Better Code: Human Interface

Sean Parent | Principal Scientist

#AdobeRemix
Sougwen Chung

Topics for Discussion

#AdobeRemix
Sougwen Chung

Better Code: Design and Ethics

Sean Parent | Principal Scientist

#AdobeRemix
Sougwen Chung

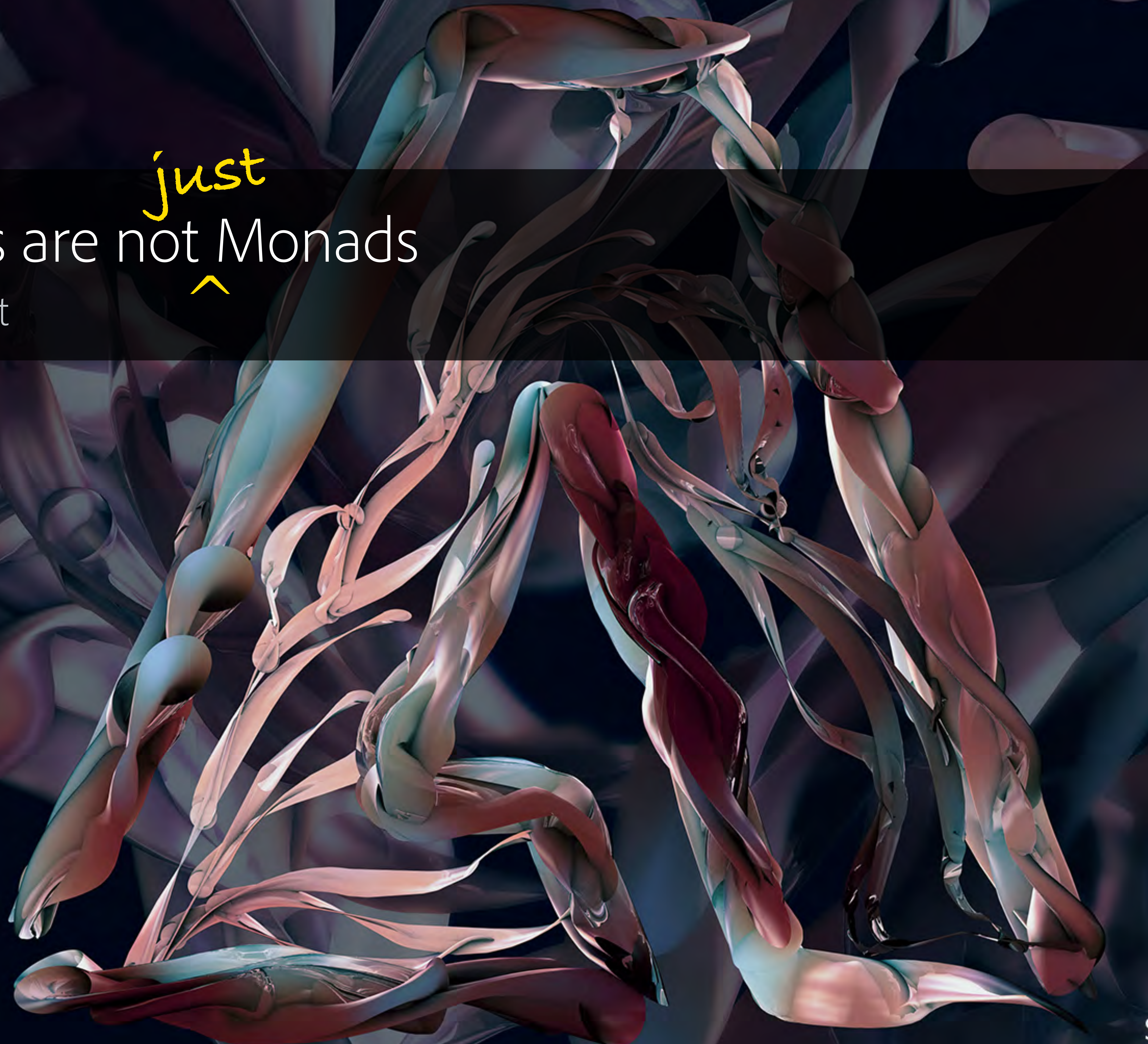
Better Code: Futures are not Monads

Sean Parent | Principal Scientist

#AdobeRemix
Sougwen Chung

Better Code: Futures are not just Monads

Sean Parent | Principal Scientist



#AdobeRemix
Sougwen Chung

Faster Bresenham's Algorithm

Sean Parent | Principal Scientist

#AdobeRemix
Sougwen Chung

Old Guy Reminiscing

Sean Parent | Principal Scientist

#AdobeRemix
Sougwen Chung

Today's Talk

#AdobeRemix
Sougwen Chung

Better Code: Human Interface

Sean Parent | Principal Scientist

#AdobeRemix
Sougwen Chung

Relationship Between HI and Code

> "The purpose of a human interface is not to hide what the code does but to accurately convey what the code does."

– Darin Adler (personal conversation, best of my recollection)

Goal: Don’t Lie

#AdobeRemix
Sougwen Chung

Demo

Photoshop

#AdobeRemix
Sougwen Chung

Taxonomy of Everything

Taxonomy of Everything

- Objects

Taxonomy of Everything

- Objects
  - Properties

Taxonomy of Everything

- Collections
  - Objects
    - Properties

Taxonomy of Everything

- Collections
  - Objects
    - Properties
- Operations

Taxonomy of Everything

- Collections
  - Objects
    - Properties
- Operations
- Relationships

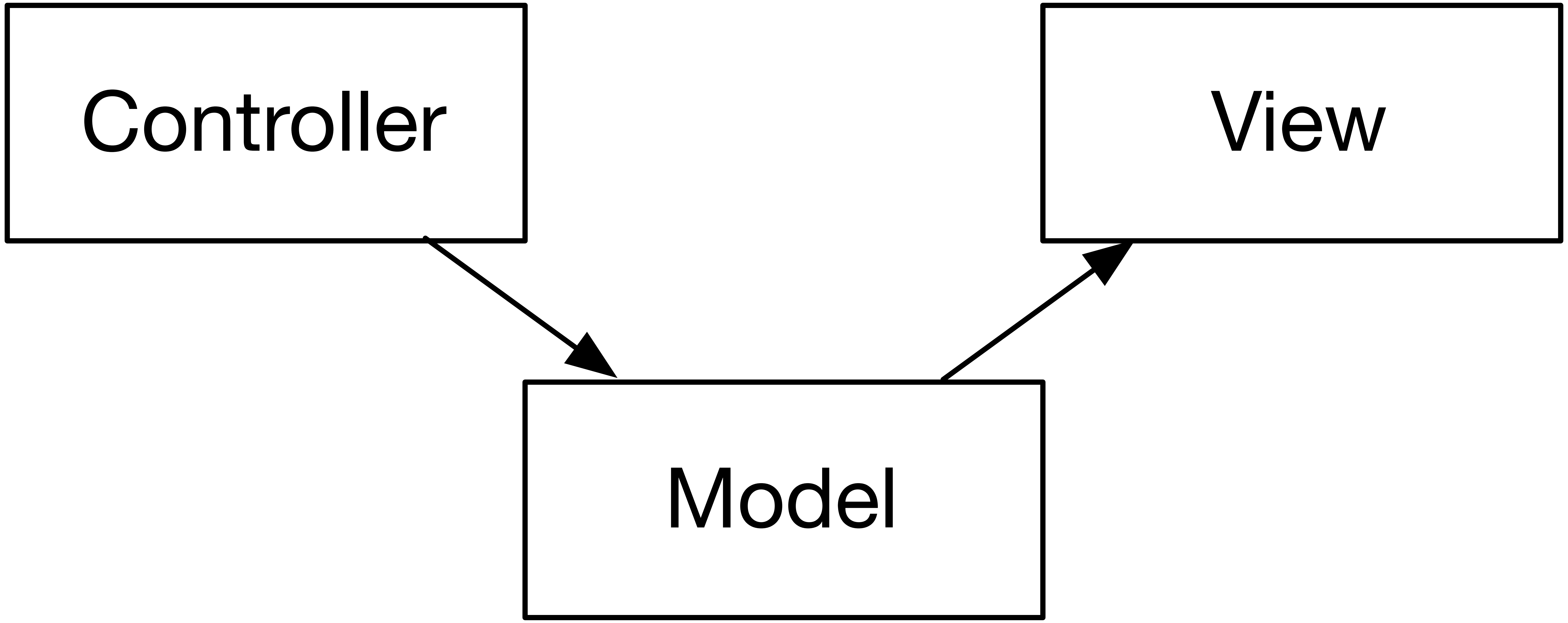
Model View Controller

Model View Controller

“MVC consists of three kinds of objects. The Model is the application object, the View is its screen presentation, and the Controller defines the way the user interface reacts to user input.”

– *Design Patterns: Elements of Reusable Object-Oriented Software,* section 1.2

Model-View-Controller



Controller

View

Model

How did MVC get so F’ed up?

How did MVC get so F’ed up?

- http://stlab.cc/tips/about-mvc.html

Observable Models

- Application model is Objects + Operations + Relationships

Observable Models

- Application model is Objects + Operations + Relationships
- Controllers bind to operations

Observable Models

- Application model is Objects + Operations + Relationships
- Controllers bind to operations
  - Trivial controller binds to *set property*

Observable Models

- Application model is Objects + Operations + Relationships
- Controllers bind to operations
  - Trivial controller binds to *set property*
- Views bind to objects and their properties

Observable Models

- Application model is Objects + Operations + Relationships
- Controllers bind to operations
  - Trivial controller binds to *set property*
- Views bind to objects and their properties

- A view/controller is a *control* or *widget*

Objects

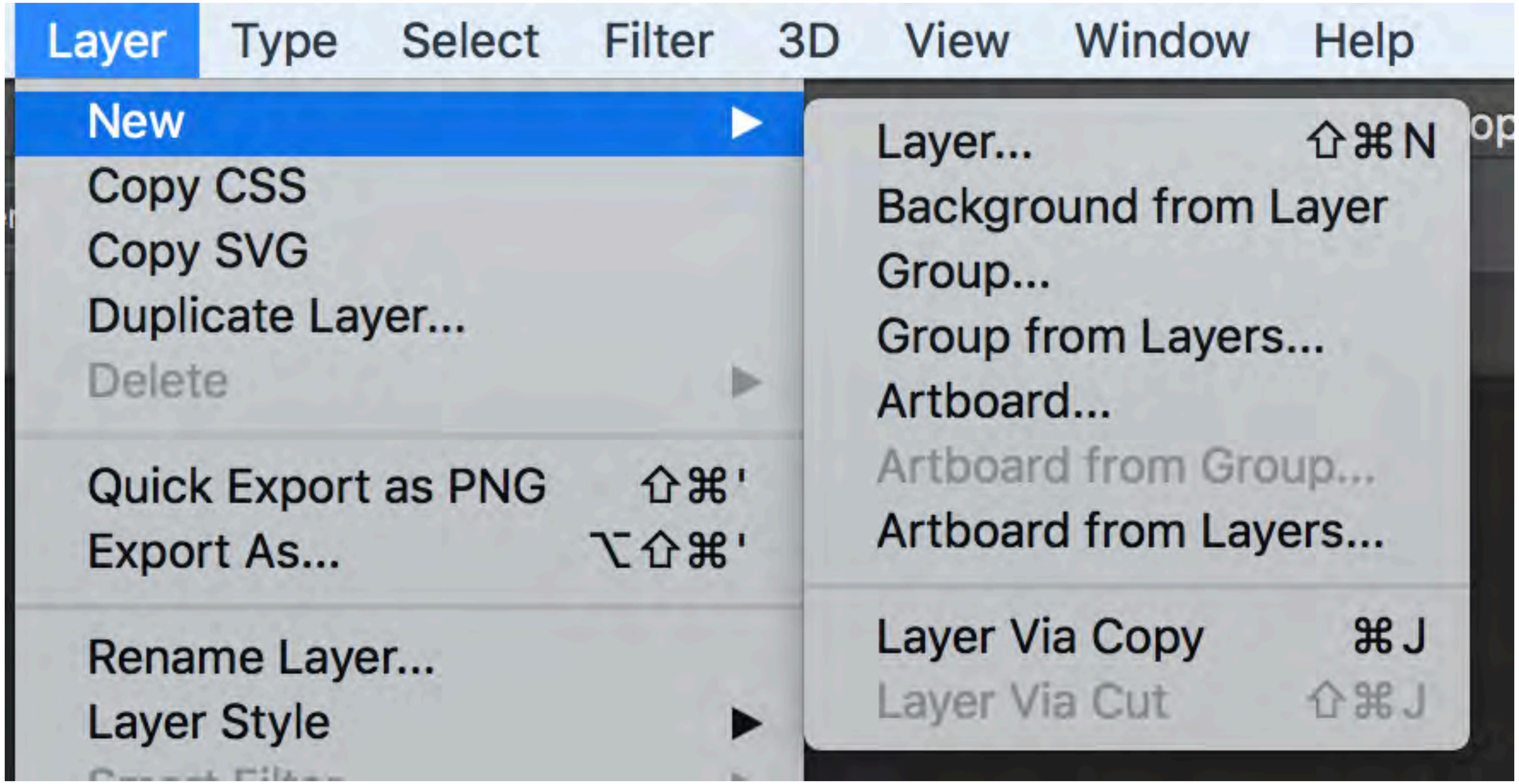
- Operations
  - Construct
  - Copy
  - Move
  - Delete

- Properties
  - Location
  - Size
  - Name (common)

Objects

- Operations
  - Construct
  - Copy
  - Move
  - Delete

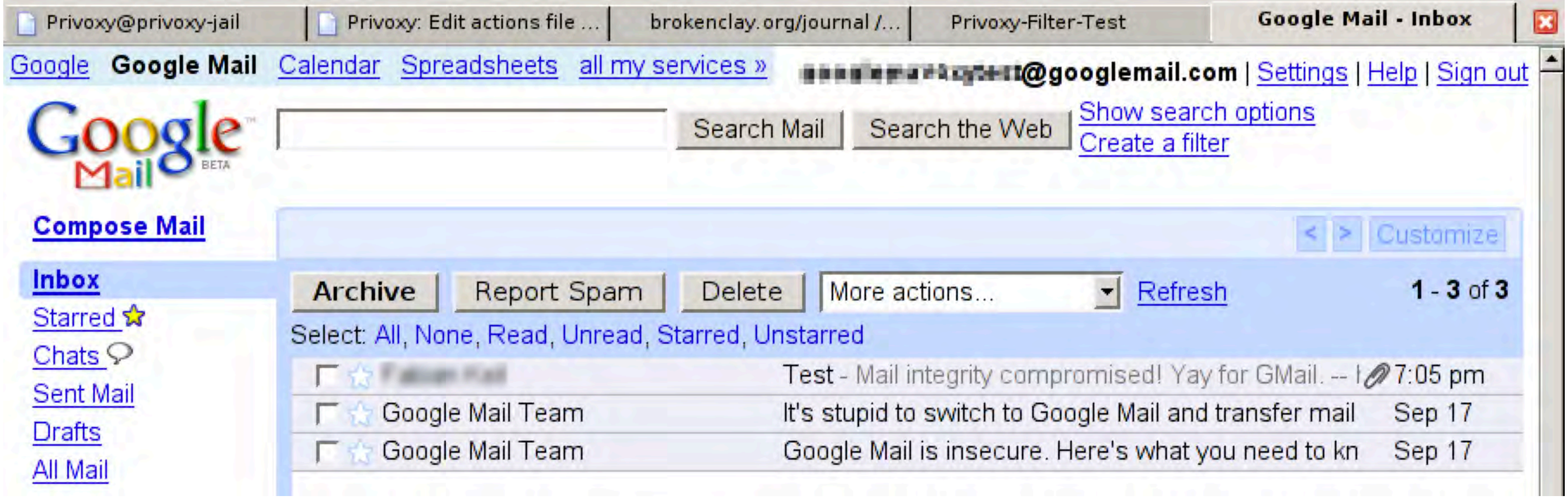
- Properties
  - Location
  - Size
  - Name (common)



Objects

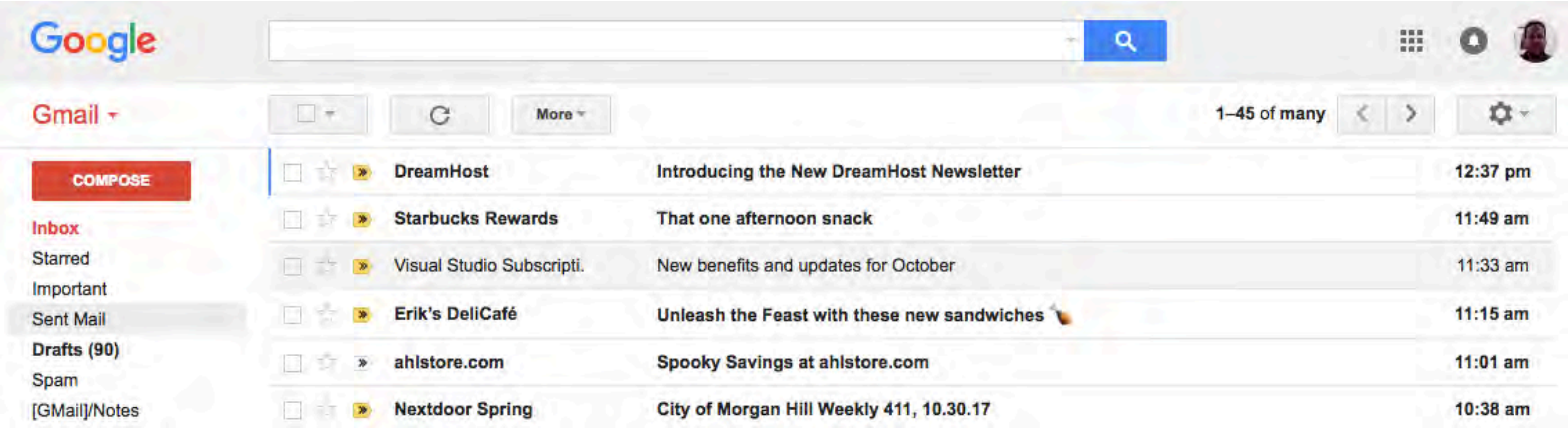
- We associate visual constructs, names, icons, and behaviors with semantics
  - In programs operations like *construct* have specific semantics
  - In the HI we associate semantics with controls

Objects



☐	[redacted]	Test - Mail integrity compromised! Yay for GMail. -- I	7:05 pm
☐	Google Mail Team	It's stupid to switch to Google Mail and transfer mail	Sep 17
☐	Google Mail Team	Google Mail is insecure. Here's what you need to kn	Sep 17

Objects



DreamHost	Introducing the New DreamHost Newsletter	12:37 pm
Starbucks Rewards	That one afternoon snack	11:49 am
Visual Studio Subscripti.	New benefits and updates for October	11:33 am
Erik's DeliCafé	Unleash the Feast with these new sandwiches	11:15 am
ahlstore.com	Spooky Savings at ahlstore.com	11:01 am
Nextdoor Spring	City of Morgan Hill Weekly 411, 10.30.17	10:38 am

Collections

- Operations
  - Insert
  - Remove

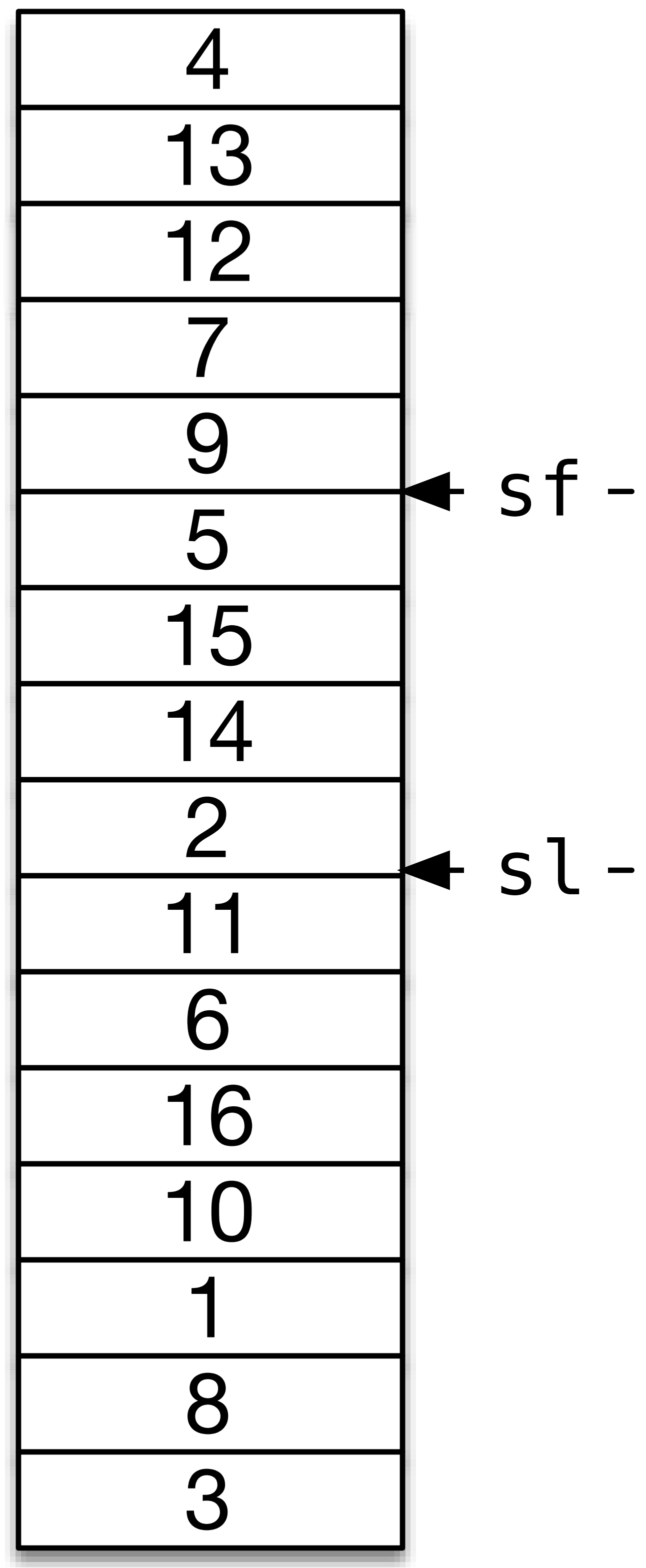
- Properties
  - Count

- Relationships
  - Whole/Part

Collections

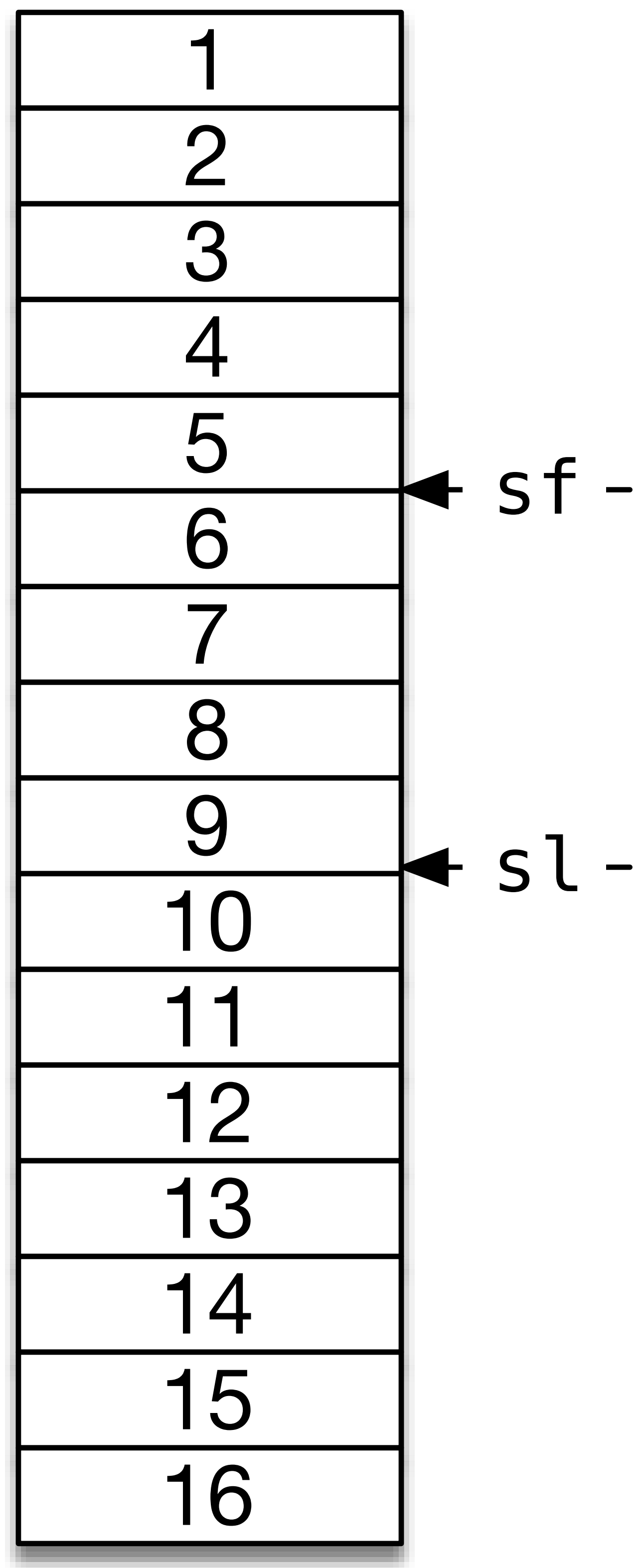
- Large collections pose a problem
  - How to observe the collection interactively, allowing the user to arrange, filter, and browse

