

```
<intT> fact(intT
S;
= i) {if (num
= intT(2); }
_back(num); retu
int main() {bool
n; while (todes
"; cin >> n; v.c
in()); i != v.end
t << ", "; cout
urn 0; }
) { if (N < 0) r
long double res
{ result *= i; }
nt N; cout << "E
< N << " = "<< f
"); return 0; }
VO
cout << num[i] <
m = new int[1000
for(int i =
num[i] = rand();
-first+1)];
mid) i++; cout >
while(a[j]>mid)
j]); i++; j--;}
; j--; i
```

# سی++

# سی بعلاوه

std::cout

<< قانون سه/پنج/صفر

<< مدیریت منابع

<< Move Semantics

<< std::endl;

# قانون سه/پنج/صفر

## مقدمه

تا الان با انواع constructor-ها در C++ آشنا شده‌اید. در این مطلب می‌خواهیم یک مرور کلی از آنها داشته باشیم، انواع value-ها را معرفی کرده و نگاهی به move semantics داشته باشیم.

برای یک کلاس به جز کانستراکتور اصلی، می‌توان متدهای زیر را تعریف کرد:

1. destructor
  2. copy constructor
  3. copy assignment
- و از C++11 متدهای زیر را نیز می‌توان تعریف کرد:
4. move constructor
  5. move assignment

به جز کانستراکتورهای اصلی، استفاده از بقیه موارد فقط در صورتی نیاز است که کلاس یک منبع (resource) را مدیریت می‌کند. در صورت تعریف یکی از 5 تا به دلیل مدیریت منابع در کلاس، باید بقیه آنها نیز تعریف شوند. این را با نام قانون سه یا پنج در C++11 یاد می‌کنند.

در صورت مدیریت نکردن منابع، نیازی به آنها نیست و به آن قانون صفر می‌گویند.

نمای کلی یک کلاس که هر پنج متد خاص را دارد به این شکل است:

```
class Test {
public:
    // default constructor
    Test();
    // constructor
    Test(int a);

    // destructor
    ~Test();
    // copy constructor
    Test(const Test& other);
    // copy assignment
    Test& operator=(const Test& rhs);
    // move constructor
    Test(Test&& other) noexcept;
    // move assignment
    Test& operator=(Test&& rhs) noexcept;
};
```

## تولید خودکار

در صورتی که کلاسی یکی از 5 متد خاص را تعریف نکند، کامپایلر سعی می‌کند آنها را به صورت implicit تولید کند. مثلا اگر یک کلاس هیچ کانستراکتوری نداشته باشد، کامپایلر یک دیفالت کانستراکتور می‌سازد که در تعریف آن، دیفالت کانستراکتور همه فیلدهای کلاس صدا زده می‌شوند.

نکته: همانطور که می‌دانید، وقتی کلاسی قرار است برای چندریختی به کار برود و از آن ارث برده شود، باید دیستراکتور آن virtual باشد. چون در غیر این صورت با delete کردن پوینتر به کلاس پدر، دیستراکتور فرزند اجرا نمی‌شود. اگر کلاس پدر نیازمند دیستراکتور خاصی نمی‌باشد، می‌توان آن را default کرد:

```
virtual ~Class() = default;
```

نکته‌ای در اینجا هست که باید به آن توجه کرد؛ کلیدواژه delete را در بقیه توابع از جمله کانستراکتورهای اصلی هم می‌توان استفاده کرد تا جلوی implicit conversion-ها گرفته شود. به طور مثال، اگر کلاس Point-ای داریم که کانستراکتور آن int a را می‌گیرد، می‌توان Point(10.2) هم صدا زد و double به int تبدیل می‌شود. ولی اگر در کلاس،

```
Point(double a) = delete;
```

را بنویسیم، صدا زدن کانستراکتور این کلاس با مقدار ورودی اعشاری غیرمجاز خواهد شد.

این در مورد copy assignment, copy constructor, destructor نیز برقرار است و در صورتی که هر یک از این سه مورد در کد به صورت صریح تعریف نشود، آن مورد به صورت implicit توسط کامپایلر تولید می‌شود.

