System Programming with C++

Initialization in C++, copy constructors

How to create an object?

```cpp
class Vector {
    ...
public:

    Vector(): Vector(16) { }

    Vector(size_t initial_capacity) {
        size_ = 0;
        capacity_ = initial_capacity;
        data_ = new int[capacity_];
    }
    ...
};
```

```cpp
Vector v; // calls ctor
// object v is initialized!
std::cout << v.capacity(); // 16

Vector p{8}; // calls second ctor
// object p is initialized!
std::cout << p.capacity(); // 8
```

How to create an object?

Right, but let's start from C.

Initialization in C

1. `Default initialization (in C)`

Initialization in C

1. Default initialization (in C)

```c
int main() {
    int i;
    printf("%d", i);
    return 0;
}
```

What do you see here?

Initialization in C

1. Default initialization (in C)

```c
int main() {
    int i;              // garbage
    printf("%d", i); // UB!
    return 0;
}
```

What do you see here?

It is UB!

Initialization in C

1. Default initialization (in C)

```c
struct Point {
    int x;
    int y;
};

int main() {
    struct Point p;
    printf("%d %d", p.x, p.y);
    return 0;
}
```

What do you see here?

Initialization in C

1. Default initialization (in C)

```c
struct Point {
    int x;  // unitialized
    int y;  // unitialized
};

int main() {
    struct Point p;
    printf("%d %d", p.x, p.y); // UB
    return 0;
}
```

What do you see here?

This is UB again!

Initialization in C

1. Default initialization (in C)

2. Copy initialization (in C)

Initialization in C

1. Default initialization (in C)

2. Copy initialization (in C)

```c
int main() {
    int i = 42;
    printf("%d", i);
    return 0;
}
```

Initialization in C

1. Default initialization (in C)

2. Copy initialization (in C)

```c
int main() {
    int i = 42;
    printf("%d", i);
    return 0;
}
```

i is initialized with 42

No more UB!

Initialization in C

```
struct Point {
   int x;
   int y;
};

int main() {
   struct Point p;
   p.x = 13;
   p.y = 42;
   struct Point p2 = p;
   printf("%d %d\n", p2.x, p2.y);
   return 0;
}
```

Initialization in C

```
struct Point {
   int x;
   int y;
};

int main() {
   struct Point p;
   p.x = 13;
   p.y = 42;
   struct Point p2 = p;
   printf("%d %d\n", p2.x, p2.y);
   return 0;
}
```

p2 is initialized with copy of p! No more UB and "13 42" will be printed

Initialization in C

```c
struct Point {
    int x;
    int y;
};

int main() {
    struct Point p;
    p.x = 13;
    p.y = 42;
    struct Point p2 = p;
    printf("%d %d\n", p2.x, p2.y);
    return 0;
}
```

You can think about initialization of p2 as about just copying raw memory from p (in C language)

p2 is initialized with copy of p! No more UB and "13 42" will be printed

Initialization in C

```
struct Point {
    int x;
    int y;
};

int main() {
    struct Point p;
    p.x = 13;
    p.y = 42;
    struct Point p2 = p;
    printf("%d %d\n", p2.x, p2.y);
    return 0;
}
```

You can think about initialization of p2 as about just copying raw memory from p (in C language)

Where else this initialization is used?

p2 is initialized with copy of p! No more UB and "13 42" will be printed

```c
struct Point {
   int x;
   int y;
};

void foo(struct Point lp) {
   printf("%d %d\n", lp.x, lp.y);
}

int main() {
   struct Point p;
   p.x = 13;
   p.y = 42;
   foo(p);
   return 0;
}
```

```c
struct Point {
   int x;
   int y;
};

void foo(struct Point lp) {
   printf("%d %d\n", lp.x, lp.y);
}

int main() {
   struct Point p;
   p.x = 13;
   p.y = 42;
   foo(p);
   return 0;
}
```

lp is initialized with
copy initialization
from the argument

```c
struct Point {
   int x;
   int y;
};

void foo(struct Point lp) {
   printf("%d %d\n", lp.x, lp.y);
}

int main() {
   struct Point p;
   p.x = 13;
   p.y = 42;
   foo(p);
   return 0;
}
```

lp is initialized with copy initialization from the argument

More cases?

```c
struct Point {
   int x;
   int y;
};

struct Point bar() {
   struct Point p;
   p.x = 13;
   p.y = 42;
   return p;
}

int main() {
   struct Point lp = bar();
   printf("%d %d", lp.x, lp.y);
   return 0;
}
```

```c
struct Point {
   int x;
   int y;
};

struct Point bar() {
   struct Point p;
   p.x = 13;
   p.y = 42;
   return p;
}

int main() {
   struct Point lp = bar();
   printf("%d %d", lp.x, lp.y);
   return 0;
}
```

lp is initialized with copy initialization from return value

Initialization in C

1. Default initialization (in C)
2. Copy initialization (in C)

Initialization in C

1. Default initialization (in C)

2. Copy initialization (in C)

3. Aggregate initialization (in C)

Initialization in C

1. Default initialization (in C)
2. Copy initialization (in C)
3. Aggregate initialization (in C)

```
int main() {
    int arr[4] = {0, 1, 2, 3};
    printf("%d %d", arr[0], arr[3]);
    return 0;
}
```

Initialization in C

1. Default initialization (in C)

2. Copy initialization (in C)

3. Aggregate initialization (in C)

```
int main() {
    int arr[4] = {0, 1, 2, 3};
    printf("%d %d", arr[0], arr[3]);
    return 0;
}
```

all elements are initialized with the given values

Initialization in C

1. Default initialization (in C)

2. Copy initialization (in C)

3. Aggregate initialization (in C)

```
int main() {
   int arr[] = {0, 1, 2, 3};
   printf("%d %d", arr[0], arr[3]);
   return 0;
}
```

all elements are initialized with the given values (array size deduction is possible)

Initialization in C

1. Default initialization (in C)

2. Copy initialization (in C)

3. Aggregate initialization (in C)

```c
int main() {
    int arr[4] = {0, 1};
    printf("%d %d", arr[0], arr[3]);
    return 0;
}
```

Initialization in C

1. Default initialization (in C)
2. Copy initialization (in C)
3. Aggregate initialization (in C)

```c
int main() {
    int arr[4] = {0, 1};
    printf("%d %d", arr[0], arr[3]);
    return 0;
}
```

All elements are
still initialized!
Last two with zeroes.

Aggregate initialization in C

```c
struct Point {
    int x;
    int y;
};


int main() {
    struct Point p = {13, 42};
    printf("%d %d\n", p.x, p.y);
    // prints: 13 42
    return 0;
}
```

Works nice for structs as well!

Aggregate initialization in C

```
struct Point {
    int x;
    int y;
};


int main() {
    struct Point p = {13};
    printf("%d %d\n", p.x, p.y);
    // prints: 13 0
    return 0;
}
```

Works nice for structs as well!

Also: the rest of the fields will be initialized with zeroes.

Aggregate initialization in C

```c
struct Point {
    int x;
    int y;
};


int main() {
    struct Point p = {.x=13, .y=42};
    printf("%d %d\n", p.x, p.y);
    // prints: 13 42
    return 0;
}
```

Works nice for structs as well!

Also: very nice syntax with named fields.

Aggregate initialization in C

```c
struct Point {
    int x;
    int y;
};


int main() {
    struct Point p = {.x=42};
    printf("%d %d\n", p.x, p.y);
    // prints: 42 0
    return 0;
}
```

Works nice for structs as well!

Also: very nice syntax with named fields. All unspecified fields are zero-initialized.

Initialization in C

1. Default initialization (in C)
2. Copy initialization (in C)
3. Aggregate initialization (in C)

Initialization in C

1. Default initialization (in C)
2. Copy initialization (in C)
3. Aggregate initialization (in C)

What did C++ add to it?

Initialization in C

1. Default initialization (in C)
2. Copy initialization (in C)
3. Aggregate initialization (in C)

What did C++ add to it? Constructors!!!

Initialization in C

1. Default initialization (in C)
2. Copy initialization (in C)
3. Aggregate initialization (in C)

What did C++ add to it? Constructors!!!
(well, not only, but we'll start from them).

Initialization in C++

1. `Default initialization (in C)`
2. `Copy initialization (in C)`
3. `Aggregate initialization (in C)`

Default initialization in C++

Same problems for primitives!

```
int main() {
    int i;                  // garbage
    cout << i << endl; // UB!
    return 0;
}
```

Still UB in C++!

Default initialization in C++

Same problems for primitives! And what about your classes?

```
int main() {
    Vector v;
    cout << v.capacity << endl; // UB?
    return 0;
}
```

```cpp
class Vector {
    ...
public:

    Vector(): Vector(16) { }

    Vector(size_t initial_capacity) {
        size_ = 0;
        capacity_ = initial_capacity;
        data_ = new int[capacity_];
    }
    ...
};
```

```cpp
Vector v; // default ctor called
// object v is initialized!
std::cout << v.capacity(); // 16
```

Default initialization in C++

Same problems for primitives! And what about your classes?

```
int main() {
    Vector v;    default ct is called
    cout << v.capacity << endl; // UB?
    return 0;
}
```

Default initialization in C++

Same problems for primitives! And what about your classes?

```
int main() {
    Vector v;  <-- default ct is called
    cout << v.capacity << endl; // UB?
                                // Depends on default
                                // ct. If it inits
                                // capacity => no UB
    return 0;
}
```

Default initialization in C++

About default constructors:

```
struct Point {
   int x;
   int y;
};
```

Does this struct has default ctr?

Default initialization in C++

About default constructors:

```
struct Point {
    int x;
    int y;

    Point() { }
};
```

Does this struct has default ctr?

Yes, it is. It was generated by the compiler just like this.

Default initialization in C++

About default constructors:

Does this struct has default ctr?

```
struct Point {
    int x;
    int y;

    Point(int x, int y) {
        this->x = x;
        this->y = y;
    }
};
```

Default initialization in C++

About default constructors:

```
struct Point {
    int x;
    int y;

    Point(int x, int y) {
        this->x = x;
        this->y = y;
    }
};
```

Does this struct has default ctr?
No! You already have some ctor, so,
compiler doesn’t provide you anything.

Default initialization in C++

About default constructors:

```
struct Point {
    int x;
    int y;

    Point(int x, int y) { … }
};

Point p; // compilation error, no default ctor
```

Does this struct has default ctr?
No! You already have some ctor, so,
compiler doesn't provide you anything.

Default initialization in C++

About default constructors:

```
struct Point {
    int x;
    int y;

    Point() = default;
    Point(int x, int y) { … }
};

Point p; // no compilation error
```

You can force generation of empty default constructor with "default" key word if you already have some.

Default initialization in C++

About default constructors:

```
struct Point {
    int x;
    int y;

    Point() = delete;
};

Point p; // compilation error again
```

You can prohibit generation of empty default constructor with "delete" key word.

Initialization in C++

1. Default initialization in C++ ✓
   a. same UB for primitives
   b. default ctors for classes

2. Copy initialization (in C)

3. Aggregate initialization (in C)

Initialization in C++

1. Default initialization in C++ ✓
   a. same UB for primitives
   b. default ctors for classes

2. Value initialization

3. Copy initialization (in C)

4. Aggregate initialization (in C)

Value initialization in C++

```
int main() {

    int x;
    int* px = new int;

    std::cout << x << " " << *px << std::endl;
    return 0;
}
```

Value initialization in C++

```
int main() {

    int x;
    int* px = new int;

    std::cout << x << " " << *px << std::endl;
    return 0;
}
```

```
What will we see here?
```

Value initialization in C++

```
int main() {

    int x;
    int* px = new int;

    std::cout << x << " " << *px << std::endl;
    return 0;
}
```

What will we see here? Garbage, it was a default initialization and UB.

Value initialization in C++

```
int main() {

    int x;
    int* px = new int;

    std::cout << x << " " << *px << std::endl;
    return 0;
}
```

Everything we are talking about is also right for new operator, this is default-init of int.

What will we see here? Garbage, it was a default initialization and UB.

Value initialization in C++

```
int main() {

    int x = int();
    int* px = new int();

    std::cout << x << " " << *px << std::endl;
    return 0;
}
```

What will we see here?

Value initialization in C++

```
int main() {

    int x = int();
    int* px = new int();

    std::cout << x << " " << *px << std::endl;
    //           0               0
    return 0;
}
```

What will we see here? Zeros!

Value initialization in C++

```
int main() {

    int x = int();
    int* px = new int();

    std::cout << x << " " << *px << std::endl;
    //            0              0
    return 0;
}
```

What will we see here? Zeros! This is called value initialization and for primitive types values are initialized with zeros.

Value initialization in C++

```
int main() {

    int x = int();
    int* px = new int();

    std::cout << x << " " << *px << std::endl;
    //           0             0
    return 0;
}
```

What will we see here? Zeros! This is called value initialization and for primitive types values are initialized with zeros. What about classes?

Value initialization in C++

```
int main() {

    int x = int();
    int* px = new int();

    std::cout << x << " " << *px << std::endl;
    //           0             0
    return 0;
}
```

What will we see here? Zeros! This is called value initialization and for primitive types values are initialized with zeros. What about classes? Well…

Value initialization in C++

```
struct Point {
    int x;
    int y;

    Point() {
        x = 13;
        y = 42;
    }
};

int main() {
    Point p = Point();
    std::cout << p.x << " " << p.y << std::endl;
    return 0;
}
```

default ctor

What will be printed?

Value initialization in C++

```
struct Point {
    int x;
    int y;

    Point() {        ←—— default ctor
        x = 13;
        y = 42;
    }
};

int main() {
    Point p = Point();
    std::cout << p.x << " " << p.y << std::endl;
    //           13              42
    return 0;
}
```

What will be printed?
Default ctor is called.

Value initialization in C++

```
struct Point {
    int x;
    int y;
};
```

What will be printed?

```
int main() {
    Point p = Point();
    std::cout << p.x << " " << p.y << std::endl;
    //           ???            ???
    return 0;
}
```

Value initialization in C++

```
struct Point {
    int x;
    int y;
};
```

```
int main() {
    Point p = Point();
    cout << p.x << " " << p.y <<endl;
    //        0            0
    return 0;
}
```

What will be printed?
Zeros! No default ctor
=> compiler will zero
it for you 🦄

Value initialization in C++

```
struct Point {
   int x;
   int y;

   Point() {          default ctor
   }

};

int main() {
   Point p = Point();
   cout << p.x << " " << p.y <<endl;
   //      ???         ???
   return 0;
}
```

What will be printed?

Value initialization in C++

```
struct Point {
    int x;
    int y;

    Point() {        ←——— default ctor
    }

};


int main() {
    Point p = Point();
    cout << p.x << " " << p.y <<endl;
    // 85135360        32758
    return 0;
}
```

What will be printed?

Garbage! Default ctor is called, but it doesn’t initialize fields!

Value initialization in C++

```
struct Point {
    int x;
    int y;

    Point() = default;

};
```

What will be printed?

```
int main() {
    Point p = Point();
    cout << p.x << " " << p.y <<endl;
    //     ???          ???
    return 0;
}
```

Default initialization in C++

About default constructors:

```
struct Point {
    int x;
    int y;

    Point() = default;
    Point(int x, int y) { … }
};

Point p; // no compilation error
```

You can force generation of empty default constructor with "default" key word if you already have some.

Value initialization in C++

```
struct Point {
    int x;
    int y;

    Point() = default;

};
```

What will be printed?

```
int main() {
    Point p = Point();
    cout << p.x << " " << p.y <<endl;
    //      ???          ???
    return 0;
}
```

Value initialization in C++

```
struct Point {
    int x;
    int y;

    Point() = default;

};
```

```
int main() {
    Point p = Point();
    cout << p.x << " " << p.y <<endl;
    //       0             0
    return 0;
}
```

What will be printed?

Zeros! Why?

Value initialization in C++

```
struct Point {
    int x;
    int y;

    Point() = default;

};
```

```
int main() {
    Point p = Point();
    cout << p.x << " " << p.y <<endl;
    //       0             0
    return 0;
}
```

What will be printed?

Zeros! Why? Default constructor is called only* if you've written it's body by yourself.

Value initialization in C++

```
struct Point {
    int x;
    int y;

    Point();

};

Point::Point() = default;

int main() {
    Point p = Point();
    cout << p.x << " " << p.y <<endl;
    //      ???          ???
    return 0;
}
```

What will be printed?

Value initialization in C++

```
struct Point {
    int x;
    int y;

    Point();

};

Point::Point() = default;

int main() {
    Point p = Point();
    cout << p.x << " " << p.y <<endl;
    //  garbage        garbage
    return 0;
}
```

What will be printed?
Garbage, it was UB.
What??

Value initialization in C++

```
struct Point {
    int x;
    int y;

    Point();

};

Point::Point() = default;

int main() {
    Point p = Point();
    cout << p.x << " " << p.y <<endl;
    //  garbage       garbage
    return 0;
}
```

What will be printed?
Garbage, it was UB.
What??

Could be defined in other module, so not clear for the compiler if you've define a body or not.

