функции освобождения памяти (`free` , `delete`) должны быть `noexcept`
- Функции обмена (`swap`) должны быть `noexcept`
- Хорошими кандидатами на использование `noexcept` являются `const` методы
- Move-операции должны быть `noexcept` , когда это возможно
- **НЕ** следует использовать `noexcept(false)` без веской причины

Категория выражений (value categories)

Помимо типа каждое выражение в C++ характеризуется категорией.

Классификация до C++11:

- lvalue - имеет адрес, может быть слева от присваивания `=`
- rvalue - временное значение, справа от присваивания `=`

Современная классификация (C++11) построена на двух свойствах:

- Идентичность (I)- параметр, по которому можно понять ссылаются ли два объекта на одну и ту же сущность (например, наличие адреса в памяти)
- Перемещаемость (M) - возможно ли переместить объект, отдать владение ресурсом

В основу классификации легли три первичных категории

- lvalue (locator/left value)
- xvalue (expiring value) - временный материализованный объект (с истекающим временем жизни)
- prvalue (pure rvalue)

```
        glvalue(I)     rvalue(M)
         /     \       /      \
lvalue(I~M)   xvalue(IM)     prvalue(~IM)
```

---

rvalue-ссылка

В C++11 вводится rvalue-ссылка `&&`.

Основные отличия rvalue-ссылка `&&` от lvalue-ссылки `&`:

- можно проинициализировать только rvalue выражением
- при возврате из функции `&&` возвращается rvalue выражение.
- продлевает жизнь временным объектам

Во всем остальном ведет себя также как ссылка.
Сам по себе идентификатор, объявленный rvalue-ссылкой, является lvalue.

```cpp
int l = 18;
int&& r = l; // compilation error (l is not rvalue)
int&& x = 18; // ok
int&& y = x; //compilation error (x is not rvalue)
int&& m = std::move(l); // ok
int&& sc = static_cast<int&&>(x); // ok
m = 0; // ok l = 0, m = 0
```

В языке присутствует базовое неявное приведение *lvalue-to-prvalue*

```
int x, y;
x = x + 1;        // lvalue = prvalue
y = x;            // lvalue = lvalue to prvalue
y = std::move(x); // lvalue = xvalue
```

Связывание ссылок

Rvalue-ссылка **НЕ** может быть связана с **lvalue**
Lvalue-ссылка **НЕ** может быть связана с **rvalue**

```
int x = 1;
int&& y = x * 3; //ok
int&& b = x;     //fail, not rvalue

int& c = x * 3;       // fail, not lvalue;
const int& d = x * 3; //ok, lifetime prolongation for const ref

int&& e = y; // fail, not rvalue
int& f = y;  // ok
```

Перегрузка функции по ссылке

Функция может быть перегружен по типу принимаемой ссылке:

```
int foo(int& v); // 1
int foo(int&& v); // 2
int foo(const int& v); // 3
int foo(const int&& v); // 4

int x = 1;

foo(x); // -> 1
foo(1); // -> 2

const int y = 1;

foo(y); // -> 3
foo(std::move(y)); // -> 4
```

Квалификация метода по виду ссылки

Метод может быть перегружен, по факту вызова для **lvalue** или **rvalue** ссылки.

Для этого после закрывающей скобки аргументов добавляется квалификатор **&** или **&&**. Таким образом, можно определить различное поведение при вызове функции от временного объекта `&&` и постоянного `&`

```
struct S {
    int foo() &;
    int foo() &&;
};
```

Оператор присваивания можно пометить одним амперсандом `&` и не создавать версию с `&&`, чтобы к временному объекту нельзя было присваивать, тогда будет недоступно присваивание к временному объекту, что достаточно логично

```
struct S {
    S() = default;
    S& operator=(const S& other) &;
};

S a;
S() = a; // compile error
```

Temporary materialization (C++17)

Материализация временного объекта (*temporary materialization*) представляет собой неявное преобразование из *prvalue* в *xvalue*.