با این حال در صورتی که حداقل یکی از سه مورد مذکور تعریف شوند، کامپایلر، move constructor و move assignment را نمی‌سازد؛ این رفتار کامپایلر هنگامی جالب‌تر می‌شود که اگر هیچ یک از سه مورد بالا تعریف نشوند، move constructor و move assignment تولید خواهند شد. دلیل این موضوع سازگاری با نسخه‌های قبلی C++ است که در آن‌ها مفاهیم move وجود نداشت.

## کلیدواژه‌ها

برای پنج تابع خاص و دیفالت کانستراکتور، می‌توان از کلیدواژه‌های default و delete استفاده کرد:

```
class A {
public:
    A() = default;
    A(const A& other) = delete;
};
```

کلید واژه default به کامپایلر اطلاع می‌دهد که همان تعریف implicit-اش را برای آن تابع در نظر بگیرد. با این کار خودمان مستقیم ذکر می‌کنیم و درگیر پیچیدگی قوانین تولید خودکار این متدها نمی‌شویم.

کلیدواژه delete، تابع مورد نظر را حذف کرده و از اجرای آن جلوگیری می‌کند. برای مثال، در اینجا چون کپی کانستراکتور حذف شده است، کامپایلر اجازه کپی شدن نمونه‌ای از کلاس را نمی‌دهد.

# مدیریت منابع

## copy constructor

حال به مراحل ساخت کپی کانستراکتور می پردازیم.

```
Array(const Array& other)
: Array(other.size_) {
std::copy(other.data_,
          other.data_ + other.size_,
          data_);
}
```

در اینجا آبیجکتی از کلاس Array را به عنوان آرگومان داریم که باید از آن کپی بگیریم. این وقتی صدا می شود که کد زیر را می نویسیم:

```
Array test(10);
Array testCopy(test);
Array testCopy = test;
```

توجه کنید که علامت = چون در خط initialization است، در واقع کپی کانستراکتور را صدا می زند.

در ابتدای کپی کانستراکتور (توجه کنید که اکنون داخل testCopy هستیم) کانستراکتور اصلی را با سایز test صدا می زنیم (delegating constructor)؛ این کار تخصیص حافظه آرایه را انجام می دهد. حال المان های آن را با استفاده از std::copy کپی می کنیم.

## مقدمه

در این بخش می خواهیم کلاسی برای یک آرایه هیپ با سایز ثابت پیاده سازی کنیم. برای این کار از یک کلاس با صرفا کانستراکتور اصلی شروع می کنیم و در هر مرحله متد های جدیدی به آن اضافه می کنیم و دلیل آن ها را بررسی می کنیم.

```
class Array {
public:
    Array(int size);
    int size() const;
    // ...
private:
    int* data_ = nullptr;
    int size_;
};

Array::Array(int size)
: data_(new int[size]()),
  size_(size) {}

int Array::size() const { return size_; }
```

## destructor

از آنجایی که در این کلاس تخصیص حافظه کرده ایم، نیاز به destructor ای داریم که آن را آزاد کند:

```
~Array() { delete[] data_; }
```

## copy assignment

کار ما اینجا تمام نشده و همانطور که قبلا گفتیم به copy assignment operator هم نیاز داریم. یک پیاده‌سازی ساده این اپراتور می‌تواند به صورت زیر باشد:

```
Array& operator=(const Array& rhs) {
    if (this != &rhs) { // (1)
        // prepare the new data
        int* newData = new int[rhs.size_];

        // replace the old data
        // (non-throwing)
        delete[] data_; // (3)
        data_ = newData; // (3)
        size_ = rhs.size_; // (3)
        std::copy(rhs.data_,
                  rhs.data_ + rhs.size_,
                  data_); // (3)
    }
    return *this; // (4)
}
```

در اینجا قبل از اینکه `data_` را پاک کنیم، آن را در متغیری لوکال ذخیره می‌کنیم تا مطمئن شویم exception-ای رخ نمی‌دهد و در نهایت داده کلاس را تغییر می‌دهیم. با این کار اگر اکسپشن رخ دهد، داده کلاس بدون تغییر باقی می‌ماند. به این موضوع exception safety می‌گویند.

