Not So Tiny Task №4 (1 point)

Improve solution from NSTT #1 by adding move constructors and move assignment operators where needed.

Don’t forget to add tests that show that copy constructors and assign operators work correctly.

System Programming with C++

Move semantics

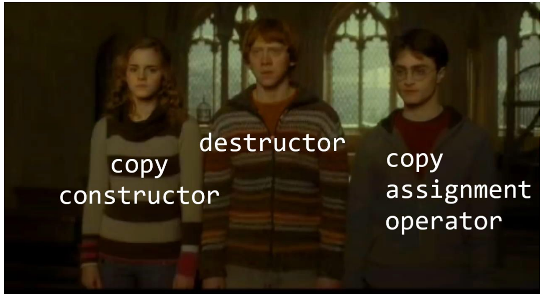
Where are we

Where are we

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

Where are we

We follow rule of 3 and can somehow hope that object is valid state.




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

Operators overloading example

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

```
Vector v1;
v1.push(13);
Vector v2;
v2.push(42);

Vector v3 = v1 + v2;
```

Operators overloading example

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

```
Vector v1;
v1.push(13);
Vector v2;
v2.push(42);

Vector v3 = v1 + v2;
```

What we want here: a new Vector object that contains elements from both v1 and v2

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

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

Vector v3 = v1 + v2;
```

What we want here: a new Vector object that contains elements from both v1 and v2

```cpp
class Vector {
    size_t size_ = 0;
    size_t cap_ ;
    int* data_ ;
public:
    ...
    Vector operator+(const Vector& other) {
        Vector result = *this;
        for (size_t i = 0; i < other.size_; i++) {
            result.push(other.data_[i]);
        }
        return result;
    }
    ...
};
```

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

Vector v3 = v1 + v2;
```

What we want here: a new Vector object that contains elements from both v1 and v2

```cpp
class Vector {
    size_t size_ = 0;
    size_t cap_ ;
    int* data_ ;
public:
    ...
    Vector operator+(const Vector& other) {
        Vector result = *this;
        for (size_t i = 0; i < other.size_; i++) {
            result.push(other.data_[i]);
        }
        return result;
    }
    ...
};
```

We have copy ctr,
it will work here

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

Vector v3 = v1 + v2;
```

What we want here: a new Vector object that contains elements from both v1 and v2

```cpp
class Vector {
    size_t size_ = 0;
    size_t cap_ ;
    int* data_ ;
public:
    Vector operator+(const Vector& other) {
        if (size_ == 0) {
            return other;
        }
        Vector result = *this;
        for (size_t i = 0; i < other.size_; i++) {
            result.push(other.data_[i]);
        }
        return result;
    }
};
```

Small optimization for situation where this is empty (possibly premature)

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

Vector v3 = v1 + v2;
```

What we want here: a new Vector object that contains elements from both v1 and v2

```
class Vector {
    size_t size_ = 0;
    size_t cap_ ;
    int* data_ ;
public:
    Vector operator+(const Vector& other) {
        if (size_ == 0) {
            return other;
        }
        Vector result = *this;
        for (size_t i = 0; i < other.size_; i++) {
            result.push(other.data_[i]);
        }
        return result;
    }
};
```

Small optimization for situation where this is empty (possibly premature)

Still not the best implementation (why?), but we will use it during this lecture

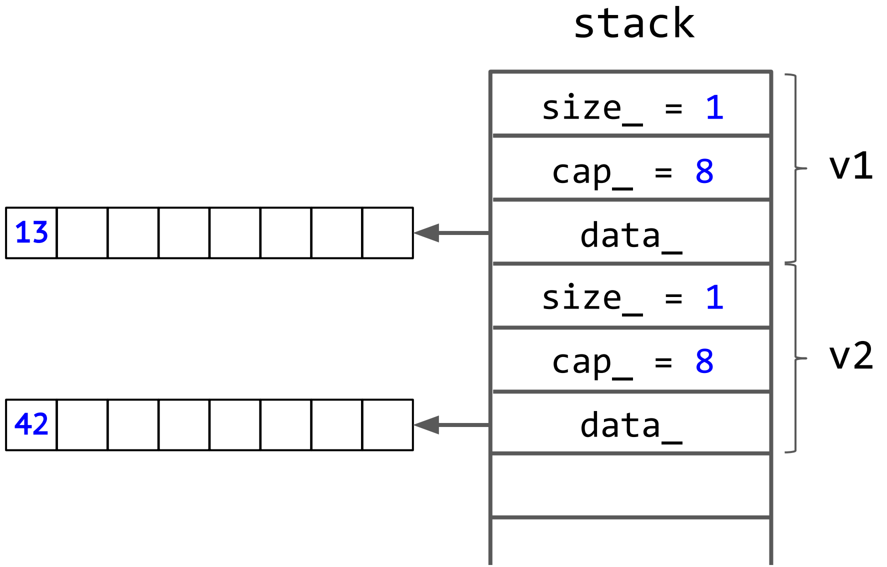
```
Vector v1;
v1.push(13);
Vector v2;
v2.push(42);

Vector v3 = v1 + v2;
```

What we want here: a new Vector object that contains elements from both v1 and v2

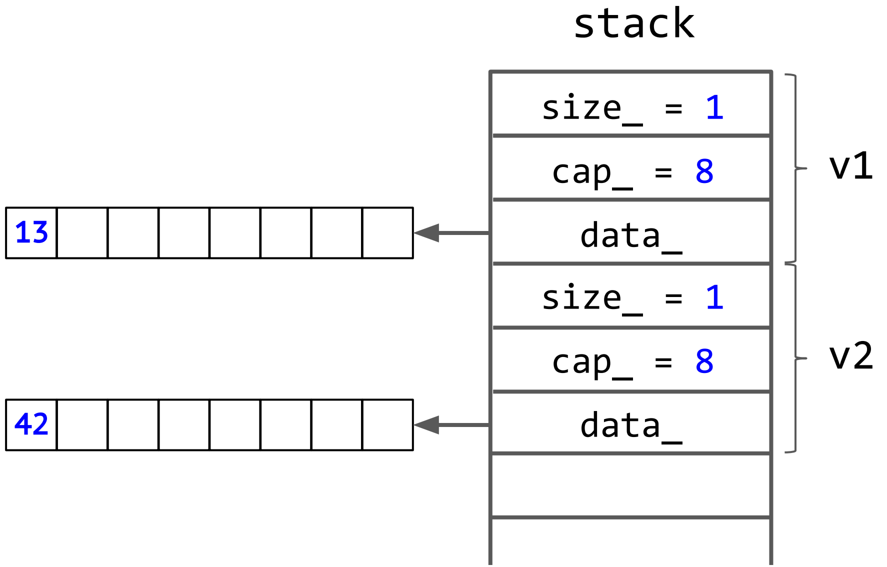
```cpp
class Vector {
    size_t size_ = 0;
    size_t cap_ ;
    int* data_ ;
public:
    Vector operator+(const Vector& other) {
        if (size_ == 0) {
            return other;
        }
        Vector result = *this;
        for (size_t i = 0; i < other.size_; i++) {
            result.push(other.data_[i]);
        }
        return result;
    }
};
```

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

Vector v3 = v1 + v2;

cout << v3.at(1) << endl;
// 42
```

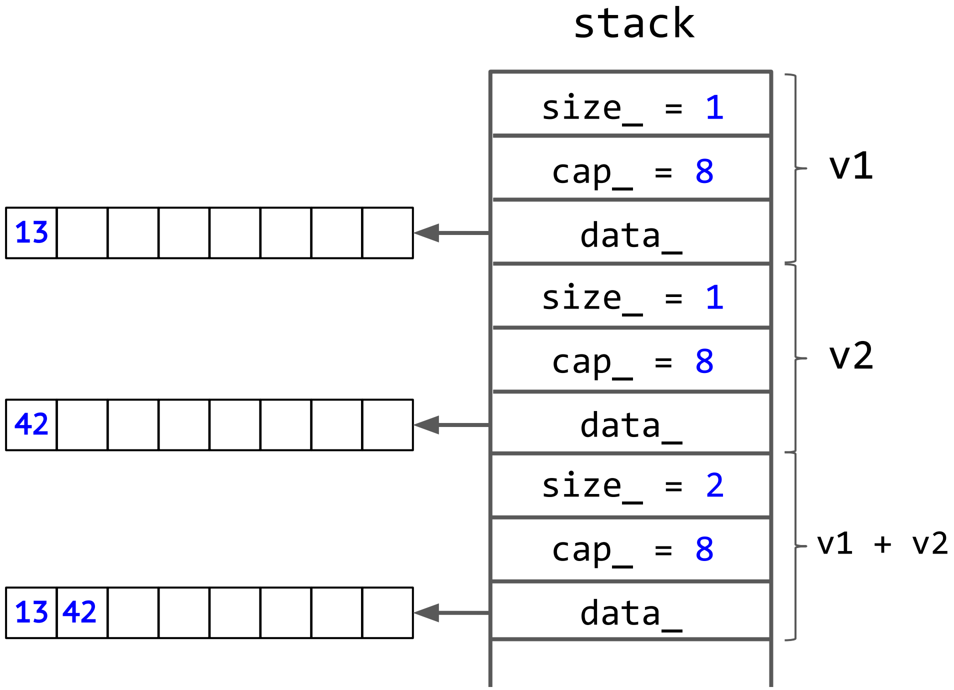
```
Vector v1;
v1.push(13);
Vector v2;
v2.push(42);

Vector v3 = v1 + v2;

cout << v3.at(1) << endl;
// 42
```



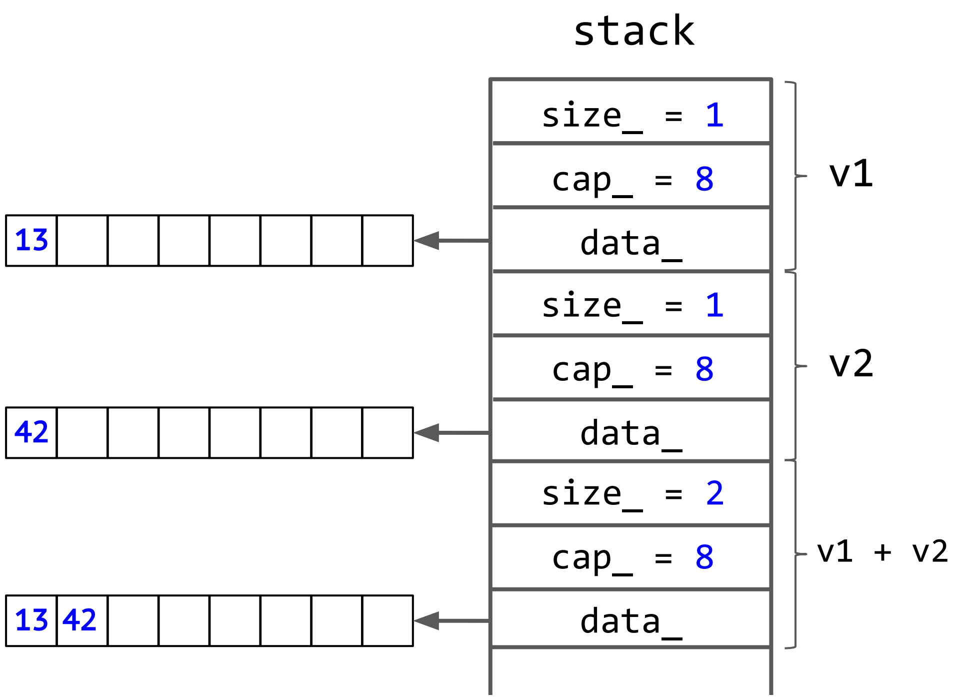
```
Vector v1;
v1.push(13);
Vector v2;
v2.push(42);

Vector v3 = v1 + v2;

cout << v3.at(1) << endl;
// 42
```



```
Vector v1;
v1.push(13);
Vector v2;
v2.push(42);

Vector v3 = v1 + v2;

cout << v3.at(1) << endl;
// 42
```



```
Vector v1;
v1.push(13);
Vector v2;
v2.push(42);

Vector v3 = v1 + v2;

cout << v3.at(1) << endl;
// 42
```



What about NRVO?

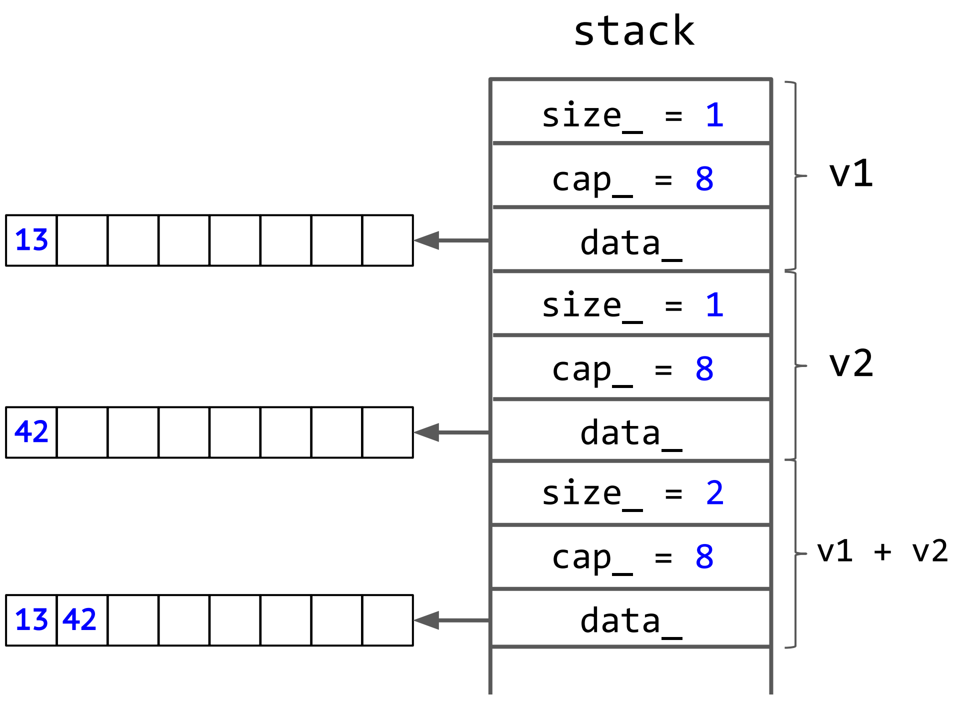
```cpp
class Vector {
    size_t size_ = 0;
    size_t cap_ ;
    int* data_ ;
public:
    Vector operator+(const Vector& other) {
        if (size_ == 0) {
            return other;
        }
        Vector result = *this;
        for (size_t i = 0; i < other.size_; i++) {
            result.push(other.data_[i]);
        }
        return result;
    }
};
```

Failed because of this!

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

Vector v3 = v1 + v2;
```

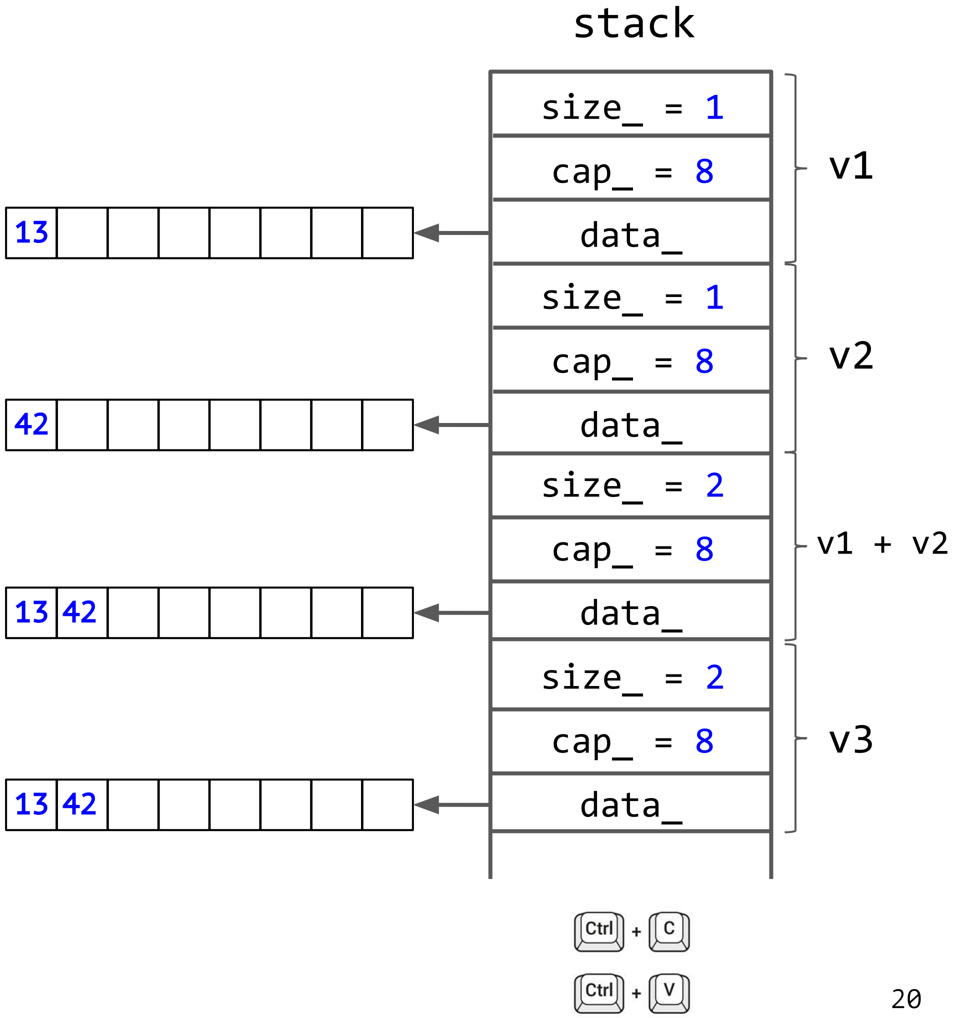
What we want here: a new Vector object that contains elements from both v1 and v2