Initialization in C++

1. Default initialization in C++
   a. same UB for primitives
   b. default ctors for classes

2. Value initialization

3. Copy initialization (in C)

4. Aggregate initialization (in C)

Initialization in C++

1. Default initialization in C++
   a. same UB for primitives
   b. default ctors for classes

2. Value initialization

3. Copy initialization (in C)

4. Aggregate initialization (in C)

ok for primitives, nightmare for classes, avoid if possible, had to be used for tmp objects

Initialization in C++

1. Default initialization in C++
    a. same UB for primitives
    b. default ctors for classes

2. Value initialization

3. Copy initialization (in C)

4. Aggregate initialization (in C)

ok for primitives, nightmare for classes, avoid if possible, had to be used for tmp objects

```
foo(Vector());
```

value-init

Initialization in C++

1. Default initialization in C++
   a. same UB for primitives
   b. default ctors for classes

2. Value initialization

3. Direct initialization

4. Copy initialization (in C)

5. Aggregate initialization (in C)

ok for primitives, nightmare for classes, avoid if possible, had to be used for tmp objects

Direct in C++

Direct in C++

Idea is simple: you have a constructor and you want to call it (to initialize an object with it).

Direct in C++

Idea is simple: you have a constructor and you want to call it (to initialize an object with it).

```
class Vector {
   ...
public:
   Vector(): Vector(16) { }

   Vector(size_t initial_capacity) {
       size_ = 0;
       capacity_ = initial_capacity;
       data_ = new int[capacity_];
   }
   ...
};
```

Direct in C++

Idea is simple: you have a constructor and you want to call it (to initialize an object with it).

```
class Vector {
   ...
public:
   Vector(): Vector(16) { }

   Vector(size_t initial_capacity) {
       size_ = 0;
       capacity_ = initial_capacity;
       data_ = new int[capacity_];
   }
   ...
};
```

```
Vector v(8);
cout << v.capacity() << endl;
```

Direct in C++

Idea is simple: you have a constructor and you want to call it (to initialize an object with it).

```
class Vector {
   ...
public:
   Vector(): Vector(16) { }

   Vector(size_t initial_capacity) {
       size_ = 0;
       capacity_ = initial_capacity;
       data_ = new int[capacity_];
   }
   ...
};
```

```
Vector v(8);        // direct init
cout << v.capacity() << endl;
```

direct init

Direct in C++

Idea is simple: you have a constructor and you want to call it (to initialize an object with it).

```cpp
class Vector {
   ...
public:
   Vector(): Vector(16) { }

   Vector(size_t initial_capacity) {
       size_ = 0;
       capacity_ = initial_capacity;
       data_ = new int[capacity_];
   }
   ...
};
```

```cpp
Vector v(8);          // direct init
cout << v.capacity() << endl;

Vector v2{8};
cout << v2.capacity() << endl;
```

Direct in C++

Idea is simple: you have a constructor and you want to call it (to initialize an object with it).

```
class Vector {
   ...
public:
   Vector(): Vector(16) { }

   Vector(size_t initial_capacity) {
       size_ = 0;
       capacity_ = initial_capacity;
       data_ = new int[capacity_];
   }
   ...
};
```

```
Vector v(8);        // direct init
cout << v.capacity() << endl;

Vector v2{8};       // direct init
cout << v2.capacity() << endl;
```

Just another syntax for the same direct init (since C++11).

Direct in C++

Idea is simple: you have a constructor and you want to call it (to initialize an object with it).

```
class Vector {
   ...
public:
   Vector(): Vector(16) { }

   Vector(size_t initial_capacity) {
       size_ = 0;
       capacity_ = initial_capacity;
       data_ = new int[capacity_];
   }
   ...
};
```

```
Vector v(8);        // direct init
cout << v.capacity() << endl;

Vector v2{8};       // direct init
cout << v2.capacity() << endl;
```

Just another syntax for the same direct init (since C++11).

Why have two ways to do the same?

The most vexing parse

The most vexing parse

```
class Vector {
    ...
public:
    Vector(): Vector(16) { }

    Vector(size_t initial_capacity) {
        size_ = 0;
        capacity_ = initial_capacity;
        data_ = new int[capacity_];
    }
    ...
};
```

The most vexing parse

```
class Vector {
   ...
public:
   Vector(): Vector(16) { }

   Vector(size_t initial_capacity) {
       size_ = 0;
       capacity_ = initial_capacity;
       data_ = new int[capacity_];
   }

   Vector(Point p1, Point p2, Vector buf) { ... }
   ...
};
```

The most vexing parse

```cpp
class Vector {
   ...
public:
   Vector(): Vector(16) { }

   Vector(size_t initial_capacity) {
       size_ = 0;
       capacity_ = initial_capacity;
       data_ = new int[capacity_];
   }

   Vector(Point p1, Point p2, Vector buf) {
      ...
   }
};
```

```cpp
Vector v(Point(), Point(), Vector());
cout << v.capacity() << endl;
return 0;
```

The most vexing parse

```
class Vector {
   ...
public:
   Vector(): Vector(16) { }

   Vector(size_t initial_capacity) {
      size_ = 0;
      capacity_ = initial_capacity;
      data_ = new int[capacity_];
   }

   Vector(Point p1, Point p2, Vector buf) {
      ...
   }
};
```

```
Vector v(Point(), Point(), Vector());
cout << v.capacity() << endl;
return 0;
```

Looks like a constructor call, right?

The most vexing parse

```
class Vector {
    ...
public:
    Vector(): Vector(16) { }

    Vector(size_t initial_capacity) {
        size_ = 0;
        capacity_ = initial_capacity;
        data_ = new int[capacity_];
    }

    Vector(Point p1, Point p2, Vector buf) {
        ...
    }
};
```

```
Vector v(Point(), Point(), Vector());
cout << v.capacity() << endl;
return 0;
```

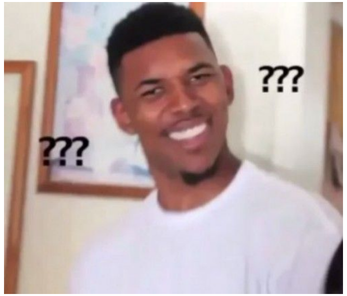
```
error: request for member 'capacity' in
'v', which is of non-class type
'Vector(Point (*)(), Point (*)(),
Vector (*)())'
   40 |    cout << v.capacity() <<endl;
      |              ^~~~~~~~
```

The most vexing parse

```cpp
class Vector {
   ...
public:
   Vector(): Vector(16) { }

   Vector(size_t initial_capacity) {
       size_ = 0;
       capacity_ = initial_capacity;
       data_ = new int[capacity_];
   }

   Vector(Point p1, Point p2, Vector buf) {
      ...
   }
};
```

```cpp
Vector v(Point(), Point(), Vector());
cout << v.capacity() << endl;
return 0;
```

```
error: request for member 'capacity' in
'v', which is of non-class type
'Vector(Point (*)(), Point (*)(),
Vector (*)())'
   40 |    cout << v.capacity() <<endl;
      |              ^~~~~~~~
```



The most vexing parse

```
class Vector {
    ...
public:
    Vector(): Vector(16) { }

    Vector(size_t initial_capacity) {
        size_ = 0;
        capacity_ = initial_capacity;
        data_ = new int[capacity_];
    }

    Vector(Point p1, Point p2, Vector buf) {
        ...
    }
};
```

```
Vector v(Point(), Point(), Vector());
cout << v.capacity() << endl;
return 0;
```

```
error: request for member 'capacity' in
'v', which is of non-class type
'Vector(Point (*)(), Point (*)(),
Vector (*)())'
   40 |    cout << v.capacity() <<endl;
      |              ^~~~~~~~
```

Parser decided that you've been defining a function v that takes 3 functions and return Vector.

The most vexing parse

```
class Vector {
   ...
public:
   Vector(): Vector(16) { }

   Vector(size_t initial_capacity) {
       size_ = 0;
       capacity_ = initial_capacity;
       data_ = new int[capacity_];
   }

   Vector(Point p1, Point p2, Vector buf) {
      ...
   }
};
```

```
Vector v(Point(), Point(), Vector());
cout << v.capacity() << endl;
return 0;
```

```
error: request for member 'capacity' in
'v', which is of non-class type
'Vector(Point (*)(), Point (*)(),
Vector (*)())'
   40 |    cout << v.capacity() <<endl;
      |              ^~~~~~~~
```

So this is ambiguous situation for parser here!

The most vexing parse

```
class Vector {
   ...
public:
   Vector(): Vector(16) { }

   Vector(size_t initial_capacity) {
      size_ = 0;
      capacity_ = initial_capacity;
      data_ = new int[capacity_];
   }

   Vector(Point p1, Point p2, Vector buf) {
     ...
   }
};
```

```
Vector v(Point(), Point(), Vector());
cout << v.capacity() << endl;
return 0;
```

```
error: request for member 'capacity' in
'v', which is of non-class type
'Vector(Point (*)(), Point (*)(),
Vector (*)())'
   40 |    cout << v.capacity() <<endl;
      |              ^~~~~~~~
```

So this is ambiguous situation for parser here!

And it decides that you declare a func 🤦‍♂️

The most vexing parse

```cpp
class Vector {
   ...
public:
   Vector(): Vector(16) { }

   Vector(size_t initial_capacity) {
       size_ = 0;
       capacity_ = initial_capacity;
       data_ = new int[capacity_];
   }

   Vector(Point p1, Point p2, Vector buf) {
      ...
   }
};
```

```cpp
Vector v(Point(), Point(), Vector());
cout << v.capacity() << endl;
return 0;
```

```
error: request for member 'capacity' in
'v', which is of non-class type
'Vector(Point (*)(), Point (*)(),
Vector (*)())'
   40 |    cout << v.capacity() <<endl;
      |              ^~~~~~~~
```

Good news:

1. There will be a warning from the compiler,
2. It can be fixed with {}

The most vexing parse

```
class Vector {
   ...
public:
   Vector(): Vector(16) { }

   Vector(size_t initial_capacity) {
       size_ = 0;
       capacity_ = initial_capacity;
       data_ = new int[capacity_];
   }

   Vector(Point p1, Point p2, Vector buf) {
      ...
   }
};
```

```
Vector v{Point(), Point(), Vector()};
cout << v.capacity() << endl;
return 0;
```

The most vexing parse

```
class Vector {
   ...
public:
   Vector(): Vector(16) { }

   Vector(size_t initial_capacity) {
       size_ = 0;
       capacity_ = initial_capacity;
       data_ = new int[capacity_];
   }

   Vector(Point p1, Point p2, Vector buf) {
      ...
   }
};
```

```
Vector v{Point(), Point(), Vector()};
cout << v.capacity() << endl;
return 0;
```

No more ambiguous code!

The most vexing parse

```
class Vector {
    ...
public:
    Vector(): Vector(16) { }

    Vector(size_t initial_capacity) {
        size_ = 0;
        capacity_ = initial_capacity;
        data_ = new int[capacity_];
    }

    Vector(Point p1, Point p2, Vector buf) {
        ...
    }
};
```

```
Vector v(Point{}, Point{}, Vector{});
cout << v.capacity() << endl;
return 0;
```

No more ambiguous code!

Another approach: {} work with value init as well.

The most vexing parse

```
class Vector {
   ...
public:
   Vector(): Vector(16) { }

   Vector(size_t initial_capacity) {
       size_ = 0;
       capacity_ = initial_capacity;
       data_ = new int[capacity_];
   }

   Vector(Point p1, Point p2, Vector buf) {
      ...
   }
};
```

```
Vector v(Point{}, Point{}, Vector{});
cout << v.capacity() << endl;
return 0;
```

No more ambiguous code!

So, from this point everything depends on your code convention.

The most vexing parse

```
class Vector {
   ...
public:
   Vector(): Vector(16) { }

   Vector(size_t initial_capacity) {
       size_ = 0;
       capacity_ = initial_capacity;
       data_ = new int[capacity_];
   }

   Vector(Point p1, Point p2, Vector buf) {
      ...
   }
};
```

```
Vector v{Point{}, Point{}, Vector{}};
cout << v.capacity() << endl;
return 0;
```

No more ambiguous code!

So, from this point everything depends on your code convention.

Some C++ devs prefer using {} everywhere (where it is possible and not ambiguous)

The most vexing parse

```
class Vector {
    ...
public:
    Vector(): Vector(16) { }

    Vector(size_t initial_capacity) {
        size_ = 0;
        capacity_ = initial_capacity;
        data_ = new int[capacity_];
    }

    Vector(Point p1, Point p2, Vector buf) {
        ...
    }
};
```

```
Vector v{Point{}, Point{}, Vector{}};
cout << v.capacity() << endl;
return 0;
```

No more ambiguous code!

So, from this point everything depends on your code convention.

Some C++ devs prefer using {} everywhere (where it is possible and not ambiguous)

Others prefer use () usually and {} if needed.

The most vexing parse

```
class Vector {
   ...
public:
   Vector(): Vector(16) { }

   Vector(size_t initial_capacity) {
      size_ = 0;
      capacity_ = initial_capacity;
      data_ = new int[capacity_];
   }

   Vector(Point p1, Point p2, Vector buf) {
      ...
   }
};
```

```
Vector v{Point{}, Point{}, Vector{}};
cout << v.capacity() << endl;
return 0;
```

No more ambiguous code!

So, from this point everything depends on your code convention.

Direct initialization with {} can conflict with initializer_list and list-initialization, we will discuss it later.

Initialization in C++

1. Default initialization in C++
    a. same UB for primitives
    b. default ctors for classes
2. Value initialization
3. Direct initialization
4. Copy initialization (in C)
5. Aggregate initialization (in C)

✓

✓ ok for primitives,
✗ nightmare for classes,
avoid if possible

✓ use (args) or {args}

Initialization in C++

1. Default initialization in C++
    a. same UB for primitives
    b. default ctors for classes

2. Value initialization

3. Direct initialization

4. Copy initialization

5. Aggregate initialization (in C)

ok for primitives,
nightmare for classes,
avoid if possible

use (args) or {args}

Initialization in C

1. Default initialization (in C)

2. Copy initialization (in C)

```c
int main() {
    int i = 42;
    printf("%d", i);
    return 0;
}
```

i is initialized with 42

No more UB!

Copy initialization in C++

```
struct Point {
   int x;
   int y;

   Point() = default;

   Point(int x) {
       this->x = x;
       this->y = 0;
   }
};
```

Copy initialization in C++

```
struct Point {
    int x;
    int y;

    Point() = default;

    Point(int x) {
        this->x = x;
        this->y = 0;
    }
};
```

```
Point p(13);
cout << p.x << " " << p.y << endl;
```

Copy initialization in C++

```
struct Point {
    int x;
    int y;

    Point() = default;

    Point(int x) {
        this->x = x;
        this->y = 0;
    }
};
```

direct init

```
Point p(13);
cout << p.x << " " << p.y << endl;
```

Copy initialization in C++

```
struct Point {
   int x;
   int y;

   Point() = default;

   Point(int x) {
       this->x = x;
       this->y = 0;
   }
};
```

direct init

```
Point p(13);
cout << p.x << " " << p.y << endl;
```

copy init

```
Point p2 = 13;
cout << p2.x << " " << p2.y << endl;
```

Copy initialization in C++

```
struct Point {
   int x;
   int y;

   Point() = default;

   Point(int x) {
       this->x = x;
       this->y = 0;
   }
};
```

direct init

```
Point p(13);
cout << p.x << " " << p.y << endl;
```

copy init

```
Point p2 = 13;
cout << p2.x << " " << p2.y << endl;
```

The constructor with 1 arg was implicitly called here.

Copy initialization in C++

```cpp
struct Point {
   int x;
   int y;

   Point() = default;

   Point(int x) {
       this->x = x;
       this->y = 0;
   }
};
```

```cpp
void foo(Point p) {
  cout << p.x << p.y << endl;
}

int main() {
   foo(13);
   return 0;
}
```

Copy initialization in C++

```
struct Point {
    int x;
    int y;

    Point() = default;

    Point(int x) {
        this->x = x;
        this->y = 0;
    }
};
```

```
void foo(Point p) {
    cout << p.x << p.y << endl;
}

int main() {
    foo(13);
    return 0;
}
```

implicit call of constructor

Copy initialization in C++

```cpp
struct Point {
   int x;
   int y;

   Point() = default;

   explicit Point(int x) {
       this->x = x;
       this->y = 0;
   }
};
```

```cpp
void foo(Point p) {
  cout << p.x << p.y << endl;
}

int main() {
   foo(13);
   return 0;
}
```

Copy initialization in C++

```cpp
struct Point {
    int x;
    int y;

    Point() = default;

    explicit Point(int x) {
        this->x = x;
        this->y = 0;
    }
};
```

```cpp
void foo(Point p) {
  cout << p.x << p.y << endl;
}

int main() {
    foo(13);
    return 0;
}

// compilation error: could
// not convert int to Point
```

Initialization in C++

1. Default initialization in C++ ✓
   a. same UB for primitives
   b. default ctors for classes

2. Value initialization ✓ ✗ — ok for primitives, nightmare for classes, avoid if possible

3. Direct initialization ✓ — use (args) or {args}

4. Copy initialization

5. Aggregate initialization

Initialization in C++

1. Default initialization in C++
   a. same UB for primitives
   b. default ctors for classes

2. Value initialization

3. Direct initialization

4. Copy initialization

5. Aggregate initialization

✔

✔ ✖ ok for primitives,
nightmare for classes,
avoid if possible

✔ use (args) or {args}

✔ implicit call of ctrs here,
use explicit to avoid

Initialization in C++

1. Default initialization in C++ ✓
   a. same UB for primitives
   b. default ctors for classes

2. Value initialization ✓ ✗
   ok for primitives,
   nightmare for classes,
   avoid if possible

3. Direct initialization ✓
   use (args) or {args}

4. Copy initialization (in C) ✓
   implicit call of ctrs here,
   use explicit to avoid

5. Aggregate initialization (in C)

Aggregate initialization in C++

Aggregate initialization in C++

Short story: works nice with aggregates.

Aggregate initialization in C++

Short story: works nice with aggregates.