3. اگر دقت کنیم می‌بینیم که این قسمت از کد را در copy constructor هم تکرار کرده‌ایم. با اینکه اینجا فقط چند خط است، ولی برای منابع پیچیده‌تر می‌تواند زیاد باشد. بنابراین بهتر است که راه حلی برای این مشکل پیدا کنیم.

4. در نهایت copy assignment، خود کلاس را ریترن می‌کند. دلیل این موضوع این است که بتوانیم پس از اساینمنت آن را در زنجیره‌ای از کارها قرار دهیم و از مقدار کلاس استفاده کنیم. مثلا `a = b = c` یا اگر کلاس قابلیت conversion به `bool` را دارد، آن را در یک دستور `if` استفاده کنیم.

```
Array& operator=(const Array& rhs) {
    if (this != &rhs) { // (1)
        // delete existing array
        delete[] data_; // (2)
        data_ = nullptr; // (2)

        // copy rhs' data
        size_ = rhs.size_; // (3)
        data_ = new int[size_]; // (3)
        std::copy(rhs.data_,
                  rhs.data_ + rhs.size_,
                  data_); // (3)
    }
    return *this; // (4)
}
```

1. در ابتدا self-assignment check را انجام می‌دهیم. این یعنی چک می‌کنیم آبجکت به خودش اساین می‌شود یا نه؛ اگر بله نباید اتفاقی رخ دهد. self-assignment به ندرت رخ می‌دهد؛ بنابراین در بیشتر مواقع این چک کردن بی‌هوده است.
2. اگر در `new` کردن جلوتر اکسپشن رخ دهد، `data_` فعلی‌مان را از دست داده‌ایم و `size_` هم مقدار اشتباهی دارد. از آنجایی که ممکن است در ادامه کار دیستراکتور کلاس صدا شود، `data` که خودمان `delete` کردیم دوباره در دیستراکتور `delete` می‌شود. برای جلوگیری از این اتفاق، آن را برابر `nullptr` قرار می‌دهیم چون `delete nullptr` معادل `no operation` است.

## Copy-and-Swap Idiom

با این روش می‌توان تمام مشکلاتی که بالاتر در copy assignment operator مطرح شد را حل کرد. برای این منظور باید یک تابع swap به کلاسمان اضافه کنیم. در این متد allocation یا copy انجام نمی‌شود و فقط پوینتر و سایز دو آبجکت به صورت shallow تعویض می‌شوند.

```
friend void swap(Array& first,
                 Array& second)
    noexcept {
    using std::swap;
    swap(first.data_, second.data_);
    swap(first.size_, second.size_);
}
```

این تابع خارج از کلاس تعریف می‌شود و ورودی آن دو رفرنس به کلاس‌مان است. برای دسترسی به فیلدهای پرایوت، تابع را داخل کلاس friend می‌کنیم (در صورت نوشتن تعریف تابع friend داخل کلاس مانند مثال بالا، همچنان تابعی خارج از کلاس محسوب می‌شود).

جلوی این تابع noexcept زده شده که یعنی این تابع، استثنائی را throw نمی‌کند. داخل تابع در ابتدا using std::swap زده شده که دلیل آن به ADL (Argument Dependent Lookup) برمی‌گردد. در آخر هم تمام فیلدها را swap می‌کنیم.

پس از نوشتن swap حالا می‌توان با داشتن یک copy constructor که بالاتر پیاده‌سازی شده بود، بقیه متدهای خاص (copy assignment و در جلوتر move constructor و move assignment) را به راحتی در چند خط پیاده‌سازی کرد:

```
Array& operator=(const Array& rhs) {
    Array temp(rhs);
    swap(*this, temp);
    return *this;
}
```

در این copy assignment، ابتدا با استفاده از copy constructor یک کپی از rhs می‌گیریم و سپس آن را با کلاس خود swap می‌کنیم. با این کار duplication نداریم و تمام منطق کپی کردن داخل copy constructor است. توجه کنید که اکنون به self-assignment check هم نیاز نداریم و در حالت بسیار خاص آن، کد به درستی کار می‌کند. در این کد exception safety نیز برقرار است و تا ساخته نشدن کامل کپی، فیلد کلاسمان تغییر نمی‌کند و swap هم چیزی throw نمی‌کند.