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

Vector v3 = v1 + v2;

cout << v3.at(1) << endl;
// 42
```

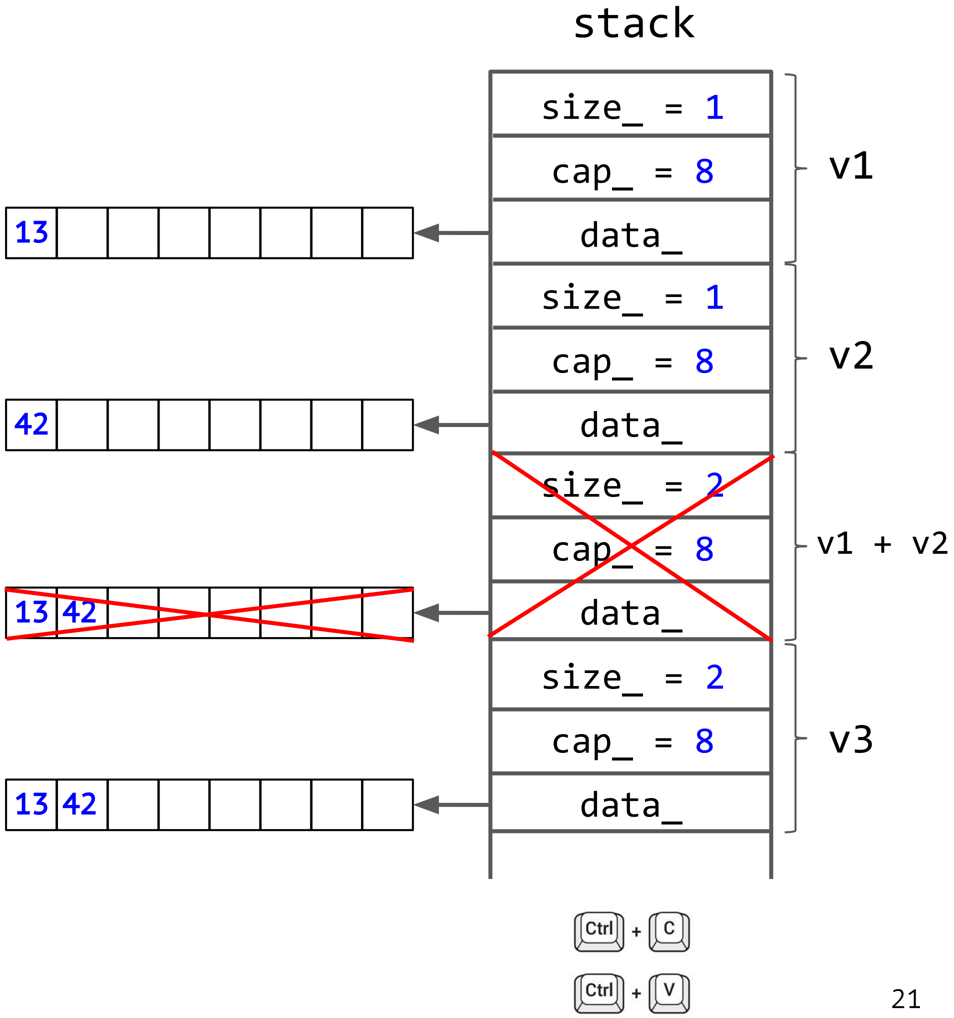


The result of v1 + v2 is a temporary object that is alive till the end of full expression (;)

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

Vector v3 = v1 + v2;

cout << v3.at(1) << endl;
// 42
```

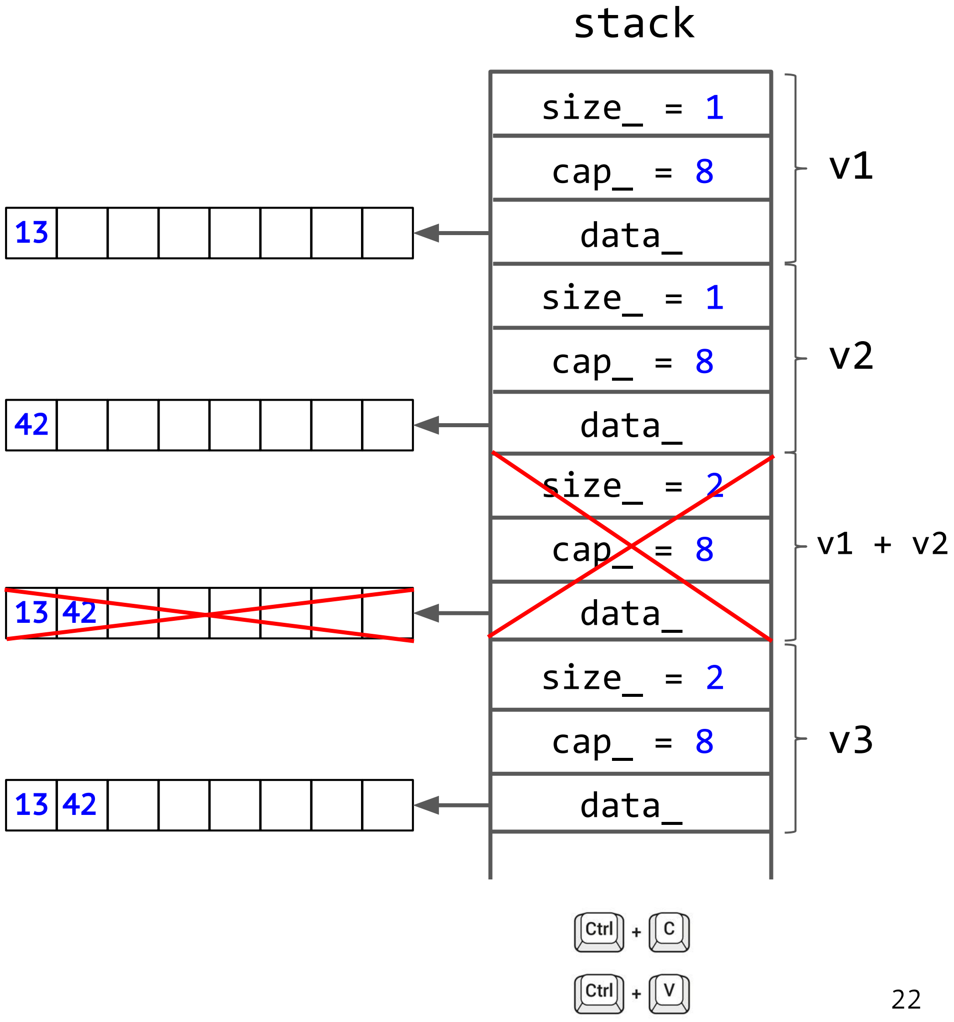


```cpp
Vector v1;
v1.push(13);
Vector v2;
v2.push(42);

Vector v3 = v1 + v2;

cout << v3.at(1) << endl;
// 42
```



```
Vector v1;
v1.push(13);
Vector v2;
v2.push(42);

Vector v3 = v1 + v2;

cout << v3.at(1) << endl;
// 42
```

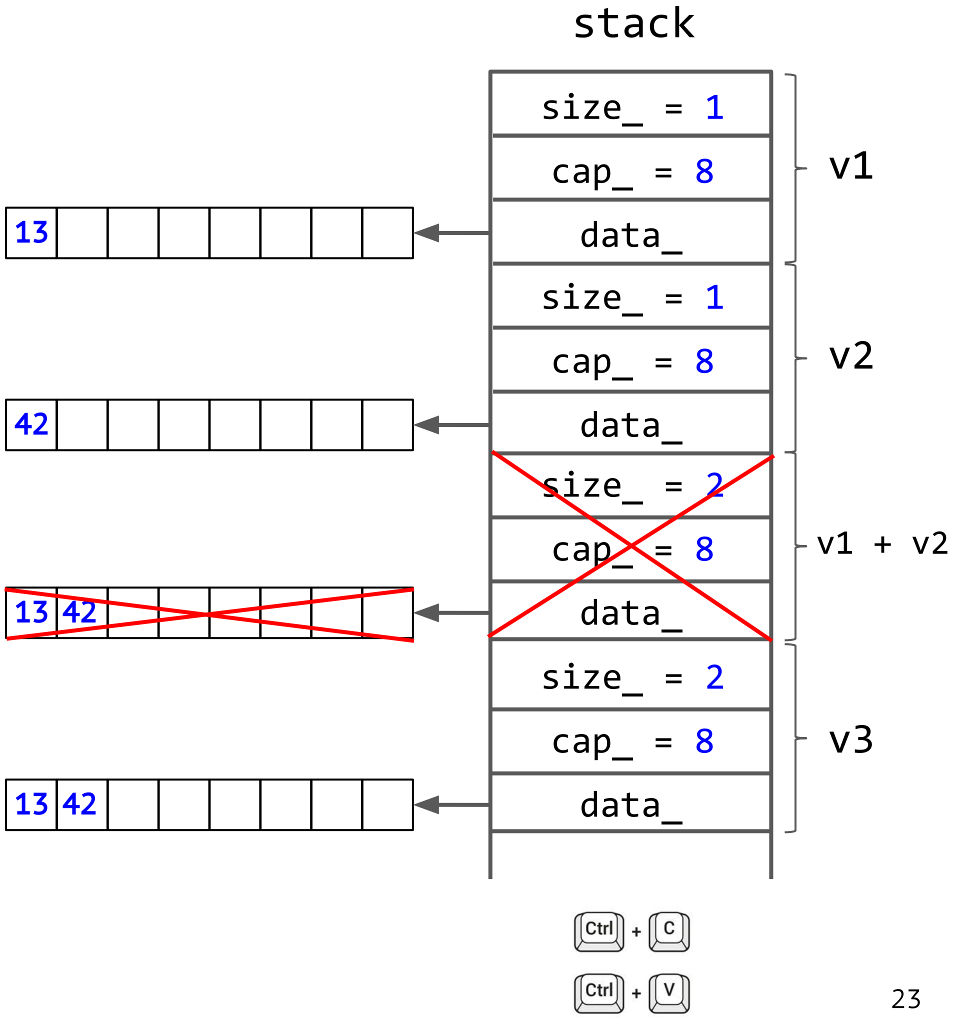
Everything works correctly
(thanks to the rule of 3), but...



```
Vector v1;
v1.push(13);
Vector v2;
v2.push(42);

Vector v3 = v1 + v2;

cout << v3.at(1) << endl;
// 42
```

Everything works correctly
(thanks to the rule of 3), but...

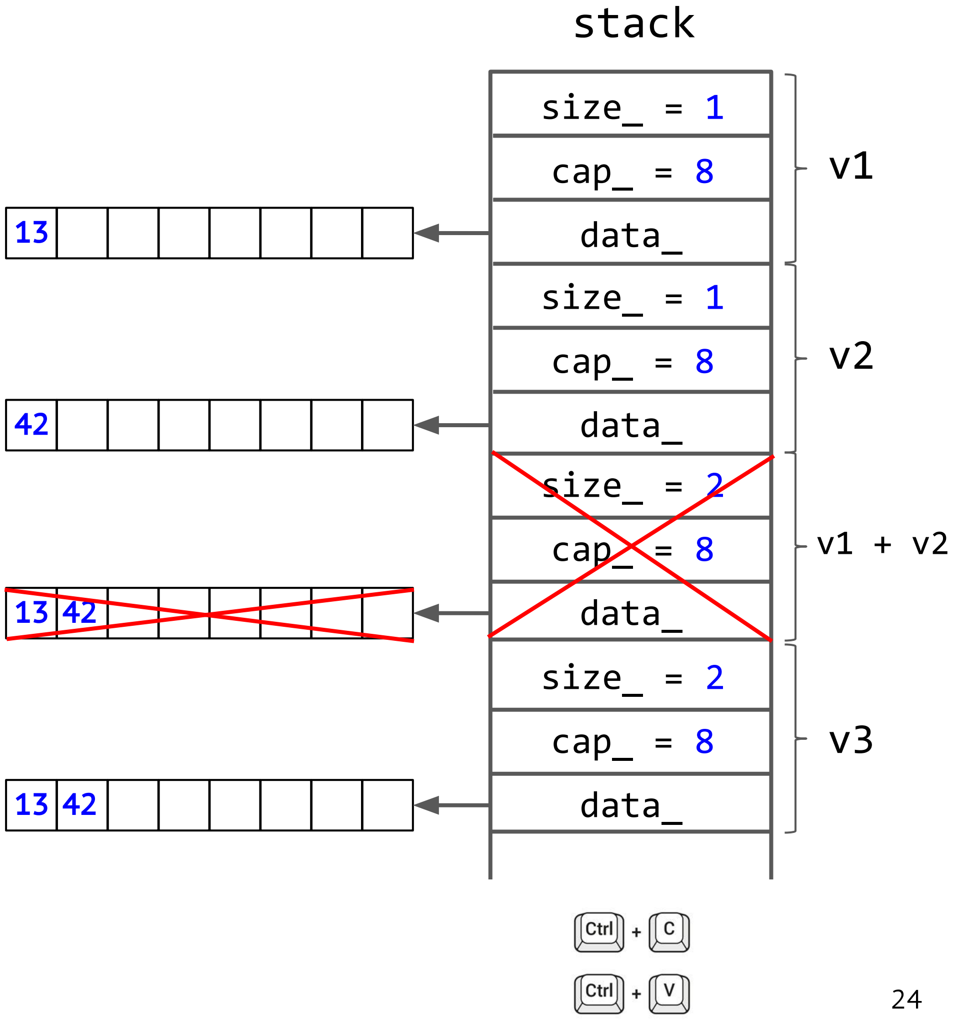
a lot of work to create and initialize temporary object...