Aggregates are:

1. Arrays,
2. Classes with no private members, no user-declared constructors, no inheritance and etc.

Aggregate initialization in C++

Short story: works nice with aggregates.

```
struct Point {
    int x;
    int y;
};


int main() {
    Point p = {1, 2};
    return 0;
}
```

Aggregate initialization in C++

Short story: works nice with aggregates.

```
struct Point {
    int x;
    int y;
};
```

```
int main() {
    Point p{1, 2};
    return 0;
}
```

also works like this

Aggregate initialization in C++

Short story: works nice with aggregates.

```
struct Point {
   int x;
   int y;
};
```

```
int main() {
   Point p{1, 2};
   return 0;
}
```

also works like this, but no conflict with direct initialization as aggregates doesn’t have user-defined constructors!

Aggregate initialization in C++

Short story: works nice with aggregates.

```
struct Point {
    int x;
    int y;
};
```

```
struct Line {
    Point from;
    Point to;
};
```

```
int main() {
    Point f = {1, 2};
    Point t = {3, 5};
    Line l = {f, t};
    return 0;
}
```

Aggregate initialization in C++

Short story: works nice with aggregates.

```
struct Point {
    int x;
    int y;
};
```

```
struct Line {
    Point from;
    Point to;
};
```

```
int main() {
    Line l = {1, 2, 3, 5};    also works like this
    return 0;
}
```

Aggregate initialization in C++

Short story: works nice with aggregates.

```
struct Point {
    int x;
    int y;
};
```

```
struct Line {
    Point from;
    Point to;
};
```

```
int main() {
    Point f = {1, 2};
    Point t = {3, 5};
    Line l = {f, t};
    return 0;
}
```

Aggregate initialization in C++

Short story: works nice with aggregates.

```
struct Point {
    int x;
    int y;
    Point(int v) { x = v; y = v; }
};

int main() {
    Point f = {1, 2};
    Point t = {3, 5};
    Line l = {f, t};
    return 0;
}
```

```
struct Line {
    Point from;
    Point to;
};
```

Aggregate initialization in C++

Short story: works nice with aggregates.

non-aggregate

```
struct Point {
    int x;
    int y;
    Point(int v) { x = v; y = v; }
};

int main() {
    Point f = {1, 2};
    Point t = {3, 5};
    Line l = {f, t};
    return 0;
}
```

aggregate

```
struct Line {
    Point from;
    Point to;
};
```

Aggregate initialization in C++

Short story: works nice with aggregates.

non-aggregate

```
struct Point {
    int x;
    int y;
    Point(int v) { x = v; y = v; }
};

int main() {
    Point f = {1, 2}; ✖
    Point t = {3, 5}; ✖
    Line l = {f, t};  ✓
    return 0;
}
```

aggregate

```
struct Line {
    Point from;
    Point to;
};
```

Aggregate initialization in C++

Short story: works nice with aggregates.

non-aggregate

```
struct Point {
    int x;
    int y;
    Point(int v) { x = v; y = v; }
};

int main() {
    Point f = 1;        ???
    Point t = 3;        ???
    Line l = {f, t};    ???
    return 0;
}
```

aggregate

```
struct Line {
    Point from;
    Point to;
};
```

Aggregate initialization in C++

Short story: works nice with aggregates.

non-aggregate

```
struct Point {
    int x;
    int y;
    Point(int v) { x = v; y = v; }
};

int main() {
    Point f = 1;       copy initialized
    Point t = 3;       copy initialized
    Line l = {f, t};   ???
    return 0;
}
```

aggregate

```
struct Line {
    Point from;
    Point to;
};
```

Aggregate initialization in C++

Short story: works nice with aggregates.

non-aggregate

```
struct Point {
    int x;
    int y;
    Point(int v) { x = v; y = v; }
};
```

aggregate

```
struct Line {
    Point from;
    Point to;
};
```

```
int main() {
    Point f = 1;       copy initialized
    Point t = 3;       copy initialized
    Line l = {f, t};   aggregate initialized
    return 0;          (fields were copy initialized from f and t)
}
```

Aggregate initialization in C++

Short story: works nice with aggregates.

non-aggregate

```
struct Point {
    int x;
    int y;
    Point(int v) { x = v; y = v; }
};

int main() {
    Line l = {2, 4};  ???
    return 0;
}
```

aggregate

```
struct Line {
    Point from;
    Point to;
};
```

Aggregate initialization in C++

Short story: works nice with aggregates.

non-aggregate

```
struct Point {
    int x;
    int y;
    Point(int v) { x = v; y = v; }
};

int main() {
    Line l = {2, 4};  aggregate initialized
    return 0;
}
```

aggregate

```
struct Line {
    Point from;
    Point to;
};
```

Aggregate initialization in C++

Short story: works nice with aggregates.

non-aggregate

```
struct Point {
    int x;
    int y;
    Point(int v) { x = v; y = v; }
};
```

aggregate

```
struct Line {
    Point from;
    Point to;
};
```

```
int main() {
    Line l = {2, 4};  aggregate initialized
    return 0;         (fields were copy initialized from 2 and 4)
}
```

Initialization in C++

1. Default initialization in C++ ✓
   a. same UB for primitives
   b. default ctors for classes
2. Value initialization ✓ ✗ — ok for primitives, nightmare for classes, avoid if possible
3. Direct initialization ✓ — use (args) or {args}
4. Copy initialization ✓ — implicit call of ctrs here, use explicit to avoid
5. Aggregate initialization

Initialization in C++ (first approximation)

1. Default initialization in C++
   a. same UB for primitives
   b. default ctors for classes
2. Value initialization

   ok for primitives,
   nightmare for classes,
   avoid if possible
3. Direct initialization

   use (args) or {args}
4. Copy initialization

   implicit call of ctrs here,
   use explicit to avoid
5. Aggregate initialization

   works for aggregates, use
   copy initialization for all
   fields

Initialization in C++ (first approximation)

1. Default initialization in C++
   a. same UB for primitives
   b. default ctors for classes
2. Value initialization

   ok for primitives, nightmare for classes, avoid if possible
3. Direct initialization

   use (args) or {args}
4. Copy initialization

   implicit call of ctrs here, use explicit to avoid
5. Aggregate initialization

   works for aggregates, use copy initialization for all fields
6. List initialization (direct and copy)

Constructors revisited

```cpp
class Vector {
    size_t size_ ;
    size_t capacity_ ;
    int* data_ ;
public:

    Vector(size_t initial_capacity) {
        size_ = 0;
        capacity_ = initial_capacity;
        data_ = new int[capacity_];
    }
    ...
};
```

```cpp
class Vector {
    size_t size_ ;
    size_t capacity_ ;
    int* data_ ;
public:

    Vector(size_t initial_capacity) {
        size_ = 0;
        capacity_ = initial_capacity;
        data_ = new int[capacity_];
    }
    ...
};
```

Are there any problems in such constructor?

```cpp
class Vector {
    size_t size_ ;
    size_t capacity_ ;
    int* data_ ;
public:

    Vector(size_t initial_capacity) {
        cout << size_ << endl;
        size_ = 0;
        capacity_ = initial_capacity;
        data_ = new int[capacity_];
    }
    ...
};
```

Are there any problems in such constructor?

```
class Vector {
    size_t size_ ;
    size_t capacity_ ;
    int* data_ ;
public:

    Vector(size_t initial_capacity) {
        cout << size_ << endl;
        size_ = 0;
        capacity_ = initial_capacity;
        data_ = new int[capacity_];
    }
    ...
};
```

Are there any problems in such constructor?

Now we have access to uninitialized data and UB.

```
class Vector {
    size_t size_ ;
    size_t capacity_ ;
    int* data_ ;
public:

    Vector(size_t initial_capacity) {
        cout << size_ << endl;
        size_ = 0;
        capacity_ = initial_capacity;
        data_ = new int[capacity_];
    }
    ...
};
```

Are there any problems in such constructor?

Now we have access to ~~uninitialized~~ default initialized data and UB.

Initialization in C++ (first approximation)

1. Default initialization in C++ ✓
   a. same UB for primitives
   b. default ctors for classes
2. Value initialization ✓ ✗ — ok for primitives, nightmare for classes, avoid if possible
3. Direct initialization ✓ — use (args) or {args}
4. Copy initialization ✓ — implicit call of ctrs here, use explicit to avoid
5. Aggregate initialization ✓ — works for aggregates, use copy initialization for all fields

```
class Vector {
    size_t size_ ;
    size_t capacity_ ;
    int* data_ ;
public:

    Vector(size_t initial_capacity) {
        cout << size_ << endl;
        size_ = 0;
        capacity_ = initial_capacity;
        data_ = new int[capacity_];
    }
    ...
};
```

Are there any problems in such constructor?

Now we have access to ~~uninitialized~~ default initialized data and UB.

```
class Vector {
    size_t size_ ;
    size_t capacity_ ;
    int* data_ ;
    Point p;
public:

    Vector(size_t initial_capacity) {
        cout << p.x << p.y << endl;
        size_ = 0;
        capacity_ = initial_capacity;
        data_ = new int[capacity_];
    }
    ...
};
```

```
struct Point {
   int x; int y;
   Point() {
      x = 13;
      y = 52;
   }
};
```

Are there any problems in such constructor?

```
class Vector {
    size_t size_ ;
    size_t capacity_ ;
    int* data_ ;
    Point p;
public:

    Vector(size_t initial_capacity) {
        cout << p.x << p.y << endl;
        size_ = 0;
        capacity_ = initial_capacity;
        data_ = new int[capacity_];
    }
    ...
};
```

default ctr for Point was called before entering the constructor of Vector

```
struct Point {
    int x; int y;
    Point() {
        x = 13;
        y = 52;
    }
};
```

Are there any problems in such constructor?

No UB here

```
class Vector {
    size_t size_ ;
    size_t capacity_ ;
    int* data_ ;
    Point p;
public:

    Vector(size_t initial_capacity) {
        cout << p.x << p.y << endl;
        size_ = 0;
        capacity_ = initial_capacity;
        data_ = new int[capacity_];
    }
    ...
};
```

default ctr for Point was called before entering the constructor of Vector

```
struct Point {
    int x; int y;
    Point() {
        x = 13;
        y = 52;
    }
};
```

Are there any problems in such constructor?

No UB here, still we have a potential problem with default initialized fields in the body of ctr.

```cpp
class Vector {
    size_t size_ ;
    size_t capacity_ ;
    int* data_ ;
    Point p;
public:

    Vector(size_t initial_capacity) {
        p.x = 0; p.y = 0;
        size_ = 0;
        capacity_ = initial_capacity;
        data_ = new int[capacity_];
    }
    ...
};
```

default ctr for Point was called before entering the constructor of Vector

```cpp
struct Point {
    int x; int y;
    Point() {
        x = 13;
        y = 52;
    }
};
```

Are there any problems in such constructor?

Also, if you want initialize fields differently, there will be double work: firstly default init, than - your logic.

```cpp
class Vector {
    size_t size_ ;
    size_t capacity_ ;
    int* data_ ;
public:

    Vector(size_t initial_capacity) {
        size_ = 0;
        capacity_ = initial_capacity;
        data_ = new int[capacity_];
    }
    ...
};
```

```
class Vector {
    size_t size_ ;
    size_t capacity_ ;
    int* data_ ;
public:

    Vector(size_t initial_capacity): size_(0),
                                     capacity_(initial_capacity),
                                     data_(new int[capacity_]) { }
    ...
};
```

```
class Vector {
    size_t size_ ;
    size_t capacity_ ;
    int* data_ ;
public:

    Vector(size_t initial_capacity): size_(0),
                                     capacity_(initial_capacity),
                                     data_(new int[capacity_])    { }
    ...
};
```

Member initializer list

```cpp
class Vector {
    size_t size_ ;
    size_t capacity_ ;
    int* data_ ;
public:
    Vector(size_t initial_capacity): size_(0),
                                     capacity_(initial_capacity),
                                     data_(new int[capacity_]) { }
    ...
};
```

Member initializer list

If a field listed in member initializer list, it is initialized with it before body of the constructor (and only once, without double init).

```
class Vector {
    size_t size_ ;
    size_t capacity_ ;
    int* data_ ;

public:
    Vector(size_t initial_capacity): size_(0),
                                     capacity_(initial_capacity),
                                     data_(new int[capacity_]) { }
    ...
};
```

```cpp
class Vector {
    size_t size_ ;
    size_t capacity_ ;
    int* data_ ;
    Point p;
public:
    Vector(size_t initial_capacity): size_(0),
                                     capacity_(initial_capacity),
                                     data_(new int[capacity_]),
                                     p(13, 42) { }
    ...
};
```

```cpp
struct Point {
    int x; int y;
    Point() {
        this->x = 13;
        this->y = 42;
    }
    Point(int x, int y) {
        this->x = x;
        this->y = y;
    }
};
```

```cpp
class Vector {
    size_t size_ ;
    size_t capacity_ ;
    int* data_ ;
    Point p;
public:
    Vector(size_t initial_capacity): size_(0),
                                     capacity_(initial_capacity),
                                     data_(new int[capacity_]),
                                     p(13, 42) { }

    ...
};
```

```cpp
struct Point {
    int x; int y;

    Point(): x(13), y(42) {
    }

    Point(int x, int y) {
        this->x = x;
        this->y = y;
    }
};
```

```cpp
class Vector {
    size_t size_ ;
    size_t capacity_ ;
    int* data_ ;
    Point p;
public:
    Vector(size_t initial_capacity): size_(0),
                                     capacity_(initial_capacity),
                                     data_(new int[capacity_]),
                                     p(13, 42) { }

    ...
};
```

```cpp
struct Point {
    int x; int y;

    Point(): x(13), y(42) {}
    Point(int x, int y): x(x), x(y) {}
};
```

```cpp
class Vector {
    size_t size_ ;
    size_t capacity_ ;
    int* data_ ;
    Point p;
public:
    Vector(size_t initial_capacity): size_(0),
                                     capacity_(initial_capacity),
                                     data_(new int[capacity_]),
                                     p(13, 42) { }
    ...
};
```

```cpp
class Vector {
    size_t size_ ;
    size_t capacity_ ;
    int* data_ ;
    Point p;
public:
    Vector(size_t initial_capacity): size_(0),
                                     capacity_(initial_capacity),
                                     data_(new int[capacity_]),
                                     p(13, 42) { }

    ...
};
```

```cpp
struct Point {
    int x; int y;

    Point(): x(13), y(42) {}
    Point(int x, int y): x(x), x(y) {}
};
```

Which type of initialization do we use for fields here?

Initialization in C++ (first approximation)

1. Default initialization in C++ ✓
   a. same UB for primitives
   b. default ctors for classes
2. Value initialization ✓ ✗ — ok for primitives, nightmare for classes, avoid if possible
3. Direct initialization ✓ — use (args) or {args}
4. Copy initialization ✓ — implicit call of ctrs here, use explicit to avoid
5. Aggregate initialization ✓ — works for aggregates, use copy initialization for all fields
6. List initialization (direct and copy)

```cpp
class Vector {
    size_t size_ ;
    size_t capacity_ ;
    int* data_ ;
    Point p;
public:
    Vector(size_t initial_capacity): size_(0),
                                     capacity_(initial_capacity),
                                     data_(new int[capacity_]),
                                     p(13, 42) { }
    ...
};
```

```cpp
struct Point {
    int x; int y;

    Point(): x(13), y(42) {}
    Point(int x, int y): x(x), x(y) {}
};
```

Which type of initialization do we use for fields here?
Direct initialization!

```cpp
class Vector {                                             struct Point { int x; int y; };
    size_t size_ ;
    size_t capacity_ ;
    int* data_ ;
    Point p;
public:
    Vector(size_t initial_capacity): size_(0),
                                     capacity_(initial_capacity),
                                     data_(new int[capacity_]),
                                     p{.x=13, .y=42} { }
    ...
};
```

Which type of initialization do we use for fields here?
Direct initialization!

```cpp
class Vector {
    size_t size_ ;
    size_t capacity_ ;
    int* data_ ;
    Point p;
public:
    Vector(size_t initial_capacity): size_(0),
                                    capacity_(initial_capacity),
                                    data_(new int[capacity_]),
                 aggregate init for p p{.x=13, .y=42} { }
    ...
};
```

```cpp
struct Point { int x; int y; };
```

Now it is aggregate

Which type of initialization do we use for fields here?
Direct initialization! Value, aggregate and list initialization are also possible here, but usually we use direct.

```cpp
class Vector {
    size_t size_ ;
    size_t capacity_ ;
    int* data_ ;
public:
    Vector(size_t initial_capacity): size_(0),
                                     capacity_(initial_capacity),
                                     data_(new int[capacity_]) { }
    ...
};
```

```
class Vector {
    size_t size_ ;
    size_t capacity_ ;
    int* data_ ;
public:
    Vector(size_t initial_capacity): size_(0),
                                      capacity_(initial_capacity),
                                      data_(new int[capacity_]) { }
    ...
};
```

what if I'll change the order?

```
class Vector {
    size_t size_ ;
    size_t capacity_ ;
    int* data_ ;
public:
    Vector(size_t initial_capacity): capacity_(initial_capacity),
                                     size_(0),
                                     data_(new int[capacity_]) { }
    ...
};
```

```
class Vector {
    size_t size_ ;
    size_t capacity_ ;
    int* data_ ;
public:
    Vector(size_t initial_capacity): capacity_(initial_capacity),
                                     size_(0),
                                     data_(new int[capacity_]) { }
    ...
};
```

Nothing will change actually: order of initialization is just the order of declaration of fields in a class!

```
class Vector {
    size_t size_ ;
    size_t capacity_ ;
    int* data_ ;
public:
    Vector(size_t initial_capacity): capacity_(initial_capacity),
                                     size_(0),
                                     data_(new int[capacity_]) { }
    ...
};
```

