Counterfeit Object-oriented Programming

Felix Schuster, Thomas Tendyck, Christopher Liebchen, Lucas Davi, Ahmad-Reza Sadeghi, and Thorsten Holz (S&P '15)

Jaemin Hong 20203698

Fake

Fabricate

Mimic

Counterfeit Object-oriented Programming

Simulate

Felix Schuster, Thomas Tendyck, Christopher Liebchen, Lucas Davi, Ahmad-Reza Sadeghi, and Thorsten Holz (S&P '15)

Jaemin Hong 20203698

Fake

Fabricate

Mimic

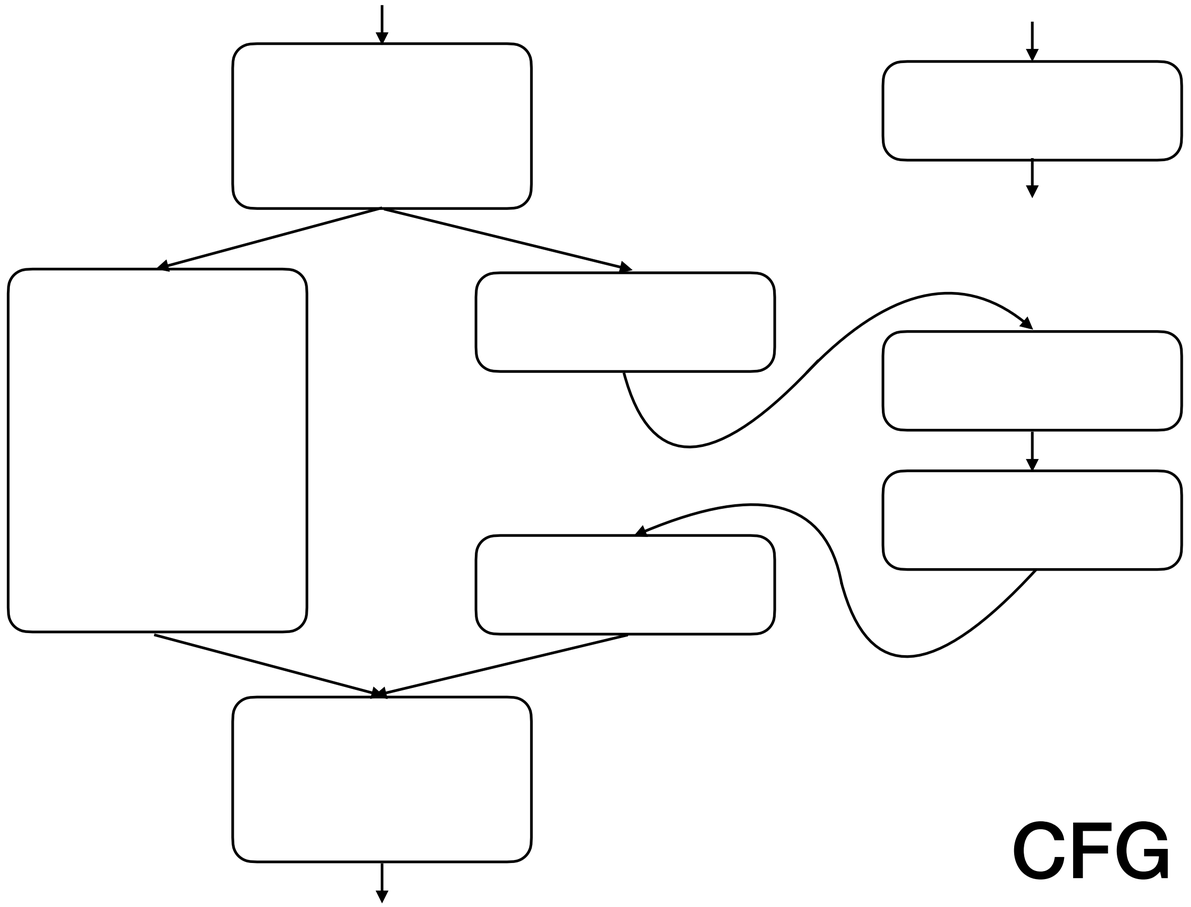
Counterfeit Object-oriented Programming

Simulate

Felix Schuster, Thomas Tendyck, Christopher Liebchen, Lucas Davi, Ahmad-Reza Sadeghi, and Thorsten Holz (S&P '15)

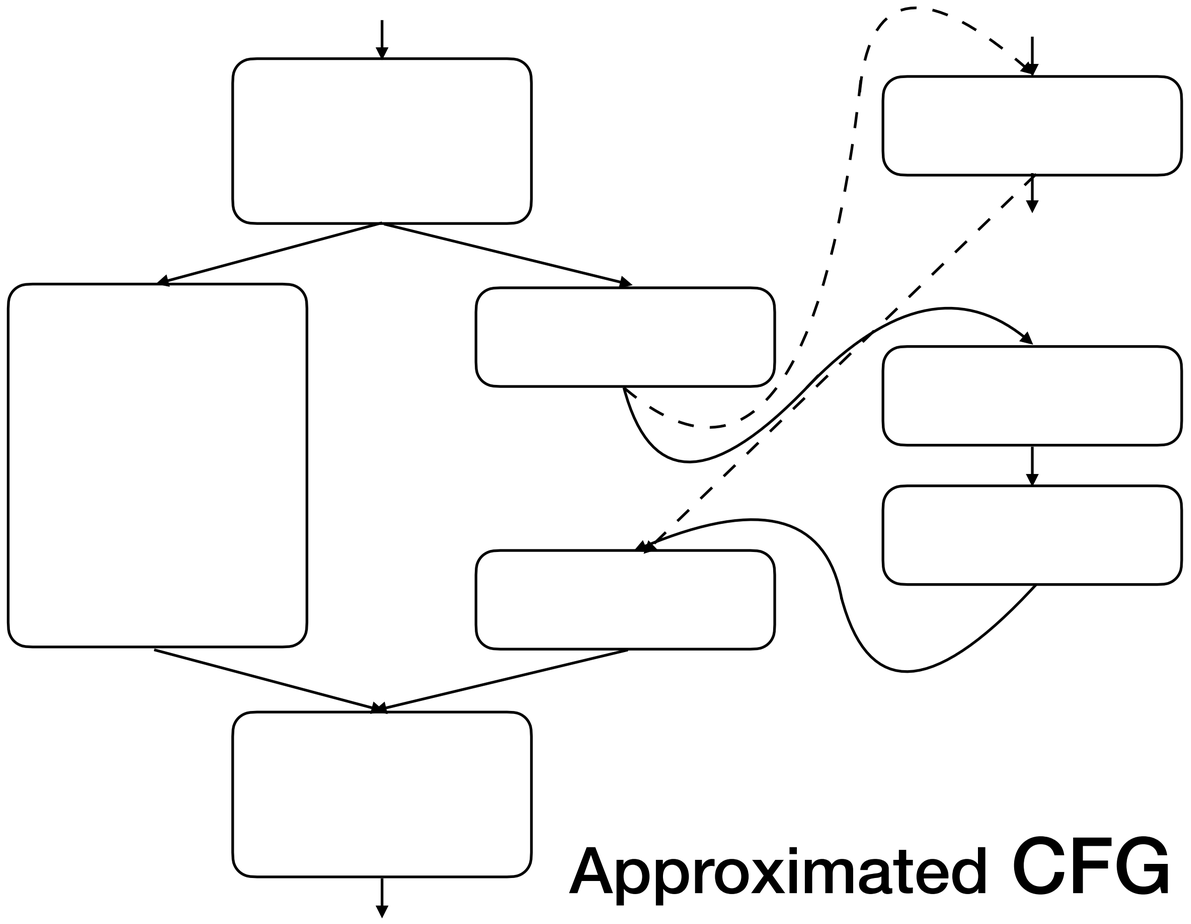
Jaemin Hong 20203698

CFI uses an approximated CFG to detect spurious function calls at runtime.



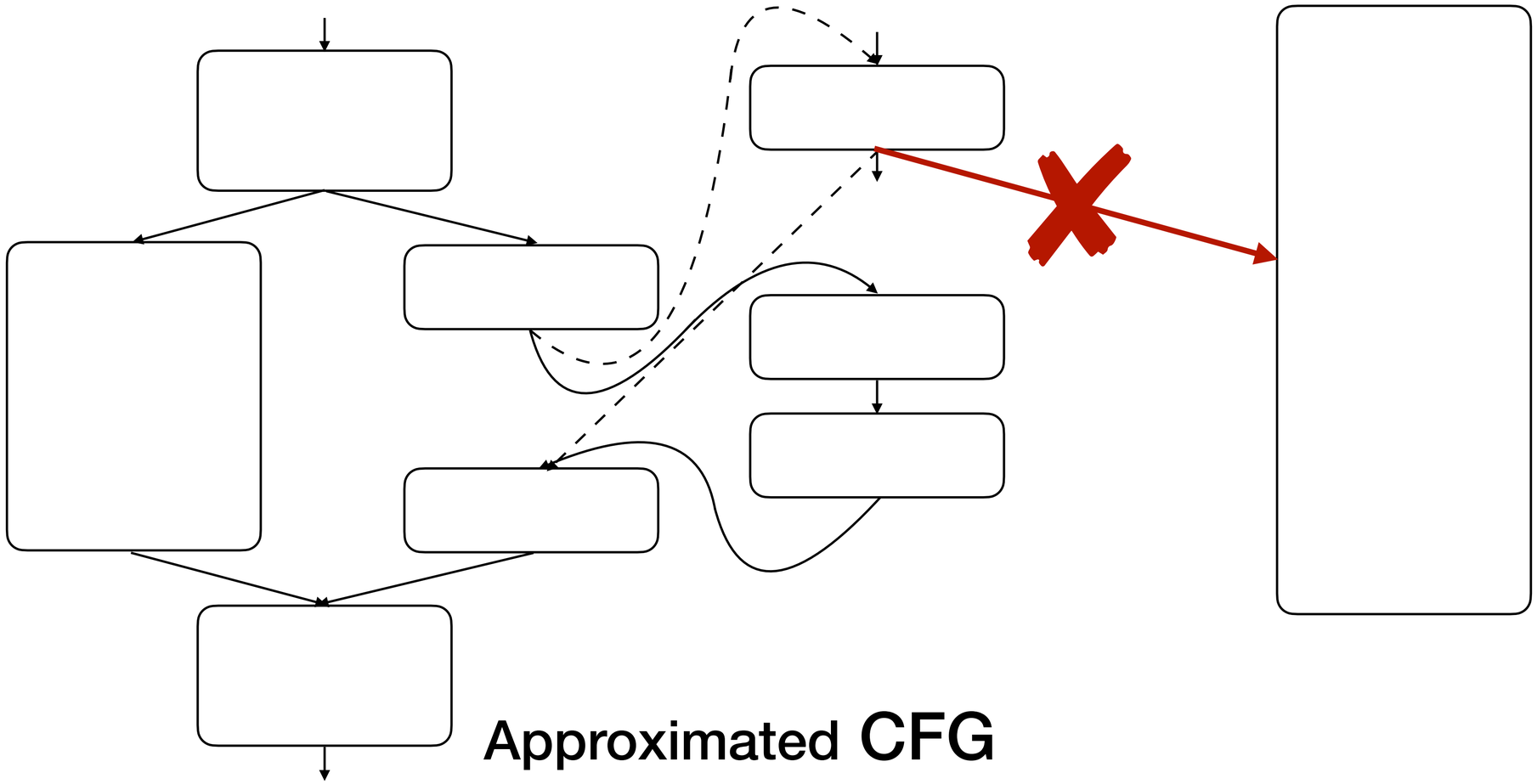
CFG

CFI uses an approximated CFG to detect spurious function calls at runtime.



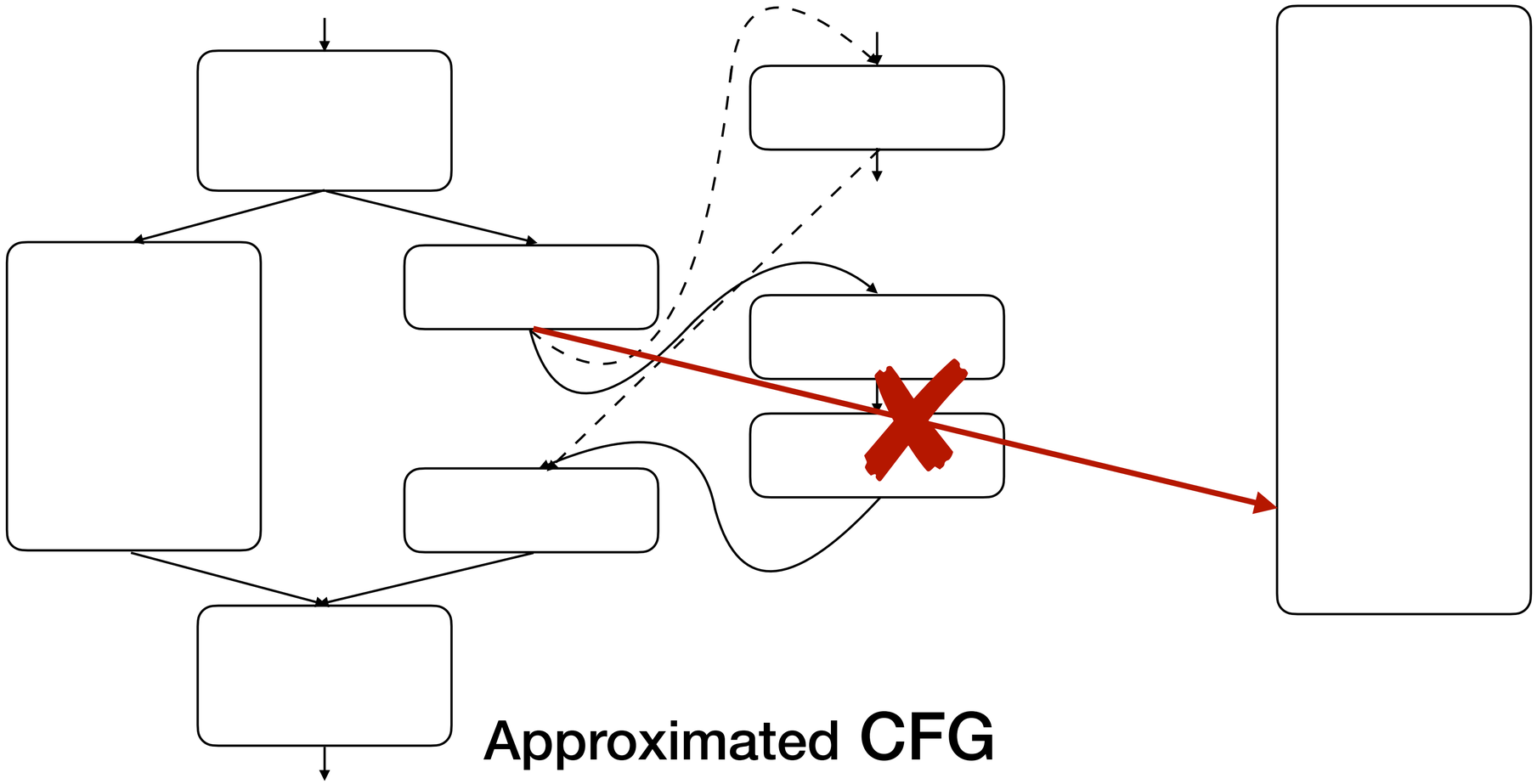
Approximated CFG

CFI uses an approximated CFG to detect spurious function calls at runtime.



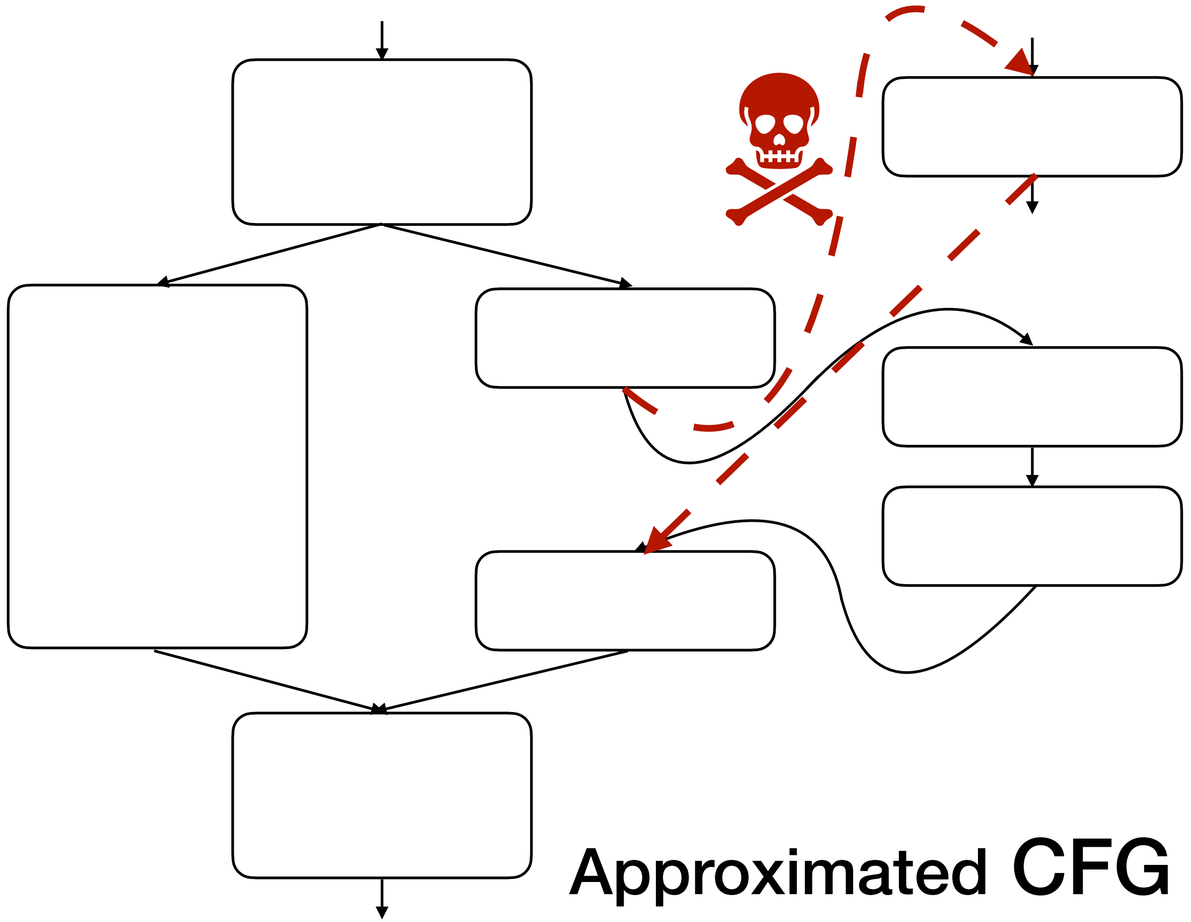
Approximated CFG

CFI uses an approximated CFG to detect spurious function calls at runtime.

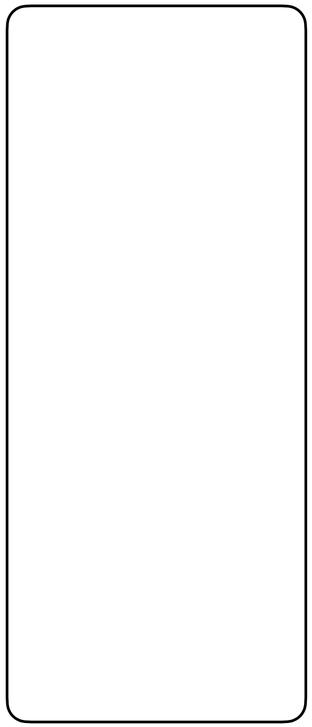


Approximated CFG

COOP can bypass CFI
by exhibiting function calls that *seem* benign.