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

Vector v3 = v1 + v2;

cout << v3.at(1) << endl;
// 42
```

Everything works correctly
(thanks to the rule of 3), but...

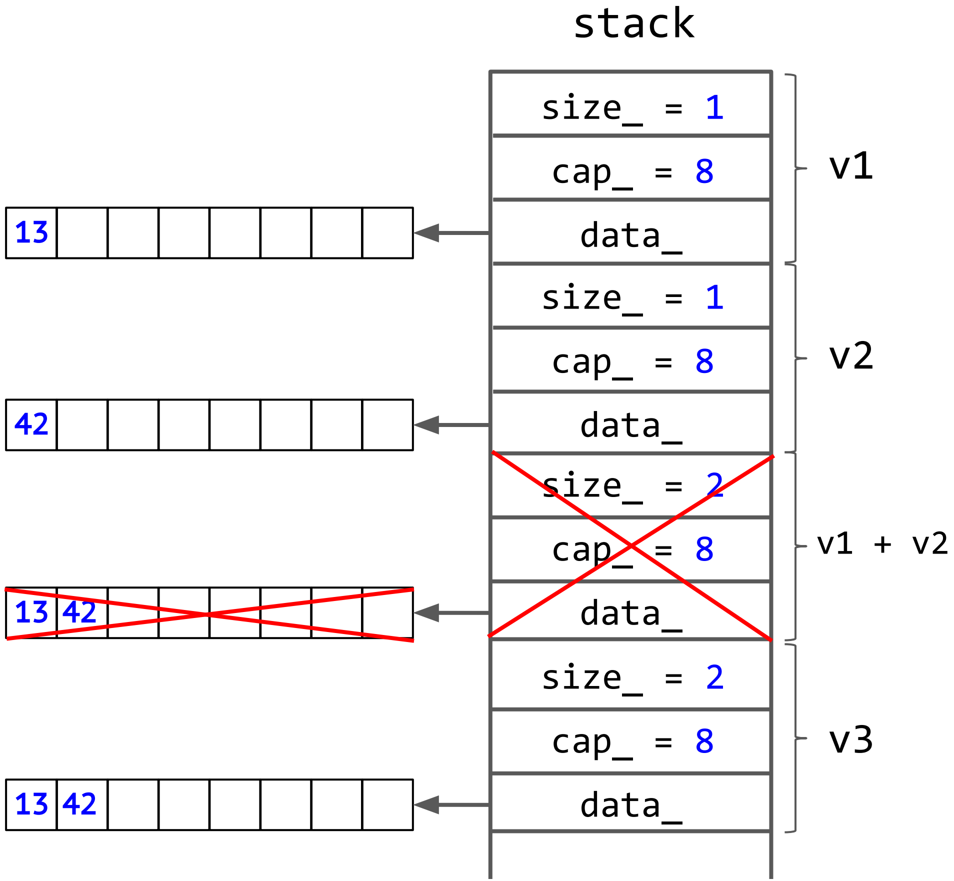
a lot of work to create and initialize temporary object which is only used to initialize v3!

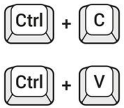


```cpp
Vector v1;
v1.push(13);
Vector v2;
v2.push(42);

Vector v3 = v1 + v2;

cout << v3.at(1) << endl;
// 42
```

Everything works correctly
(thanks to the rule of 3), but...

a lot of work to create and initialize temporary object which is only used to initialize v3! Is it really C++ way?





Another example

```
Vector getRandomVector(size_t max_size) {
    if (max_size == 0) {
        return Vector{};
    }

    Vector result{max_size};
    std::random_device rd;
    std::mt19937_64 gen(rd());
    std::uniform_int_distribution<int> dis;

    for (size_t i = 0; i < max_size; i++) {
        result.at(i) = dis(gen);
    }
    return result;
}
```

Another example

```
cout <<
getRandomVector(10).at(9)
<< endl;
```

```
Vector getRandomVector(size_t max_size) {
    if (max_size == 0) {
        return Vector{};
    }

    Vector result{max_size};
    std::random_device rd;
    std::mt19937_64 gen(rd());
    std::uniform_int_distribution<int> dis;

    for (size_t i = 0; i < max_size; i++) {
        result.at(i) = dis(gen);
    }
    return result;
}
```

Another example

Imagine we have debug prints in constructor, copy constructor and destructor. What will be printed?

```cpp
cout <<
getRandomVector(10).at(9)
<< endl;
```

```cpp
Vector getRandomVector(size_t max_size) {
    if (max_size == 0) {
        return Vector{};
    }

    Vector result{max_size};
    std::random_device rd;
    std::mt19937_64 gen(rd());
    std::uniform_int_distribution<int> dis;

    for (size_t i = 0; i < max_size; i++) {
        result.at(i) = dis(gen);
    }
    return result;
}
```

Another example

Imagine we have debug prints in constructor, copy constructor and destructor. What will be printed?

```
cout <<
getRandomVector(10).at(9)
<< endl;


// vector created
```

```
Vector getRandomVector(size_t max_size) {
    if (max_size == 0) {
        return Vector{};
    }

    Vector result{max_size};
    std::random_device rd;
    std::mt19937_64 gen(rd());
    std::uniform_int_distribution<int> dis;

    for (size_t i = 0; i < max_size; i++) {
        result.at(i) = dis(gen);
    }
    return result;
}
```

Another example

Imagine we have debug prints in constructor, copy constructor and destructor. What will be printed?

```
cout <<
getRandomVector(10).at(9)
<< endl;

// vector created
// vector copied
```

```
Vector getRandomVector(size_t max_size) {
    if (max_size == 0) {
        return Vector{};
    }

    Vector result{max_size};
    std::random_device rd;
    std::mt19937_64 gen(rd());
    std::uniform_int_distribution<int> dis;

    for (size_t i = 0; i < max_size; i++) {
        result.at(i) = dis(gen);
    }
    return result;
}
```

Another example

Imagine we have debug prints in constructor, copy constructor and destructor. What will be printed?

```
cout <<
getRandomVector(10).at(9)
<< endl;

// vector created
// vector copied
// vector deleted
```

```
Vector getRandomVector(size_t max_size) {
    if (max_size == 0) {
        return Vector{};
    }

    Vector result{max_size};
    std::random_device rd;
    std::mt19937_64 gen(rd());
    std::uniform_int_distribution<int> dis;

    for (size_t i = 0; i < max_size; i++) {
        result.at(i) = dis(gen);
    }
    return result;
}
```

Another example

Imagine we have debug prints in constructor, copy constructor and destructor. What will be printed?

```
cout <<
getRandomVector(10).at(9)
<< endl;

// vector created
// vector copied
// vector deleted
// 301989906
```

```
Vector getRandomVector(size_t max_size) {
    if (max_size == 0) {
        return Vector{};
    }

    Vector result{max_size};
    std::random_device rd;
    std::mt19937_64 gen(rd());
    std::uniform_int_distribution<int> dis;

    for (size_t i = 0; i < max_size; i++) {
        result.at(i) = dis(gen);
    }
    return result;
}
```

Another example

Imagine we have debug prints in constructor, copy constructor and destructor. What will be printed?

```
cout <<
getRandomVector(10).at(9)
<< endl;

// vector created
// vector copied
// vector deleted
// 301989906
// vector deleted
```

```
Vector getRandomVector(size_t max_size) {
    if (max_size == 0) {
        return Vector{};
    }

    Vector result{max_size};
    std::random_device rd;
    std::mt19937_64 gen(rd());
    std::uniform_int_distribution<int> dis;

    for (size_t i = 0; i < max_size; i++) {
        result.at(i) = dis(gen);
    }
    return result;
}
```

Another example

Imagine we have debug prints in constructor, copy constructor and destructor. What will be printed?

```
cout <<
getRandomVector(10).at(9)
<< endl;

// vector created
// vector copied
// vector deleted
// 301989906
// vector deleted
```

2 allocations, 1 copying, 2 deleting.
Isn't it too heavy for such sample?

```
Vector getRandomVector(size_t max_size) {
    if (max_size == 0) {
        return Vector{};
    }

    Vector result{max_size};
    std::random_device rd;
    std::mt19937_64 gen(rd());
    std::uniform_int_distribution<int> dis;

    for (size_t i = 0; i < max_size; i++) {
        result.at(i) = dis(gen);
    }
    return result;
}
```

One more motivation example

One more motivation example

```
void swap(int& a, int& b) {
    int tmp = a;
    a = b;
    b = tmp;
}
```

Classical use case for references

One more motivation example

```
void swap(int& a, int& b) {
    int tmp = a;
    a = b;
    b = tmp;
}
```

```
Classical use case for references
```

```
void swap(Vector& a, Vector& b) {
    Vector tmp = a;
    a = b;
    b = tmp;
}
```

```
Is it good for our Vectors?
```

One more motivation example

```
void swap(int& a, int& b) {
    int tmp = a;
    a = b;
    b = tmp;
}
```

Classical use case for references

```
void swap(Vector& a, Vector& b) {
    Vector tmp = a;
    a = b;
    b = tmp;
}
```

Is it good for our Vectors? It will work, but…

One more motivation example

```
void swap(int& a, int& b) {
   int tmp = a;
   a = b;
   b = tmp;
}
```

Classical use case for references

```
void swap(Vector& a, Vector& b) {
   Vector tmp = a;
   a = b;
   b = tmp;
}
```

Is it good for our Vectors? It will work, but…

1 copy ctr, 2 copy assign operators, 1 destructor call 😢

One more motivation example

```
void swap(int& a, int& b) {
    int tmp = a;
    a = b;
    b = tmp;
}
```

Classical use case for references

Of course you can do better than this manually, but what about generalization?

```
void swap(Vector& a, Vector& b) {
    Vector tmp = a;
    a = b;
    b = tmp;
}
```

Is it good for our Vectors? It will work, but…

1 copy ctr, 2 copy assign operators, 1 destructor call 😢

One more motivation example

```
template<typename T>
void swap(T& a, T& b) {
    T tmp = a;
    a = b;
    b = tmp;
}
```

Classical use case for references

One more motivation example

```
template<typename T>
void swap(T& a, T& b) {
   T tmp = a;
   a = b;
   b = tmp;
}
```

Classical use case for references

This the first time we use templates, we will discuss them later in deep details.

Currently just think about it as about a language feature for generalization (like generics in Java)

One more motivation example

```
template<typename T>
void swap(T& a, T& b) {
   T tmp = a;
   a = b;
   b = tmp;
}
```

Classical use case for references

You can use this swap for Vectors, but it will work terribly slow because of copy ctr, additional allocations and deallocations.

What was in common for all examples?

```
Vector v1;
v1.push(13);
Vector v2;
v2.push(42);

Vector v3 = v1 + v2;

cout << v3.at(1) << endl;
// 42
```

```
cout <<
getRandomVector(10).at(9)
<< endl;

// vector created
// vector copied
// vector deleted
// 301989906
// vector deleted
```

```
void swap(Vector& a,
          Vector& b) {
    Vector tmp = a;
    a = b;
    b = tmp;
}
```

What was in common for all examples?

```
Vector v1;
v1.push(13);
Vector v2;
v2.push(42);

Vector v3 = v1 + v2;

cout << v3.at(1) << endl;
// 42
```

```
cout <<
getRandomVector(10).at(9)
<< endl;

// vector created
// vector copied
// vector deleted
// 301989906
// vector deleted
```

```
void swap(Vector& a,
          Vector& b) {
    Vector tmp = a;
    a = b;
    b = tmp;
}
```

We are spending time to work carefully with objects which will be dead in a moment.

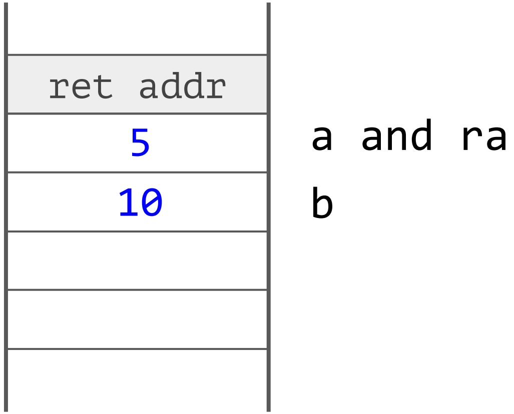
Value categories

Any expression in C++ belongs to one of the categories.

Value categories

Any expression in C++ belongs to one of the categories.

```
int a = 5;
int b = 10;

int& ra = a;
```

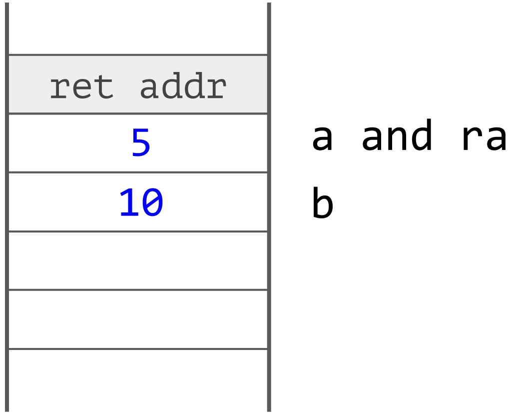


Value categories

Any expression in C++ belongs to one of the categories.

If expression refers to some named object in memory, it is lvalue expression.

```cpp
int a = 5;
int b = 10;

int& ra = a;
```



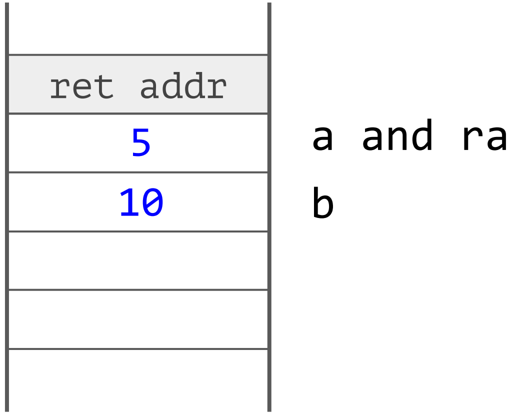
Value categories

Any expression in C++ belongs to one of the categories.

If expression refers to some named object in memory, it is lvalue expression.

```
int a = 5;
int b = 10;

int& ra = a;
```

a is lvalue



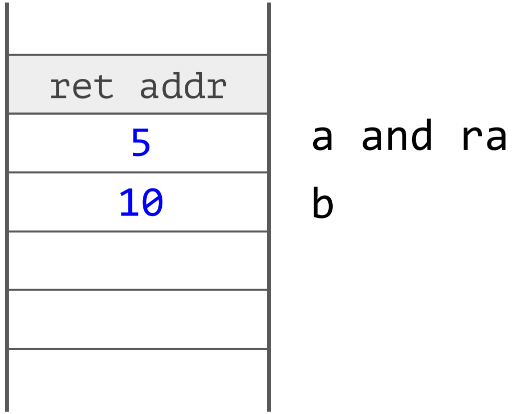
Value categories

Any expression in C++ belongs to one of the categories.

If expression refers to some named object in memory, it is lvalue expression.

```
int a = 5;
int b = 10;

int& ra = a;
ra;
```

a is lvalue



Value categories

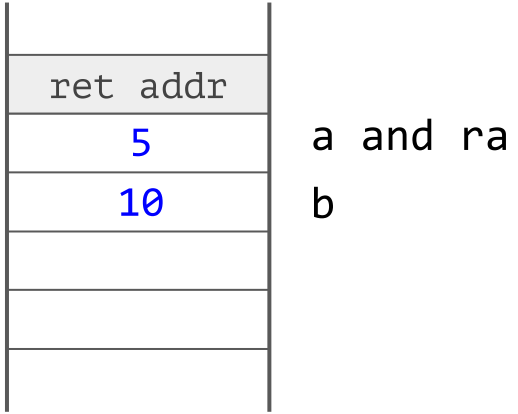
Any expression in C++ belongs to one of the categories.

If expression refers to some named object in memory, it is lvalue expression.

```
int a = 5;
int b = 10;

int& ra = a;
ra;
```

a is lvalue

ra is lvalue as well



Value categories

Any expression in C++ belongs to one of the categories.

If expression refers to some named object in memory, it is lvalue expression.

```
int a = 5;
int b = 10;

int& ra = a;
ra;
```

a is lvalue

ra is lvalue as well

```
int x = 13;

int& foo() {
    return x;
}
```

Value categories

Any expression in C++ belongs to one of the categories.

If expression refers to some named object in memory, it is lvalue expression.

```
int a = 5;
int b = 10;

int& ra = a;
ra;
foo();
```

a is lvalue

ra is lvalue as well

```
int x = 13;

int& foo() {
    return x;
}
```

Value categories

Any expression in C++ belongs to one of the categories.

If expression refers to some named object in memory, it is lvalue expression.

```
int a = 5;
int b = 10;
                    a is lvalue
int& ra = a;
ra;        ra is lvalue as well
foo();     foo() is lvalue as well
```

```
int x = 13;

int& foo() {
    return x;
}
```

Any expression in C++ belongs to one of the categories.

If expression refers to some named object in memory, it is lvalue expression.

```
int a = 5;
int b = 10;

int& ra = a;
```

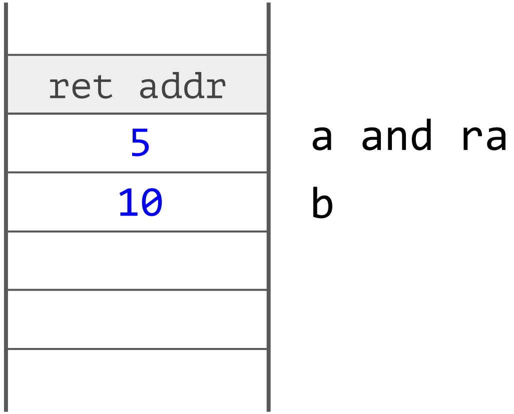
a is lvalue

Any expression in C++ belongs to one of the categories.

If expression refers to some named object in memory, it is lvalue expression.

If expression defines how to initialize an object, it is called prvalue expression.

```
int a = 5;
int b = 10;
                    a is lvalue

int& ra = a;
42;                "42" is prvalue!
```



Any expression in C++ belongs to one of the categories.

If expression refers to some named object in memory, it is lvalue expression.

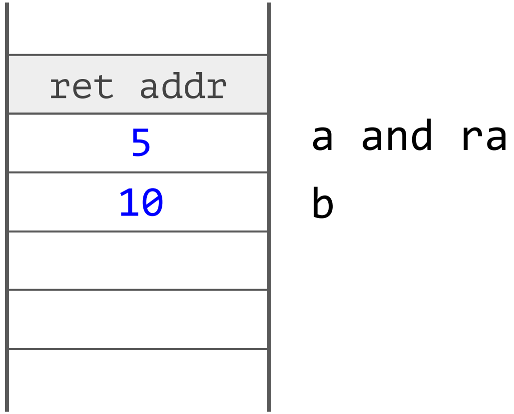
If expression defines how to initialize an object, it is called prvalue expression.

