Better Code: Relationships

Sean Parent | Senior Principal Scientist, Photoshop

#AdobeRemix
Hiroyuki-Mitsume Takahashi

Goal: No Contradictions

“A novice sees only the chessmen.
An amateur sees the board.
A master sees the game.”

– Unknown

Adobe

"Computer scientists are bad at relationships."

– Me

The Pieces

Relationships

#AdobeRemix
Hiroyuki-Mitsume Takahashi

Relations in Math

- A *relation* is a set of ordered pairs mapping entities from a *domain* to a *range*

- Distinct from a *function* in that the first *entity* does not uniquely determine the second
- A *relationship* is the way two entities are connected

$$\{(x_0, y_0), (x_1, y_1), (x_2, y_2), ...\}$$

Adobe

Predicates

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

Predicates

- A relation implies a *corresponding predicate* that tests if a pair exists in the relation
  - If it is true, the relationship is *satisfied* or *holds*

Predicates

- A relation implies a *corresponding predicate* that tests if a pair exists in the relation
  - If it is true, the relationship is *satisfied* or *holds*

- John is married to Jane

Predicates

- A relation implies a *corresponding predicate* that tests if a pair exists in the relation
  - If it is true, the relationship is *satisfied* or *holds*

- John is married to Jane
- Is John married to Jane?

Constraints

- A *constraint* is a relationship which *must* be satisfied

Constraints

- A *constraint* is a relationship which *must* be satisfied
  - For another relationship to be satisfied

Constraints

- A *constraint* is a relationship which *must* be satisfied
  - For another relationship to be satisfied

- The denominator must not be 0 for the result of division to be defined

Implication

$$a \Rightarrow b$$

(a implies b)

Implication

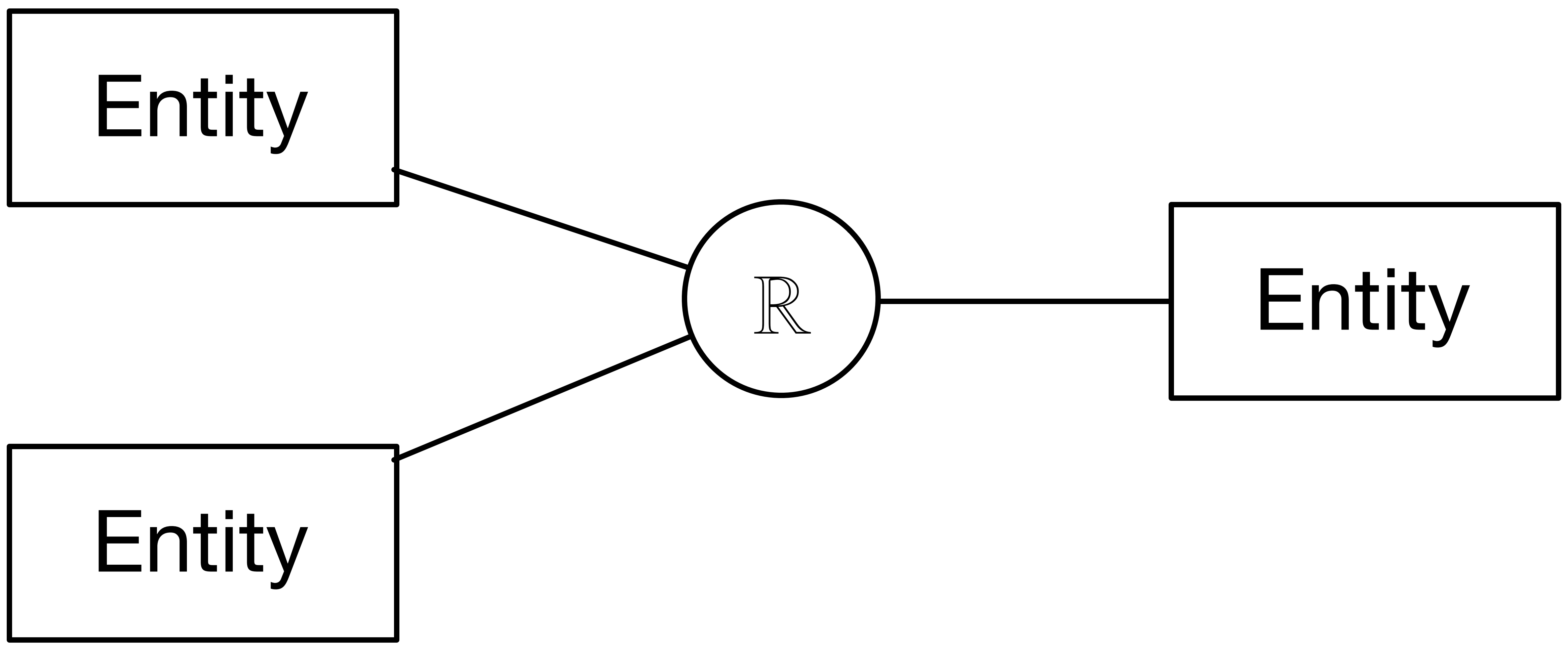
$$a \Rightarrow b$$

$(a \text{ implies } b)$

a	b	a ⇒ b
0	0	1
0	1	1
1	0	0
1	1	1

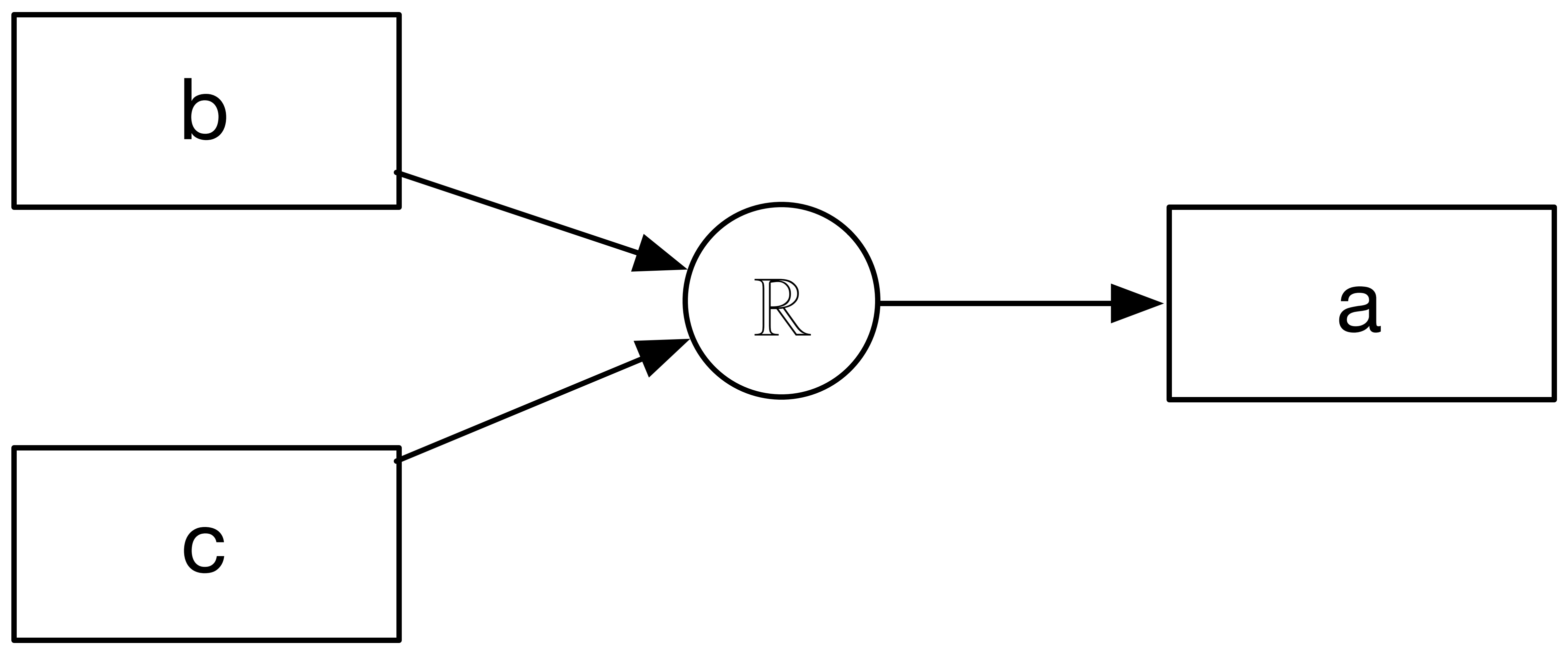
A simple, but incomplete, notation

- Entities are represented with a rectangle, and relationships with a circle
- This forms a *bipartite* graph



A simple notation

- Implication is represented with directional edges



- This is shorthand for *given entities b and c, a is any entity such that R holds*
- Read as, *b and c imply a*

Relationships and Objects

Relationships and Objects

- As soon as we have two entities we have implicit relationships

Relationships and Objects

- As soon as we have two entities we have implicit relationships
  - A memory space is an entity

Relationships and Objects

- As soon as we have two entities we have implicit relationships
  - A memory space is an entity
- When an object is copied or moved, any relationship that object was involved in is either *maintained* or *severed* with respect to the destination object

Relationships and Objects

- As soon as we have two entities we have implicit relationships
  - A memory space is an entity
- When an object is copied or moved, any relationship that object was involved in is either *maintained* or *severed* with respect to the destination object
- When an object is destructed, any relationship that object was involved in is *severed*

Witnessed Relationships

Witnessed Relationships

- A *witnessed* relationship is a relationship represented by an object

Witnessed Relationships

- A *witnessed* relationship is a relationship represented by an object
  - As an object, a witnessed relationship is copyable and equality comparable

Witnessed Relationships

- A *witnessed* relationship is a relationship represented by an object
  - As an object, a witnessed relationship is copyable and equality comparable
- When an object is copied or moved, any witnessed relationship that object was involved in is either maintained, severed, or *invalidated* with respect to the destination object

Witnessed Relationships

- A *witnessed* relationship is a relationship represented by an object
  - As an object, a witnessed relationship is copyable and equality comparable
- When an object is copied or moved, any witnessed relationship that object was involved in is either maintained, severed, or *invalidated* with respect to the destination object
  - This includes copying or moving the object witnessing the relationship

Witnessed Relationships

- A *witnessed* relationship is a relationship represented by an object
  - As an object, a witnessed relationship is copyable and equality comparable
- When an object is copied or moved, any witnessed relationship that object was involved in is either maintained, severed, or *invalidated* with respect to the destination object
  - This includes copying or moving the object witnessing the relationship
- When an object is destructed, any witnessed relationship that object was involved in is either severed, or *invalidated*.

Witnessed Relationships

- A *witnessed* relationship is a relationship represented by an object
  - As an object, a witnessed relationship is copyable and equality comparable
- When an object is copied or moved, any witnessed relationship that object was involved in is either maintained, severed, or *invalidated* with respect to the destination object
  - This includes copying or moving the object witnessing the relationship
- When an object is destructed, any witnessed relationship that object was involved in is either severed, or *invalidated*.

  - We may choose not to implement copy or move for witnessed relationships

Witnessed Relationships

- A *witnessed* relationship is a relationship represented by an object
  - As an object, a witnessed relationship is copyable and equality comparable
- When an object is copied or moved, any witnessed relationship that object was involved in is either maintained, severed, or *invalidated* with respect to the destination object
  - This includes copying or moving the object witnessing the relationship
- When an object is destructed, any witnessed relationship that object was involved in is either severed, or *invalidated.*

  - We may choose not to implement copy or move for witnessed relationships
  - This is how we get iterator invalidation "at a distance"