Observing Collections

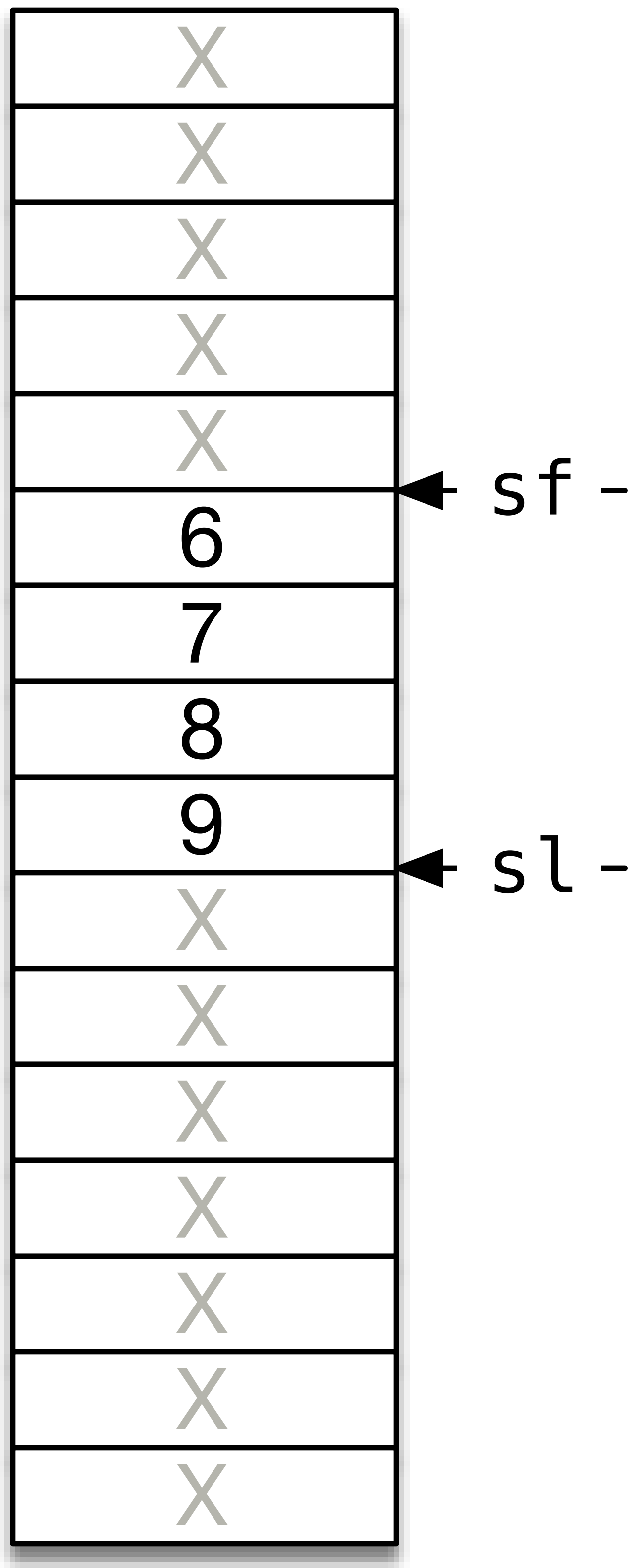


4
13
12
7
9
5
15
14
2
11
6
16
10
1
8
3

Observing Collections



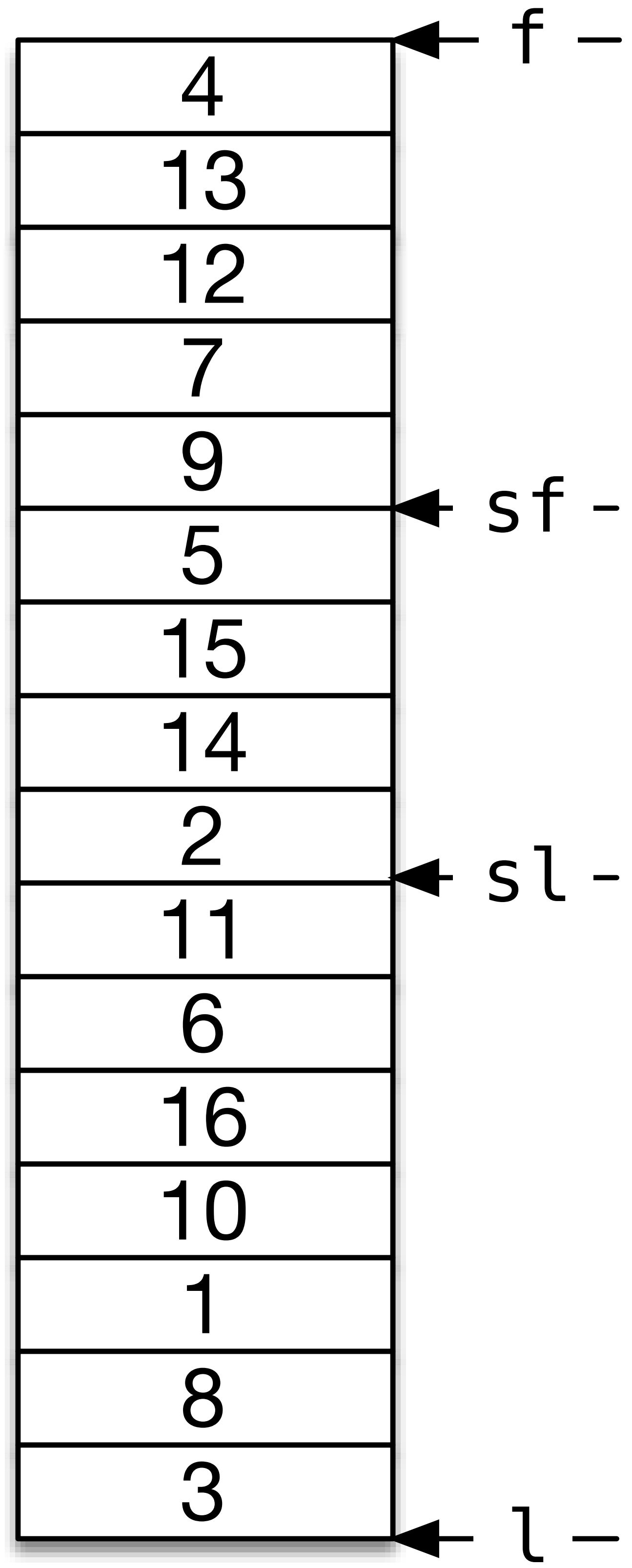
Observing Collections



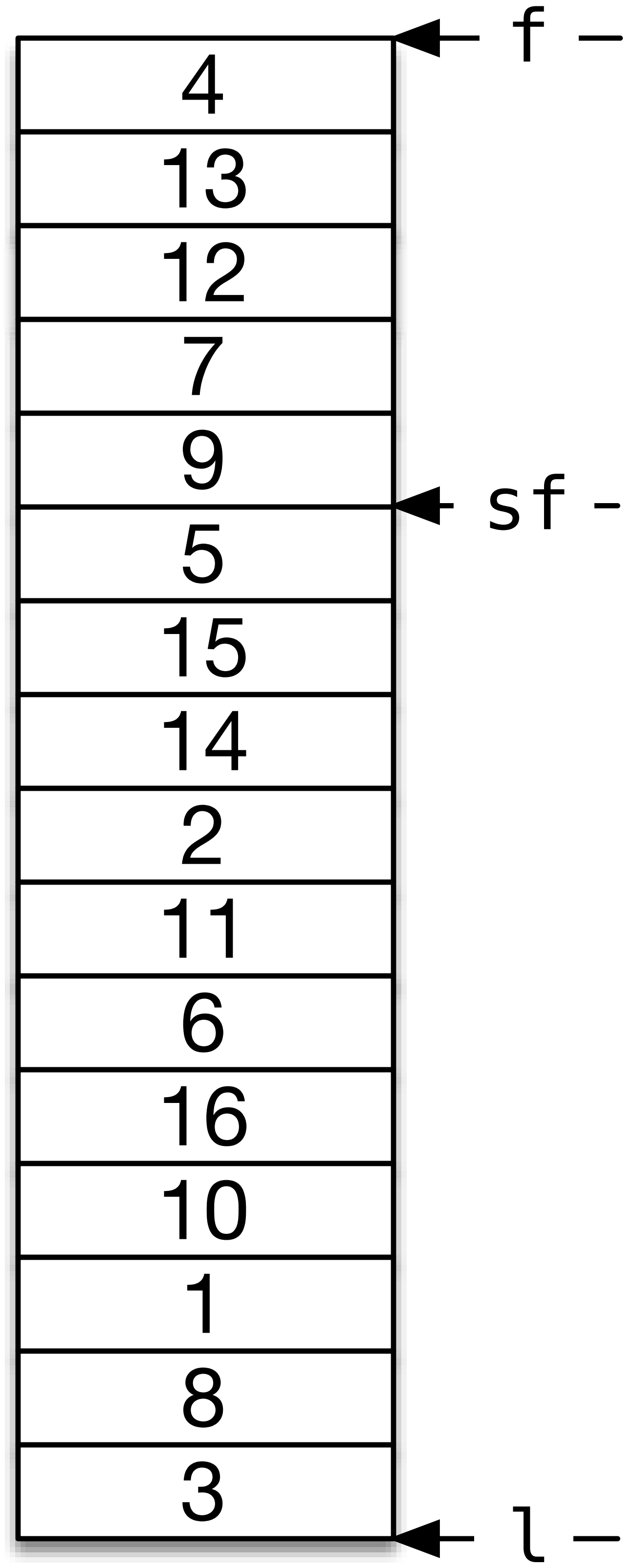
X
X
X
X
X
6
7
8
9
X
X
X
X
X
X
X

sf -
sl -

Observing Collections

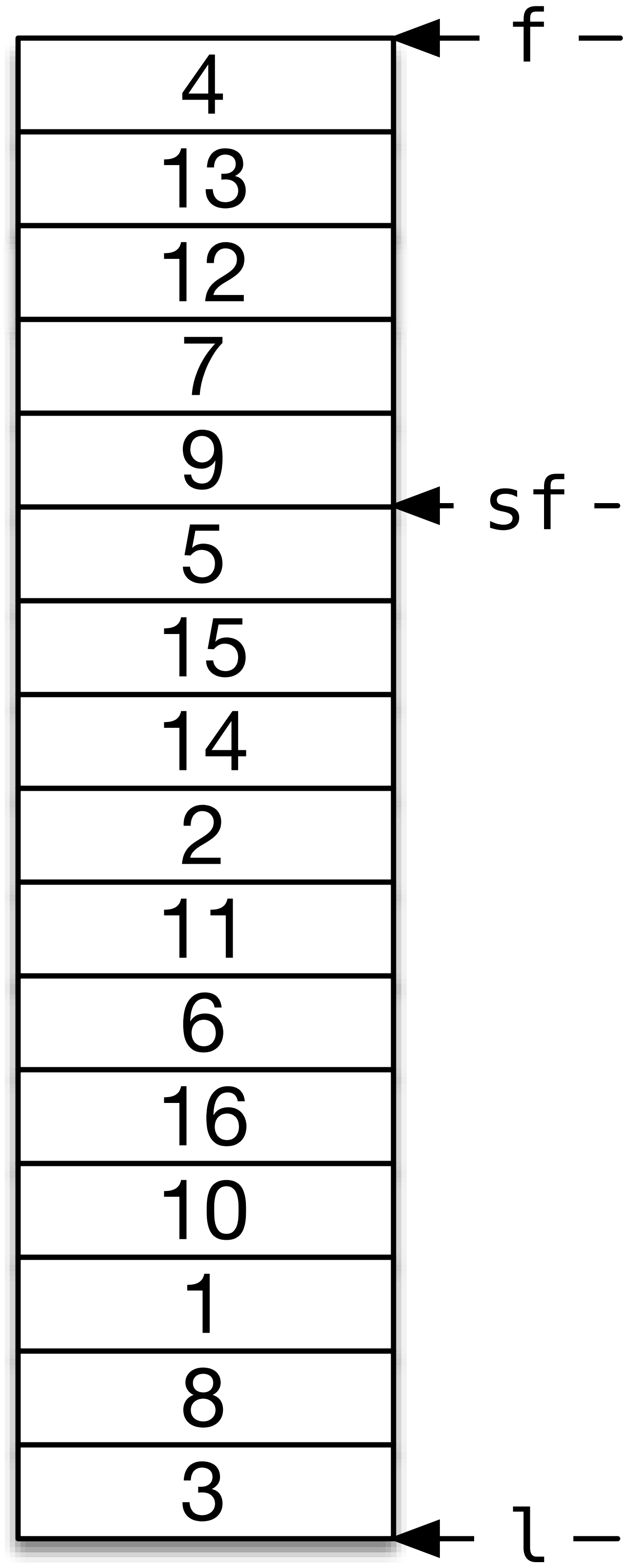


Observing Collections



4 ← f –
13
12
7
9
5 ← sf -
15
14
2
11
6
16
10
1
8
3
← l –

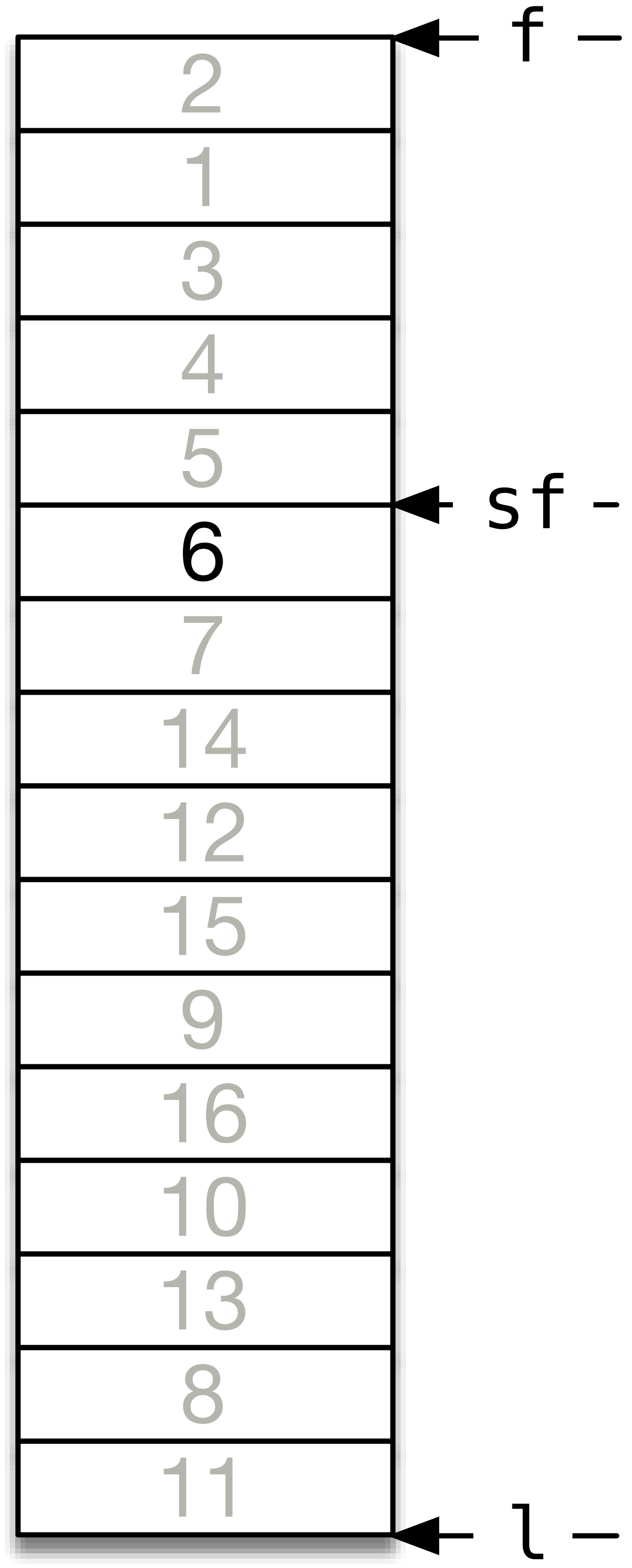
Observing Collections



4
13
12
7
9
5
15
14
2
11
6
16
10
1
8
3

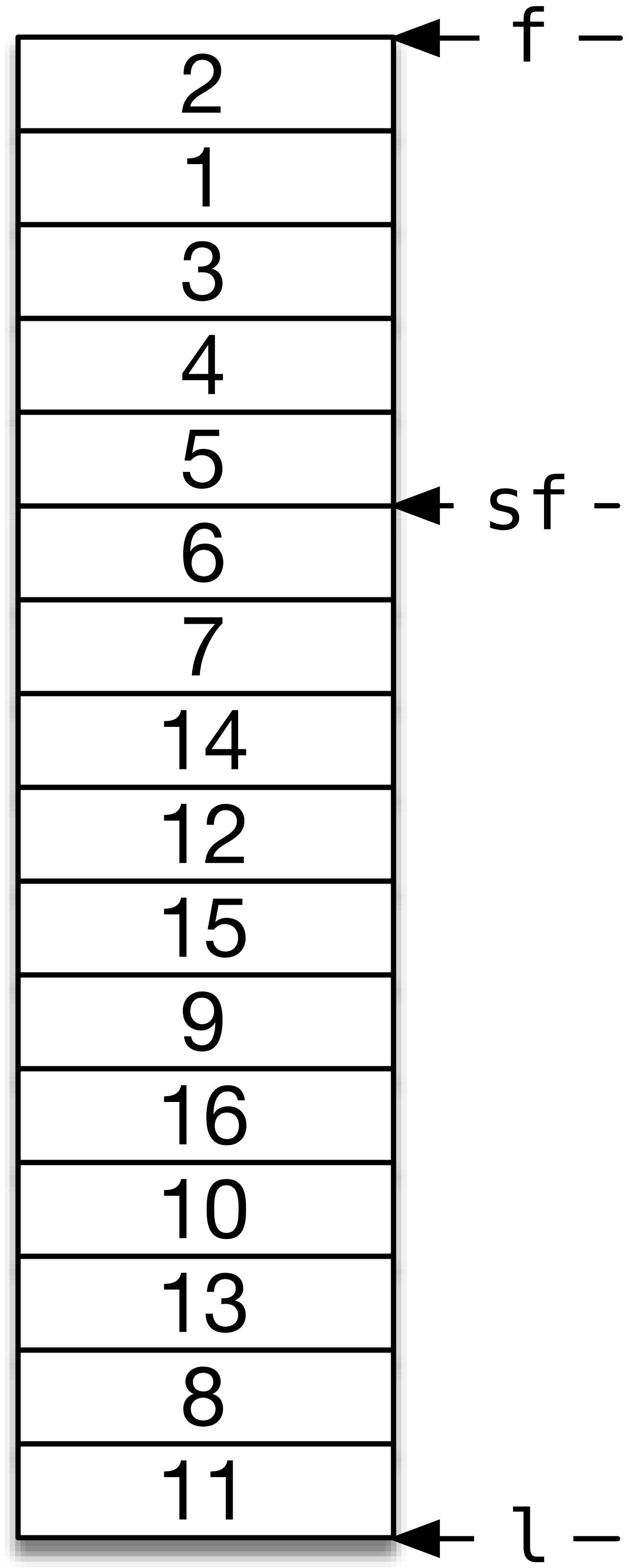
```
nth_element(f, sf, l);
```

Observing Collections



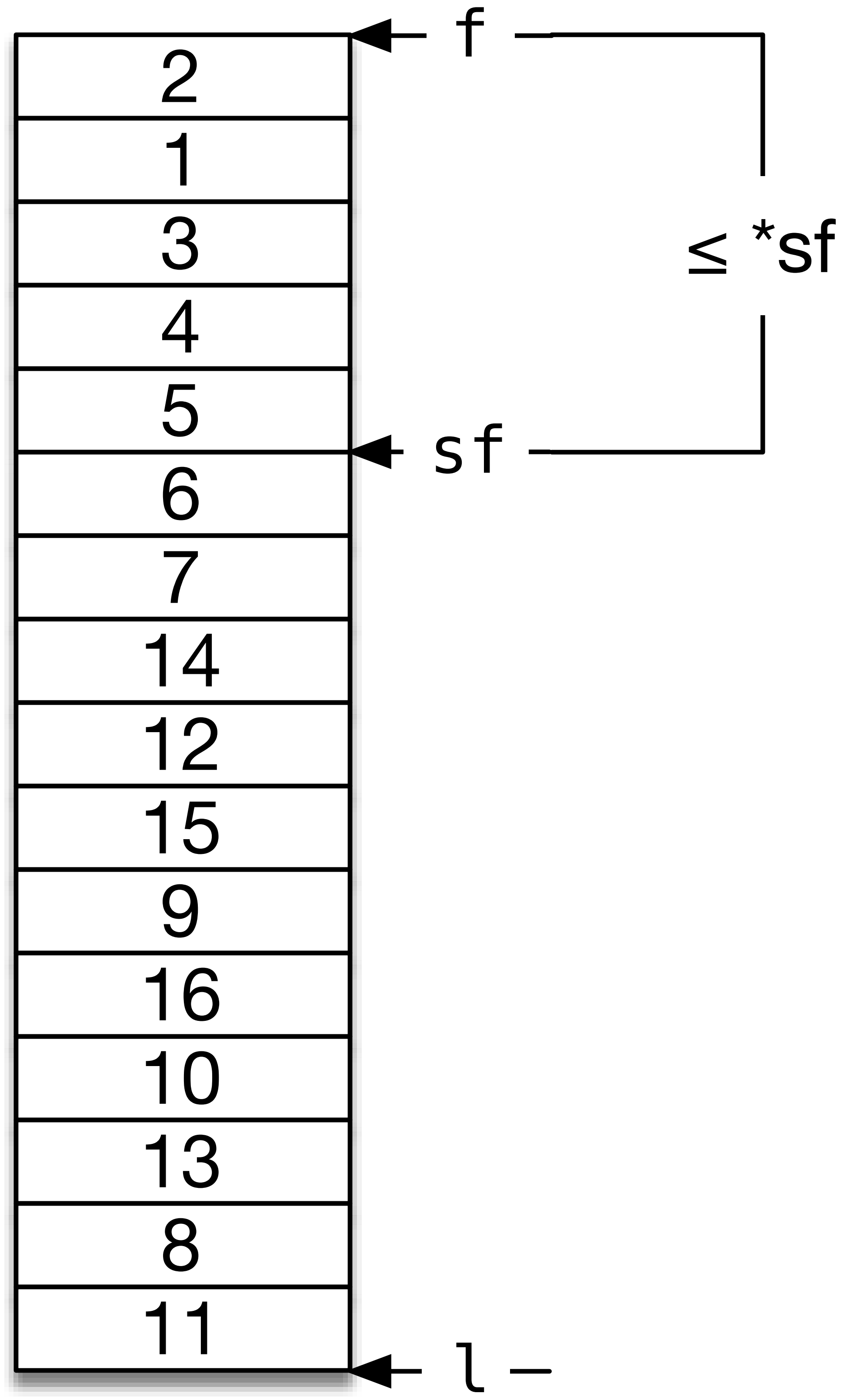
```
nth_element(f, sf, l);
```

Observing Collections



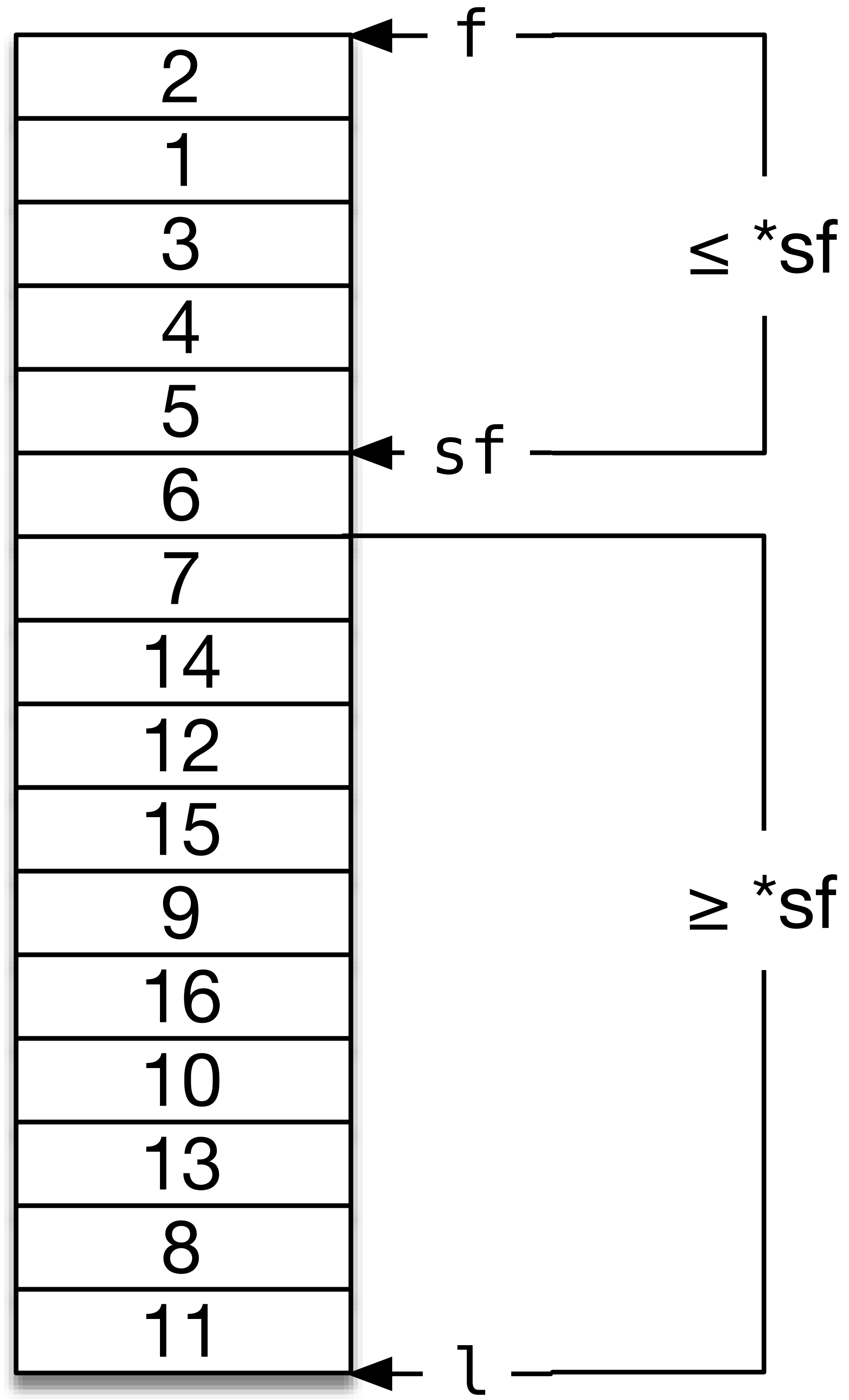
```
nth_element(f, sf, l);
```

Observing Collections



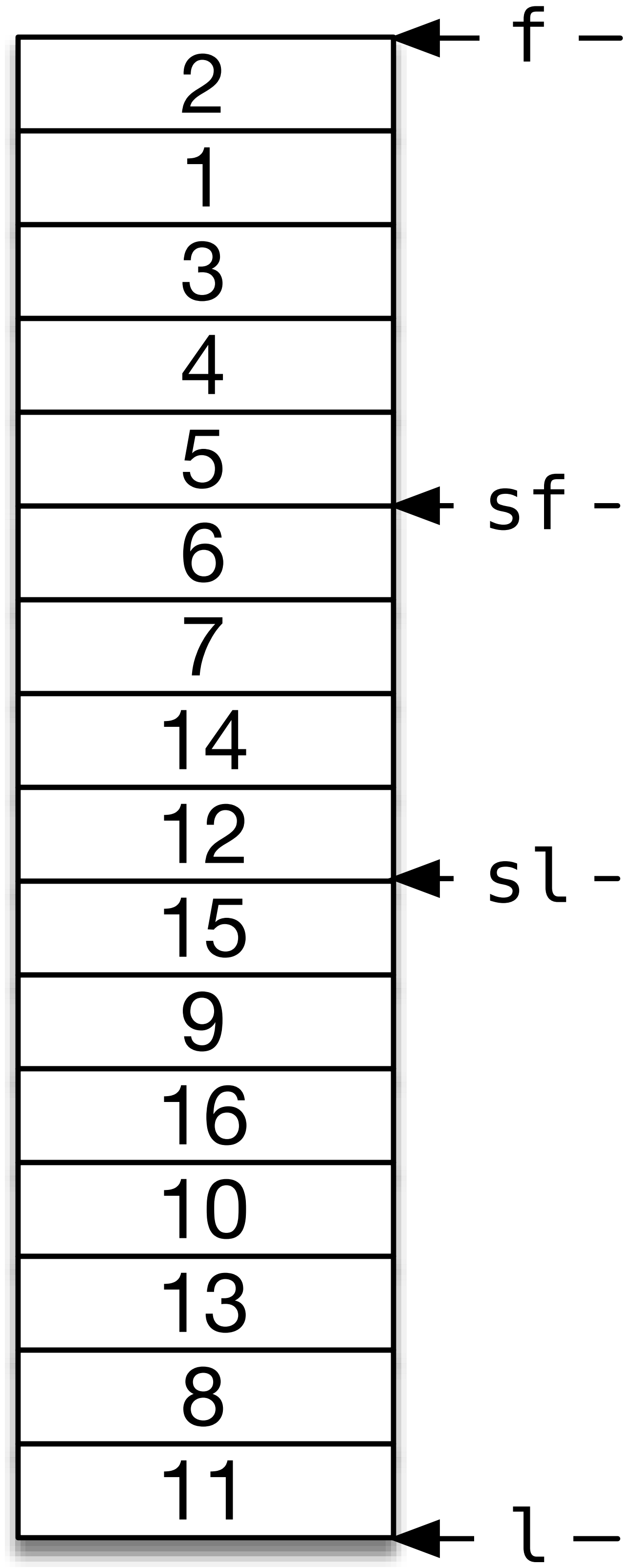
```
nth_element(f, sf, l);
```

Observing Collections



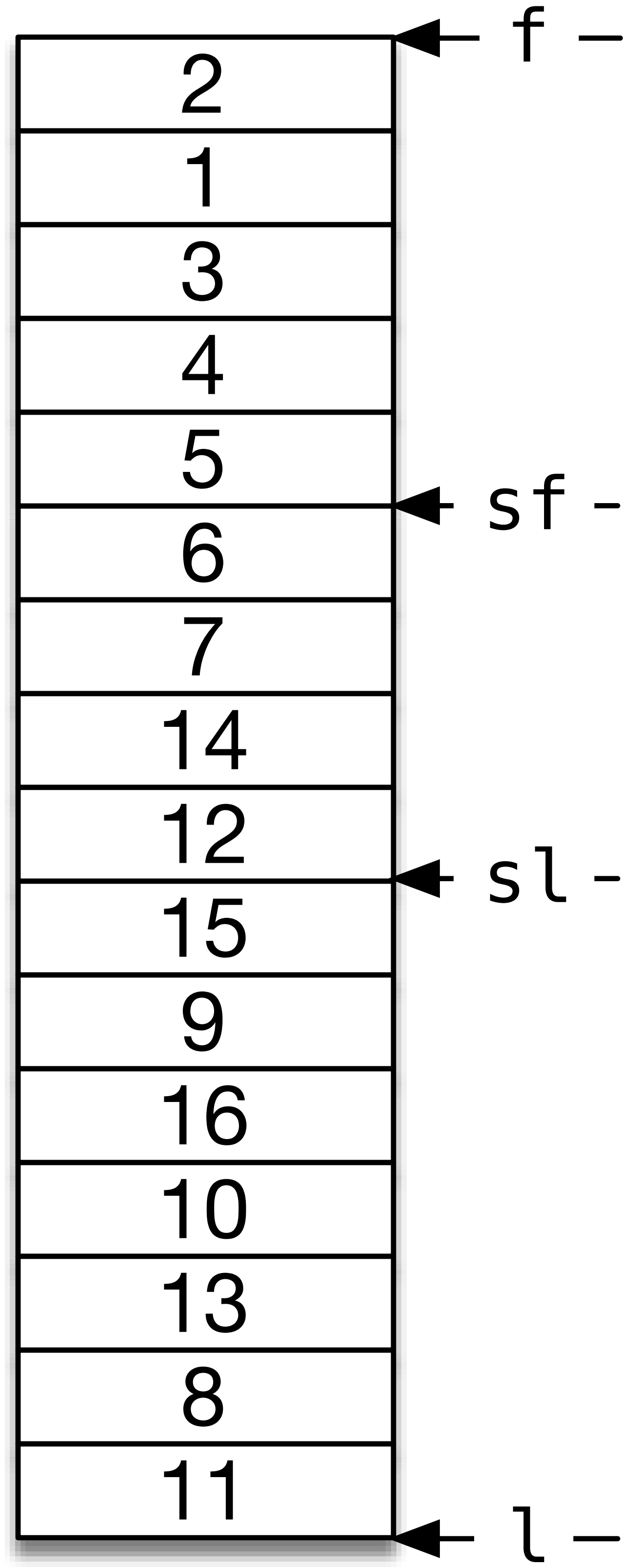
```
nth_element(f, sf, l);
```

Observing Collections



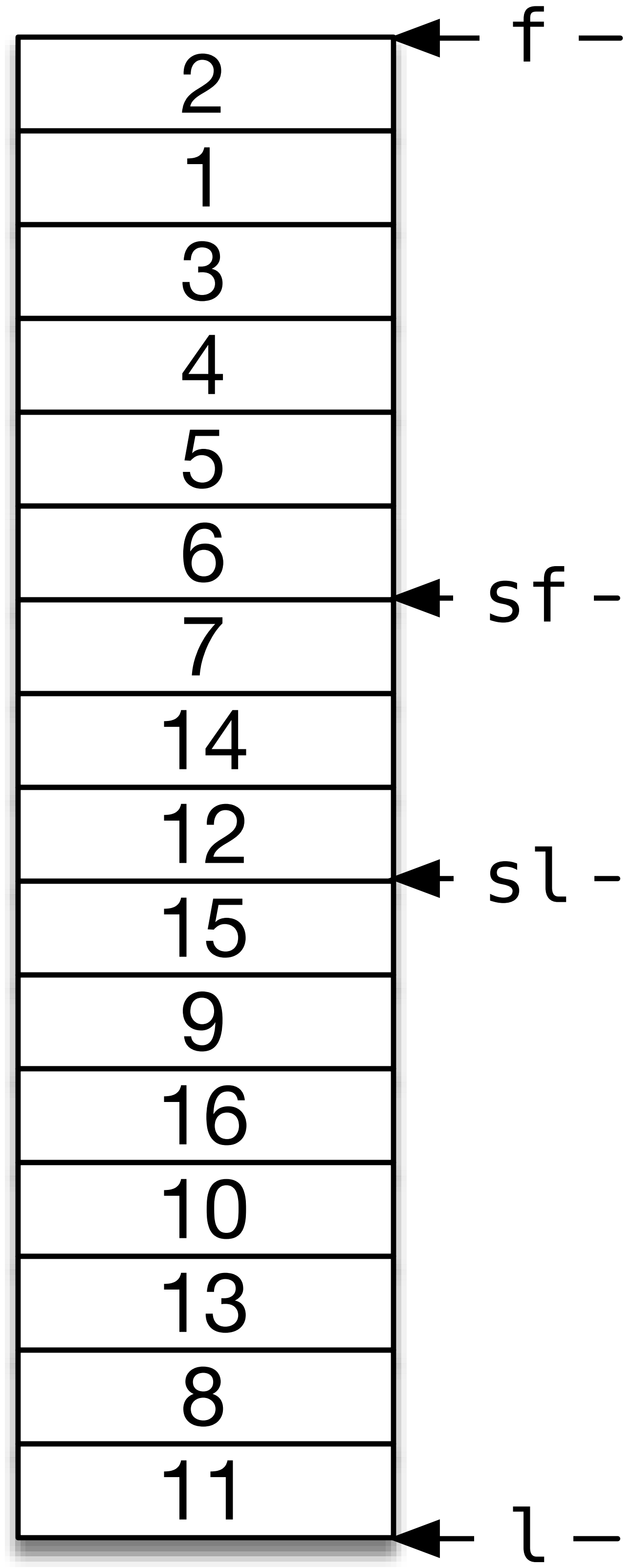
```
nth_element(f, sf, l);
```

Observing Collections



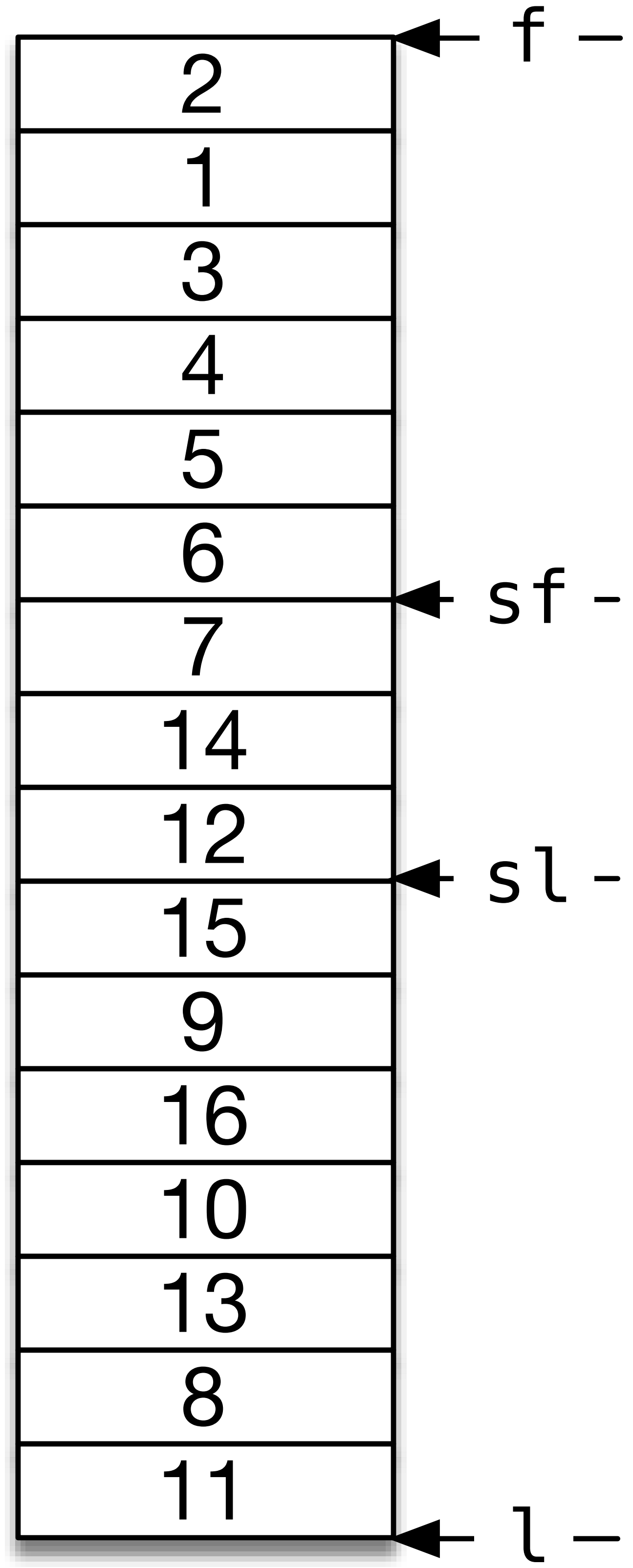
```
nth_element(f, sf, l);
++sf;
```

Observing Collections



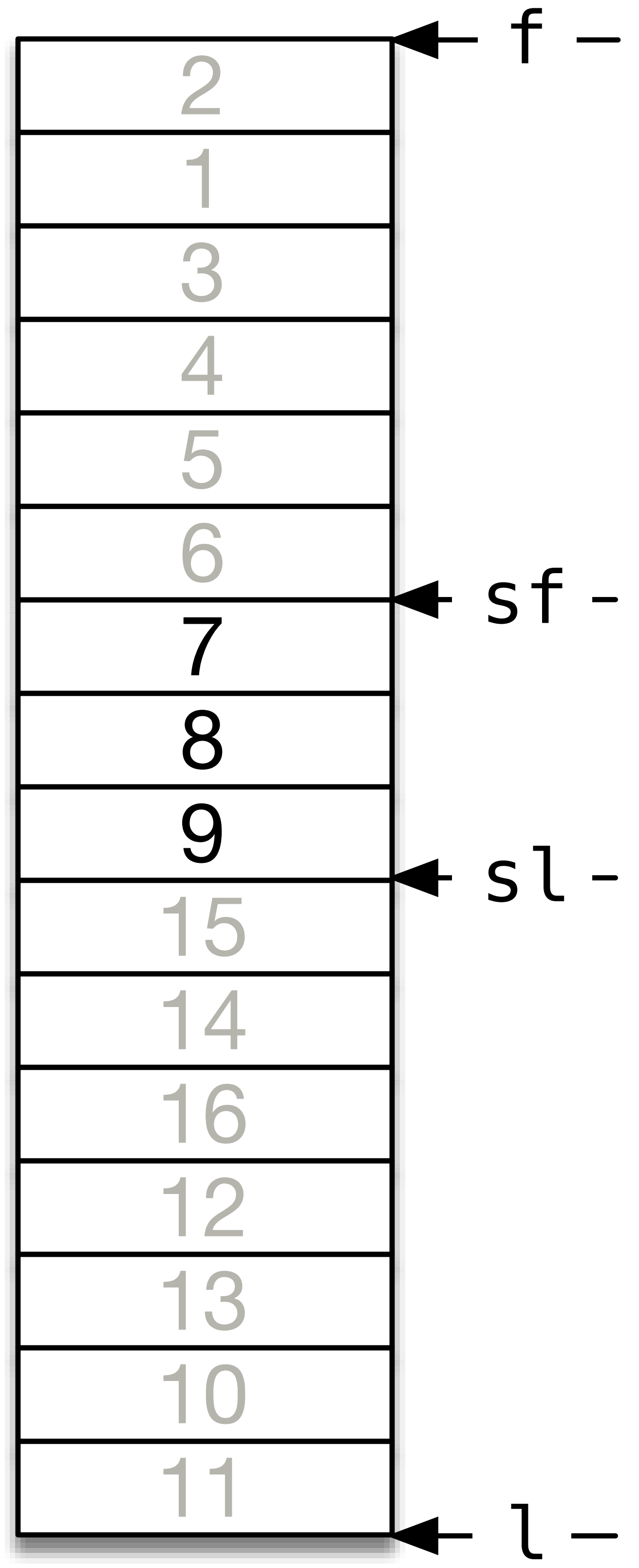
```
nth_element(f, sf, l);
++sf;
```

Observing Collections



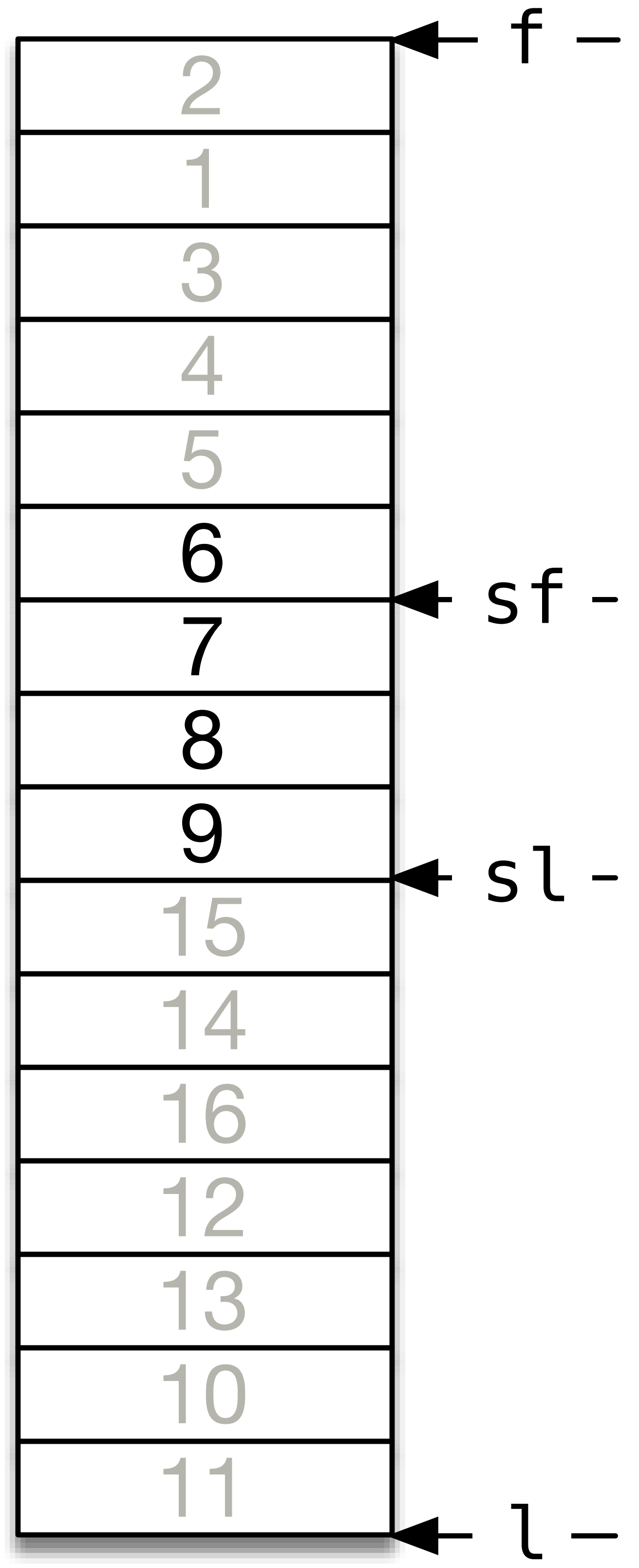
```
    nth_element(f, sf, l);
    ++sf;

partial_sort(sf, sl, l);
```