```
int a = 5;
int b = 10;
```

a is lvalue

```
int& ra = a;
a + 42;
```

"a + 42" is prvalue!



Any expression in C++ belongs to one of the categories.

If expression refers to some named object in memory, it is lvalue expression.

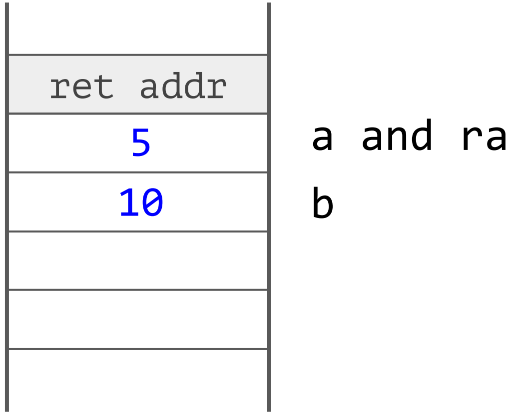
If expression defines how to initialize an object, it is called prvalue expression.

```
int a = 5;
int b = 10;

int& ra = a;
a + 42;
```

a is lvalue

"a + 42" is prvalue!

there is no such object (yet) =>
it has no name and no location



Any expression in C++ belongs to one of the categories.

If expression refers to some named object in memory, it is lvalue expression.

If expression defines how to initialize an object, it is called prvalue expression.

```
int a = 5;
int b = 10;

int& ra = a;    a is lvalue
a + 42;         "a + 42" is prvalue!
S{1};
```

```
struct S { int x; };
```

Any expression in C++ belongs to one of the categories.

If expression refers to some named object in memory, it is lvalue expression.

If expression defines how to initialize an object, it is called prvalue expression.

```
int a = 5;
int b = 10;

int& ra = a;
a + 42;
S{1};
```

a is lvalue

"a + 42" is prvalue!

"S{1}" is prvalue! No object yet, no name, no location.

```
struct S { int x; };
```

Any expression in C++ belongs to one of the categories.

If expression refers to some named object in memory, it is lvalue expression.

If expression defines how to initialize an object, it is called prvalue expression.

```
int a = 5;
int b = 10;

int& ra = a;        // a is lvalue
a + 42;             // "a + 42" is prvalue
S{1};               // "S{1}" is prvalue
```

```
struct S { int x; };

S bar() {
    S res{1};
    if (...)
        return res;
    else
        return {1};
}
```

Any expression in C++ belongs to one of the categories.

If expression refers to some named object in memory, it is lvalue expression.

If expression defines how to initialize an object, it is called prvalue expression.

```
int a = 5;
int b = 10;

int& ra = a;      // a is lvalue
a + 42;           // "a + 42" is prvalue
S{1};             // "S{1}" is prvalue
bar();            // "bar();" is prvalue
```

```
struct S { int x; };

S bar() {
    S res{1};
    if (...)
        return res;
    else
        return {1};
}
```

Any expression in C++ belongs to one of the categories.

If expression refers to some named object in memory, it is lvalue expression.

If expression defines how to initialize an object, it is called prvalue expression.

```
int a = 5;
int b = 10;

int& ra = a;      // a is lvalue
a + 42;           // "a + 42" is prvalue
S{1};             // "S{1}" is prvalue
bar();            // "bar();" is prvalue (no object, no location)
```

Any expression in C++ belongs to one of the categories.

If expression refers to some named object in memory, it is lvalue expression.

If expression defines how to initialize an object, it is called prvalue expression.

```
int a = 5;
int b = 10;
                a is lvalue
int& ra = a;
a + 42;        "a + 42" is prvalue
S{1};          "S{1}" is prvalue
bar();         "bar();" is prvalue
```

Often (but not always) prvalue expressions define temporary objects

What was in common for all examples?

```
Vector v1;
v1.push(13);
Vector v2;
v2.push(42);

Vector v3 = v1 + v2;

cout << v3.at(1) << endl;
// 42
```

```
cout <<
getRandomVector(10).at(9)
<< endl;

// vector created
// vector copied
// vector deleted
// 301989906
// vector deleted
```

We are spending time to work carefully with objects which will be dead in a moment.

What was in common for all examples?

```
Vector v1;
v1.push(13);
Vector v2;
v2.push(42);

Vector v3 = v1 + v2;

cout << v3.at(1) << endl;
// 42
```

```
cout <<
getRandomVector(10).at(9)
<< endl;

// vector created
// vector copied
// vector deleted
// 301989906
// vector deleted
```

Both "v1 + v2" and "getRandomVector(10)" are prvalue expressions!

We are spending time to work carefully with objects which will be dead in a moment.

Any expression in C++ belongs to one of the categories.

If expression refers to some named object in memory, it is lvalue expression.

If expression defines how to initialize an object, it is called prvalue expression.

---

lvalue-references can be bound to lvalues.

```
int a = 5;
int& ra = a;        lvalue-reference
a + 42;
S{1};
bar();
```

Any expression in C++ belongs to one of the categories.

If expression refers to some named object in memory, it is lvalue expression.

If expression defines how to initialize an object, it is called prvalue expression.

---

lvalue-references can be bound to lvalues only.

```
int a = 5;
int& ra = a;
int& ra2 = a + 42; // compilation error
S& rs = S{1};      // compilation error
S& rs2 = bar();    // compilation error
```

Any expression in C++ belongs to one of the categories.

If expression refers to some named object in memory, it is lvalue expression.

If expression defines how to initialize an object, it is called prvalue expression.

---

lvalue-references can be bound to lvalues only.
constant lvalue-refs can be bound to rvalues as well.

```
int a = 5;
int& ra = a;
const int& ra2 = a + 42; // OK
const S& rs = S{1};      // OK
const S& rs2 = bar();    // OK
```

They "materialize" temporary objects, now they have address, name, etc

Any expression in C++ belongs to one of the categories.

If expression refers to some named object in memory, it is lvalue expression.

If expression defines how to initialize an object, it is called prvalue expression.

---

lvalue-references can be bound to lvalues only.
constant lvalue-refs can be bound to rvalues as well.
rvalue-references can be bound to rvalues only.

```
int a = 5;
int& ra = a;
int&& ra2 = a + 42; // OK
S&& rs = S{1};      // OK
S&& rs2 = bar();    // OK
```

Any expression in C++ belongs to one of the categories.

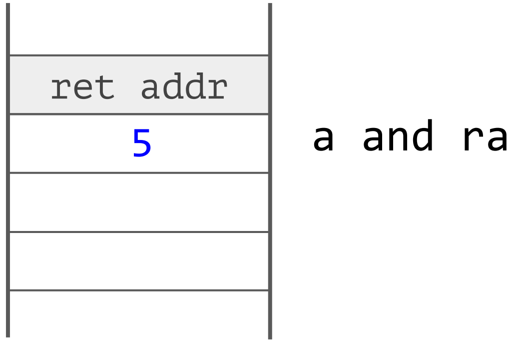
If expression refers to some named object in memory, it is lvalue expression.

If expression defines how to initialize an object, it is called prvalue expression.

---

lvalue-references can be bound to lvalues only.
constant lvalue-refs can be bound to rvalues as well.
rvalue-references can be bound to rvalues only.

```
int a = 5;
int& ra = a;
int&& ra2 = a + 42;
```



Any expression in C++ belongs to one of the categories.

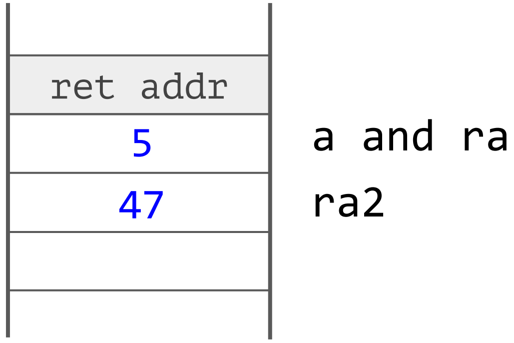
If expression refers to some named object in memory, it is lvalue expression.

If expression defines how to initialize an object, it is called prvalue expression.

---

lvalue-references can be bound to lvalues only.
constant lvalue-refs can be bound to rvalues as well.
rvalue-references can be bound to rvalues only.

```cpp
int a = 5;
int& ra = a;
int&& ra2 = a + 42;
```



Any expression in C++ belongs to one of the categories.

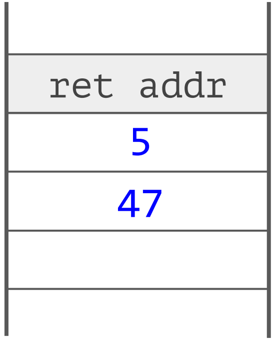
If expression refers to some named object in memory, it is lvalue expression.

If expression defines how to initialize an object, it is called prvalue expression.

---

lvalue-references can be bound to lvalues only.
constant lvalue-refs can be bound to rvalues as well.
rvalue-references can be bound to rvalues only.

```cpp
int a = 5;
int& ra = a;
int&& ra2 = a + 42;
```



a and ra

ra2

"a + 42" materialized!!!

Any expression in C++ belongs to one of the categories.

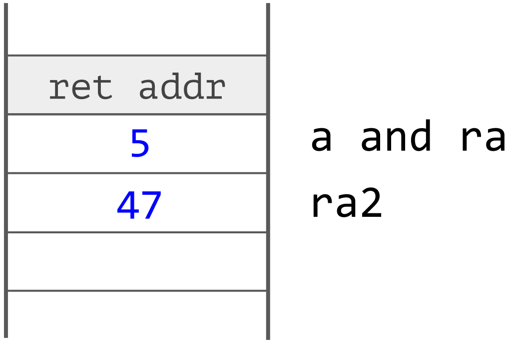
If expression refers to some named object in memory, it is lvalue expression.

If expression defines how to initialize an object, it is called prvalue expression.

---

lvalue-references can be bound to lvalues only.
constant lvalue-refs can be bound to rvalues as well.
rvalue-references can be bound to rvalues only.

```cpp
int a = 5;
int& ra = a;
int&& ra2 = a + 42;
ra2 = 13;
```



"a + 42" materialized!!!

ra2 is not constant, you can change the object.

Any expression in C++ belongs to one of the categories.

If expression refers to some named object in memory, it is lvalue expression.

If expression defines how to initialize an object, it is called prvalue expression.

---

lvalue-references can be bound to lvalues only.
constant lvalue-refs can be bound to rvalues as well.
rvalue-references can be bound to rvalues only.

```
int a = 5;
int& ra = a;
int&& ra2 = a; // compilation error
```

Any expression in C++ belongs to one of the categories.

If expression refers to some named object in memory, it is lvalue expression.

If expression defines how to initialize an object, it is called prvalue expression.

---

lvalue-references can be bound to lvalues only.
constant lvalue-refs can be bound to rvalues as well.
rvalue-references can be bound to rvalues only.

Any expression in C++ belongs to one of the categories.

If expression refers to some named object in memory, it is lvalue expression.

If expression defines how to initialize an object, it is called prvalue expression.

---

lvalue-references can be bound to lvalues only.
constant lvalue-refs can be bound to rvalues as well.
rvalue-references can be bound to rvalues only.

Last, but not the least: functions can be overloaded for both lvalue-references and rvalue-references arguments.

```
struct S { int x; };

S bar() {
    S res{1};
    if (...)
        return res;
    else
        return {1};
}
```

```
void test(S& s) {
    cout << "lvalue ref" << endl;
}

void test(const S& s) {
    cout << "const lvalue ref" << endl;
}
```

```
S s(12);

test(s);
test(S{1});
test(foo());
```

```cpp
struct S { int x; };

S bar() {
    S res{1};
    if (...)
        return res;
    else
        return {1};
}
```

```cpp
void test(S& s) {
    cout << "lvalue ref" << endl;
}

void test(const S& s) {
    cout << "const lvalue ref" << endl;
}
```

```cpp
S s(12);

test(s);        // lvalue ref
test(S{1});     // const lvalue ref
test(foo());    // const lvalue ref
```

Quite logical: we just can't bind l-value refs to temporals

```cpp
struct S { int x; };

S bar() {
    S res{1};
    if (...)
        return res;
    else
        return {1};
}
```

```cpp
void test(S& s) {
    cout << "lvalue ref" << endl;
}
void test(const S& s) {
    cout << "const lvalue ref" << endl;
}
void test(S&& s) {
    cout << "rvalue ref" << endl;
}
```

```cpp
S s(12);

test(s);       // lvalue ref
test(S{1});    // ???
test(foo());   // ???
```

```cpp
struct S { int x; };

S bar() {
    S res{1};
    if (...)
        return res;
    else
        return {1};
}
```

```cpp
void test(S& s) {
    cout << "lvalue ref" << endl;
}
void test(const S& s) {
    cout << "const lvalue ref" << endl;
}
void test(S&& s) {
    cout << "rvalue ref" << endl;
}
```

```cpp
S s(12);

test(s);        // lvalue ref
test(S{1});     // rvalue ref
test(foo());    // rvalue ref
```

rvalue refs have a higher priority than const lvalue refs

Any expression in C++ belongs to one of the categories.

If expression refers to some named object in memory, it is lvalue expression.

If expression defines how to initialize an object, it is called prvalue expression.

---

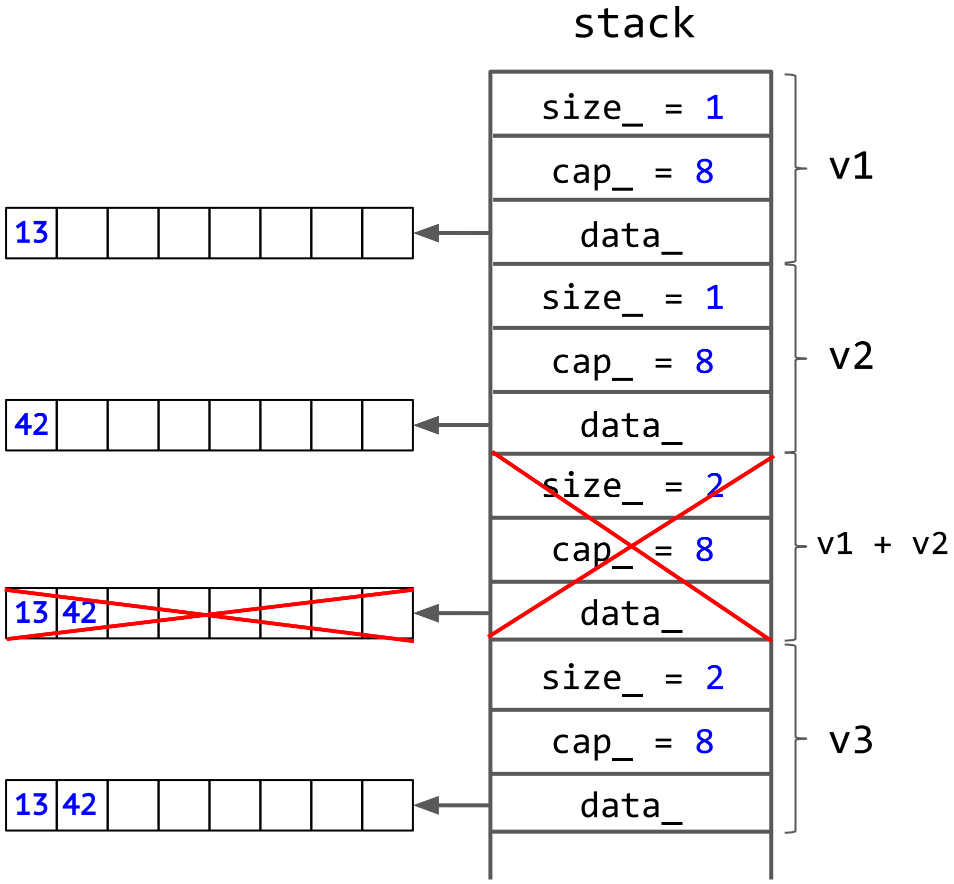
lvalue-references can be bound to lvalues only.
constant lvalue-refs can be bound to rvalues as well.
rvalue-references can be bound to rvalues only.

Last, but not the least: functions can be overloaded for both lvalue-references and rvalue-references arguments. And rvalue-references have higher priority.

```
Vector v1;
v1.push(13);
Vector v2;
v2.push(42);

Vector v3 = v1 + v2;

cout << v3.at(1) << endl;
// 42
```

Everything works correctly (thanks to the rule of 3), but...

a lot of work to create and initialize temporary object which is only used to initialize v3!