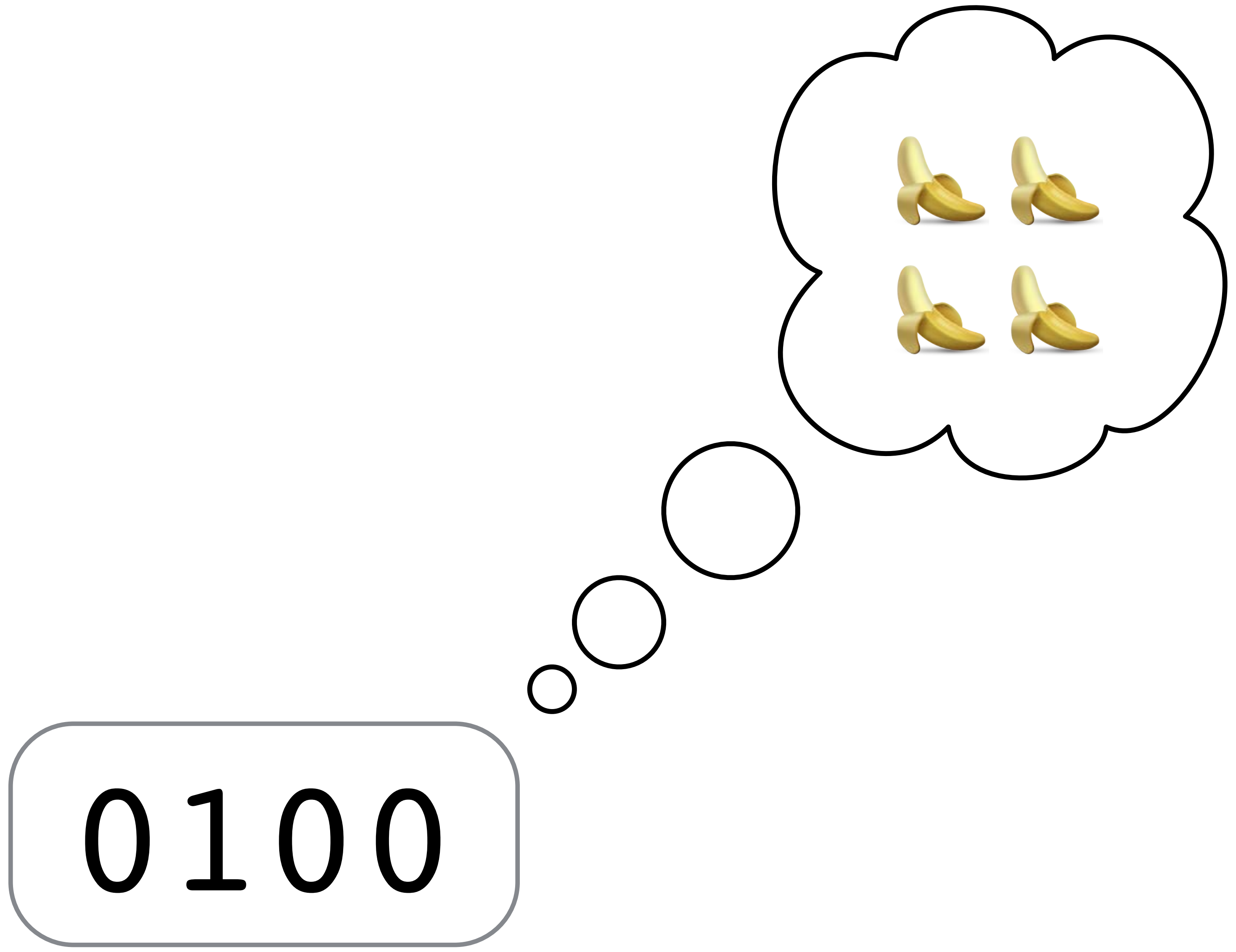
The Board

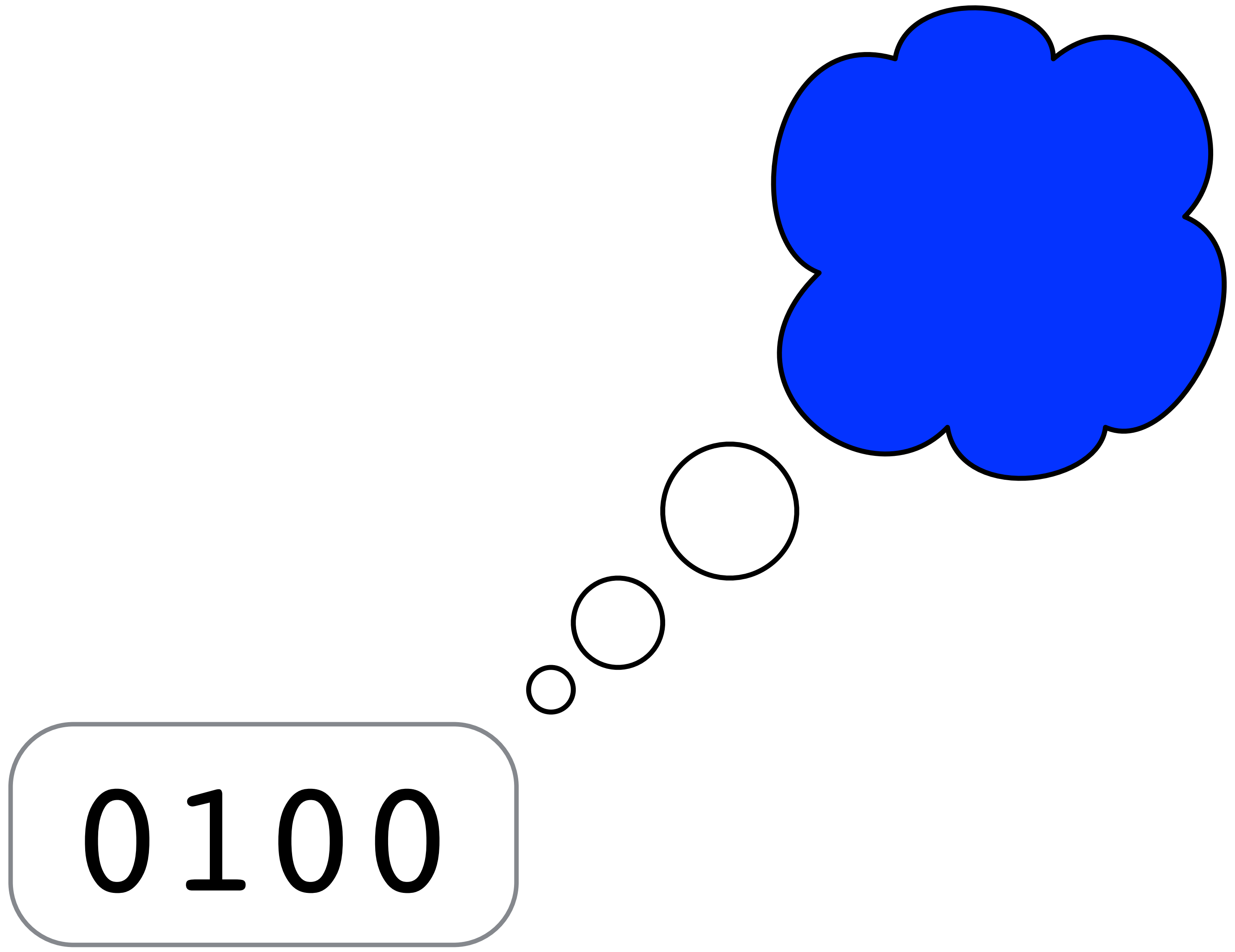
Structures

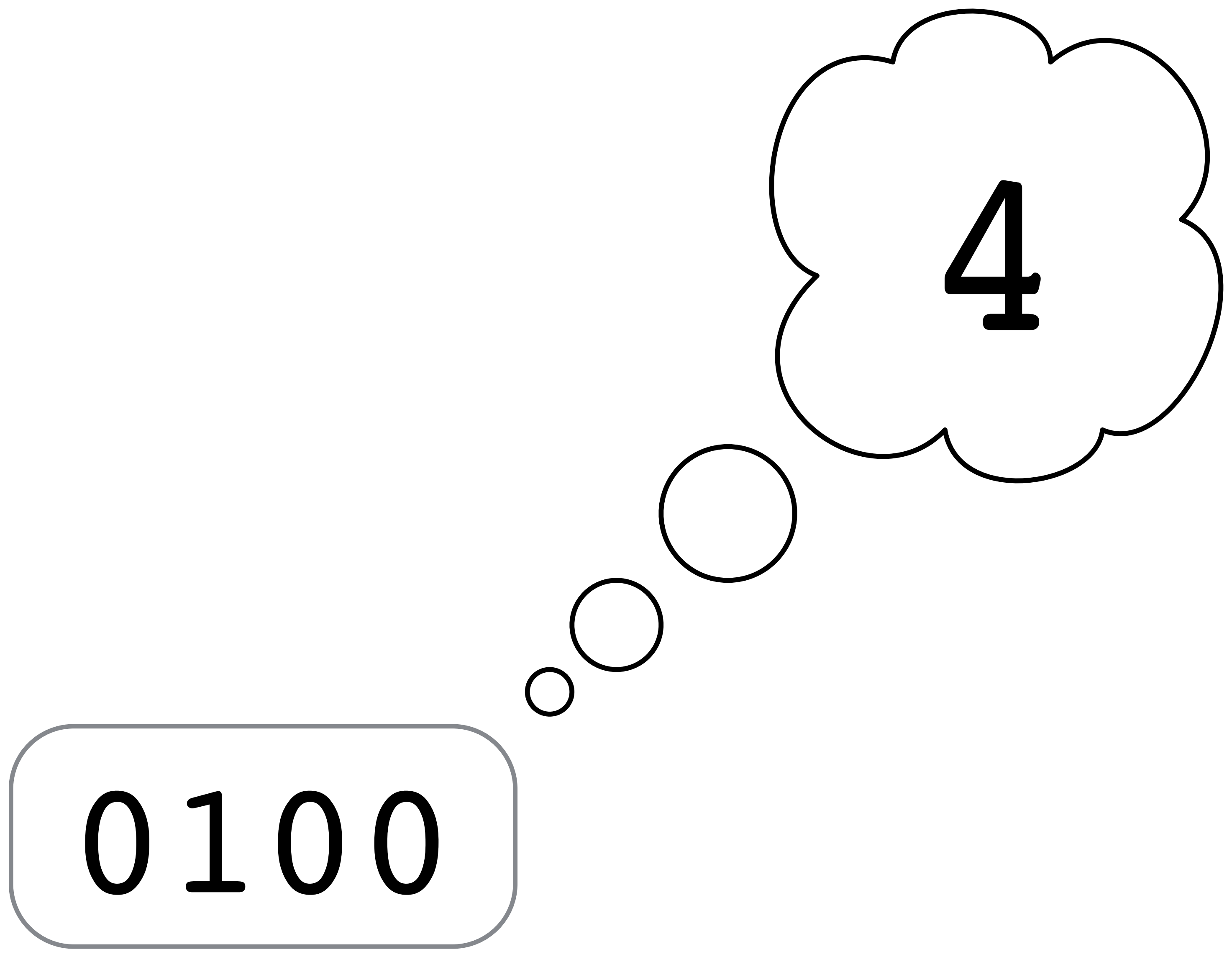
#AdobeRemix
Hiroyuki-Mitsume Takahashi

A *structure* on a set consists of additional entities that, in some manner, relate to the set, endowing the collection with meaning or significance.