Approximated CFG



A class inherits methods from its parents.

```
class A {
  virtual int f() { return 0; }
};

class B : public A {

};
```

```
A *a = new A();
a->f();  // 0



B *b = new B();
b->f();  // 0
```

A class can override methods from its parents.

```
class A {
  virtual int f() { return 0; }
};

class B : public A {
  virtual int f() { return 1; }
};
```

```
A *a = new A();
a->f();  // 0


B *b = new B();
b->f();  // 1
```

Virtual methods allow dynamic dispatch.

```
class A {
  virtual int f() { return 0; }
};

class B : public A {
  virtual int f() { return 1; }
};
```

```
A *a = new A();
a->f();  // 0


A *b = (A *) new B();
b->f();  // 1
```

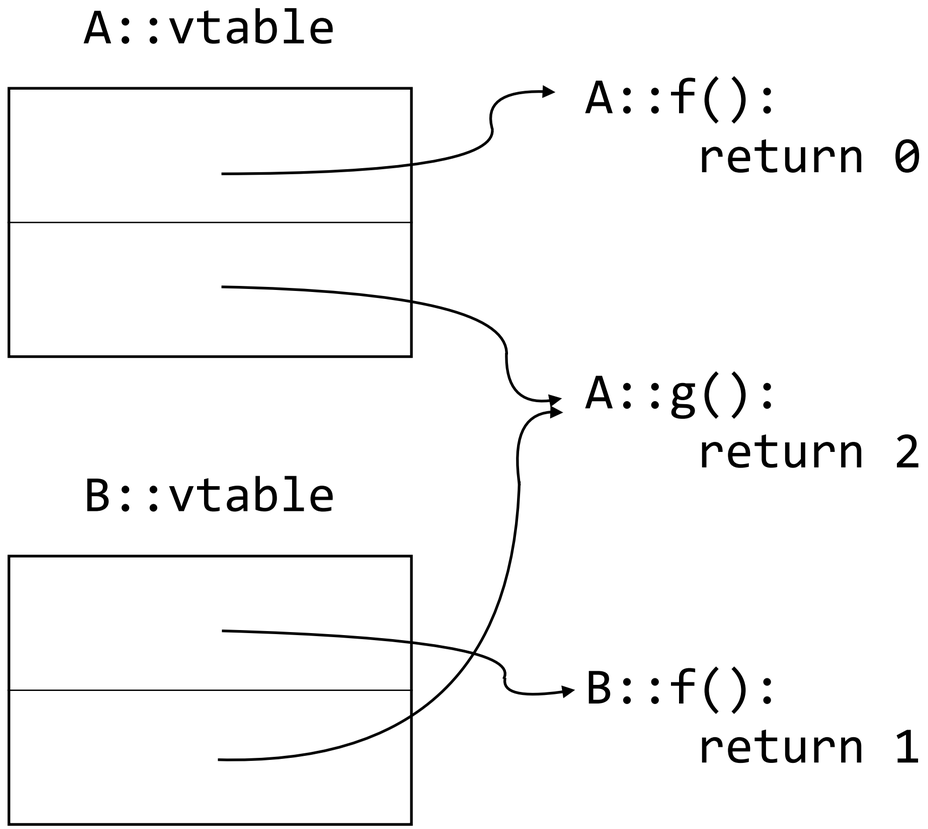
Virtual methods are implemented via `vtable`s.

```
class A {
  virtual int f() { return 0; }
  virtual int g() { return 2; }
};

class B : public A {
  virtual int f() { return 1; }
};
```

```
A::f():
    return 0

A::g():
    return 2

B::f():
    return 1
```

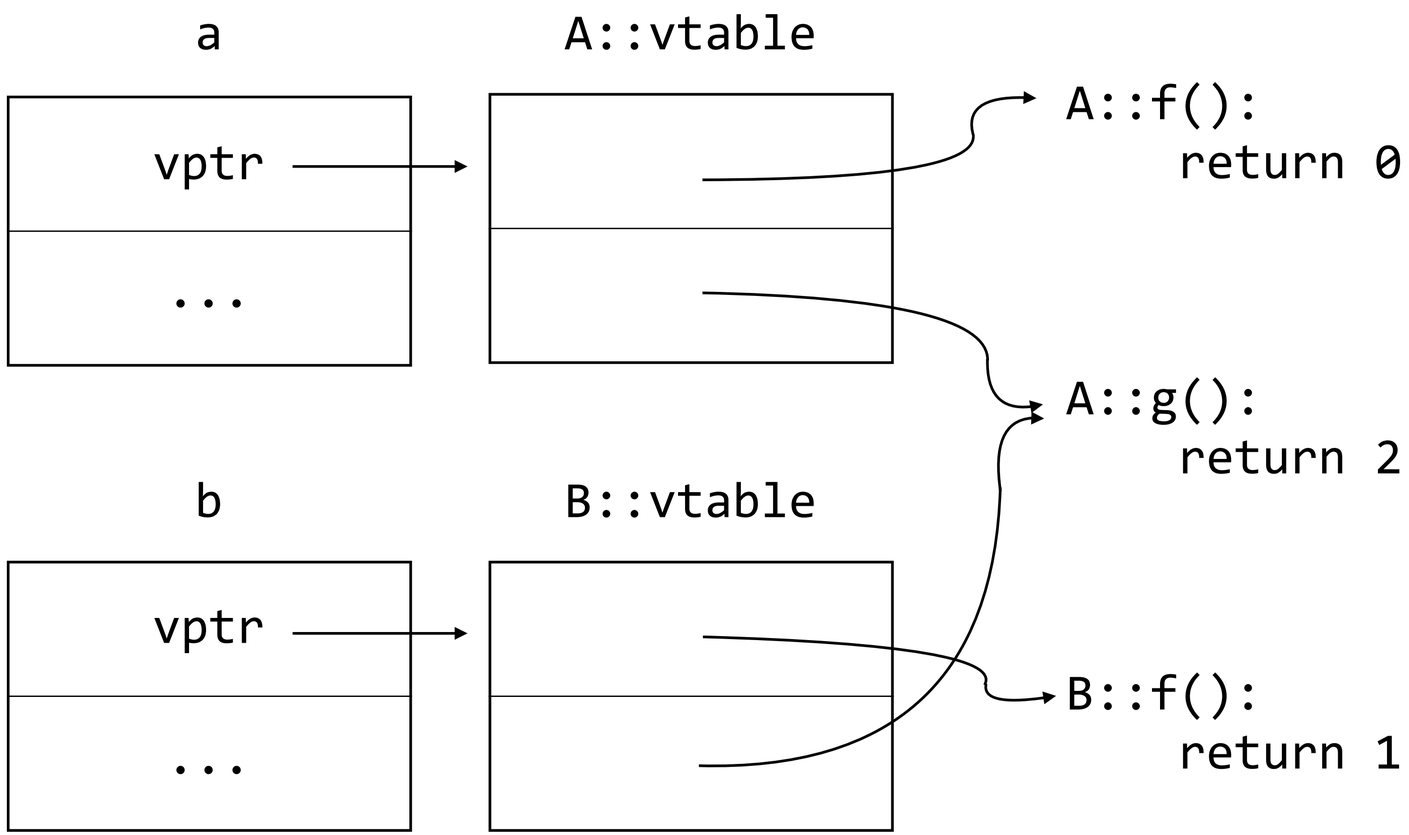
Virtual methods are implemented via `vtable`s.

```
class A {
  virtual int f() { return 0; }
  virtual int g() { return 2; }
};

class B : public A {
  virtual int f() { return 1; }
};
```



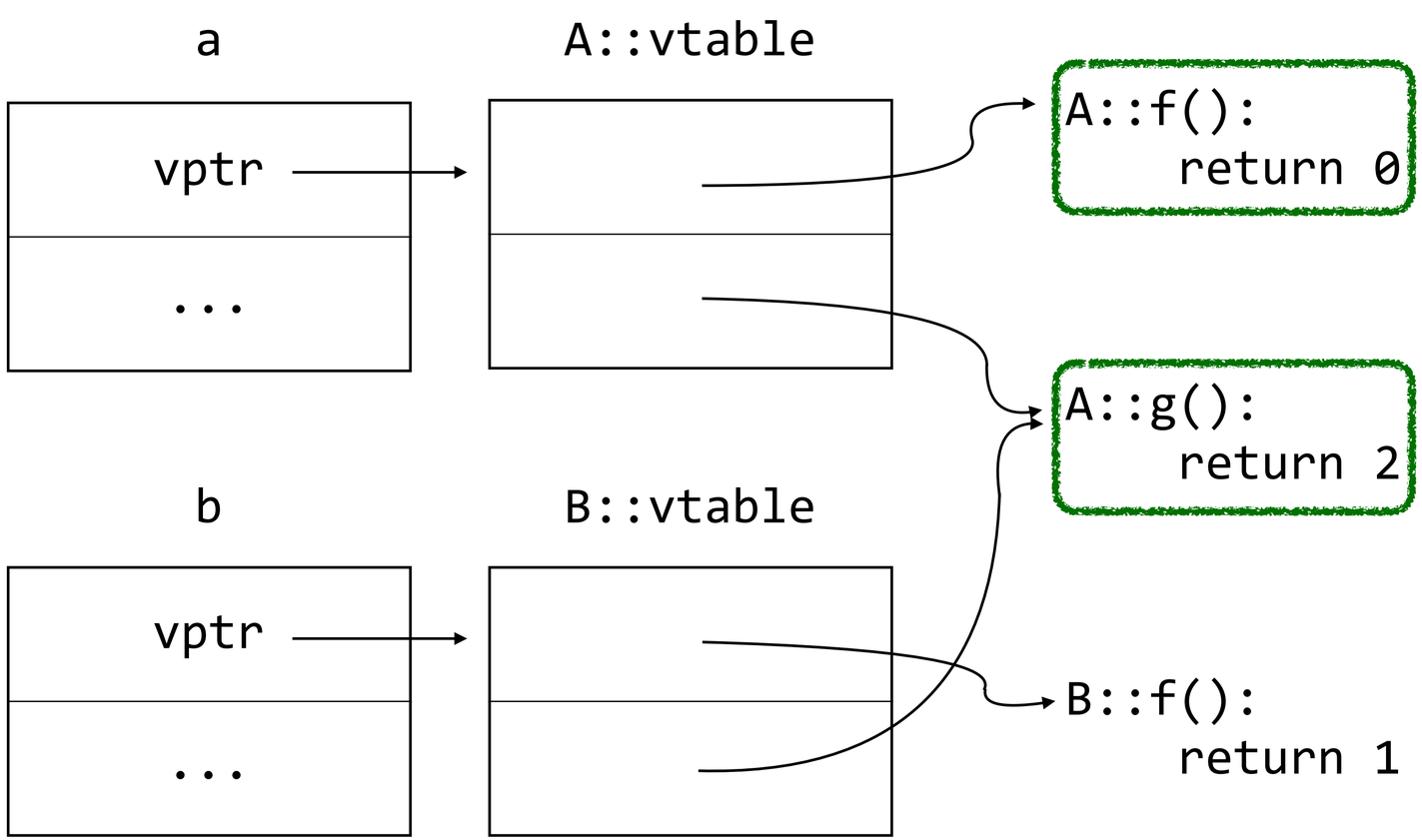
Virtual methods are implemented via `vtable`s.

```
A *a = new A();

A *b = (A *) new B();
```



Virtual methods are implemented via `vtable`s.

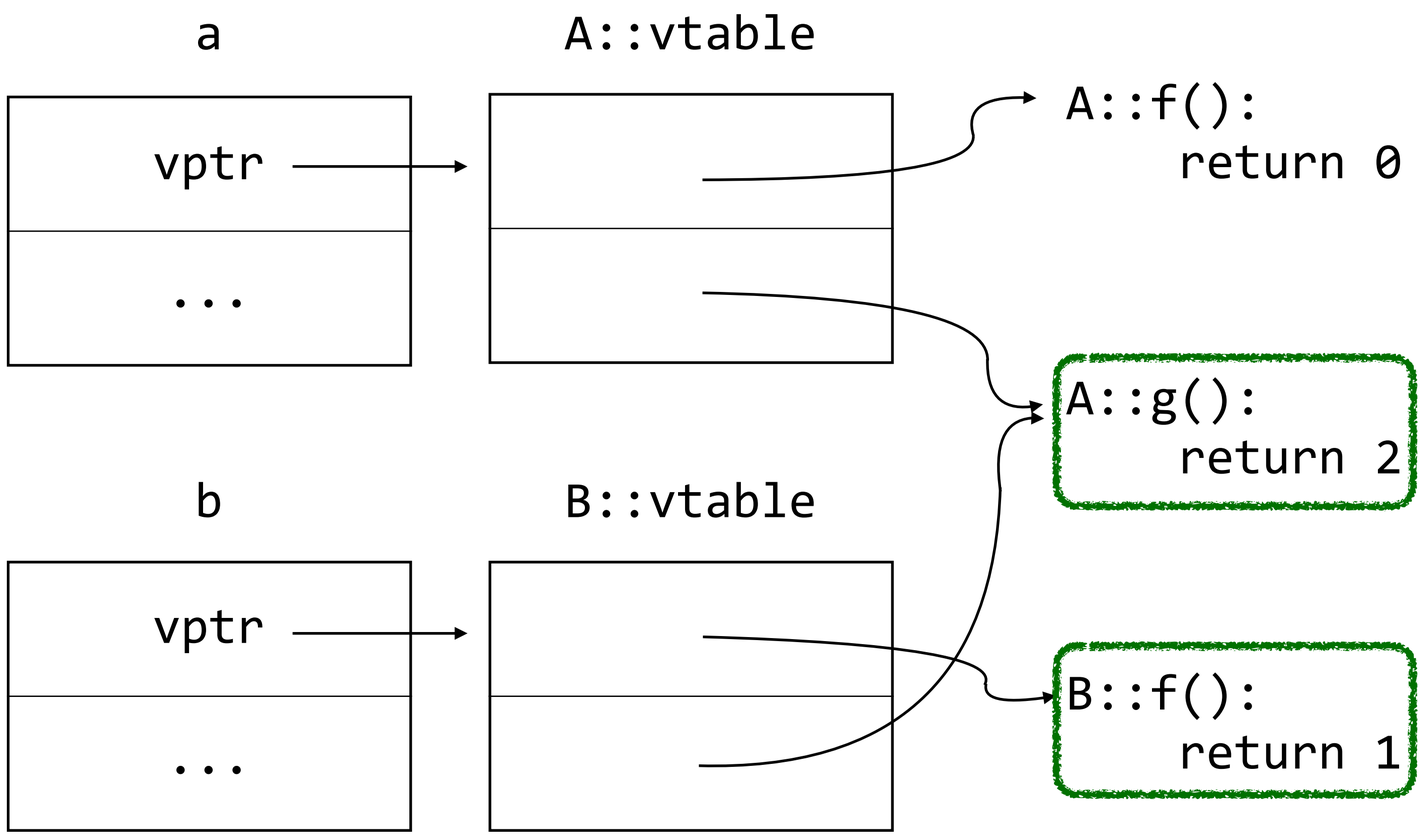
```
A *a = new A();
a->f();  // 0
a->g();  // 2


A *b = (A *) new B();
```



Virtual methods are implemented via `vtable`s.

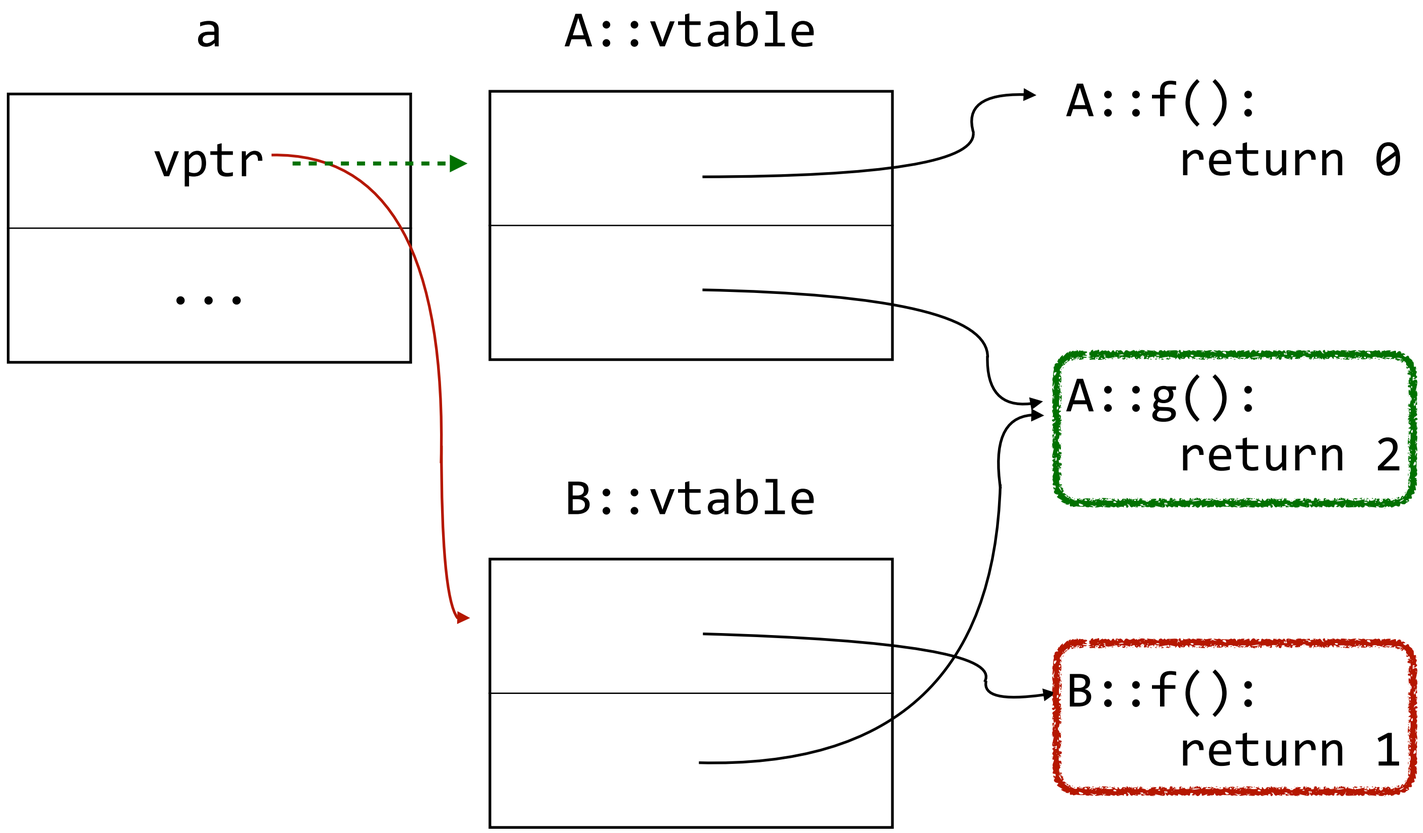
```
A *a = new A();
```

```
A *b = (A *) new B();
b->f();  // 1
b->g();  // 2
```



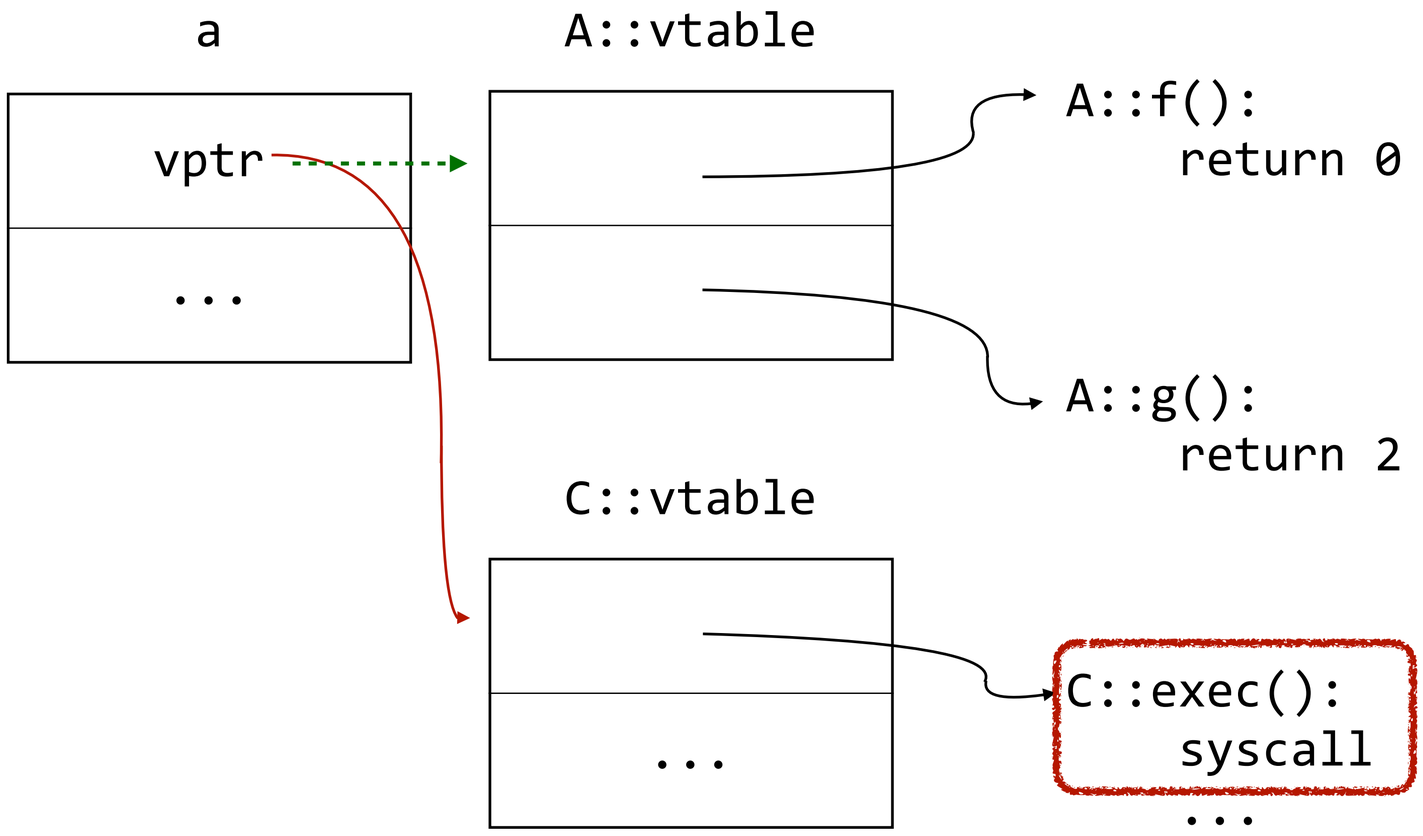
We can change the behavior of an object by changing `vptr` of the object.

```
A *a = new A();
a->f();  // 1
a->g();  // 2
```



We can change the behavior of an object by changing `vptr` of the object.