```
class Vector {
   size_t size_;
   size_t cap_;
   int *data_;
public:
   ...
   Vector(const Vector& other): size_(other.size_), cap_(other.cap_) {
      data_ = new int[cap_];
      for (int i = 0; i < size_; i++) {
         data_[i] = other.data_[i];
      }
   }
   ...
};
```

copy constructor, used to initialize v3 from previous slide

```cpp
class Vector {
   size_t size_;
   size_t cap_;
   int *data_;
public:
   ...
   Vector(const Vector& other): size_(other.size_), cap_(other.cap_) {
      data_ = new int[cap_];
      for (int i = 0; i < size_; i++) {
         data_[i] = other.data_[i];
      }
   }

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

   ...
};
```

copy constructor, used to initialize v3 from previous slide

move constructor.

```cpp
class Vector {
   size_t size_;
   size_t cap_;
   int *data_;
public:
   ...
   Vector(const Vector& other): size_(other.size_), cap_(other.cap_) {
      data_ = new int[cap_];
      for (int i = 0; i < size_; i++) {
         data_[i] = other.data_[i];
      }
   }

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

   ...
};
```

copy constructor, used to initialize v3 from previous slide

move constructor. The argument is an rvalue => temporary object

```cpp
class Vector {
    size_t size_;
    size_t cap_;
    int *data_;
public:
    ...
    Vector(const Vector& other): size_(other.size_), cap_(other.cap_) {
        data_ = new int[cap_];
        for (int i = 0; i < size_; i++) {
            data_[i] = other.data_[i];
        }
    }

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

    ...
};
```

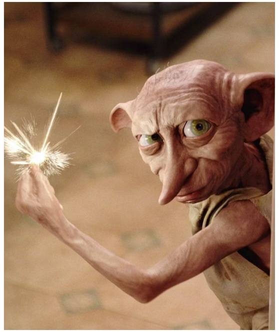
copy constructor, used to initialize v3 from previous slide

move constructor. The argument is an rvalue => temporary object

It is almost a dead man!
What can we do with him?

```
class Vector {
   size_t size_;
   size_t cap_;
   int *data_;
public:
   ...
   Vector(const Vector& other): size_(other.size_), cap_(other.cap_) {
      data_ = new int[cap_];
      for (int i = 0; i < size_; i++) {
         data_[i] = other.data_[i];
      }
   }

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

   ...
};
```

copy constructor, used to initialize v3 from previous slide



move constructor. The argument is an rvalue => temporary object

It is almost a dead man!
What can we do with him?

We can steal memory from him!

```
class Vector {
   size_t size_;
   size_t cap_;
   int *data_;
public:
   ...
   Vector(const Vector& other): size_(other.size_), cap_(other.cap_) {
      data_ = new int[cap_];
      for (int i = 0; i < size_; i++) {
         data_[i] = other.data_[i];
      }
   }

   Vector(Vector&& other): size_(other.size_), cap_(other.cap_) {
      ...
   }

   ...
};
```

copy constructor, used to initialize v3 from previous slide

```
class Vector {
   size_t size_;
   size_t cap_;
   int *data_;
public:
   ...
   Vector(const Vector& other): size_(other.size_), cap_(other.cap_) {
      data_ = new int[cap_];
      for (int i = 0; i < size_; i++) {
         data_[i] = other.data_[i];
      }
   }

   Vector(Vector&& other): size_(other.size_), cap_(other.cap_) {
      data_ = other.data_;
      ...
   }

   ...
};
```

copy constructor, used to initialize v3 from previous slide

```
class Vector {
   size_t size_;
   size_t cap_;
   int *data_;
public:
   ...
   Vector(const Vector& other): size_(other.size_), cap_(other.cap_) {
      data_ = new int[cap_];
      for (int i = 0; i < size_; i++) {
         data_[i] = other.data_[i];
      }
   }

   Vector(Vector&& other): size_(other.size_), cap_(other.cap_) {
      data_ = other.data_;
      other.data_ = nullptr;
   }

   ...
};
```

copy constructor, used to initialize v3 from previous slide

```
class Vector {
    size_t size_;
    size_t cap_;
    int *data_;
public:
    ...
    Vector(const Vector& other): size_(other.size_), cap_(other.cap_) {
        data_ = new int[cap_];
        for (int i = 0; i < size_; i++) {
            data_[i] = other.data_[i];
        }
    }

    Vector(Vector&& other): size_(other.size_), cap_(other.cap_) {
        data_ = other.data_;
        other.data_ = nullptr;
    }

    ...
};
```

copy constructor, used to initialize v3 from previous slide

Why should we do this?

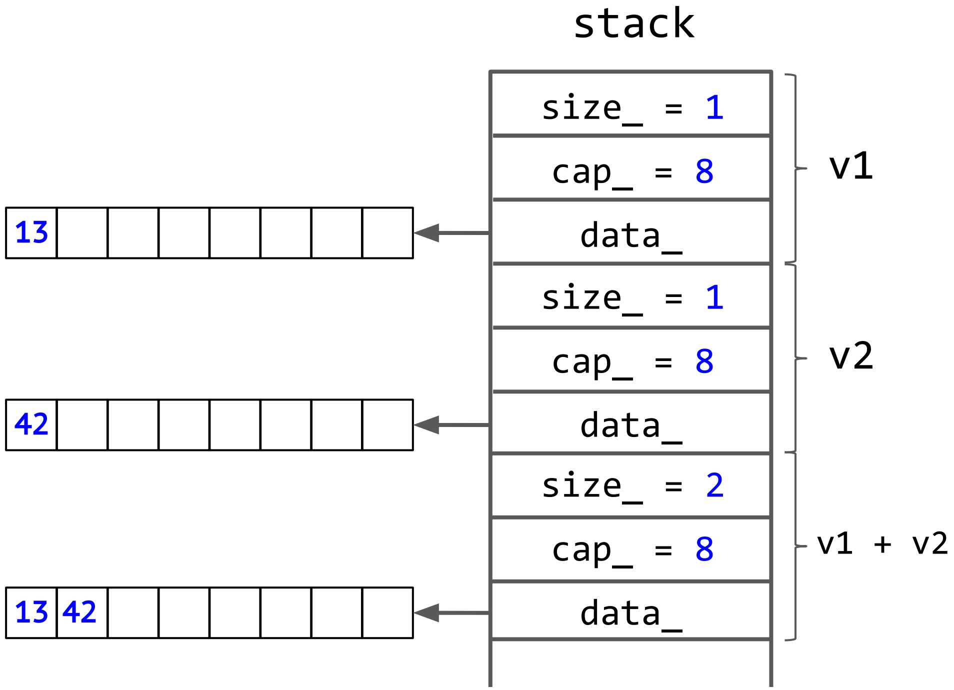
```
class Vector {
   size_t size_;
   size_t cap_;
   int *data_;
public:
   ...
   Vector(const Vector& other): size_(other.size_), cap_(other.cap_) {
      data_ = new int[cap_];
      for (int i = 0; i < size_; i++) {
         data_[i] = other.data_[i];
      }
   }

   Vector(Vector&& other): size_(other.size_), cap_(other.cap_) {
      data_ = other.data_;
      other.data_ = nullptr;
   }

   ...
};
```

copy constructor, used to initialize v3 from previous slide

Why should we do this?

Because otherwise, the destructor of temporary object will free the corresponding memory.

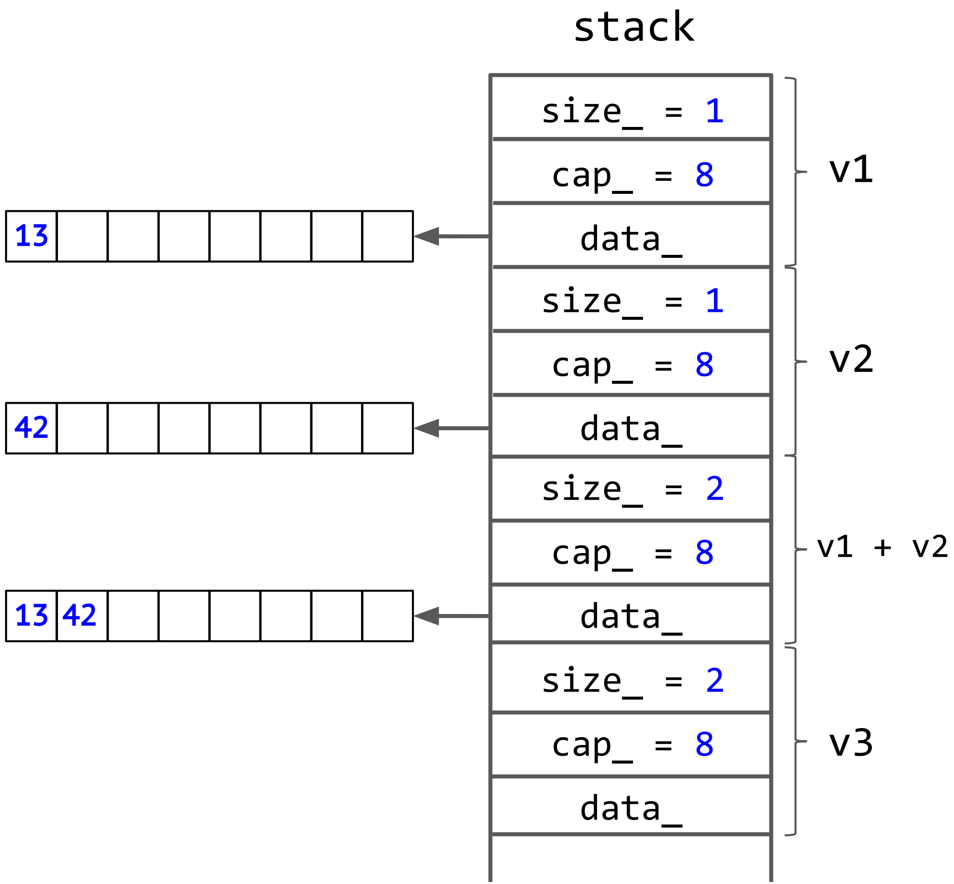
```
Vector v1;
v1.push(13);
Vector v2;
v2.push(42);

Vector v3 = v1 + v2;

cout << v3.at(1) << endl;
// 42
```



The result of v1 + v2 is a temporary object that is alive till the end of full expression (;)

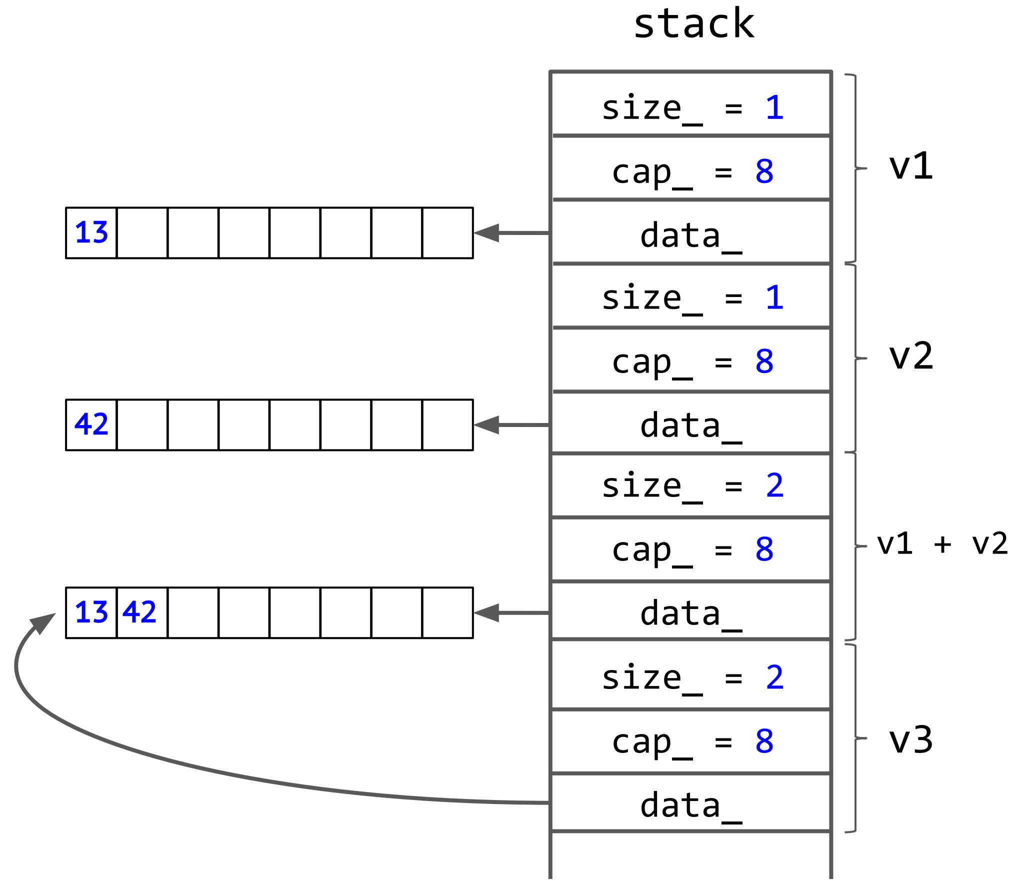
```
Vector v1;
v1.push(13);
Vector v2;
v2.push(42);

Vector v3 = v1 + v2;

cout << v3.at(1) << endl;
// 42
```

```
Vector(Vector&& other):
              size_(other.size_),
              cap_(other.cap_) {

   data_ = other.data_;
   other.data_ = nullptr;
}
```



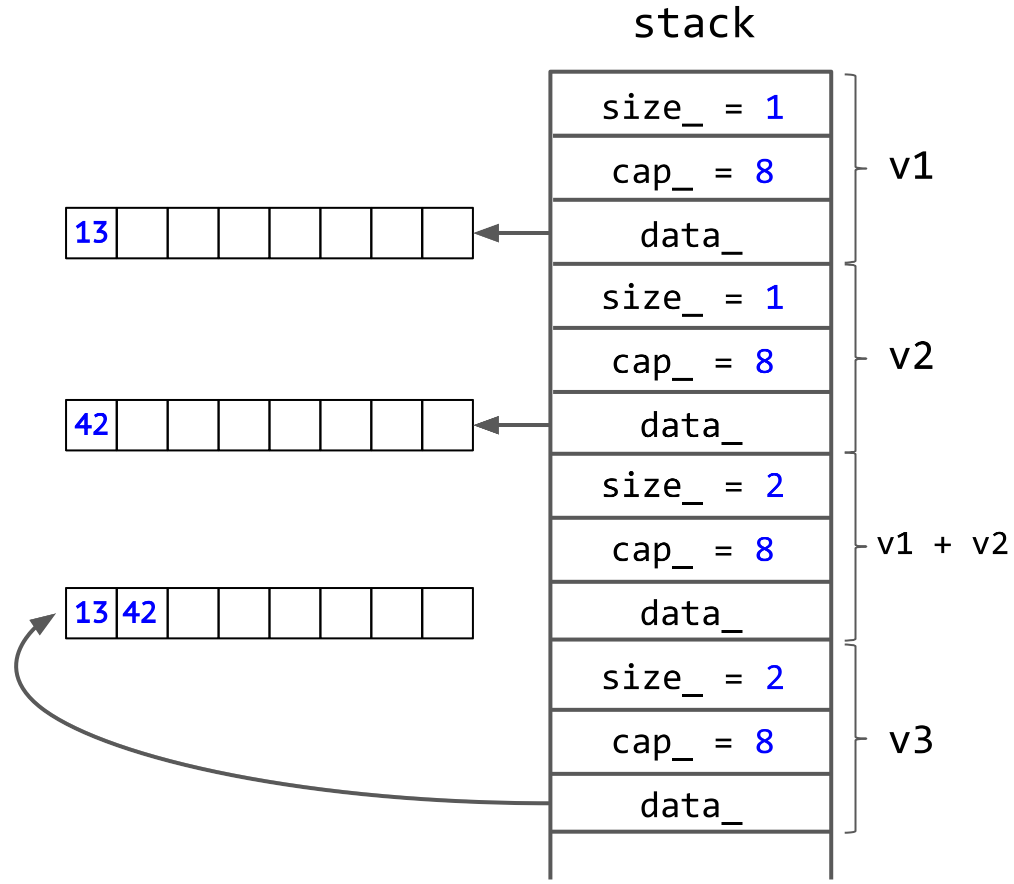
```cpp
Vector v1;
v1.push(13);
Vector v2;
v2.push(42);