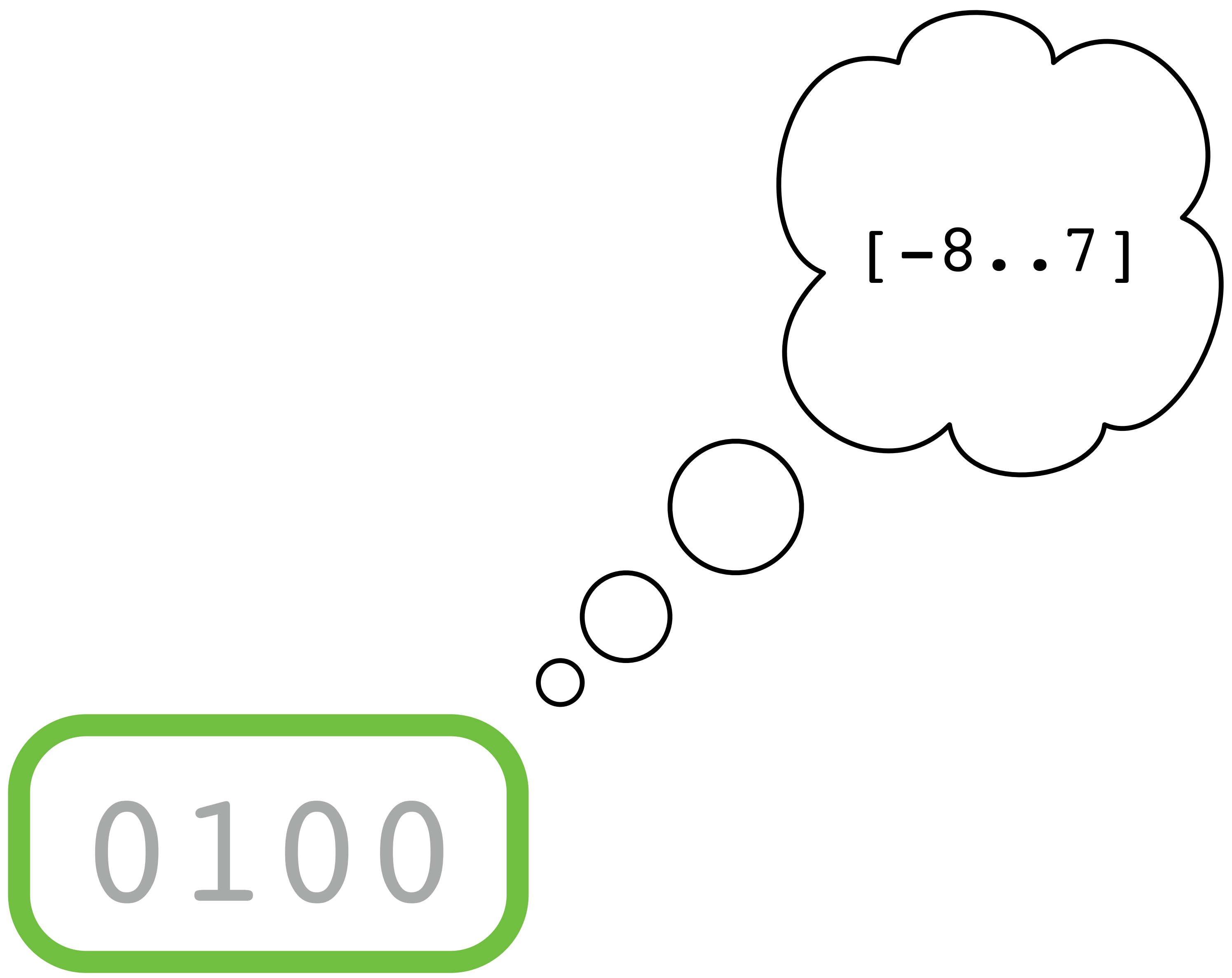
0100

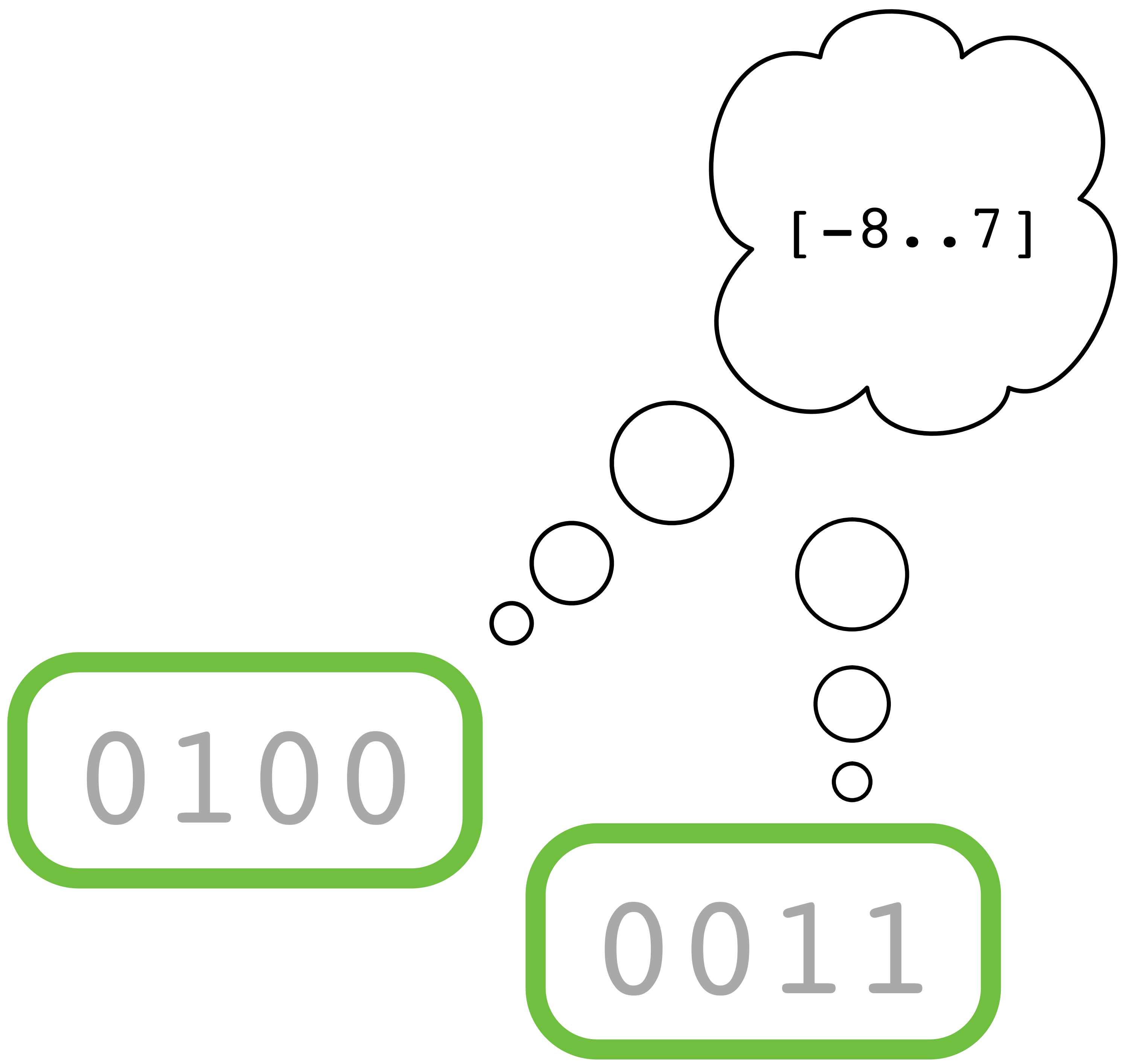

0100

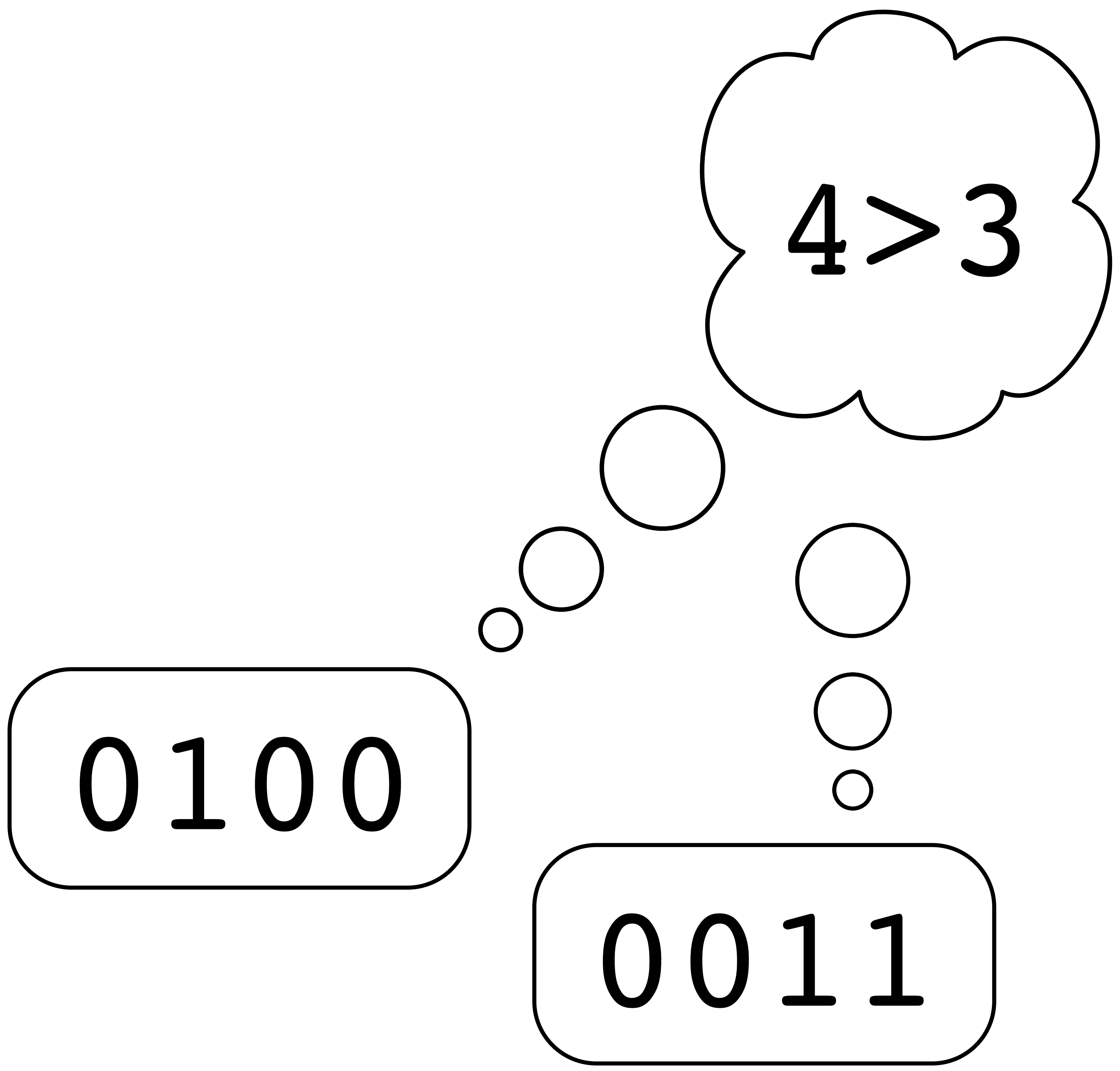

0100


0100
4

0100


[-8..7]
0100

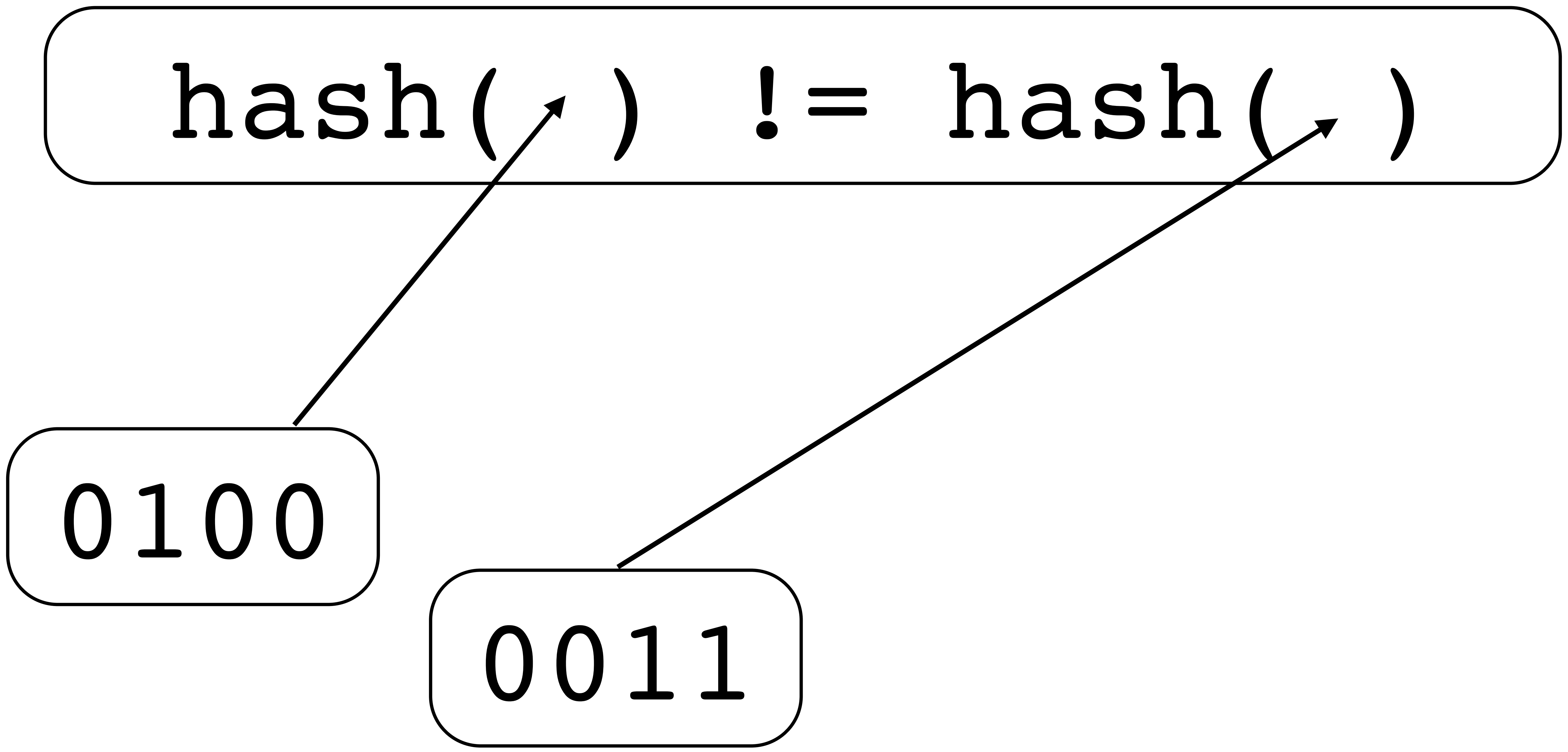

[-8..7]
0100
0011


4>3
0100
0011

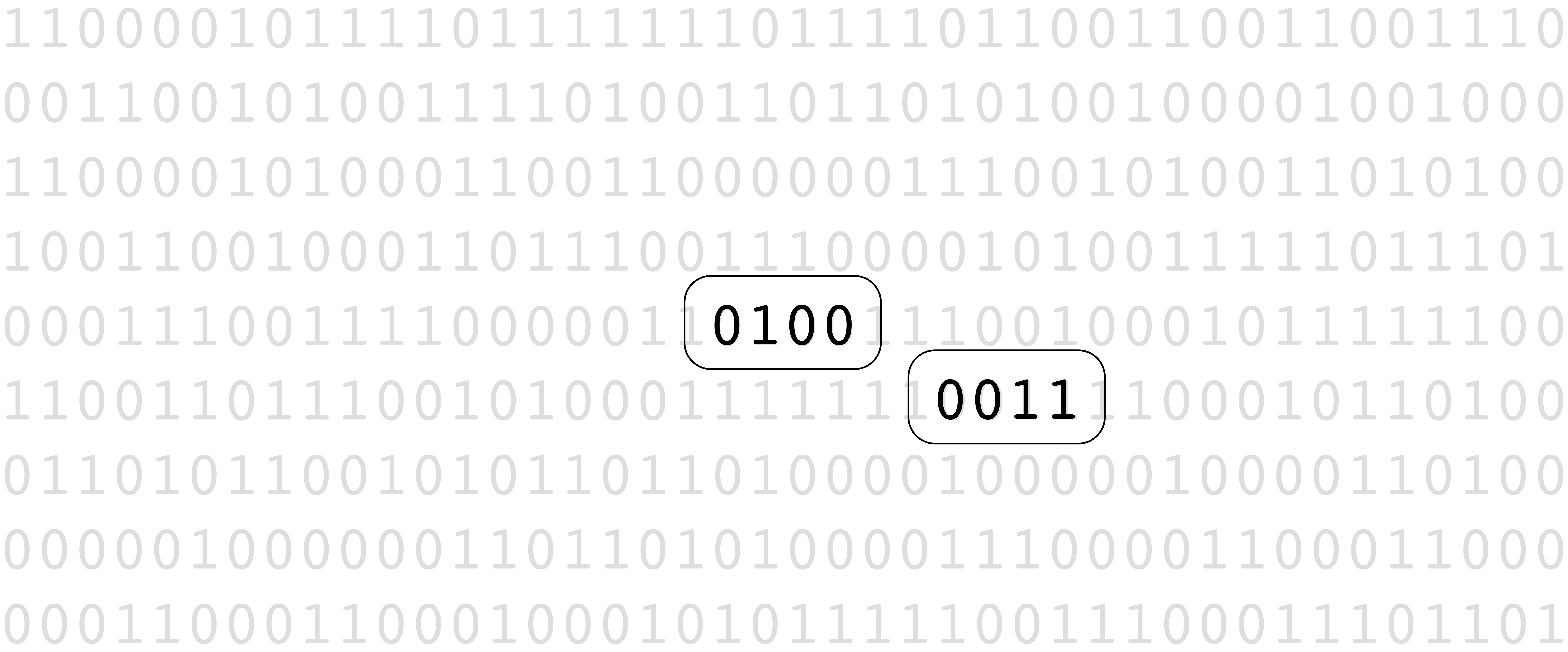
hash( ) != hash( )