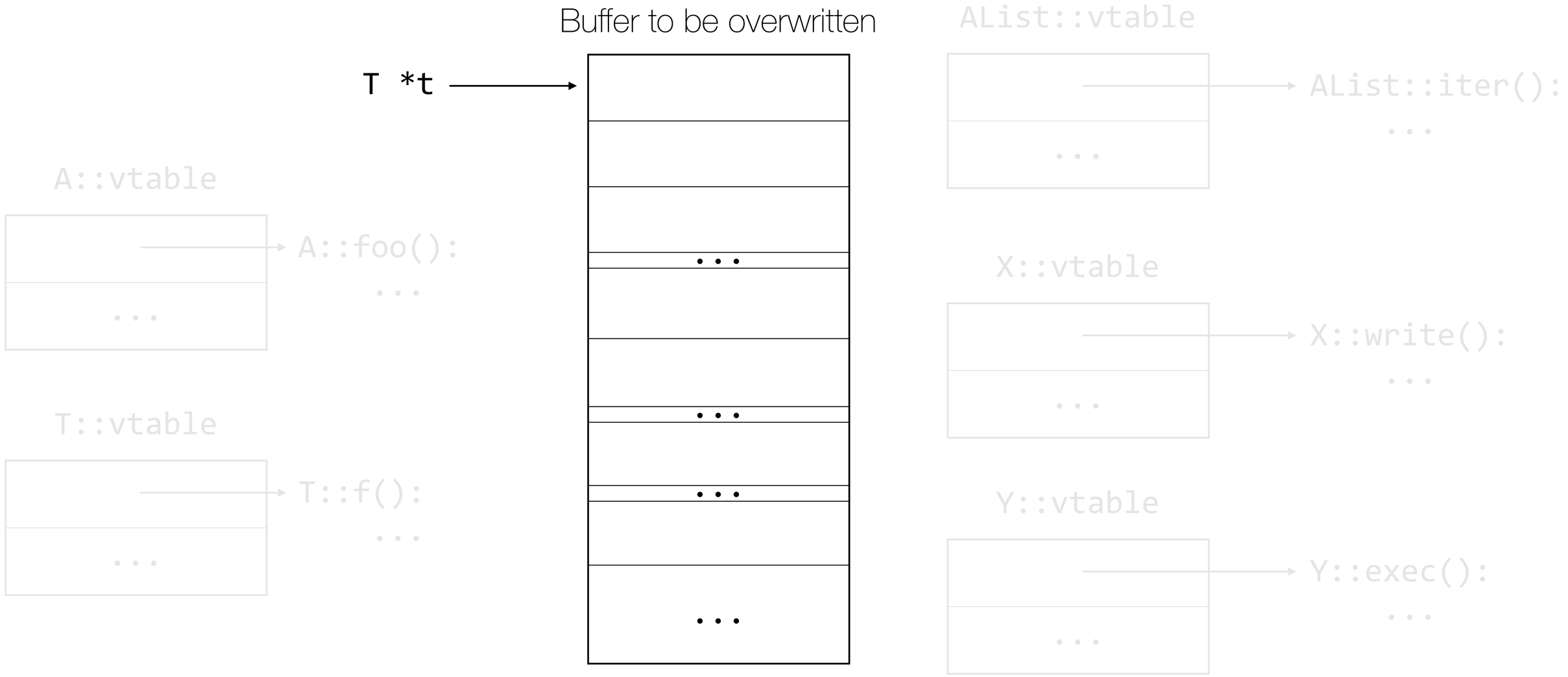
```
A *a = new A();
a->f();  // ???
a->g();  // ...
```



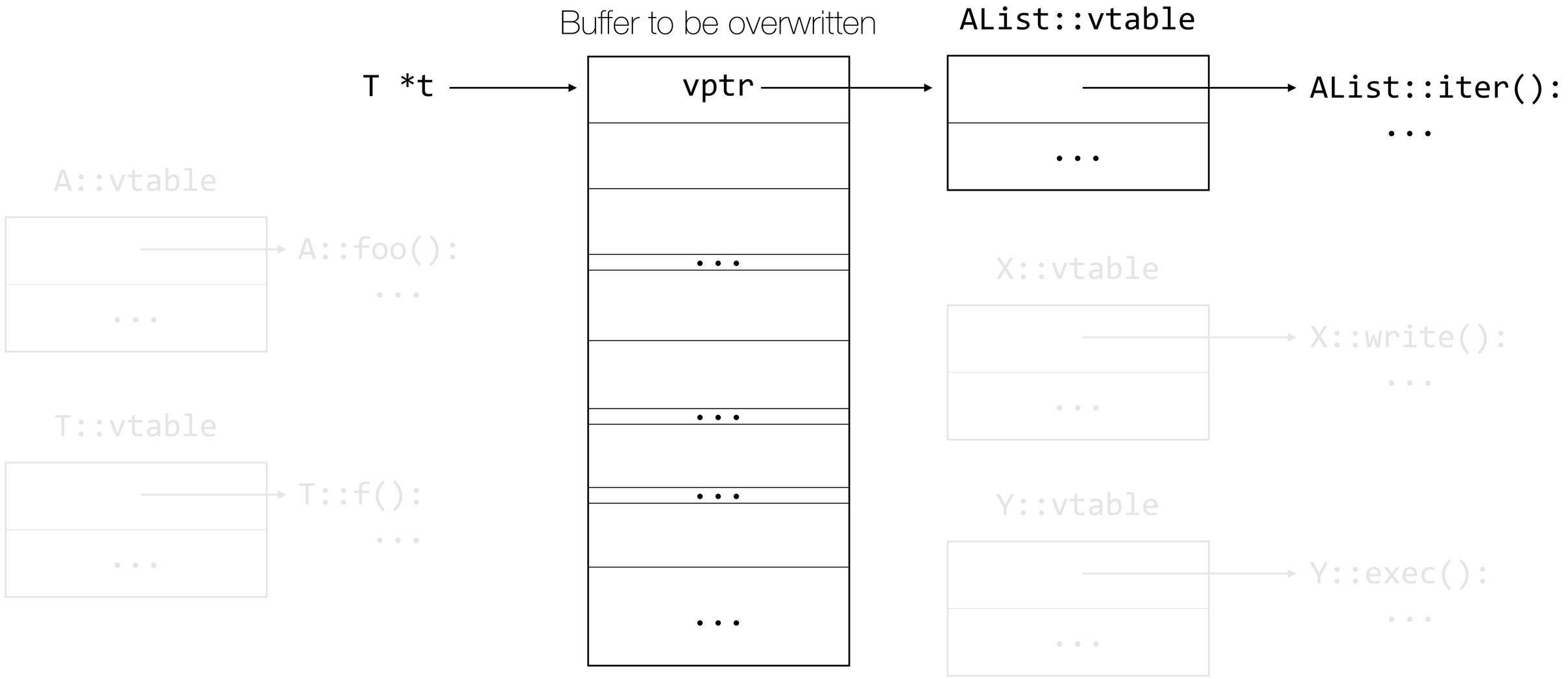
The main loop gadget allows invoking an arbitrary sequence of methods.

```
class AList {
    A **arr;
    int num;
    virtual void iter() {
        for (int i = 0; i < num; i++)
            arr[i]->foo();
    }
};
```

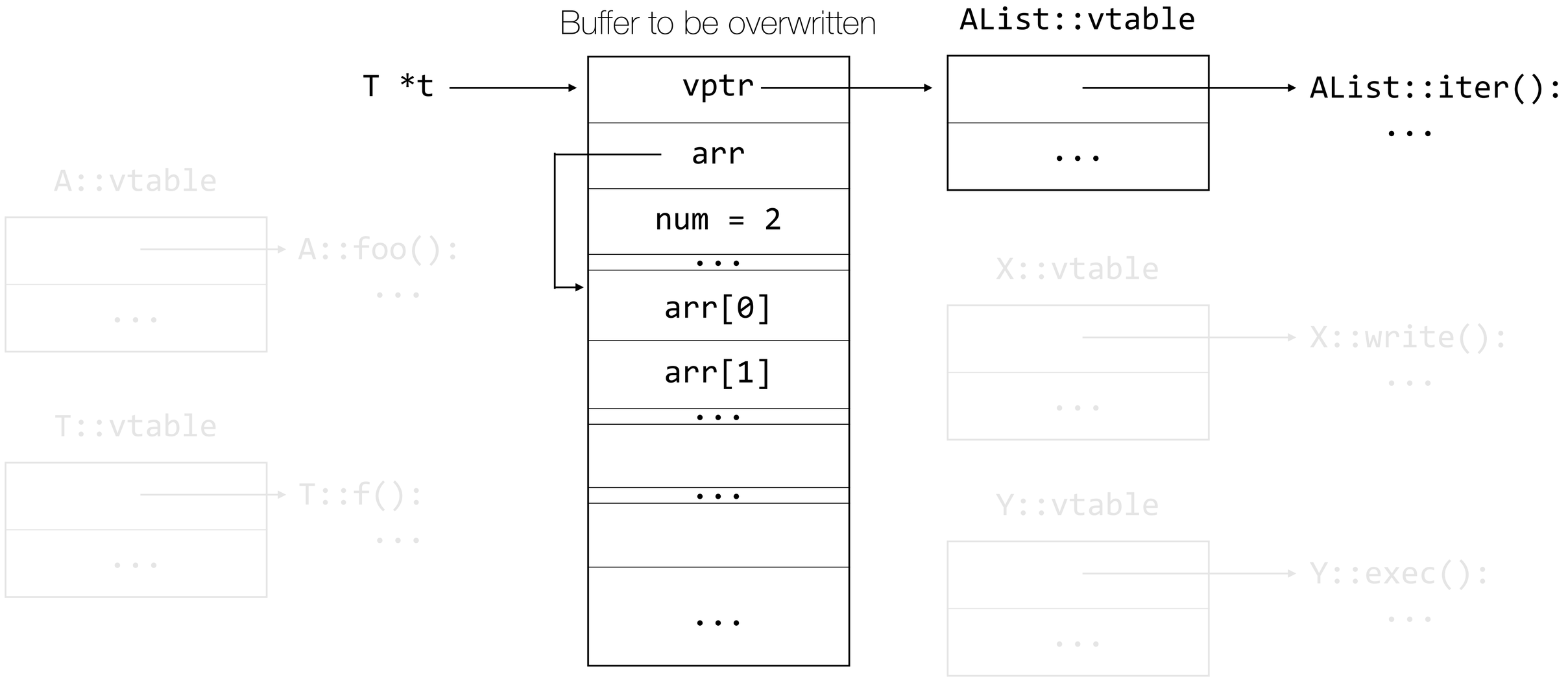
The main loop gadget allows invoking an arbitrary sequence of methods.



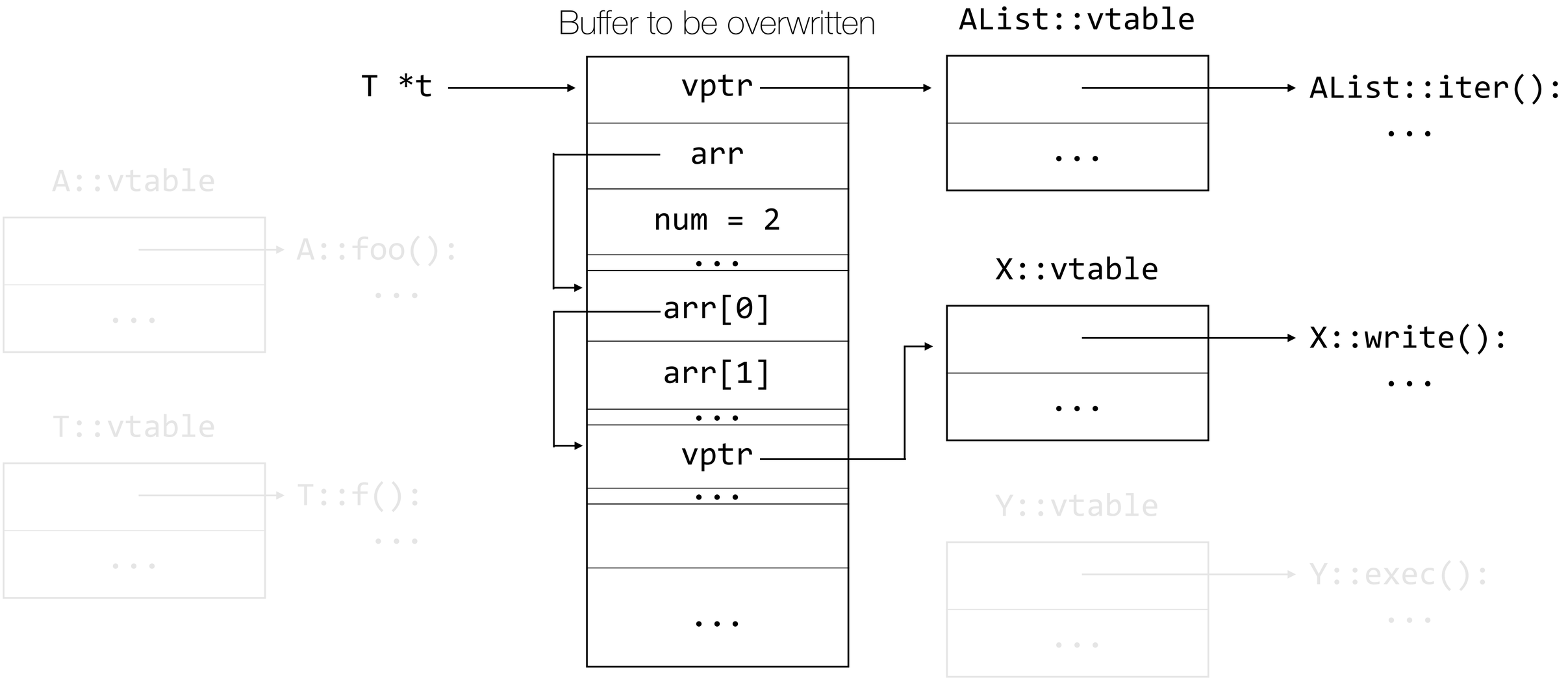
The main loop gadget allows invoking an arbitrary sequence of methods.



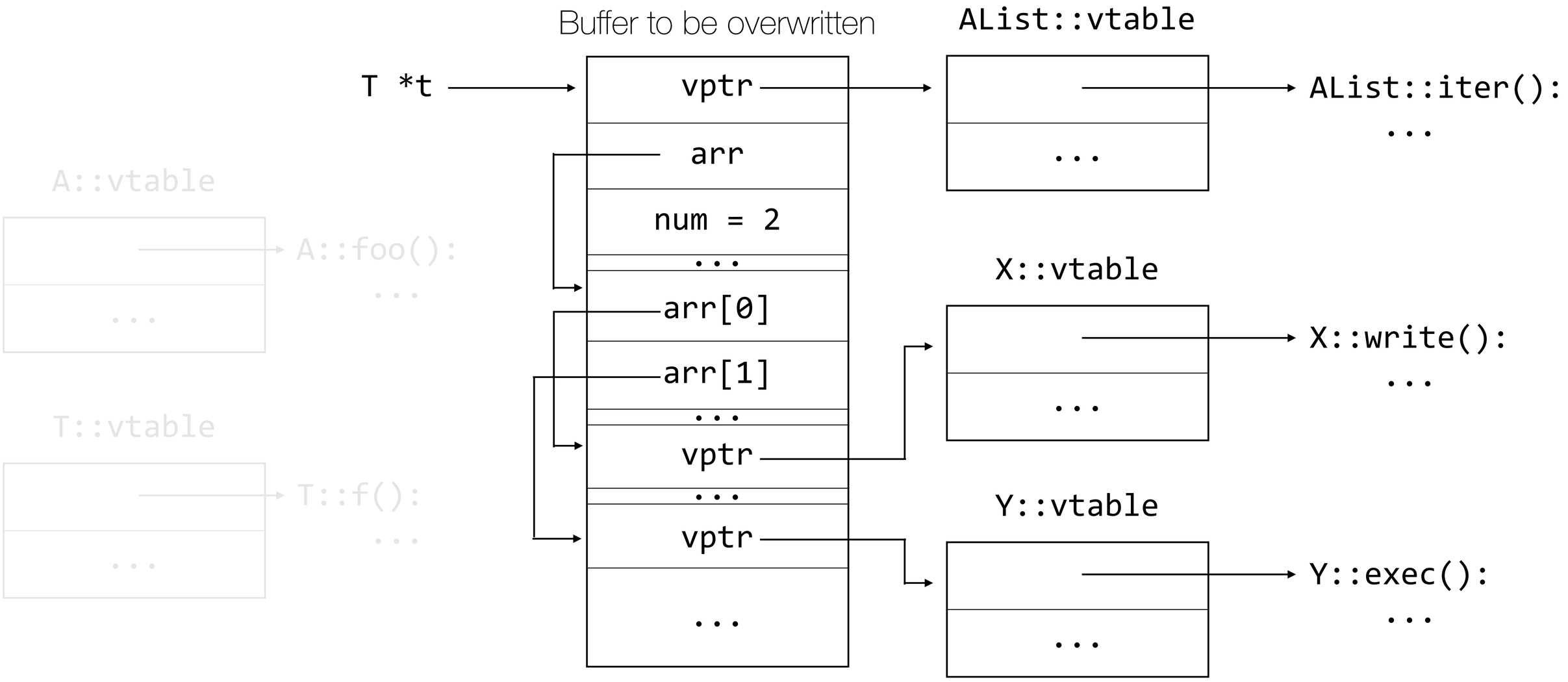
The main loop gadget allows invoking an arbitrary sequence of methods.



The main loop gadget allows invoking an arbitrary sequence of methods.

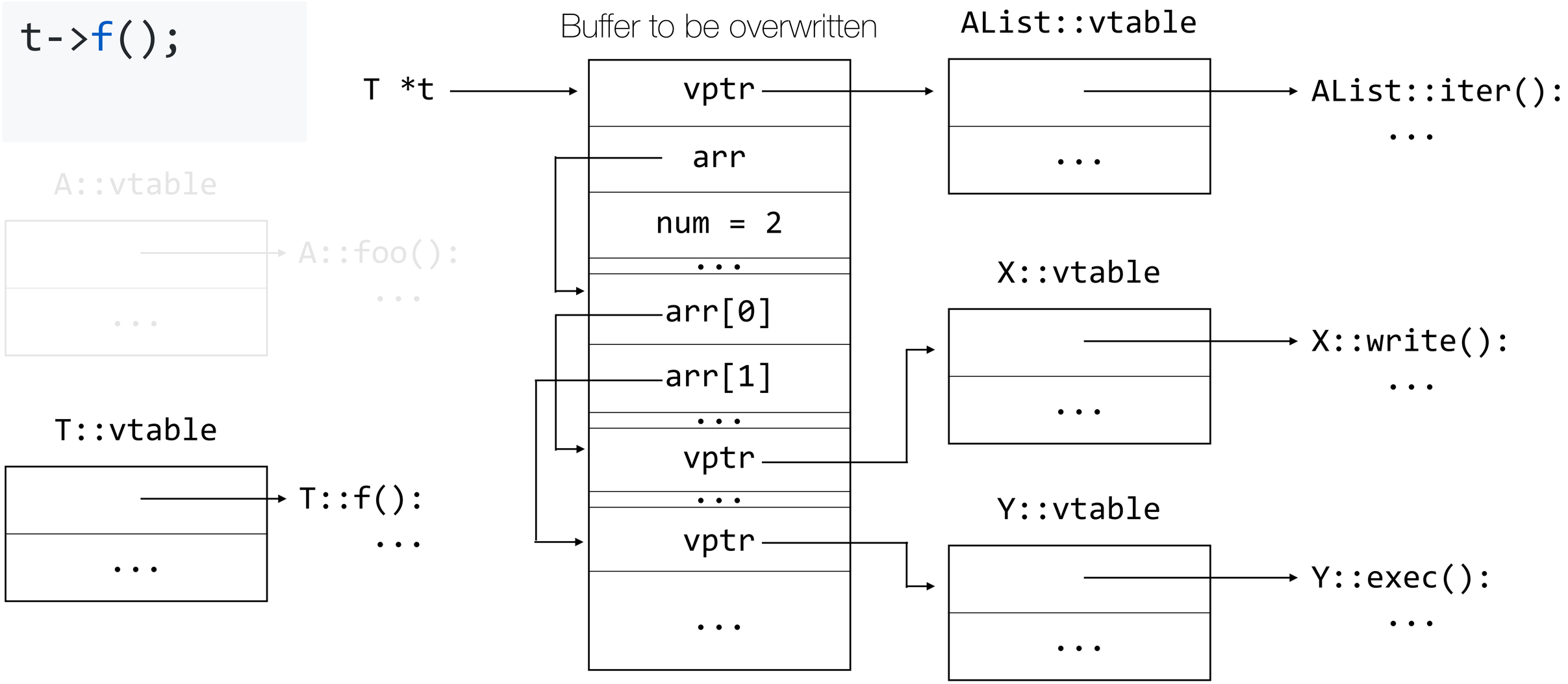


The main loop gadget allows invoking an arbitrary sequence of methods.