# انواع Value

## مقدمه

به طور کلی، دو نوع value داریم که به آنها lvalue و rvalue می‌گویند (در اصل دسته‌بندی جزئی‌تری هست که به آن نمی‌پردازیم).

lvalue مخفف left value است چون که می‌تواند در سمت چپ یک عبارت = قرار بگیرد و rvalue مخفف right value است چون که می‌تواند سمت راست = قرار بگیرد.

rvalue-ها موقت (temporary) هستند؛ از بین می‌روند و نام ندارند. مثلا در عبارت:

```
int a = 2 + 3;
```

مقدار  $2 + 3$  که 5 است یک rvalue است و در `a` که lvalue است ذخیره می‌شود و از بین می‌رود. خود  $3 + 2$  در جایی ذخیره نشده، نام ندارد و موقت است. مقدار بازگشتی تابع هم به همین صورت است و تا جایی که ذخیره نشود rvalue می‌ماند:

```
int b = 2 * func();
```

## رفرنس‌ها

می‌توان به lvalue و rvalue رفرنس زد که برای lvalue با استفاده از کاراکتر `&` و برای rvalue با استفاده از `&&` است:

```
int a = 10;
int& b = a;           // lvalue reference
int&& c = func();     // rvalue reference
```

یک lvalue reference یک بار در ابتدا initialize می‌شود تا بداند به چه متغیری اشاره می‌کند و پس از آن، قابلیت مقدارهی ندارد و متغیری که به آن اشاره می‌کند عوض نمی‌شود.

یک rvalue reference صرفا طول عمر مقدار rvalue را بیشتر می‌کند. همانطور که گفتیم rvalue مقداریست که نام ندارد. پس در مثال قبیل `c` یک lvalue است که تایپ آن رفرنس به rvalue است.

این دو می‌توانند ورودی تابع هم باشند و در صورت overload کردن تابع به صورت زیر:

```
void func(int& a);
void func(int&& a);
```

صدا زدن تابع با rvalue به دومی می‌رود.

اگر فقط تابع `int&` را داشته باشیم، نمی‌توانیم `func(10)` را صدا بزنیم چون که rvalue به lvalue reference نمی‌تواند bind شود.

ولی طبق قانون، rvalue می‌تواند به const lvalue reference بایند شود. برای همین وقتی تابعی `const string&` می‌گیرد می‌تواند با `"test"` صدا شود (که اینجا `"test"` کانستراکتور `const char*` برای استرینگ را صدا می‌زند، یک rvalue از تایپ `string` ساخته شده و آن را به const lvalue reference بایند می‌کند).

# Move Semantics

```
Class(Class&& other);
```

در `move constructor`، با `other`، مانند هر `lvalue` دیگری برخورد می‌کنیم و می‌دانیم که تغییر مقدار آن مهم نیست چون که رفرنس به یک مقدار موقت است.

مثلا می‌دانیم که در `std::string` از یک پوینتر به کاراکتر، برای دسترسی به مقدار رشته ذخیره شده در هیپ استفاده می‌شود. داخل `move constructor`، به جای کپی گرفتن از حافظه `other`، کل حافظه `other` را مال کلاس خود می‌کنیم. به عبارتی، `ownership` را انتقال می‌دهیم (یعنی همانطور که جلوتر خواهیم دید، `char*` را مستقیم اساین می‌کنیم و مال `other` را `nullptr` می‌کنیم که در دیستراکتور آن مشکلی پیش نیاید).

## Move

گاهی به یک `lvalue` دیگر نیازی نداریم و می‌خواهیم آن را `move` کنیم:

```
std::string str = "test";
std::string test = std::move(str);
```

در اینجا با استفاده از تابع کمکی `std::move`، می‌توانیم مالکیت `str` را انتقال دهیم. پس از آن، نباید از `str` استفاده کرد؛ در غیر این صورت موجب `undefined behaviour` می‌شود.