Nothing will change actually: order of initialization is just the order of declaration of fields in a class!

size_ -> capacity_ -> data_

```
class Vector {
    size_t size_ ;
    size_t capacity_ ;
    int* data_ ;
public:
    Vector(size_t initial_capacity): data_(new int[capacity_]),
                                     capacity_(initial_capacity),
                                     size_(0) { }
    ...
};
```

Still works! For good or evil

Nothing will change actually: order of initialization is just the order of declaration of fields in a class!

size_ -> capacity_ -> data_

```
class Vector {
    size_t size_ ;
    size_t capacity_ ;
    int* data_ ;
public:
    Vector(size_t initial_capacity): size_(0),
                                     capacity_(initial_capacity),
                                     data_(new int[capacity_]) { }
    ...
};
```

```cpp
class Vector {
    size_t size_ ;
    size_t capacity_ ;
    int* data_ ;
public:
    Vector(size_t initial_capacity): size_(0),
                                     capacity_(initial_capacity),
                                     data_(new int[capacity_]) { }
    ...
};
```

```cpp
class Vector {
    size_t size_ = 0;
    size_t capacity_ ;
    int* data_ ;
public:
    Vector(size_t initial_capacity): capacity_(initial_capacity),
                                     data_(new int[capacity_]) { }
    ...
};
```

```cpp
class Vector {
    size_t size_ = 0;
    size_t capacity_ ;
    int* data_ ;
public:
    Vector(size_t initial_capacity): capacity_(initial_capacity),
                                     data_(new int[capacity_]) { }
    ...
};
```

default member initializer

```cpp
class Vector {
    size_t size_ = 0;
    size_t capacity_ ;
    int* data_ ;
public:
    Vector(size_t initial_capacity): capacity_(initial_capacity),
                                     data_(new int[capacity_]) { }
    ...
};
```

default member initializer

You can think about it as about implicit prepending of `size_(0)` to any initializer list.

```
class Vector {
    size_t size_ = 0;
    size_t capacity_ ;
    int* data_ ;
public:
    Vector(size_t initial_capacity): // size_(0),
                                     capacity_(initial_capacity),
                                     data_(new int[capacity_]) { }
    ...
};
```

default member initializer

You can think about it as about implicit prepending of size_(0) to any initializer list.

```
class Vector {
    size_t size_ = 0;   ←——— default member initializer
    size_t capacity_ ;
    int* data_ ;
public:
    Vector(size_t initial_capacity): // size_(0),
                                     capacity_(initial_capacity),
                                     data_(new int[capacity_]) { }
    ...
};
```

You can think about it as about implicit prepending of size_(0) to any initializer list (so, it is direct initialization).

```
class Vector {
    size_t size_ = 0;   ◄———— default member initializer
    size_t capacity_ ;
    int* data_ ;
public:
    Vector(size_t initial_capacity): // size_(0),
                                     capacity_(initial_capacity),
                                     data_(new int[capacity_]) { }
    ...
};
```

You can think about it as about implicit prepending of size_(0) to any initializer list (so, it is direct initialization).

If you have both, initializer list will win.

```
class Vector {
    size_t size_ = 0;
    size_t capacity_ ;
    int* data_ ;
public:

    Vector(): Vector(16) { }

    Vector(size_t initial_capacity): capacity_(initial_capacity),
                                     data_(new int[capacity_]) { }
    ...
};
```

```cpp
class Vector {
    size_t size_ = 0;
    size_t capacity_ ;
    int* data_ ;
public:

    Vector(): Vector(16) { }

    Vector(size_t initial_capacity): capacity_(initial_capacity),
                                     data_(new int[capacity_]) { }
    ...
};
```

Delegating constructor

```cpp
class Vector {
    size_t size_ = 0;
    size_t capacity_ ;
    int* data_ ;
public:

    Vector(): Vector(16) { }

    Vector(size_t initial_capacity): capacity_(initial_capacity),
                                     data_(new int[capacity_]) { }
    ...
};
```

Delegating constructor

Member initializer lists can't be used with delegating constructors

Member initializer lists can’t be used with delegating constructors

Delegating constructor

```
class Vector {
    size_t size_ = 0;
    size_t capacity_ ;
    int* data_ ;
public:

    Vector(): Vector(16), size_(42) { } // compilation error

    Vector(size_t initial_capacity): capacity_(initial_capacity),
                                     data_(new int[capacity_]) { }
    ...
};
```

```cpp
class Vector {
    size_t size_ = 0;
    size_t capacity_ ;
    int* data_ ;
public:

    Vector(size_t initial_capacity = 16): capacity_(initial_capacity),
                                          data_(new int[capacity_]) { }
    ...
};
```

Default arguments + member initializer lists can be a good alternative to delegating constructors.

```
class Vector {
    size_t size_ = 0;
    size_t capacity_ ;
    int* data_ ;
public:

    Vector(size_t initial_capacity = 16): capacity_(initial_capacity),
                                          data_(new int[capacity_]) { }
    ...
};
```

```cpp
class Vector {
    size_t size_ = 0;
    size_t capacity_ ;
    int* data_ ;
public:

    Vector(size_t initial_capacity = 16): capacity_(initial_capacity),
                                          data_(new int[capacity_]) { }
    ...
};
```

Default arguments + member initializer lists can be a good alternative to delegating constructors.

Works as default constructor when argument is not specified. Adding default constructor here will cause compilation error.

Member initializer lists

- Needed to guarantee that fields are initialized before the body of constructor,

Member initializer lists

- Needed to guarantee that fields are initialized before the body of constructor,

- Also to initialize const fields,

Member initializer lists

- Needed to guarantee that fields are initialized before the body of constructor,

- Also to initialize const fields,

- Delicate moments:
  - Order depends on the class declaration,
  - Different type of initialization in lists,
  - Default member initialization as syntax sugar,
  - Doesn’t work with delegating ctrs.

Member initializer lists

- Needed to guarantee that fields are initialized before the body of constructor,
- Also to initialize const fields,
- Delicate moments:
  - Order depends on the class declaration,
  - Different type of initialization in lists,
  - Default member initialization as syntax sugar,
  - Doesn't work with delegating ctrs.

Member initializer lists

- Needed to guarantee that fields are initialized before the body of constructor,

- Also to initialize const fields,

- Delicate moments:
  - Order depends on the class declaration,
  - Different type of initialization in lists,
  - Default member initialization as syntax sugar,
  - Doesn’t work with delegating ctrs.

In general, it is wonderful feature (needed for language with flat fields and uninitialized values). Please use it!

Copy constructors

"can I copy your homework?"

"yeah just change it up a bit so it doesn't look obvious you copied"

"ok"

```
struct Point {
   int x;
   int y;
};

void foo(Point lp) {
   printf("%d %d\n", lp.x, lp.y);
}

int main() {
   Point p{13, 42};
   foo(p);
   return 0;
}
```

lp is initialized with
copy initialization
from the argument

```cpp
struct Point {
   int x;
   int y;
};

void foo(Point lp) {
   printf("%d %d\n", lp.x, lp.y);
}

int main() {
   Point p{13, 42};
   foo(p);
   return 0;
}
```

lp is initialized with copy initialization from the argument

Previously we've said that all raw memory of all fields of p was just copied into lp (this is true for C).

But is it always correct behavior?

```cpp
class Vector {
    size_t size_ = 0;
    size_t capacity_ ;
    int* data_ ;
public:
    Vector(size_t ic = 16):
            capacity_(ic),
            data_(new int[capacity_]) {}

    int& at(size_t idx) { return data_[idx]; }
    ~Vector() { delete[] data_; }
};
```

```cpp
class Vector {
    size_t size_ = 0;
    size_t capacity_ ;
    int* data_ ;
public:
    Vector(size_t ic = 16):
            capacity_(ic),
            data_(new int[capacity_]) {}

    int& at(size_t idx) { return data_[idx]; }
    ~Vector() { delete[] data_; }
};

Vector v{8};
v.push(42);
foo(v);
cout << v.at(0);
```

```
class Vector {
    size_t size_ = 0;
    size_t capacity_ ;
    int* data_ ;
public:
    Vector(size_t ic = 16):
            capacity_(ic),
            data_(new int[capacity_]) {}

    int& at(size_t idx) { return data_[idx]; }
    ~Vector() { delete[] data_; }
};

Vector v{8};
v.push(42);
foo(v);
cout << v.at(0);
```

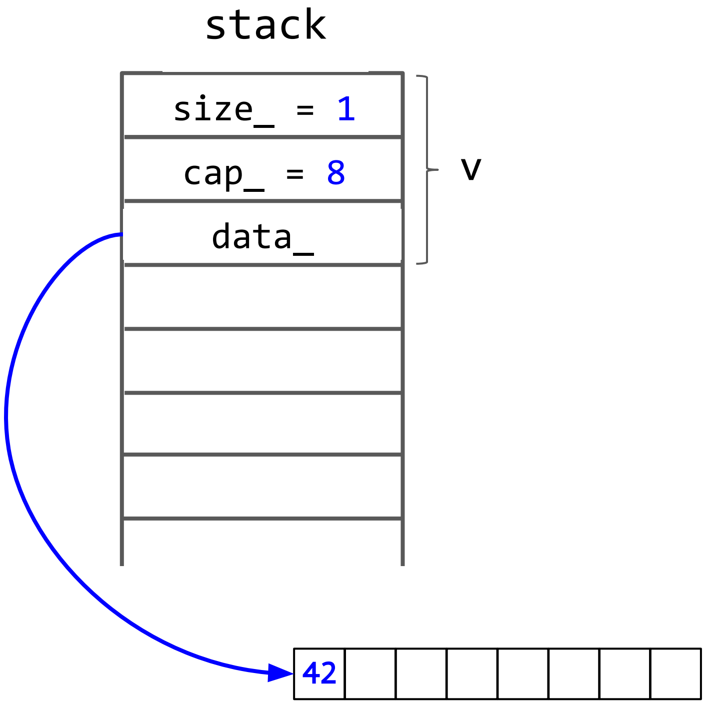
What will be printed?

```
void foo(Vector lv) {
    lv.at(0) = 78;
}
```

```cpp
class Vector {
    size_t size_ = 0;
    size_t capacity_ ;
    int* data_ ;
public:
    Vector(size_t ic = 16):
            capacity_(ic),
            data_(new int[capacity_]) {}

    int& at(size_t idx) { return data_[idx]; }
    ~Vector() { delete[] data_; }
};

Vector v{8};
v.push(42);
foo(v);
cout << v.at(0);
```

What will be printed?

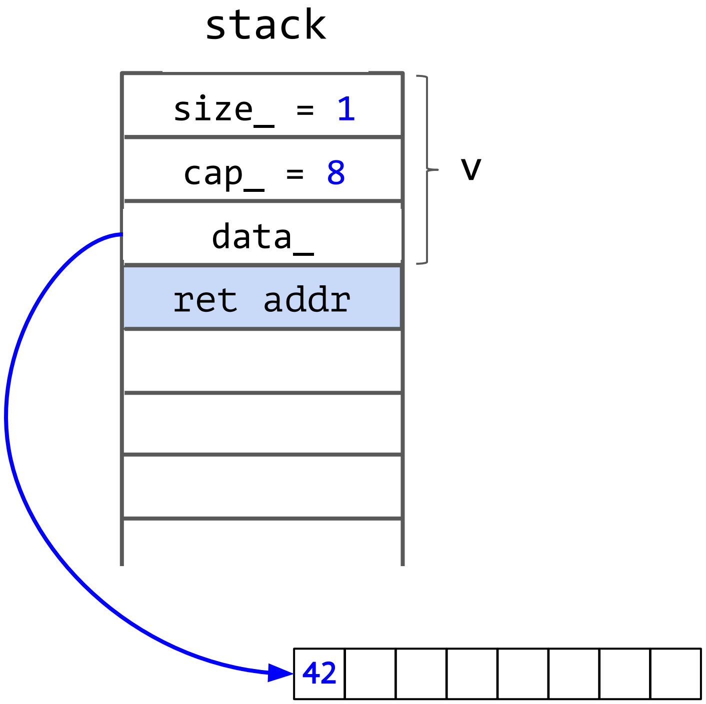
```cpp
void foo(Vector lv) {
    lv.at(0) = 78;
}
```



```
class Vector {
    size_t size_ = 0;
    size_t capacity_ ;
    int* data_ ;
public:
    Vector(size_t ic = 16):
            capacity_(ic),
            data_(new int[capacity_]) {}

    int& at(size_t idx) { return data_[idx]; }
    ~Vector() { delete[] data_; }
};

Vector v{8};
v.push(42);
foo(v);
cout << v.at(0);
```

What will be printed?

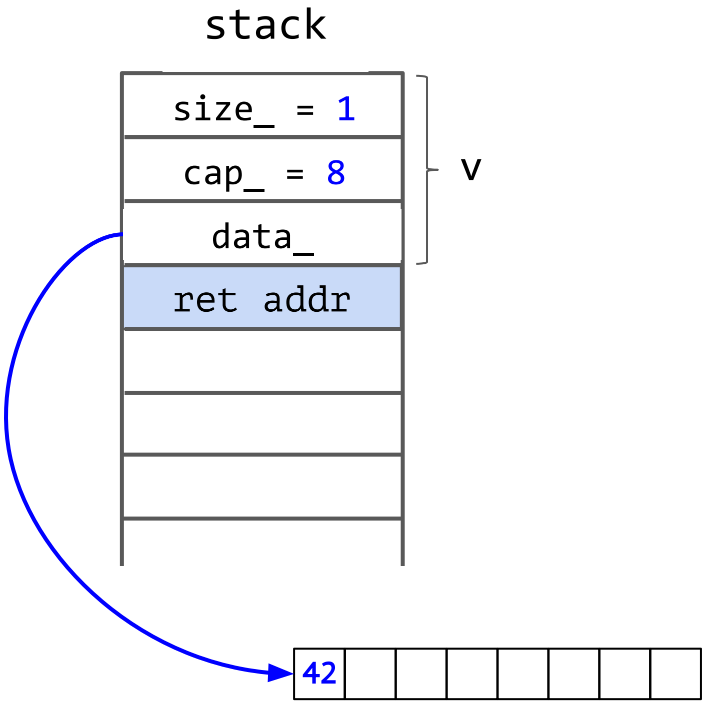
```
void foo(Vector lv) {
   lv.at(0) = 78;
}
```



```cpp
class Vector {
    size_t size_ = 0;
    size_t capacity_ ;
    int* data_ ;
public:
    Vector(size_t ic = 16):
            capacity_(ic),
            data_(new int[capacity_]) {}

    int& at(size_t idx) { return data_[idx]; }
    ~Vector() { delete[] data_; }
};

Vector v{8};
v.push(42);
foo(v);
cout << v.at(0);
```

What will be printed?

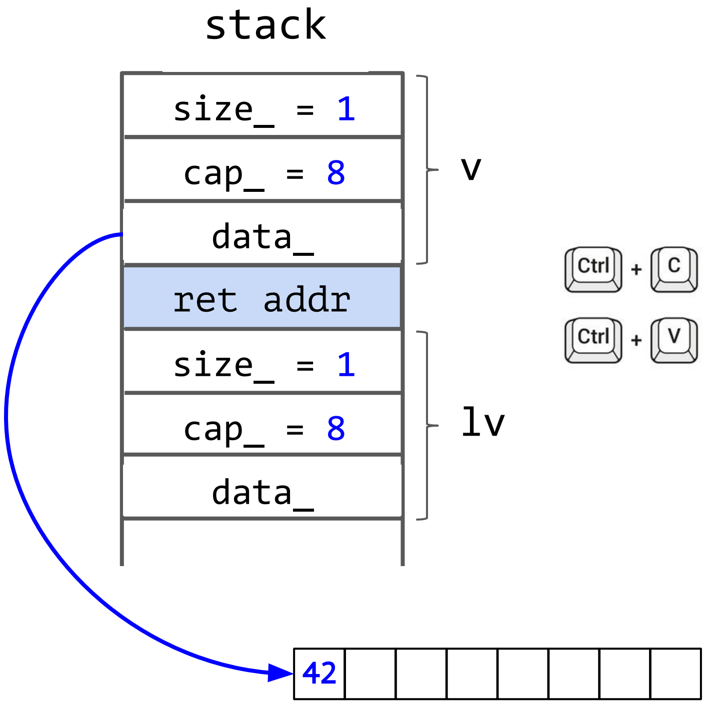
```cpp
void foo(Vector lv) {
    lv.at(0) = 78;
}
```



```cpp
class Vector {
    size_t size_ = 0;
    size_t capacity_ ;
    int* data_ ;
public:
    Vector(size_t ic = 16):
            capacity_(ic),
            data_(new int[capacity_]) {}

    int& at(size_t idx) { return data_[idx]; }
    ~Vector() { delete[] data_; }
};

Vector v{8};
v.push(42);
foo(v);
cout << v.at(0);
```

What will be printed?