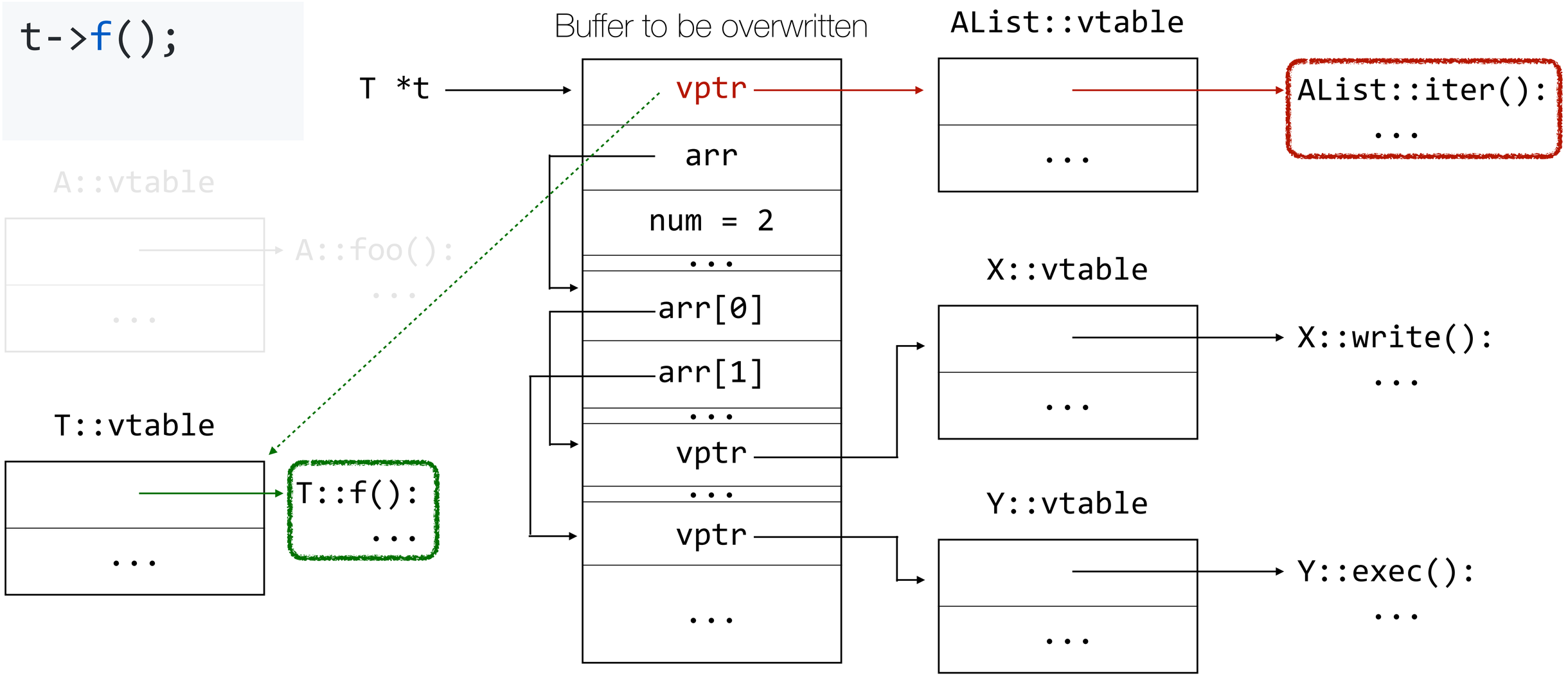
The main loop gadget allows invoking an arbitrary sequence of methods.

```
t->f();
```



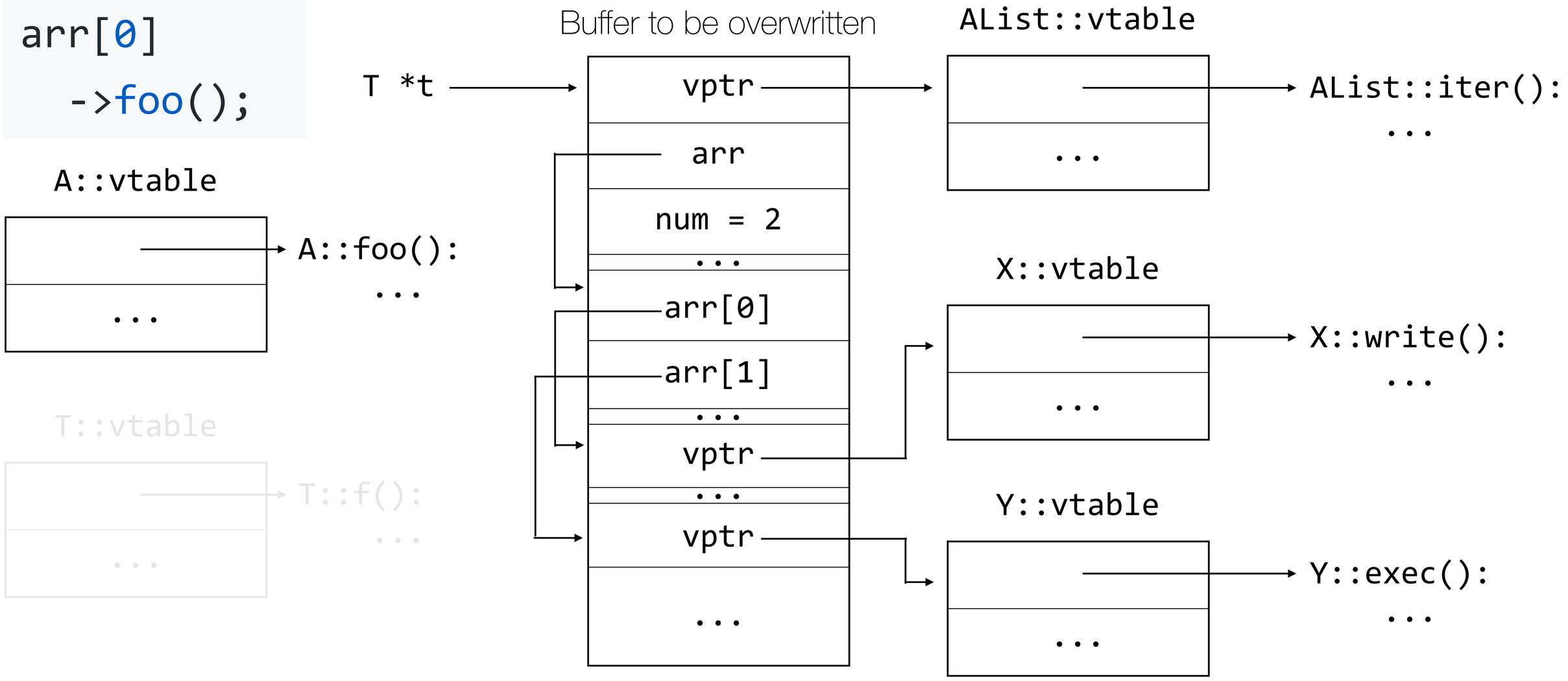
The main loop gadget allows invoking an arbitrary sequence of methods.

```
t->f();
```



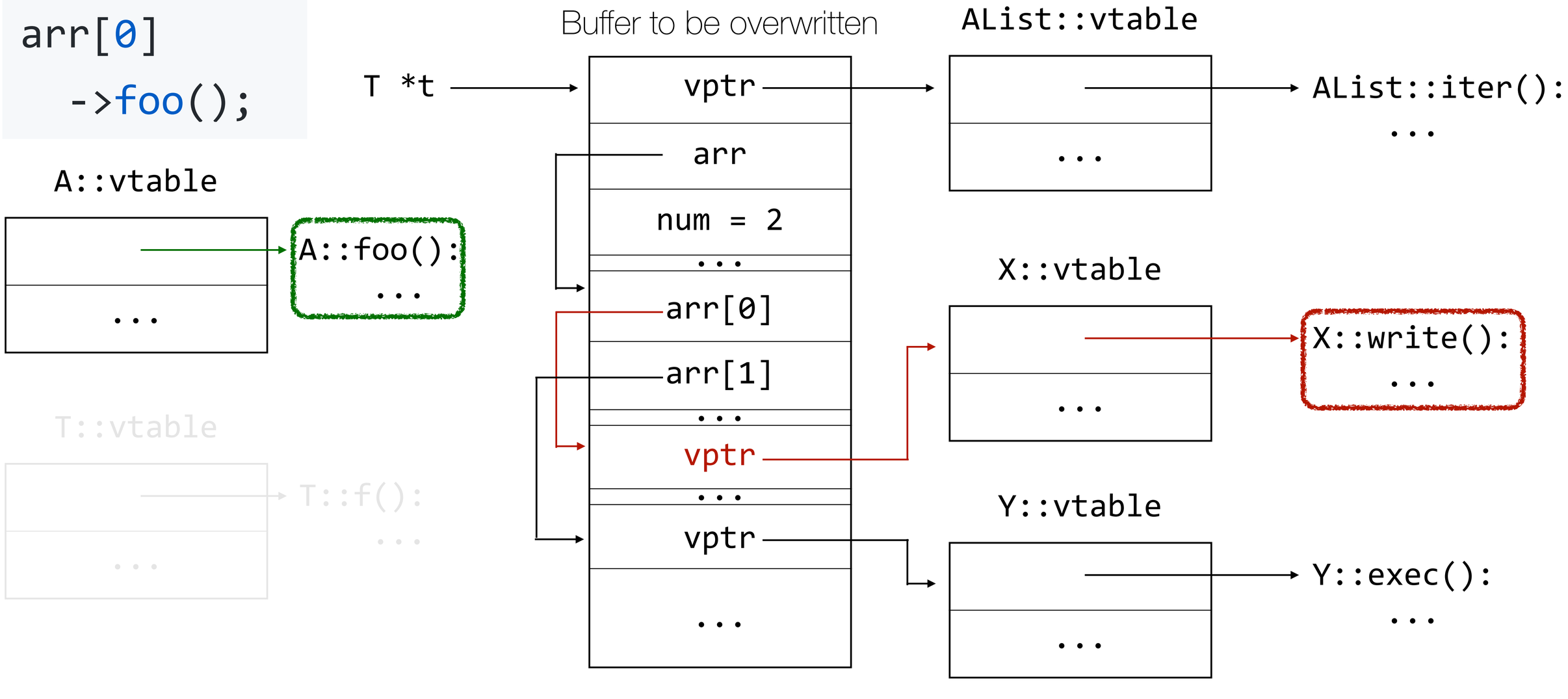
The main loop gadget allows invoking an arbitrary sequence of methods.

```
arr[0]
  ->foo();
```



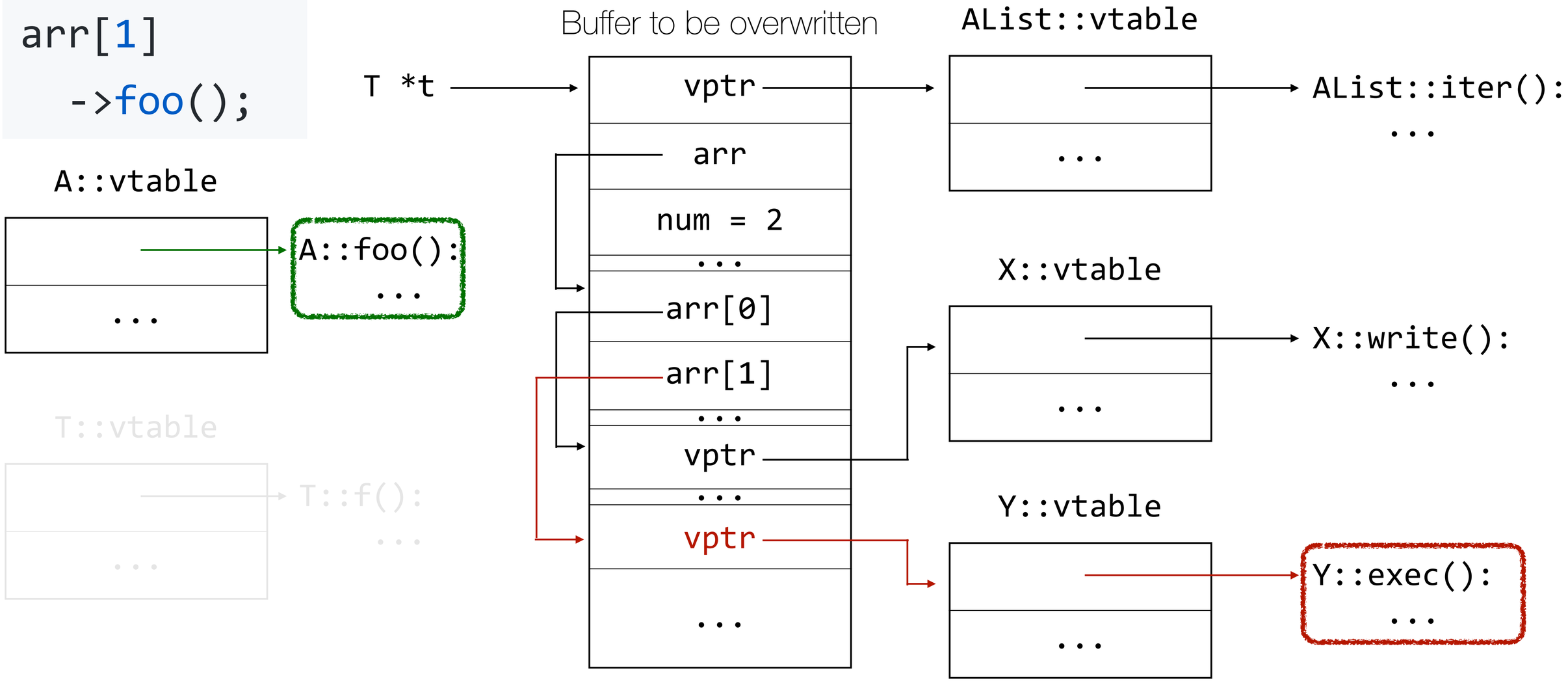
The main loop gadget allows invoking an arbitrary sequence of methods.

```
arr[0]
  ->foo();
```

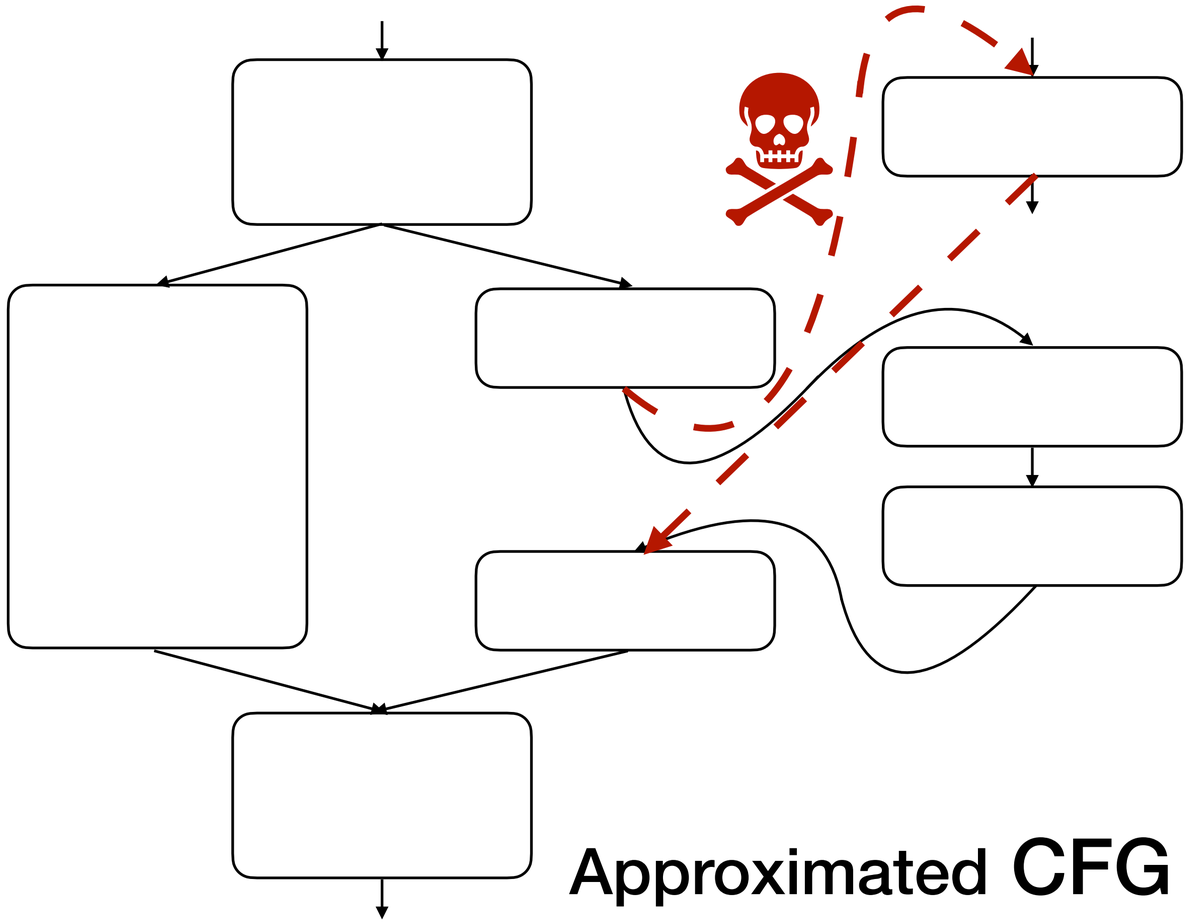


The main loop gadget allows invoking an arbitrary sequence of methods.

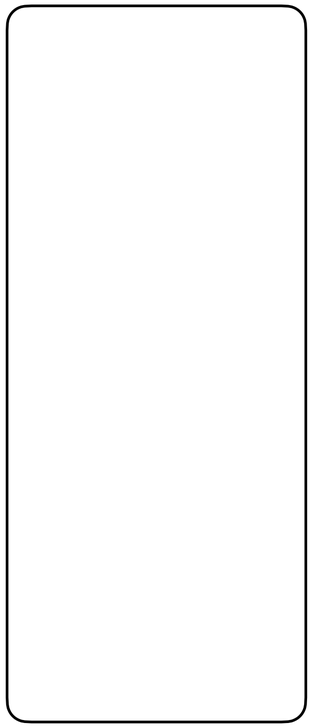
```
arr[1]
  ->foo();
```



COOP can bypass CFI
by exhibiting function calls that *seem* benign.

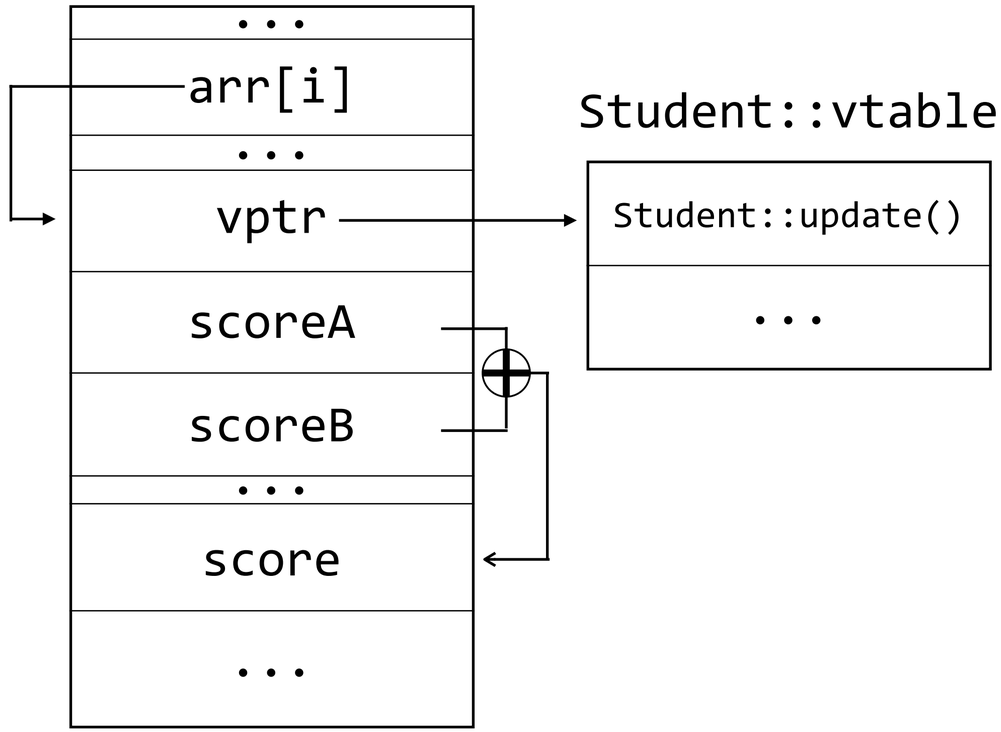


Approximated CFG



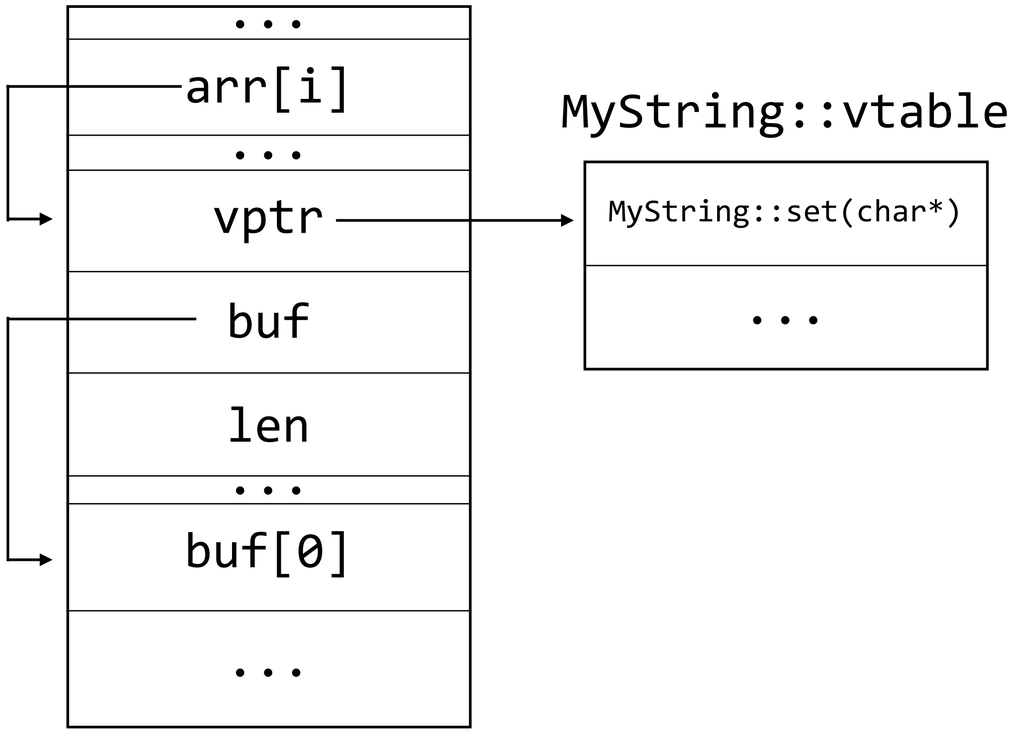
Arithmetic gadgets can be found easily.

```
class Student {
    int scoreA, scoreB;
    ...
    int score;

    virtual void update() {
        score = scoreA + scoreB;
    }
};
```