Observing Collections



```
    nth_element(f, sf, l);
    ++sf;

partial_sort(sf, sl, l);
```

Observing Collections



```
    nth_element(f, sf, l);
    ++sf;

partial_sort(sf, sl, l);
```

Observing Collections

```
if (sf == sl) return;

    nth_element(f, sf, l);
    ++sf;

partial_sort(sf, sl, l);
```

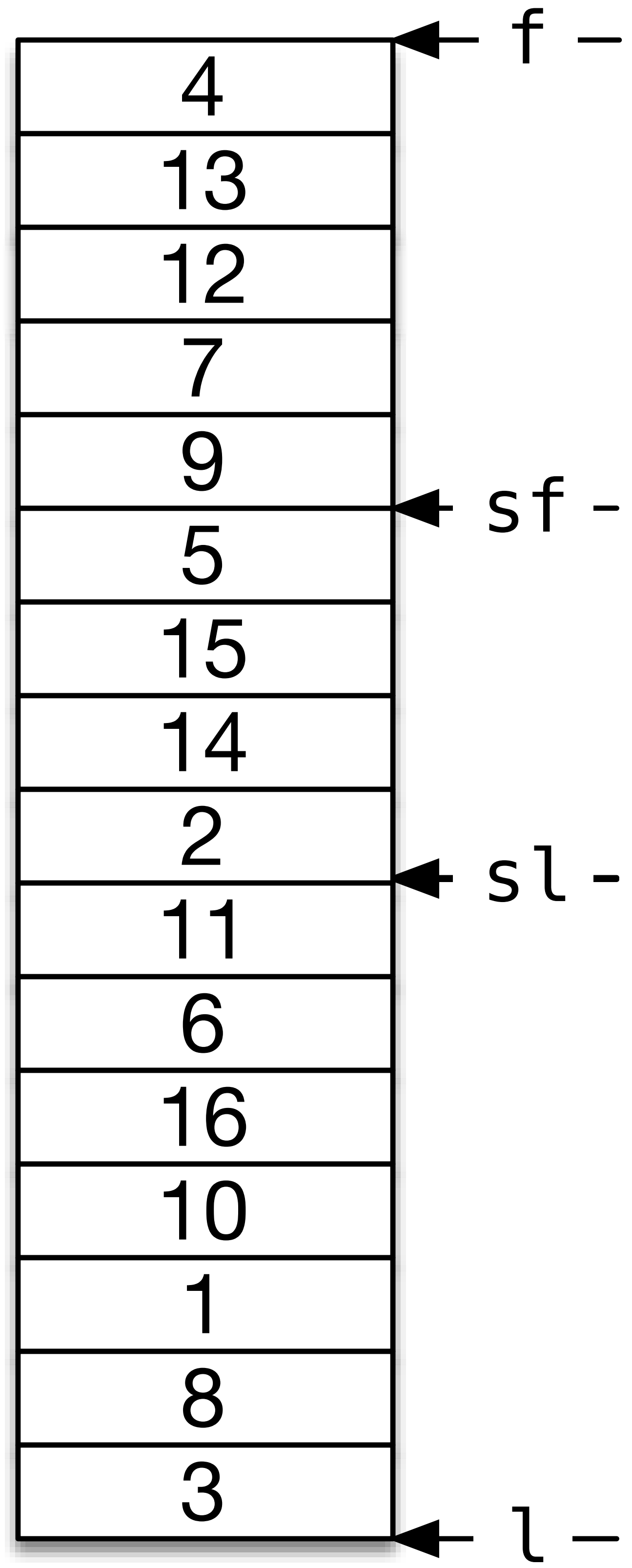
Observing Collections

```
if (sf == sl) return;
if (sf != f) {
    nth_element(f, sf, l);
    ++sf;
}
partial_sort(sf, sl, l);
```

Observing Collections

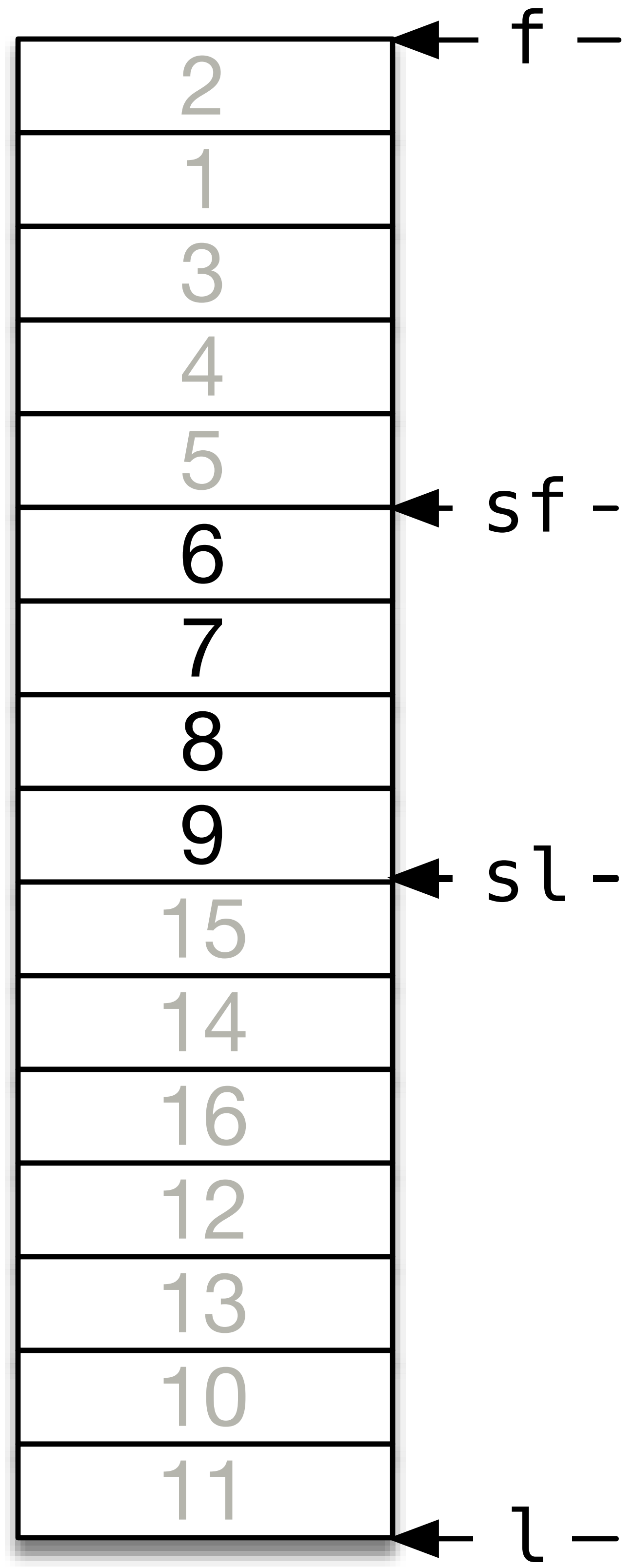
```
template <typename I> // I models RandomAccessIterator
void sort_subrange(I f, I l, I sf, I sl)
{
    if (sf == sl) return;
    if (sf != f) {
        nth_element(f, sf, l);
        ++sf;
    }
    partial_sort(sf, sl, l);
}
```

Observing Collections



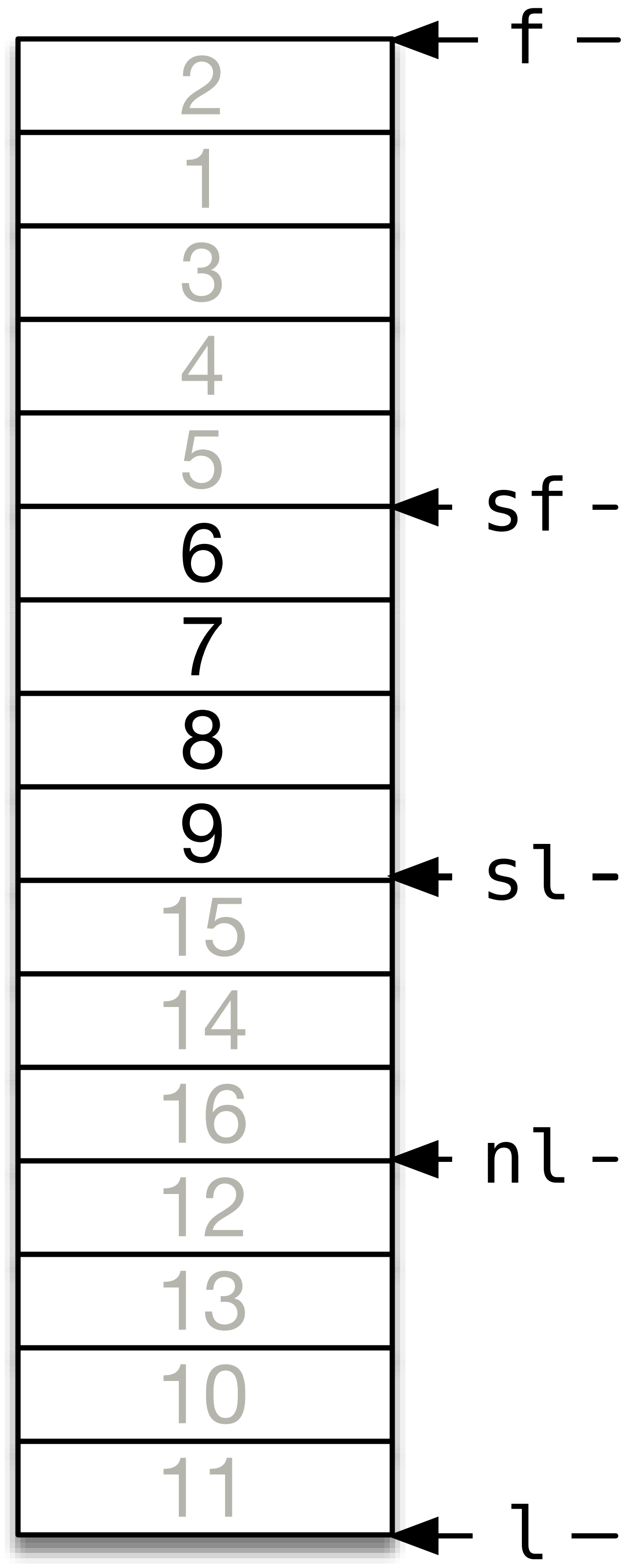
```
sort_subrange(f, l, sf, sl);
```

Observing Collections



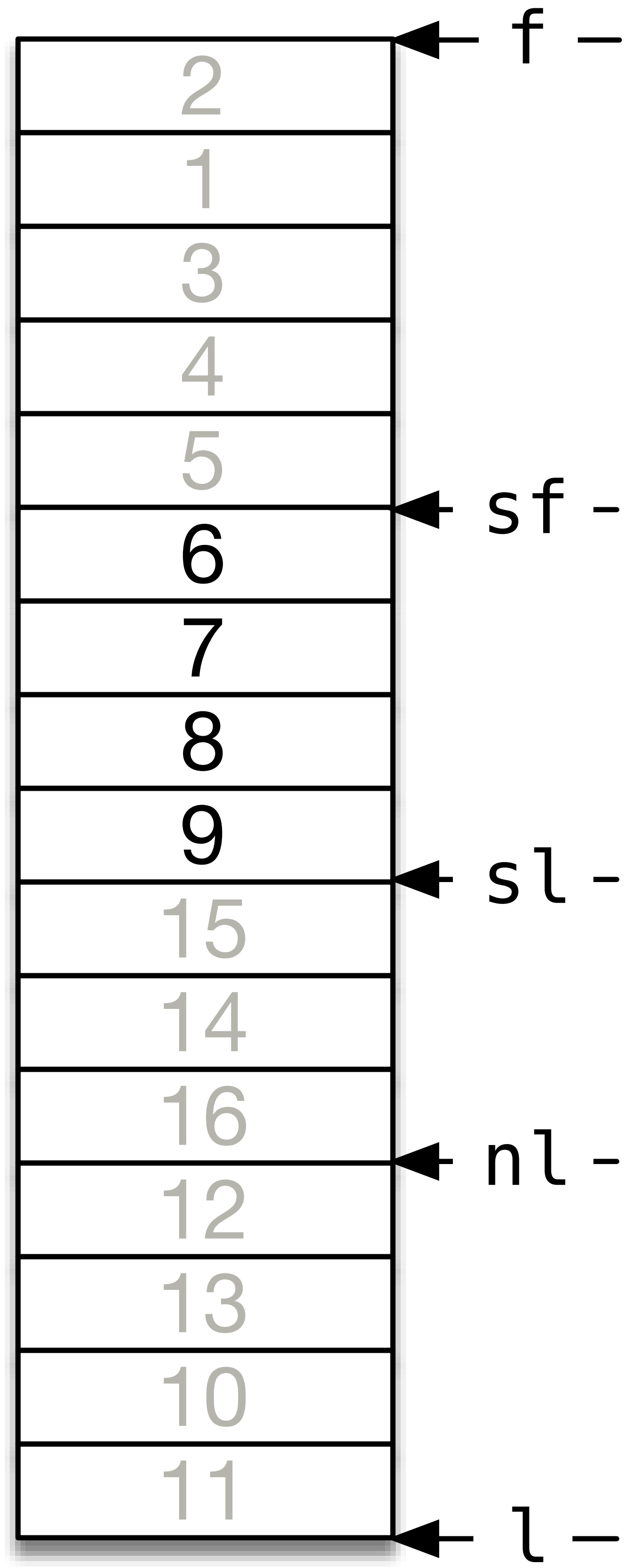
```
sort_subrange(f, l, sf, sl);
```

Observing Collections



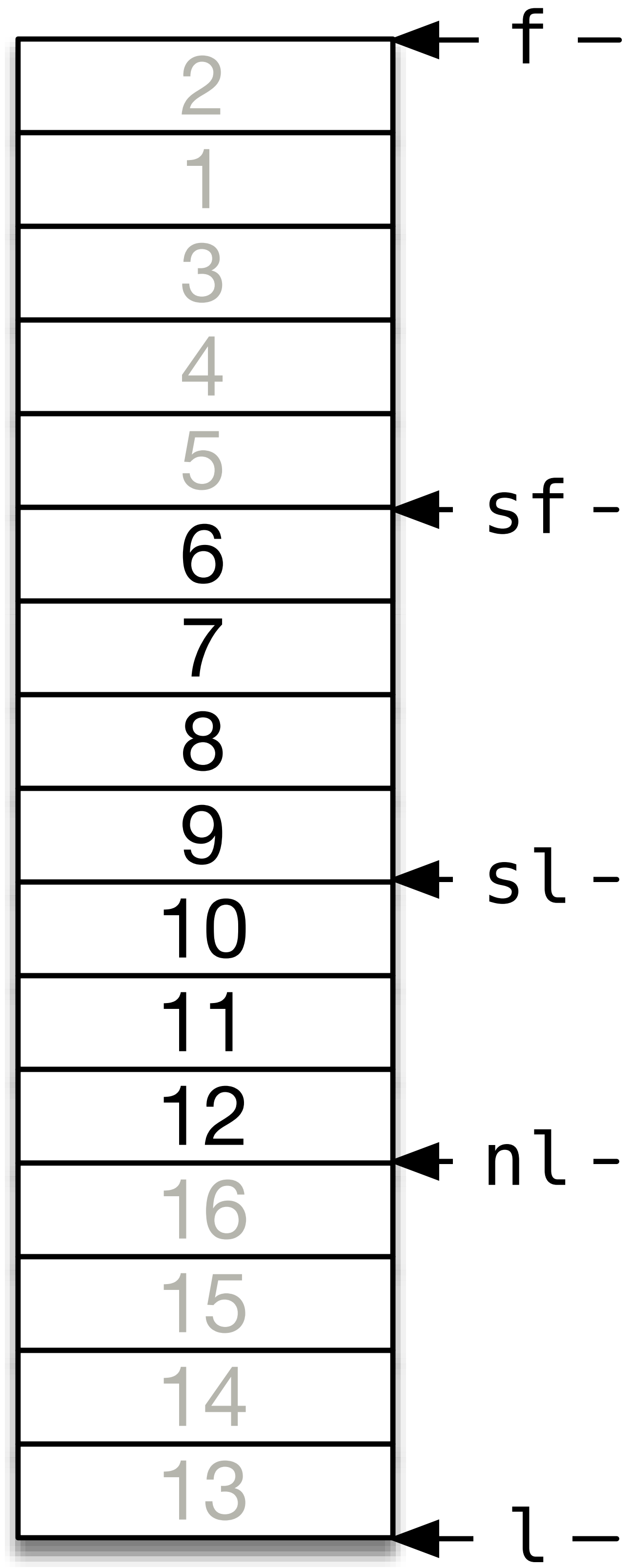
```
sort_subrange(f, l, sf, sl);
```

Observing Collections



```
sort_subrange(f, l, sf, sl);
partial_sort(sl, nl, l);
```

Observing Collections



```
sort_subrange(f, l, sf, sl);
partial_sort(sl, nl, l);
```

Operations

- Operations act on one or more objects
  - Additional arguments to the operation are bound as properties
  - Operations are represented by buttons, menu items, gestures, tools, direct manipulation
- Subject or *target* of an operation is identified by
  - Selections
  - Direct Manipulation

Selections

- Selecting objects within the hierarchy specifies one or more target paths
  - Application->Document->Object

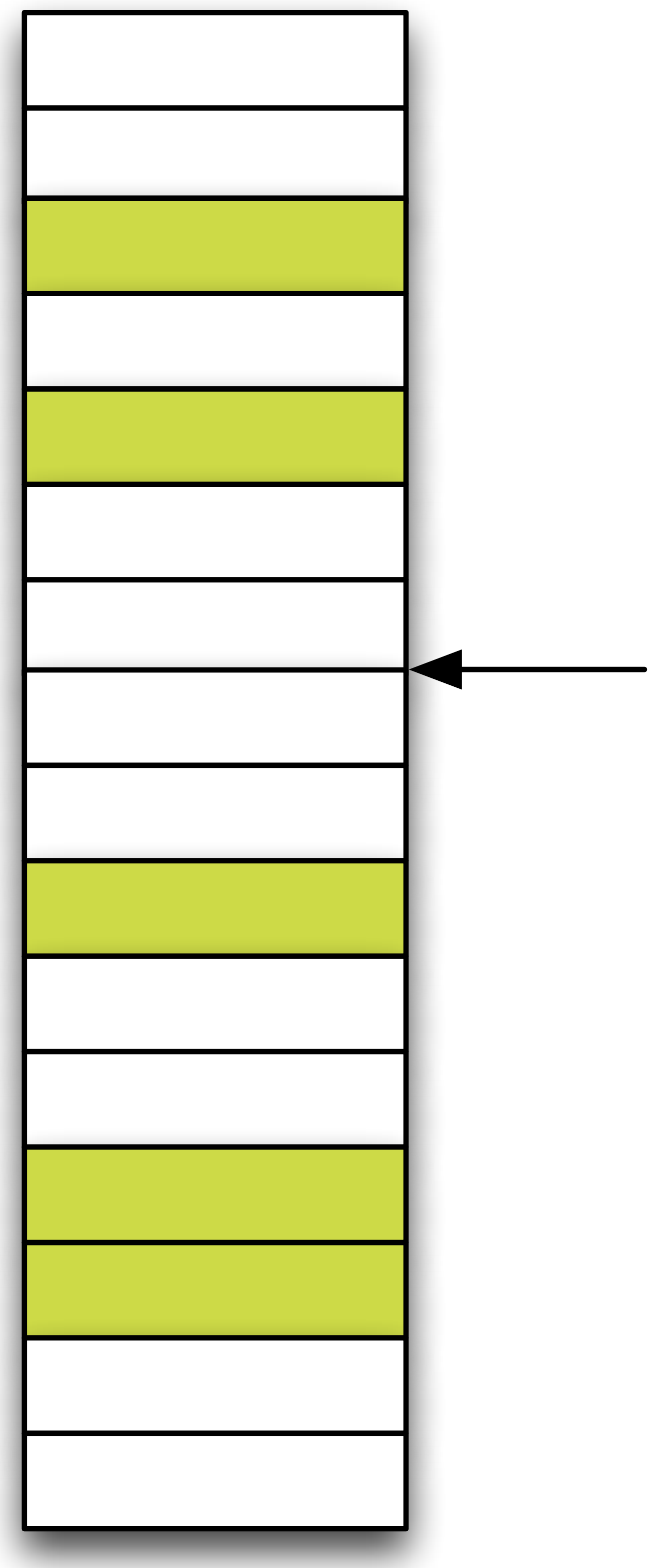
Selections

- Selecting objects within the hierarchy specifies one or more target paths
  - Application->Document->Object