Материализация временного объекта происходит:

- при связывании ссылки с чисто временным объектом (категории prvalue)
- при доступе к нестатическому полю класса, prvalue объекта
- при вызове нестатических методов от prvalue объекта (поскольку в метод неявно передается указатель `this`)
- при выполнении преобразования array-to-pointer или обращению по индексу `[]` к prvalue массиву
- при инициализации объекта `std::initializer_list<T>` посредством списка инициализации в фигурных скобках
- когда prvalue появляется в отбрасывающем результат выражении (игнорируем результат prvalue выражения, возвращающего тип какого-либо класса)

1. Связывание ссылки с чисто временным объектом

```
const int& r1 = 10; // prvalue 10 is materialized to get cref
int&& r2 = 5 + 3;   // prvalue from expr '5+3' is materialized to get &&

void foo(const std::string&);
foo("hello"); // std::string materializes from "hello" to get cref
```

2. Доступ к нестатическому полю класса, prvalue объекта

```
struct Point { int x = 0, int y = 0};
int val = Point{3, 4}.x; // Point{3,4} is materialized to access.x;
```

3. Вызов нестатического метода от rvalue объекта

```
struct Logger {
    void log() const { std::cout << "Logging\n"; }
};
Logger().log(); // Logger() is materialized to call log(), need this.
std::string("hello").length(); // the same case
```

4. Обращение к prvalue массиву `[]` или при приобразовании array-to-pointer

```
void bar(const int*);
bar((int[]){10, 20});

int* p1 = (int[3]){1, 2, 3};.
int val = ((int[]){10, 20, 30})[1];
```

5. Инициализация объекта `std::initializer_list<T>`

```
#include <initializer_list>
void func(std::initializer_list<int> lst) {}
func({1, 2, 3});
```

6. Когда prvalue в discarded-value expression

```
struct Widget {
    Widget() { std::cout << "Created\n"; }
    ~Widget() { std::cout << "Destroyed\n"; }
};
Widget(); // materialized to the end of expression ;
```

Материализации временного объекта **НЕ** возникает, когда используется prvalue того же типа (при `direct-initialization` или `copy-initialization`). Такие объекты создаются напрямую из инициализирующего prvalue. Это является гарантированным *copy elision*, начиная с C++17

Семантика перемещения

Чтобы избежать излишнего копирования, когда создается объект из объекта, который больше не нужен, в языке реализована семантика перемещения (move-семантика).

Семантика перемещения позволяет забрать у другого объекты ресурсы, которыми он владеет. Причем, объект, у которого забрали данные, должен остаться в консистентном состоянии, то есть после перемещения из него данных, его состоянии было корректным (часто дефолтным, нулевым) и им можно было продолжать пользоваться.

Начиная с C++11 у класса появляется перемещающий конструктор и перемещающий оператор присваивания.

```
Class(Class&& other);
Class& operator=(Class&& other);
```

Компилятор генерирует move-конструктор и move оператор присваивания, вызывая соответствующие move операции для полей класса.

Для примитивных типов перемещающие методы выполняют **копирование**.

Владение ресурсом подразумевает, например указатель на динамическую память внутри класса и выделение памяти в конструкторе и освобождение в деструкторе. Пусть экземпляр класса создается на стеке, а данные в динамической памяти. Копирование по умолчанию подразумевает копирование значений всех полей, в случае указателя скопируются не данные, а указатель (адрес на выделенную память). В такой ситуации возникает проблема двойного удаления по указателю, так как два объекта имеют одинаковые указатели на одну память и при выходе из области видимости объектов в деструкторе будет двойное освобождение памяти по указателю.

Для такого класса переопределяется конструктор копирования и оператор присваивания копированием, чтобы данные в динамической памяти действительно копировались:

```
class String {
public:
    String(size_t size) : size_(size), cap_(size), data_(new char[size + 1]) {
std::fill_n(data, size + 1, '\0');
    }
    String(const String& other) :
        size_(other.size_),
        cap_(other.size_),
        data_(new char[other.size_ + 1]) {
        std::strcpy(data_, other.data_);
    }
    String& operator=(const String& other) {
        if (this != &other) {
            delete[] data;
            size = other.size;
            cap_ = other.size;
            data_ = new char[size + 1];
            std::strcpy(data_, other.data_);
        }
        return *this;
    }
    ~String() { delete[] data_; }
private:
```

```
    size_t cap_;
    size_t size_;
    char* data_;
};
```

Когда объект больше не нужен, а его данные нужны, или как минимум нужна выделенная память, чтобы не выделять её снова, можно переместить ресурс (память, файловый дескриптор, сокет и т.д.) из одного объекта в другой, а не копировать их. Это обеспечивает значительный прирост производительности, особенно для объектов, владеющих "тяжелыми" ресурсами.