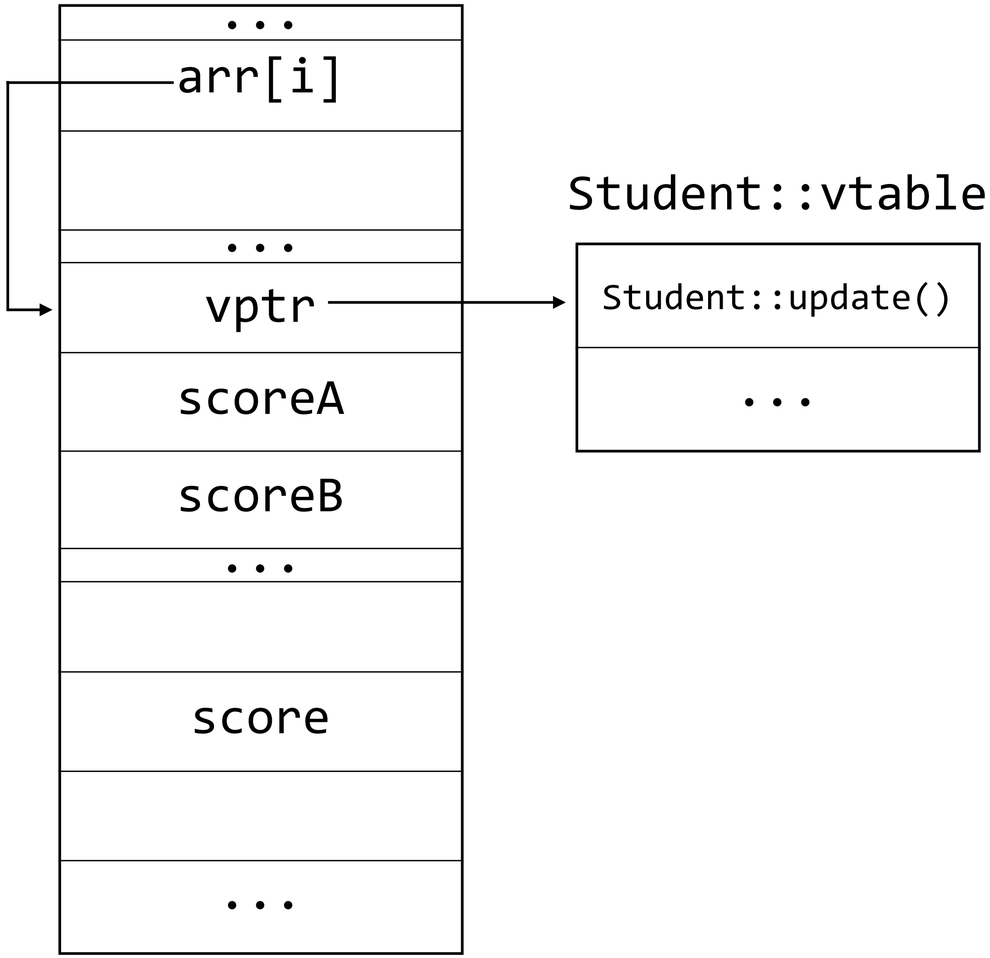
Read/write gadgets can be found easily.

```
class MyString {
    char *buf;
    int len;

    virtual void set(char *src) {
        strncpy(buf, src, len);
    }
};
```



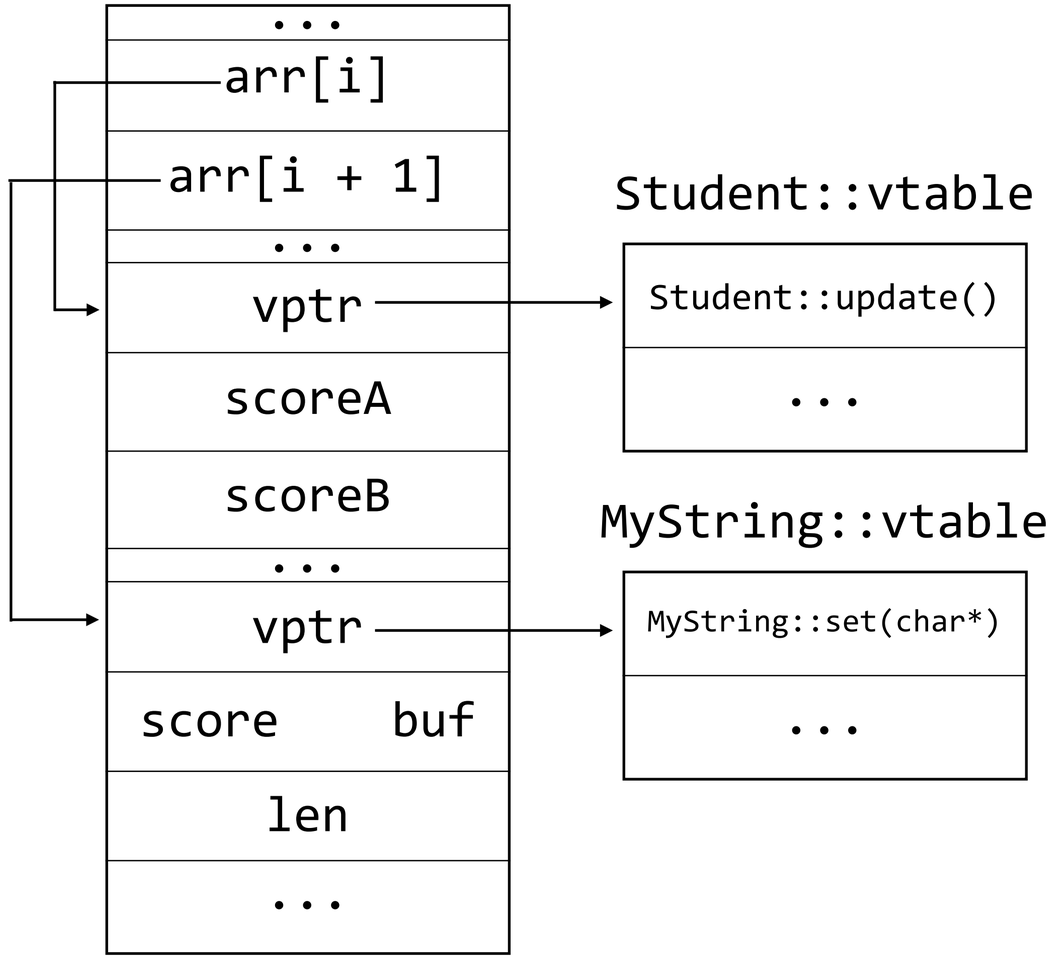
Overlapping objects allow arbitrary read/write.

```
class Student {
    int scoreA, scoreB, score;
    virtual void update() { .. }
};
class MyString {
    char *buf; int len;
    virtual void set(char *src)
    { .. }
};
```



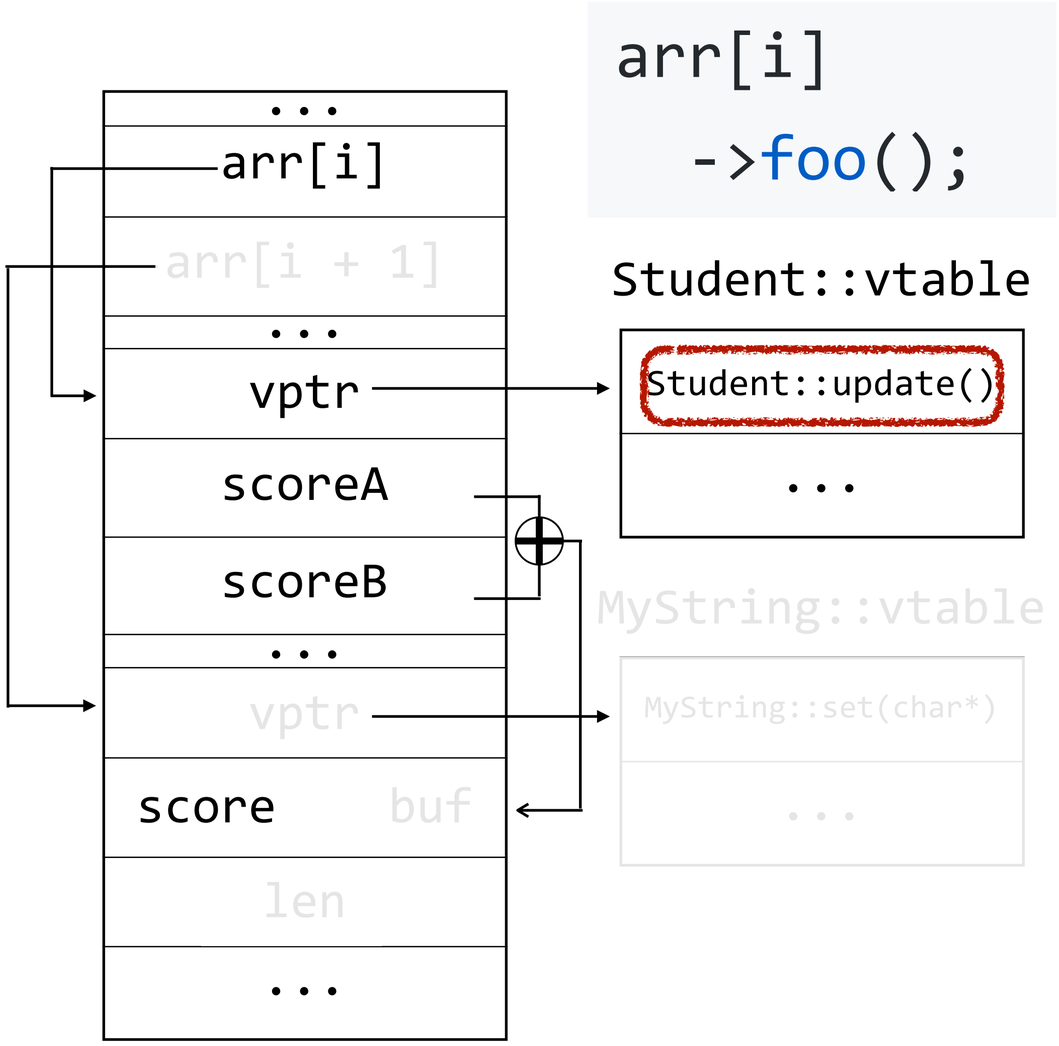
Overlapping objects allow arbitrary read/write.

```
class Student {
    int scoreA, scoreB, score;
    virtual void update() { .. }
};
class MyString {
    char *buf; int len;
    virtual void set(char *src)
    { .. }
};
```



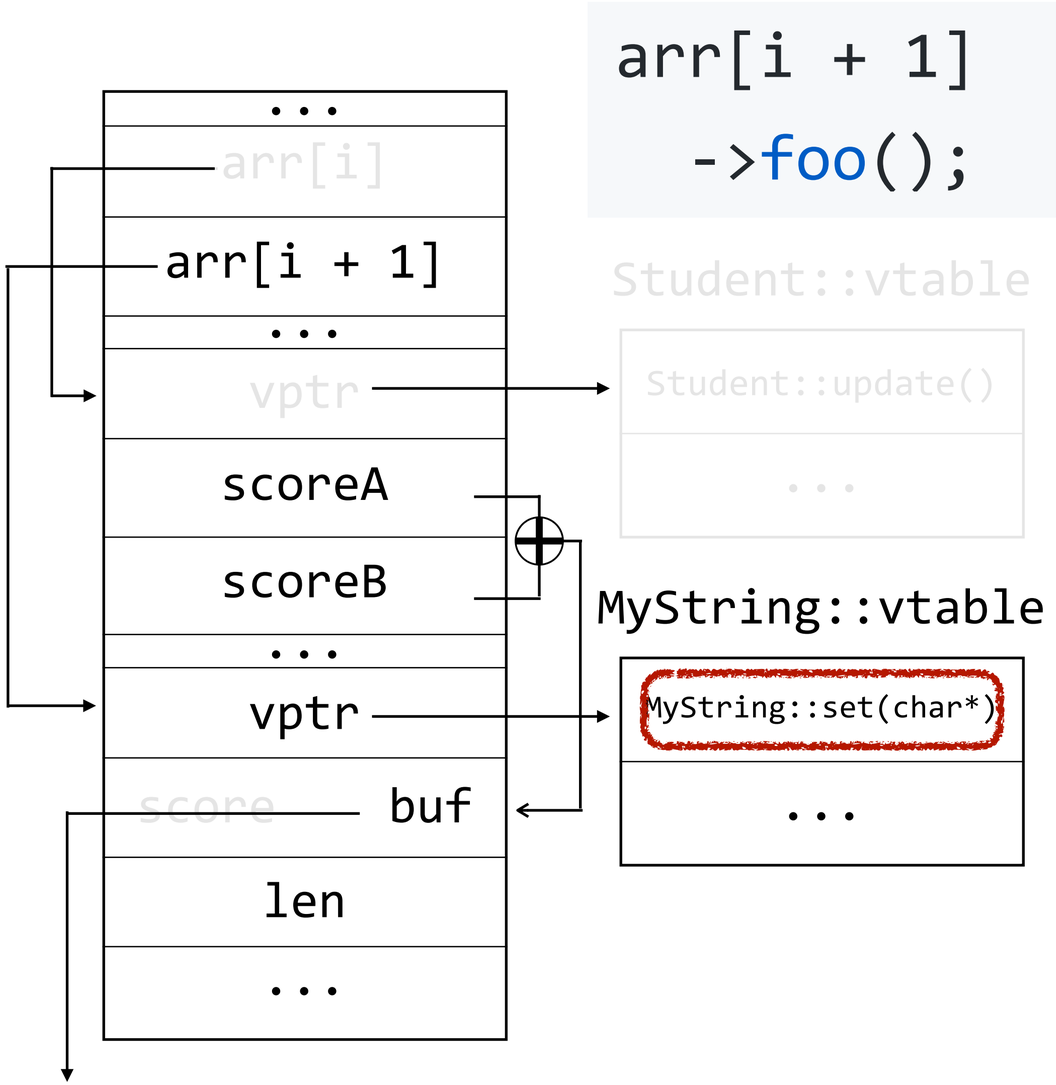
Overlapping objects allow arbitrary read/write.

```
class Student {
    int scoreA, scoreB, score;
    virtual void update() { .. }
};
class MyString {
    char *buf; int len;
    virtual void set(char *src)
    { .. }
};
```



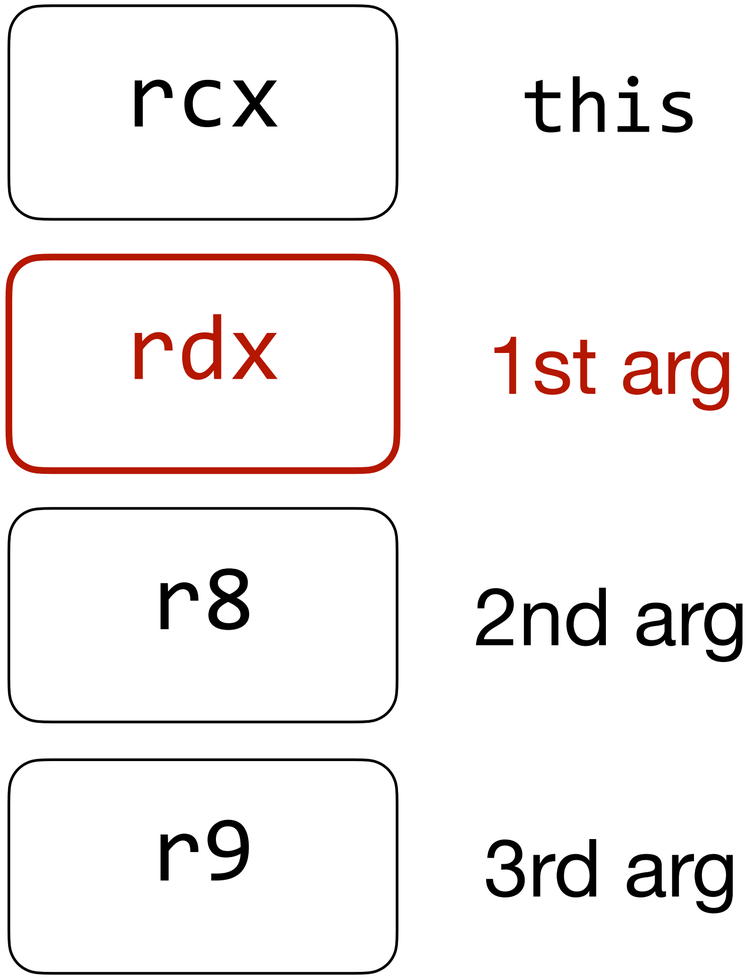
Overlapping objects allow arbitrary read/write.

```
class Student {
    int scoreA, scoreB, score;
    virtual void update() { .. }
};
class MyString {
    char *buf; int len;
    virtual void set(char *src)
    { .. }
};
```



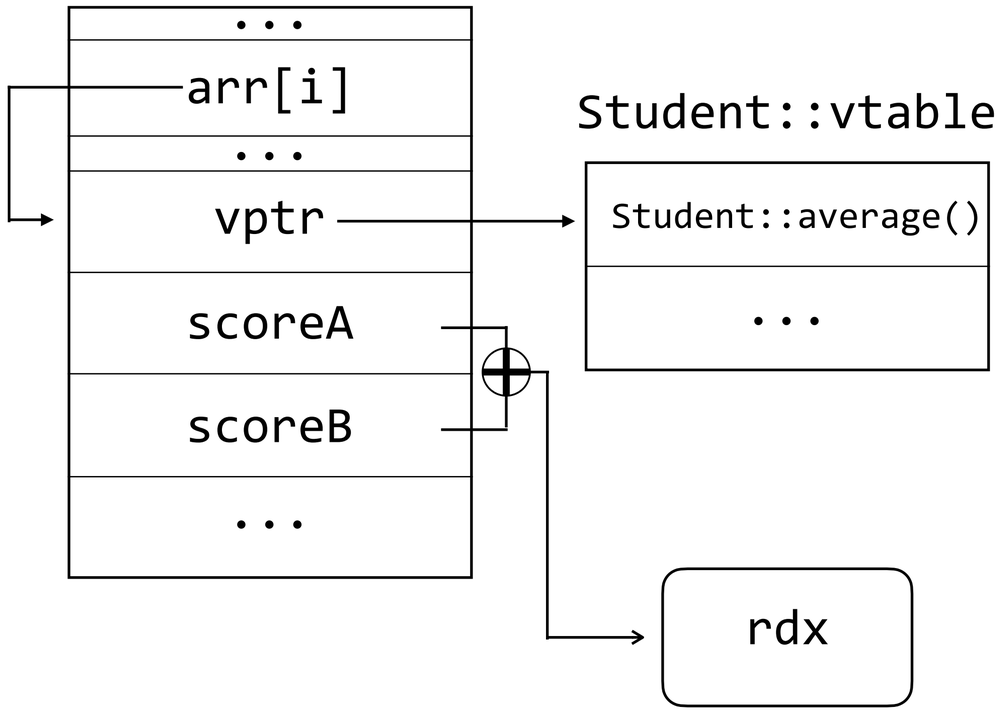
x64 uses registers to pass arguments.

```
class MyString {
    char *buf;
    int len;

    virtual void set(char *src) {
        strncpy(buf, src, len);
    }
};
```