Vector v3 = v1 + v2;

cout << v3.at(1) << endl;
// 42
```

```cpp
Vector(Vector&& other):
              size_(other.size_),
              cap_(other.cap_) {

   data_ = other.data_;
   other.data_ = nullptr;
}
```



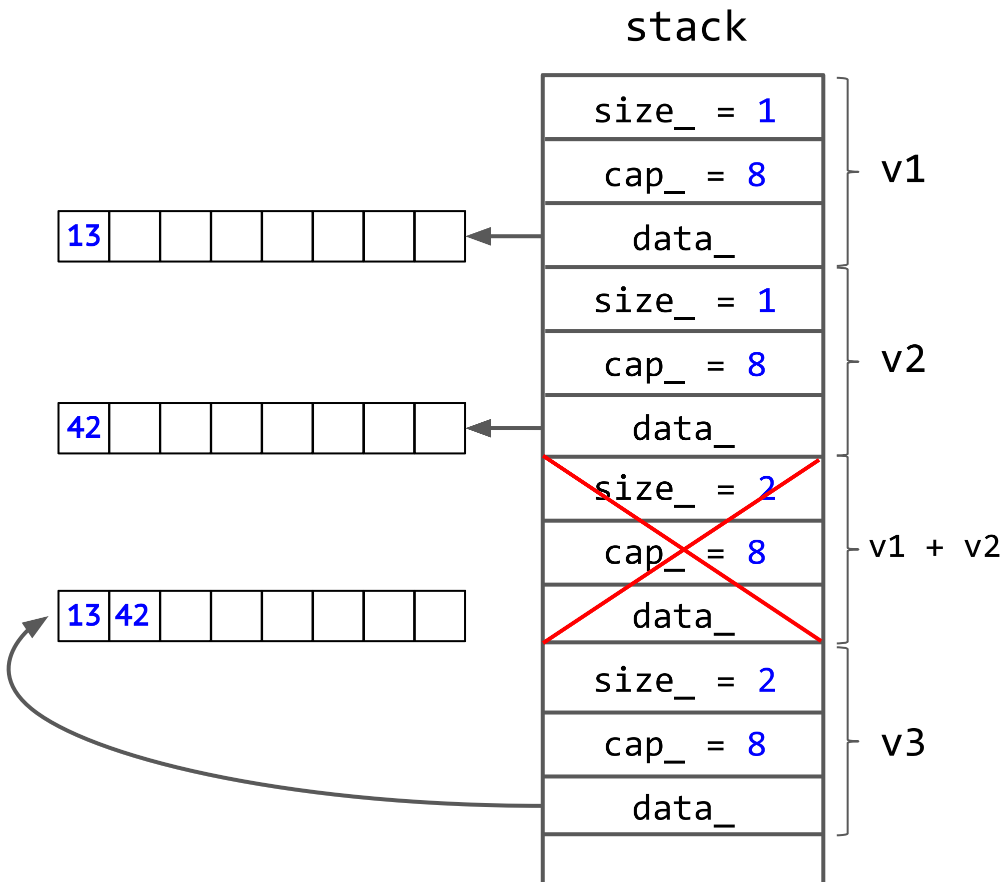
```cpp
Vector v1;
v1.push(13);
Vector v2;
v2.push(42);

Vector v3 = v1 + v2;

cout << v3.at(1) << endl;
// 42
```

```cpp
Vector(Vector&& other):
             size_(other.size_),
             cap_(other.cap_) {

   data_ = other.data_;
   other.data_ = nullptr;
}
```



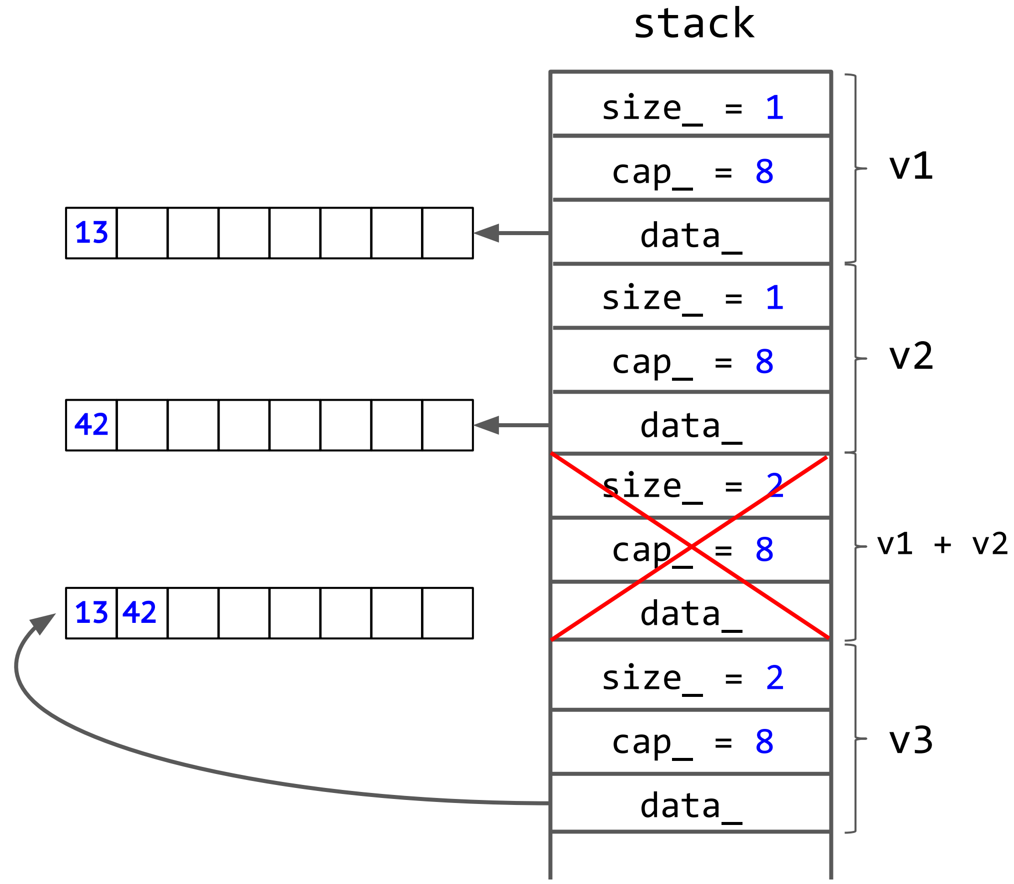
```
Vector v1;
v1.push(13);
Vector v2;
v2.push(42);

Vector v3 = v1 + v2;

cout << v3.at(1) << endl;
// 42
```

```
Vector(Vector&& other):
            size_(other.size_),
            cap_(other.cap_) {

   data_ = other.data_;
   other.data_ = nullptr;
}
```



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

Vector v3 = v1 + v2;

cout << v3.at(1) << endl;
// 42
```

```cpp
Vector(Vector&& other):
                size_(other.size_),
                cap_(other.cap_) {

    data_ = other.data_;
    other.data_ = nullptr;
}
```



We've just saved 1 allocation, a deallocation and 1 copying of the array. That's a lot.

Another example

Imagine we have debug prints in constructor, copy constructor and destructor. What will be printed?

```
cout <<
getRandomVector(10).at(9)
<< endl;

// vector created
// vector copied
// vector deleted
// 301989906
// vector deleted
```

2 allocations, 1 copying, 2 deleting.
Isn't it too heavy for such sample?

```
Vector getRandomVector(size_t max_size) {
    if (max_size == 0) {
        return Vector{};
    }

    Vector result{max_size};
    std::random_device rd;
    std::mt19937_64 gen(rd());
    std::uniform_int_distribution<int> dis;

    for (size_t i = 0; i < max_size; i++) {
        result.at(i) = dis(gen);
    }
    return result;
}
```

Another example

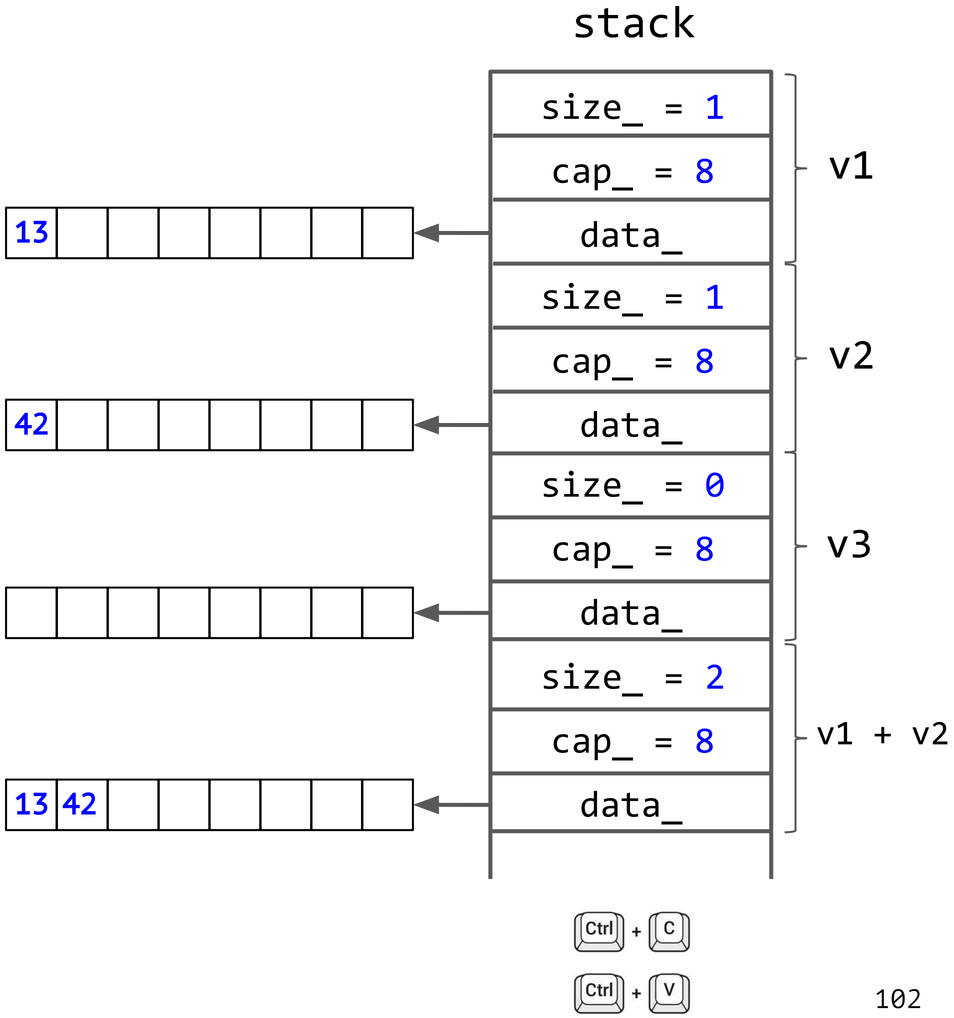
Imagine we have debug prints in constructor, copy constructor and destructor. What will be printed?

```cpp
cout <<
getRandomVector(10).at(9)
<< endl;

// vector created
// vector moved
// vector deleted
// 301989906
// vector deleted
```

With move constructor it will be 1 allocation, no copying and 1 deleting (one more for nullptr).

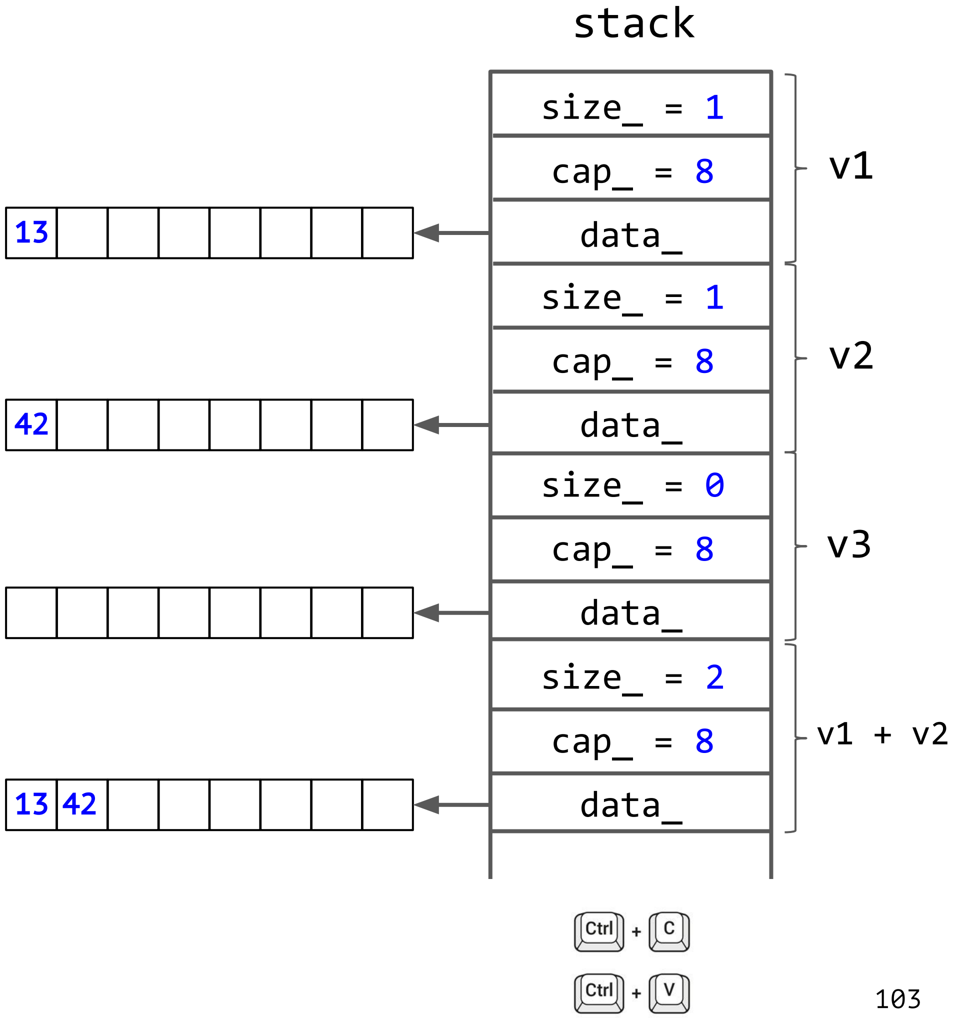
```cpp
Vector getRandomVector(size_t max_size) {
    if (max_size == 0) {
        return Vector{};
    }

    Vector result{max_size};
    std::random_device rd;
    std::mt19937_64 gen(rd());
    std::uniform_int_distribution<int> dis;

    for (size_t i = 0; i < max_size; i++) {
        result.at(i) = dis(gen);
    }
    return result;
}
```

```
Vector v1;
v1.push(13);
Vector v2;
v2.push(42);

Vector v3;
v3 = v1 + v2;

cout << v3.at(1) << endl;
// 42
```



```
Vector v1;
v1.push(13);
Vector v2;
v2.push(42);

Vector v3;
v3 = v1 + v2;

cout << v3.at(1) << endl;
// 42
```

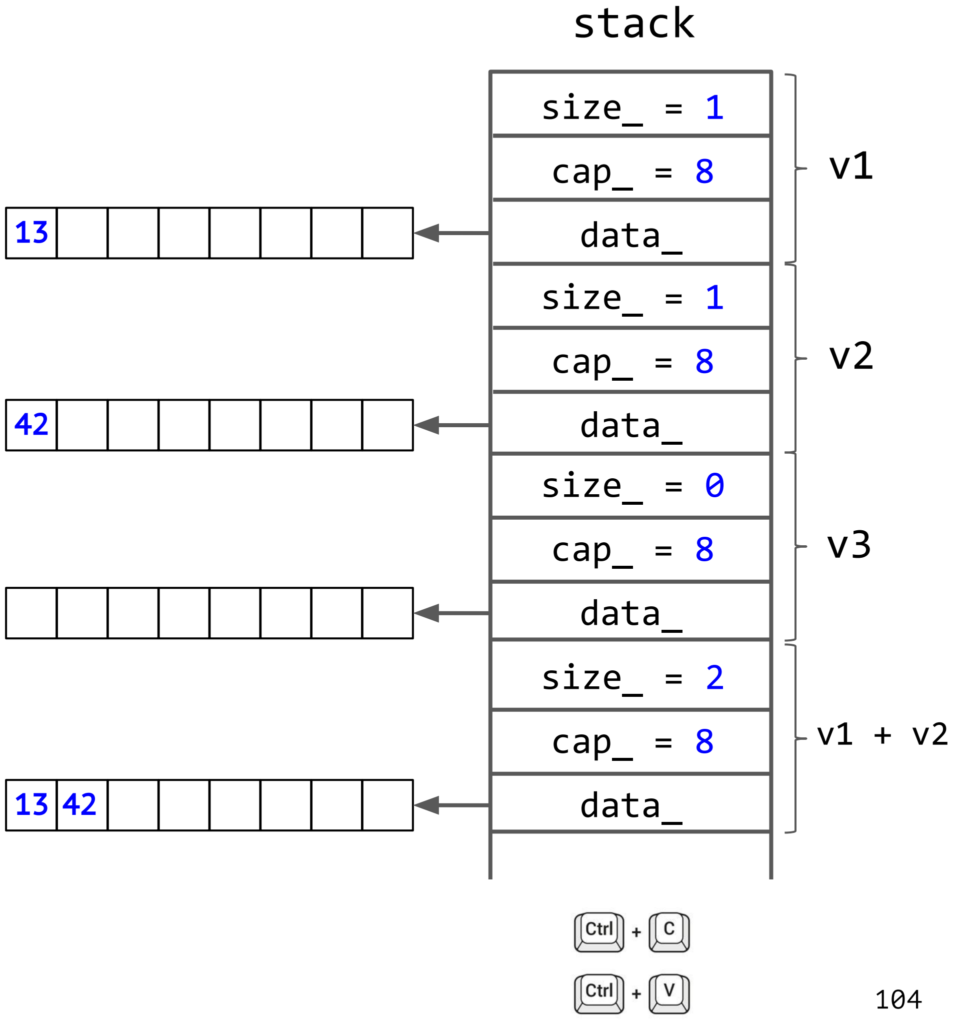
What about assignment operator?
Aren't we in the same situation here?