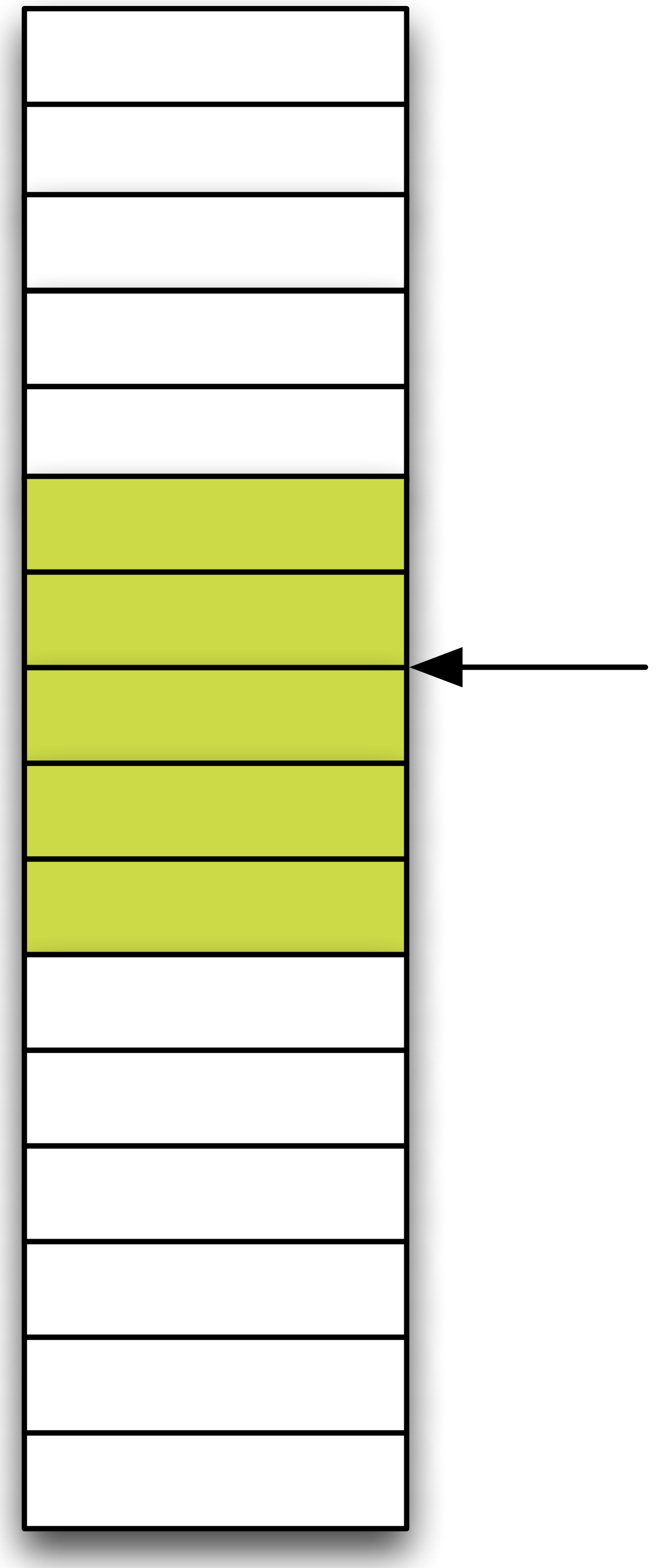
Selections

- Interval sets are a good data structure to represent selections

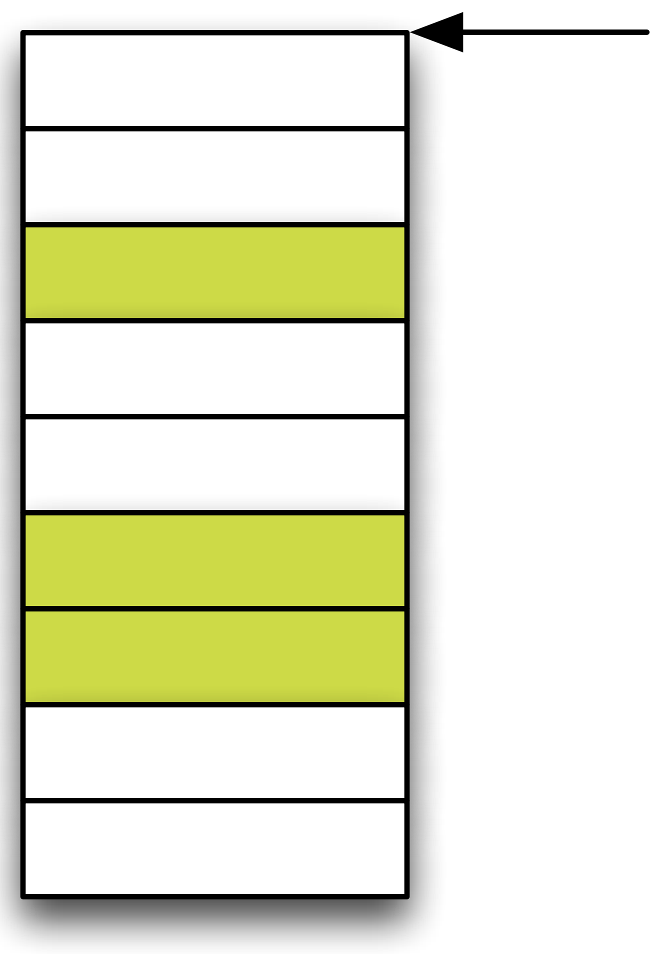
Gather



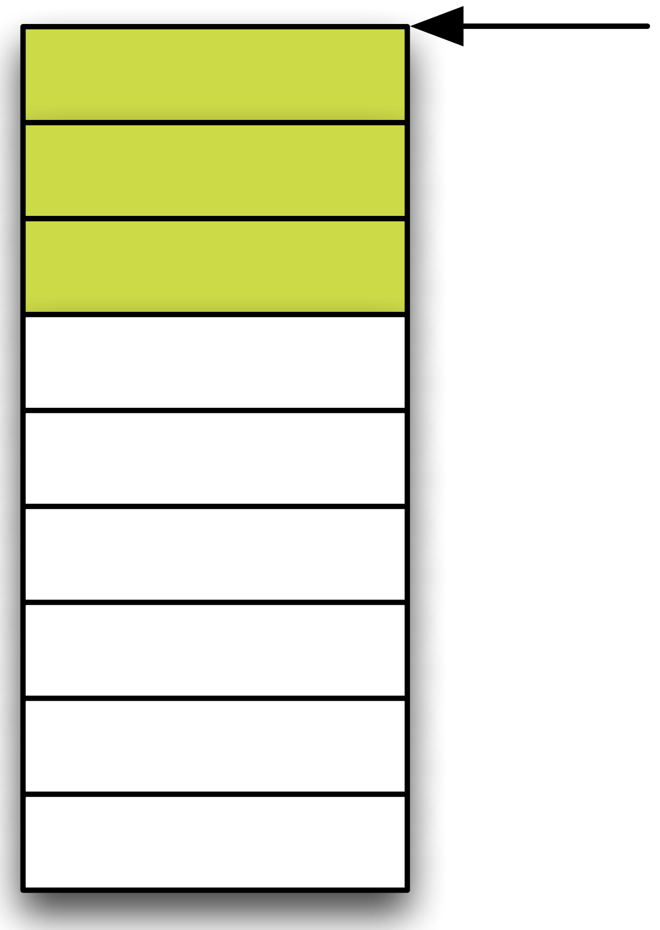
Gather



Gather

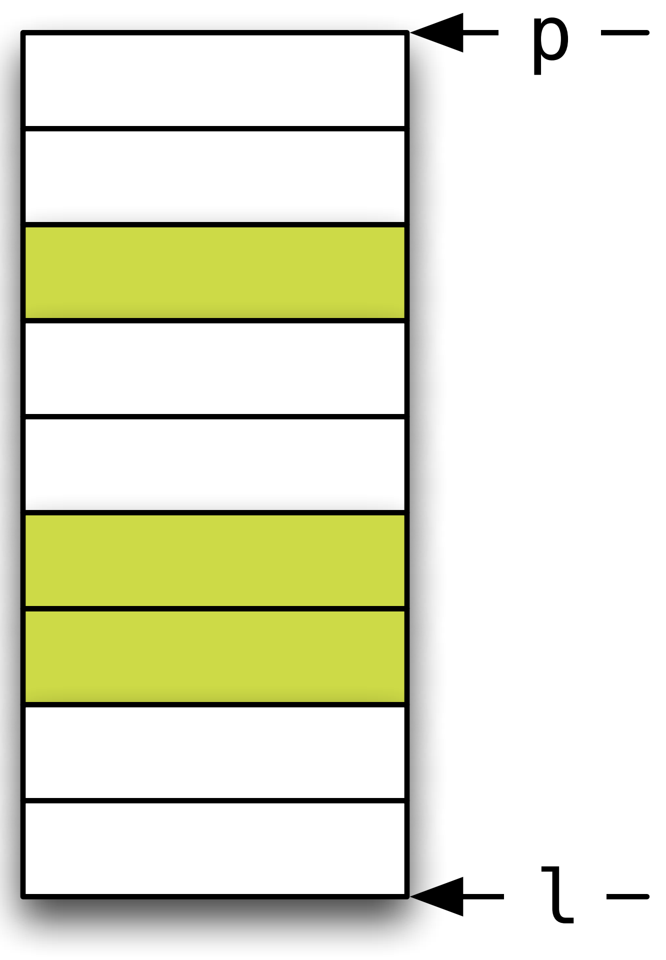


Gather



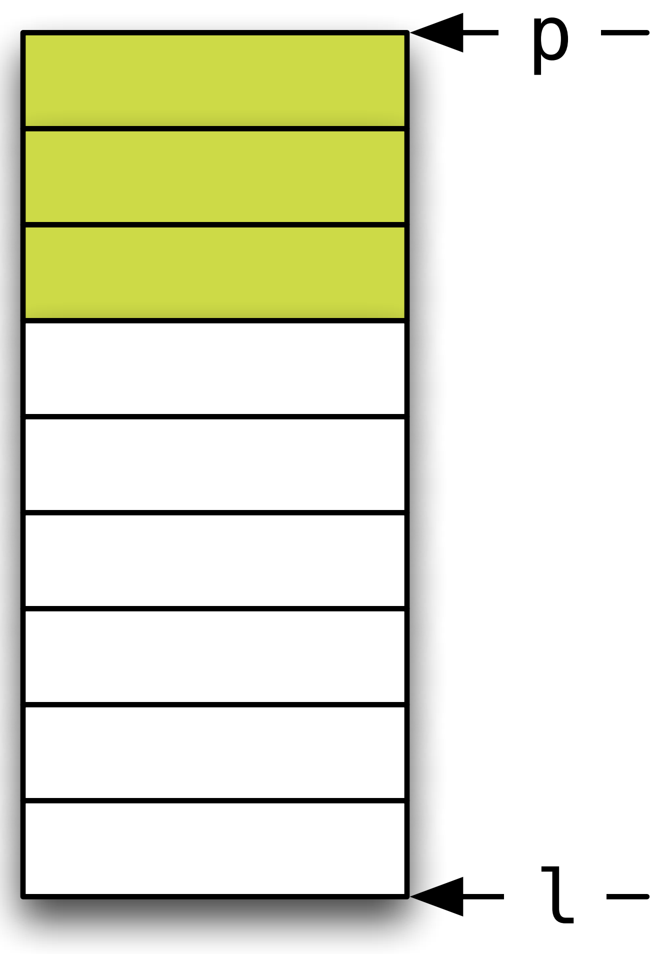
Gather

`stable_partition(p, l, s)`

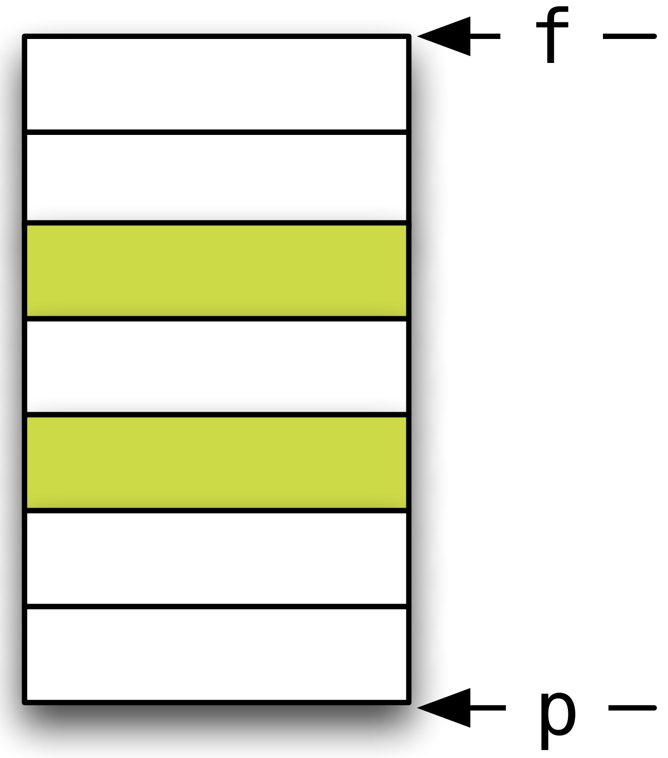


Gather

`stable_partition(p, l, s)`

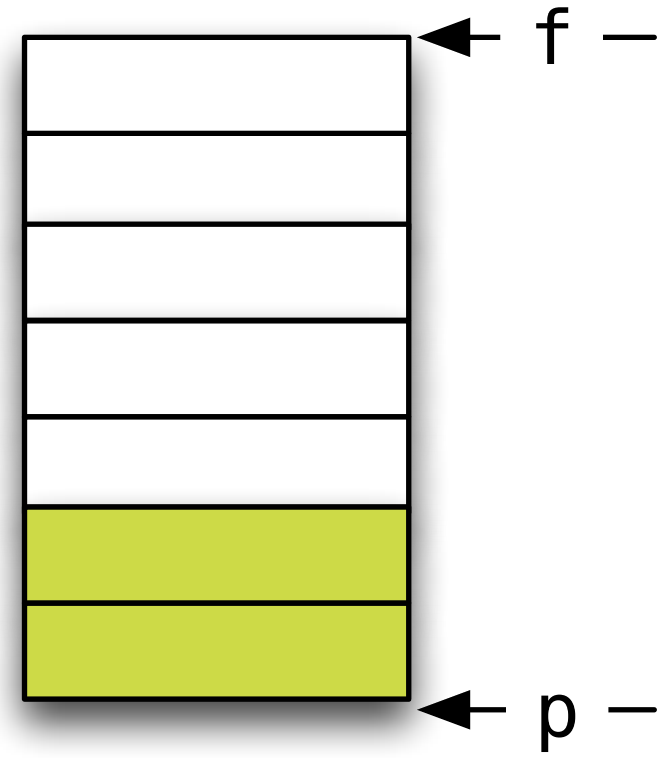


Gather



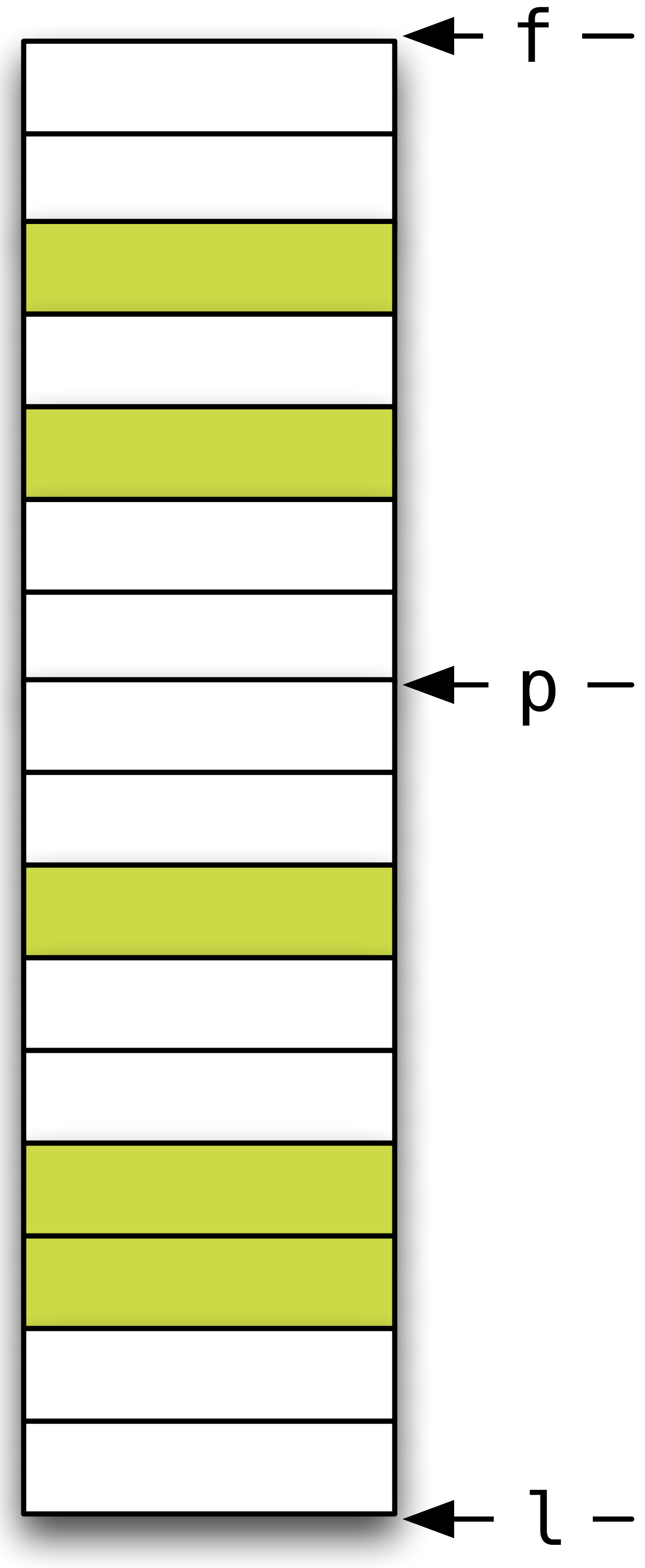
```
stable_partition(f, p, not1(s))
```

Gather



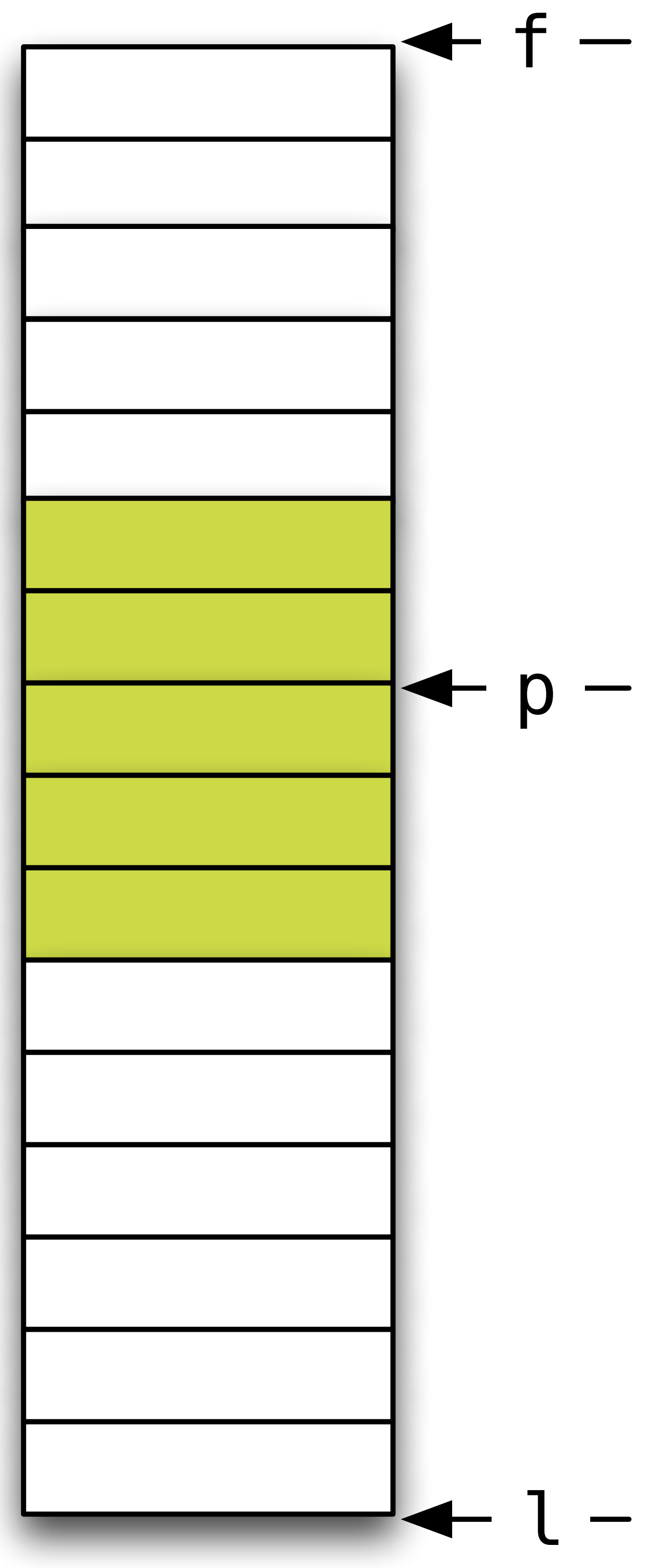
```
stable_partition(f, p, not1(s))
```

Gather



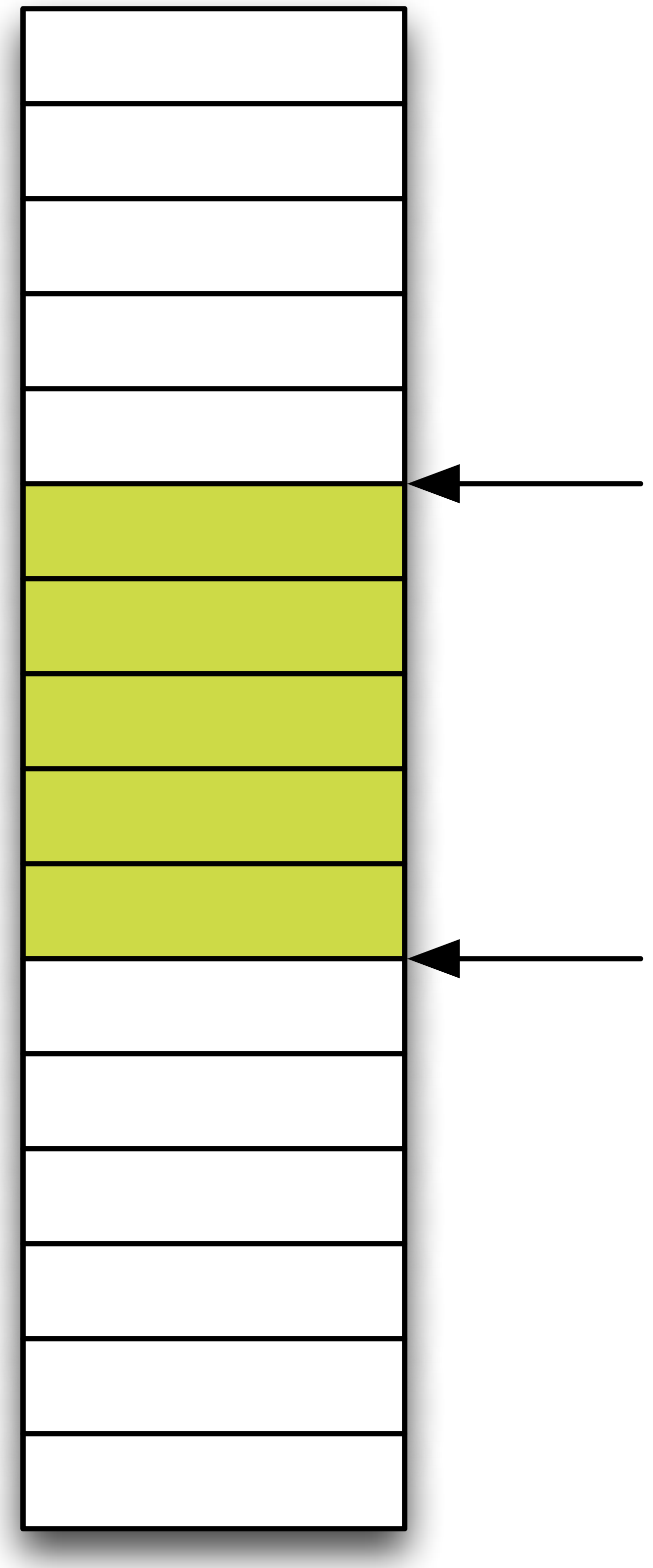
```
stable_partition(f, p, not1(s))
stable_partition(p, l, s)
```

Gather



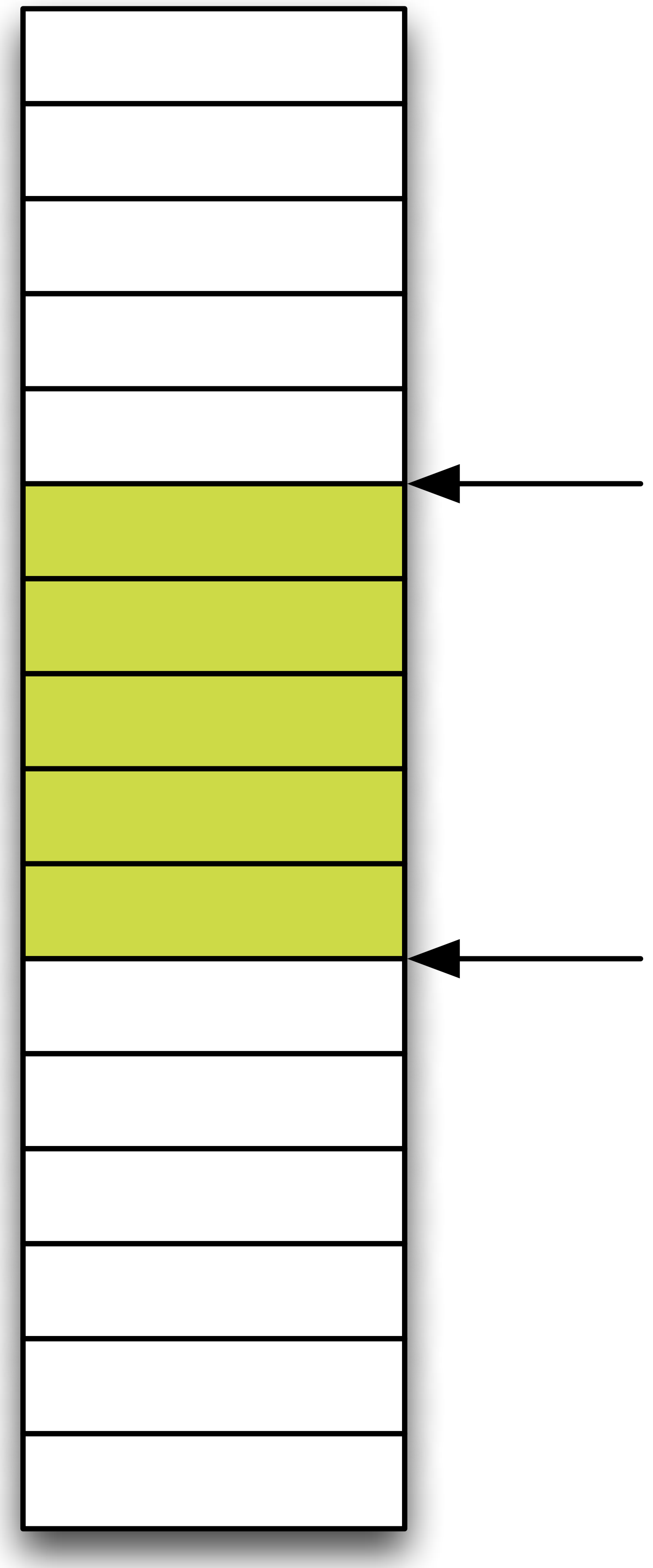
```
stable_partition(f, p, not1(s))
stable_partition(p, l, s)
```

Gather



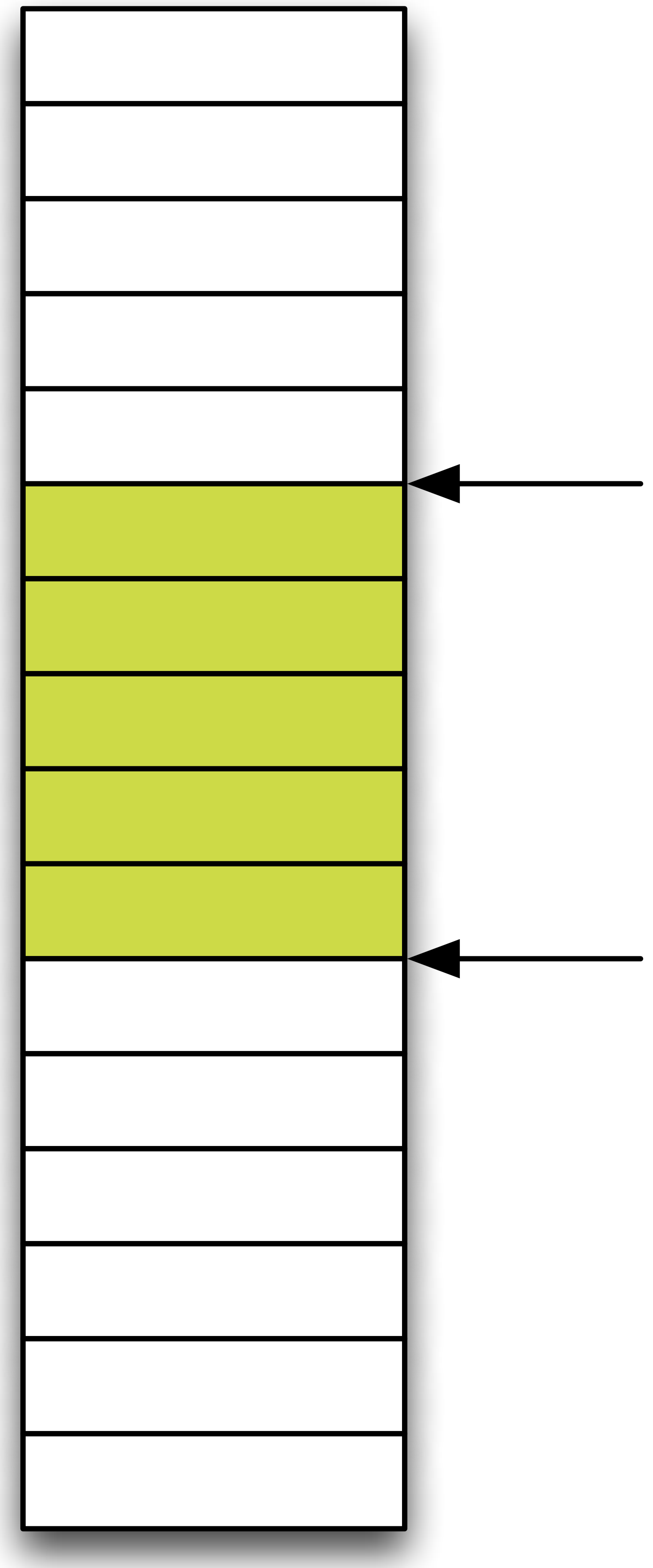
```
stable_partition(f, p, not1(s))
stable_partition(p, l, s)
```

Gather



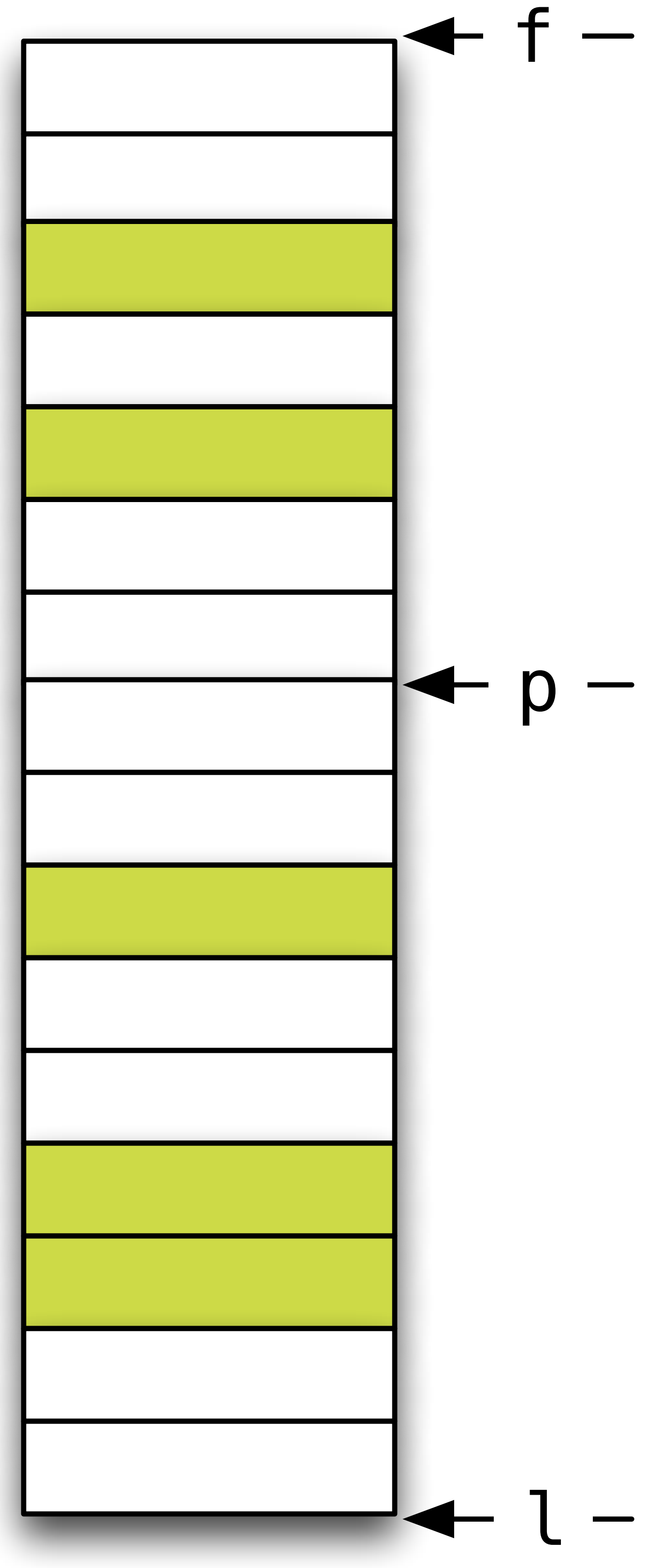
```
return { stable_partition(f, p, not1(s)),
         stable_partition(p, l, s) };
```

Gather



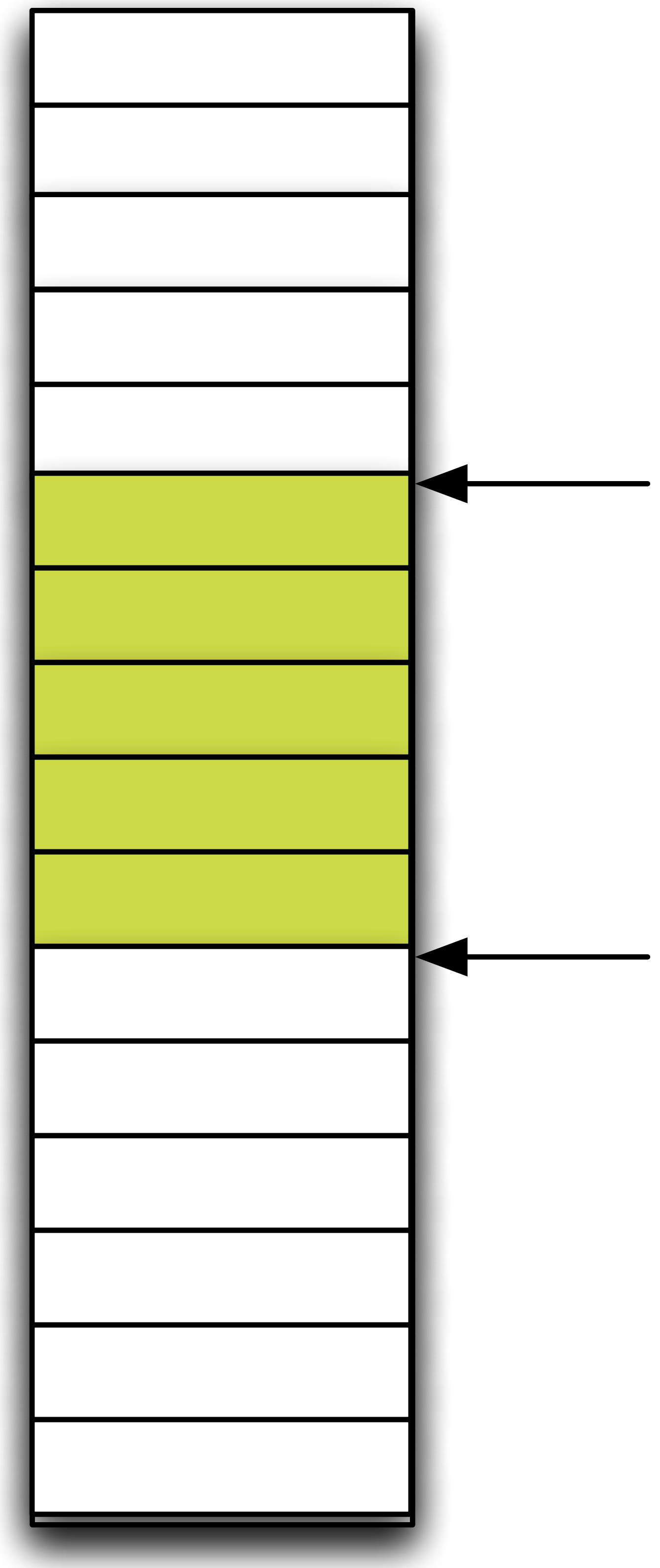
```
template <typename I, // I models BidirectionalIterator
          typename S> // S models UnaryPredicate
auto gather(I f, I l, I p, S s) -> pair<I, I>
{
    return { stable_partition(f, p, not1(s)),
             stable_partition(p, l, s) };
}
```

Gather



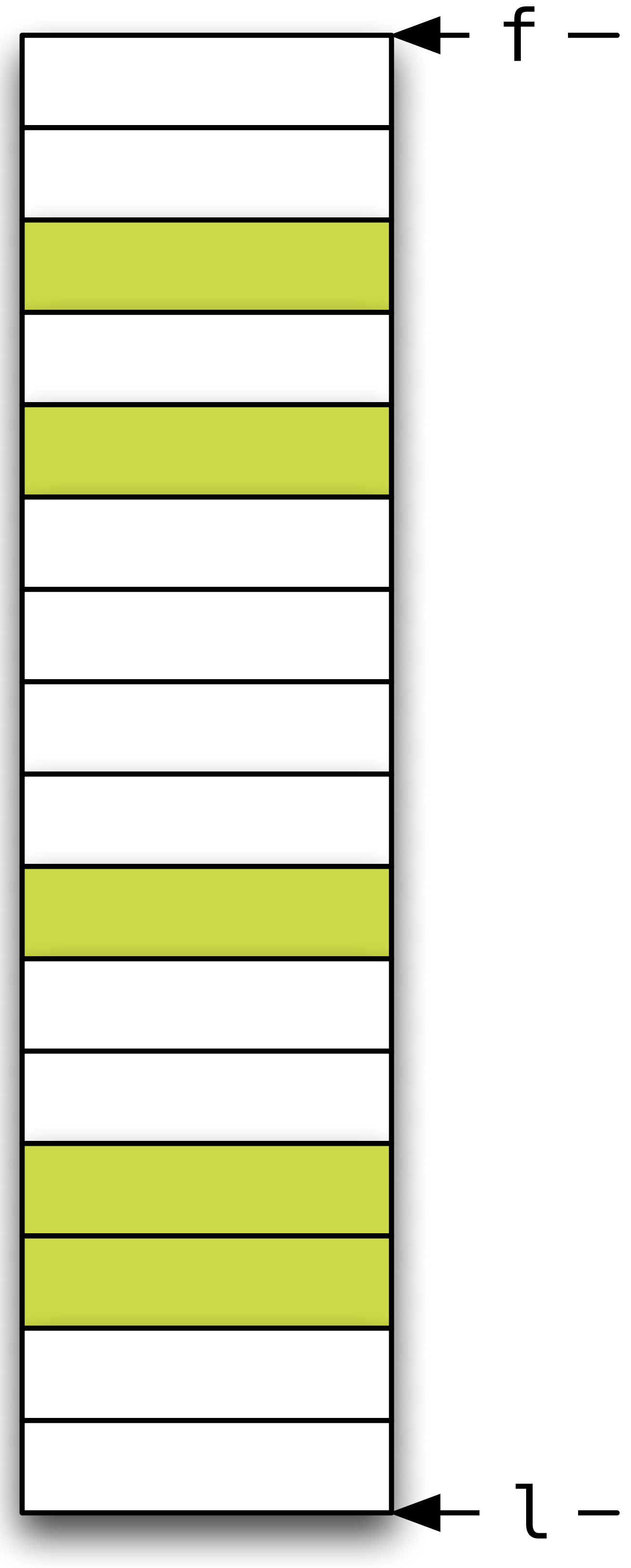
```
template <typename I, // I models BidirectionalIterator
          typename S> // S models UnaryPredicate
auto gather(I f, I l, I p, S s) -> pair<I, I>
{
    return { stable_partition(f, p, not1(s)),
             stable_partition(p, l, s) };
}
```

Gather

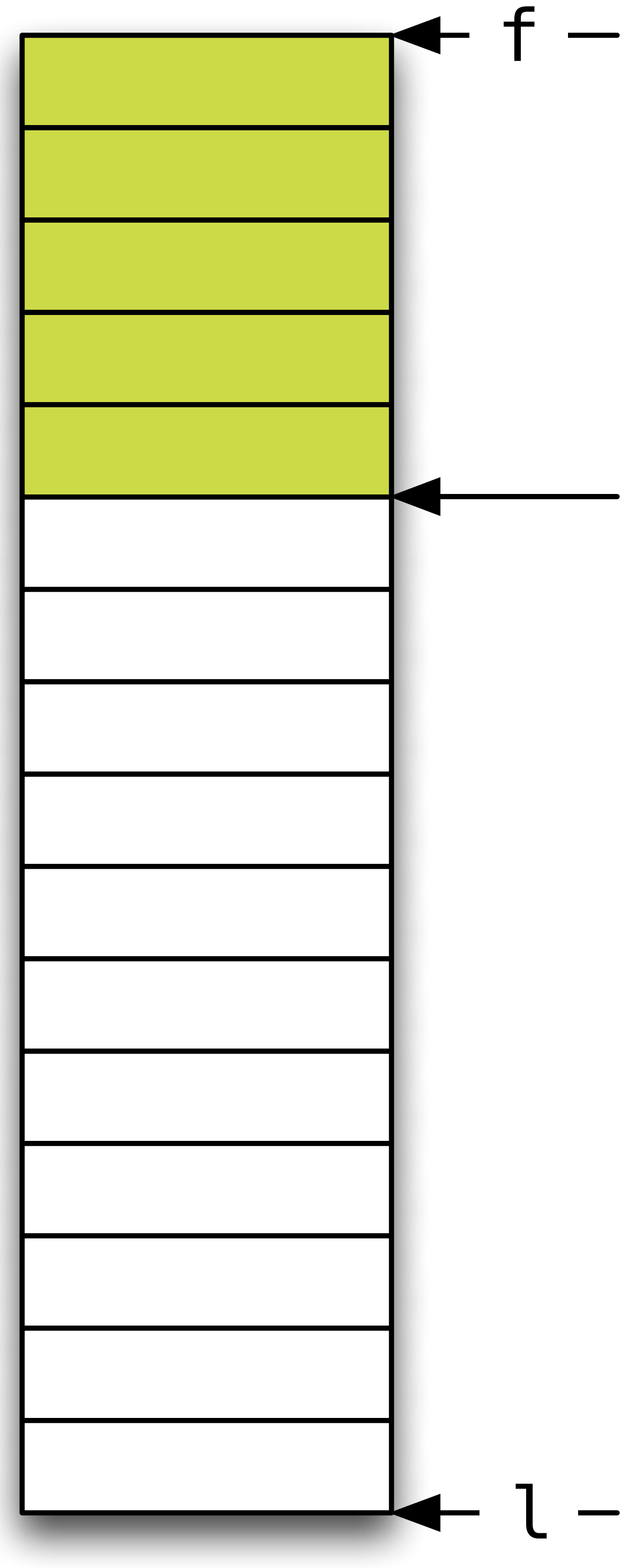


```
template <typename I, // I models BidirectionalIterator
          typename S> // S models UnaryPredicate
auto gather(I f, I l, I p, S s) -> pair<I, I>
{
    return { stable_partition(f, p, not1(s)),
             stable_partition(p, l, s) };
}
```