## مقدمه

از نسخه C++11، تعدادی متد جدید برای `move` به زبان اضافه شد. با استفاده از مکانیزم‌های ارائه شده در زبان از جمله `rvalue reference` ها و `std::move`، می‌توان از تخصیص حافظه و کپی‌گیری‌های اضافی جلوگیری کرد و کد را `exception safe` تر کرد.

به طور مثال می‌خواهیم یک کپی از استرینگ بسازیم:

```
std::string str = "test";
std::string test(str);
```

دومین خط، کپی کانستراکتور استرینگ را صدا می‌زند. این عملکرد مطلوب ما است چون که در آنجا یک کپی از `str` گرفته می‌شود و `str` که یک `lvalue` است دست‌نخورده باقی می‌ماند.

به مثال‌های زیر توجه کنید:

```
std::string test(func());
std::string test(s1 + s2);
std::string test(s.substr(...));
```

اگر کپی کانستراکتور صدا شود، از `rvalue` ای که داشتیم یک کپی گرفته می‌شود. این در حالیست که می‌شود مستقیم از `rvalue` که مهم نیست دست‌خورده شود و تغییر کند استفاده کنیم و از کپی اضافی (که در `string` شامل یک `allocation` اضافی است) جلوگیری کنیم.

برای همین برای استرینگ `move constructor` تعریف شده که ورودی آن یک `rvalue reference` است:



یک روش تقریباً ایده‌آل برای ذخیره استرینگ در کلاس، در شکل بالا نشان داده شده است.

این کار به انتقال `const std::string&` و سپس کپی کردن آن ترجیح داده می‌شود. توجه که اگر از `std::move` استفاده نمی‌کردیم، با انتقال `str` به کانستراکتور این کلاس، دو بار از استرینگ کپی گرفته می‌شد (یک بار برای انتقال `by value` و یک بار برای کپی کانستراکتور فیلد کلاس).

اینجا می‌توان `Person(std::move(str))` یا `Person(func(x))` یا هر `rvalue` ای هم پاس داد و در این حالت هیچ تخصیص حافظه‌ای نخواهیم داشت.

### پیاده‌سازی Move

در ادامه بحثی که در مدیریت منابع داشتیم، اکنون می‌خواهیم متدهای مربوط به بخش `move` را به کلاس آرایه هیپ اضافه کنیم.

### move constructor

اگر از `copy-and-swap idiom` برای پیاده‌سازی `rule of 3` استفاده کرده‌ایم، اضافه کردن `move constructor` و `move assignment` برای `rule of 5` کار راحتی خواهد بود. این دو را به مثال `Array` اضافه می‌کنیم:

```
Array(Array && other) noexcept {
    swap(*this, other);
}
```

این کانستراکتور یک `rvalue` می‌گیرد و `noexcept` است. داخل آن مانند `copy assignment`، کلاس را با دیگری `swap` می‌کنیم با این تفاوت که در آنجا باید یک کپی می‌ساختیم چون که ورودی `lvalue` بود، ولی اینجا ورودی `rvalue` بوده و می‌توان مستقیماً با همان `swap` کرد.

در صورتی که تابعی پارامتری را `by value` می‌گیرد، چیزی که به آن پاس می‌دهیم برای کانستراکت کردن آن استفاده می‌شود. پس اگر به آن `lvalue` پاس دهیم، کپی کانستراکتور و اگر `rvalue` بدهیم `move constructor` صدا زده می‌شود. یعنی کانستراکتورها هم مثل متدهایی هستند که `overload` شده‌اند.