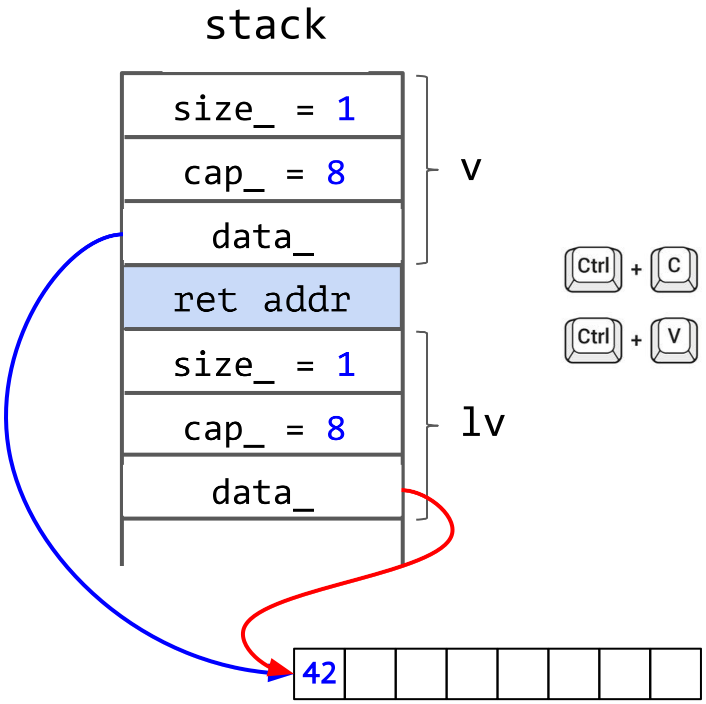
```cpp
void foo(Vector lv) {
    lv.at(0) = 78;
}
```



```cpp
class Vector {
    size_t size_ = 0;
    size_t capacity_ ;
    int* data_ ;
public:
    Vector(size_t ic = 16):
            capacity_(ic),
            data_(new int[capacity_]) {}

    int& at(size_t idx) { return data_[idx]; }
    ~Vector() { delete[] data_; }
};

Vector v{8};
v.push(42);
foo(v);
cout << v.at(0);
```

What will be printed?

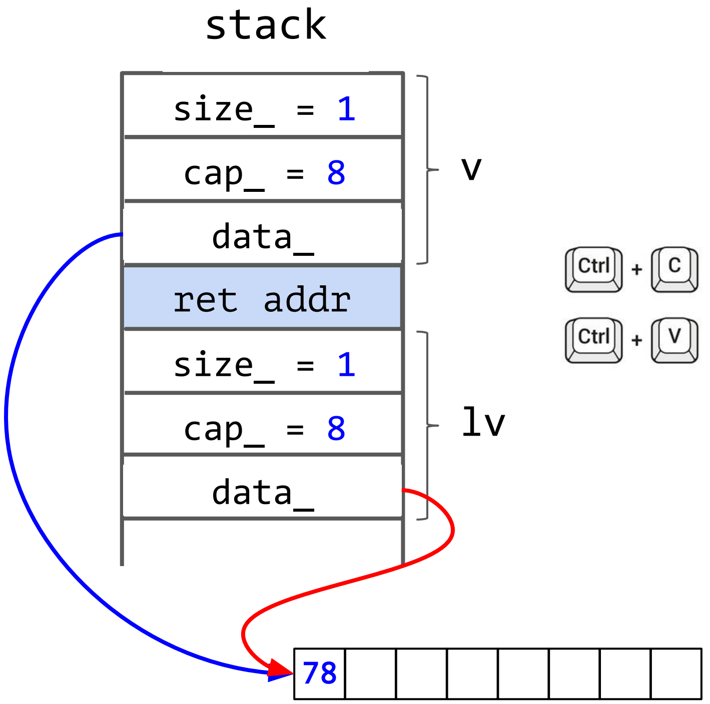
```cpp
void foo(Vector lv) {
   lv.at(0) = 78;
}
```



```
class Vector {
    size_t size_ = 0;
    size_t capacity_ ;
    int* data_ ;
public:
    Vector(size_t ic = 16):
            capacity_(ic),
            data_(new int[capacity_]) {}

    int& at(size_t idx) { return data_[idx]; }
    ~Vector() { delete[] data_; }
};

Vector v{8};
v.push(42);
foo(v);
cout << v.at(0);
```

What will be printed?

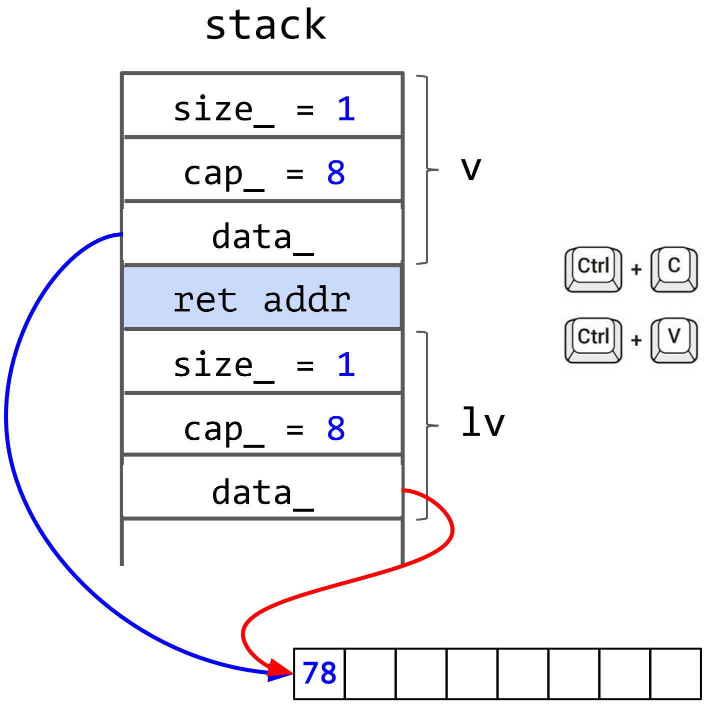
```
void foo(Vector lv) {
    lv.at(0) = 78;
}
```



```cpp
class Vector {
    size_t size_ = 0;
    size_t capacity_ ;
    int* data_ ;
public:
    Vector(size_t ic = 16):
            capacity_(ic),
            data_(new int[capacity_]) {}

    int& at(size_t idx) { return data_[idx]; }
    ~Vector() { delete[] data_; }
};

Vector v{8};
v.push(42);
foo(v);
cout << v.at(0);
```

What will be printed?

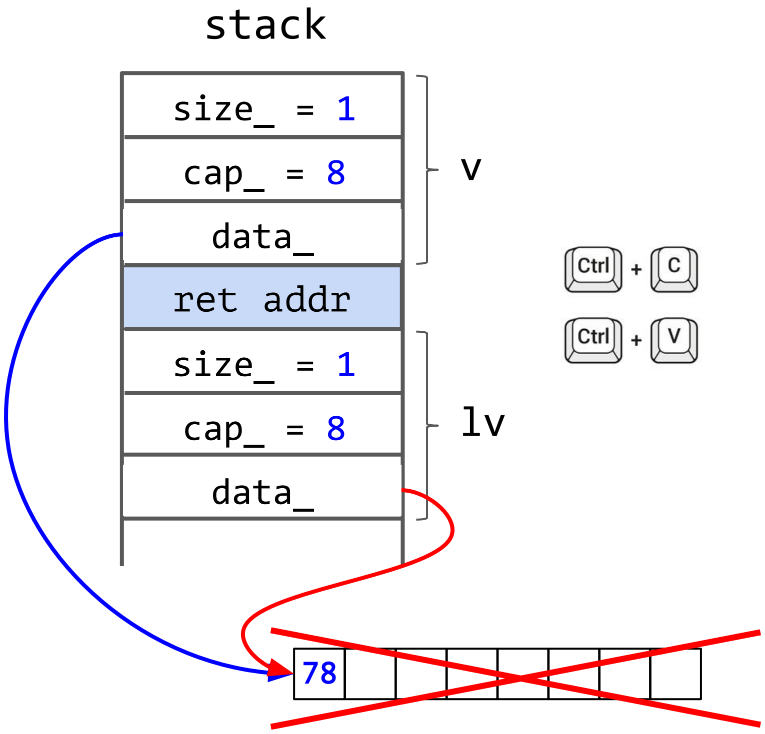
```cpp
void foo(Vector lv) {
    lv.at(0) = 78;
}
```



```cpp
class Vector {
    size_t size_ = 0;
    size_t capacity_ ;
    int* data_ ;
public:
    Vector(size_t ic = 16):
            capacity_(ic),
            data_(new int[capacity_]) {}

    int& at(size_t idx) { return data_[idx]; }
    ~Vector() { delete[] data_; }
};

Vector v{8};
v.push(42);
foo(v);
cout << v.at(0);
```

What will be printed?

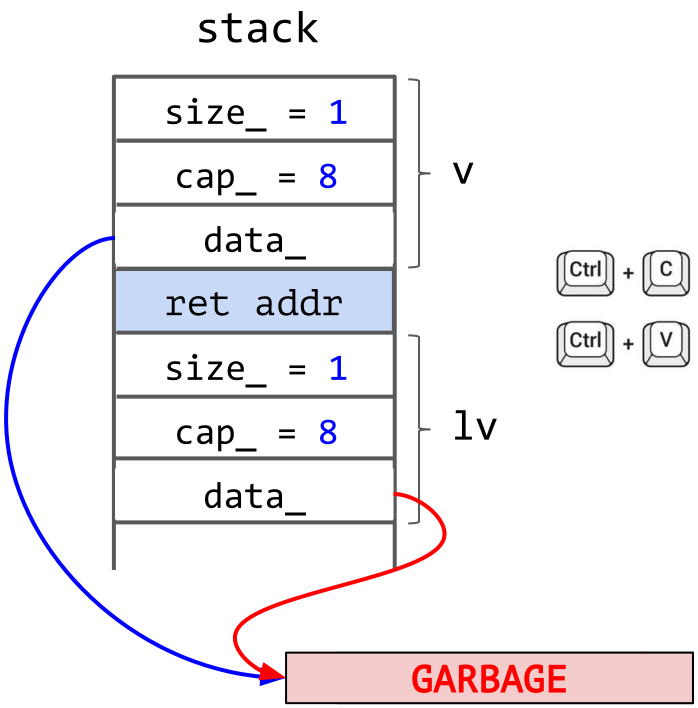
```cpp
void foo(Vector lv) {
    lv.at(0) = 78;
}
```



```cpp
class Vector {
    size_t size_ = 0;
    size_t capacity_ ;
    int* data_ ;
public:
    Vector(size_t ic = 16):
            capacity_(ic),
            data_(new int[capacity_]) {}

    int& at(size_t idx) { return data_[idx]; }
    ~Vector() { delete[] data_; }
};

Vector v{8};
v.push(42);
foo(v);
cout << v.at(0);
```

What will be printed?

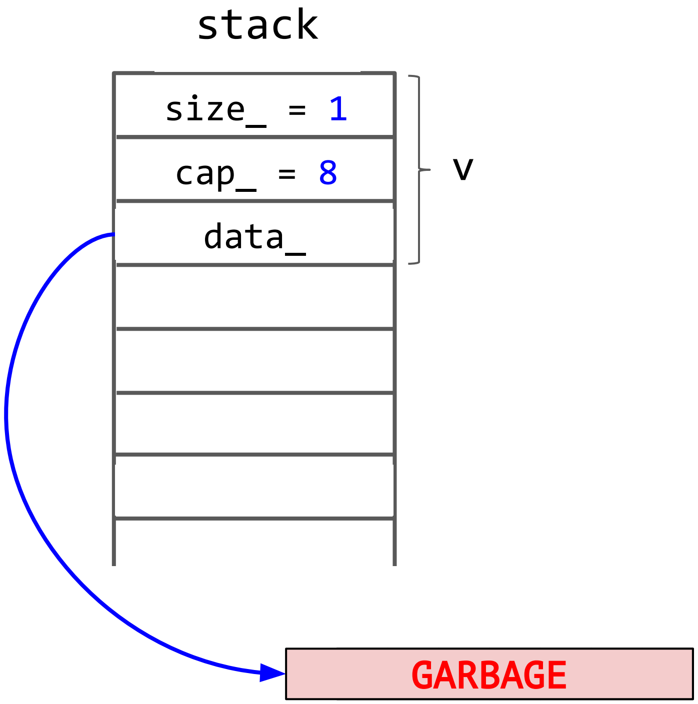
```cpp
void foo(Vector lv) {
    lv.at(0) = 78;
}
```



```cpp
class Vector {
    size_t size_ = 0;
    size_t capacity_ ;
    int* data_ ;
public:
    Vector(size_t ic = 16):
            capacity_(ic),
            data_(new int[capacity_]) {}

    int& at(size_t idx) { return data_[idx]; }
    ~Vector() { delete[] data_; }
};

Vector v{8};
v.push(42);
foo(v);
cout << v.at(0);
```

What will be printed?

```cpp
void foo(Vector lv) {
    lv.at(0) = 78;
}
```

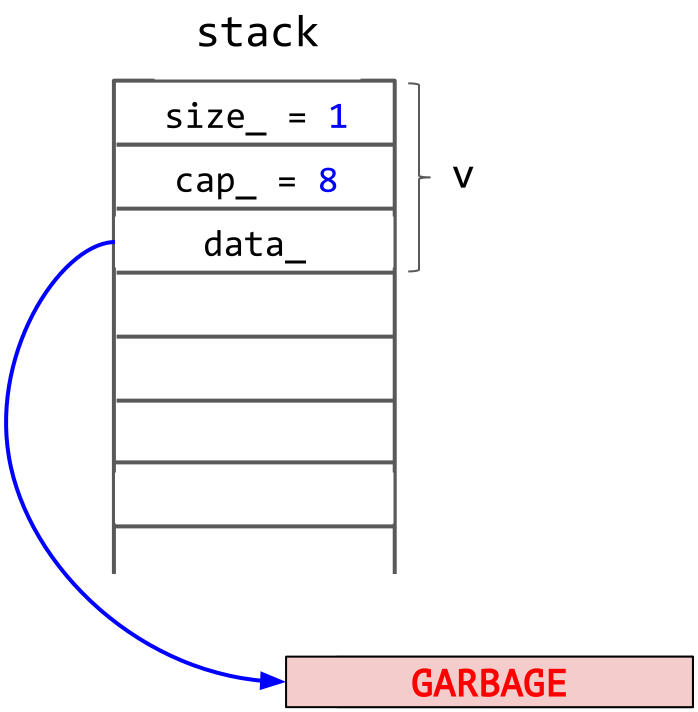


```
class Vector {
    size_t size_ = 0;
    size_t capacity_ ;
    int* data_ ;
public:
    Vector(size_t ic = 16):
            capacity_(ic),
            data_(new int[capacity_]) {}

    int& at(size_t idx) { return data_[idx]; }
    ~Vector() { delete[] data_; }
};

Vector v{8};
v.push(42);
foo(v);
cout << v.at(0);
```

What will be printed?

UB happens!

```
void foo(Vector lv) {
    lv.at(0) = 78;
}
```



```cpp
class Vector {
    size_t size_ = 0;
    size_t capacity_ ;
    int* data_ ;
public:
    Vector(size_t ic = 16):
            capacity_(ic),
            data_(new int[capacity_]) {}

    int& at(size_t idx) { return data_[idx]; }
    ~Vector() { delete[] data_; }
};

Vector v{8};
v.push(42);
foo(v);
cout << v.at(0);
```

What will be printed?

UB happens!

```cpp
void foo(Vector lv) {
    lv.at(0) = 78;
}
```

So, we need some way to customize logic during copying.

Here comes copy constructor.

```
class Vector {
    size_t size_ = 0;
    size_t capacity_ ;
    int* data_ ;
public:
    Vector(size_t ic = 16):
            capacity_(ic),
            data_(new int[capacity_]) {}

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

```cpp
class Vector {
    size_t size_ = 0;
    size_t capacity_ ;
    int* data_ ;
public:
    Vector(size_t ic = 16):
            capacity_(ic),
            data_(new int[capacity_]) {}

    Vector(const Vector& other) ... {
        ...
    }

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

```
class Vector {
    size_t size_ = 0;
    size_t capacity_ ;
    int* data_ ;
public:
    Vector(size_t ic = 16):
            capacity_(ic),
            data_(new int[capacity_]) {}

    Vector(const Vector& other): size_(other.size_), capacity_(other.capacity_) {
        ...
    }

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

```
class Vector {
    size_t size_ = 0;
    size_t capacity_ ;
    int* data_ ;
public:
    Vector(size_t ic = 16):
            capacity_(ic),
            data_(new int[capacity_]) {}

    Vector(const Vector& other): size_(other.size_), capacity_(other.capacity_) {
       data_ = new int[capacity_];
         for (size_t i = 0; i < size_; i++) {
             data_[i] = other.data_[i];
         }
    }
    ~Vector() { delete[] data_; }
};
```

```cpp
class Vector {
    size_t size_ = 0;
    size_t capacity_ ;
    int* data_ ;
public:
    Vector(size_t ic = 16):
            capacity_(ic),
            data_(new int[capacity_]) {}

    Vector(const Vector& other): size_(other.size_), capacity_(other.capacity_) {
       data_ = new int[capacity_];
         for (size_t i = 0; i < size_; i++) {
              data_[i] = other.data_[i];
         }
    }
    ~Vector() { delete[] data_; }
};
```

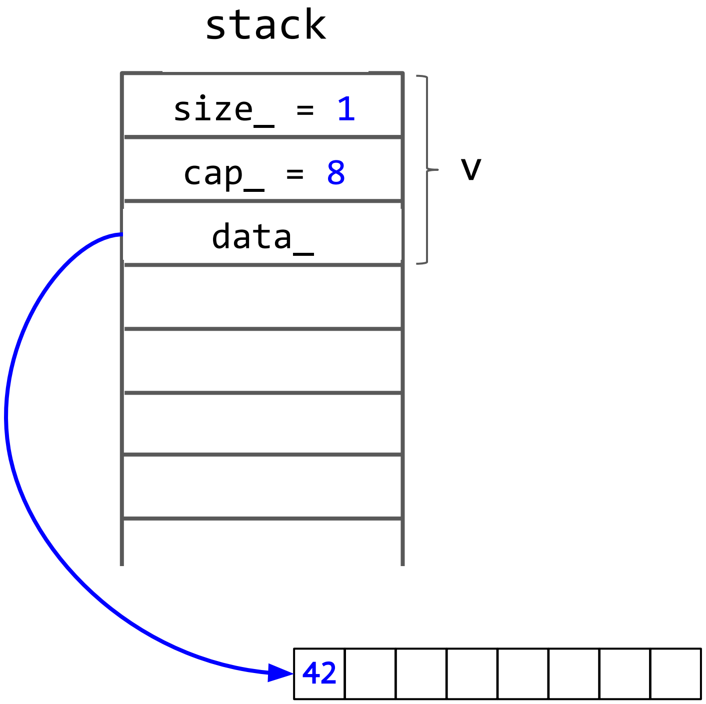
This constructor is called copy constructor.

It is used when copy initialization from other Vector is invoked.