0100

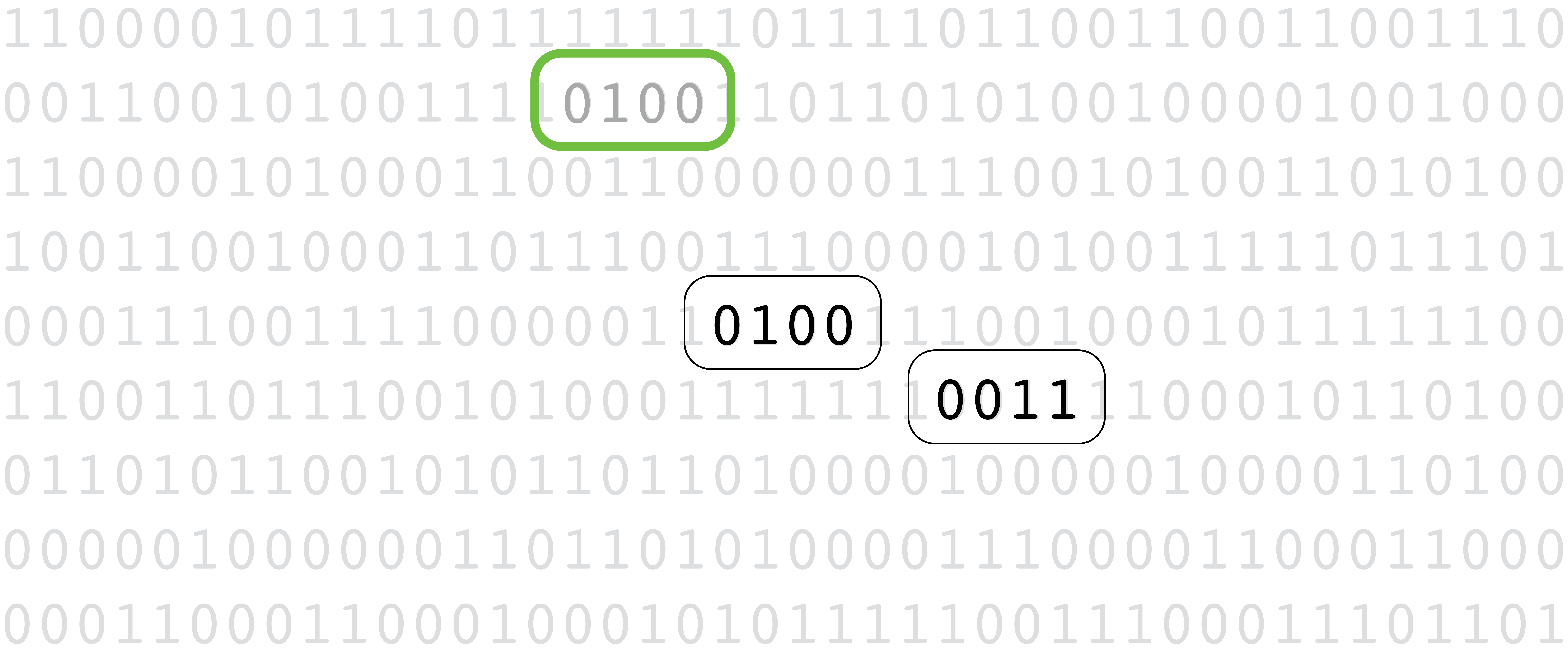
0011


hash( ) != hash( )
0100
0011

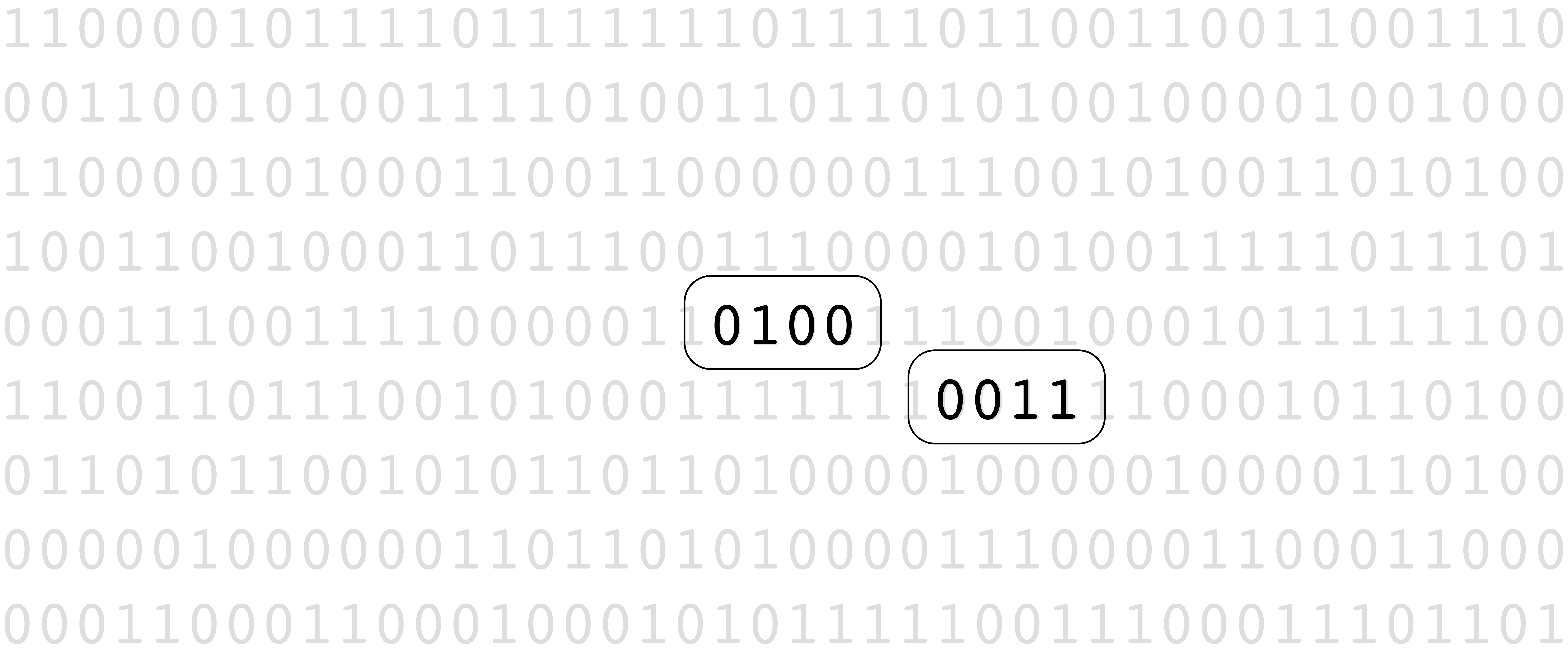
Memory Space



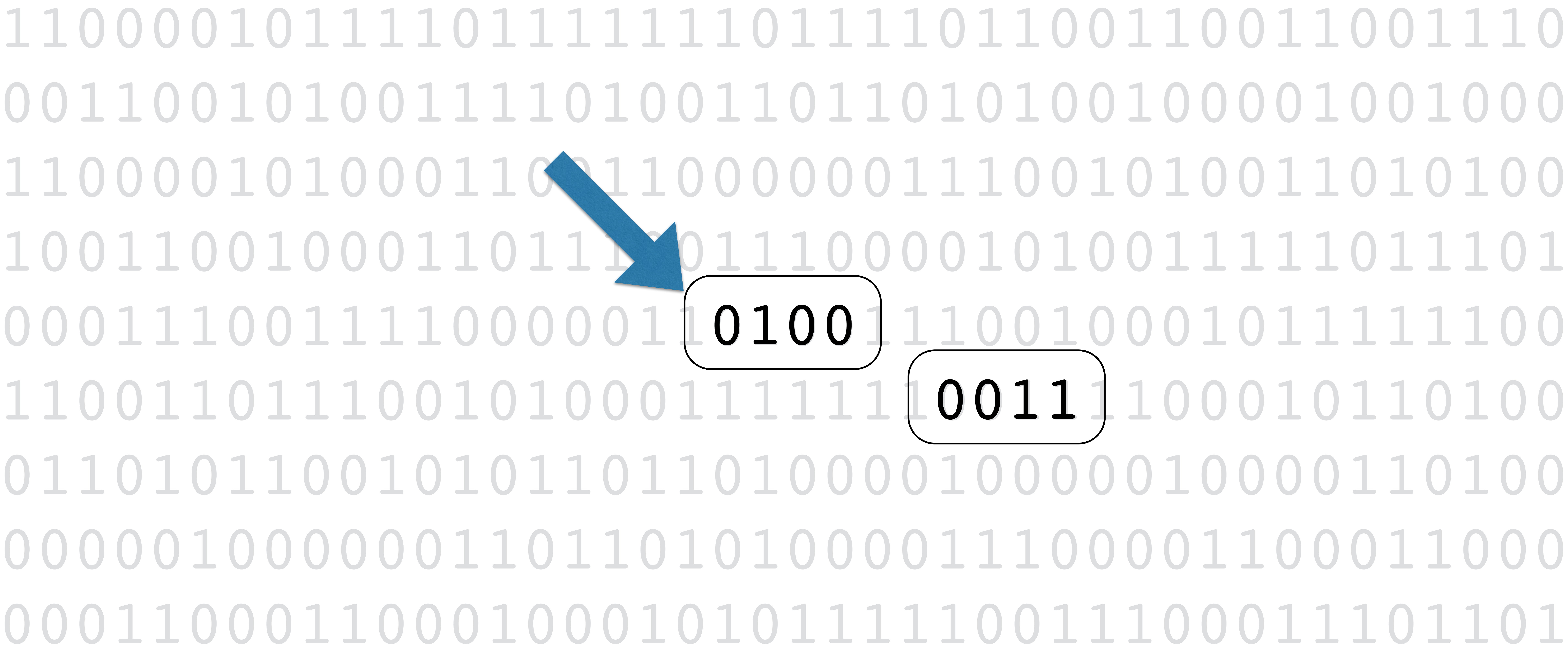
Memory Space



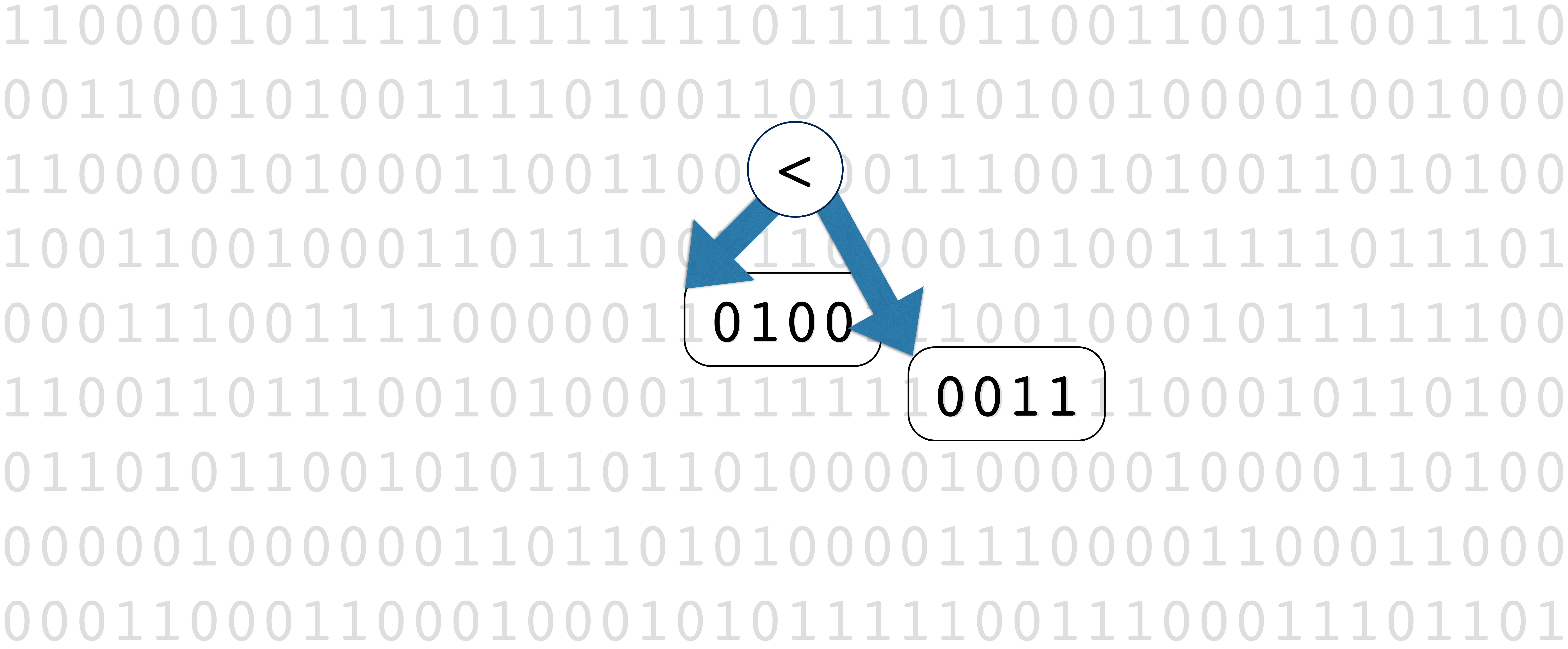
Memory Space



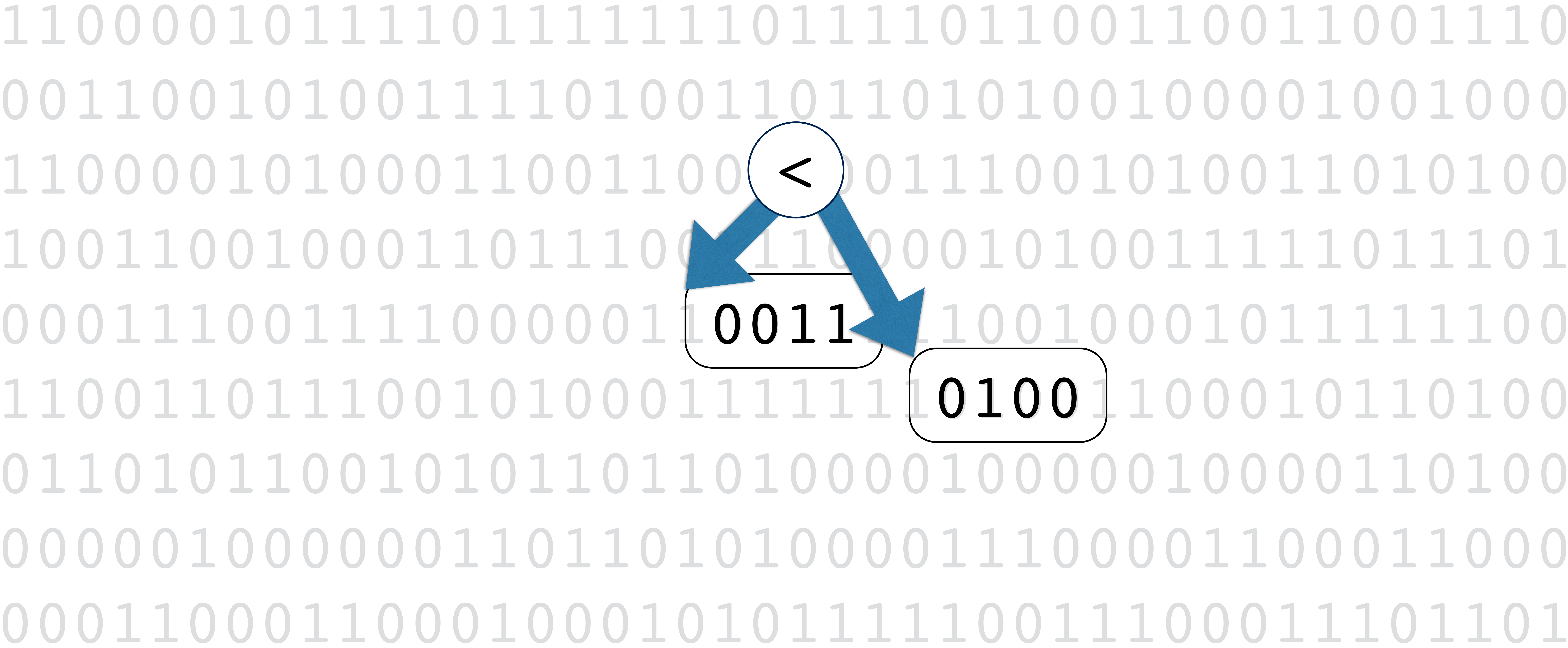
Memory Space



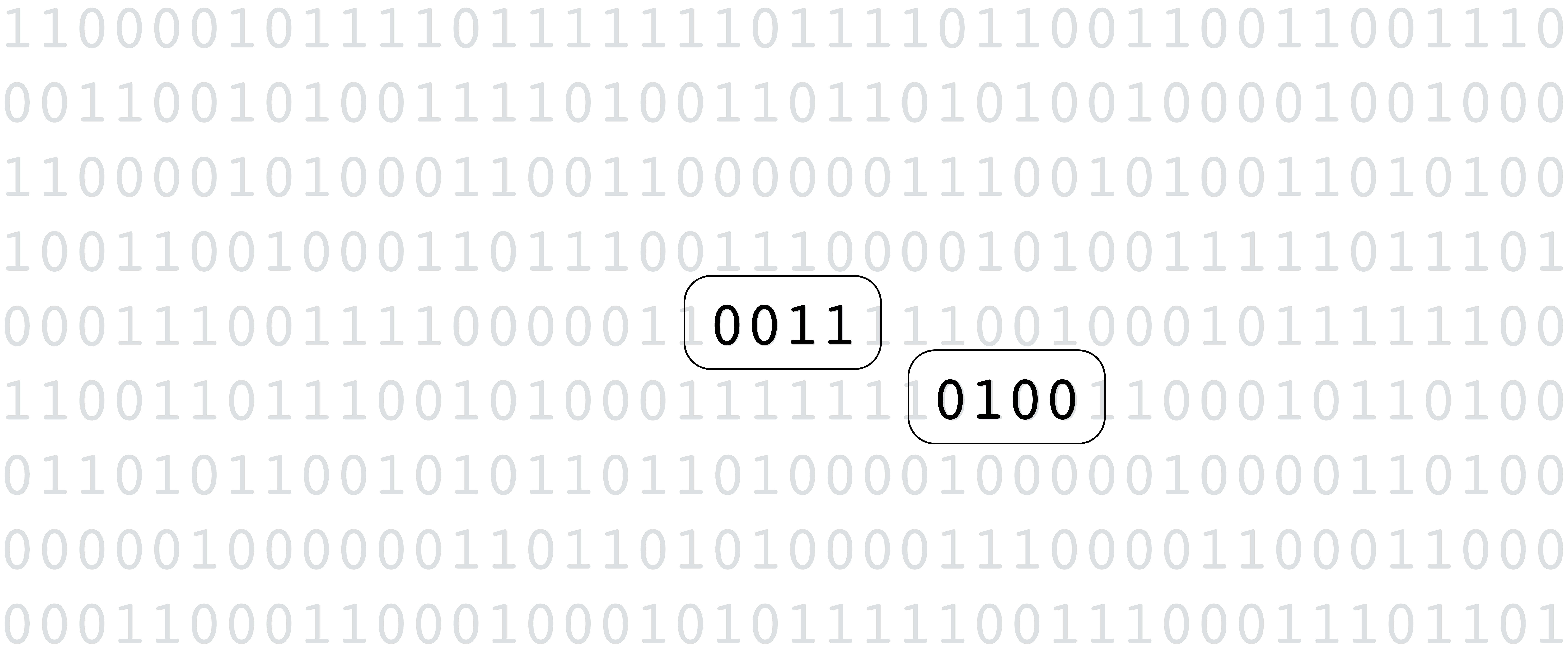
Memory Space



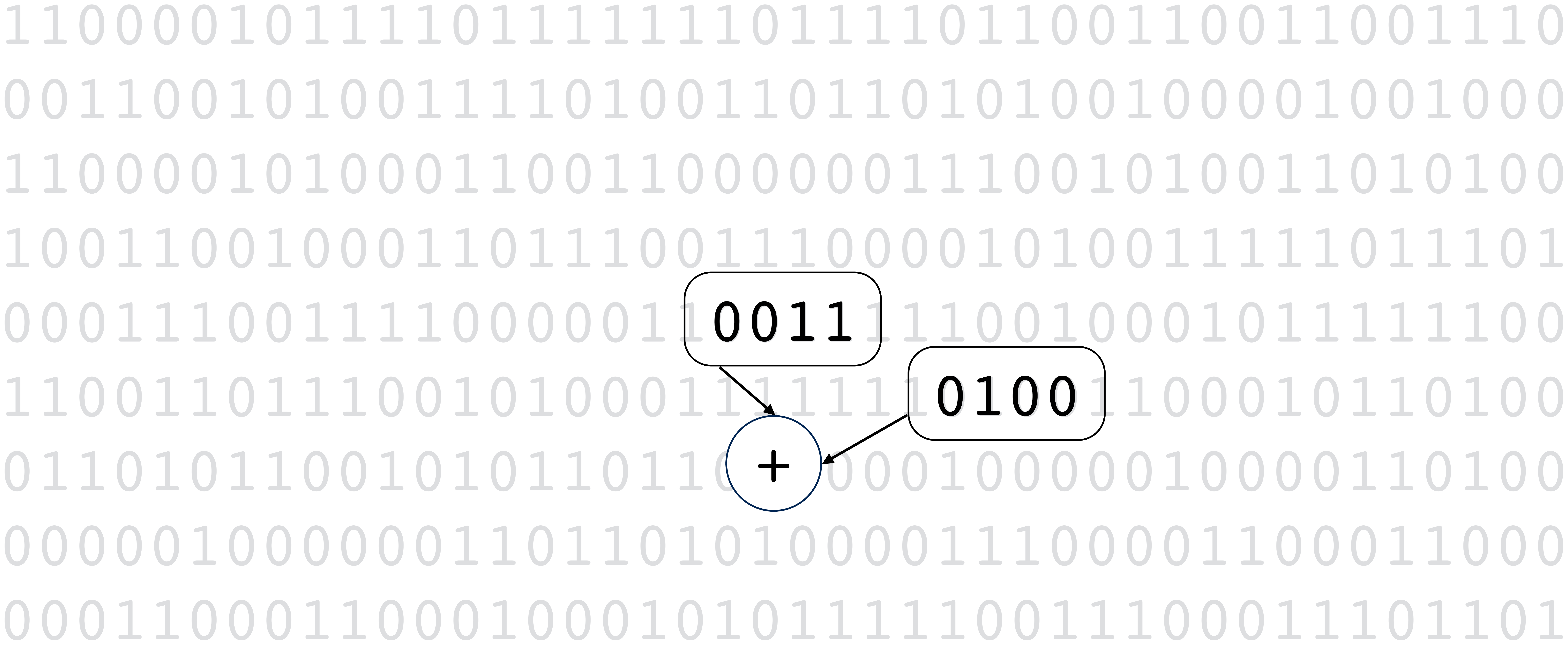
Memory Space



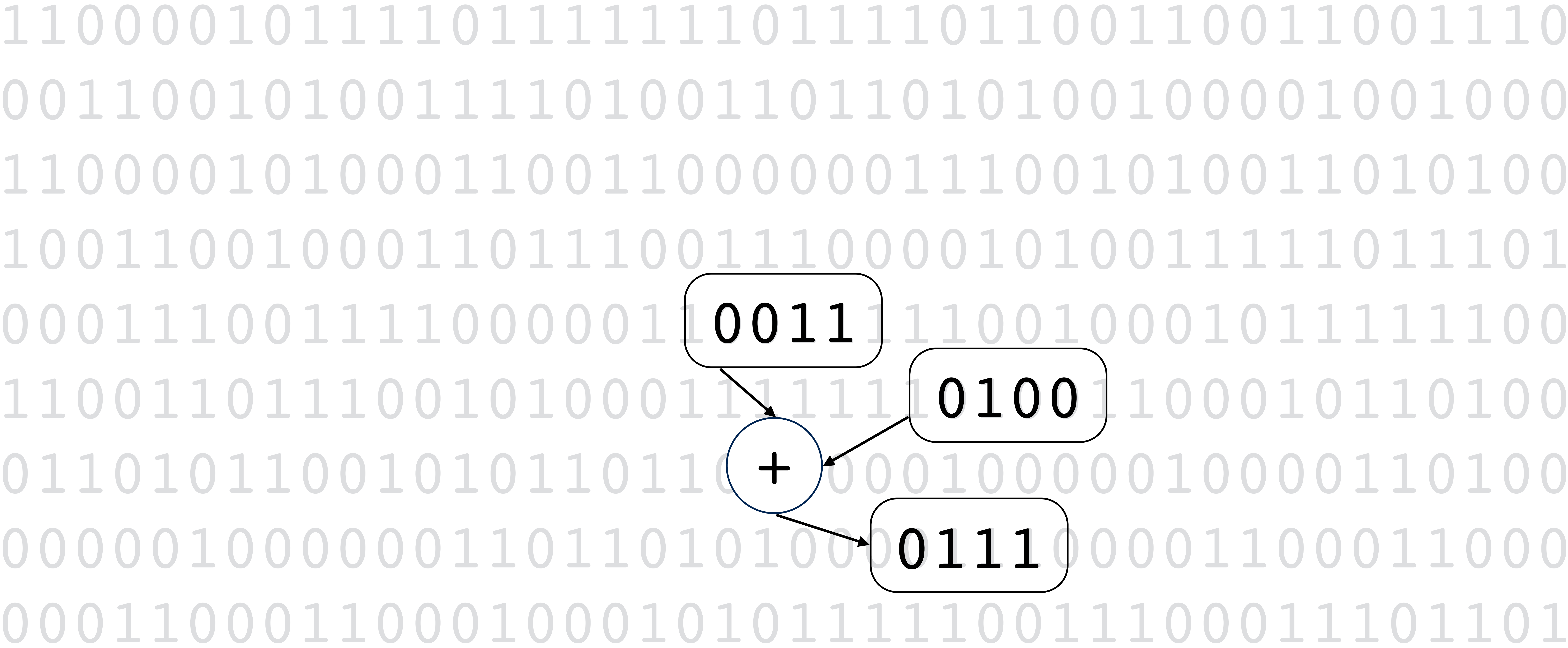
Memory Space



Memory Space



Memory Space



Safety

- An object instance, without meaning, is *invalid*

Safety

- An object instance, without meaning, is *invalid*
  - An object in an invalid state, must either be restored to a valid state, or destroyed

Safety

- An object instance, without meaning, is *invalid*
  - An object in an invalid state, must either be restored to a valid state, or destroyed
  - This is related to the idea of a *partially formed* object

Safety

- An object instance, without meaning, is *invalid*
  - An object in an invalid state, must either be restored to a valid state, or destroyed
  - This is related to the idea of a *partially formed* object
- An operation which leaves an object in an invalid state is *unsafe*

Safety

- An object instance, without meaning, is *invalid*
  - An object in an invalid state, must either be restored to a valid state, or destroyed
  - This is related to the idea of a *partially formed* object
- An operation which leaves an object in an invalid state is *unsafe*

- `std::move()` is an unsafe operation

Adobe

C++20

C++20

- Two new features specifically about relationships

C++20

- Two new features specifically about relationships
  - Concepts

C++20

- Two new features specifically about relationships
  - Concepts
  - Contracts

C++20

- ~~Two~~One new feature~~s~~ specifically about relationships
  - Concepts
  - ~~Contracts~~

Fundamentals of Generic Programming

James C. Dehnert and Alexander Stepanov

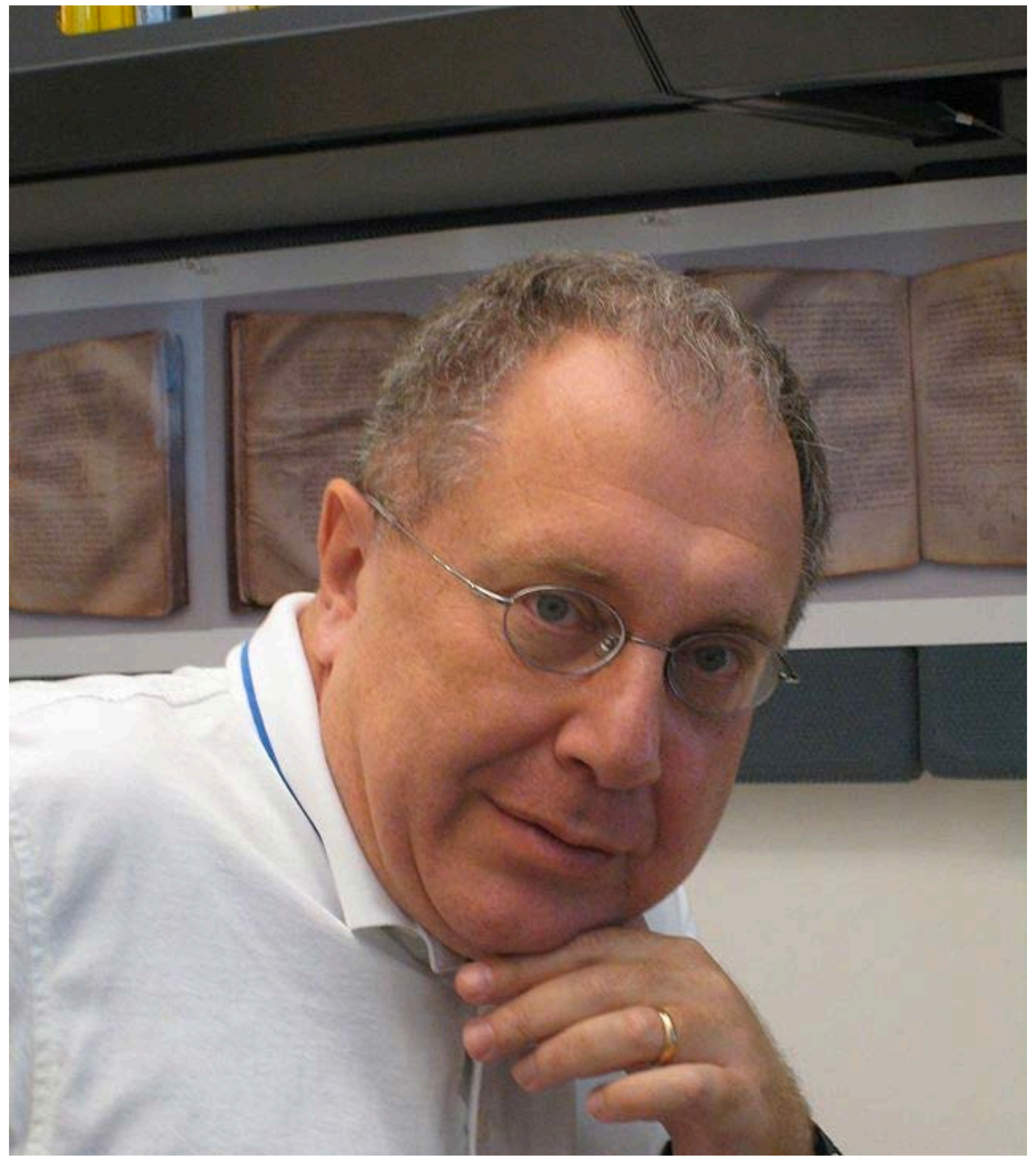
Silicon Graphics, Inc.
dehnertj@acm.org, stepanov@attlabs.att.com

Keywords: Generic programming, operator semantics, concept, regular type.

Abstract. Generic programming depends on the decomposition of programs into components which may be developed separately and combined arbitrarily, subject only to well-defined interfaces. Among the interfaces of interest, indeed the most pervasively and unconsciously used, are the fundamental operators common to all C++ built-in types, as extended to user-defined types, e.g. copy constructors, assignment, and equality. We investigate the relations which must hold among these operators to preserve consistency with their semantics for the built-in types and with the expectations of programmers. We can produce an axiomatization of these operators which yields the required consistency with built-in types, matches the intuitive expectations of programmers, and also reflects our underlying mathematical expectations.

1998

Adobe



Fundamentals of Generic Programming

James C. Dehnert and Alexander Stepanov

Silicon Graphics, Inc.
dehnertj@acm.org, stepanov@attlabs.att.com

1

1998

Adobe

"We call the set of axioms satisfied by a data type and a set of operations on it a *concept*."

“We call the set of axioms satisfied by a data type and a set of operations on it a ***concept***.”

An Axiomatic Basis for Computer Programming

C. A. R. Hoare
The Queen's University of Belfast, Northern Ireland*

In this paper an attempt is made to explore the logical foundations of computer programming by use of techniques which were first applied in the study of geometry and have later been extended to other branches of mathematics. This involves the elucidation of sets of axioms and rules of inference which can be used in proofs of the properties of computer programs. Examples are given of such axioms and rules, and a formal proof of a simple theorem is displayed. Finally, it is argued that important advantages, both theoretical and practical, may follow from a pursuance of these topics.

KEY WORDS AND PHRASES: axiomatic method, theory of programming' proofs of programs, formal language definition, programming language design, machine-independent programming, program documentation
CR CATEGORY: 4.0, 4.21, 4.22, 5.20, 5.21, 5.23, 5.24

1. Introduction

Computer programming is an exact science in that all the properties of a program and all the consequences of executing it in any given environment can, in principle, be found out from the text of the program itself by means of purely deductive reasoning. Deductive reasoning involves the application of valid rules of inference to sets of valid axioms. It is therefore desirable and interesting to elucidate the axioms and rules of inference which underlie our reasoning about computer programs. The exact choice of axioms will to some extent depend on the choice of programming language. For illustrative purposes, this paper is confined to a very simple language, which is effectively a subset of all current procedure-oriented languages.

2. Computer Arithmetic

The first requirement in valid reasoning about a program is to know the properties of the elementary operations which it invokes, for example, addition and multiplication of integers. Unfortunately, in several respects computer arithmetic is not the same as the arithmetic familiar to mathematicians, and it is necessary to exercise some care in selecting an appropriate set of axioms. For example, the axioms displayed in Table I are rather a small selection of axioms relevant to integers. From this incomplete set of axioms it is possible to deduce such simple theorems as:

$$x = x + y \times 0$$

$$y \leqslant r \supset r + y \times q = (r - y) + y \times (1 + q)$$

The proof of the second of these is:

A5	$(r - y) + y \times (1 + q)$
	$= (r - y) + (y \times 1 + y \times q)$
A9	$= (r - y) + (y + y \times q)$
A3	$= ((r - y) + y) + y \times q$
A6	$= r + y \times q$ provided $y \leqslant r$

The axioms A1 to A9 are, of course, true of the traditional infinite set of integers in mathematics. However, they are also true of the finite sets of "integers" which are manipulated by computers provided that they are confined to *nonnegative* numbers. Their truth is independent of the size of the set; furthermore, it is largely independent of the choice of technique applied in the event of "overflow"; for example:

(1) Strict interpretation: the result of an overflowing operation does not exist; when overflow occurs, the offending program never completes its operation. Note that in this case, the equalities of A1 to A9 are strict, in the sense that both sides exist or fail to exist together.

(2) Firm boundary: the result of an overflowing operation is taken as the maximum value represented.