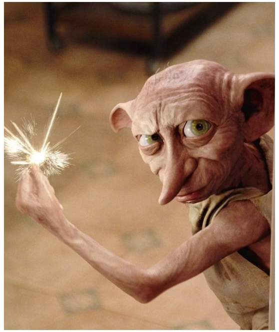
```cpp
Vector v1;
v1.push(13);
Vector v2;
v2.push(42);

Vector v3;
v3 = v1 + v2;

cout << v3.at(1) << endl;
// 42
```

What about assignment operator?
Aren't we in the same situation here?

Should we copy a data from tmp obj again?



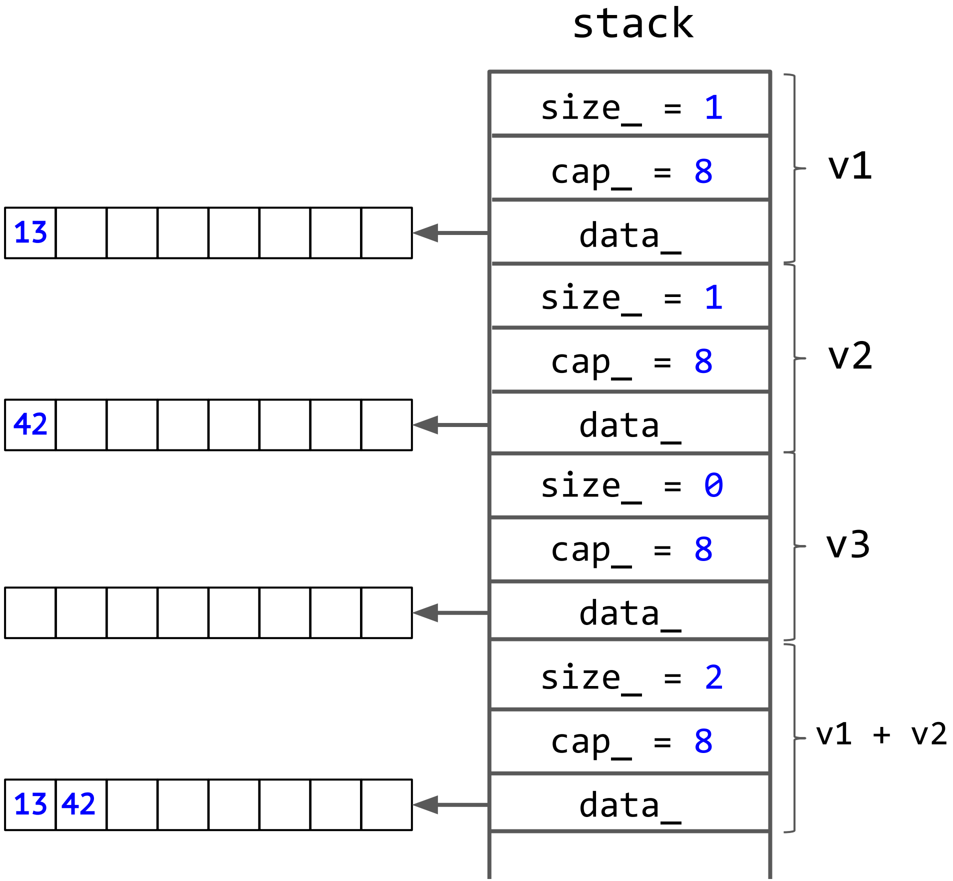
```
Vector v1;
v1.push(13);
Vector v2;
v2.push(42);
```

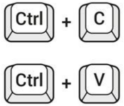


```
Vector v3;
v3 = v1 + v2;

cout << v3.at(1) << endl;
// 42
```

What about assignment operator?
Aren't we in the same situation here?

Should we copy a data from tmp obj again? No, we should steal again!





```
class Vector {
   size_t size_;
   size_t cap_;
   int *data_;
public:
   ...
   Vector& operator=(const Vector& other) {
      if (this == &other) {
         size_ = other.size_;
         capacity_ = other.capacity_;

         delete[] data_;
         data_ = new int[capacity_];

         for (int i = 0; i < size_; i++) {
            data_[i] = other.data_[i];
         }
         return *this;
      }
   }
};
```

copy assignment operator

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

   Vector& operator=(Vector&& other) {
      if (this != &other) {
         size_ = other.size_;
         capacity_ = other.capacity_;

         delete[] data_;
         data_ = other.data_;
         other.data_ = nullptr;
      }
      return *this;
   }

};
```

copy assignment operator

move assignment operator

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

   Vector& operator=(Vector&& other) {
      if (this != &other) {
         size_ = other.size_;
         capacity_ = other.capacity_;

         delete[] data_;
         data_ = other.data_;
         other.data_ = nullptr;
      }
      return *this;
   }

};
```

copy assignment operator

move assignment operator

Again: instead of allocating new memory and copying of it we just steal it from a dead man.

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

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

};
```

If you have one of these, you should have all 3 of them.

This is rule of 3.

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

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

};
```

If you have one of these, you should have all 5 of them.

This is rule of 5.

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

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

};
```

If you have one of these, you should have all 5 of them.

This is rule of 5.

You can think that while rule of 3 is about correctness, rule of 5 is more about performance improvements.

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

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

};
```

If you have one of these, you should have all 5 of them.

This is rule of 5.

You can think that while rule of 3 is about correctness, rule of 5 is more about performance improvements. Without them copy constructor /assignment operator will be used for temporals.

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

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

};
```

If you have one of these, you should have all 5 of them.

This is rule of 5.

Not quite, will discuss later.

You can think that while rule of 3 is about correctness, rule of 5 is more about performance improvements. Without them copy constructor /assignment operator will be used for temporals.

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

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

};
```

But we can do even better.

If you have one of these, you should have all 5 of them.

This is rule of 5.

Not quite, will discuss later.

You can think that while rule of 3 is about correctness, rule of 5 is more about performance improvements. Without them copy constructor /assignment operator will be used for temporals.

But we can do even better.

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

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

};
```

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

   Vector& operator=(Vector other) {
      std::swap(size_, other.size_);
      std::swap(cap_, other.cap_);
      std::swap(data_, other.data_);
      return *this;
   }

};
```

But we can do even better.

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

    Vector& operator=(Vector other) {
        std::swap(size_, other.size_);
        std::swap(cap_, other.cap_);
        std::swap(data_, other.data_);
        return *this;
    }

};
```

But we can do even better.

We have both copy and move constructors.

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

    Vector& operator=(Vector other) {
        std::swap(size_, other.size_);
        std::swap(cap_, other.cap_);
        std::swap(data_, other.data_);
        return *this;
    }

};
```

But we can do even better.

We have both copy and move constructors.

Depending on the argument, one of them will be used to initialize local copy other.

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

    Vector& operator=(Vector other) {
        std::swap(size_, other.size_);
        std::swap(cap_, other.cap_);
        std::swap(data_, other.data_);
        return *this;
    }

};
```

But we can do even better.

We have both copy and move constructors.

Depending on the argument, one of them will be used to initialize local copy other.

Than we swap data_ of this and local copy other. Why?

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

    Vector& operator=(Vector other) {
        std::swap(size_, other.size_);
        std::swap(cap_, other.cap_);
        std::swap(data_, other.data_);
        return *this;
    }

};
```

But we can do even better.

We have both copy and move constructors.

Depending on the argument, one of them will be used to initialize local copy other.

Than we swap data_ of this and local copy other. Why? Because our old memory will be freed in the destructor of old copy (and we will get new one)

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

    Vector& operator=(Vector other) {
        std::swap(size_, other.size_);
        std::swap(cap_, other.cap_);
        std::swap(data_, other.data_);
        return *this;
    }

};
```

But we can do even better.

This is called
copy-and-swap idiom.

We have both copy and move constructors.

Depending on the argument, one of them will be used to initialize local copy other.

Than we swap data_ of this and local copy other. Why? Because our old memory will be freed in the destructor of old copy (and we will get new one)

One more motivation example

```
template<typename T>
void swap(T& a, T& b) {
    T tmp = a;
    a = b;
    b = tmp;
}
```

Classical use case for references

You can use this swap for Vectors, but it will work terribly slow because of copy ctr, additional allocations and deallocations.

One more motivation example

```
template<typename T>
void swap(T& a, T& b) {
   T tmp = a;
   a = b;
   b = tmp;
}
```

Classical use case for references

It would be nice to use move constructor here to create tmp (steal from a) and than use move assign operator twice.

One more motivation example

```
template<typename T>
void swap(T& a, T& b) {
    T tmp = a;
    a = b;
    b = tmp;
}
```

Classical use case for references

It would be nice to use move constructor here to create tmp (steal from a) and than use move assign operator twice.

But how? a, b and tmp are lvalue expressions!

One more motivation example

```
template<typename T>
void swap(T& a, T& b) {
    T tmp = a;
    a = b;
    b = tmp;
}
```

Classical use case for references

It would be nice to use move constructor here to create tmp (steal from a) and than use move assign operator twice.

But how? a, b and tmp are lvalue expressions!

Here were black magic appears.

One more motivation example

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

Classical use case for references

It would be nice to use move constructor here to create tmp (steal from a) and than use move assign operator twice.

But how? a, b and tmp are lvalue expressions!

Here were black magic appears.

One more motivation example

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

std::move is very simple thing: it just casts any expression to rvalue reference!

Classical use case for references

It would be nice to use move constructor here to create tmp (steal from a) and than use move assign operator twice.

But how? a, b and tmp are lvalue expressions!

Here were black magic appears.

One more motivation example

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

std::move is very simple thing: it just casts any expression to rvalue reference!

```
Vector a;
Vector&& rra = std::move(a);
// no compilation errors!
```

Classical use case for references

It would be nice to use move constructor here to create tmp (steal from a) and than use move assign operator twice.

But how? a, b and tmp are lvalue expressions!

Here were black magic appears.

One more motivation example

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

std::move is very simple thing: it just casts any expression to rvalue reference!

And by doing this it allows to move from its argument!

Classical use case for references

It would be nice to use move constructor here to create tmp (steal from a) and than use move assign operator twice.

But how? a, b and tmp are lvalue expressions!

Here were black magic appears.

One more motivation example

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

std::move is very simple thing: it just casts any expression to rvalue reference!

And by doing this it allows to move from its argument!

So, move ctr and assign op will work.

Classical use case for references

It would be nice to use move constructor here to create tmp (steal from a) and than use move assign operator twice.

But how? a, b and tmp are lvalue expressions!

Here were black magic appears.

And one more motivation example

```
struct PairOfVectors {
    Vector vec1;
    Vector vec2;

    PairOfVectors(const Vector& v1, const Vector& v2): vec1(v1), vec2(v2) {}

    PairOfVectors (PairOfVectors&& other) {
        this->vec1 = other.vec1;
        this->vec2 = other.vec2;
    }
};
```

And one more motivation example

```
struct PairOfVectors {
    Vector vec1;
    Vector vec2;

    PairOfVectors(const Vector& v1, const Vector& v2): vec1(v1), vec2(v2) {}

    PairOfVectors (PairOfVectors&& other) {
        this->vec1 = other.vec1;
        this->vec2 = other.vec2;
    }
};
```

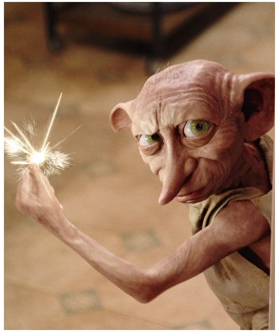
Ok, we are stealing from PairOfVectors, but what about fields? Looks like we are still copying them! How to fix?

And one more motivation example

```
struct PairOfVectors {
    Vector vec1;
    Vector vec2;

    PairOfVectors(const Vector& v1, const Vector& v2): vec1(v1), vec2(v2) {}

    PairOfVectors (PairOfVectors&& other) {
        this->vec1 = std::move(other.vec1);
        this->vec2 = std::move(other.vec2);
    }
};
```

Ok, we are stealing from PairOfVectors, but what about fields? Looks like we are still copying them! How to fix?

Ok, now we are stealing fields as well!



Beware of std::move!

Beware of std::move!

```
Vector a{16};
a.push(13);

Vector b = std::move(a);

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

Beware of std::move!

```
Vector a{16};
a.push(13);

Vector b = std::move(a);    <-----    What will happen here?

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

Beware of std::move!

```
Vector a{16};
a.push(13);

Vector b = std::move(a);

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

What will happen here?

1. a is casted to Vector&&
2. move constructor is called
3. memory from a is stolen (now used by b)
4. a.data_ is nullptr

Beware of std::move!

```
Vector a{16};
a.push(13);

Vector b = std::move(a);

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

What will happen here?

1. a is casted to Vector&&
2. move constructor is called
3. memory from a is stolen (now used by b)
4. a.data_ is nullptr

UB

Beware of std::move!

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

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

};
```

Beware of std::move!

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

   Vector(Vector&& other) ... { ... }
   Vector& operator=(Vector&& other) = default;

};
```

Beware of std::move!

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

    Vector(Vector&& other) ... { ... }
    Vector& operator=(Vector&& other) = default;

};
```

Or just like this
(in such case default move assignment operator will be generated by the compiler)

Beware of std::move!

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

   Vector(Vector&& other) ... { ... }
   Vector& operator=(Vector&& other) = default;
};
```

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

How this will work?

Beware of std::move!

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

    Vector(Vector&& other) ... { ... }
    Vector& operator=(Vector&& other) = default;
};
```

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

How this will work?

1. Default move assign oper just tries to move assign each field

Beware of std::move!

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

    Vector(Vector&& other) ... { ... }
    Vector& operator=(Vector&& other) = default;
};
```

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

How this will work?

1. Default move assign oper just tries to move assign each field
2. In case of primitives it is copying

Beware of std::move!

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

   Vector(Vector&& other) ... { ... }
   Vector& operator=(Vector&& other) = default;
};
```

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

UB

How this will work?

1. Default move assign oper just tries to move assign each field

2. In case of primitives it is copying

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

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

};
```

But we can do even better.

If you have one of these, you should have all 5 of them.

This is rule of 5.

Not quite, will discuss later.

You can think that while rule of 3 is about correctness, rule of 5 is more about performance improvements. Without them copy constructor /assignment operator will be used for temporals.

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

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

};
```

But we can do even better.

If you have one of these, you should have all 5 of them.

This is rule of 5.

You can think that while rule of 3 is about correctness, rule of 5 is more about performance improvements. But actually no, you can shoot your feet if you break rule of 5 as well.

What was in common for all examples?

```
Vector v1;
v1.push(13);
Vector v2;
v2.push(42);

Vector v3 = v1 + v2;

cout << v3.at(1) << endl;
// 42
```

```
cout <<
getRandomVector(10).at(9)
<< endl;

// vector created
// vector copied
// vector deleted
// 301989906
// vector deleted
```

```
void swap(Vector& a,
          Vector& b) {
    Vector tmp = a;
    a = b;
    b = tmp;
}
```

We are spending time to work carefully with objects which will be dead in a moment.

What was in common for all examples?

```
Vector v1;
v1.push(13);
Vector v2;
v2.push(42);

Vector v3 = v1 + v2;

cout << v3.at(1) << endl;
// 42
```

```
cout <<
getRandomVector(10).at(9)
<< endl;

// vector created
// vector copied
// vector deleted
// 301989906
// vector deleted
```

```
void swap(Vector& a,
          Vector& b) {
    Vector tmp =
           std::move(a);
    a = std::move(b);
    b = std::move(tmp);
}
```

We are spending time to work carefully with objects which will be dead in a moment.

Value categories

Any expression in C++ belongs to one of the categories.

If expression refers to some named object in memory, it is lvalue expression.

If expression defines how to initialize an object, it is called prvalue expression.

Value categories

Any expression in C++ belongs to one of the categories.

If expression refers to some named object in memory, it is lvalue expression.

If expression defines how to initialize an object, it is called prvalue expression.

Actually it is a bit more complicated, and now you are ready to understand the whole picture!

Value categories

Any expression in C++ has two properties:

Value categories

Any expression in C++ has two properties:

1) Whether it refers to something with identity or not.