```cpp
class Vector {
    size_t size_ = 0;
    size_t capacity_ ;
    int* data_ ;
public:
    Vector(size_t ic = 16):
            capacity_(ic),
            data_(new int[capacity_]) {}

    int& at(size_t idx) { return data_[idx]; }
    ~Vector() { delete[] data_; }
};

Vector v{8};
v.push(42);
foo(v);
cout << v.at(0);
```

What will be printed?

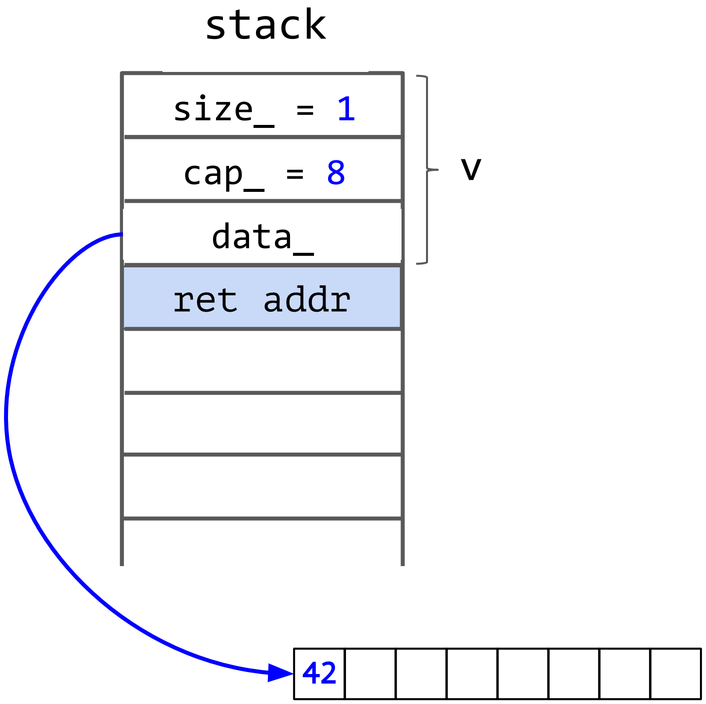
```cpp
void foo(Vector lv) {
    lv.at(0) = 78;
}
```



```
class Vector {
    size_t size_ = 0;
    size_t capacity_ ;
    int* data_ ;
public:
    Vector(size_t ic = 16):
            capacity_(ic),
            data_(new int[capacity_]) {}
    int& at(size_t idx) { return data_[idx]; }
    Vector(const Vector& other) { … }
    ~Vector() { delete[] data_; }
};

Vector v{8};
v.push(42);
foo(v);
cout << v.at(0);
```

What will be printed?

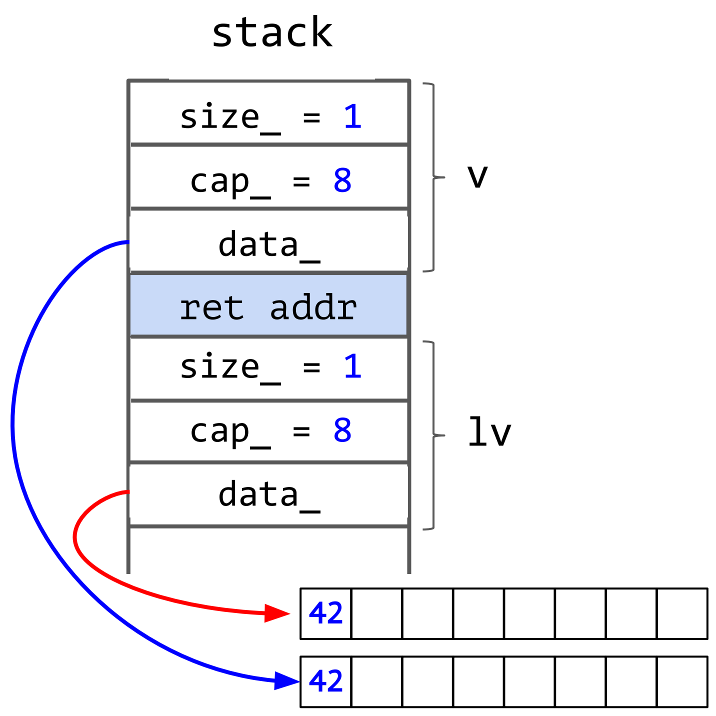
```
void foo(Vector lv) {
    lv.at(0) = 78;
}
```



```cpp
class Vector {
    size_t size_ = 0;
    size_t capacity_ ;
    int* data_ ;
public:
    Vector(size_t ic = 16):
            capacity_(ic),
            data_(new int[capacity_]) {}
    int& at(size_t idx) { return data_[idx]; }
    Vector(const Vector& other) { … }
    ~Vector() { delete[] data_; }
};

Vector v{8};
v.push(42);
foo(v);
cout << v.at(0);
```

What will be printed?

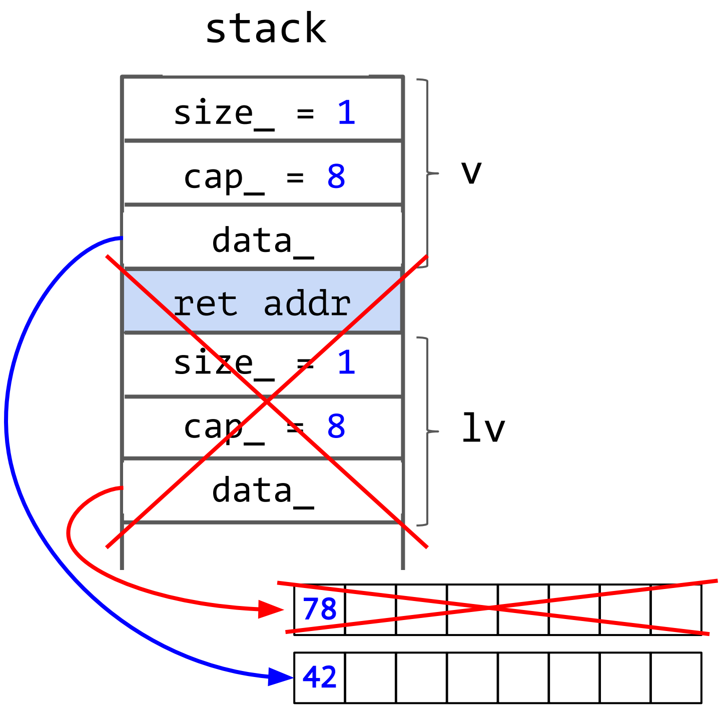
```cpp
void foo(Vector lv) {
    lv.at(0) = 78;
}
```



```cpp
class Vector {
    size_t size_ = 0;
    size_t capacity_ ;
    int* data_ ;
public:
    Vector(size_t ic = 16):
            capacity_(ic),
            data_(new int[capacity_]) {}
    int& at(size_t idx) { return data_[idx]; }
    Vector(const Vector& other) { … }
    ~Vector() { delete[] data_; }
};

Vector v{8};
v.push(42);
foo(v);
cout << v.at(0);
```

What will be printed?

```cpp
void foo(Vector lv) {
    lv.at(0) = 78;
}
```

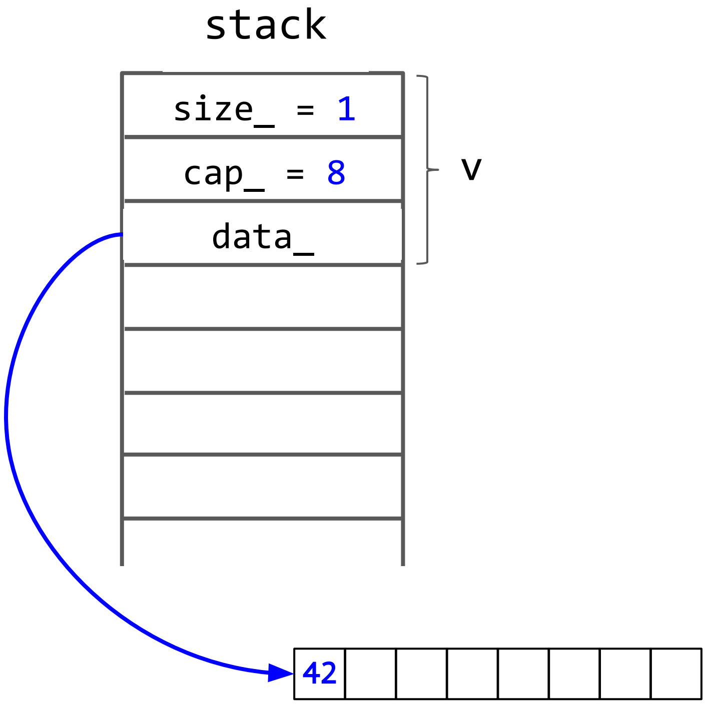


```
class Vector {
    size_t size_ = 0;
    size_t capacity_ ;
    int* data_ ;
public:
    Vector(size_t ic = 16):
            capacity_(ic),
            data_(new int[capacity_]) {}
    int& at(size_t idx) { return data_[idx]; }
    Vector(const Vector& other) { … }
    ~Vector() { delete[] data_; }
};

Vector v{8};
v.push(42);
foo(v);
cout << v.at(0);
```

What will be printed?

42

```
void foo(Vector lv) {
   lv.at(0) = 78;
}
```



```cpp
class Vector {
    size_t size_ = 0;
    size_t capacity_ ;
    int* data_ ;
public:
    Vector(size_t ic = 16):
            capacity_(ic),
            data_(new int[capacity_]) {}
    int& at(size_t idx) { return data_[idx]; }
    Vector(const Vector& other) { … }
    ~Vector() { delete[] data_; }
};

Vector v{8};
v.push(42);
foo(v);
cout << v.at(0);
```

```cpp
void foo(Vector lv) {
    lv.at(0) = 78;
}
```

copy ctr
is called

```cpp
class Vector {
    size_t size_ = 0;
    size_t capacity_ ;
    int* data_ ;
public:
    Vector(size_t ic = 16):
            capacity_(ic),
            data_(new int[capacity_]) {}
    int& at(size_t idx) { return data_[idx]; }
    Vector(const Vector& other) { … }
    ~Vector() { delete[] data_; }
};

Vector v{8};
Vector v2(v);
Vector v3 = v;
```

copy ctr
is called

```cpp
void foo(Vector lv) {
   lv.at(0) = 78;
}
```

copy ctr
is called

```
class Vector {
    size_t size_ = 0;
    size_t capacity_ ;
    int* data_ ;
public:
    Vector(size_t ic = 16):
            capacity_(ic),
            data_(new int[capacity_]) {}
    int& at(size_t idx) { return data_[idx]; }
    Vector(const Vector& other) { … }
    ~Vector() { delete[] data_; }
};

Vector v{8};
Vector v2(v);
Vector v3 = v;
```

copy ctr
is called

```
void foo(Vector lv) {
   lv.at(0) = 78;
}
```

copy ctr
is called

When else it will be called?

```cpp
class Vector {
    size_t size_ = 0;
    size_t capacity_ ;
    int* data_ ;
public:
    Vector(size_t ic = 16):
            capacity_(ic),
            data_(new int[capacity_]) {}
    int& at(size_t idx) { return data_[idx]; }
    Vector(const Vector& other) { … }
    ~Vector() { delete[] data_; }
};

Vector v{8};
Vector v2(v);       copy ctr
Vector v3 = v;      is called
```

```cpp
void foo(Vector lv) {     copy ctr
    lv.at(0) = 78;        is called
}
```

When else it will be called?

Aggregate initialization, for copy initialization of fields!

```cpp
class Vector {
    size_t size_ = 0;
    size_t capacity_ ;
    int* data_ ;
public:
    Vector(size_t ic = 16):
            capacity_(ic),
            data_(new int[capacity_]) {}
    int& at(size_t idx) { return data_[idx]; }
    Vector(const Vector& other) { … }
    ~Vector() { delete[] data_; }
};
```

```cpp
Vector v{8};
Vector v2(v);
Vector v3 = v;
```

copy ctr
is called

```cpp
void foo(Vector lv) {
    lv.at(0) = 78;
}
```

copy ctr
is called

When else it will be called?

Aggregate initialization, for copy initialization of fields!

```cpp
struct PairOfVectors {
    Vector v1;
    Vector v2;
};
```

```
class Vector {
    size_t size_ = 0;
    size_t capacity_ ;
    int* data_ ;
public:
    Vector(size_t ic = 16):
            capacity_(ic),
            data_(new int[capacity_]) {}
    int& at(size_t idx) { return data_[idx]; }
    Vector(const Vector& other) { … }
    ~Vector() { delete[] data_; }
};


Vector v{8};
Vector v2(v);
Vector v3 = v;
```

copy ctr
is called

```
void foo(Vector lv) {
   lv.at(0) = 78;
}
```

copy ctr
is called

When else it will be called?

Aggregate initialization, for copy initialization of fields!

```
struct PairOfVectors {
   Vector v1;
   Vector v2;
};

Vector v1{8};
Vector v2{12};

PairOfVectors pv{v1, v2};
```

```cpp
class Vector {
    size_t size_ = 0;
    size_t capacity_ ;
    int* data_ ;
public:
    Vector(size_t ic = 16):
            capacity_(ic),
            data_(new int[capacity_]) {}
    int& at(size_t idx) { return data_[idx]; }
    Vector(const Vector& other) { … }
    ~Vector() { delete[] data_; }
};


Vector v{8};
Vector v2(v);
Vector v3 = v;
```

copy ctr
is called

```cpp
void foo(Vector lv) {
    lv.at(0) = 78;
}
```

copy ctr
is called

When else it will be called?

Aggregate initialization, for copy initialization of fields!

```cpp
struct PairOfVectors {
    Vector v1;
    Vector v2;
};

Vector v1{8};
Vector v2{12};

PairOfVectors pv{v1, v2};
```

copy ctr
is called
twice.

```cpp
class Vector {
    size_t size_ = 0;
    size_t capacity_ ;
    int* data_ ;
public:
    Vector(size_t ic = 16):
            capacity_(ic),
            data_(new int[capacity_]) {}
    int& at(size_t idx) { return data_[idx]; }
    Vector(const Vector& other) { … }
    ~Vector() { delete[] data_; }
};

Vector v{8};
Vector v2(v);
Vector v3 = v;
```

copy ctr
is called

```cpp
void foo(Vector lv) {
   lv.at(0) = 78;
}
```

copy ctr
is called

When else it will be called?

```cpp
class Vector {
    size_t size_ = 0;
    size_t capacity_ ;
    int* data_ ;
public:
    Vector(size_t ic = 16):
            capacity_(ic),
            data_(new int[capacity_]) {}
    int& at(size_t idx) { return data_[idx]; }
    Vector(const Vector& other) { … }
    ~Vector() { delete[] data_; }
};

Vector v{8};
Vector v2(v);
Vector v3 = v;
```

copy ctr
is called

```cpp
void foo(Vector lv) {
    lv.at(0) = 78;
}
```

copy ctr
is called

When else it will be called?

Return values?

```cpp
class Vector {
    size_t size_ = 0;
    size_t capacity_ ;
    int* data_ ;
public:
    Vector(size_t ic = 16):
            capacity_(ic),
            data_(new int[capacity_]) {}
    int& at(size_t idx) { return data_[idx]; }
    Vector(const Vector& other) { … }
    ~Vector() { delete[] data_; }
};

Vector v{8};
Vector v2(v);
Vector v3 = v;
```

copy ctr
is called

```cpp
void foo(Vector lv) {
    lv.at(0) = 78;
}
```

copy ctr
is called

When else it will be called?

Return values?

```cpp
class Vector {
    size_t size_ = 0;
    size_t cap_ ;
    int* data_ ;
public:
    ...
    Vector(const Vector& other):
         size_(other.size_), cap_(other.cap_) {

        cout << "vector copied" << endl;
        data_ = new int[cap_];
        for (size_t i = 0; i < size_; i++) {
            data_[i] = other.data_[i];
        }
    }
    ...
};
```

```cpp
class Vector {
    size_t size_ = 0;
    size_t cap_ ;
    int* data_ ;
public:
    ...
    Vector(const Vector& other):
         size_(other.size_), cap_(other.cap_) {

        cout << "vector copied" << endl;
        data_ = new int[cap_];
        for (size_t i = 0; i < size_; i++) {
            data_[i] = other.data_[i];
        }
    }
    ...
};
```

```cpp
Vector bar() {
    Vector v{8};
    for (size_t i = 0; i < 8; i++) {
        v.push(i);
    }
    return v;
}
```

What will be printed?