Но объект из которого забирают ресурс должен остаться в консистентном состоянии и его можно было использовать

```
    String(String&& other) noexcept :
        cap_(other.cap_),
        size_(other.size_),
        data_(other.data_) {
        // необходимо оставить other в валидном состоянии
        other.cap_ = 0;
        other.size_ = 0;
        other.data_ = nullptr;
    }
    String& operator=(String&& other) noexcept {
        if (this != &other) {
            // очищаем свои данные
            delete[] data_;
            // копируем
            cap_ = other.cap_;
            size_ = other.size_;
            // забираем данные
            data_ = other.data_;
            // необходимо оставить other в валидном состоянии
            other.cap_ = 0;
            other.size_ = 0;
            other.data_ = nullptr;
        }
        return *this;
    }
```

Вызвать данные конструкторы можно следующим образом:

```
String str(1000);
auto str_move = std::move(str); // call move ctor

String str_other(10);
String str_moved(100);
str_moved = std::move(str_other); // call move assignment
```

Функция `std::move`

Функция `std::move` необходима для того, чтобы при вызове функции от объекта перенаправить компилятор на использование версии с rvalue-ссылкой.

Функция действует на этапе компиляции и всё что она делает, это возвращает rvalue-ссылку на переданный ей объект

```
std::string str = "I will be moved";
auto str_copy = str;
auto str_moved = std::move(str); // call move ctor for string
std::cout << str_copy == str_moved) << std::endl; // 1 (true)
std::cout << str.empty() << std::endl; // 1 (true)
```

Правило нуля (*Rule of Zero*)

"Правило нуля" - принцип программирования на C++, который гласит, что класс **НЕ** должен определять ни один из пяти специальных методов, если их можно автоматически сгенерировать компилятором:

- деструктор
- конструктор копирования
- оператор присваивания копированием
- конструктор перемещения
- оператор присваивания перемещением

Если ваш класс **НЕ** управляет ресурсами (памятью), а его данные — это простые типы или классы, которые сами управляют ресурсами (например, `std::string`, `std::vector`, умные указатели), то вам **НЕ** нужно писать свои специальные методы.

Класс **НЕ** должен самостоятельно управлять ресурсами, если это не класс управления ресурсами.

Правило пяти (*Rule of Five*)

"Правило пяти" - принцип программирования на C++, который гласит, что если пользовательский класс управляет ресурсами (память, файловые дескрипторы) и определяет хотя бы один из пяти специальных методов, то он **должен явно определить все пять**:

- деструктор
- конструктор копирования
- оператор присваивания копированием
- конструктор перемещения
- оператор присваивания перемещением

Такой подход позволяет избежать ошибок управления памятью (висячие указатели, двойное удаление).

До возникновения семантики перемещения в C++11 использовали "Правило трёх".

Правила для специальных методов

Компилятор генерирует специальные методы самостоятельно в зависимости от условий.

Деструктор по умолчанию создается:

- default если нет ни одного пользовательского деструктора

Конструктор копирования (аналогично копирующее присваивание):

- delete если его нет, но есть пользовательское перемещение или move ctor
- иначе default, если его нет (но если при этом есть деструктор или парный копирующий метод, то deprecated)

Перемещающий конструктор (аналогично перемещающее присваивание)

- delete если его нет, но при этом есть парный перемещающий метод или copy ctor, или copy operator=, или dtor (при наследовании если деструктор объявлен virtual)
- иначе default, если его нет

Copy Elision

Copy elision (пропуск копирования) в C++ — это оптимизация компилятора, устраняющая создание временных объектов при возврате из функций или передаче по значению, что исключает вызов конструкторов копирования/перемещения.

Начиная с C++17 компилятор обязан применять *copy elision* при инициализации объекта результатом функции, возвращающей тот же тип, что повышает производительность и делает код более эффективным.

Оптимизация позволяет не тратить время на копирование/перемещения, а деструкторы для временных объектов не вызываются.

Return Value Optimization (RVO)

Return Value Optimization (RVO) - оптимизация компилятора, позволяющая не создавать локальный объект, который используется в качестве возвращаемого значения, а конструировать возвращаемый объект на месте вызова функции. Такая оптимизация позволяет устранить лишний вызов конструктора копирования или перемещения.

RVO применима, если возвращаемое значение prvalue-выражение, то есть временный объект.

Начиная с *C++17* RVO - это не оптимизация, а правило, которое должно выполняться компилятором.

Named Return Value Optimization (NRVO)

Named Return Value Optimization (NRVO) - оптимизация компилятора, аналогичная RVO, но действующая для локального lvalue-объекта, имеющего имя, и возвращаемого из функции.

На месте вызова функции вставляется инициализация объекта, принимающего возвращаемое значение функции. В аргументы функции добавляется указатель на данный объект и все вычисления возвращаемого значения выполняются над объектом под указателем.

NRVO может быть применена только когда тип возвращаемого объекта и тип возвращаемого значения сигнатуры функции точно совпадают

По этой причине **НЕ** следует возвращаемое значение оборачивать в `std::move()`, так как тип возвращаемого значения не будет совпадать и, следовательно, оптимизация не будет применена.