Value categories

Any expression in C++ has two properties:

1) Whether it refers to something with identity or not. It means: can you understand by two such expressions whether they refer to the same object or not.

Value categories

Any expression in C++ has two properties:

1) Whether it refers to something with identity or not. It means: can you understand by two such expressions whether they refer to the same object or not.

   Example #1:

```
int v = 13;
int& pv = v;
v;   <——  both v and pv refers to
pv;  <——  object with identity
```

Value categories

Any expression in C++ has two properties:

1) Whether it refers to something with identity or not. It means: can you understand by two such expressions whether they refer to the same object or not.

Example #2:

```
int v = 13;
int v2 = v + 3;
int v3 = v + v2;
```

both 13, v + 3 and v + v2 doesn't actually refer to objects (instead they define how to create them)

Value categories

Any expression in C++ has two properties:

1) Whether it refers to something with identity or not. It means: can you understand by two such expressions whether they refer to the same object or not.

2) Can or can not be moved from such expression.

Value categories

Any expression in C++ has two properties:

1) Whether it refers to something with identity or not. It means: can you understand by two such expressions whether they refer to the same object or not.

2) Can or can not be moved from such expression. Can this expression be bound to the argument of move ctr or move assignment operator.

Value categories

Any expression in C++ has two properties:

2) Can or can not be moved from such expression. Can this expression be bound to the argument of move ctr or move assignment operator.

Example #1:

```
Vector v1{13};
Vector v2 = v1;
```

Value categories

Any expression in C++ has two properties:

2) Can or can not be moved from such expression. Can this expression be bound to the argument of move ctr or move assignment operator.

Example #1:

```
Vector v1{13};
Vector v2 = v1;
```

v1 can not be moved from

Value categories

Any expression in C++ has two properties:

2) Can or can not be moved from such expression. Can this expression be bound to the argument of move ctr or move assignment operator.

Example #2:

```
Vector v1{13};
Vector v2 = v1;       v1 can not be moved from

Vector v3 = v1 + v2;
```

Value categories

Any expression in C++ has two properties:

2) Can or can not be moved from such expression. Can this expression be bound to the argument of move ctr or move assignment operator.

Example #2:

```
Vector v1{13};
Vector v2 = v1;
```

v1 can not be moved from

```
Vector v3 = v1 + v2;
```

v1 + v2 produces tmp object, so it can be moved from

Value categories

Any expression in C++ has two properties:

2) Can or can not be moved from such expression. Can this expression be bound to the argument of move ctr or move assignment operator.

Example #3:

```
Vector v1{13};
Vector v2 = std::move(v1);
```

std::move(v1) can be moved from!

Value categories

Any expression in C++ has two properties:

1) Whether refers to something with identity or not.
2) Can or can not be moved from such expression.

Value categories

Any expression in C++ has two properties:

1) Whether refers to something with identity or not.
2) Can or can not be moved from such expression.

Any expression in C++ belongs to one of the categories:

Value categories

Any expression in C++ has two properties:

1) Whether refers to something with identity or not.
2) Can or can not be moved from such expression.

Any expression in C++ belongs to one of the categories:

1. lvalues: refers to identity/can't be moved from

Value categories

Any expression in C++ has two properties:

1) Whether refers to something with identity or not.
2) Can or can not be moved from such expression.

Any expression in C++ belongs to one of the categories:

1. lvalues: refers to identity/can't be moved from

   Examples: variable names, lvalue and rvalue references, call of a function that returns lvalue reference, …

Value categories

Any expression in C++ has two properties:

1) Whether refers to something with identity or not.
2) Can or can not be moved from such expression.

Any expression in C++ belongs to one of the categories:

1. lvalues: refers to identity/can’t be moved from
2. prvalues: no identity/ can be moved from

Value categories

Any expression in C++ has two properties:

1) Whether refers to something with identity or not.
2) Can or can not be moved from such expression.

Any expression in C++ belongs to one of the categories:

1. lvalues: refers to identity/can’t be moved from
2. prvalues: no identity/ can be moved from

Examples: literals (except strings), call of a function that returns by value, call of ctrs, etc

Value categories

Any expression in C++ has two properties:

1) Whether refers to something with identity or not.

2) Can or can not be moved from such expression.

Any expression in C++ belongs to one of the categories:

1. lvalues: refers to identity/can’t be moved from
2. prvalues: no identity/ can be moved from
3. xvalues:

Value categories

Any expression in C++ has two properties:

1) Whether refers to something with identity or not.

2) Can or can not be moved from such expression.

Any expression in C++ belongs to one of the categories:

1. lvalues: refers to identity/can't be moved from
2. prvalues: no identity/ can be moved from
3. xvalues: identity/ can be moved from

Value categories

Any expression in C++ has two properties:

1) Whether refers to something with identity or not.

2) Can or can not be moved from such expression.

Any expression in C++ belongs to one of the categories:

1. lvalues: refers to identity/can't be moved from
2. prvalues: no identity/ can be moved from
3. xvalues: identity/ can be moved from

Examples: any ideas?

Value categories

Any expression in C++ has two properties:

1) Whether refers to something with identity or not.

2) Can or can not be moved from such expression.

Any expression in C++ belongs to one of the categories:

1. lvalues: refers to identity/can't be moved from
2. prvalues: no identity/ can be moved from
3. xvalues: identity/ can be moved from

Examples: std::move(lvalue)

Value categories

Any expression in C++ has two properties:

1) Whether refers to something with identity or not.

2) Can or can not be moved from such expression.

Any expression in C++ belongs to one of the categories:

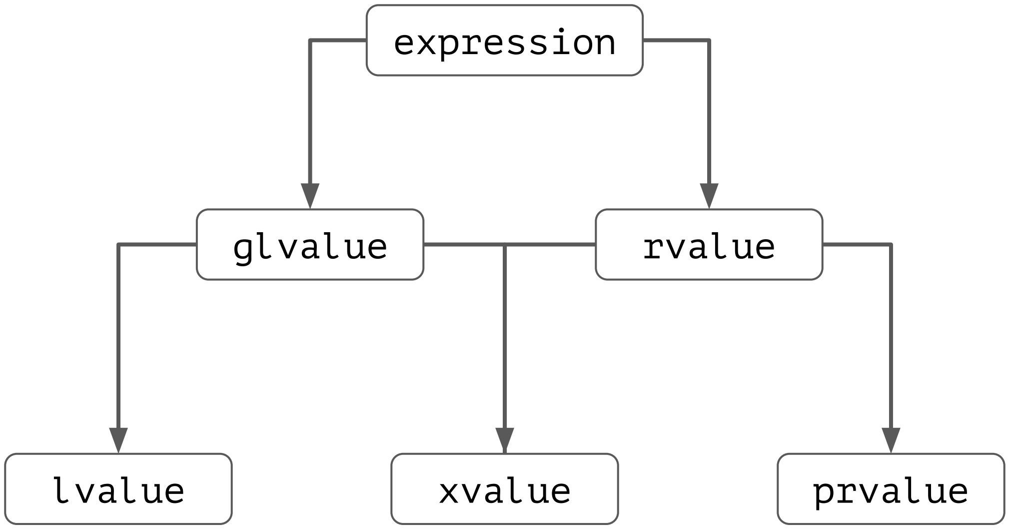
1. lvalues: refers to identity/can't be moved from
2. prvalues: no identity/ can be moved from
3. xvalues: identity/ can be moved from

Examples: std::move(lvalue), and some others

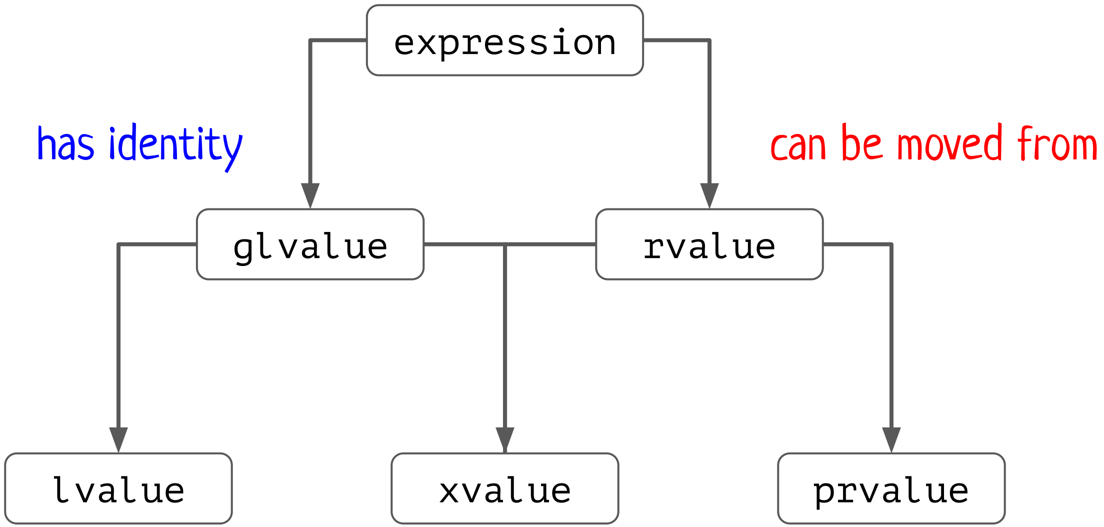
Value categories

`lvalue` `xvalue` `prvalue`

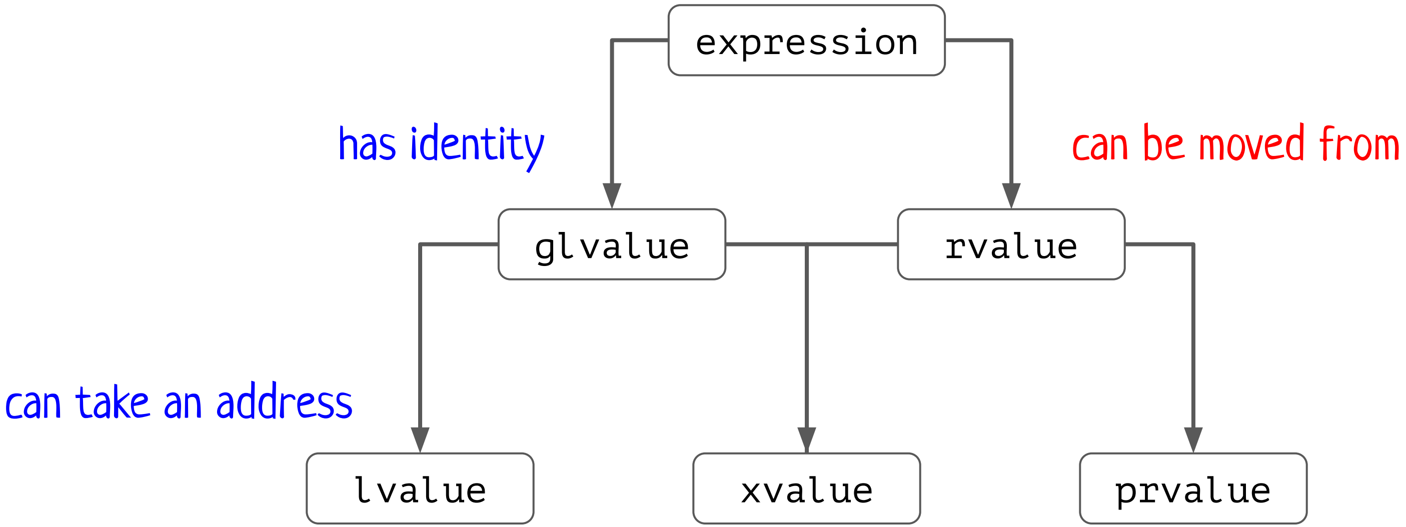
Value categories



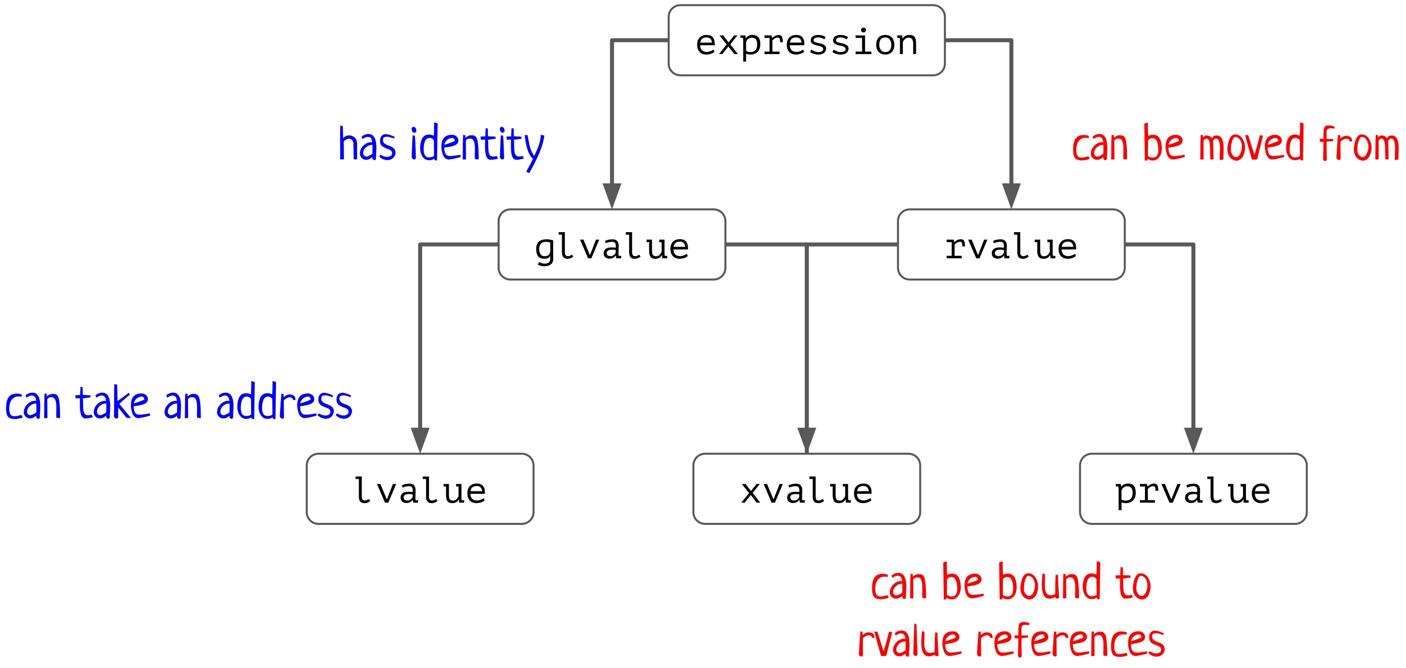
Value categories



Value categories



Value categories



Value categories: footnotes

1. `lvalues can be converted into xvalues by std::move`

Value categories: footnotes

1. lvalues can be converted into xvalues by std::move

2. prvalues also can be a source of xvalues! Example: materialization of temporary object defined by prvalue

Value categories: footnotes

1. lvalues can be converted into xvalues by std::move

2. prvalues also can be a source of xvalues! Example: materialization of temporary object defined by prvalue

```
struct S { int i; };

S{1}.i; // S{1} - prvalue, but S{1}.i - xvalue
```

Value categories: footnotes

1. lvalues can be converted into xvalues by std::move

2. prvalues also can be a source of xvalues! Example: materialization of temporary object defined by prvalue

3. Why we can't we take an address of xvalue? Not quite clear actually.

Value categories: footnotes

1. lvalues can be converted into xvalues by std::move

2. prvalues also can be a source of xvalues! Example: materialization of temporary object defined by prvalue

3. Why we can’t we take an address of xvalue? Not quite clear actually. Less possibilities to shoot your leg.

Value categories: footnotes

4. Great practical rule!

A PRACTICAL RULE

Scott Meyer has published a very useful rule of thumb to distinguish rvalues from lvalues.

> - If you can take the address of an expression, the expression is an lvalue.
> - If the type of an expression is an lvalue reference (e.g., T& or const T&, etc.), that expression is an lvalue.
> - Otherwise, the expression is an rvalue. Conceptually (and typically also in fact), rvalues correspond to temporary objects, such as those returned from functions or created through implicit type conversions. Most literal values (e.g., 10 and 5.3) are also rvalues.

Share Improve this answer Follow

edited Apr 14, 2019 at 3:35

answered Jan 20, 2016 at 13:46

Dániel Sándor

1,256 ● 10 ● 15

Not So Tiny Task №4 (1 point)

Improve solution from NSTT #1 by adding move constructors and move assignment operators where needed.

Don’t forget to add tests that show that copy constructors and assign operators work correctly.

Takeaways

- Move semantics. Stealing resources from a dead man is not a crime, but optimization!
- Rule of five.
- Rvalue references. Different value categories in C++: the whole picture.