(3) Modulo arithmetic: the result of an overflowing operation is computed modulo the size of the set of integers represented.

These three techniques are illustrated in Table II by addition and multiplication tables for a trivially small model in which 0, 1, 2, and 3 are the only integers represented.

It is interesting to note that the different systems satisfying axioms A1 to A9 may be rigorously distinguished from each other by choosing a particular one of a set of mutually exclusive supplementary axioms. For example, infinite arithmetic satisfies the axiom:

$A10_I$ $\neg \exists x \forall y \quad (y \leqslant x)$,

where all finite arithmetics satisfy:

$A10_F$ $\forall x \quad (x \leqslant \max)$

where "max" denotes the largest integer represented.

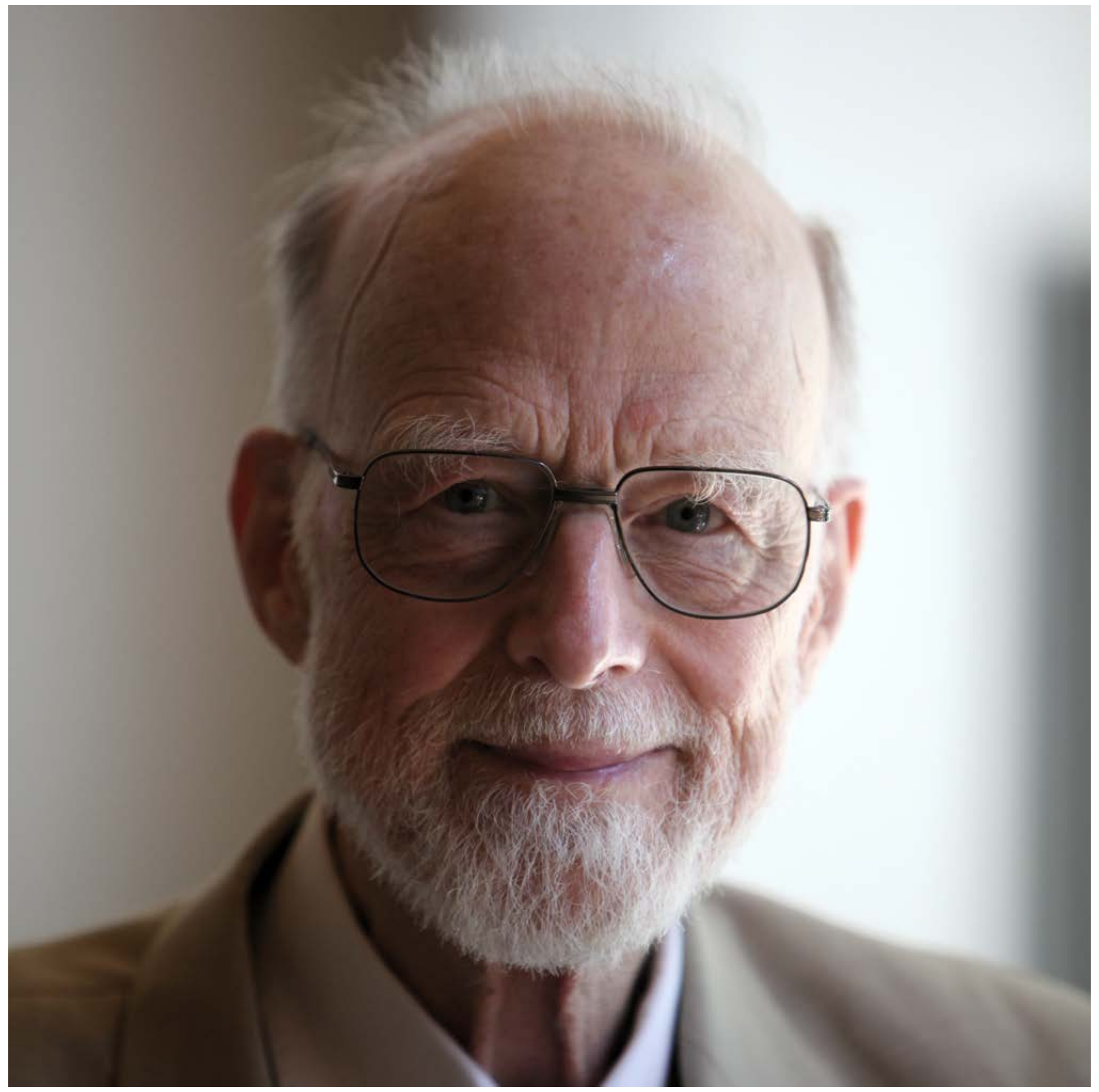
Similarly, the three treatments of overflow may be distinguished by a choice of one of the following axioms relating to the value of max + 1:

$A11_S$	$\neg \exists x \; (x = \max + 1)$	(strict interpretation)
$A11_B$	$\max + 1 = \max$	(firm boundary)
$A11_M$	$\max + 1 = 0$	(modulo arithmetic)

Having selected one of these axioms, it is possible to use it in deducing the properties of programs; however,

* Department of Computer Science

1969



An Axiomatic Basis for Computer Programming

C. A. R. Hoare
The Queen's University of Belfast, Northern Ireland*

of purely deductive reasoning. Deductive reasoning involves the application of valid rules of inference to sets of valid axioms. It is therefore desirable and interesting to elucidate the axioms and rules of inference which underlie our reasoning about computer programs. The exact choice of axioms will to some extent depend on the choice of programming language. For illustrative purposes, this paper is confined to a very simple language, which is effectively a subset of all current procedure-oriented languages.

2. Computer Arithmetic

The first requirement in valid reasoning about a program is to know the properties of the elementary operations which it invokes, for example, addition and multiplication of integers. Unfortunately, in several respects computer arithmetic is not the same as the arithmetic familiar to mathematicians, and it is necessary to exercise some care in selecting an appropriate set of axioms. For example, the axioms displayed in Table I are rather a small selection of axioms relevant to integers. From this incomplete set

* Department of Computer Science

It is interesting to note that the different systems satisfying axioms A1 to A9 may be rigorously distinguished from each other by choosing a particular one of a set of mutually exclusive supplementary axioms. For example, infinite arithmetic satisfies the axiom:

$A10_I \quad \neg\exists x \forall y \quad (y \leqslant x),$

where all finite arithmetics satisfy:

$A10_F \quad \forall x \quad (x \leqslant \max)$

where "max" denotes the largest integer represented.

Similarly, the three treatments of overflow may be distinguished by a choice of one of the following axioms relating to the value of max + 1:

$A11_S \quad \neg\exists x \quad (x = \max + 1)$ (strict interpretation)

$A11_B \quad \max + 1 = \max$ (firm boundary)

$A11_M \quad \max + 1 = 0$ (modulo arithmetic)

Having selected one of these axioms, it is possible to use it in deducing the properties of programs; however,

1969

Equality

- Two objects are equal iff their values correspond to the same entity

Equality

- Two objects are equal iff their values correspond to the same entity
- From this definition we can derive the following properties:

$$(\forall a) a = a. \qquad \text{(Reflexivity)}$$

$$(\forall a, b) a = b \Rightarrow b = a. \qquad \text{(Symmetry)}$$

$$(\forall a, b, c) a = b \land b = c \Rightarrow a = c. \qquad \text{(Transitivity)}$$

Concepts

- *Axioms* follow from the definition
- A collection of connected axioms form an *algebraic structure*
- Connected type requirements form a *concept*

Adobe

Copy and Assignment

- Properties of copy and assignment:

$$b \to a \Rightarrow a = b \qquad \text{(copies are equal)}$$

$$a = b = c \wedge d \neq a, d \to a \Rightarrow a \neq b \wedge b = c \qquad \text{(copies are disjoint)}$$

- Copy is connected to equality

Natural Total Order

- The natural total order is a total order that respects the other fundamental operations of the type
- A total order has the following properties:

$$(\forall a, b)\text{exactly one of the following holds:}$$

$$a < b, b < a, \text{or } a = b. \qquad \text{(Trichotomy)}$$

$$(\forall a, b, c) a < b \wedge b < c \Rightarrow a < c. \qquad \text{(Transitivity)}$$

Natural Total Order

- Example: Integer < is consistent with addition.

$$(\forall n \in \mathbb{Z}) n < (n + 1).$$

Concepts

- Quantified axioms are (generally) not actionable
  - Concepts in C++20 work by associating semantics with the name of an operation

Software is defined on Algebraic Structures

Adobe

Applying "Design by Contract"

Bertrand Meyer
Interactive Software Engineering

Reliability is even more important in object-oriented programming than elsewhere. This article shows how to reduce bugs by building software components on the basis of carefully designed contracts.

As object-oriented techniques steadily gain ground in the world of software development, users and prospective users of these techniques are clamoring more and more loudly for a "methodology" of object-oriented software construction — or at least for some methodological guidelines. This article presents such guidelines, whose main goal is to help improve the reliability of software systems. *Reliability* is here defined as the combination of correctness and robustness or, more prosaically, as the absence of bugs.

Everyone developing software systems, or just using them, knows how pressing this question of reliability is in the current state of software engineering. Yet the rapidly growing literature on object-oriented analysis, design, and programming includes remarkably few contributions on how to make object-oriented software more reliable. This is surprising and regrettable, since at least three reasons justify devoting particular attention to reliability in the context of object-oriented development:

- The cornerstone of object-oriented technology is reuse. For reusable components, which may be used in thousands of different applications, the potential consequences of incorrect behavior are even more serious than for application-specific developments.
- Proponents of object-oriented methods make strong claims about their beneficial effect on software quality. Reliability is certainly a central component of any reasonable definition of quality as applied to software.
- The object-oriented approach, based on the theory of abstract data types, provides a particularly appropriate framework for discussing and enforcing reliability.

The pragmatic techniques presented in this article, while certainly not providing infallible ways to guarantee reliability, may help considerably toward this goal. They rely on the theory of *design by contract*, which underlies the design of the Eiffel analysis, design, and programming language[1] and of the supporting libraries, from which a number of examples will be drawn.

The contributions of the work reported below include

- a coherent set of *methodological principles* helping to produce correct and robust software;
- a systematic approach to the delicate problem of how to deal with abnormal cases, leading to a simple and powerful *exception-handling* mechanism; and

0018-9162/92/1000-0040$03.00 © 1992 IEEE

1986 (original)

Applying "Design by Contract"

Bertrand Meyer
Interactive Software Engineering

Reliability is even more important in object-oriented programming than elsewhere. This article shows how to reduce bugs by building software components on the basis of carefully designed contracts.

- The cornerstone of object-oriented technology is reuse. For reusable components, which may be used in thousands of different applications, the potential consequences of incorrect behavior are even more serious than for application-specific developments.
- Proponents of object-oriented methods make strong claims about their beneficial effect on software quality. Reliability is certainly a central component of any reasonable definition of quality as applied to software.
- The object-oriented approach, based on the theory of abstract data types, provides a particularly appropriate framework for discussing and enforcing reliability.

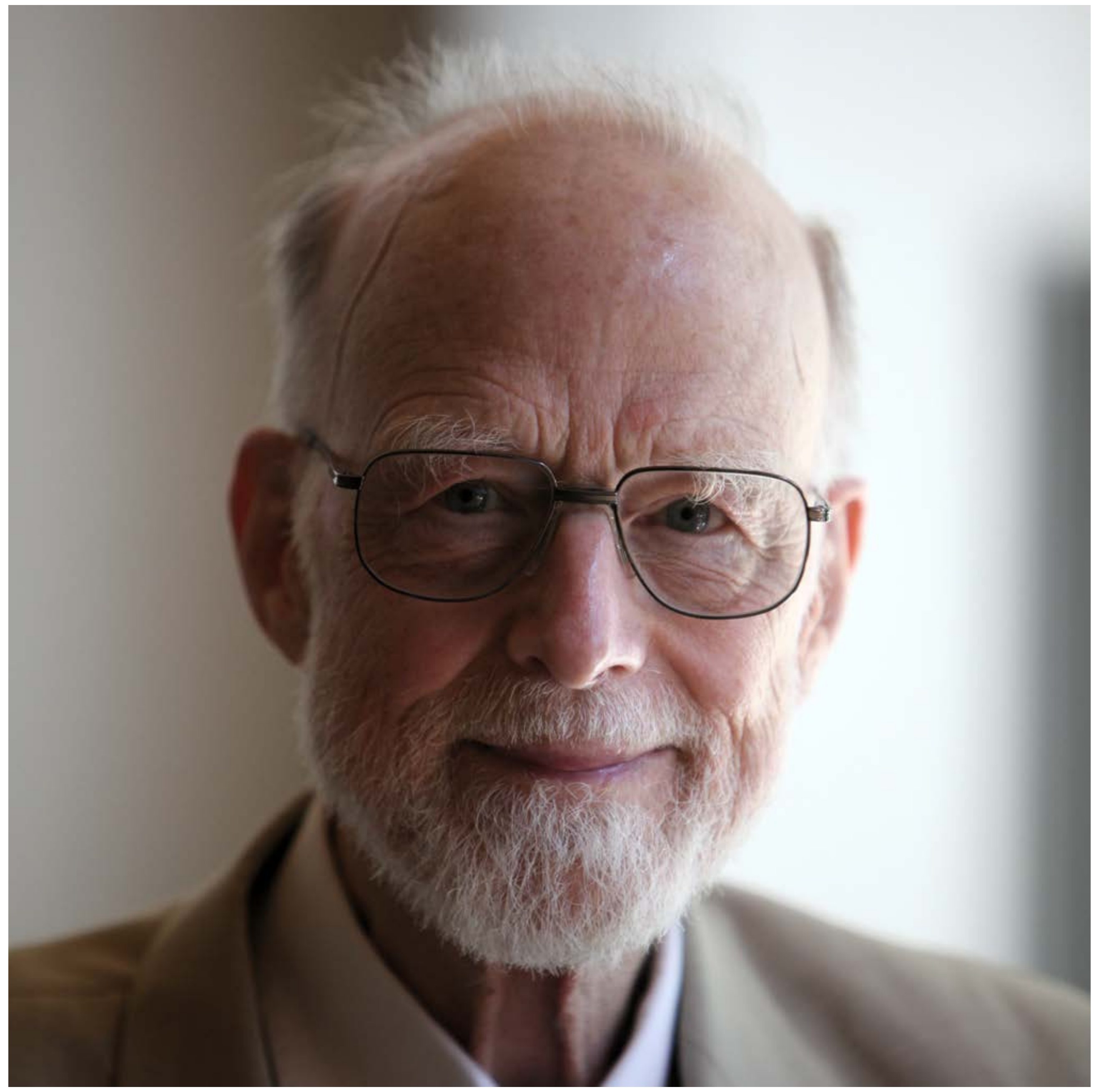
The pragmatic techniques presented in this article, while certainly not providing infallible ways to guarantee reliability, may help considerably toward this goal. They rely on the theory of *design by contract*, which underlies the design of the Eiffel analysis, design, and programming language[1] and of the supporting libraries, from which a number of examples will be drawn.

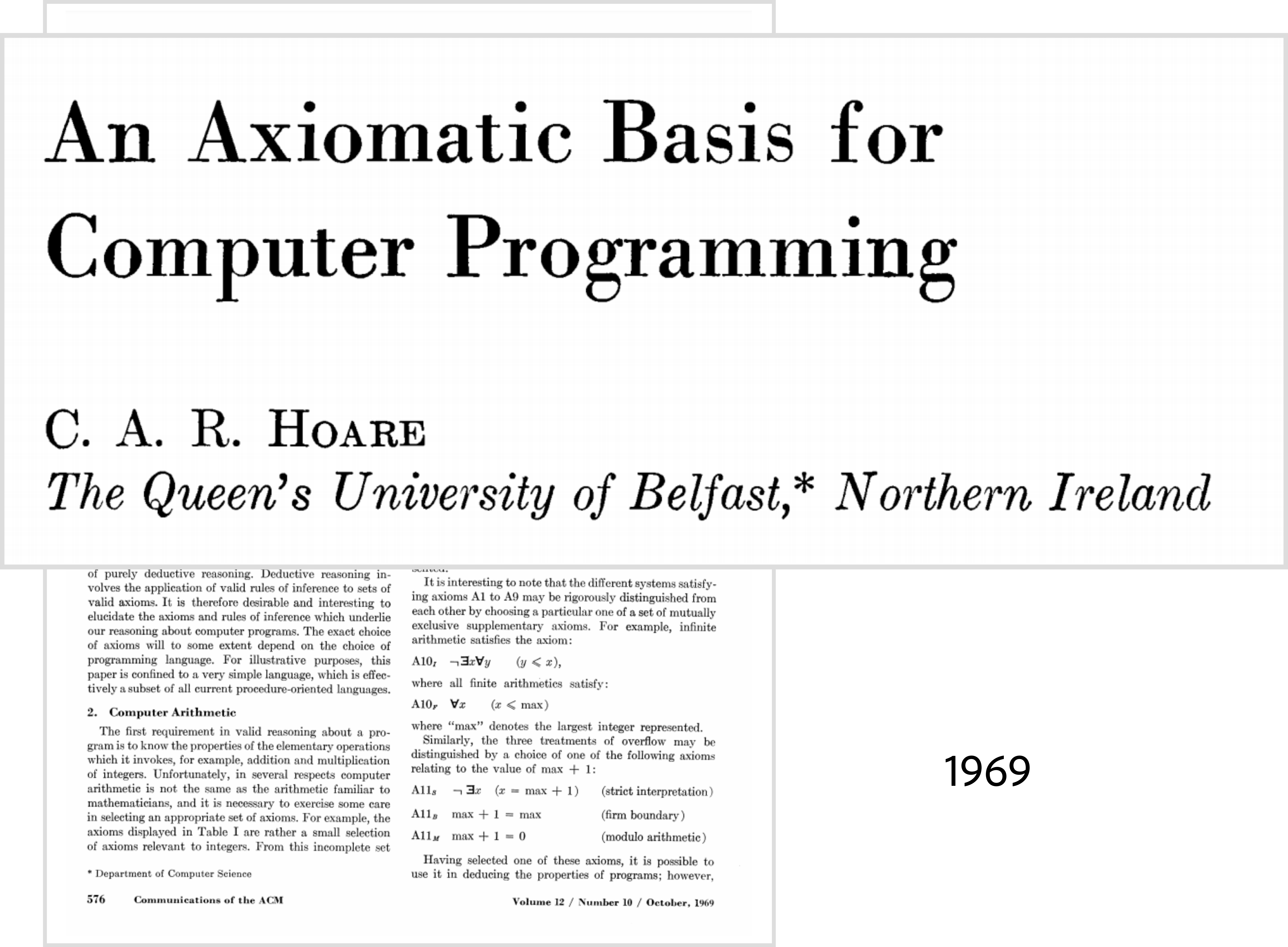
The contributions of the work reported below include

- a coherent set of *methodological principles* helping to produce correct and robust software;
- a systematic approach to the delicate problem of how to deal with abnormal cases, leading to a simple and powerful *exception-handling* mechanism; and

0018-9162/92/1000-0040$03.00 © 1992 IEEE

1986 (original)





An Axiomatic Basis for Computer Programming

C. A. R. Hoare
The Queen's University of Belfast, Northern Ireland*

of purely deductive reasoning. Deductive reasoning involves the application of valid rules of inference to sets of valid axioms. It is therefore desirable and interesting to elucidate the axioms and rules of inference which underlie our reasoning about computer programs. The exact choice of axioms will to some extent depend on the choice of programming language. For illustrative purposes, this paper is confined to a very simple language, which is effectively a subset of all current procedure-oriented languages.