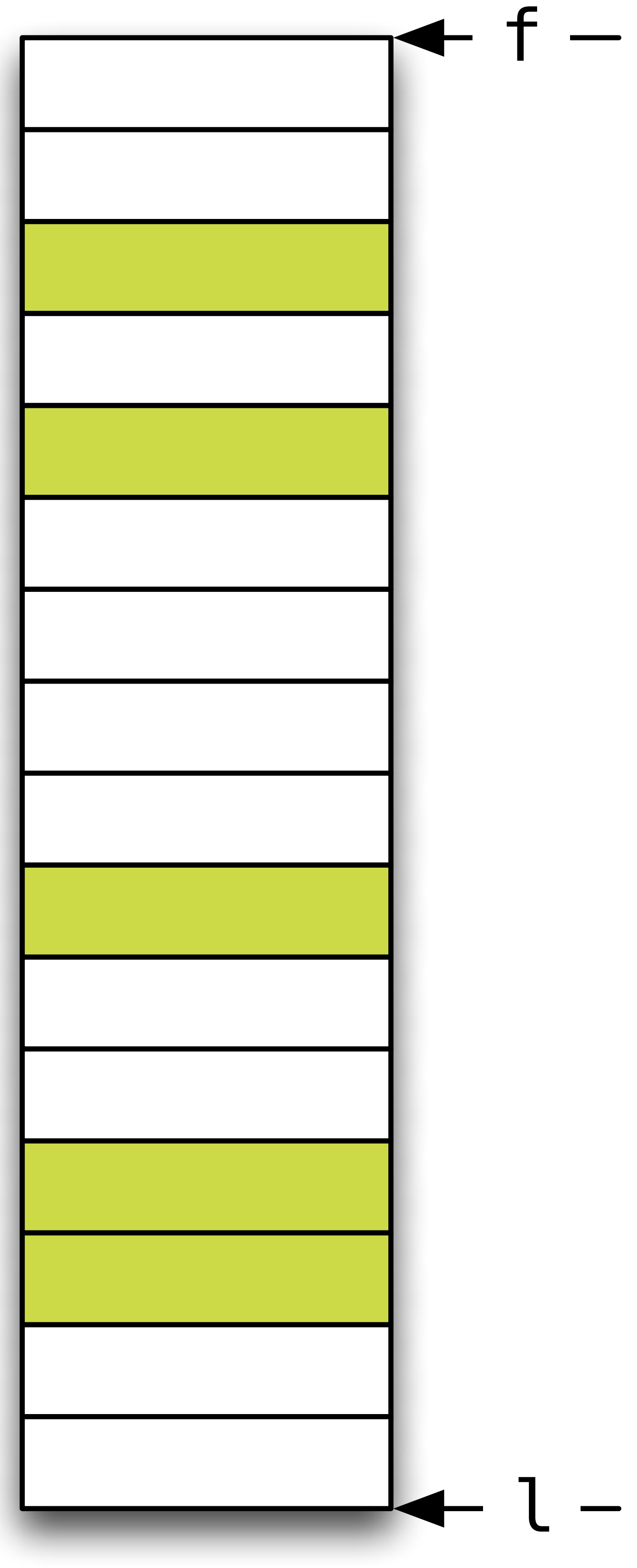
Stable Partition



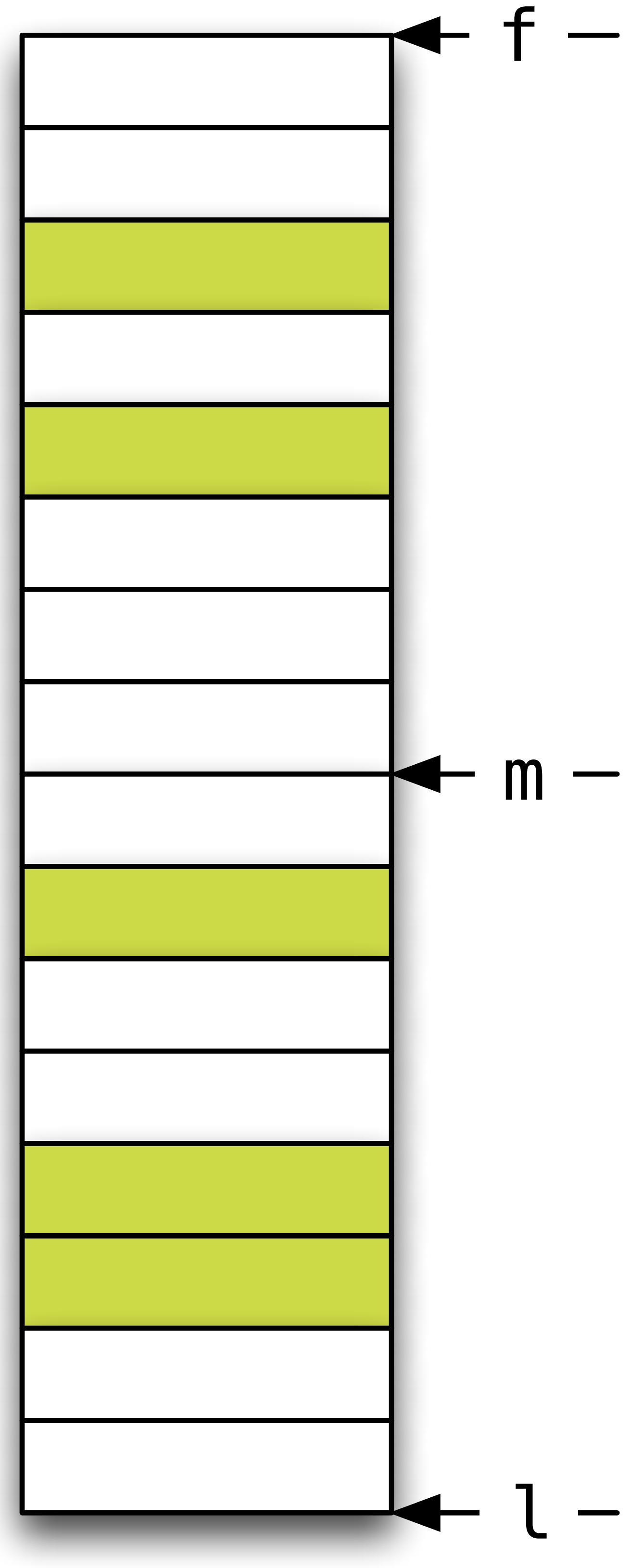
Stable Partition



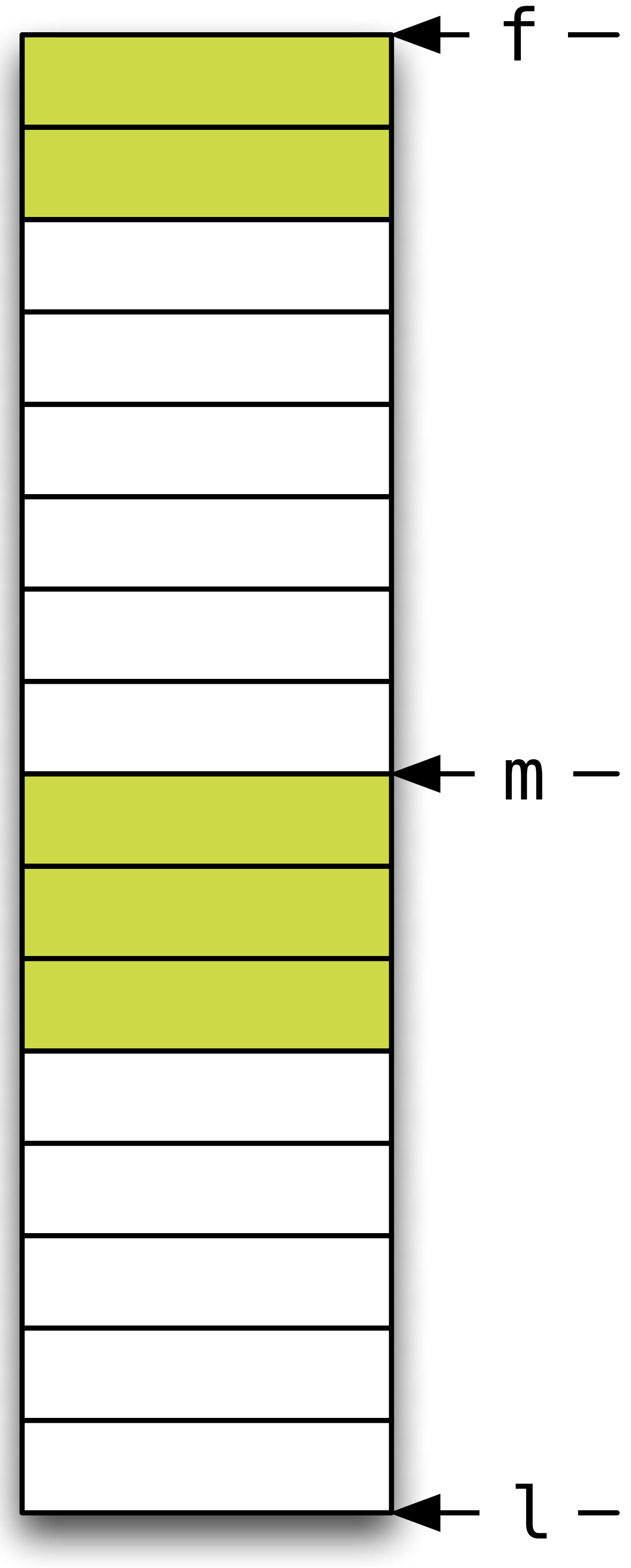
Stable Partition



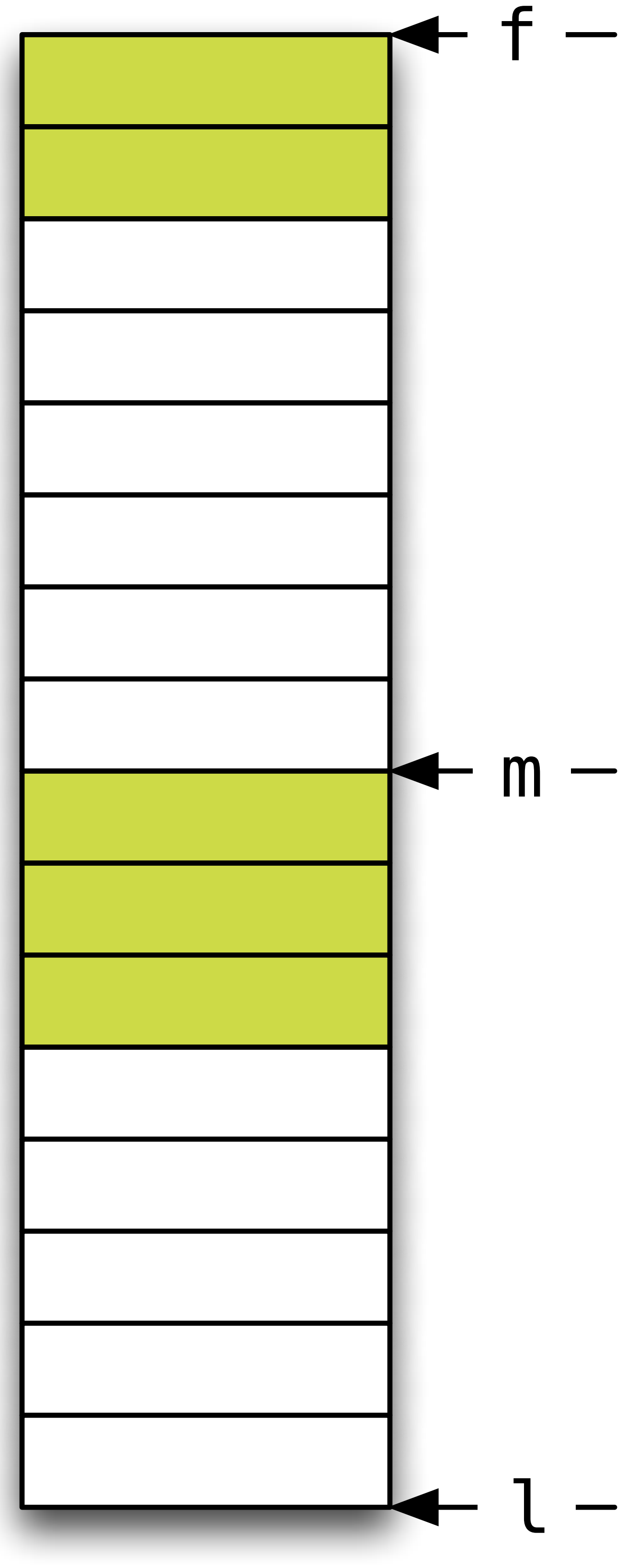
Stable Partition



Stable Partition



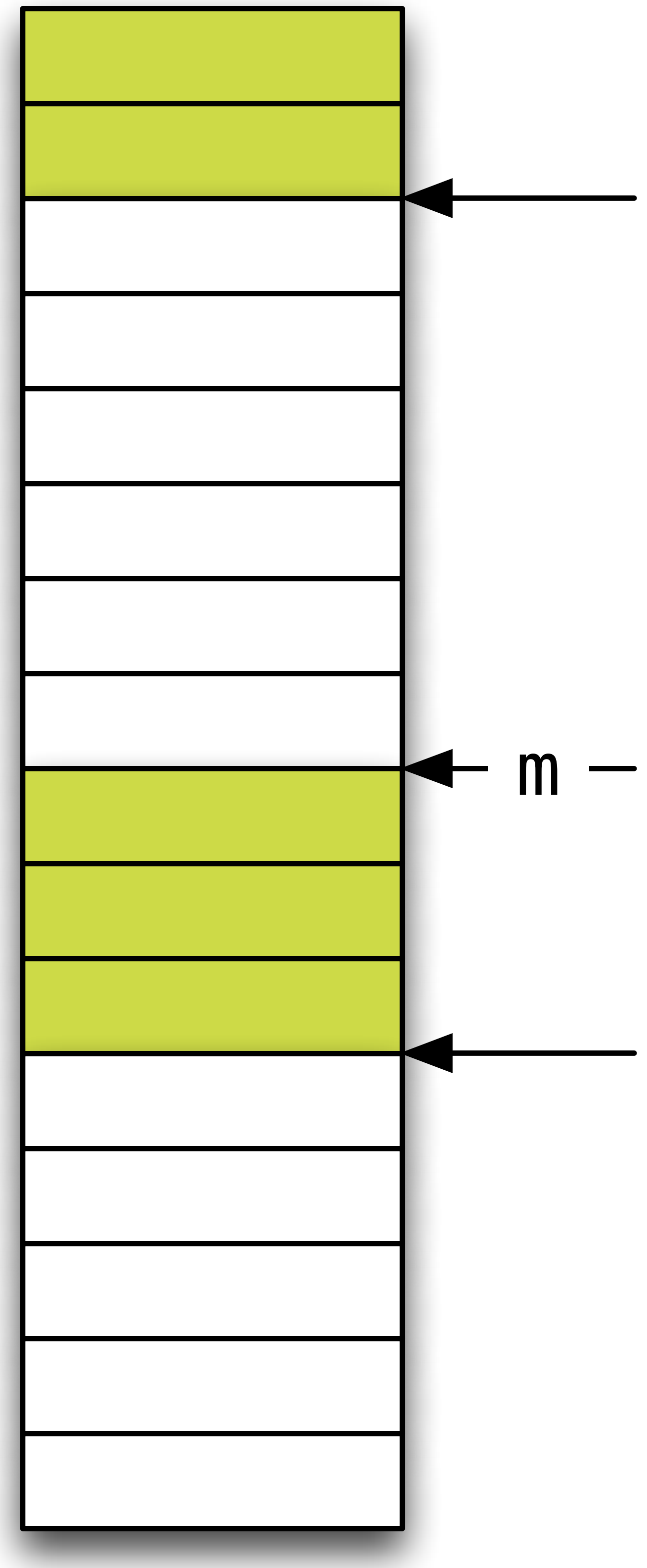
Stable Partition



```
stable_partition(f, m, p)

stable_partition(m, l, p)
```

Stable Partition

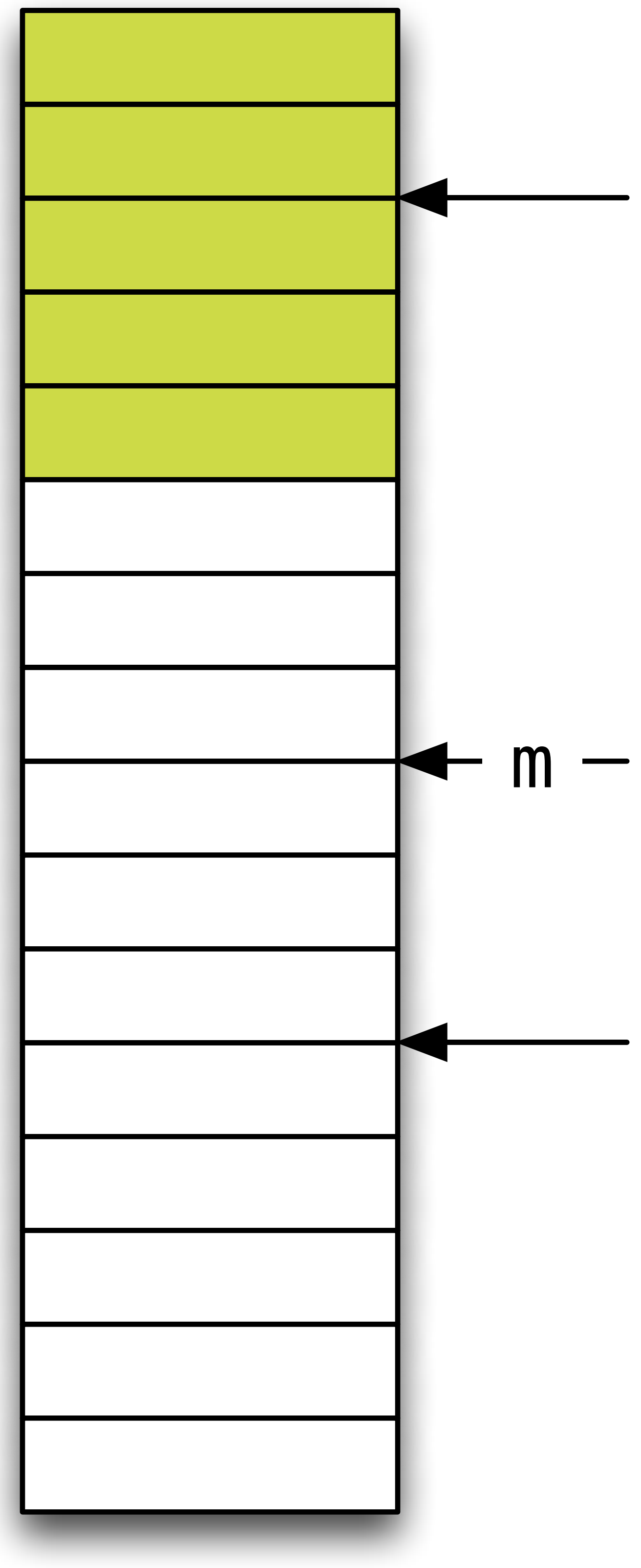
Stable Partition



```
stable_partition(f, m, p)

stable_partition(m, l, p)
```

Stable Partition

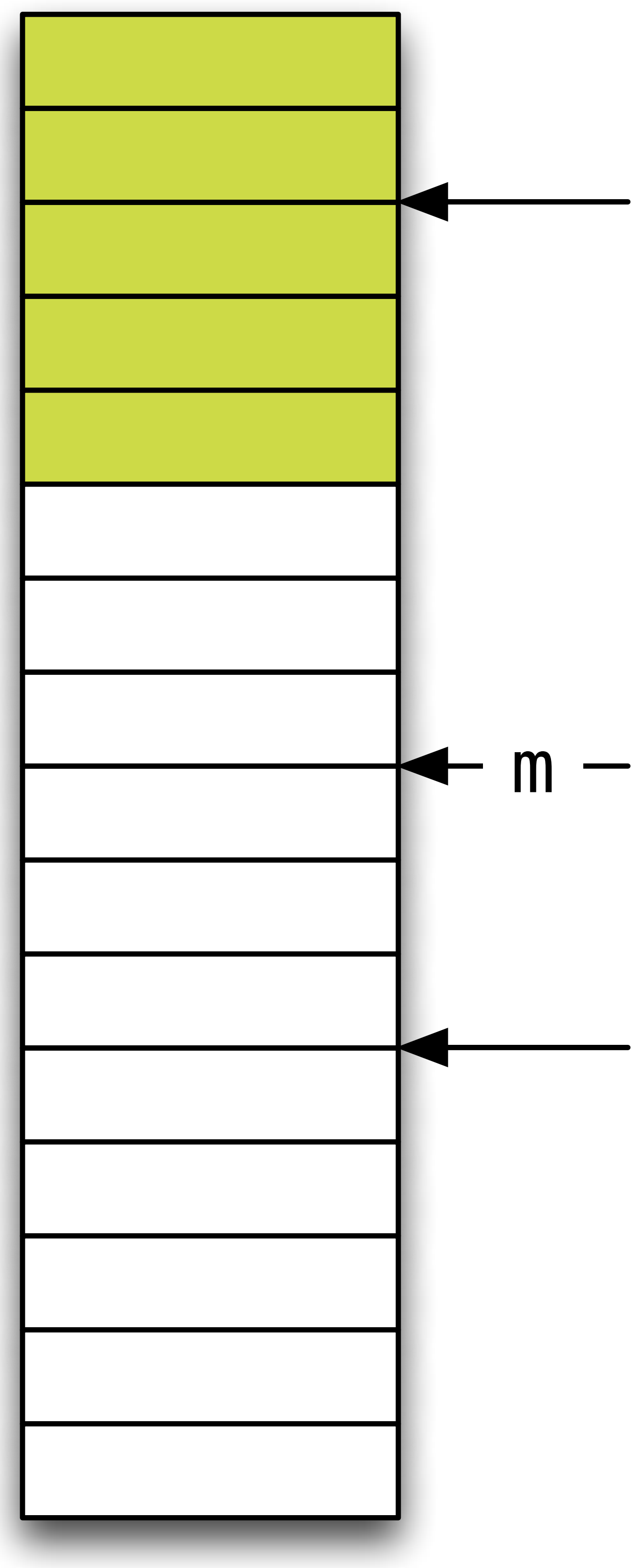
Stable Partition



```
stable_partition(f, m, p)

stable_partition(m, l, p)
```

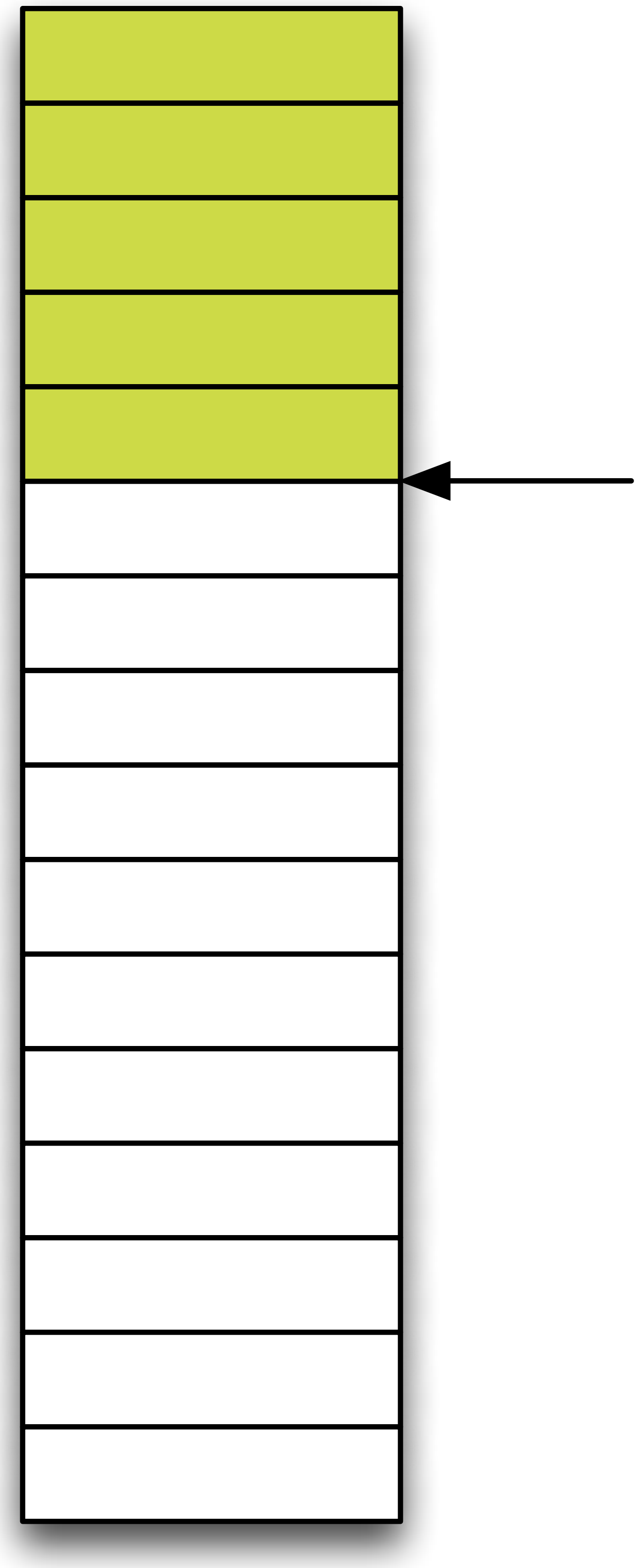
Stable Partition

Stable Partition



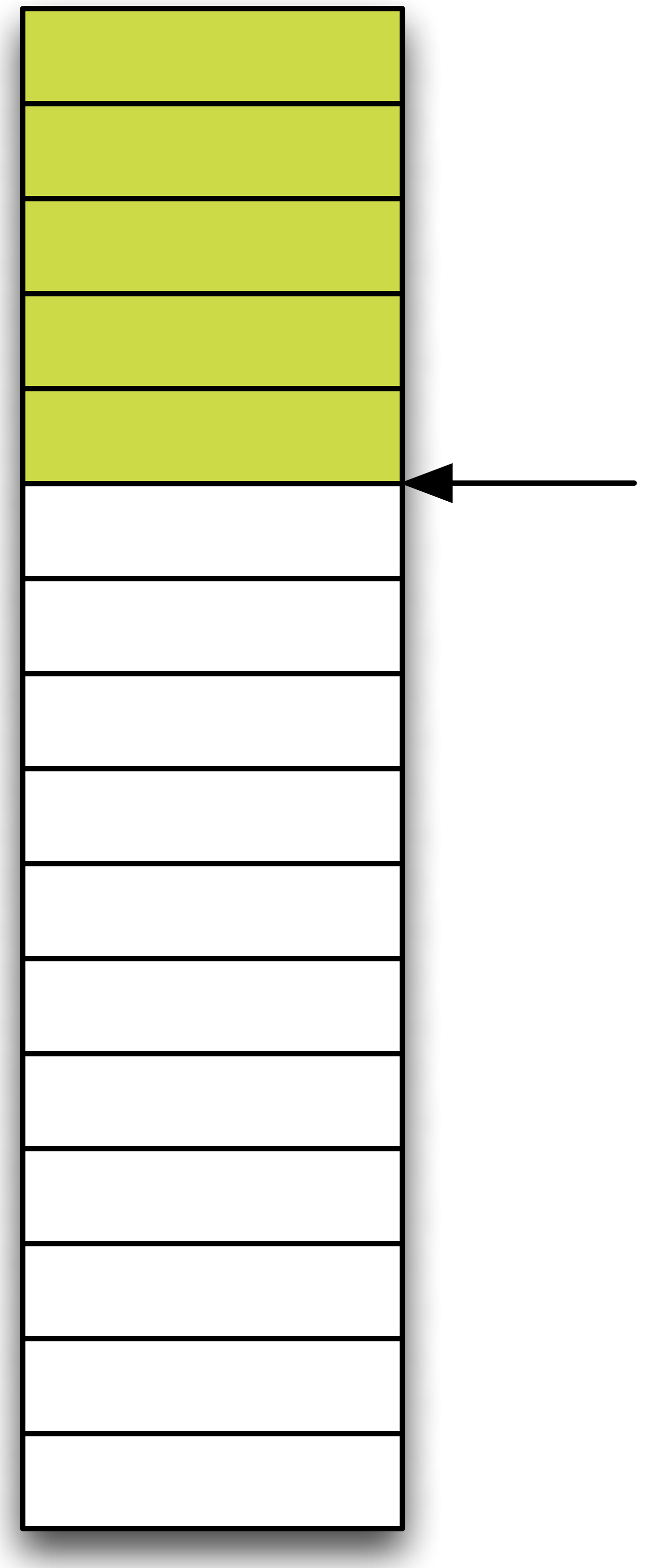
```
rotate(stable_partition(f, m, p),
       m,
       stable_partition(m, l, p));
```

Stable Partition



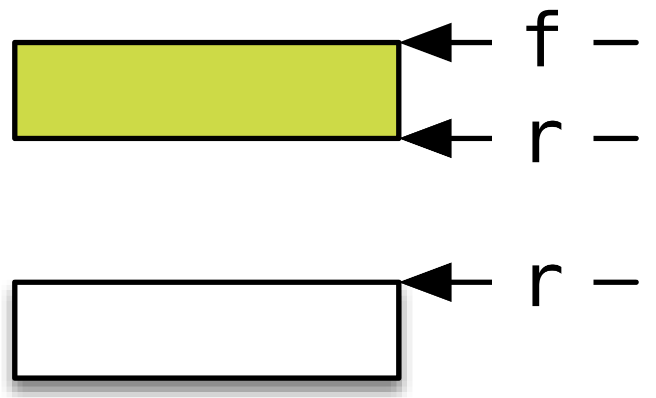
```
rotate(stable_partition(f, m, p),
       m,
       stable_partition(m, l, p));
```

Stable Partition



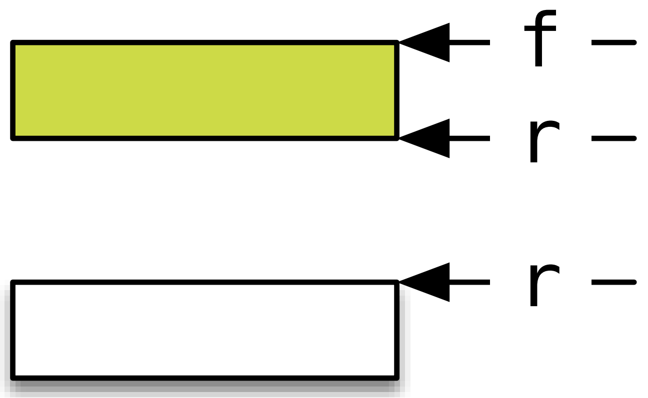
```
return rotate(stable_partition(f, m, p),
              m,
              stable_partition(m, l, p));
```

Stable Partition



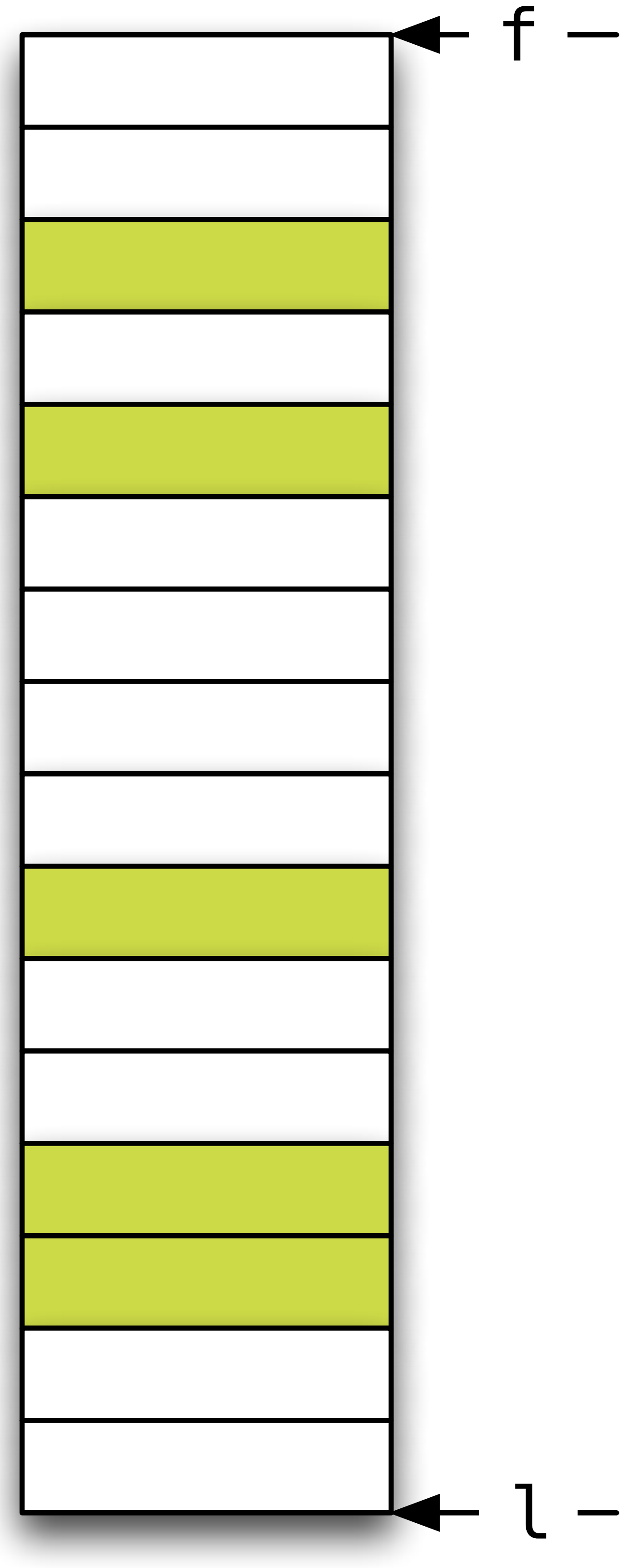
```
return rotate(stable_partition(f, m, p),
              m,
              stable_partition(m, l, p));
```