```cpp
cout << bar().at(0) << endl;
```

```cpp
class Vector {
    size_t size_ = 0;
    size_t cap_ ;
    int* data_ ;
public:
    ...
    Vector(const Vector& other):
         size_(other.size_), cap_(other.cap_) {

        cout << "vector copied" << endl;
        data_ = new int[cap_];
        for (size_t i = 0; i < size_; i++) {
            data_[i] = other.data_[i];
        }
    }
    ...
};
```

```cpp
Vector bar() {
   Vector v{8};
   for (size_t i = 0; i < 8; i++) {
       v.push(i);
   }
   return v;
}
```

What will be printed?

```cpp
cout << bar().at(0) << endl;
```

Expectation:

"vector copied"

0

```cpp
class Vector {
    size_t size_ = 0;
    size_t cap_ ;
    int* data_ ;
public:
    ...
    Vector(const Vector& other):
        size_(other.size_), cap_(other.cap_) {

        cout << "vector copied" << endl;
        data_ = new int[cap_];
        for (size_t i = 0; i < size_; i++) {
            data_[i] = other.data_[i];
        }
    }
    ...
};
```

```cpp
Vector bar() {
    Vector v{8};
    for (size_t i = 0; i < 8; i++) {
        v.push(i);
    }
    return v;
}
```

What will be printed?

```cpp
cout << bar().at(0) << endl;
```

Expectation:

```
"vector copied"
0
```

Reality (gcc 13.2.0):

```
0
```

```
class Vector {
    size_t size_ = 0;
    size_t cap_ ;
    int* data_ ;
public:
    ...
    Vector(const Vector& other):
         size_(other.size_), cap_(other.cap_) {

        cout << "vector copied" << endl;
        data_ = new int[cap_];
        for (size_t i = 0; i < size_; i++) {
            data_[i] = other.data_[i];
        }
    }
    ...
};
```

```
Vector bar() {
    Vector v{8};
    for (size_t i = 0; i < 8; i++) {
        v.push(i);
    }
    return v;
}
```

What will be printed?

```
cout << bar().at(0) << endl;
```

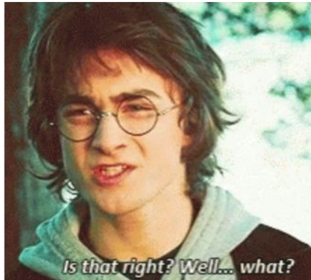
Expectation:

```
"vector copied"
0
```

Reality (gcc 13.2.0):

```
0
```

Where is my copying??



```
class Vector {
    size_t size_ = 0;
    size_t cap_ ;
    int* data_ ;
public:
    ...
    Vector(const Vector& other):
         size_(other.size_), cap_(other.cap_) {

        cout << "vector copied" << endl;
        data_ = new int[cap_];
        for (size_t i = 0; i < size_; i++) {
            data_[i] = other.data_[i];
        }
    }
    ...
};
```

```
__attribute__((noinline)) Vector bar() {
    Vector v{8};
    for (size_t i = 0; i < 8; i++) {
        v.push(i);
    }
    return v;
}
```

What will be printed?

```
cout << bar().at(0) << endl;
```

Expectation:

```
"vector copied"
0
```

Reality (gcc 13.2.0):

```
0
```

Noinline doesn't help, result is the same.

```
__attribute__((noinline)) Vector bar() {
   Vector v{8};
   for (size_t i = 0; i < 8; i++) {
       v.push(i);
   }
   return v;
}

cout << bar().at(0) << endl;
```

Expectation:

```
"vector copied"
0
```

Reality (gcc 13.2.0):

```
0
```

NRVO - **N**amed
Return
Value
Optimization

```
__attribute__((noinline)) Vector bar() {
    Vector v{8};
    for (size_t i = 0; i < 8; i++) {
        v.push(i);
    }
    return v;
}

cout << bar().at(0) << endl;
```

Expectation:

"vector copied"

0

Reality (gcc 13.2.0):

0

NRVO - Named
Return
Value
Optimization

If compiler can prove that return value was just a local object in the scope of a method, it can make an optimization:

```
__attribute__((noinline)) Vector bar() {
   Vector v{8};
   for (size_t i = 0; i < 8; i++) {
       v.push(i);
   }
   return v;
}

cout << bar().at(0) << endl;
```

Expectation:

"vector copied"

0

Reality (gcc 13.2.0):

0

NRVO - Named
Return
Value
Optimization

If compiler can prove that return value was just a local object in the scope of a method, it can make an optimization: allocate memory for this object outside of the method and work with it in bar.

```cpp
__attribute__((noinline)) Vector bar() {
    Vector v{8};
    for (size_t i = 0; i < 8; i++) {
        v.push(i);
    }
    return v;
}

cout << bar().at(0) << endl;
```

Expectation:

```
"vector copied"
0
```

Reality (gcc 13.2.0):

```
0
```

NRVO - Named
Return
Value
Optimization

If compiler can prove that return value was just a local object in the scope of a method, it can make an optimization: allocate memory for this object outside of the method and work with it in bar.

No copying here of course!

```
__attribute__((noinline)) Vector bar() {
    Vector v{8};
    for (size_t i = 0; i < 8; i++) {
        v.push(i);
    }
    return v;
}

cout << bar().at(0) << endl;
```

Expectation:

"vector copied"
0

Reality (gcc 13.2.0):

0

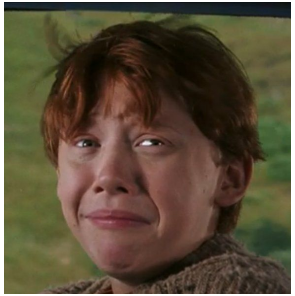
NRVO - **N**amed
Return
Value
Optimization

If compiler can prove that return value was just a local object in the scope of a method, it can make an optimization: allocate memory for this object outside of the method and work with it in bar.

No copying here of course!

Side effects?

Compiler DOESN'T CARE!



```
__attribute__((noinline)) Vector bar2() {
   return Vector{8};
}
```

```
cout << bar2().at(0) << endl;
```

Expectation:

"vector copied"
0

Reality (gcc 13.2.0):

0

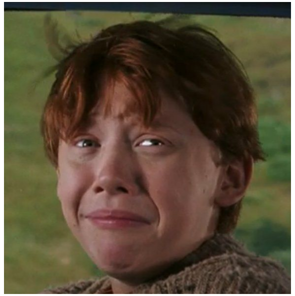
NRVO - ~~Named~~
Return
Value
Optimization

If temporary object is returned, compiler MUST make an optimization: allocate memory for this object outside of the method and work with it in bar2.

No copying here of course!

Side effects?

Compiler DOESN’T CARE!



```
Vector bar3(int a) {
   Vector v{8};
   v.push(42);
   if (a != 42) {
       v.push(13);
       return v;
   } else {
       return Vector{13};
   }
}

cout << bar3().at(0) << endl;
```

Expectation and reality:

"vector copied"

0

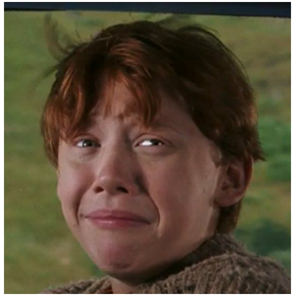
NRVO - Named
       Return
       Value
       Optimization

If compiler can prove that return value was just a local object in the scope of a method, it can make an optimization: allocate memory for this object outside of the method and work with it in bar.

No copying here of course!

Side effects?

Compiler DOESN'T CARE!



```
Vector bar3(int a) {
   Vector v{8};
   v.push(42);
   if (a != 42) {
       v.push(13);
       return v;
   } else {
       return Vector{13};
   }
}

cout << bar3().at(0) << endl;
```

Expectation and reality:

```
"vector copied"
0
```

NRVO - Named
Return
Value
Optimization

If compiler can prove that return value was just a local object in the scope of a method, it can make an optimization: allocate memory for this object outside of the method and work with it in bar.

Here compiler failed to prove that memory could be allocated outside, so, yes, copying is here.

```cpp
class Vector {
    size_t size_ = 0;
    size_t capacity_ ;
    int* data_ ;
public:
    Vector(size_t ic = 16):
            capacity_(ic),
            data_(new int[capacity_]) {}
    int& at(size_t idx) { return data_[idx]; }
    Vector(const Vector& other) { … }
    ~Vector() { delete[] data_; }
};

Vector v{8};
Vector v2(v);
Vector v3 = v;
```

copy ctr
is called

```cpp
void foo(Vector lv) {
    lv.at(0) = 78;
}
```

copy ctr
is called

When else it will be called?

Return values?

```cpp
class Vector {
    size_t size_ = 0;
    size_t capacity_ ;
    int* data_ ;
public:
    Vector(size_t ic = 16):
            capacity_(ic),
            data_(new int[capacity_]) {}
    int& at(size_t idx) { return data_[idx]; }
    Vector(const Vector& other) { … }
    ~Vector() { delete[] data_; }
};

Vector v{8};
Vector v2(v);
Vector v3 = v;
```

copy ctr
is called

```cpp
void foo(Vector lv) {
    lv.at(0) = 78;
}
```

copy ctr
is called

When else it will be called?

Return values?

Yes, but only if RVO/NRVO optimizations were not made.

```
class Vector {
    size_t size_ = 0;
    size_t capacity_ ;
    int* data_ ;
public:
    Vector(size_t ic = 16):
            capacity_(ic),
            data_(new int[capacity_]) {}
    int& at(size_t idx) { return data_[idx]; }
    Vector(const Vector& other) { … }
    ~Vector() { delete[] data_; }
};
```

```
Vector v{8};
Vector v2(v);      ← copy ctr
Vector v3 = v;     ← is called
```

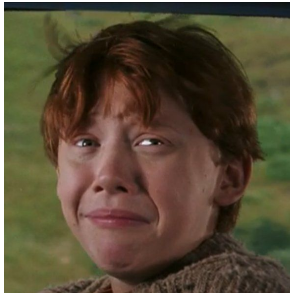
```
void foo(Vector lv) {   ← copy ctr
    lv.at(0) = 78;        is called
}
```

When else it will be called?

Return values?

Yes, but only if RVO/NRVO optimizations were not made.

Be careful with side effects in copy constructors!



```cpp
struct Point {
   int x;
   int y;
};

void foo(Point lp) {
   printf("%d %d\n", lp.x, lp.y);
}

int main() {
   Point p{13, 42};
   foo(p);
   return 0;
}
```

lp is initialized with copy initialization from the argument

Previously we've said that all raw memory of all fields of p was just copied into lp (this is true for C).

```
struct Point {
   int x;
   int y;
};

void foo(Point lp) {
   printf("%d %d\n", lp.x, lp.y);
}

int main() {
   Point p{13, 42};
   foo(p);
   return 0;
}
```

lp is initialized with copy initialization from the argument

Previously we've said that all raw memory of all fields of p was just copied into lp (this is true for C).

In case of C++ it is not quite like that even if you do not have custom copy constructor.

```cpp
class Vector {
    size_t size_ = 0;
    size_t cap_ ;
    int* data_ ;
public:
    ...
    Vector(const Vector& other):
         size_(other.size_), cap_(other.cap_) {

        cout << "vector copied" << endl;
        data_ = new int[cap_];
        for (int i = 0; i < size_; i++) {
            data_[i] = other.data_[i];
        }
    }
    ...
};
```

```cpp
struct PairOfVectors {
   Vector v1;
   Vector v2;
};
```

```
class Vector {
    size_t size_ = 0;
    size_t cap_ ;
    int* data_ ;
public:
    ...
    Vector(const Vector& other):
         size_(other.size_), cap_(other.cap_) {

        cout << "vector copied" << endl;
        data_ = new int[cap_];
        for (int i = 0; i < size_; i++) {
            data_[i] = other.data_[i];
        }
    }
    ...
};
```

```
struct PairOfVectors {
   Vector v1;
   Vector v2;
};

void foo(PairOfVectors pv) {
   ...
}


Vector v1{8};
Vector v2{12};
PairOfVectors pv{v1, v2};

foo(pv);
```

What will be printed?

```cpp
class Vector {
    size_t size_ = 0;
    size_t cap_ ;
    int* data_ ;
public:
    ...
    Vector(const Vector& other):
         size_(other.size_), cap_(other.cap_) {

        cout << "vector copied" << endl;
        data_ = new int[cap_];
        for (int i = 0; i < size_; i++) {
            data_[i] = other.data_[i];
        }
    }
    ...
};
```

```cpp
struct PairOfVectors {
   Vector v1;
   Vector v2;
};

void foo(PairOfVectors pv) {
    ...
}


Vector v1{8};
Vector v2{12};
PairOfVectors pv{v1, v2};

foo(pv);
```

What will be printed?

Copy constructor for PairOfVectors will be generated and it will init v1 and v2 with their copy ctrs.

```cpp
class Vector {
    size_t size_ = 0;
    size_t cap_ ;
    int* data_ ;
public:
    ...
    Vector(const Vector& other):
         size_(other.size_), cap_(other.cap_) {

        cout << "vector copied" << endl;
        data_ = new int[cap_];
        for (int i = 0; i < size_; i++) {
            data_[i] = other.data_[i];
        }
    }
    ...
};
```

```cpp
struct PairOfVectors {
   Vector v1;
   Vector v2;
};

void foo(PairOfVectors pv) {
   ...
}


Vector v1{8};
Vector v2{12};
PairOfVectors pv{v1, v2};

// vector copied
// vector copied

foo(pv);
```

What will be printed?

```cpp
// vector copied
// vector copied
```

Copy constructors

- Needed for implicit and explicit initialization of objects with other instances of the same class,

Copy constructors

- Needed for implicit and explicit initialization of objects with other instances of the same class,

- Needed when some non-trivial logic of memory allocation is needed to initialize an object,

Copy constructors

- Needed for implicit and explicit initialization of objects with other instances of the same class,

- Needed when some non-trivial logic of memory allocation is needed to initialize an object,

- Avoid side effects in them! RVO/NRVO can remove calls of such constructors.

Assignment operator

Assignment operator

```
int main() {
   Vector v1;
   v1.push(13);
   Vector v2;
   v2.push(42);

   v1 = v2;

   std::cout << v1.at(0);
   return 0;
}
```

Assignment operator

```
int main() {
   Vector v1;
   v1.push(13);
   Vector v2;
   v2.push(42);

   v1 = v2;

   std::cout << v1.at(0);
   return 0;
}
```

Questions:

1) Will copy cstr be called here?

2) What will happen when I run this code?

Assignment operator

```
int main() {
    Vector v1;
    v1.push(13);
    Vector v2;
    v2.push(42);

    v1 = v2;

    std::cout << v1.at(0);
    return 0;
}
```

Questions:

1) Will copy cstr be called here?

2) What will happen when I run this code?

No

Assignment operator

```
int main() {
    Vector v1;
    v1.push(13);
    Vector v2;
    v2.push(42);

    v1 = v2;

    std::cout << v1.at(0);
    return 0;
}
```

This isn't a call of copy ctr, so, what is it?

Questions:

1) Will copy cstr be called here?

No

2) What will happen when I run this code?

Assignment operator

```
int main() {
    Vector v1;
    v1.push(13);
    Vector v2;
    v2.push(42);

    v1 = v2;

    std::cout << v1.at(0);
    return 0;
}
```

This isn't a call of copy ctr, so, what is it?

Questions:

1) Will copy cstr be called here?

No

2) What will happen when I run this code?

The special method called copy assignment operator will be called here.

Assignment operator

```
int main() {
    Vector v1;
    v1.push(13);
    Vector v2;
    v2.push(42);

    v1 = v2;

    std::cout << v1.at(0);
    return 0;
}
```

This isn't a call of copy ctr, so, what is it?

Questions:

No

1) Will copy cstr be called here?

2) What will happen when I run this code?

The special method called copy assignment operator will be called here. It is somehow similar to copy constructor, but different, as we don't initialize here.

```cpp
class Vector {
    size_t size_ = 0;
    size_t cap_ ;
    int* data_ ;
public:
    ...
    Vector(const Vector& other) ... { ... }

    ...

    ~Vector() { ... }
};
```

```cpp
class Vector {
    size_t size_ = 0;
    size_t cap_ ;
    int* data_ ;
public:
    ...
    Vector(const Vector& other) ... { ... }

    Vector& operator=(const Vector& other) {
        ...
    }

    ~Vector() { ... }
};
```

```cpp
class Vector {
    size_t size_ = 0;
    size_t cap_ ;
    int* data_ ;
public:
    ...
    Vector(const Vector& other) ... { ... }

    Vector& operator=(const Vector& other) {
        ...
    }

    ~Vector() { ... }
};
```

```cpp
int main() {
   Vector v1; v1.push(13);
   Vector v2; v2.push(42);

   v1 = v2;

   std::cout << v1.at(0);
   return 0;
}
```

```cpp
class Vector {
    size_t size_ = 0;
    size_t cap_ ;
    int* data_ ;
public:
    ...
    Vector(const Vector& other) ... { ... }

    Vector& operator=(const Vector& other) {
        ...
    }

    ~Vector() { ... }
};
```

```cpp
int main() {
   Vector v1; v1.push(13);
   Vector v2; v2.push(42);

   v1 = v2;

   std::cout << v1.at(0);
   return 0;
}
```

So, what we need to do:

```cpp
class Vector {
    size_t size_ = 0;
    size_t cap_ ;
    int* data_ ;
public:
    ...
    Vector(const Vector& other) ... { ... }

    Vector& operator=(const Vector& other) {
        ...
    }

    ~Vector() { ... }
};
```

```cpp
int main() {
   Vector v1; v1.push(13);
   Vector v2; v2.push(42);

   v1 = v2;

   std::cout << v1.at(0);
   return 0;
}
```

So, what we need to do:

1) take already created and initialized object v1

2) and change it to be semantically the same as another object v2

```cpp
class Vector {
    size_t size_ = 0;
    size_t cap_ ;
    int* data_ ;
public:
    ...
    Vector(const Vector& other) ... { ... }

    Vector& operator=(const Vector& other) {
        ...
    }

    ~Vector() { ... }
};
```

```cpp
int main() {
    Vector v1; v1.push(13);
    Vector v2; v2.push(42);

    v1 = v2;

    std::cout << v1.at(0);
    return 0;
}
```

Instead of "=" our custom method will be called.