2. Computer Arithmetic

The first requirement in valid reasoning about a program is to know the properties of the elementary operations which it invokes, for example, addition and multiplication of integers. Unfortunately, in several respects computer arithmetic is not the same as the arithmetic familiar to mathematicians, and it is necessary to exercise some care in selecting an appropriate set of axioms. For example, the axioms displayed in Table I are rather a small selection of axioms relevant to integers. From this incomplete set

sented.

It is interesting to note that the different systems satisfying axioms A1 to A9 may be rigorously distinguished from each other by choosing a particular one of a set of mutually exclusive supplementary axioms. For example, infinite arithmetic satisfies the axiom:

$A10_I \quad \neg\exists x \forall y \quad (y \leqslant x),$

where all finite arithmetics satisfy:

$A10_F \quad \forall x \quad (x \leqslant \max)$

where "max" denotes the largest integer represented.

Similarly, the three treatments of overflow may be distinguished by a choice of one of the following axioms relating to the value of max + 1:

$A11_S \quad \neg\exists x \quad (x = \max + 1)$ (strict interpretation)

$A11_B \quad \max + 1 = \max$ (firm boundary)

$A11_M \quad \max + 1 = 0$ (modulo arithmetic)

Having selected one of these axioms, it is possible to use it in deducing the properties of programs; however,

* Department of Computer Science

1969

Contracts

- Originally part of the Eiffel language
- Contracts allow the specification of constraints
  - Preconditions (require)
  - Postconditions (ensure)
  - Class Invariants

Contracts

- Contracts are actionable predicates on values

"In some cases, one might want to use quantified expressions of the form "For all *x* of type *T*, *p*(*x*) holds" or "There exists *x* of type *T*, such that *p*(*x*) holds," where *p* is a certain Boolean property. Such expressions are not available in Eiffel."

Concepts and Contracts

- Concepts describe relationships between operations on a type
- Contracts describe relationships between values

- The distinction is not always clear
  - i.e. The comparison operation passed to `std::sort` must implement a *strict weak ordering relation* over the values being sorted

Pattern Matching

Pattern Matching

- Concepts are used as a compile time constraint to select an appropriate operation

Pattern Matching

- Concepts are used as a compile time constraint to select an appropriate operation
- Contracts assert at runtime if an operations preconditions are not met

Pattern Matching

- Concepts are used as a compile time constraint to select an appropriate operation
- Contracts assert at runtime if an operations preconditions are not met

- A runtime constraint to select an appropriate operation is known as *pattern matching*

Pattern Matching

- Concepts are used as a compile time constraint to select an appropriate operation
- Contracts assert at runtime if an operations preconditions are not met

- A runtime constraint to select an appropriate operation is known as *pattern matching*

```
void f(auto i) requires requires { !(i < 0) }
```

Pattern Matching

- Concepts are used as a compile time constraint to select an appropriate operation
- Contracts assert at runtime if an operations preconditions are not met

- A runtime constraint to select an appropriate operation is known as *pattern matching*

```
void f(auto i) requires requires { !(i < 0) }
void f(int i) [[expects !(i < 0)]]
```

Pattern Matching

- Concepts are used as a compile time constraint to select an appropriate operation
- Contracts assert at runtime if an operations preconditions are not met

- A runtime constraint to select an appropriate operation is known as *pattern matching*

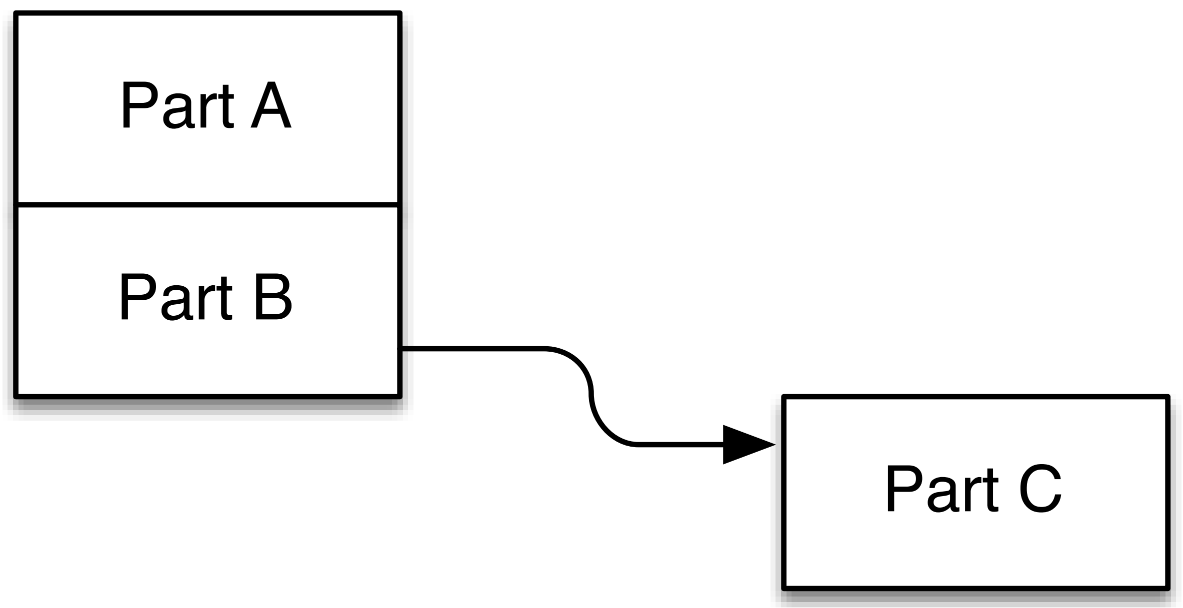
```
void f(auto i) requires requires { !(i < 0) }
void f(int i) [[expects !(i < 0)]]
void f(int i) requires !(i < 0) // Not yet in C++…
```

Whole-Part Relationships and Composite Objects

Elements of Programming, Chapter 12

Whole-Part Relationships and Composite Objects

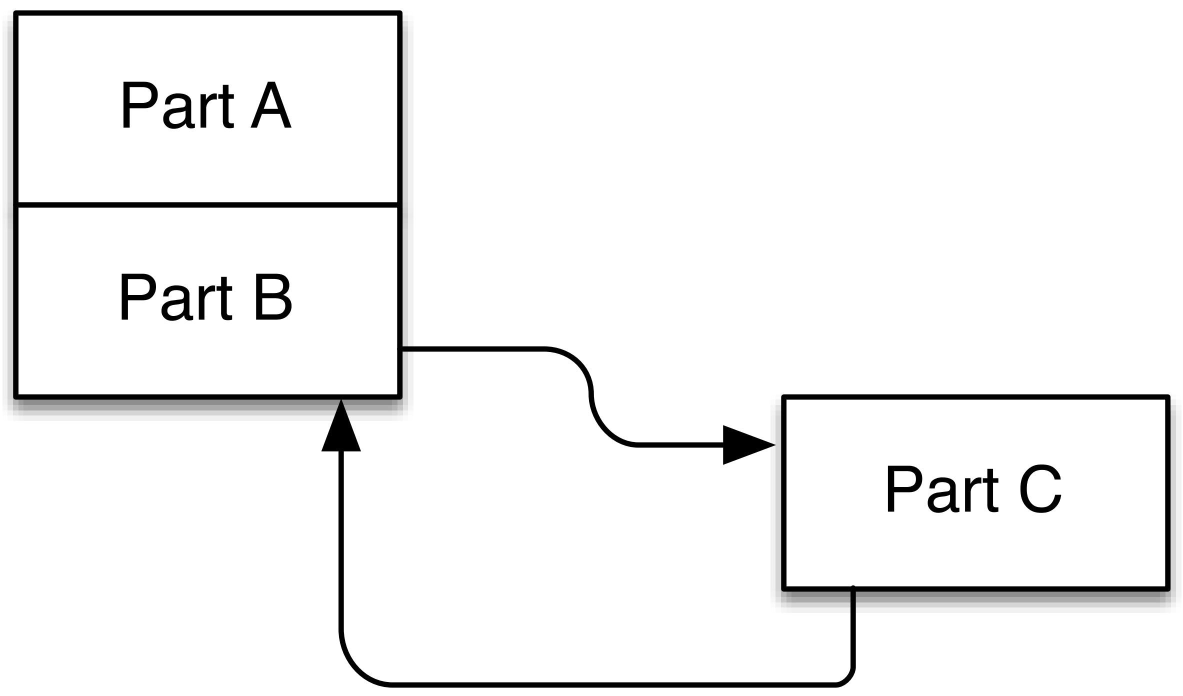
- Connected



Elements of Programming, Chapter 12

Whole-Part Relationships and Composite Objects

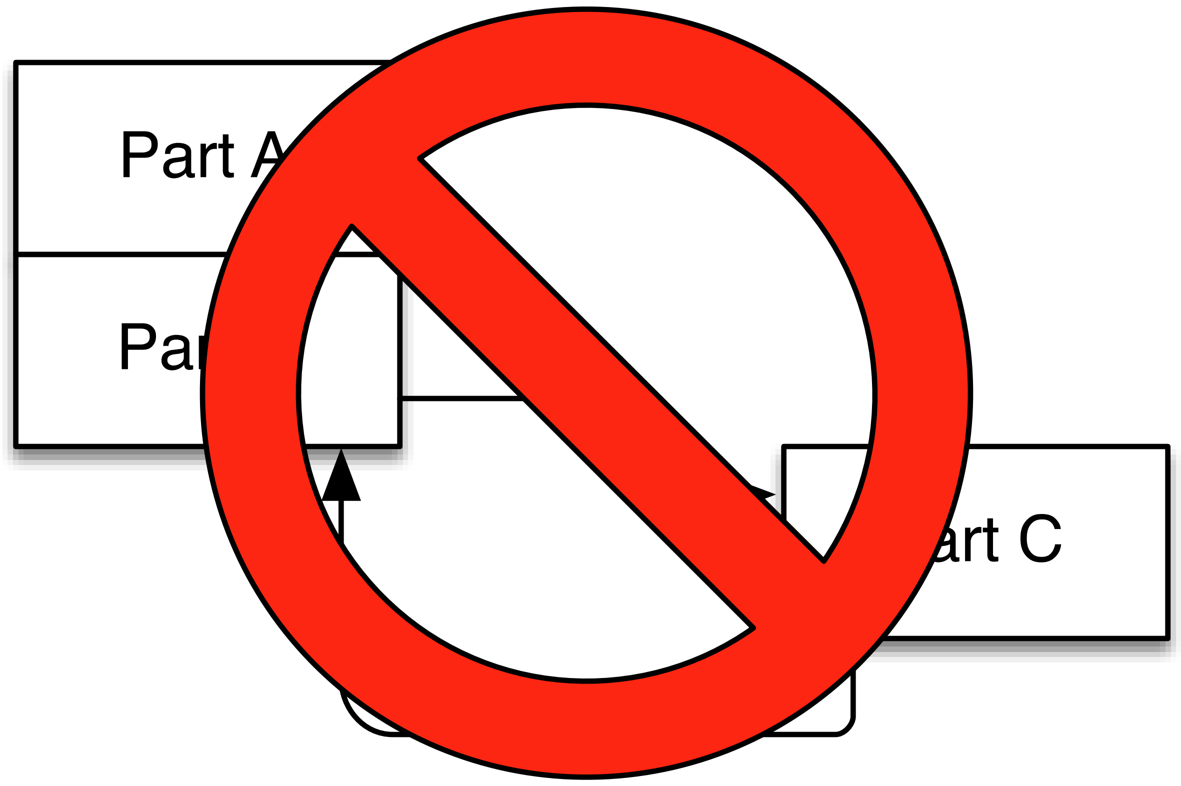
- Connected
- Noncircular



Elements of Programming, Chapter 12

Whole-Part Relationships and Composite Objects

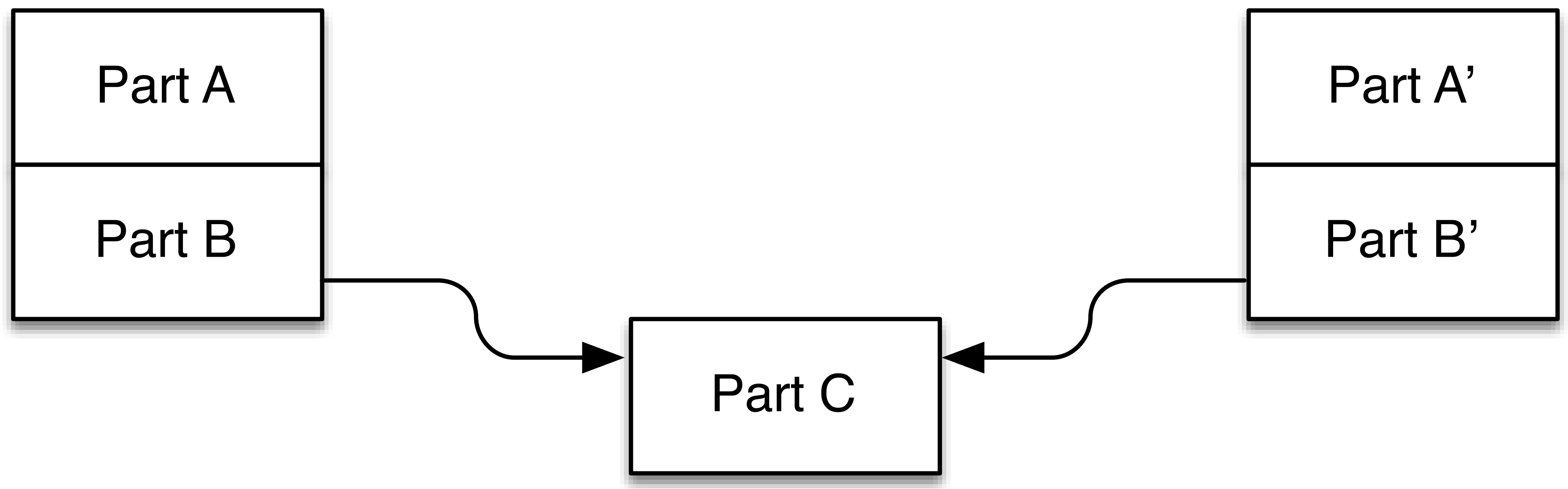
- Connected
- Noncircular



Elements of Programming, Chapter 12

Whole-Part Relationships and Composite Objects

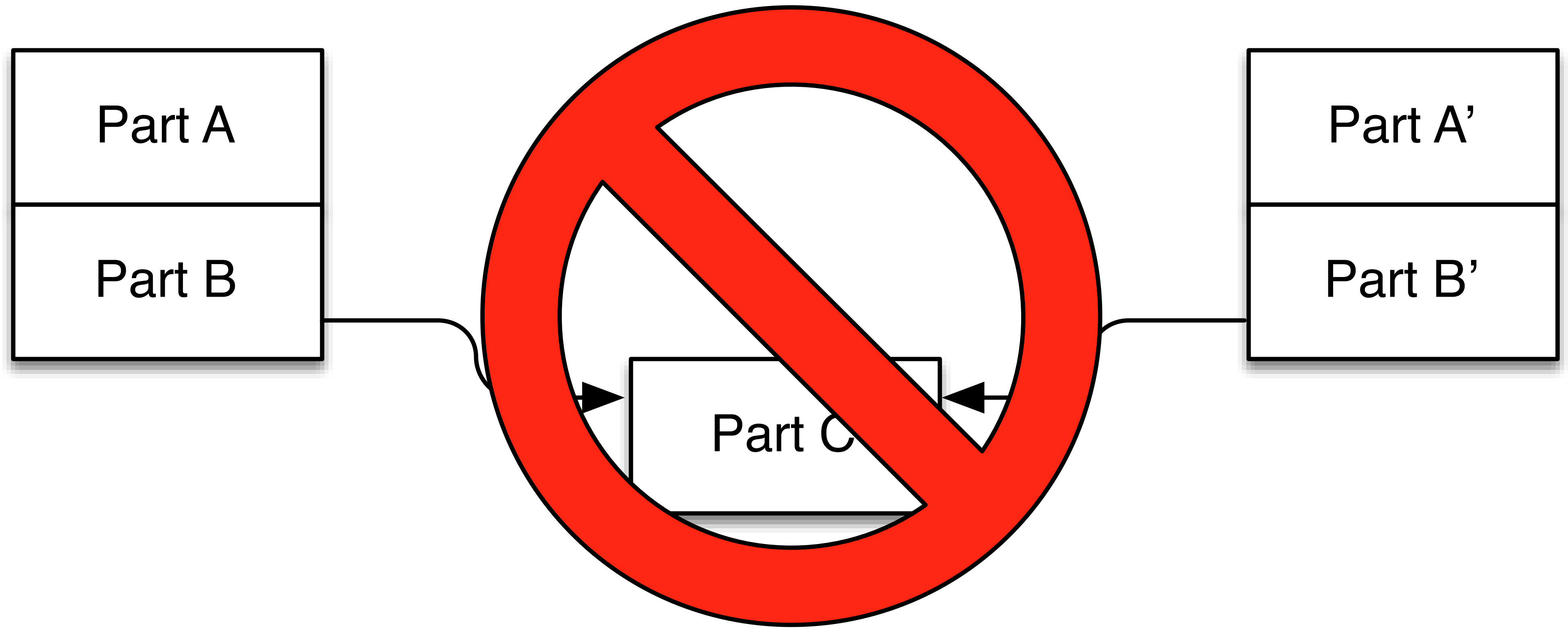
- Connected
- Noncircular
- Logically Disjoint



Elements of Programming, Chapter 12

Whole-Part Relationships and Composite Objects

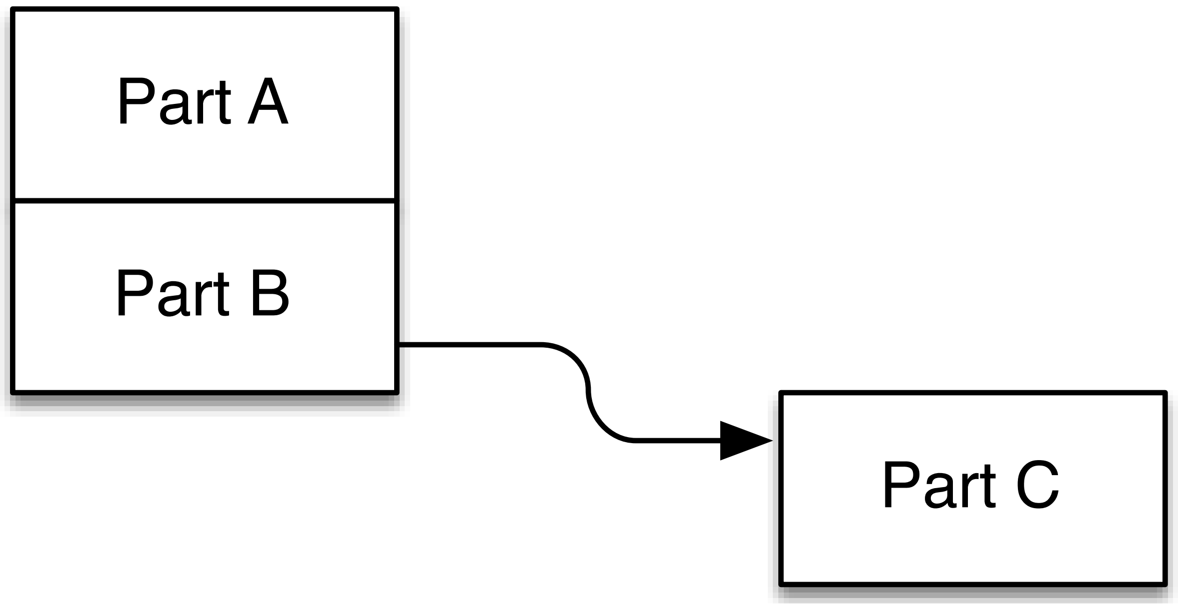
- Connected
- Noncircular
- Logically Disjoint