Stable Partition



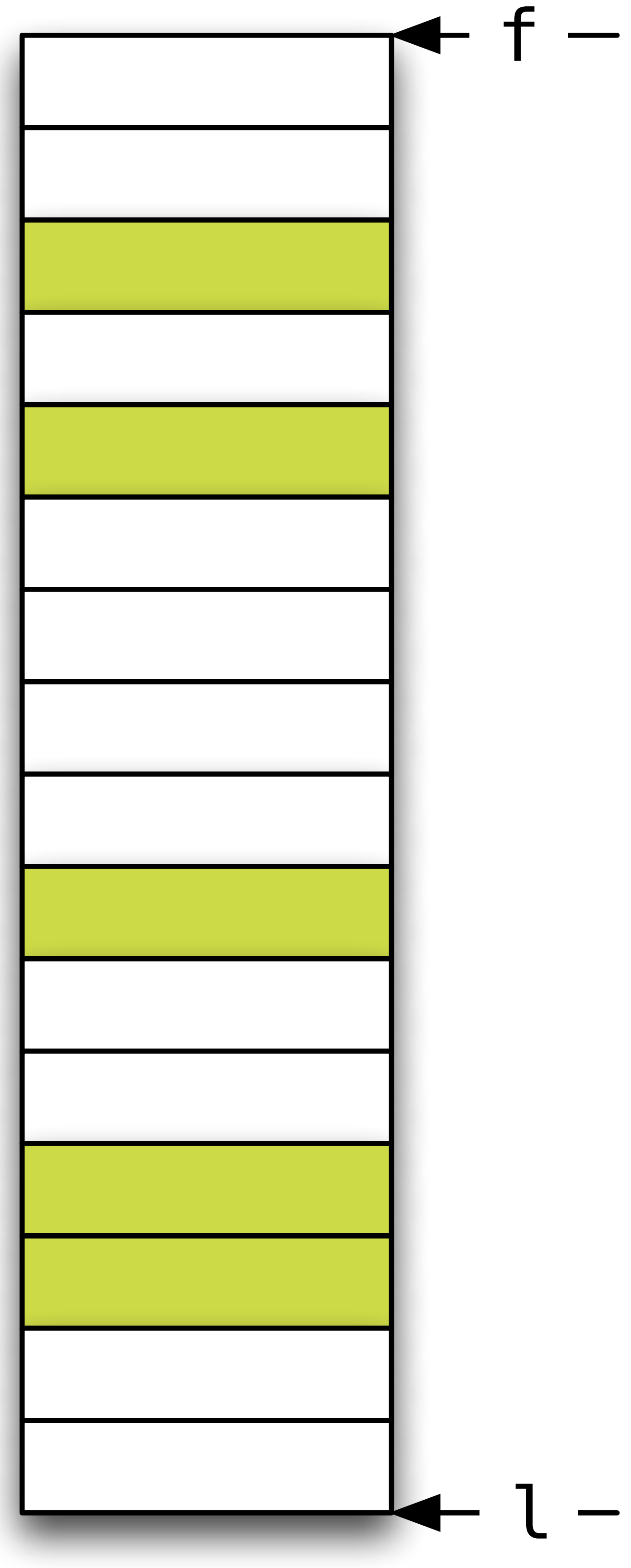
```
if (n == 1) return f + p(*f);


return rotate(stable_partition(f, m, p),
              m,
              stable_partition(m, l, p));
```

Stable Partition



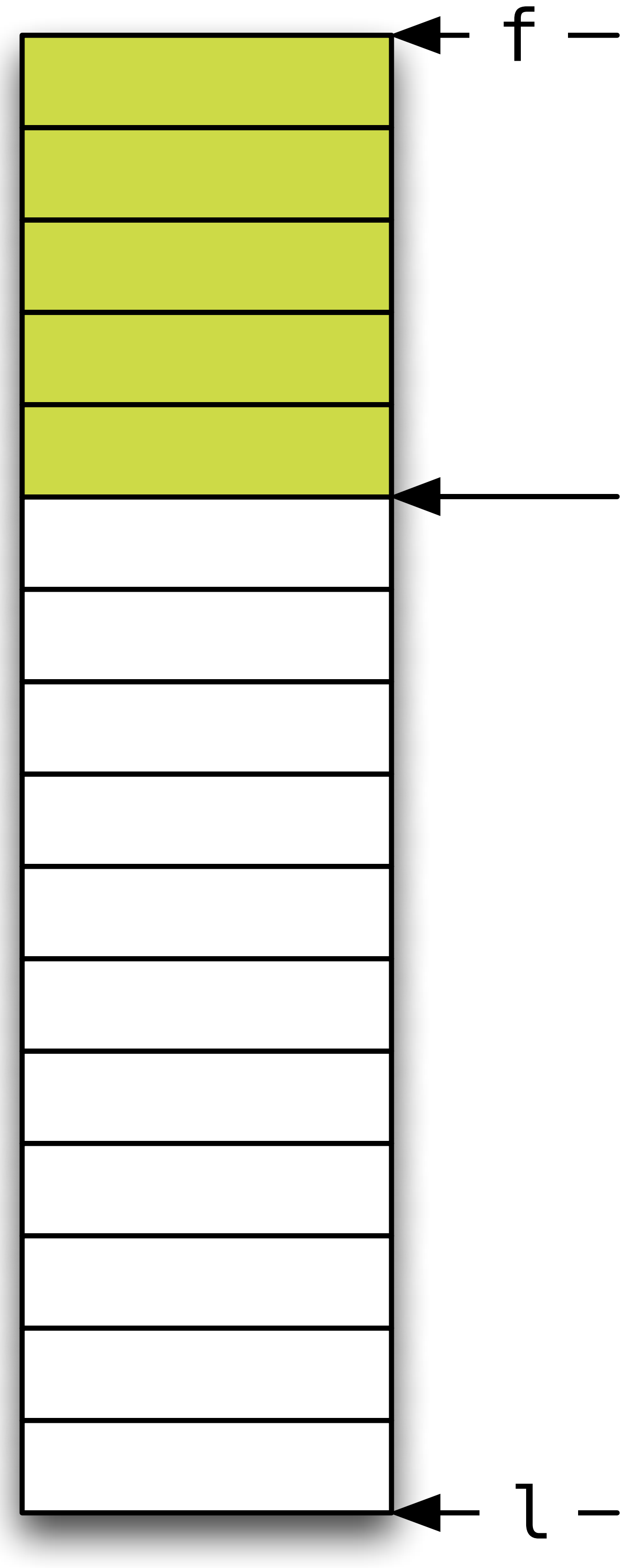
```
if (n == 1) return f + p(*f);


return rotate(stable_partition(f, m, p),
              m,
              stable_partition(m, l, p));
```

Stable Partition



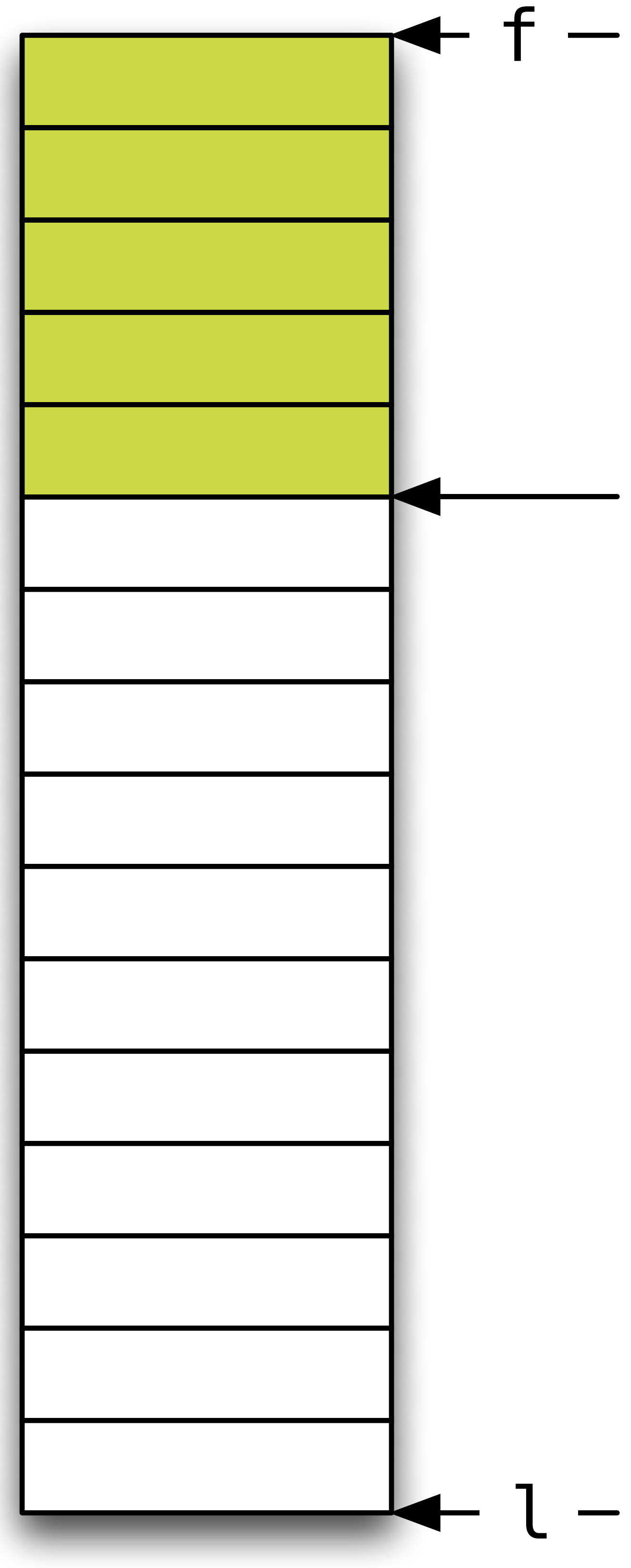
```
template <typename I,
          typename P>
auto stable_partition(I f, I l, P p) -> I
{
    auto n = l - f;
    if (n == 0) return f;
    if (n == 1) return f + p(*f);

    auto m = f + (n / 2);

    return rotate(stable_partition(f, m, p),
                  m,
                  stable_partition(m, l, p));
}
```

Stable Partition



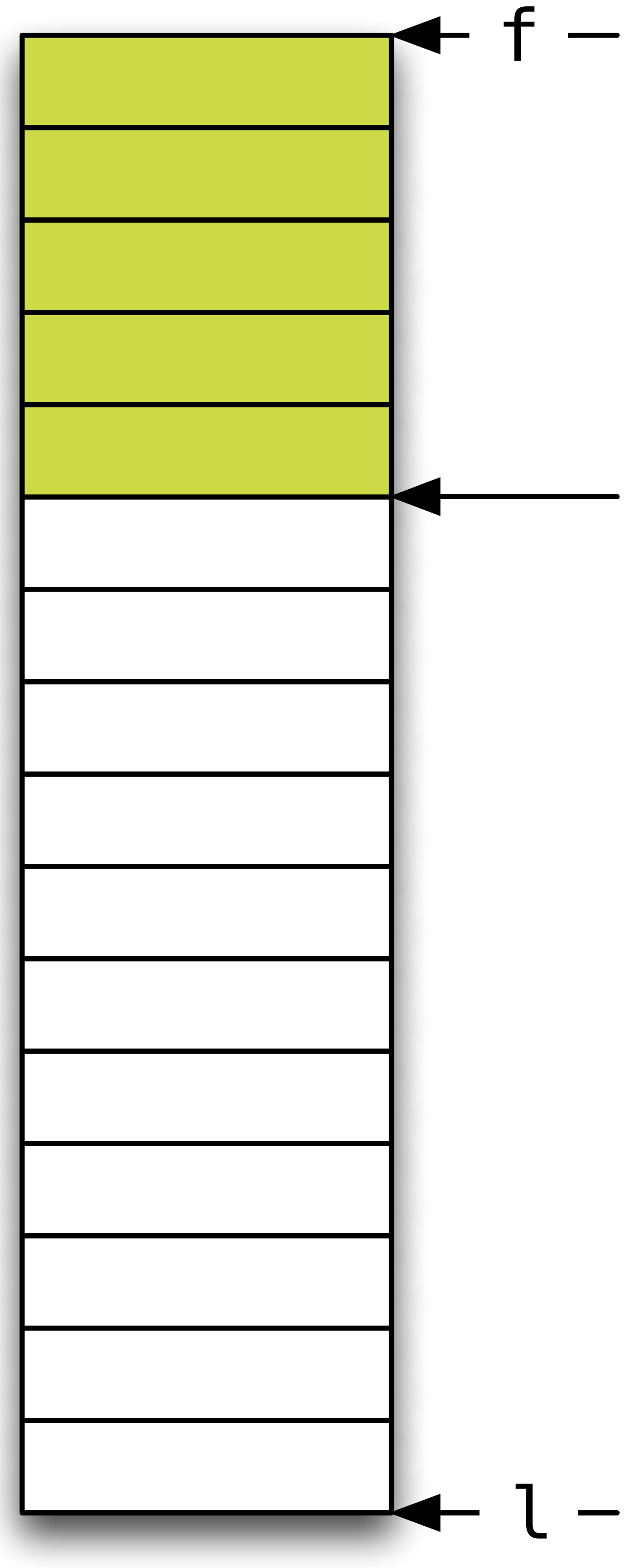
```
template <typename I,
          typename P>
auto stable_partition(I f, I l, P p) -> I
{
    auto n = l - f;
    if (n == 0) return f;
    if (n == 1) return f + p(*f);

    auto m = f + (n / 2);

    return rotate(stable_partition(f, m, p),
                  m,
                  stable_partition(m, l, p));
}
```

Stable Partition



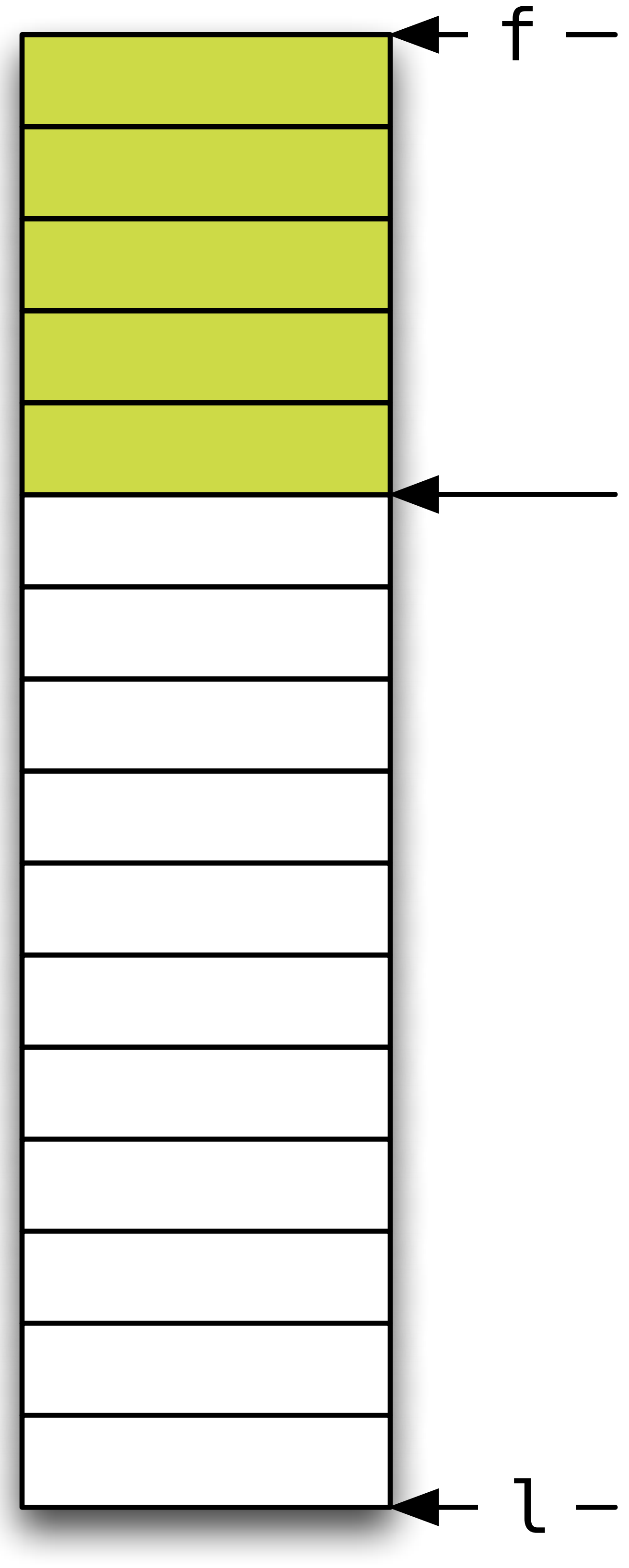
```
template <typename I,
          typename P>
auto stable_partition(I f, I l, P p) -> I
{
    auto n = l - f;
    if (n == 0) return f;
    if (n == 1) return f + p(*f);

    auto m = f + (n / 2);

    return rotate(stable_partition(f, m, p),
                  m,
                  stable_partition(m, l, p));
}
```

Stable Partition



```
template <typename I,
          typename P>
auto stable_partition(I f, I l, P p) -> I
{
    auto n = l - f;
    if (n == 0) return f;
    if (n == 1) return f + p(*f);

    auto m = f + (n / 2);

    return rotate(stable_partition(f, m, p),
                  m,
                  stable_partition(m, l, p));
}
```

Stable Partition



```
template <typename I,
          typename P>
auto stable_partition_position(I f, I l, P p) -> I
{
    auto n = l - f;
    if (n == 0) return f;
    if (n == 1) return f + p(f);

    auto m = f + (n / 2);

    return rotate(stable_partition_position(f, m, p),
                  m,
                  stable_partition_position(m, l, p));
}
```

Using gather_position

```
interval_set<I> selection;

//...

gather_position(f, l, p, [&](auto p) {
    return contains(selection, p);
});
```

Selections

- Multi-select is only sporadically implemented
  - Always inconsistently

Selections

One Way to Select Many*

Jaakko Järvi[1] and Sean Parent[2]

1 Texas A&M University
College Station, TX, USA
`jarvi@cse.tamu.edu`
2 Adobe Systems Inc.
San Jose, CA, USA
`sparent@adobe.com`

Abstract

Selecting items from a collection is one of the most common tasks users perform with graphical user interfaces. Practically every application supports this task with a selection feature different from that of any other application. Defects are common, especially in manipulating selections of non-adjacent elements, and flexible selection features are often missing when they would clearly be useful. As a consequence, user effort is wasted. The loss of productivity is experienced in small doses, but all computer users are impacted. The undesirable state of support for multi-element selection prevails because the same selection features are redesigned and reimplemented repeatedly. This article seeks to establish common abstractions for multi-selection. It gives generic but precise meanings to selection operations and makes multi-selection reusable; a JavaScript implementation is described. Application vendors benefit because of reduced development effort. Users benefit because correct and consistent multi-selection becomes available in more contexts.

1998 ACM Subject Classification D.2.11 Software Architectures: Domain-specific architectures; D.2.13 Reusable Software: Reusable libraries

Keywords and phrases User interfaces, Multi-selection, JavaScript

Digital Object Identifier 10.4230/LIPIcs.CVIT.2016.23

1 Introduction

Many, perhaps most, interactive software applications present their users one or more collections of elements in the form of lists, trees, grids, or otherwise arranged views, of which a user can select one or more elements. Examples include selecting files and folders in a file explorer; mail folders or mail messages in a mail client; music tracks in a media player; thumbnail images in a photograph organizer; "to do" list items, hours, days, weeks, or months in a calendar application; pages organized into "tabs" in a web browser; and electronic books or videos on a digital library or store. These tasks are typical daily activities for many computer users—we select elements from collections dozens of times per day.

Regardless of which set of modern applications a user chooses for mail, music, photos, calendar, web browsing, books, and videos, the features for selecting elements are likely to differ across applications—even within a single application the selection features for different collections, such as the list of mail folders and list of mail messages, are likely to be different.

The differences could presumably stem from optimizing the feature for the best possible user experience in different kinds of selection contexts, but this is not the case. The selection

---

* This work was supported in part by NSF grants CCF-0845861 and CCF-1320092.

42nd Conference on Very Important Topics (CVIT 2016).
Editors: John Q. Open and Joan R. Acces; Article No. 23; pp. 23:1–23:25
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Relationships

- A *relationship* is the way two entities are connected
  - Connective tissue between objects and properties
- A *structure* is formed by connected relationships
- *Architecture* is the art and science of designed structures

Relationships

- A *relationship* is the way two entities are connected
  - Connective tissue between objects and properties
- A *structure* is formed by connected relationships
- *Architecture* is the art and science of designed structures

- A relationship implies a *corresponding predicate* that tests if a pair exists in the relation

Relationships

- A *relationship* is the way two entities are connected
  - Connective tissue between objects and properties
- A *structure* is formed by connected relationships
- *Architecture* is the art and science of designed structures

- A relationship implies a *corresponding predicate* that tests if a pair exists in the relation

- Within an HI relationships can be challenging to represent

Relationships

Relationships

Requirement	Design	

Relationships

Requirement	**Design**	
num_showers >= 1		

Relationships

Requirement	**Design**	
num_showers >= 1	num_showers = 1;	

Relationships

Requirement	**Design**	
num_showers >= 1	num_showers = 1;	✅

Relationships

Requirement	**Design**	
num_showers >= 1	num_showers = 1;	✅
num_toilets >= 1		

Relationships

Requirement	Design	
num_showers >= 1	num_showers = 1;	✅
num_toilets >= 1	num_toilets = 1;	

Relationships

Requirement	**Design**	
num_showers >= 1	num_showers = 1;	✅
num_toilets >= 1	num_toilets = 1;	✅

Relationships

Requirement	**Design**	
num_showers >= 1	num_showers = 1;	✅
num_toilets >= 1	num_toilets = 1;	✅
privacy >= 0		

Relationships

Requirement	Design	
num_showers >= 1	num_showers = 1;	✅
num_toilets >= 1	num_toilets = 1;	✅
privacy >= 0	privacy = 0;	

Relationships

Requirement	**Design**	
num_showers >= 1	num_showers = 1;	✅
num_toilets >= 1	num_toilets = 1;	✅
privacy >= 0	privacy = 0;	✅

“Simple” Relationship

$$a \Rightarrow b$$

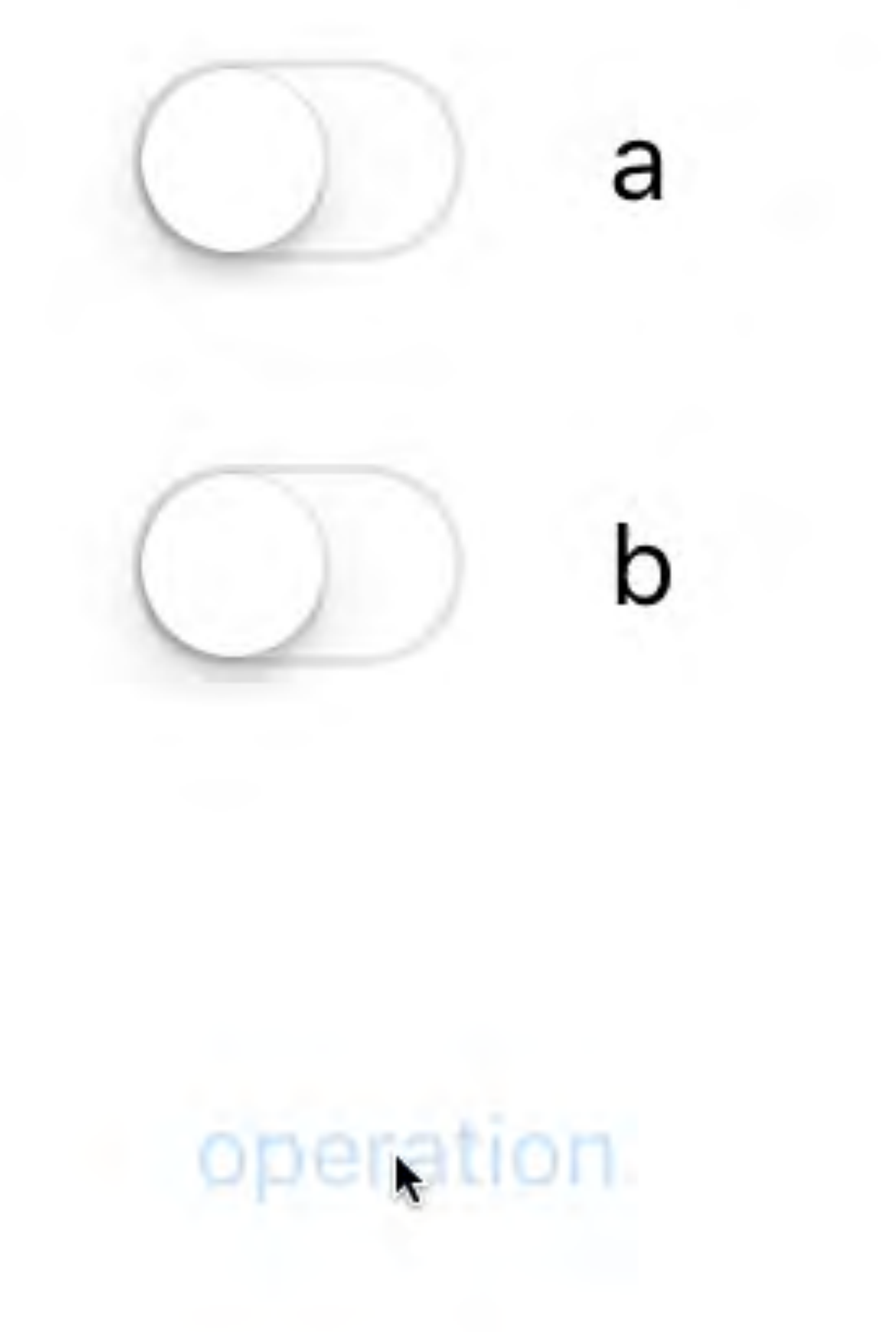
$(a \text{ implies } b)$

Implies (examples from the clang manual)

- “-ggdb, -glldb, -gsce … Each of these options **implies** -g.”
- “-f[no-]diagnostics-show-hotness … This option is **implied** when -fsave-optimization-record is used.”
- “-M, --dependencies … Like -MD, but also **implies** -E”
- “-MM, --user-dependencies … Like -MMD, but also **implies** -E”
- “-cl-unsafe-math-optimizations … Also **implies** -cl-no-signed-zeros and -cl-mad-enable.”

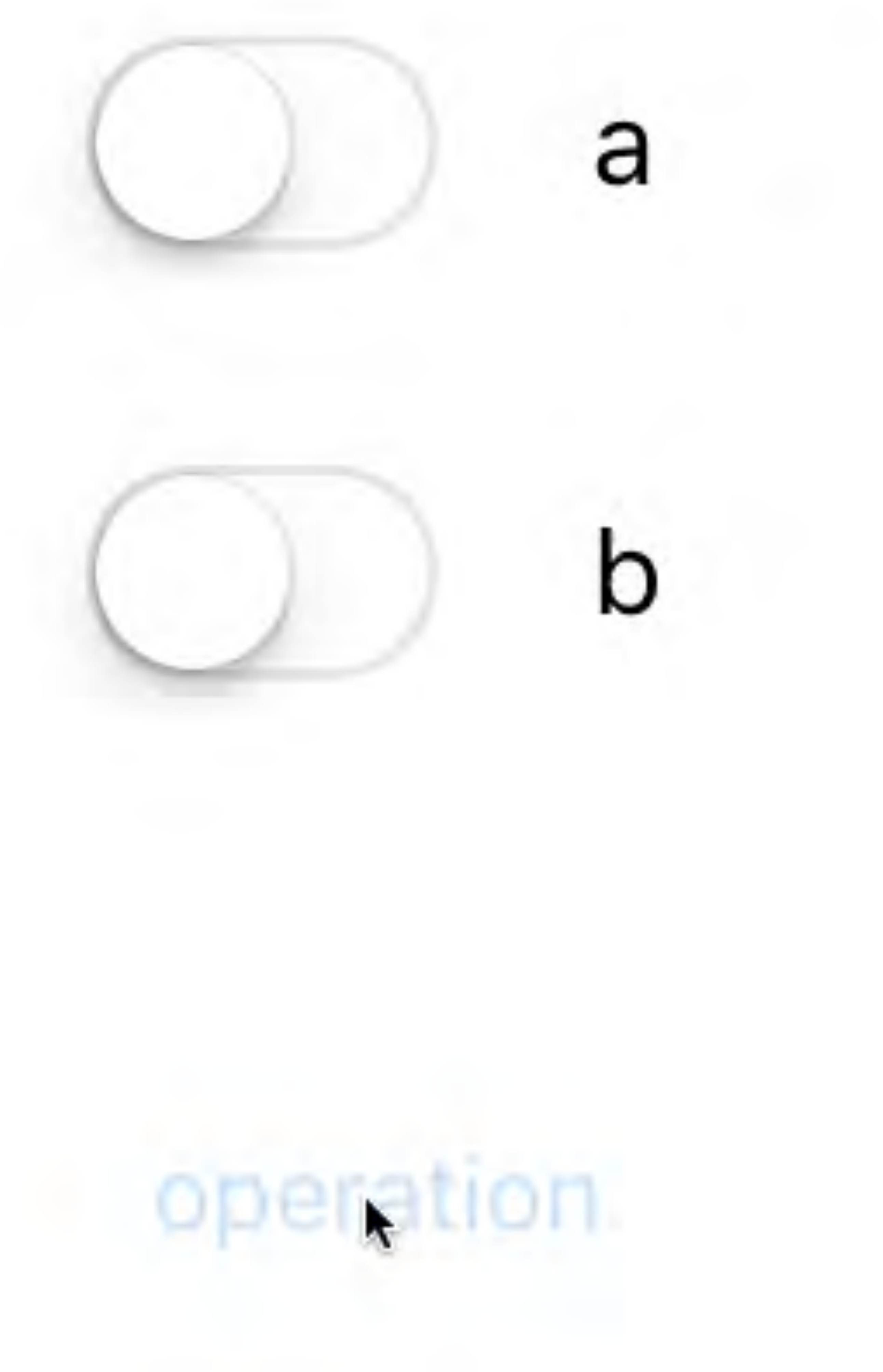
Unconstrained

```
void operation(bool a, bool b) {
    b = a || b; // a implies b
    //...
}
```



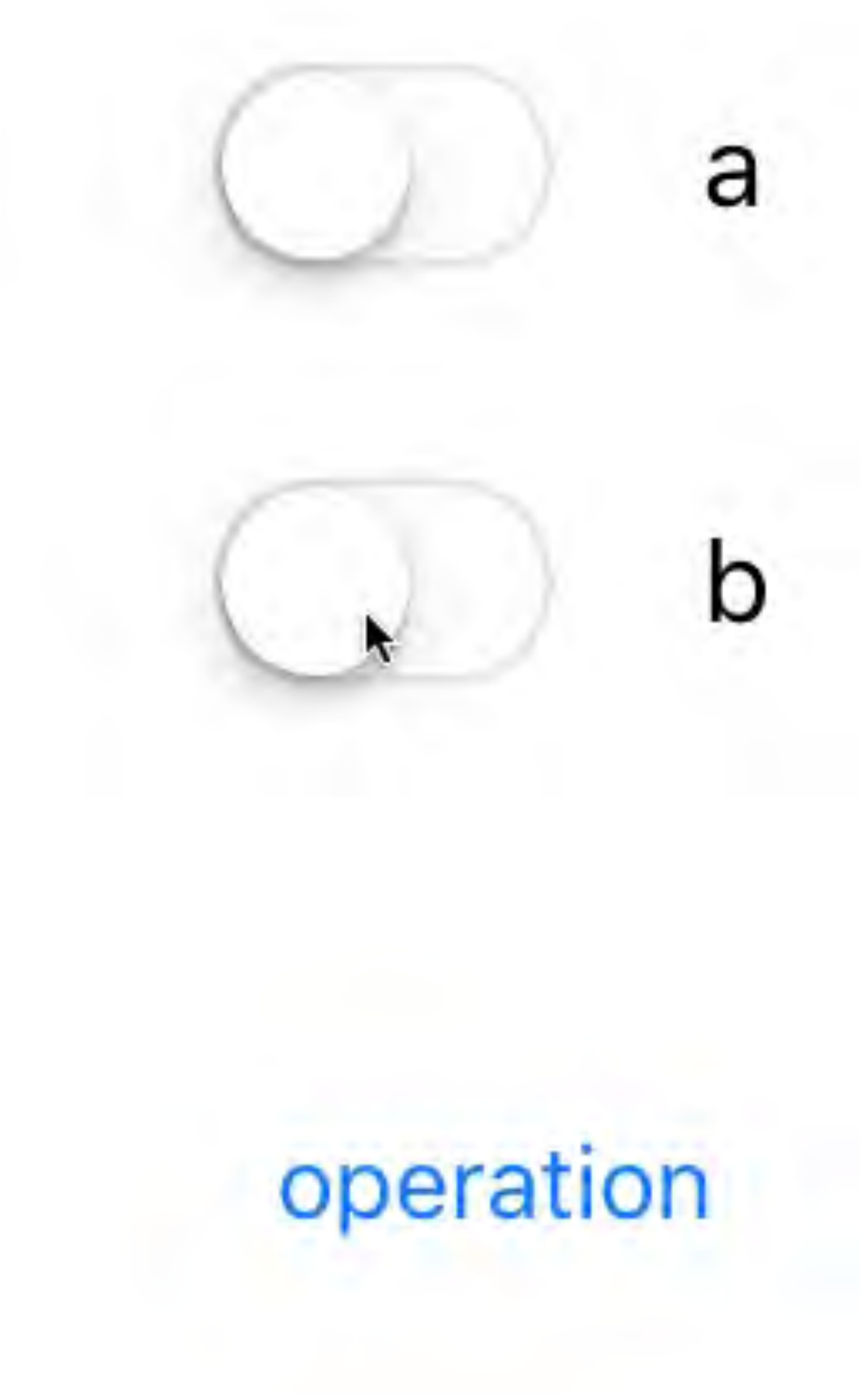
Unconstrained

```
void operation(bool a, bool b) {
    b = a || b; // a implies b
    //...
}
```