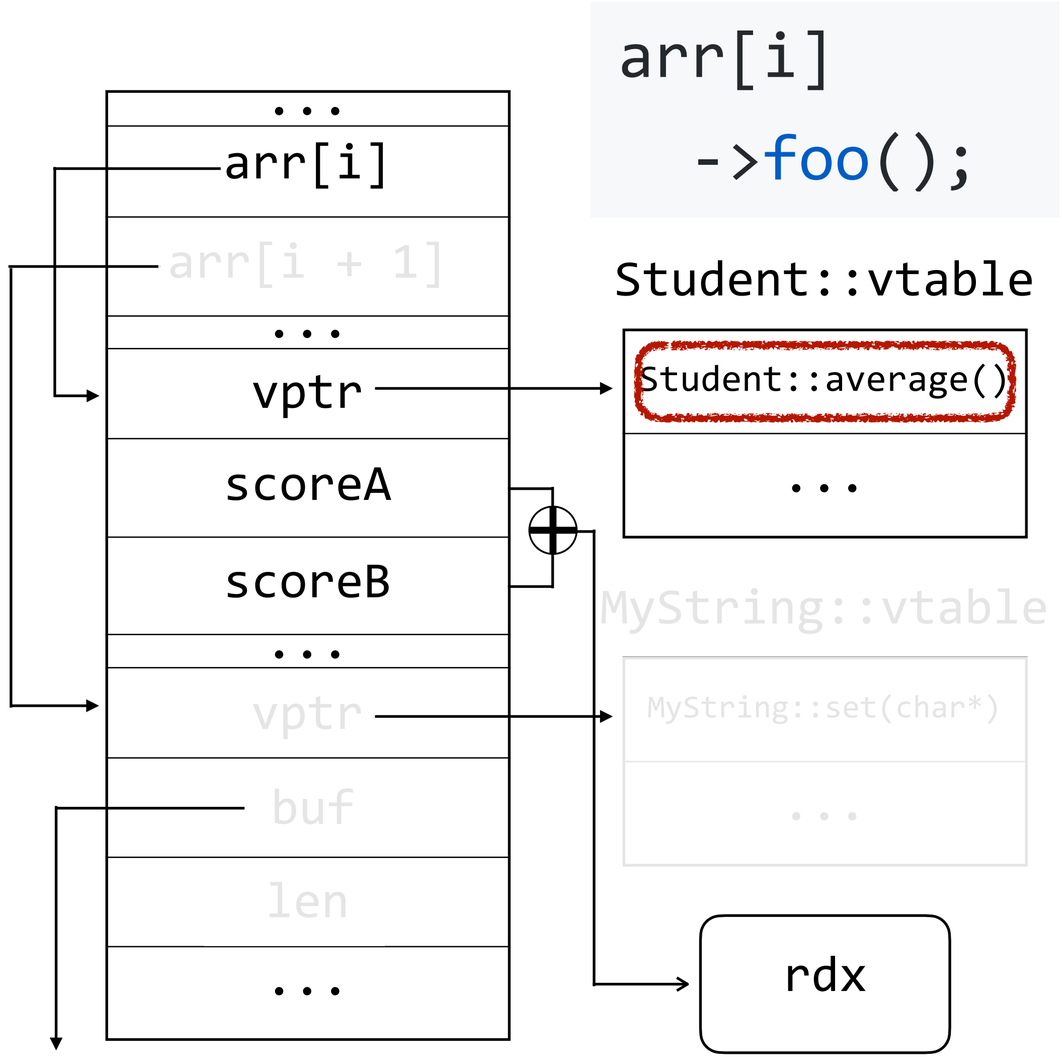
Caller-saved registers store intermediate results.

```
class Student {
    int scoreA, scoreB;

    virtual int average() {
        return (scoreA+scoreB)/2;
    }
};
```



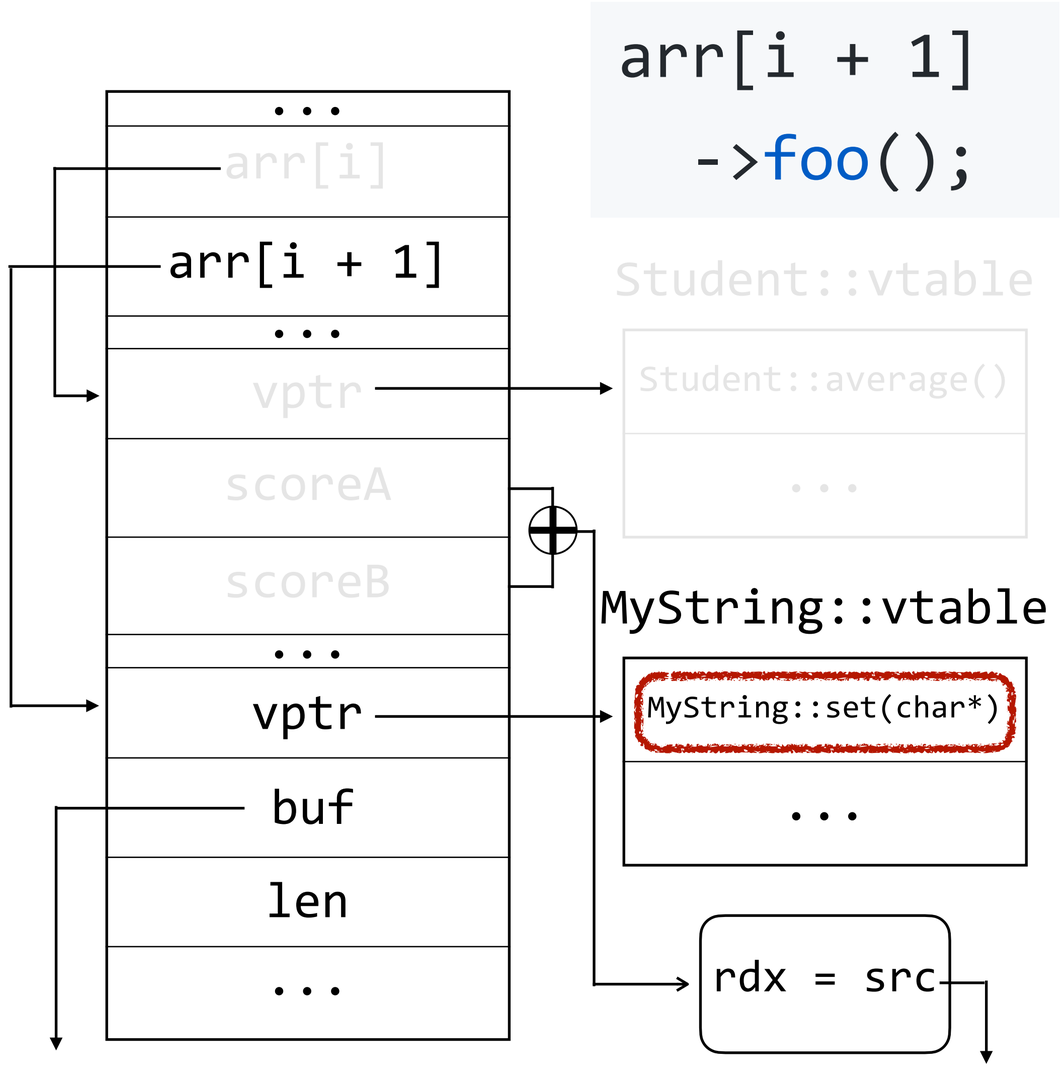
Intermediate results are arguments for the next call.

```
class Student {
    int scoreA, scoreB;
    virtual void average() { .. }
};
class MyString {
    char *buf; int len;
    virtual void set(char *src)
    { .. }
};
```



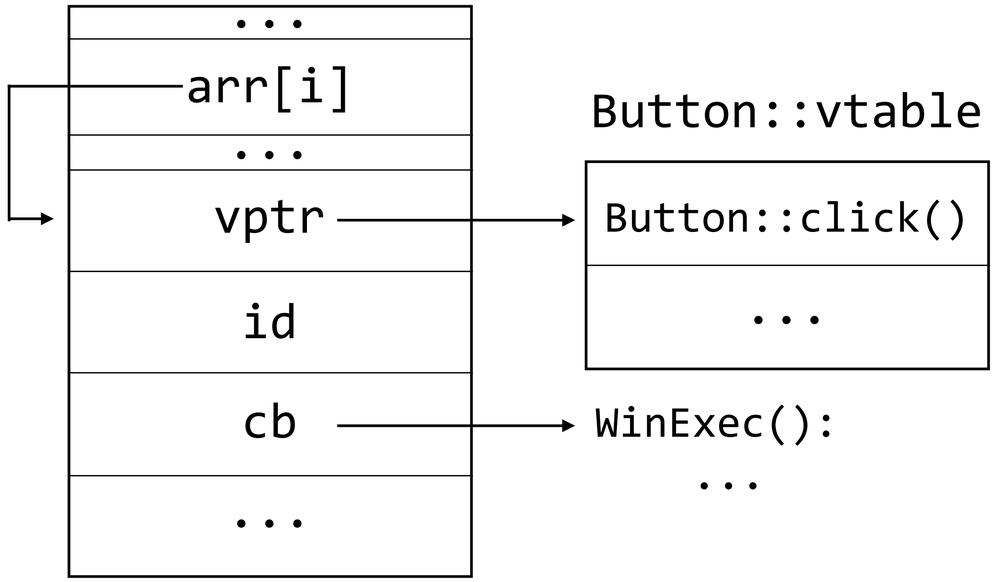
Intermediate results are arguments for the next call.

```
class Student {
    int scoreA, scoreB;
    virtual void average() { .. }
};
class MyString {
    char *buf; int len;
    virtual void set(char *src)
    { .. }
};
```



Gadgets calling function pointers can be used for system calls.

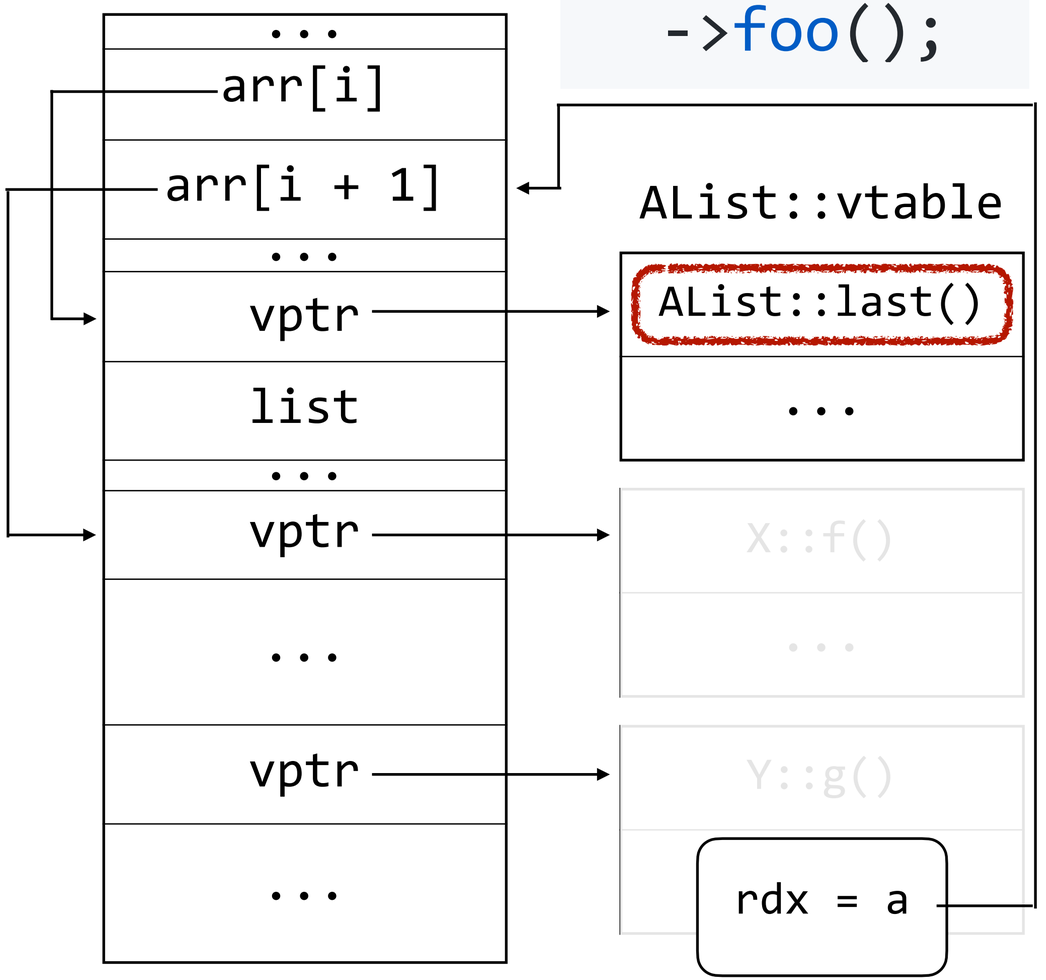
```
class Button {
    int id;
    void (* cb)(int);

    virtual void click() {
        cb(id);
    }
};
```



A conditional write is a conditional branch.

```
class AList {
    std::list<A> list;
    virtual bool last(A *a) {
        if (list->empty())
            return false;
        *a = list->back();
        return true;
    }
};
```

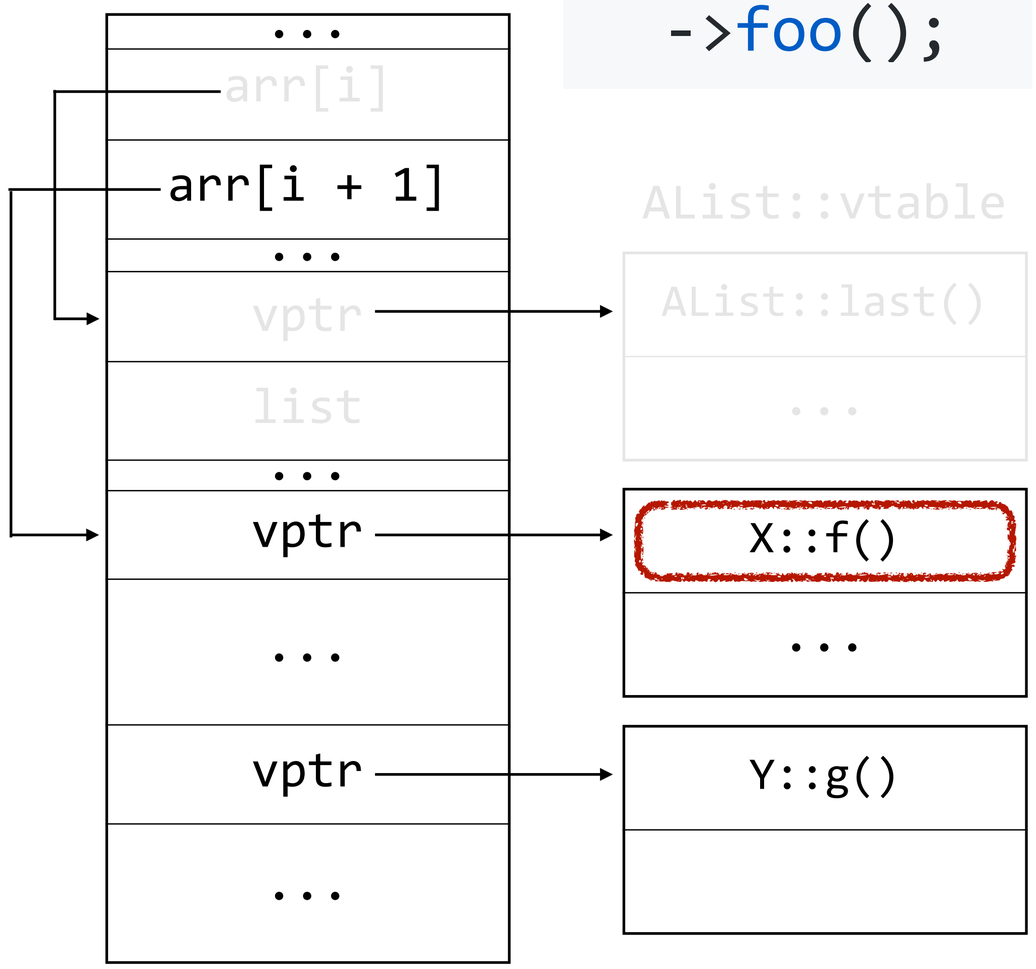
```
arr[i]
  ->foo();
```



A conditional write is a conditional branch.

```
class AList {
    std::list<A> list;
    virtual bool last(A *a) {
        if (list->empty())
            return false;
        *a = list->back();
        return true;
    }
};
```

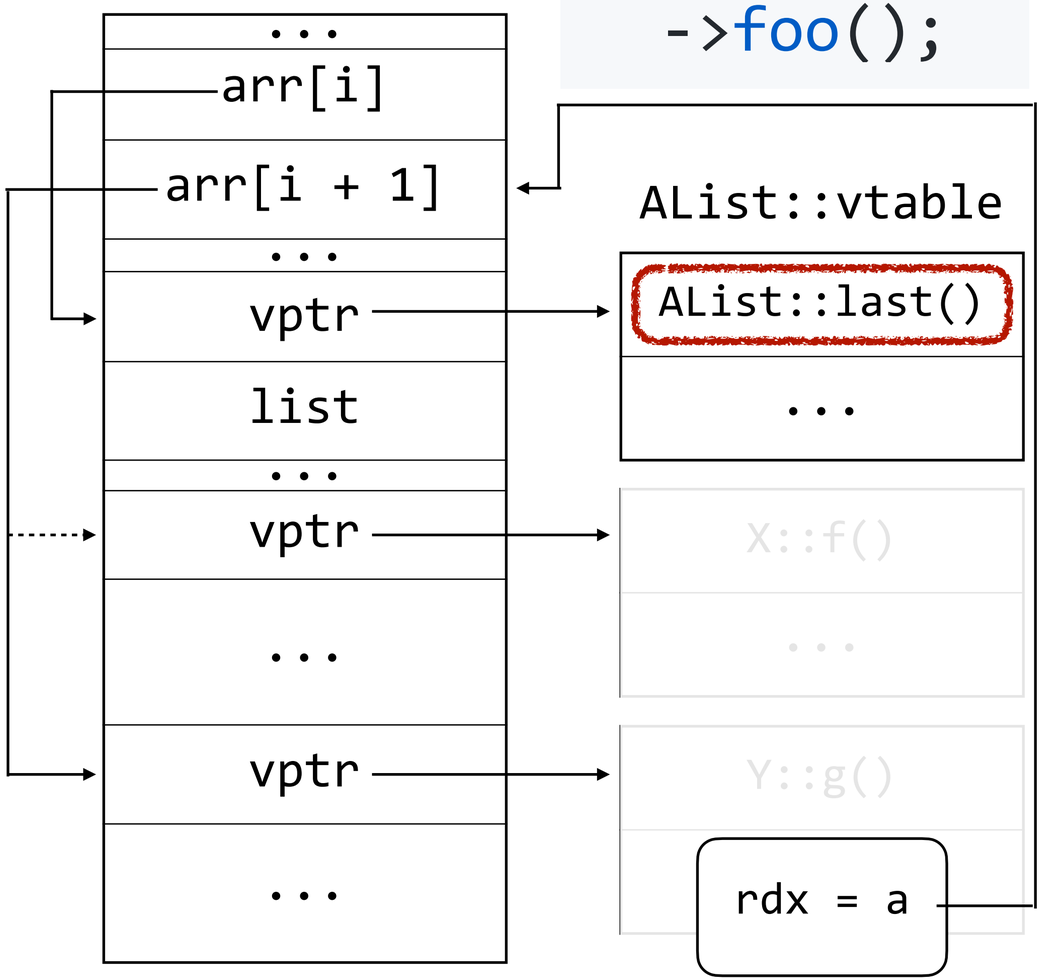
```
arr[i + 1]
  ->foo();
```



A conditional write is a conditional branch.

```
class AList {
    std::list<A> list;
    virtual bool last(A *a) {
        if (list->empty())
            return false;
        *a = list->back();
        return true;
    }
};
```

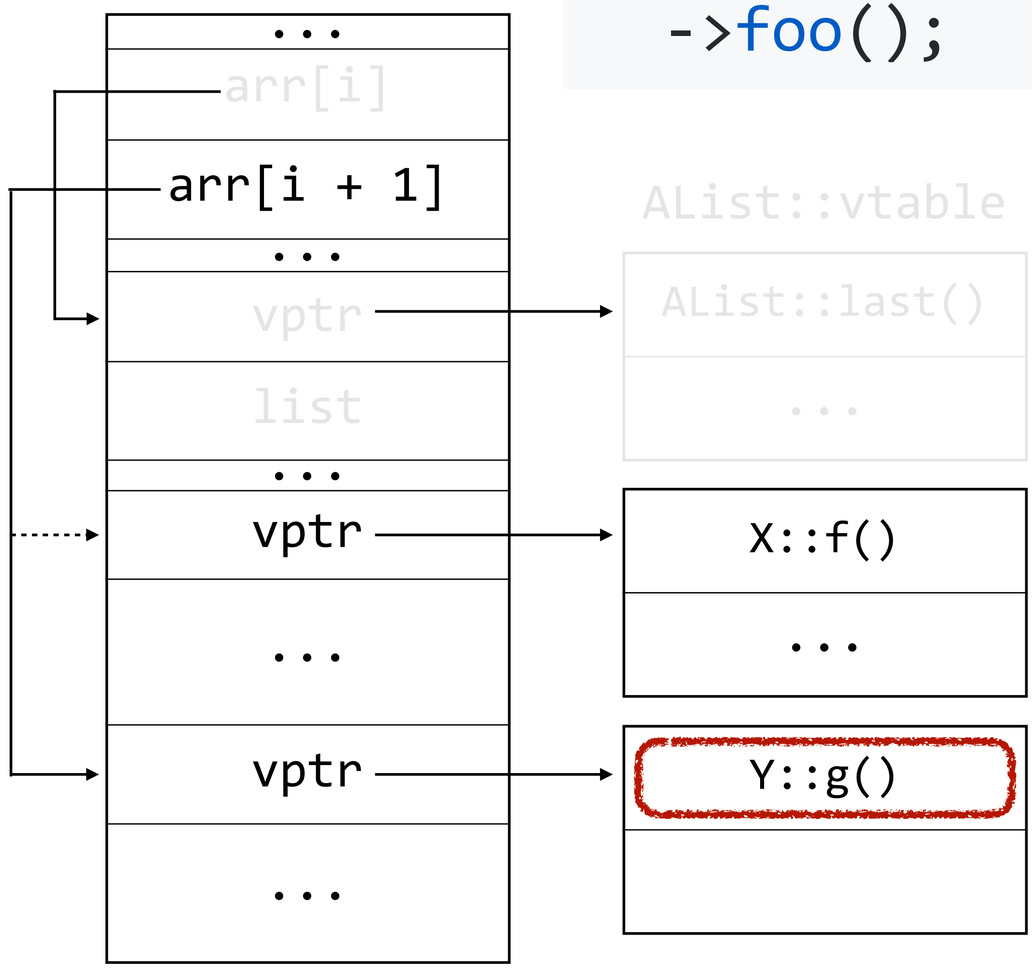
```
arr[i]
  ->foo();
```



A conditional write is a conditional branch.

```
class AList {
    std::list<A> list;
    virtual bool last(A *a) {
        if (list->empty())
            return false;
        *a = list->back();
        return true;
    }
};
```

```
arr[i + 1]
  ->foo();
```