Elements of Programming, Chapter 12

Whole-Part Relationships and Composite Objects

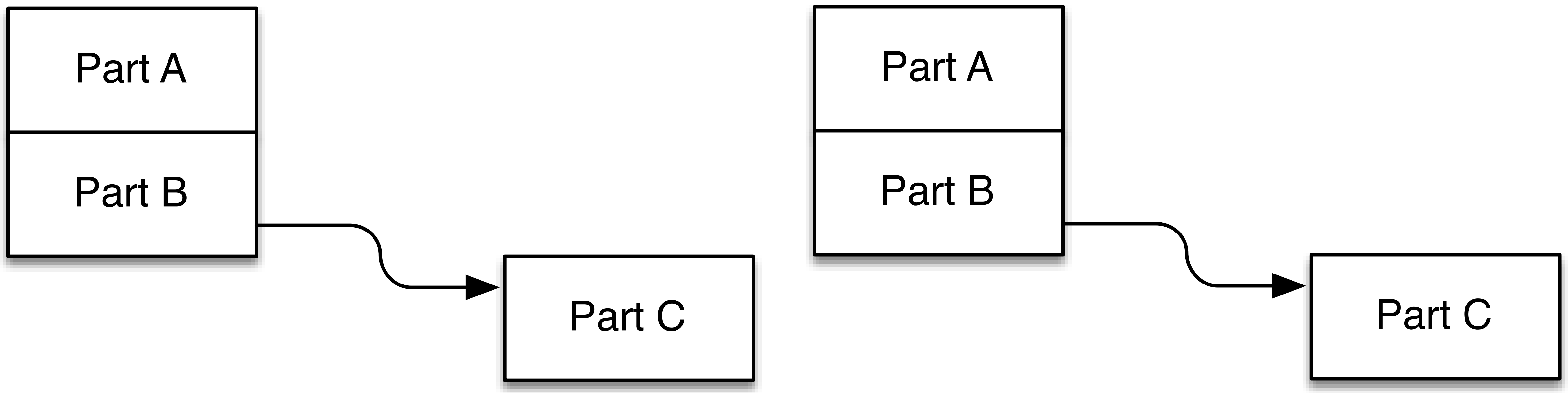
- Connected
- Noncircular
- Logically Disjoint
- Owning



Elements of Programming, Chapter 12

Whole-Part Relationships and Composite Objects

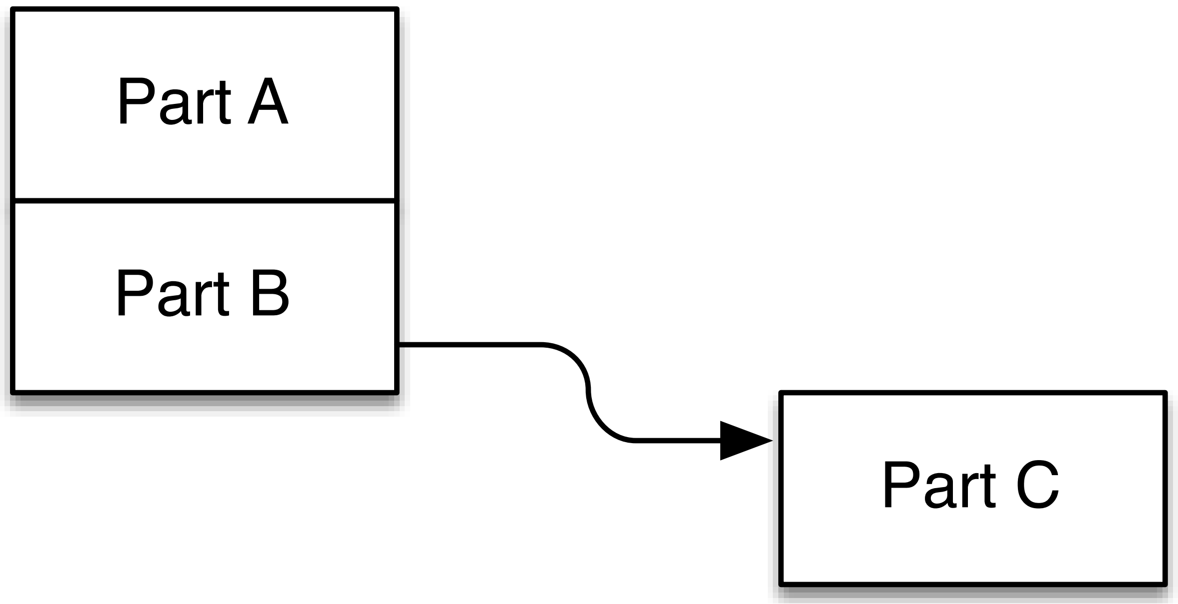
- Connected
- Noncircular
- Logically Disjoint
- Owning



Elements of Programming, Chapter 12

Whole-Part Relationships and Composite Objects

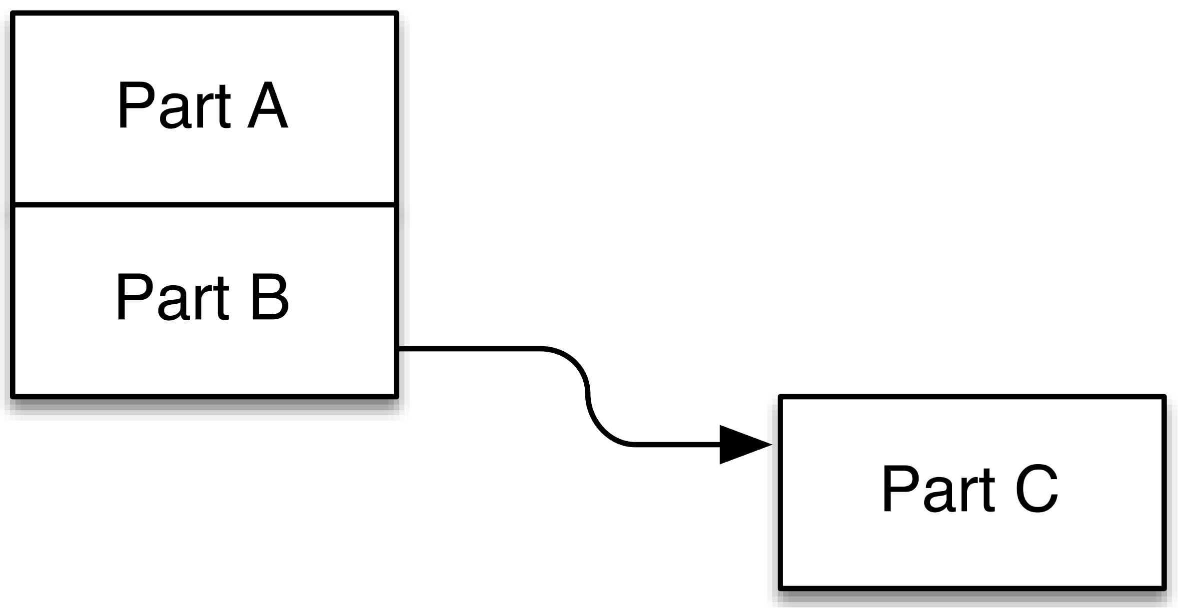
- Connected
- Noncircular
- Logically Disjoint
- Owning



Elements of Programming, Chapter 12

Whole-Part Relationships and Composite Objects

- Connected
- Noncircular
- Logically Disjoint
- Owning



- Standard Containers are Composite Objects
- Composite objects allow us to reason about a collection of objects as a single entity

Elements of Programming, Chapter 12

No Incidental Data Structures

```
class view {
    std::list<std::shared_ptr<view>> _children;
    std::weak_ptr<view> _parent;
    //...
};
```

No Incidental Data Structures

```
adobe::forest<view>
```

No Incidental Data Structures

views

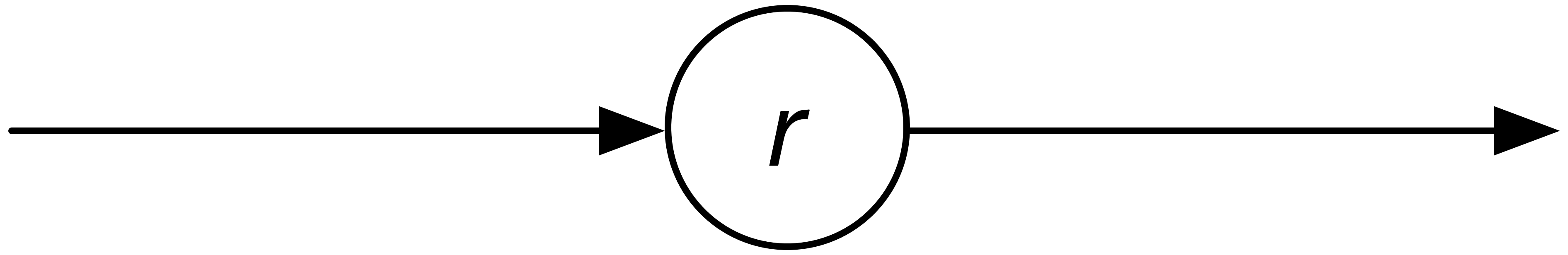
No Raw Loops

```
// Next, check if the panel has moved to the other side of another panel.
const int center_x = fixed_panel->cur_panel_center();
for (size_t i = 0; i < expanded_panels_.size(); ++i) {
  Panel* panel = expanded_panels_[i].get();
  if (center_x <= panel->cur_panel_center() ||
      i == expanded_panels_.size() - 1) {
    if (panel != fixed_panel) {
      // If it has, then we reorder the panels.
      ref_ptr<Panel> ref = expanded_panels_[fixed_index];
      expanded_panels_.erase(expanded_panels_.begin() + fixed_index);
      if (i < expanded_panels_.size()) {
        expanded_panels_.insert(expanded_panels_.begin() + i, ref);
      } else {
        expanded_panels_.push_back(ref);
      }
    }
    break;
  }
}
```

No Raw Loops

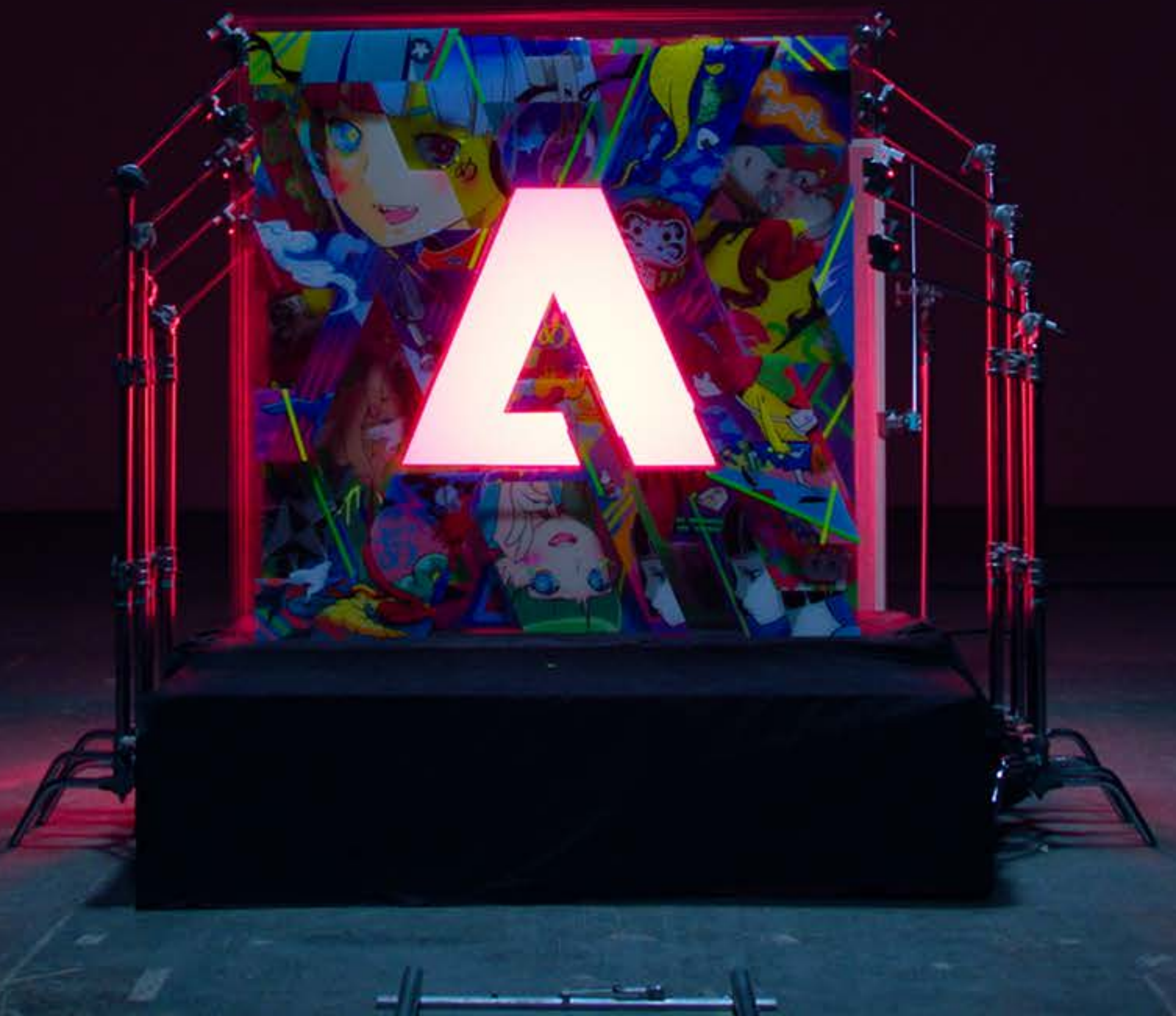
```
std::rotate(p, f, f + 1);
```

No Raw Loops



The Game

Architecture



#AdobeRemix
Hiroyuki-Mitsume Takahashi

Architecture is the art and practice of designing and constructing structures.

Task

- Save the document every 5 minutes, after the application has been idle for at least 5 seconds.

Task

- **Save the document every 5 minutes**, after the application has been idle for at least 5 seconds.

Task

- ~~Save the document every 5 minutes~~, after the application has been idle for at least 5 seconds.

Task

- After the application has been idle for at least *n* seconds do *something*

Task

- After the application has been idle for at least *n* seconds do *something*

```
extern system_clock::time_point _last_idle;
void invoke_after(system_clock::duration, function<void()>);

template <class F> // F is task of the form void()
void after_idle(F task, system_clock::duration delay) {
    auto when = delay - (system_clock::now() - _last_idle);

    if (system_clock::duration::zero() < when) {
        invoke_after(when, [=]{ after_idle(task, delay); });
    } else {
        task();
    }
}
```

Task

- After the application has been idle for at least *n* seconds do *something*

```
extern system_clock::time_point _last_idle;
void invoke_after(system_clock::duration, function<void()>);

template <class F> // F is task of the form void()
void after_idle(F task, system_clock::duration delay) {
    auto when = delay - (system_clock::now() - _last_idle);

    if (system_clock::duration::zero() < when) {
        invoke_after(when, [=]{ after_idle(task, delay); });
    } else {
        task();
    }
}
```

Task

- After the application has been idle for at least *n* seconds do *something*

```
extern system_clock::time_point _last_idle;
void invoke_after(system_clock::duration, function<void()>);

template <class F> // F is task of the form void()
void after_idle(F task, system_clock::duration delay) {
    auto when = delay - (system_clock::now() - _last_idle);

    if (system_clock::duration::zero() < when) {
        invoke_after(when, [=]{ after_idle(task, delay); });
    } else {
        task();
    }
}
```

Task

- After the application has been idle for at least *n* seconds do *something*

```
extern system_clock::time_point _last_idle;
void invoke_after(system_clock::duration, function<void()>);

template <class F> // F is task of the form void()
void after_idle(F task, system_clock::duration delay) {
    auto when = delay - (system_clock::now() - _last_idle);

    if (system_clock::duration::zero() < when) {
        invoke_after(when, [=]{ after_idle(task, delay); });
    } else {
        task();
    }
}
```

Task

- After the application has been idle for at least *n* seconds do *something*

```
extern system_clock::time_point _last_idle;
void invoke_after(system_clock::duration, function<void()>);

template <class F> // F is task of the form void()
void after_idle(F task, system_clock::duration delay) {
    auto when = delay - (system_clock::now() - _last_idle);

    if (system_clock::duration::zero() < when) {
        invoke_after(when, [=]{ after_idle(task, delay); });
    } else {
        task();
    }
}
```

Task

- After the application has been idle for at least *n* seconds do *something*

```
extern system_clock::time_point _last_idle;
void invoke_after(system_clock::duration, function<void()>);

template <class F> // F is task of the form void()
void after_idle(F task, system_clock::duration delay) {
    auto when = delay - (system_clock::now() - _last_idle);

    if (system_clock::duration::zero() < when) {
        invoke_after(when, [=]{ after_idle(task, delay); });
    } else {
        task();
    }
}
```

Task

- After the application has been idle for at least *n* seconds do *something*

```
extern system_clock::time_point _last_idle;
void invoke_after(system_clock::duration, function<void()>);

template <class F> // F is task of the form void()
void after_idle(F task, system_clock::duration delay) {
    auto when = delay - (system_clock::now() - _last_idle);

    if (system_clock::duration::zero() < when) {
        invoke_after(when, [=]{ after_idle(task, delay); });
    } else {
        task();
    }
}
```