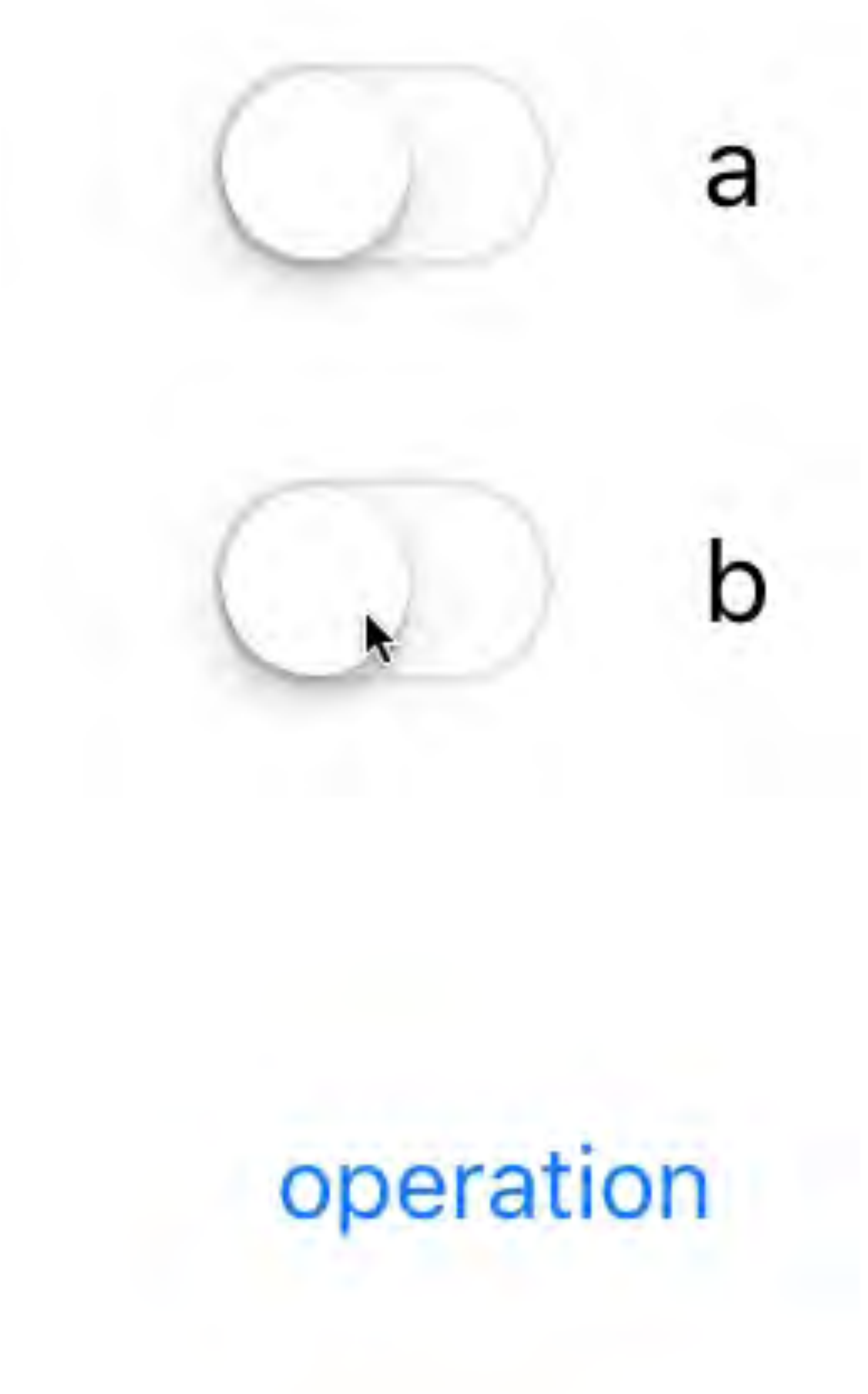
First Attempt

```
- (IBAction)aChanged {
  if (_aSwitch.on) _bSwitch.on = true;
}

void operation(bool a, bool b) {
    assert(!a || b); // a implies b
    //...
}
```



First Attempt

```
- (IBAction)aChanged {
  if (_aSwitch.on) _bSwitch.on = true;
}

void operation(bool a, bool b) {
    assert(!a || b); // a implies b
    //...
}
```



Goal

Goal

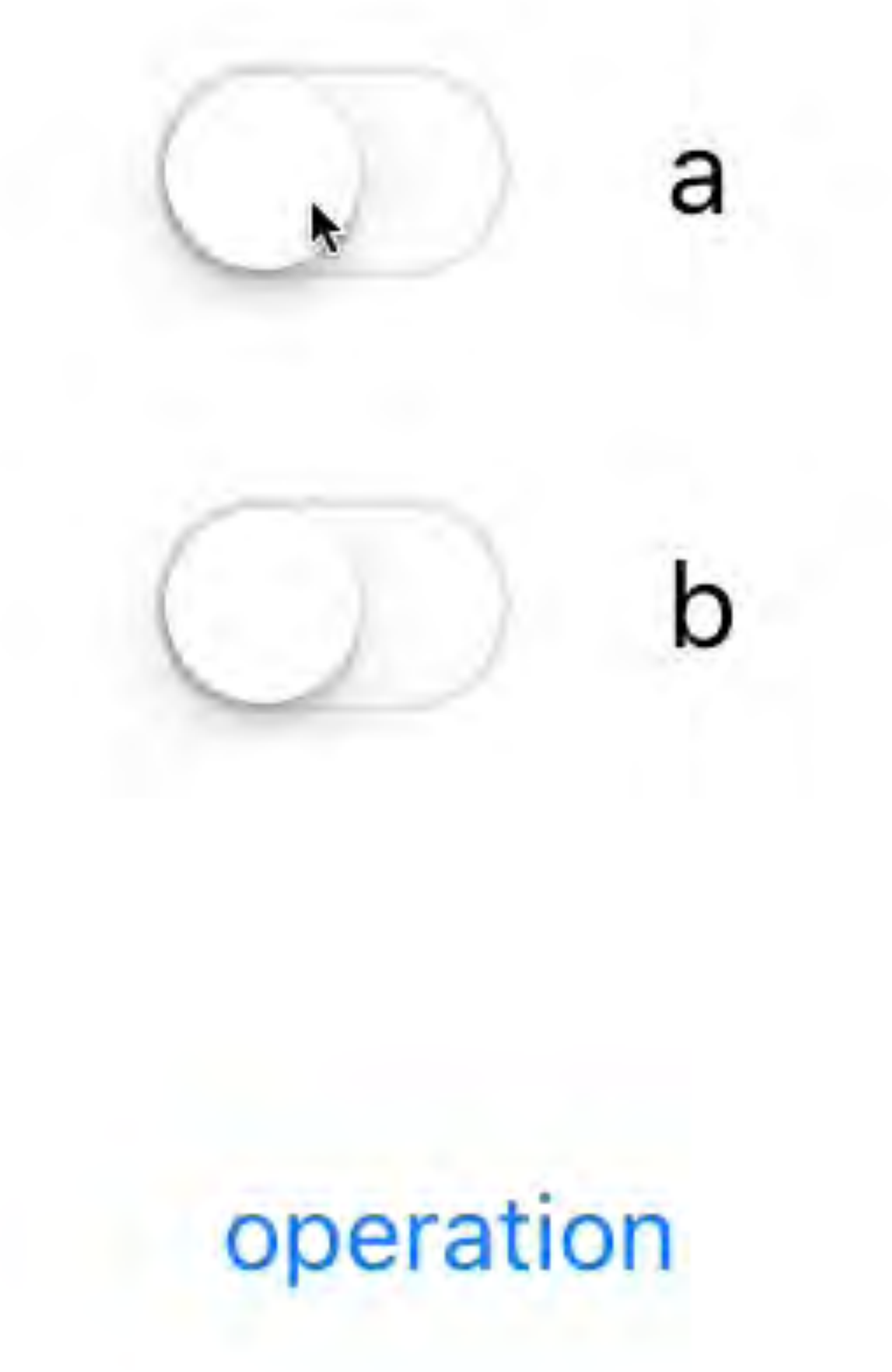
Use strong preconditions

Goal

Use strong preconditions and assert them

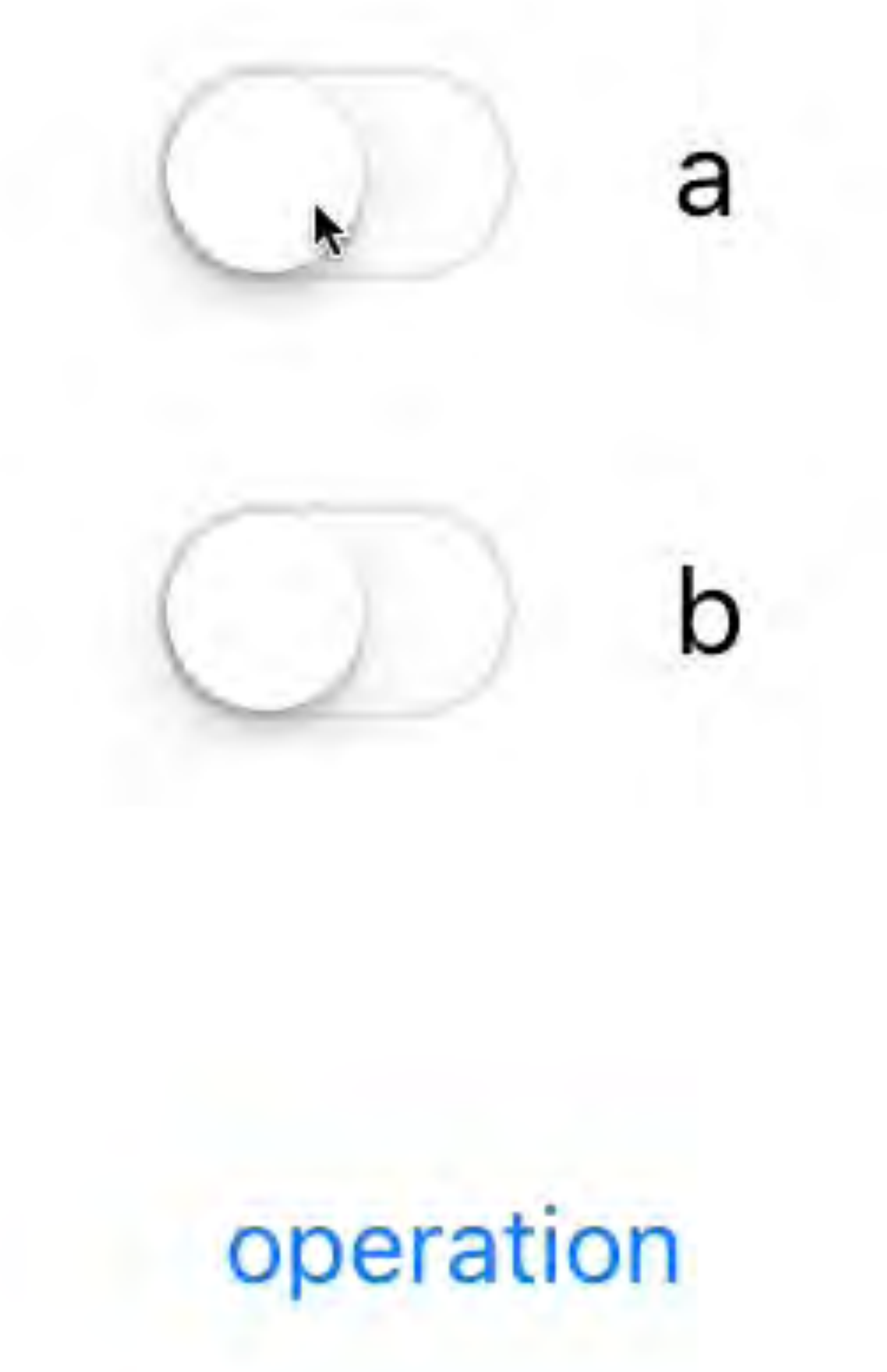
Disable

```
- (IBAction)aChanged {
    _bSwitch.enabled = !_aSwitch.on;
    if (_aSwitch.on) _bSwitch.on = true;
}

void operation(bool a, bool b) {
    assert(!a || b); // a implies b
    //...
}
```



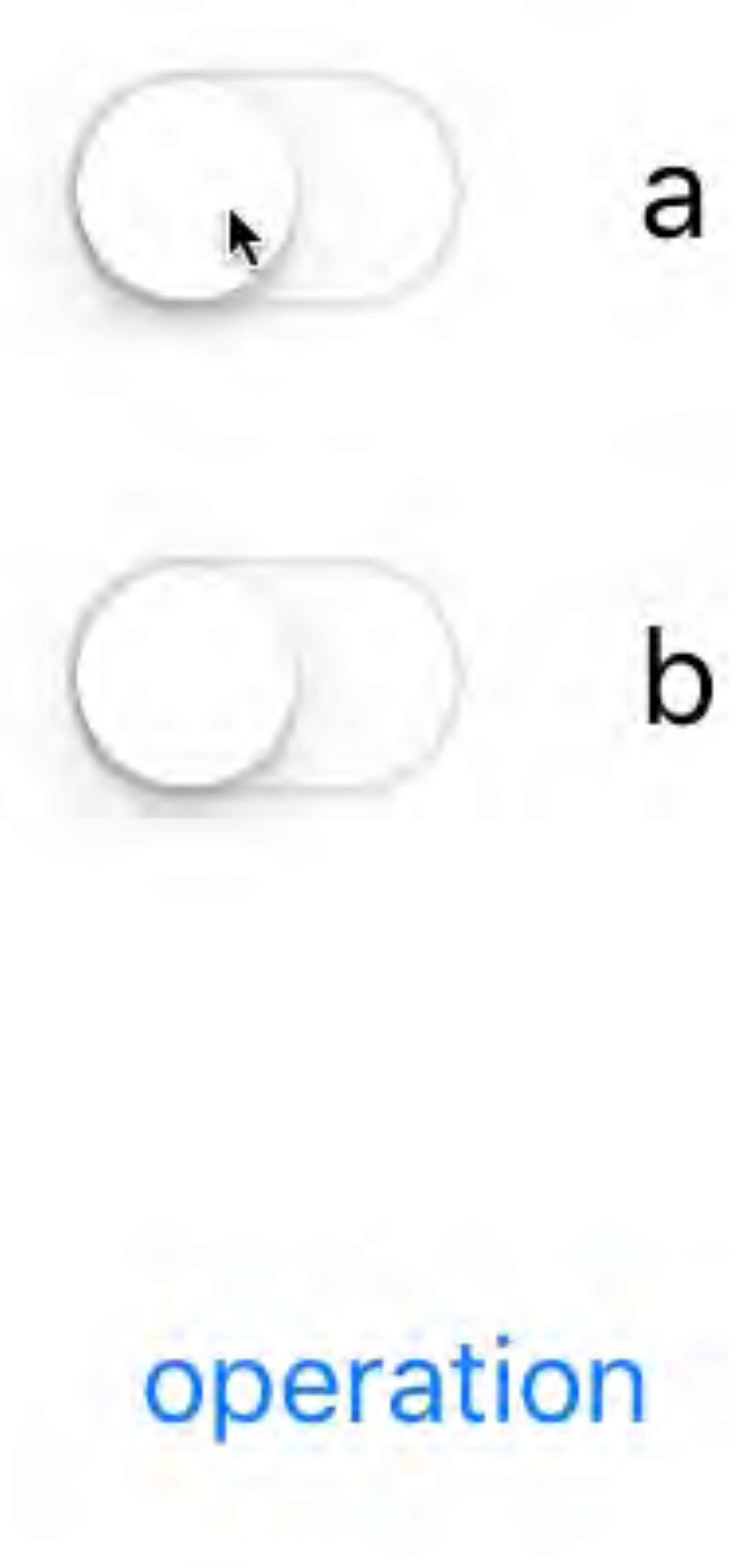
Disable

```
- (IBAction)aChanged {
    _bSwitch.enabled = !_aSwitch.on;
    if (_aSwitch.on) _bSwitch.on = true;
}

void operation(bool a, bool b) {
    assert(!a || b); // a implies b
    //...
}
```



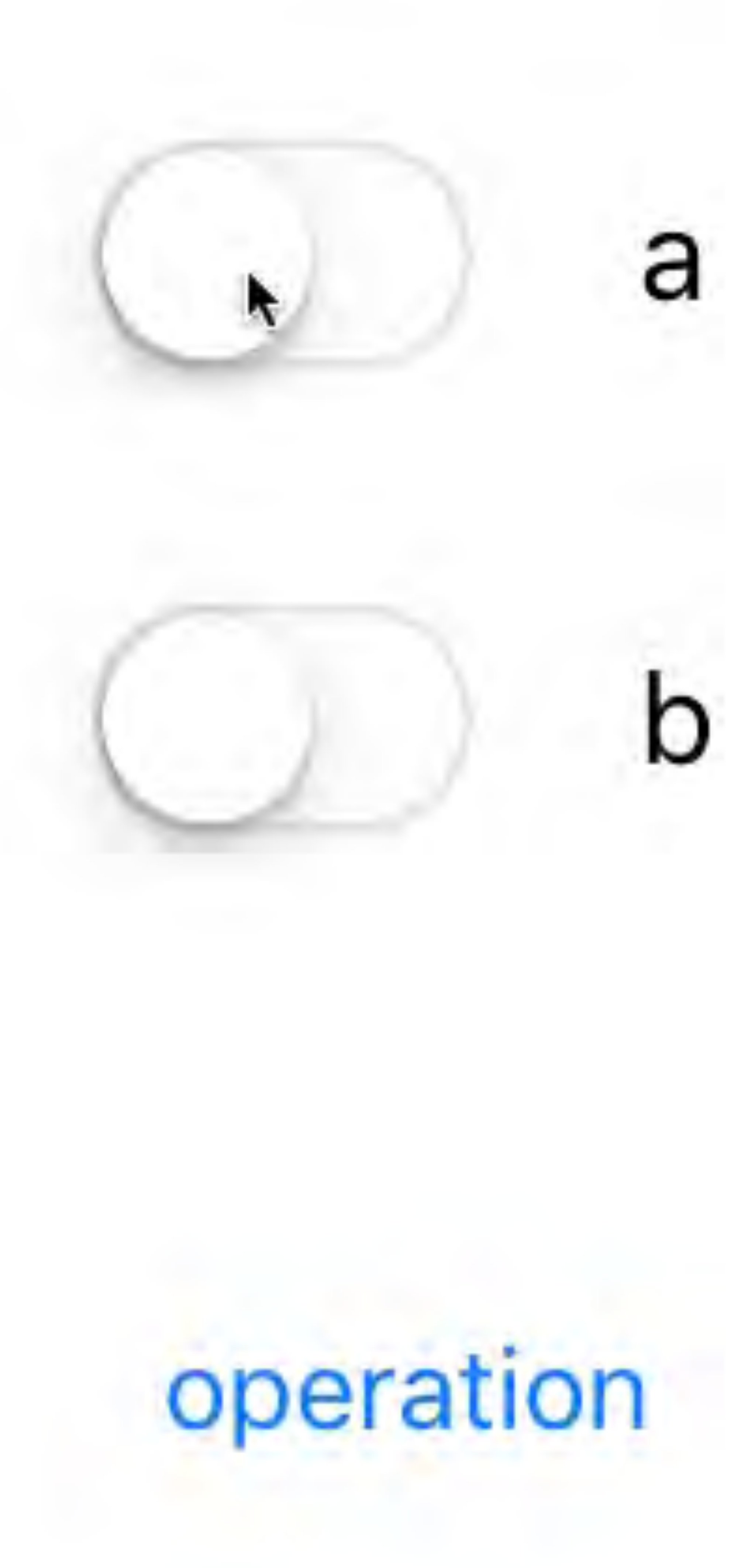
Disable + Memory

```
- (IBAction)aChanged {
    _bSwitch.enabled = !_aSwitch.on;
    _bSwitch.on = _aSwitch.on || _b;
}

- (IBAction)bChanged {
    _b = _bSwitch.on;
}

void operation(bool a, bool b) {
    assert(!a || b); // a implies b
    //...
}
```



Disable + Memory

```
- (IBAction)aChanged {
    _bSwitch.enabled = !_aSwitch.on;
    _bSwitch.on = _aSwitch.on || _b;
}

- (IBAction)bChanged {
    _b = _bSwitch.on;
}

void operation(bool a, bool b) {
    assert(!a || b); // a implies b
    //...
}
```



Contrapositive

Contrapositive

$$\neg b \Rightarrow \neg a$$

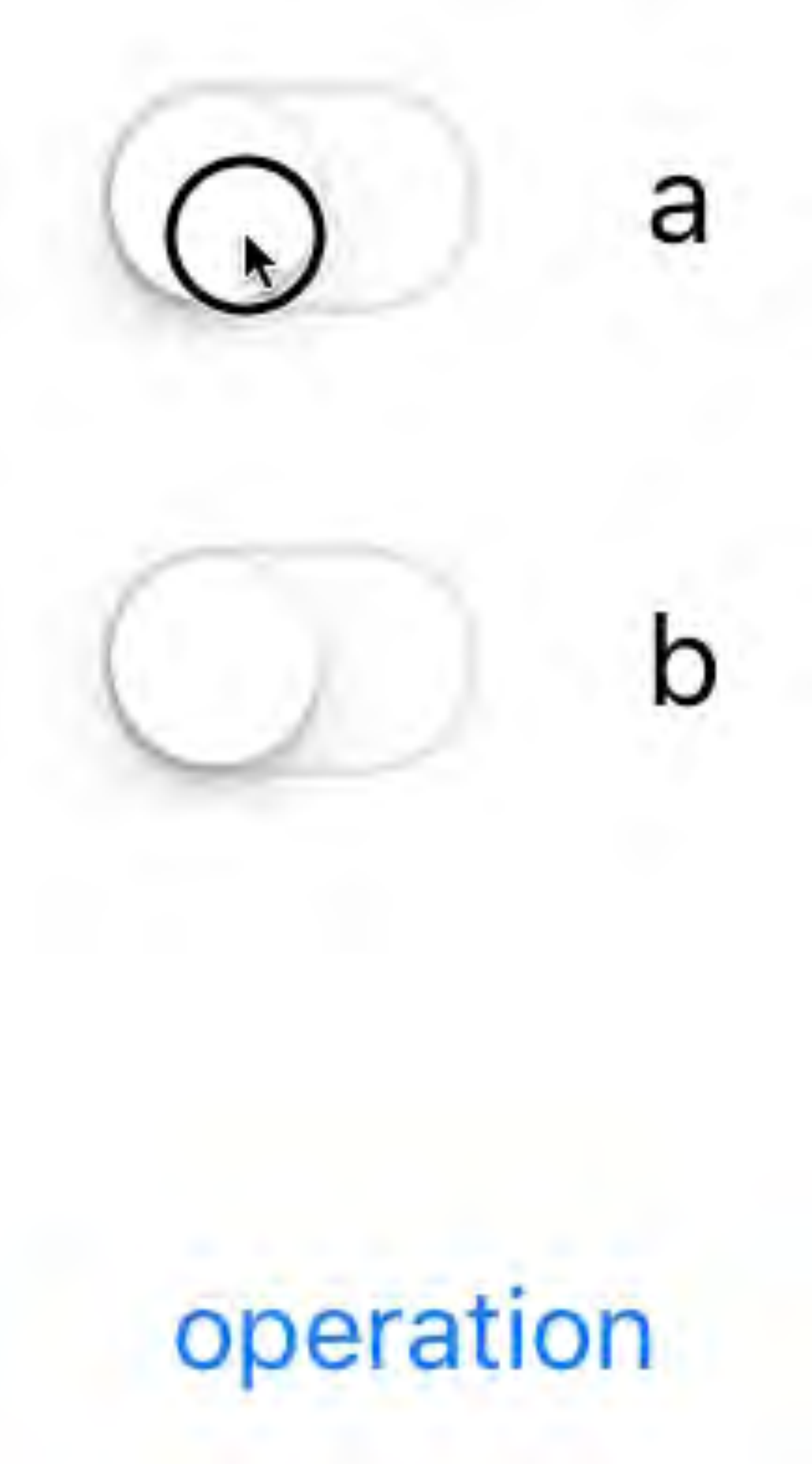
Contrapositive

$$\neg b \Rightarrow \neg a$$

(*not b* implies *not a*)

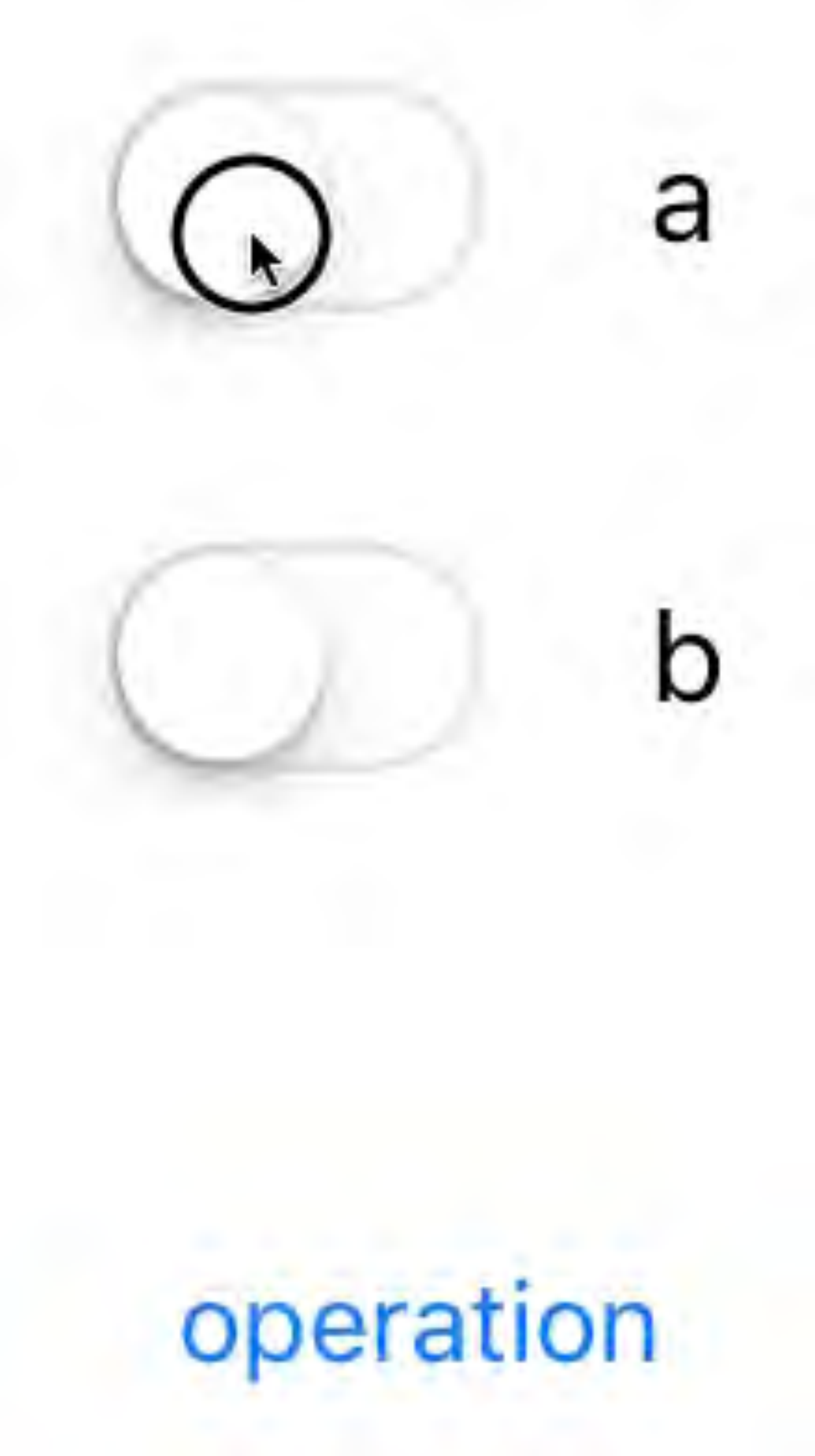
Contrapositive + Memory

```
- (IBAction)aChanged {
    _a = _aSwitch.on;
    _bSwitch.on = _a || _b;
}

- (IBAction)bChanged {
    _b = _bSwitch.on;
    _aSwitch.on = _b && _a;
}

void operation(bool a, bool b) {
    assert(!a || b); // a implies b
    //...
}
```



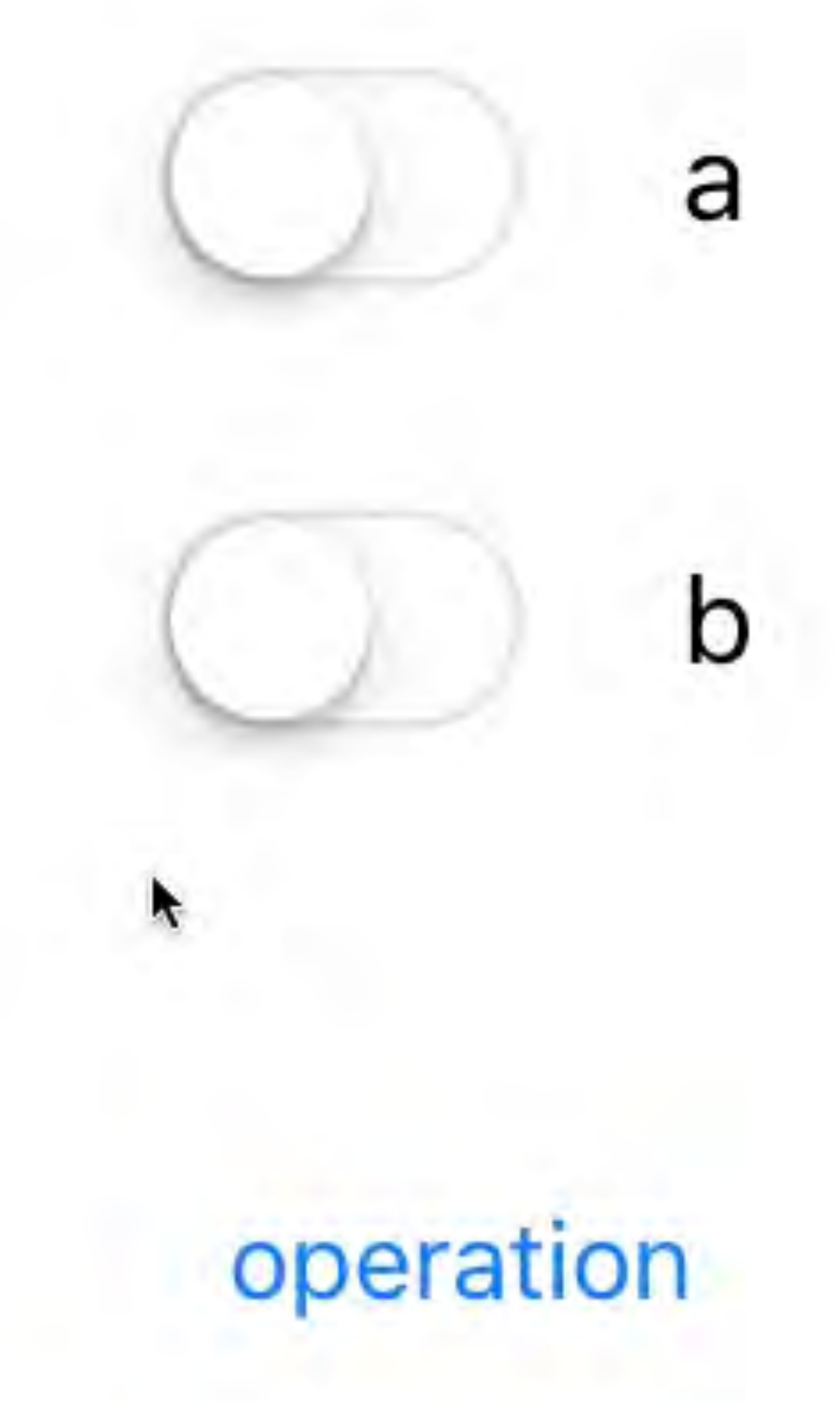
Contrapositive + Memory

```
- (IBAction)aChanged {
    _a = _aSwitch.on;
    _bSwitch.on = _a || _b;
}

- (IBAction)bChanged {
    _b = _bSwitch.on;
    _aSwitch.on = _b && _a;
}

void operation(bool a, bool b) {
    assert(!a || b); // a implies b
    //...
}
```



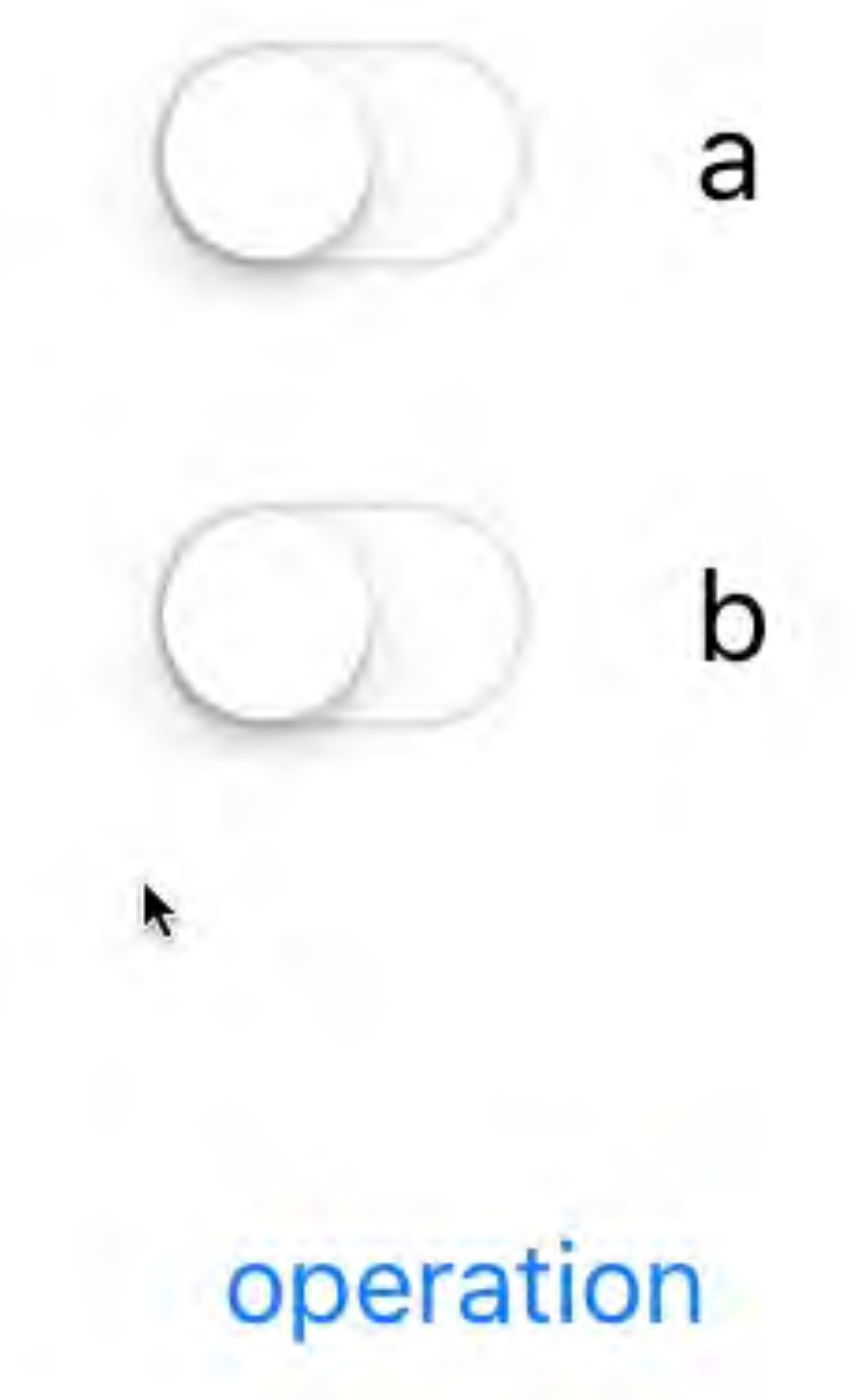
Contrapositive

```
- (IBAction)aChanged {
    _bSwitch.on = _aSwitch.on || _bSwitch.on
}

- (IBAction)bChanged {
    _aSwitch.on = _bSwitch.on && _aSwitch.on
}

void operation(bool a, bool b) {
    assert(!a || b); // a implies b
    //...
}
```