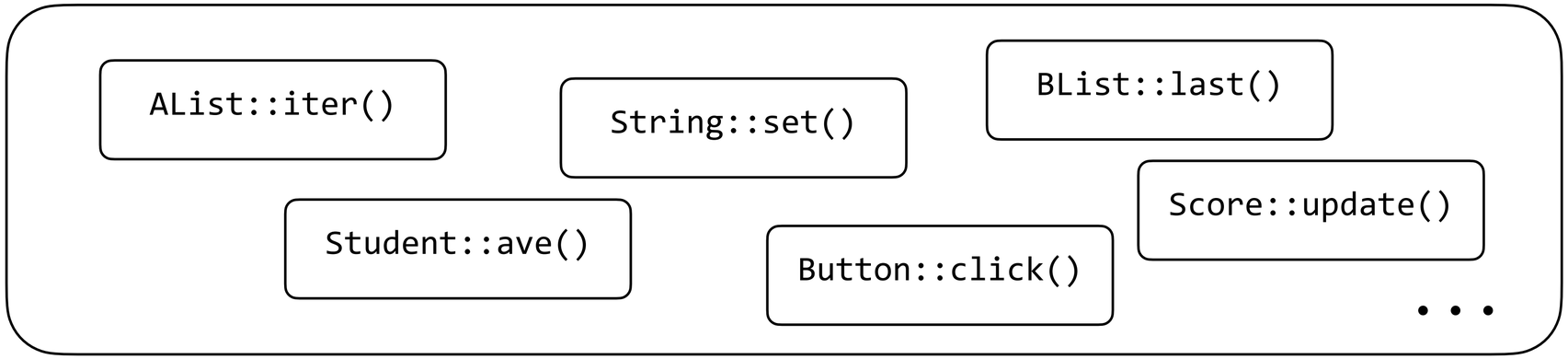


COOP achieves Turing completeness.

`AList::iter()`	Main loop
`Score::update()`	Arithmetic/logical
`String::set()`	Read/write
`Student::ave()`	Load argument
`Button::click()`	Function call
`BList::last()`	Conditional branch

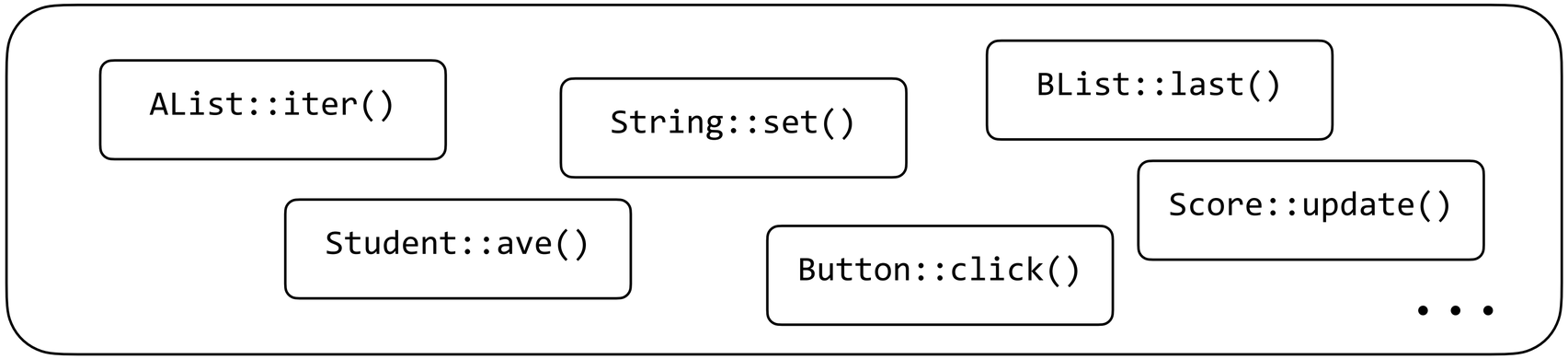
The COOP framework consists of 3 steps.

1. Find gadgets

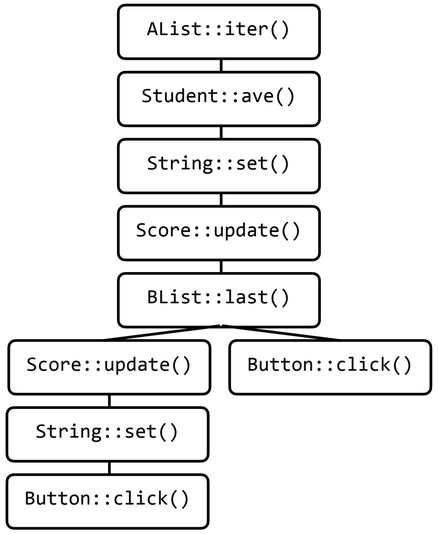


The COOP framework consists of 3 steps.

1. Find gadgets

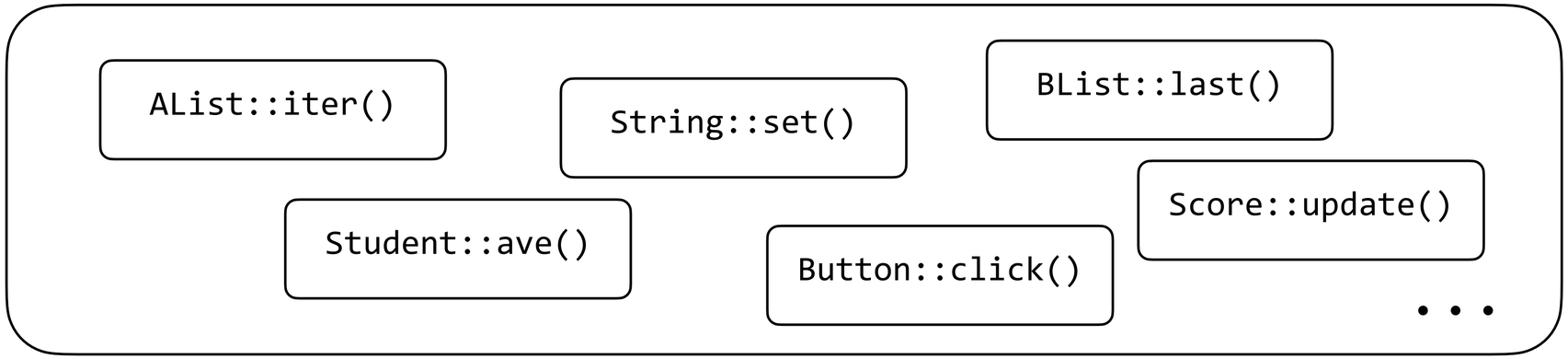


2. Design an attack

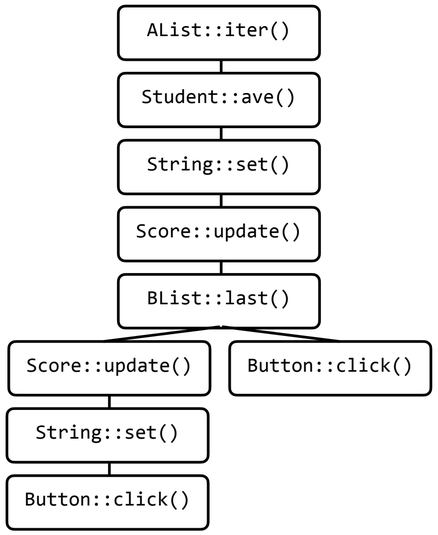


The COOP framework consists of 3 steps.

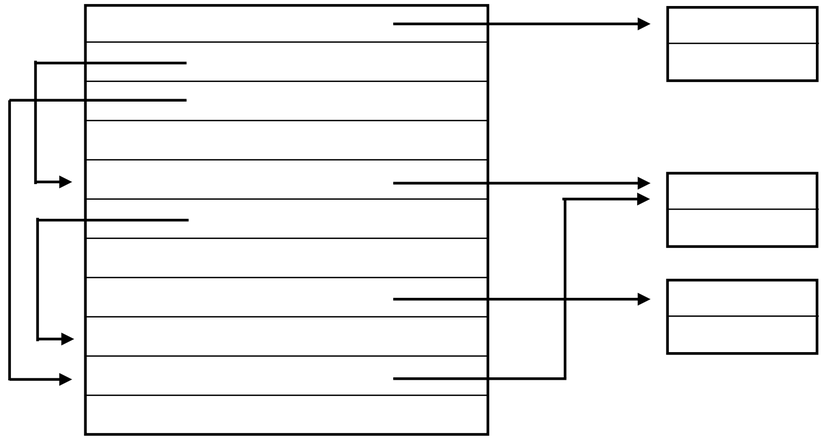
1. Find gadgets



2. Design an attack

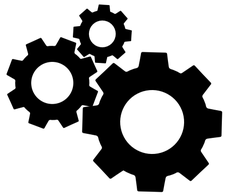


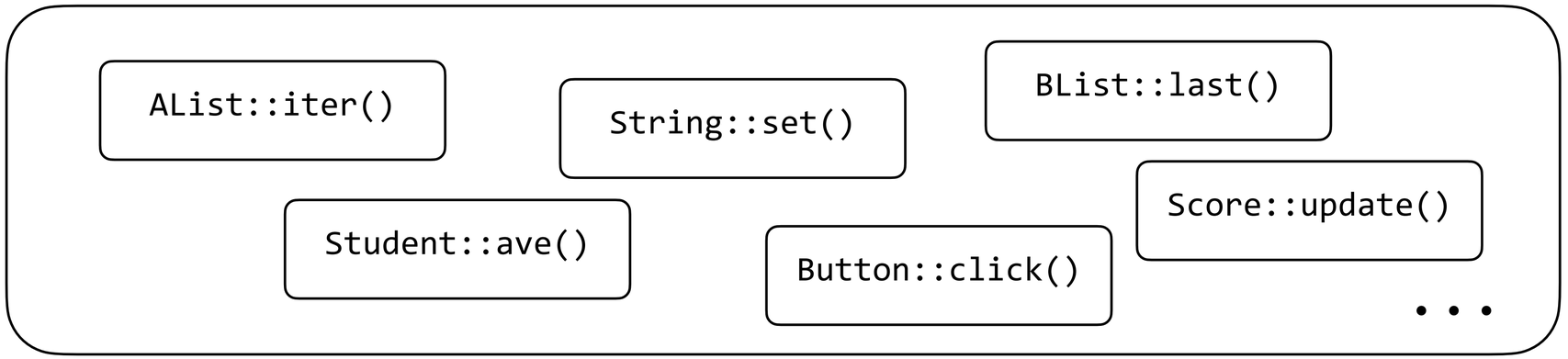
3. Arrange objects in memory



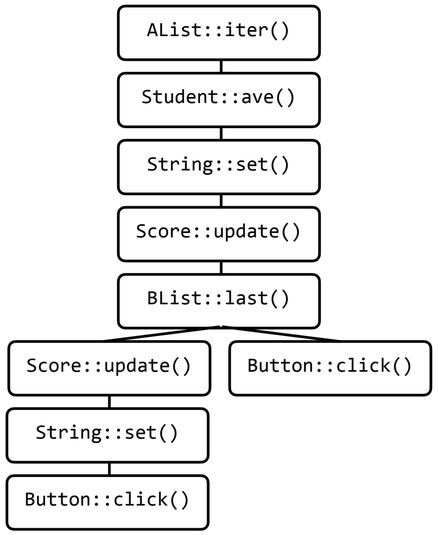
The COOP framework consists of 3 steps.

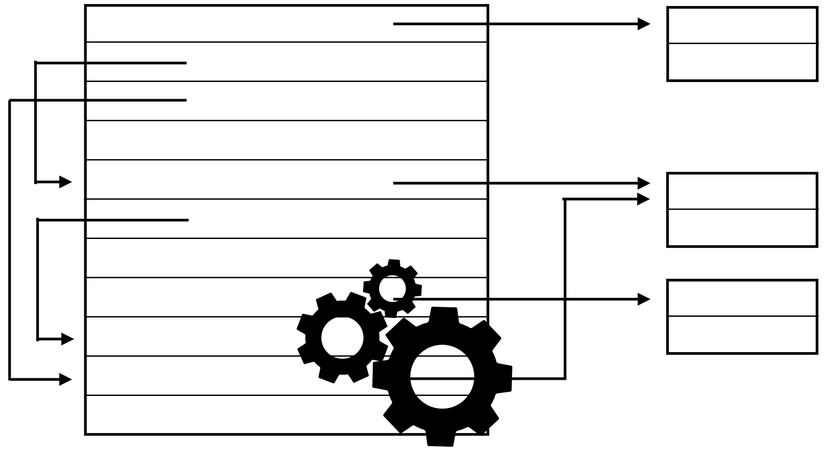
1. Find gadgets





2. Design an attack

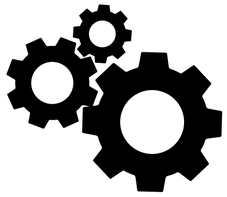


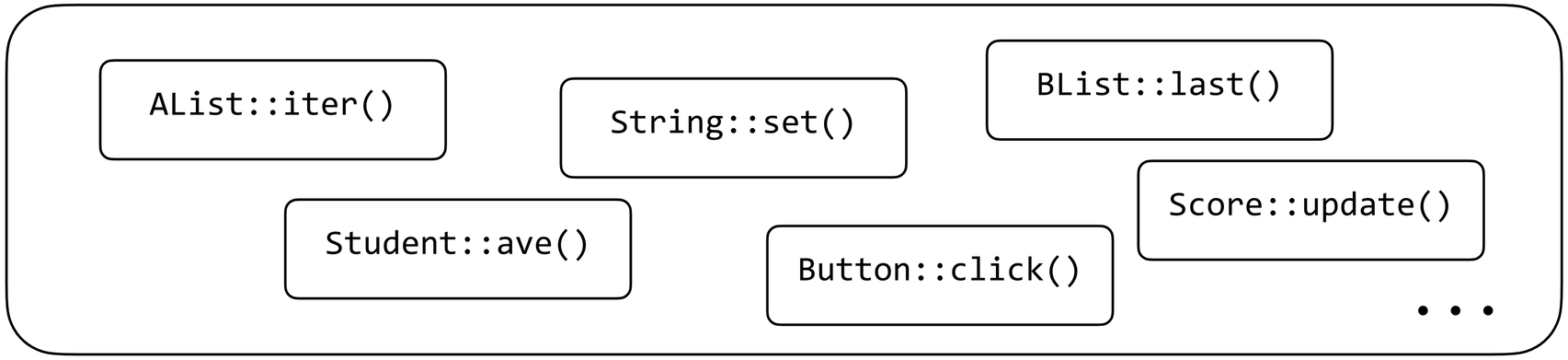


3. Arrange objects in memory

Gadgets are automatically extracted from the binary.

1. Find gadgets

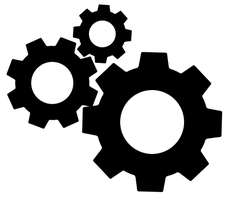


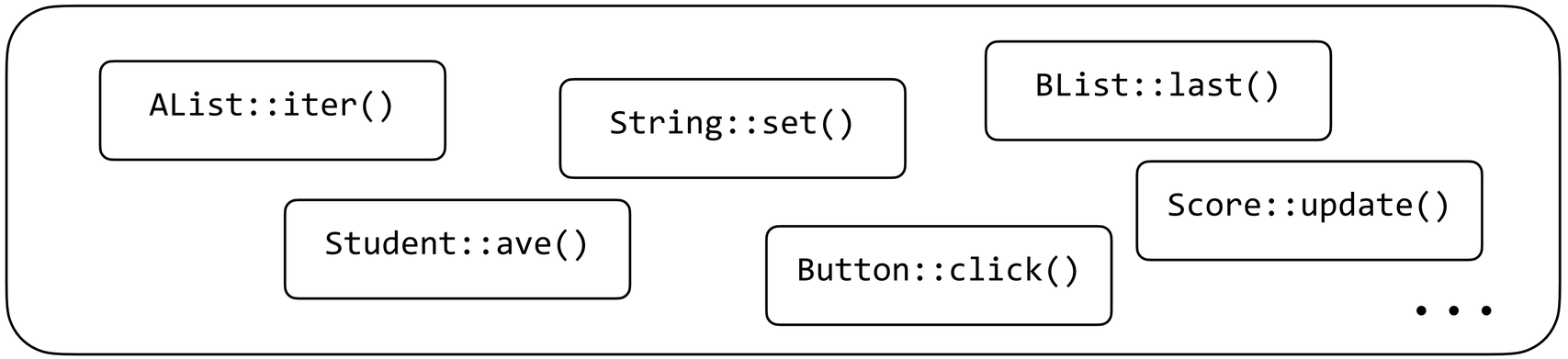


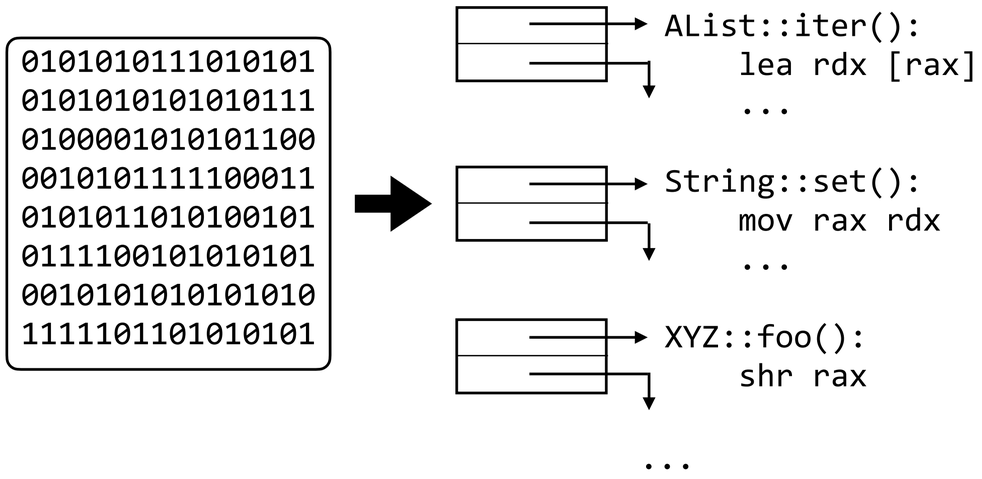

```
0101010111010101
0101010101010111
0100001010101100
0010101111100011
0101011010100101
0111100101010101
0010101010101010
1111101101010101
```

Gadgets are automatically extracted from the binary.