```cpp
class Vector {
    size_t size_ = 0;
    size_t cap_ ;
    int* data_ ;
public:
    ...
    Vector(const Vector& other) ... { ... }

    Vector& operator=(const Vector& other) {
        ...
    }

    ~Vector() { ... }
};
```

```cpp
int main() {
    Vector v1; v1.push(13);
    Vector v2; v2.push(42);

    v1 = v2;

    std::cout << v1.at(0);
    return 0;
}
```

Instead of "=" our custom method will be called.

v1 will serve as *this in it,
v2 is an argument,

```cpp
class Vector {
    size_t size_ = 0;
    size_t cap_ ;
    int* data_ ;
public:
    ...
    Vector(const Vector& other) ... { ... }

    Vector& operator=(const Vector& other) {
        ...
    }

    ~Vector() { ... }
};
```

```cpp
int main() {
    Vector v1; v1.push(13);
    Vector v2; v2.push(42);

    v1.operator=(v2);   the same!

    std::cout << v1.at(0);
    return 0;
}
```

Instead of "=" our custom method will be called.

v1 will serve as *this in it,
v2 is an argument,

```cpp
class Vector {
    size_t size_ = 0;
    size_t cap_ ;
    int* data_ ;
public:
    ...
    Vector(const Vector& other) ... { ... }

    Vector& operator=(const Vector& other) {
        ...
    }

    ~Vector() { ... }
};
```

```cpp
int main() {
    Vector v1; v1.push(13);
    Vector v2; v2.push(42);

    v1.operator=(v2);   the same!

    std::cout << v1.at(0);
    return 0;
}
```

Instead of "=" our custom method will be called.

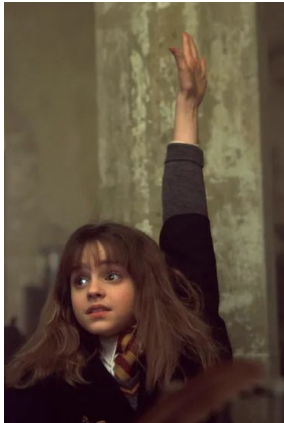
v1 will serve as *this in it, v2 is an argument,

argument can be a (const) lvalue reference or just value, the same about return value.

```cpp
class Vector {
    size_t size_ = 0;
    size_t cap_ ;
    int* data_ ;
public:
    ...
    Vector(const Vector& other) ... { ... }

    Vector& operator=(const Vector& other) {
        ...
    }

    ~Vector() { ... }
};
```

```cpp
int main() {
    Vector v1; v1.push(13);
    Vector v2; v2.push(42);

    v1.operator=(v2);   the same!

    std::cout << v1.at(0);
    return 0;
}
```

Why such method should return anything?



```cpp
class Vector {
    size_t size_ = 0;
    size_t cap_ ;
    int* data_ ;
public:
    ...
    Vector(const Vector& other) ... { ... }

    Vector& operator=(const Vector& other) {
        ...
    }

    ~Vector() { ... }
};
```

```cpp
int main() {
    Vector v1; v1.push(13);
    Vector v2; v2.push(42);

    v1.operator=(v2);   the same!

    std::cout << v1.at(0);
    return 0;
}
```

Why such method should return anything?

```cpp
int lhs = 13;
int rhs = 42;

std::cout << (lhs = rhs);
```

the result of such expression is the destination (lhs after assignment)

```cpp
class Vector {
    size_t size_ = 0;
    size_t cap_ ;
    int* data_ ;
public:
    ...
    Vector(const Vector& other) ... { ... }

    Vector& operator=(const Vector& other) {
        ...
    }

    ~Vector() { ... }
};
```

```cpp
int main() {
    Vector v1; v1.push(13);
    Vector v2; v2.push(42);

    v1.operator=(v2);

    std::cout << v1.at(0);
    return 0;
}
```

the same!

Why such methods usually return a reference, not the value?

```cpp
class Vector {
    size_t size_ = 0;
    size_t cap_ ;
    int* data_ ;
public:
    ...
    Vector(const Vector& other) ... { ... }

    Vector& operator=(const Vector& other) {
        ...
    }

    ~Vector() { ... }
};
```

```cpp
int main() {
   Vector v1; v1.push(13);
   Vector v2; v2.push(42);

   v1.operator=(v2);   the same!

   std::cout << v1.at(0);
   return 0;
}
```

Why such methods usually return a reference, not the value?

Because you can things like this:

```cpp
Vector v1; v1.push(13);
Vector v2; v2.push(42);
Vector v3; v3.push(78);

(v1 = v2) = v3;
```

```
class Vector {
    size_t size_ = 0;
    size_t cap_ ;
    int* data_ ;
public:
    ...
    Vector(const Vector& other) ... { ... }

    Vector& operator=(const Vector& other) {
        ...
    }

    ~Vector() { ... }
};
```

```
int main() {
    Vector v1; v1.push(13);
    Vector v2; v2.push(42);

    v1.operator=(v2);   the same!

    std::cout << v1.at(0);
    return 0;
}
```

Why such methods usually return a reference, not the value?

Because you can things like this:

```
Vector v1; v1.push(13);
Vector v2; v2.push(42);
Vector v3; v3.push(78);
(v1 = v2) = v3;
// it is expected v1 to be
// semantically equals v3
```

```
class Vector {
    size_t size_ = 0;
    size_t cap_ ;
    int* data_ ;
public:
    ...
    Vector(const Vector& other) ... { ... }

    Vector& operator=(const Vector& other) {
        ...
    }

    ~Vector() { ... }
};
```

```
int main() {
    Vector v1; v1.push(13);
    Vector v2; v2.push(42);

    v1.operator=(v2);   the same!

    std::cout << v1.at(0);
    return 0;
}
```

Why such methods usually return a reference, not the value?

Because you can things like this:

```
Vector v1; v1.push(13);
Vector v2; v2.push(42);
Vector v3; v3.push(78);
(v1 = v2) = v3;
// so, (v1 = v2) shouldn’t return a
// copy, but the v1 itself.
```

```cpp
class Vector {
    size_t size_ = 0;
    size_t cap_ ;
    int* data_ ;
public:
    ...
    Vector(const Vector& other) ... { ... }

    Vector& operator=(const Vector& other) {
        ...
    }

    ~Vector() { ... }
};
```

```cpp
int main() {
    Vector v1; v1.push(13);
    Vector v2; v2.push(42);

    v1.operator=(v2);   the same!

    std::cout << v1.at(0);
    return 0;
}
```

So, what we need to do:

1) take already created and initialized object v1

2) and change it to be semantically the same as another object v2

```cpp
class Vector {
    size_t size_ = 0;
    size_t cap_ ;
    int* data_ ;
public:
    ...
    Vector(const Vector& other) ... { ... }

    Vector& operator=(const Vector& other) {
        size_ = other.size_;
        cap_ = other.cap_;
        data_ = ???;
    }

    ~Vector() { ... }
};
```

```cpp
int main() {
    Vector v1; v1.push(13);
    Vector v2; v2.push(42);

    v1.operator=(v2);   the same!

    std::cout << v1.at(0);
    return 0;
}
```

So, what we need to do:

1) take already created and initialized object v1

2) and change it to be semantically the same as another object v2

```cpp
class Vector {
    size_t size_ = 0;
    size_t cap_ ;
    int* data_ ;
public:
    ...
    Vector(const Vector& other) ... { ... }

    Vector& operator=(const Vector& other) {
        size_ = other.size_;
        cap_ = other.cap_;
        delete[] data_;
    }

    ~Vector() { ... }
};
```

```cpp
int main() {
    Vector v1; v1.push(13);
    Vector v2; v2.push(42);

    v1.operator=(v2);   the same!

    std::cout << v1.at(0);
    return 0;
}
```

So, what we need to do:

1) take already created and initialized object v1

2) and change it to be semantically the same as another object v2

```cpp
class Vector {
    size_t size_ = 0;
    size_t cap_ ;
    int* data_ ;
public:
    ...
    Vector(const Vector& other) ... { ... }

    Vector& operator=(const Vector& other) {
        size_ = other.size_;
        cap_ = other.cap_;
        delete[] data_;
        data_ = new int[cap_];
    }

    ~Vector() { ... }
};
```

```cpp
int main() {
    Vector v1; v1.push(13);
    Vector v2; v2.push(42);

    v1.operator=(v2);   the same!

    std::cout << v1.at(0);
    return 0;
}
```

So, what we need to do:

1) take already created and initialized object v1

2) and change it to be **semantically the same** as another object v2

```
class Vector {
    size_t size_ = 0;
    size_t cap_ ;
    int* data_ ;
public:
    ...
    Vector(const Vector& other) ... { ... }

    Vector& operator=(const Vector& other) {
        size_ = other.size_;
        cap_ = other.cap_;
        delete[] data_;
        data_ = new int[cap_];
        for (size_t i = 0; i < size_; i++)
            data_[i] = other.data_[i];
    }
    ...
```

```
int main() {
    Vector v1; v1.push(13);
    Vector v2; v2.push(42);

    v1.operator=(v2);   the same!

    std::cout << v1.at(0);
    return 0;
}
```

So, what we need to do:

1) take already created and initialized object v1

2) and change it to be semantically the same as another object v2

```cpp
class Vector {
    size_t size_ = 0;
    size_t cap_ ;
    int* data_ ;
public:
    ...
    Vector(const Vector& other) ... { ... }

    Vector& operator=(const Vector& other) {
        size_ = other.size_;
        cap_ = other.cap_;
        delete[] data_;
        data_ = new int[cap_];
        for (size_t i = 0; i < size_; i++)
            data_[i] = other.data_[i];
        return *this;
    }
```

```cpp
int main() {
    Vector v1; v1.push(13);
    Vector v2; v2.push(42);

    v1.operator=(v2);   the same!

    std::cout << v1.at(0);
    return 0;
}
```

So, what we need to do:

1) take already created and initialized object v1

2) and change it to be semantically the same as another object v2

```cpp
class Vector {
    size_t size_ = 0;
    size_t cap_ ;
    int* data_ ;
public:
    ...
    Vector(const Vector& other) ... { ... }

    Vector& operator=(const Vector& other) {
        size_ = other.size_;
        cap_ = other.cap_;
        delete[] data_;
        data_ = new int[cap_];
        for (size_t i = 0; i < size_; i++)
            data_[i] = other.data_[i];
        return *this;
    }
```

```cpp
int main() {
    Vector v1; v1.push(13);
    Vector v2; v2.push(42);

    v1.operator=(v2);   the same!

    std::cout << v1.at(0);
    return 0;
}
```

Any problems here?

```cpp
class Vector {
    size_t size_ = 0;
    size_t cap_ ;
    int* data_ ;
public:
    ...
    Vector(const Vector& other) ... { ... }

    Vector& operator=(const Vector& other) {
        size_ = other.size_;
        cap_ = other.cap_;
        delete[] data_;
        data_ = new int[cap_];
        for (size_t i = 0; i < size_; i++)
            data_[i] = other.data_[i];
        return *this;
    }
```

```cpp
int main() {
    Vector v1; v1.push(13);
    Vector v2; v2.push(42);

    v1.operator=(v2);   the same!

    std::cout << v1.at(0);
    return 0;
}
```

Any problems here?

```cpp
    Vector v1;
    v1.push(13);

    v1 = v1;
```

```
class Vector {
    size_t size_ = 0;
    size_t cap_ ;
    int* data_ ;
public:
    ...
    Vector(const Vector& other) ... { ... }

    Vector& operator=(const Vector& other) {
        size_ = other.size_;
        cap_ = other.cap_;
        delete[] data_;
        data_ = new int[cap_];
        for (int i = 0; i < size_; i++)
            data_[i] = other.data_[i];
        return *this;
    }
```

```
int main() {
    Vector v1; v1.push(13);
    Vector v2; v2.push(42);

    v1.operator=(v2);   the same!

    std::cout << v1.at(0);
    return 0;
}
```

Any problems here?

```
Vector v1;
v1.push(13);

v1 = v1;
```

Absolutely correct code, but with our implementation will cause UB.

```cpp
class Vector {
    size_t size_ = 0;
    size_t cap_ ;
    int* data_ ;
public:
    ...
    Vector& operator=(const Vector& other) {
        if (this == &other) { ... }
        size_ = other.size_;
        cap_ = other.cap_;
        delete[] data_;
        data_ = new int[cap_];
        for (size_t i = 0; i < size_; i++)
            data_[i] = other.data_[i];
        return *this;
    }
    ...
```

```cpp
int main() {
    Vector v1; v1.push(13);
    Vector v2; v2.push(42);

    v1.operator=(v2);   the same!

    std::cout << v1.at(0);
    return 0;
}
```

Any problems here?

```cpp
    Vector v1;
    v1.push(13);

    v1 = v1;
```

Absolutely correct code, but with our implementation will cause UB.

```cpp
class Vector {
    size_t size_ = 0;
    size_t cap_ ;
    int* data_ ;
public:
    ...
    Vector& operator=(const Vector& other) {
        if (this != &other) {
            size_ = other.size_;
            cap_ = other.cap_;
            delete[] data_;
            data_ = new int[cap_];
            for (size_t i = 0; i < size_; i++)
                data_[i] = other.data_[i];
        }
        return *this;
    }
```

```cpp
int main() {
    Vector v1; v1.push(13);
    Vector v2; v2.push(42);

    v1.operator=(v2);   the same!

    std::cout << v1.at(0);
    return 0;
}
```

Any problems here?

```cpp
    Vector v1;
    v1.push(13);

    v1 = v1;
```

Now everything works correctly.

Assignment operator

```
int main() {
    Vector v1;
    v1.push(13);
    Vector v2;
    v2.push(42);

    v1 = v2;

    std::cout << v1.at(0);
    return 0;
}
```

Questions:

1) Will copy cstr be called here?

2) What will happen when I run this code?

No

Assignment operator

```
int main() {
   Vector v1;
   v1.push(13);
   Vector v2;
   v2.push(42);

   v1 = v2;

   std::cout << v1.at(0);
   return 0;
}
```

Questions:

1) Will copy cstr be called here?

No

2) What will happen when I run this code without implementing copy assignment operator?

Assignment operator

```
int main() {
    Vector v1;
    v1.push(13);
    Vector v2;
    v2.push(42);

    v1 = v2;

    std::cout << v1.at(0);
    return 0;
}
```

Questions:

1) Will copy cstr be called here?

No

2) What will happen when I run this code without implementing copy assignment operator?

UB

Assignment operator

```
int main() {
   Vector v1;
   v1.push(13);
   Vector v2;
   v2.push(42);

   v1 = v2;

   std::cout << v1.at(0);
   return 0;
}
```

Without implementing your custom copy assignment operator, the default one will be generated and used.

Assignment operator

```
int main() {
   Vector v1;
   v1.push(13);
   Vector v2;
   v2.push(42);

   v1 = v2;

   std::cout << v1.at(0);
   return 0;
}
```

Without implementing your custom copy assignment operator, the default one will be generated and used.

It just assigns each field of v2 to the corresponding fields in v1.

Assignment operator

```
int main() {
   Vector v1;
   v1.push(13);
   Vector v2;
   v2.push(42);

   v1 = v2;

   std::cout << v1.at(0);
   return 0;
}
```

Without implementing your custom copy assignment operator, the default one will be generated and used.

It just assigns each field of v2 to the corresponding fields in v1.

In case of primitive types (like int* the type of data_), it means just copying the value.

Assignment operator

```
int main() {
   Vector v1;
   v1.push(13);
   Vector v2;
   v2.push(42);

   v1 = v2;

   std::cout << v1.at(0);
   return 0;
}
```

It just assigns each field of v2 to the corresponding fields in v1.

In case of primitive types (like int* the type of data_), it means just copying the value.

So, both v1.data_ and v2.data_ will point the same memory (and it will be double freed in their destructors after return)

Rule of three

Rule of three

```
class Vector {
    size_t size_ = 0;
    size_t cap_ ;
    int* data_ ;
public:
    ...
    Vector(const Vector& other) ... { ... }

    Vector& operator=(const Vector& other) { ... }

    ~Vector() { ... }
};
```

Rule of three

```
class Vector {
    size_t size_ = 0;
    size_t cap_ ;
    int* data_ ;
public:
    ...
    Vector(const Vector& other) ... { ... }

    Vector& operator=(const Vector& other) { ... }

    ~Vector() { ... }
};
```

If you have at least one of these three: copy constructor, copy assignment operator or destructor =>

than you most probably need all three of them.

Rule of three

Takeaways

- Initialization in C++: default, value, direct, copy and aggregate.
- Member initializer list: use it, but be aware of pitfalls (fields initialization order)
- Copy constructors: implement for complex classes, be aware of side effects (RVO)
- Copy assignment operators and rule of 3.

Not So Tiny Task №3 (1 point)

Improve solution from NSTT #1 by adding copy constructors and copy assignment operators where needed.

Also, use member initializers lists in constructors.

Don't forget to add tests that show that copy constructors and assign operators work correctly.