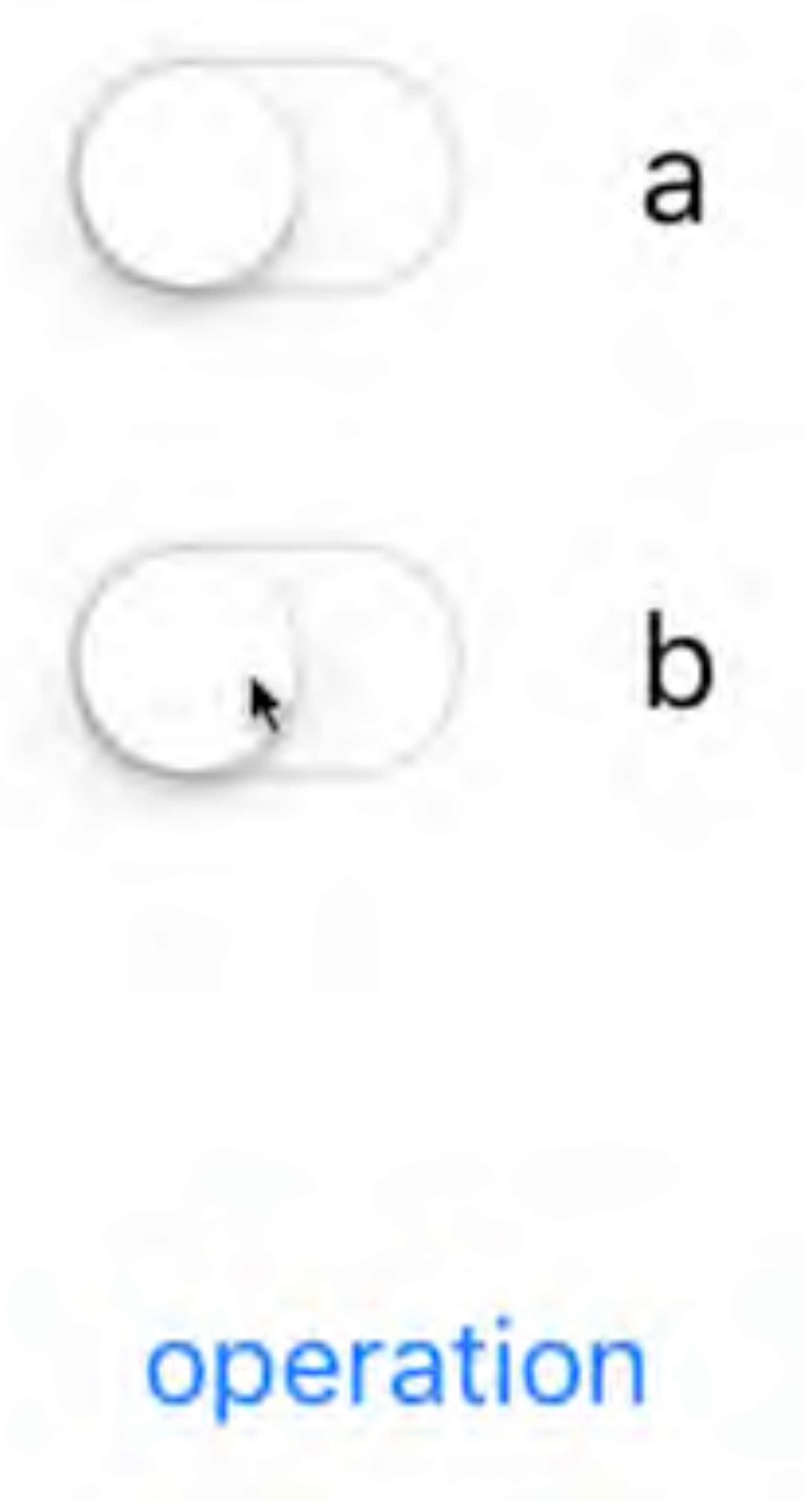
Contrapositive

```
- (IBAction)aChanged {
    _bSwitch.on = _aSwitch.on || _bSwitch.on
}

- (IBAction)bChanged {
    _aSwitch.on = _bSwitch.on && _aSwitch.on
}

void operation(bool a, bool b) {
    assert(!a || b); // a implies b
    //...
}
```



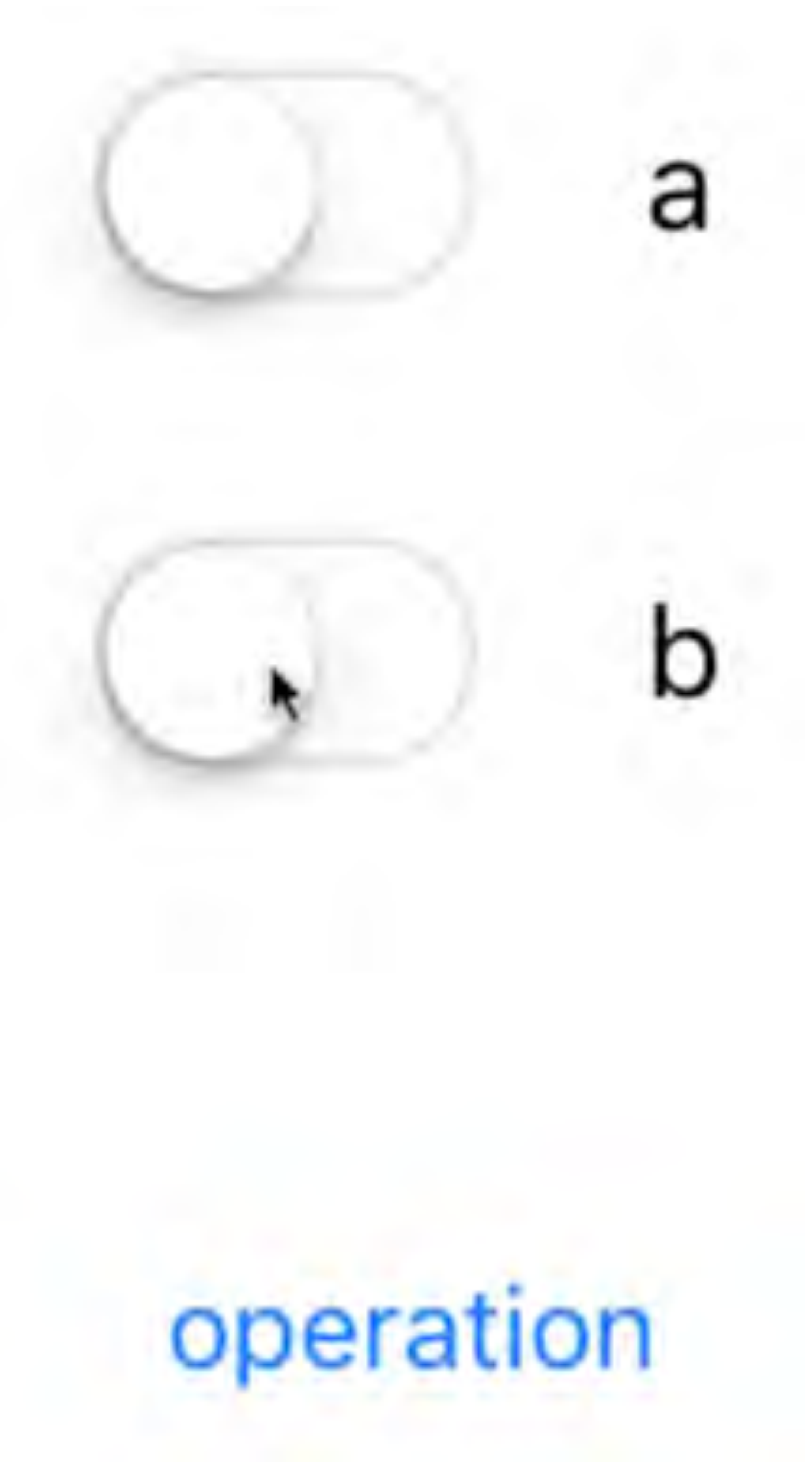
Unconstrained + Disable Operation

```
- (IBAction)changed {
    _operation.enabled =
        !_aSwitch.on || _bSwitch.on;
}

void operation(bool a, bool b) {
    assert(!a || b); // a implies b
    //...
}
```

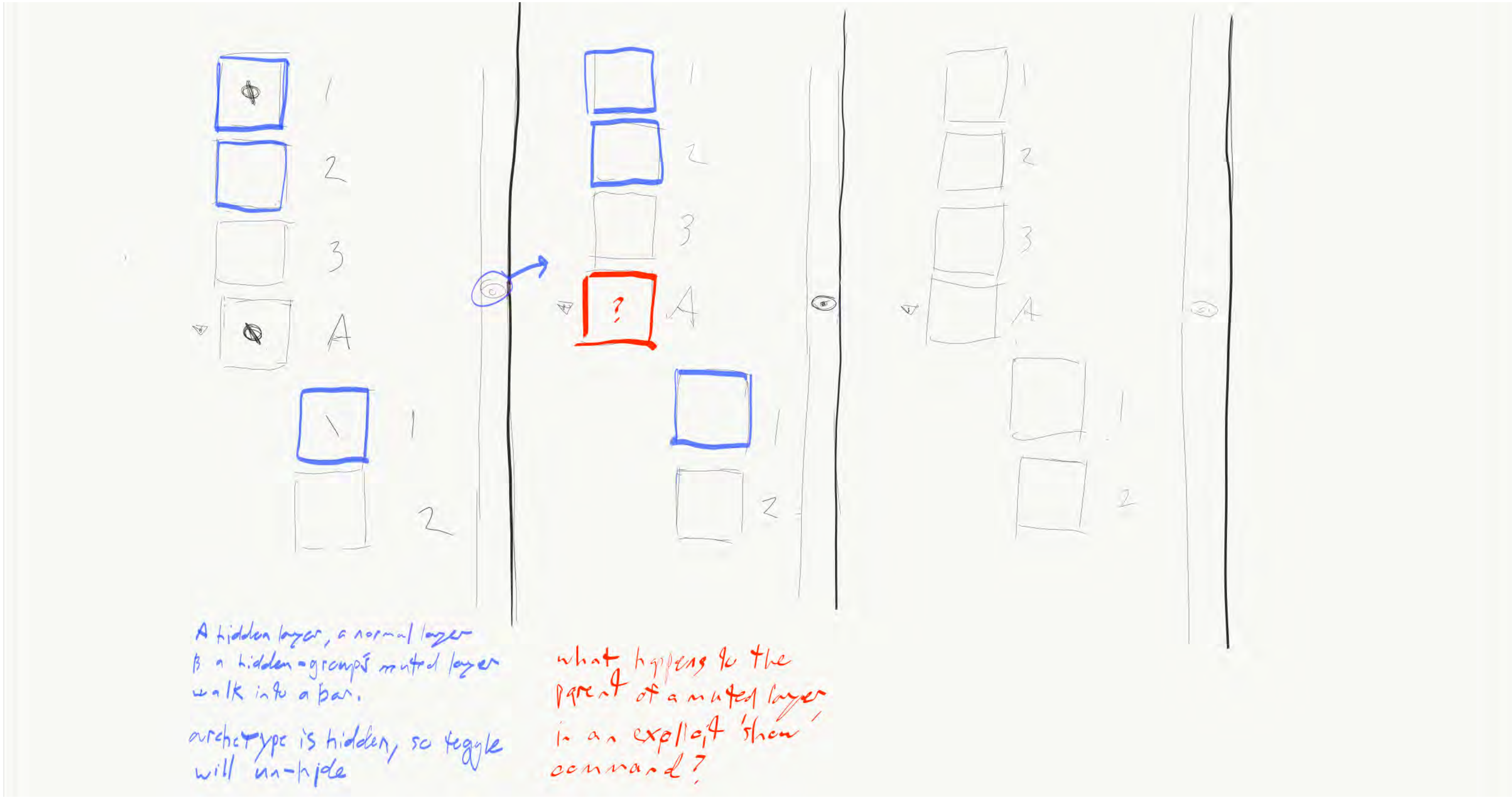


Unconstrained + Disable Operation

```
- (IBAction)changed {
    _operation.enabled =
        !_aSwitch.on || _bSwitch.on;
}

void operation(bool a, bool b) {
    assert(!a || b); // a implies b
    //...
}
```



Relationships



A hidden layer, a normal layer
B a hidden-group's muted layer
walk into a bar.

archetype is hidden, so toggle
will un-hide

what happens to the
parent of a muted layer
in an explicit 'show'
command?

What is a good design?

- Toggling a control should restore system to original state
- Result of a click should be predictable without knowing how current state was achieved
- Guided paths are preferred so long as they don’t make navigation more difficult

- But there needs to be additional rules to handle conflicts
- Rules derived from
  - Convention
  - Experience
  - Studies

Property Models

- Unconstrained

```
a; b;
```

- Disabled

```
b <== a || b;
```

- Disabled + Memory

```
unlink b <== a || b;
```

- Contrapositive + Memory

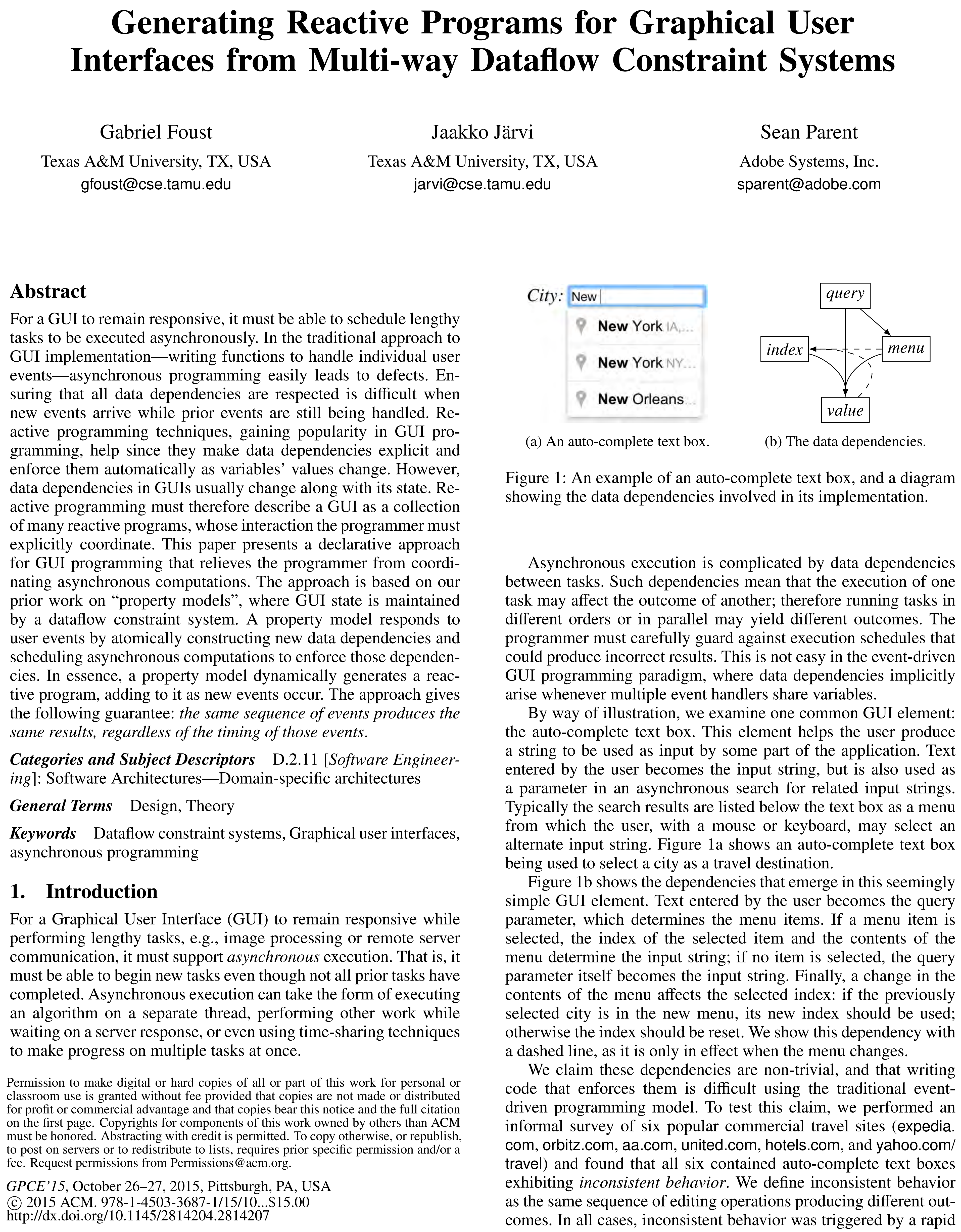
```
unlink a; unlink b;
relate {
  b <== a || b;
  a <== b && a;
}
```

- Contrapositive

```
relate {
  b <== a || b;
  a <== b && a;
}
```

- Unconstrained + Disable Operation

```
invariant:
  valid <== !a || b;
```



Generating Reactive Programs for Graphical User Interfaces from Multi-way Dataflow Constraint Systems

Gabriel Foust
Texas A&M University, TX, USA
gfoust@cse.tamu.edu

Jaakko Järvi
Texas A&M University, TX, USA
jarvi@cse.tamu.edu

Sean Parent
Adobe Systems, Inc.
sparent@adobe.com

Abstract

For a GUI to remain responsive, it must be able to schedule lengthy tasks to be executed asynchronously. In the traditional approach to GUI implementation—writing functions to handle individual user events—asynchronous programming easily leads to defects. Ensuring that all data dependencies are respected is difficult when new events arrive while prior events are still being handled. Reactive programming techniques, gaining popularity in GUI programming, help since they make data dependencies explicit and enforce them automatically as variables' values change. However, data dependencies in GUIs usually change along with its state. Reactive programming must therefore describe a GUI as a collection of many reactive programs, whose interaction the programmer must explicitly coordinate. This paper presents a declarative approach for GUI programming that relieves the programmer from coordinating asynchronous computations. The approach is based on our prior work on "property models", where GUI state is maintained by a dataflow constraint system. A property model responds to user events by atomically constructing new data dependencies and scheduling asynchronous computations to enforce those dependencies. In essence, a property model dynamically generates a reactive program, adding to it as new events occur. The approach gives the following guarantee: *the same sequence of events produces the same results, regardless of the timing of those events*.

Categories and Subject Descriptors D.2.11 [*Software Engineering*]: Software Architectures—Domain-specific architectures

General Terms Design, Theory

Keywords Dataflow constraint systems, Graphical user interfaces, asynchronous programming

1. Introduction

For a Graphical User Interface (GUI) to remain responsive while performing lengthy tasks, e.g., image processing or remote server communication, it must support *asynchronous* execution. That is, it must be able to begin new tasks even though not all prior tasks have completed. Asynchronous execution can take the form of executing an algorithm on a separate thread, performing other work while waiting on a server response, or even using time-sharing techniques to make progress on multiple tasks at once.

(a) An auto-complete text box. (b) The data dependencies.

Figure 1: An example of an auto-complete text box, and a diagram showing the data dependencies involved in its implementation.

Asynchronous execution is complicated by data dependencies between tasks. Such dependencies mean that the execution of one task may affect the outcome of another; therefore running tasks in different orders or in parallel may yield different outcomes. The programmer must carefully guard against execution schedules that could produce incorrect results. This is not easy in the event-driven GUI programming paradigm, where data dependencies implicitly arise whenever multiple event handlers share variables.

By way of illustration, we examine one common GUI element: the auto-complete text box. This element helps the user produce a string to be used as input by some part of the application. Text entered by the user becomes the input string, but is also used as a parameter in an asynchronous search for related input strings. Typically the search results are listed below the text box as a menu from which the user, with a mouse or keyboard, may select an alternate input string. Figure 1a shows an auto-complete text box being used to select a city as a travel destination.

Figure 1b shows the dependencies that emerge in this seemingly simple GUI element. Text entered by the user becomes the query parameter, which determines the menu items. If a menu item is selected, the index of the selected item and the contents of the menu determine the input string; if no item is selected, the query parameter itself becomes the input string. Finally, a change in the contents of the menu affects the selected index: if the previously selected city is in the new menu, its new index should be used; otherwise the index should be reset. We show this dependency with a dashed line, as it is only in effect when the menu changes.

We claim these dependencies are non-trivial, and that writing code that enforces them is difficult using the traditional event-driven programming model. To test this claim, we performed an informal survey of six popular commercial travel sites (expedia.com, orbitz.com, aa.com, united.com, hotels.com, and yahoo.com/travel) and found that all six contained auto-complete text boxes exhibiting *inconsistent behavior*. We define inconsistent behavior as the same sequence of editing operations producing different outcomes. In all cases, inconsistent behavior was triggered by a rapid

GPCE'15, October 26–27, 2015, Pittsburgh, PA, USA
© 2015 ACM. 978-1-4503-3687-1/15/10...$15.00
http://dx.doi.org/10.1145/2814204.2814207

Helping Programmers Help Users

John Freeman
Texas A&M University
jfreeman@cse.tamu.edu

Jaakko Järvi
Texas A&M University
jarvi@cse.tamu.edu

Wonseok Kim
Texas A&M University
guruwons@cse.tamu.edu

Mat Marcus
Canyonlands Software Design
mmarcus@emarcus.org

Sean Parent
Adobe Systems, Inc.
sparent@adobe.com

Abstract

User interfaces exhibit a wide range of features that are designed to assist users. Interaction with one widget may trigger value changes, disabling, or other behaviors in other widgets. Such automatic behavior may be confusing or disruptive to users. Research literature on user interfaces offers a number of solutions, including interface features for explaining or controlling these behaviors. To help programmers help users, the implementation costs of these features need to be much lower. Ideally, they could be generated for "free." This paper shows how several help and control mechanisms can be implemented as algorithms and reused across interfaces, making the cost of their adoption negligible. Specifically, we describe generic help mechanisms for visualizing data flow and explaining command deactivation, and a mechanism for controlling the flow of data. A reusable implementation of these features is enabled by our property model framework, where the data manipulated through a user interface is modeled as a constraint system.

Categories and Subject Descriptors H.2.2 [*Software Engineering*]: Design Tools and Techniques—user interfaces

General Terms Algorithms

Keywords user interfaces, software reuse, constraint systems, software architecture

1. Introduction

The dull, run-of-the-mill user interfaces—dialogs, forms, and such—do not get much attention from the software research community, but they collectively require a lot of attention from the programmer community. User interfaces abound, and they are laborious to develop and difficult to get correct. As an attempt to reduce the cost of constructing user interfaces, we have introduced *property models*, a declarative approach to programming user interfaces [8, 9]. The long term goal of this work is to reach a point where most (maybe all) of the functionality that we have come to expect from a high quality user interface would come from reusable algorithms or components in a software library, parametrized by a specification of the data manipulated by the user interface. In particular, we have described reusable implementations for the propagation of values between user interface elements, the enablement and disablement of user interface widgets, and the activation and deactivation of widgets that launch commands.

This paper describes our work to direct these advances to the improvement of user interfaces. One purpose of a user interface is to provide the user with an easily interpreted view of a conceptual model for the internal states of the application and the interface itself. To the extent that the interface fails to do this, there exists a *gulf of evaluation* [7]. The gulf of evaluation exacerbates the cognitive effort required to understand and use an application, and can lead to user frustration.

This paper shows that with the power of components, genericity, and reuse we can go beyond merely implementing existing behavior more economically. If a user interface behavior can be successfully packaged into a reusable component, then we should explore more functionality for assisting users and closing the gulf of evaluation. We should aim for more consistent user interfaces with less surprising behavior, more explanations of why a user interface behaves the way it does, and more abilities to change the behavior of a user interface "on the fly" to better serve users' goals. In sum, we should aim for more features that *help* users in their interactions with an interface.

This paper describes several generic realizations of help and convenience features that could be provided as standard features of dialogs and forms. In particular, we focus on (1) visualizing how data flows in a user interface, (2) providing help messages for commands that are deactivated, and (3) providing the user with means to control the direction of the flow of data. We emphasize that the main contributions of the paper are the algorithms and the software architecture that enable implementing these features in a reusable manner, applicable to a large class of user interfaces with negligible programming effort. The realizations of these algorithms build on the property models approach, in which the data that a user interface manipulates and the dependencies within this data are modeled explicitly as a constraint system. Reusable user interface algorithms are thus algorithms that inspect and manipulate this constraint system.

We are at an early stage in our effort. To not overstate our contribution, we note that we have not conducted user studies, and we have not applied the proposed tools and algorithms to a large collection of user interfaces drawn from existing software. The computer-human interaction (CHI) research community, however, has devised many help and support features for user interfaces and

GPCE'11, October 22–23, 2011, Portland, Oregon, USA.

Algorithms for User Interfaces

Jaakko Järvi
Texas A&M University
jarvi@cse.tamu.edu

Mat Marcus
mmarcus@emarcus.org

Sean Parent
Adobe Systems Inc.
sparent@adobe.com

John Freeman
Texas A&M University
jfreeman@cse.tamu.edu

Jacob Smith
Texas A&M University
jnsmith@cse.tamu.edu

Abstract

User interfaces for modern applications must support a rich set of interactive features. It is commonplace to find applications with dependencies between values manipulated by user interface elements, conditionally enabled controls, and script record-ability and playback against different documents. A significant fraction of the application programming effort is devoted to implementing such functionality, and the resulting code is typically not reusable.

This paper extends our "property models" approach to programming user interfaces. Property models allow a large part of the functionality of a user interface to be implemented in reusable libraries, reducing application specific code to a set of declarative rules. We describe how, as a by-product of computations that maintain the values of user interface elements, property models obtain accurate information of the currently active dependencies among those elements. This information enables further expanding the class of user interface functionality that we can encode as generic algorithms. In particular, we describe automating the decisions for the enablement of user interface widgets and activation of command widgets. Failing to disable or deactivate widgets correctly is a common source of user-interface defects, which our approach largely removes.

We report on the increased reuse, reduced defect rates, and improved user interface design turnarounds in a commercial software development effort as a result of adopting our approach.

Categories and Subject Descriptors D.2.2 [*Design Tools and Techniques*]: User interfaces; D.2.13 [*Reusable Software*]: Reuse models

General Terms Algorithms, Design

Keywords Software reuse, Component software, User interfaces, Declarative specifications, Constraint systems

1. Introduction

The role of a user interface, such as a dialog window, can be summarized as supporting the user in selecting valid values for a command or function to be executed in a program. In modern applications this support may mean, for example, computing values of some user interface elements automatically when values of other elements change, storing and retrieving default values, capturing user actions into a replayable script, undo and redo functionality, disabling user interface elements when their values are irrelevant for a final result, etc. This list is long—it is no small task for programmers to implement high-quality user interfaces.

In the prevailing approach to programming graphical user interfaces (GUIs), one of many GUI frameworks [6, 17, 31] provides a selection of widgets as reusable software components, and the programmer implements a user interface as a composition of widgets by specifying the interactions between the components. The interactions are typically expressed using imperative object-oriented code placed in event handlers. Even in user interfaces with relatively simple functionality, interactions between components are often surprisingly complex. Consequently, the event-handling logic that expresses the interactions is similarly complex, often scattered to many locations in the program, and seldom reusable across user interfaces. It is thus not surprising that user interface code can account for 30–50% of applications' code [21, 25], and a disproportionately higher share of the reported defects [25].

We have previously introduced *property models* [13], an approach to explicitly model many commonalities in the behavior of a class of typical user interfaces, such as dialog windows. We showed how an algorithm for computing new values of user interface elements after changing values of other elements and an algorithm for script recording and playback can be reused across user interfaces. These algorithms are generic, parametrized by a (declaratively specified) model that represents the variables manipulated by a user interface and the functional dependencies between those variables. We suggested that where property models can be applied, the amount of code is notably reduced and software quality improves compared to using a traditional GUI framework.

This paper develops the property models approach further. We focus on how to obtain, alongside computing updated values for user interface elements, accurate information of which functional dependencies between user interface elements were active in computing those values. Besides showing how to compute this information we explain how it enables further user interface functionality to be encoded as reusable algorithms. In particular, we show how this information gives the means for algorithms for the enablement and disablement of widgets when they are not relevant to the result of a user interface, and activation and de-activation of command widgets when the current result of a user interface does not satisfy stated conditions.

GPCE'09, October 4–5, 2009, Denver, Colorado, USA.

Property Models

From Incidental Algorithms to Reusable Components

Jaakko Järvi
Texas A&M University
College Station, TX, U.S.A
jarvi@cs.tamu.edu

Mat Marcus
Adobe Systems, Inc.
Seattle, WA, U.S.A
mmarcus@adobe.com

Sean Parent
Adobe Systems, Inc.
San Jose, CA, U.S.A
sparent@adobe.com

John Freeman
Texas A&M University
College Station, TX, U.S.A
jfreeman@cs.tamu.edu

Jacob N. Smith
Texas A&M University
College Station, TX, U.S.A
jnsmith@cs.tamu.edu

Abstract

A user interface, such as a dialog, assists a user in synthesising a set of values, typically parameters for a command object. Code for "command parameter synthesis" is usually application-specific and non-reusable, consisting of validation logic in event handlers and code that controls how values of user interface elements change in response to a user's actions, etc. These software artifacts are *incidental*—they are not explicitly designed and their implementation emerges from a composition of locally defined behaviors.

This article presents *property models* to capture explicitly the algorithms, validation, and interaction rules, arising from command parameter synthesis. A user interface's behavior can be derived from a declarative property model specification, with the assistance of a component akin to a constraint solver. This allows multiple interfaces, both human and programmatic, to reuse a single model along with associated validation logic and widget activation logic.

The proposed technology is deployed in large commercial software application suites. Where we have applied property models, we have measured significant reductions in source-code size with equivalent or increased functionality; additional levels of reuse are apparent, both within single applications, and across product lines; and applications are able to provide more uniform access to functionality. There is potential for wide adoption: by our measurements command parameter synthesis comprises roughly one third of the code and notably more of the defects in desktop applications.

Categories and Subject Descriptors D.2.13 [*Reusable Software*]: Reuse models; D.2.2 [*Design Tools and Techniques*]: User interfaces

General Terms Algorithms, Design

Keywords Software reuse, Component software, User interfaces, Declarative specifications, Constraint systems

1. Introduction

Software systems utilizing reusable components tend to be more robust and less costly than their hand-crafted counterparts (Basili et al. 1996; Frakes and Succi 2001; Nazareth and Rothenberger 2004). Indeed, the software industry has been successful in capturing often needed functionality into reusable generic components, witnessed by the wide availability of software libraries in all mainstream programming languages and the ubiquitous use of components from those libraries. There are, however, domains commonly encountered in mainstream day-to-day programming in which reuse remains modest—and in which the industry continues to struggle with low quality, high defect rates, and low productivity.

As the scale of software increases, software development relies more on reusable components—at the same time, there is an increase in the amount of code that composes and relates components. Often such code is not explicitly designed, and it is rarely reusable. In larger collections of components, networks of relationships between components arise. We refer to such networks as *incidental data structures*—data structures that emerge out of compositions of components and have neither an explicit encoding in the program nor an explicit run-time representation accessible to the rest of the program. Consequently, such data structures cannot be operated on by generic, reusable algorithms. Instead, they are manipulated with *incidental algorithms*, similarly emerging from the combined behavior of locally defined actions, and with no explicit encoding in the program. We believe that a large reuse potential exists within incidental algorithms and data structures.

In this paper we describe some of the architectural challenges in creating reusable libraries for rich user interfaces. We identify the communication and relationships between different elements of user interfaces as an architectural domain where incidental data structures and algorithms are prevalent; we refer to this domain as *command parameter synthesis*. We demonstrate a dramatic increase in re-usability of user interface code if the incidental structures of command parameter synthesis are modeled explicitly. To represent these explicit models, we present a new implementation mechanism, *property models*.

Command parameter synthesis assists a client in selecting and validating parameters for some command to be executed in the program. This is a common task in interactive applications—or in any application with a non-trivial, human or programmatic, interface. Typical examples of user interfaces requiring command parame-

© ACM, 2008. This is the author's version of the work. It is posted here by permission of ACM for your personal use. Not for redistribution. The definitive version was published in the Proceedings of the 7th International Conference on Generative Programming and Component Engineering (Nashville, Tennessee, October 19–20, 2008). GPCE '08 `http://doi.acm.org/10.1145/1449913.1449927`

GPCE'08, October 19–23, 2008, Nashville, Tennessee, USA.

Closing

- Good code is necessary, but not sufficient, for building a good UI.
- There is significant work in the area of data structures, algorithms to support good UI
  - And significant work remaining

References

- http://sean-parent.stlab.cc/papers-and-presentations

MAKE IT AN EXPERIENCE