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

func(str);
func(std::string("test"));
func(std::move(str));
```

در فراخوانی اول، کپی کانستراکتور صدا می‌شود و از `str` که `lvalue` از تایپ استرینگ است کپی گرفته می‌شود.

در فراخوانی دوم، ابتدا کانستراکتور استرینگ اجرا شده و یک آبجکت موقت ساخته می‌شود که `rvalue` است. پس در پاس دادن به تابع، کانستراکتور `move` برای `a` اجرا شده و با استفاده از آبجکت موقت، یک نمونه داخل تابع تولید می‌شود.

در فراخوانی سوم، `str` با استفاده از `std::move` به عنوان `rvalue` در نظر گرفته می‌شود و کانستراکتور `move` برای `a` اجرا می‌شود. پس از این صدا زدن نباید از `str` استفاده کرد.

پس همانطور که می‌بینیم کل کار `std::move` کست کردن ورودی به `rvalue reference` و بازگرداندن آن است.

```
class Person {
public:
    Person(std::string name)
        : name_(std::move(name)) {}
private:
    std::string name_;
};
```

## move assignment

assignment هم به طور مشابه کار کرده و فقط در آخر کلاس را هم باز می‌گرداند. پس از swap شدن و تمام شدن طول عمر متغیر rvalue، دیستراکتور آن صدا زده شده و cleanup انجام می‌شود (که مقادیر کلاس قبل از swap از بین می‌رود).

```
Array& operator=(Array&& rhs)
    noexcept {
    swap(*this, rhs);
    return *this;
}
```

assignment هم به طور مشابه کار کرده و فقط در آخر کلاس را هم باز می‌گرداند. پس از swap شدن و تمام شدن طول عمر متغیر rvalue، دیستراکتور آن صدا زده شده و cleanup انجام می‌شود (که مقادیر کلاس قبل از swap از بین می‌رود).

توجه کنید که در کلاس مقدار دیفالت پوینترها را nullptr می‌گذاریم تا مثلاً اگر کلاسمان default constructor داشت و بعداً به آن move assign شد، مقدار rvalue destructor آن صدا زده می‌شود delete nullptr را فراخوانی کند که مشکلی ندارد.

با ترکیب move با template کانسپت‌های پیشرفته دیگری مانند universal / forwarding references و std::forward هم داریم که از حوصله این مطلب خارج است.

می‌توانید یک مثال کامل که rule of 5 را رعایت می‌کند را در [این لینک](#) مشاهده کنید که پیاده‌سازی یک آرایه دو بعدی خطی است.

در طول سالیان متمادی، همواره تلاش و وظیفه دستیاران آموزشی، کمک به ارائه مفیدتر درس و انتقال بهتر مطالب به دیگران بوده است؛ دستیاران آموزشی درس برنامه‌سازی پیشرفته نیز از این قاعده مستثنی نبوده و در طول ترم‌های گذشته همواره سعی کرده‌اند در قالب‌های متفاوت، به انتقال مفاهیم این درس کمک کرده باشند.

سی‌بعلاوه **ی++ک** مجله‌ای در راستای همین هدف است که اولین موضوع از آن در ترم بهار 1402 ارائه شد و همچنان ادامه دارد. موضوعات این مجله فراتر از مقاصد پایه درس بوده و صرفاً برای اطلاعات بیشتر و درک بهتر مفاهیم ارائه می‌شوند. خواندن و یادگیری آن اجباری نیست ولی برای یادگیری عمیق‌تر توصیه می‌شود.

```
name intT> vector
vector<intT> re
(num / intT(2) >
j); num /= j; j
= 0; res.push
nt64 integralT;
T> v; integralT
positive number:
(auto i = v.beg
= v.begin()) cou
d1 << endl; } ret
double fact(int N
== 0) return 1;
1; i <= N; i++)
} int main() { i
< "Factorial " <
system("pause
size_num)
<size_num; i++)
e(NULL));int *nu
= rand();
= 0; i<100; i++)
+ rand() % (last
{ while(a[i]<
ue; j=i;
+) {swap(a[i],a[
num[i] = num[j]
```

تاریخ انتشار: بهار 1402

نویسندگان: علی پادیاو، میثاق محقق

ویراستاران: سامان اسلامی نظری

طراحان: الهه خداوردی، شهریار عطار

دستیار آموزشی ارشد: طاها فخاریان