1. Find gadgets

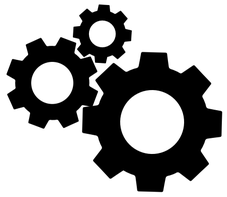


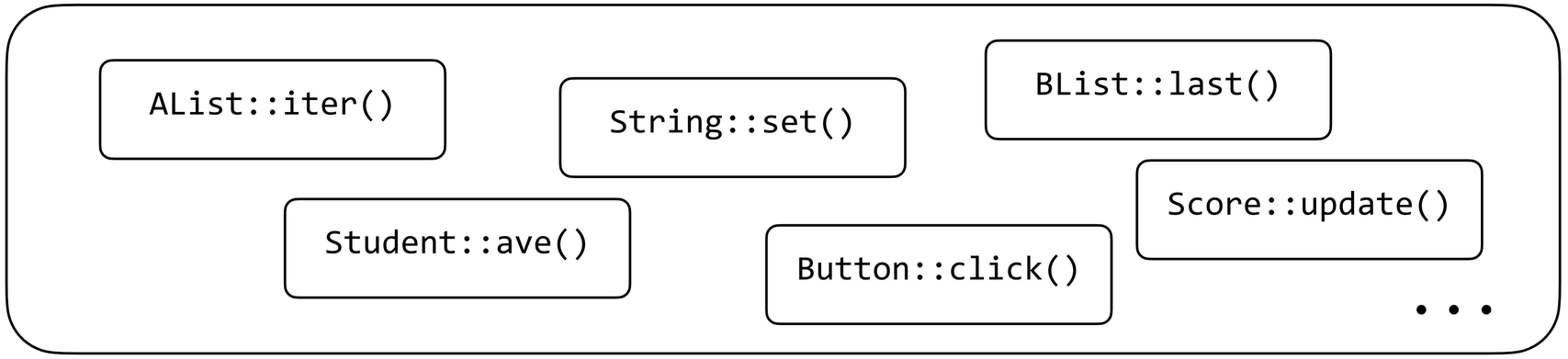


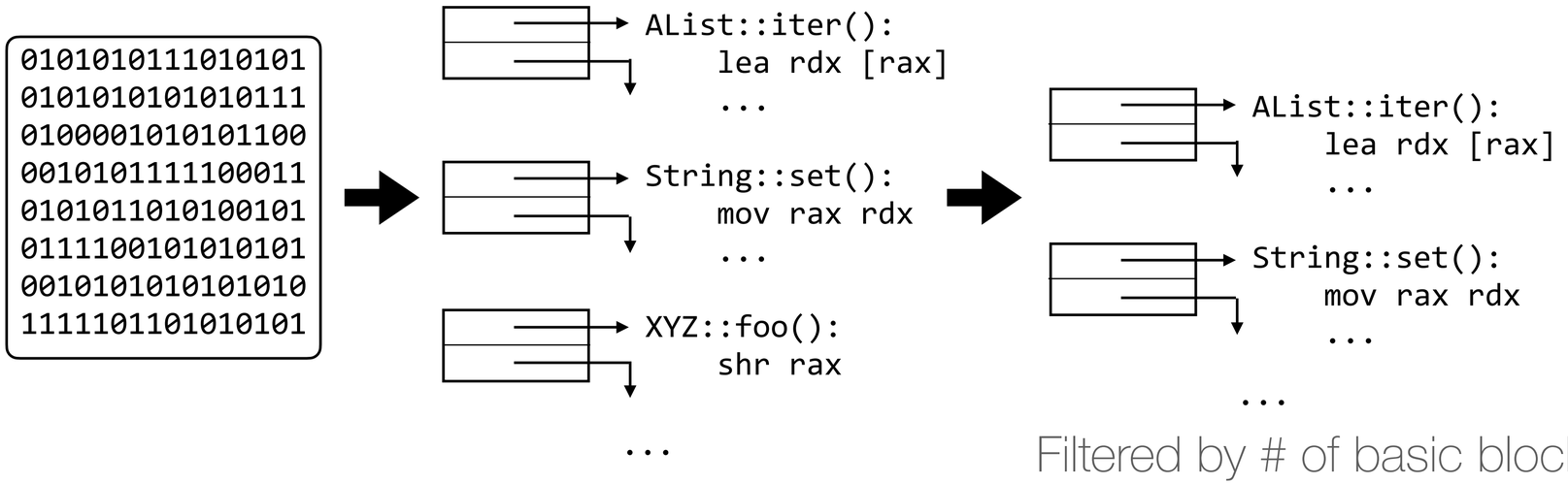


Gadgets are automatically extracted from the binary.

1. Find gadgets



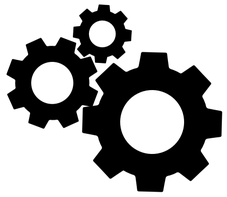


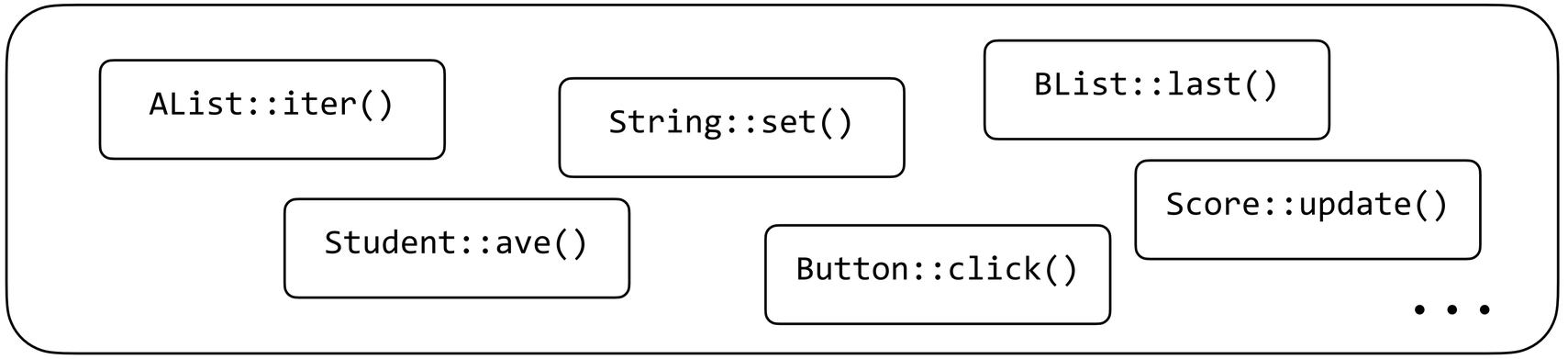


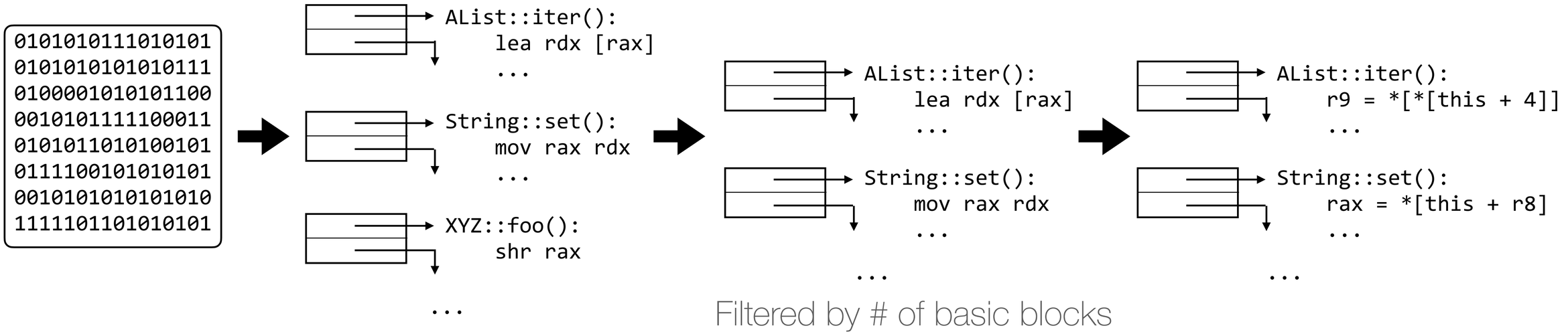
Filtered by # of basic blocks

Gadgets are automatically extracted from the binary.

1. Find gadgets



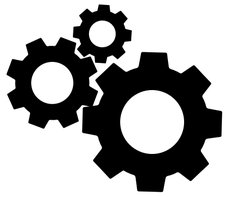


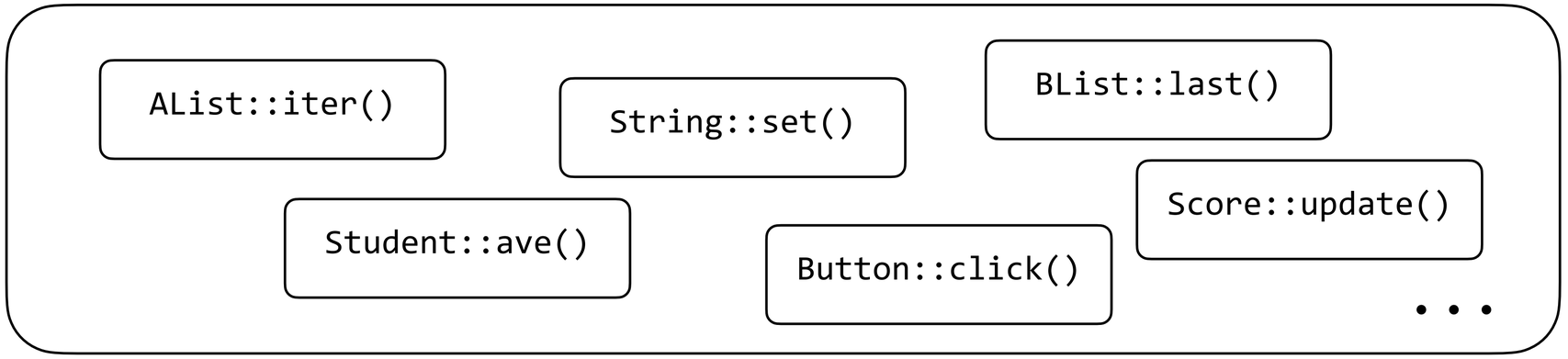


Filtered by # of basic blocks

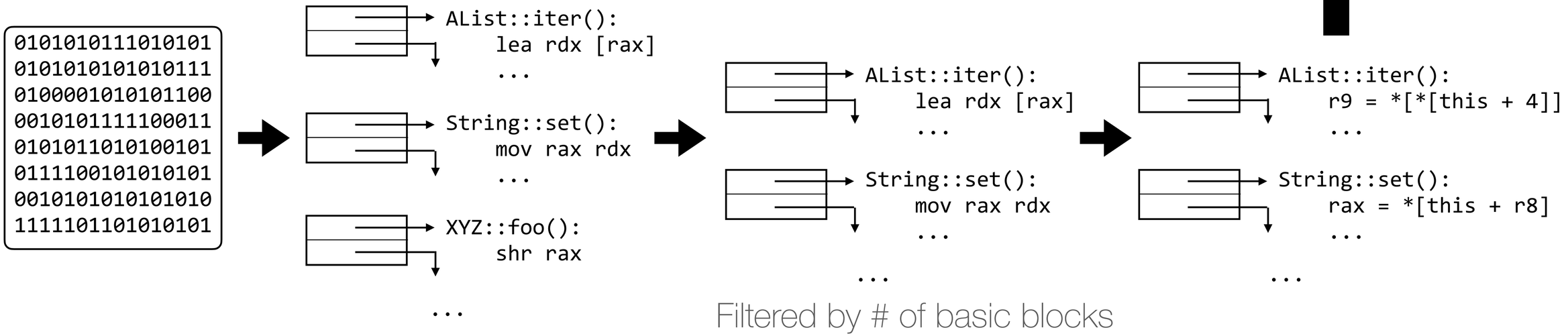
Gadgets are automatically extracted from the binary.

1. Find gadgets



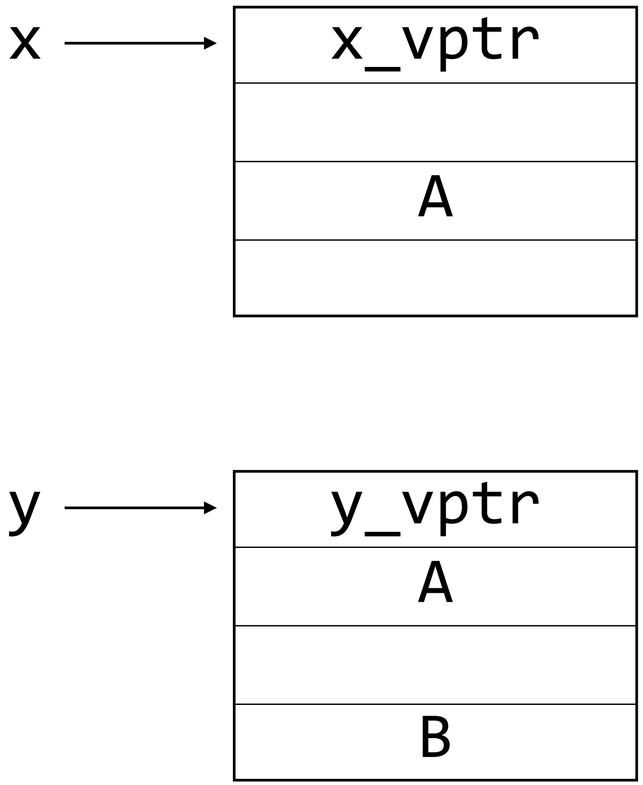


Filtered by semantics



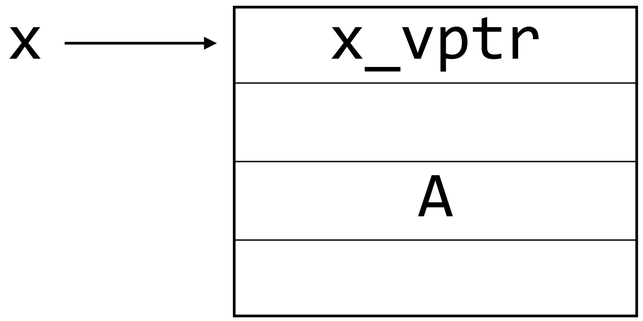
Filtered by # of basic blocks

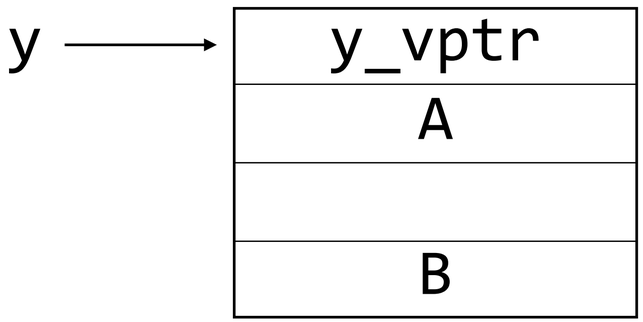
Z3 solves constraints to determine the layout.



3. Arrange objects in memory

Z3 solves constraints to determine the layout.

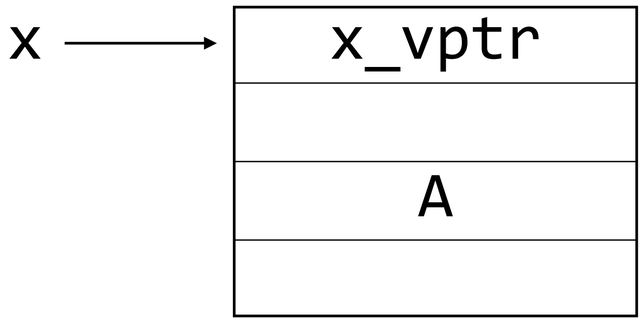




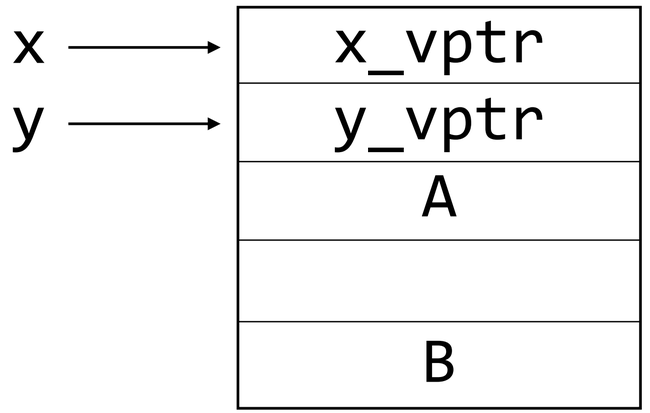
```
*x = x_vptr
*(x + 8) = A
*y = y_vptr
*(y + 4) = A
*(y + 12) = B

x + 8 = y + 4
```

3. Arrange objects in memory

Z3 solves constraints to determine the layout.




```
*x = x_vptr
*(x + 8) = A
*y = y_vptr
*(y + 4) = A
*(y + 12) = B

x + 8 = y + 4
```

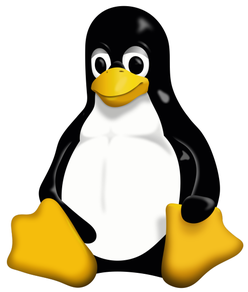



```
x = 0, y = 4
```

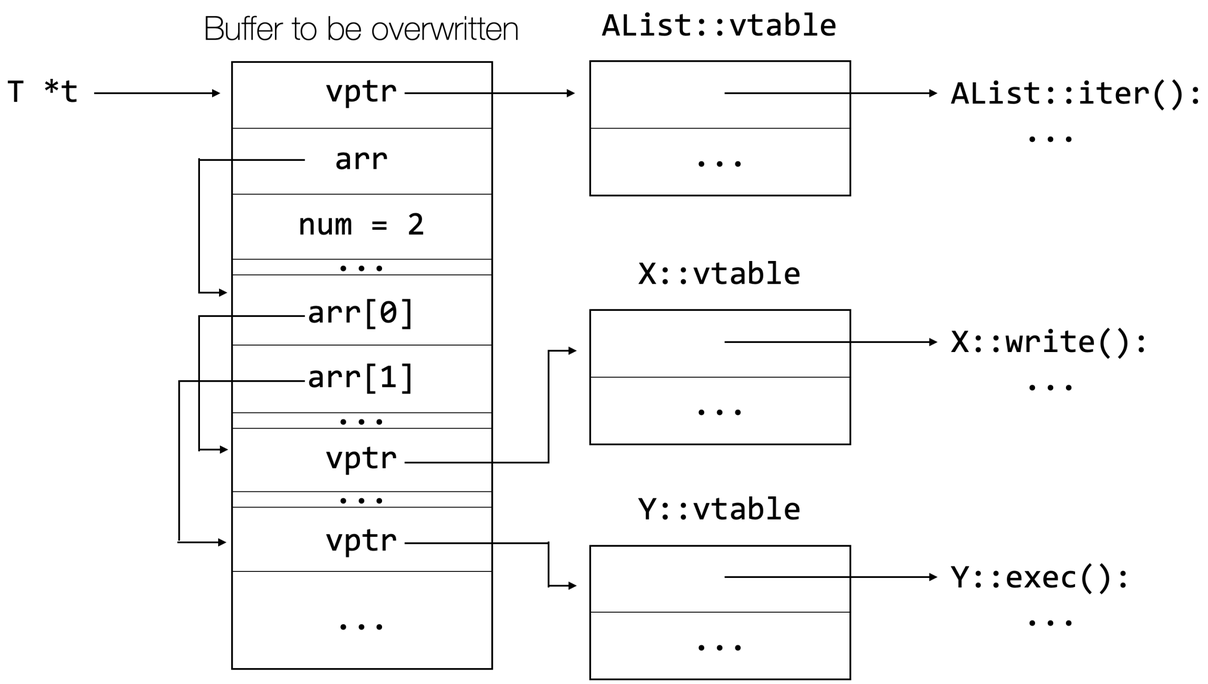
3. Arrange objects in memory

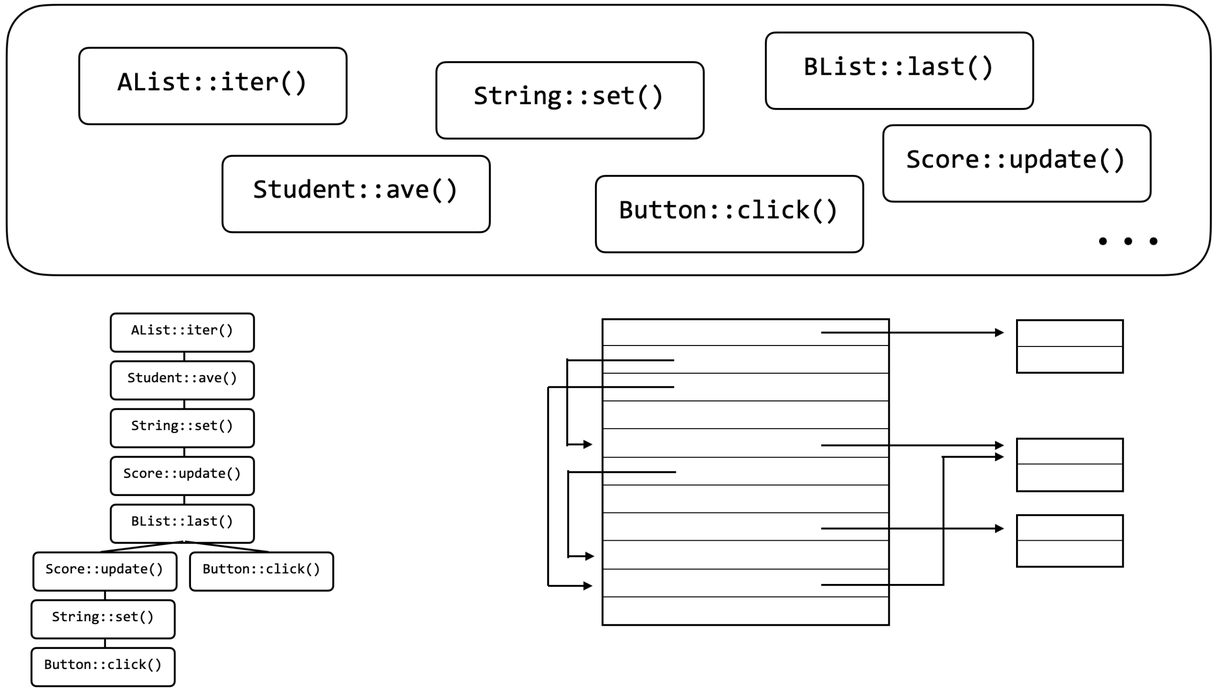
COOP is widely applicable.

x64	calc.exe / mspaint.exe
x86	calc.exe + ⊥



x64	/bin/sh





Counterfeit Object-oriented Programming