Task

- After the application has been idle for at least *n* seconds do *something*

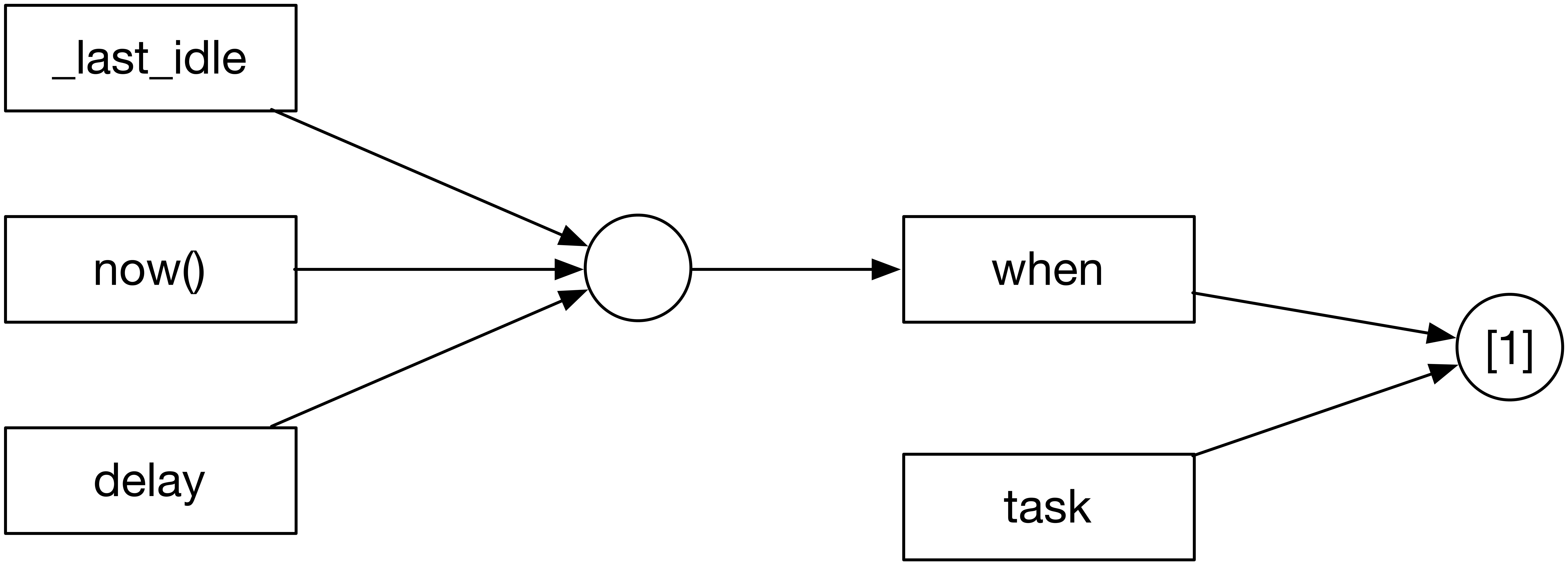
```
extern system_clock::time_point _last_idle;
void invoke_after(system_clock::duration, function<void()>);

template <class F> // F is task of the form void()
void after_idle(F task, system_clock::duration delay) {
    auto when = delay - (system_clock::now() - _last_idle);

    if (system_clock::duration::zero() < when) {
        invoke_after(when, [=]{ after_idle(task, delay); });
    } else {
        task();
    }
}
```

Visualizing the Relationships

- The structure, ignoring the recursion in **`invoke_after`**[1]

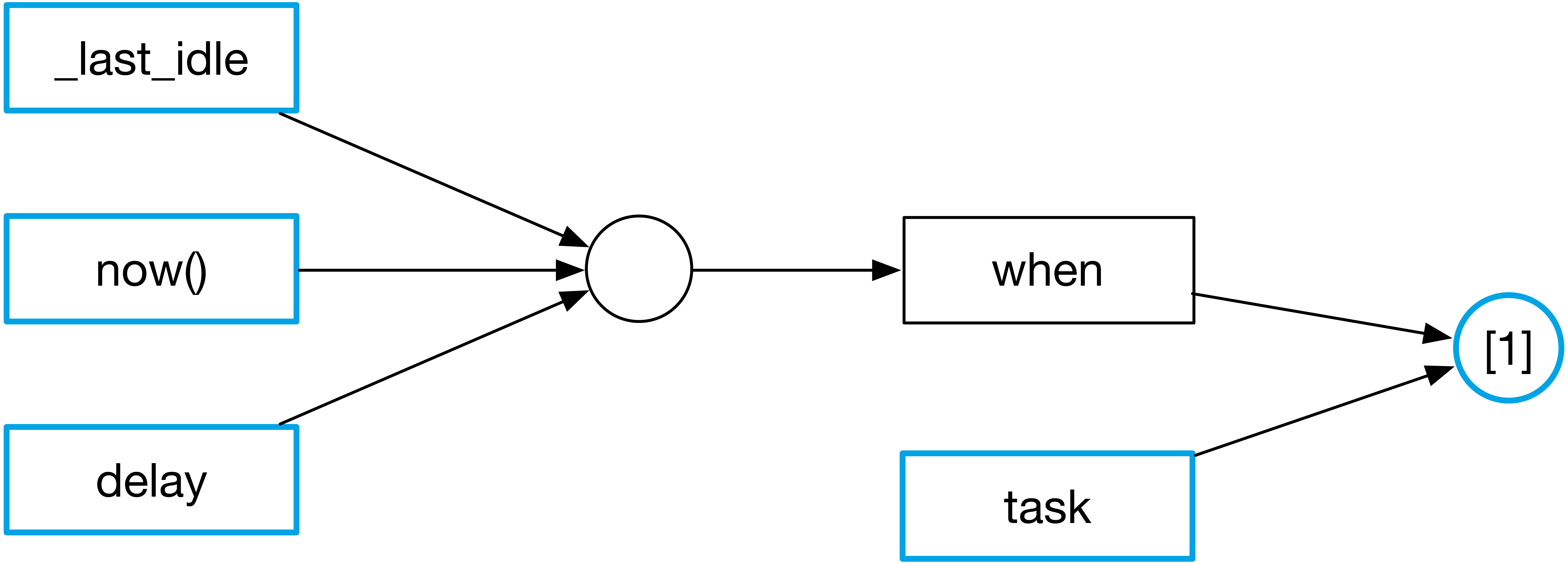
Visualizing the Relationships

- The structure, ignoring the recursion in **`invoke_after`**[1]



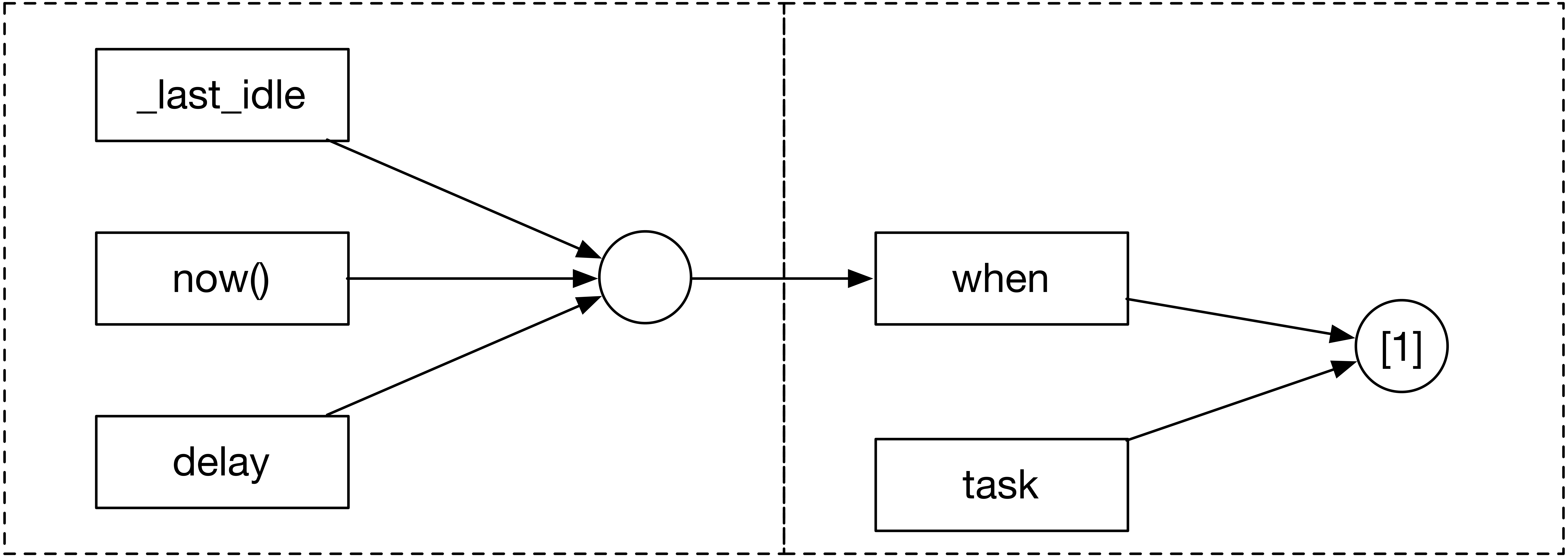
Visualizing the Relationships

- The arguments and dependencies



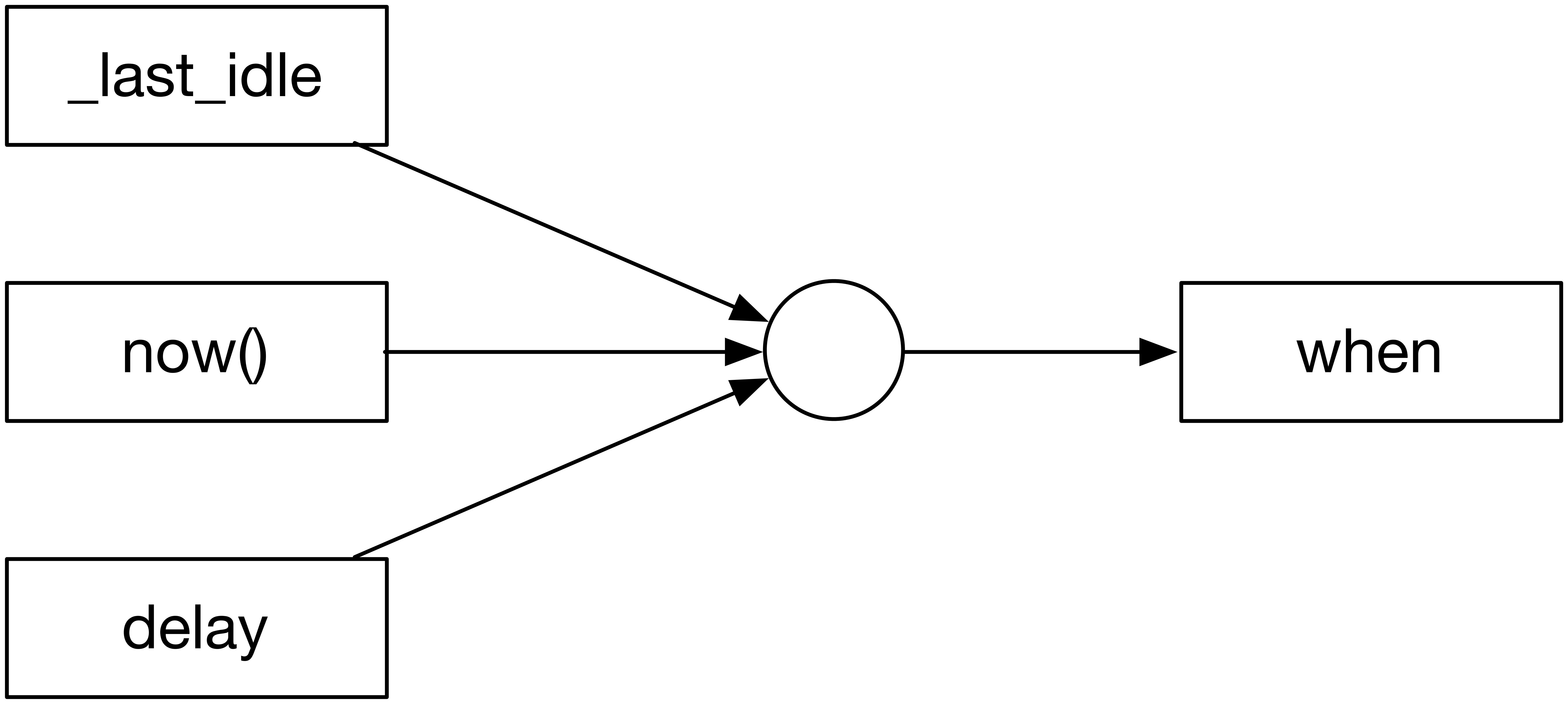
Visualizing the Relationships

- Two operations



Visualizing the Relationships

```
auto when = delay - (system_clock::now() - _last_idle);
```



Visualizing the Relationships

```
auto when = delay - (system_clock::now() - _last_idle);
```

Visualizing the Relationships

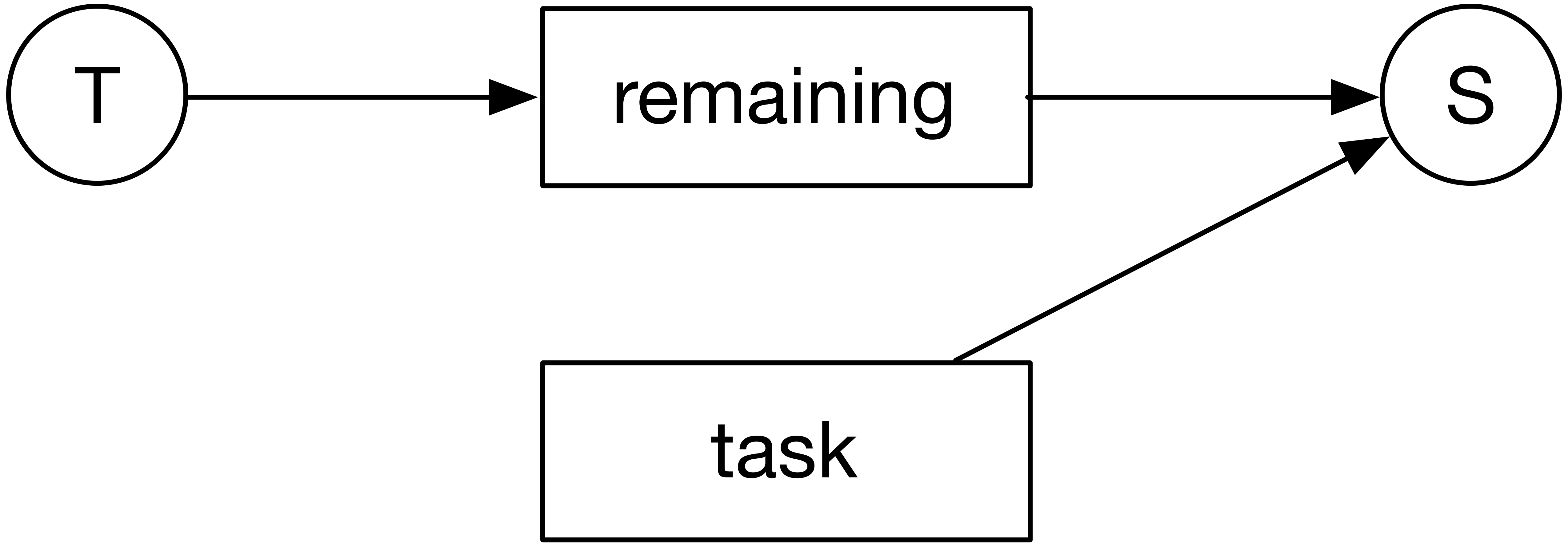
```
auto when = delay - (system_clock::now() - _last_idle);
```

Visualizing the Relationships

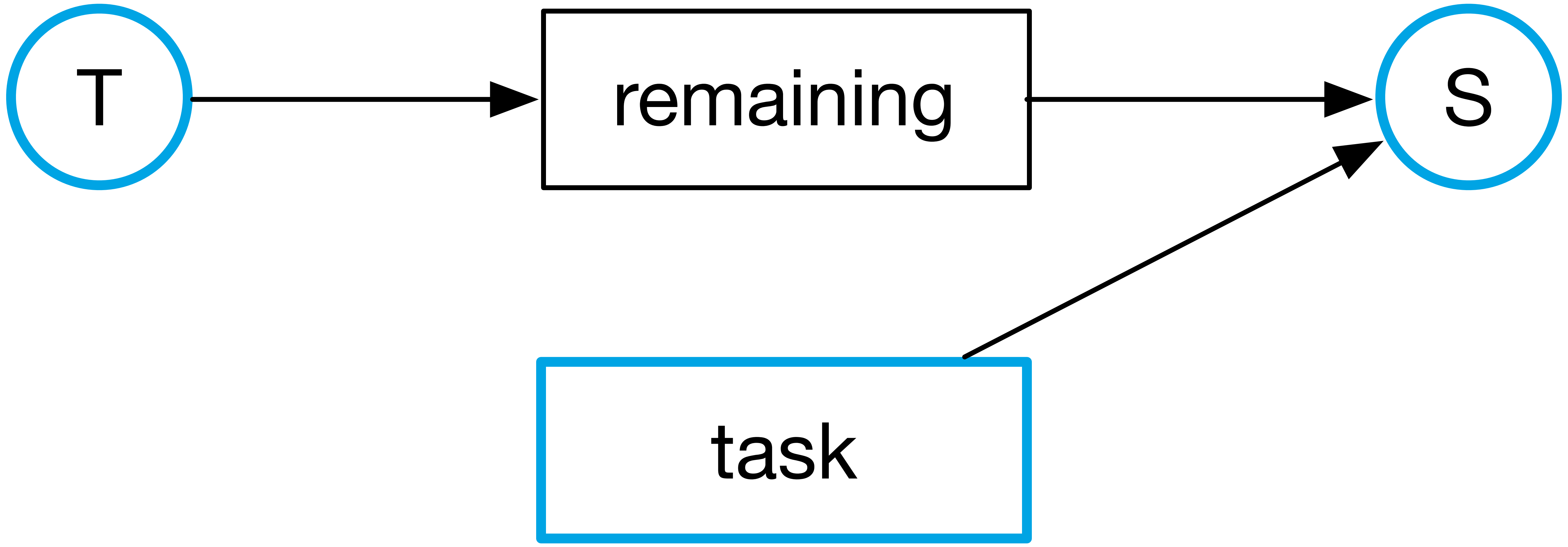
```
auto when = delay - (system_clock::now() - _last_idle);
```

On Expiration

On Expiration



On Expiration



On Expiration

```
template <class S, class T, class F>
void on_expiration_(S scheduler, T timer, F task) {
    auto remaining = timer();

    if (decltype(remaining){0} < remaining) {
        scheduler(remaining, [=] {
            on_expiration_(scheduler, timer, task);
        });
    } else {
        task();
    }
}
```

On Expiration

```
template <class S, class T, class F>
void on_expiration_(S scheduler, T timer, F task) {
    auto remaining = timer();

    if (decltype(remaining){0} < remaining) {
        scheduler(remaining, [=] {
            on_expiration_(scheduler, timer, task);
        });
    } else {
        task();
    }
}
```

On Expiration

```
template <class S, class T, class F>
void on_expiration_(S scheduler, T timer, F task) {
    auto remaining = timer();

    if (decltype(remaining){0} < remaining) {
        scheduler(remaining, [=] {
            on_expiration_(scheduler, timer, task);
        });
    } else {
        task();
    }
}
```

On Expiration

```
template <class S, class T, class F>
void on_expiration_(S scheduler, T timer, F task) {
    auto remaining = timer();

    if (decltype(remaining){0} < remaining) {
        scheduler(remaining, [=] {
            on_expiration_(scheduler, timer, task);
        });
    } else {
        task();
    }
}
```

On Expiration

```
template <class S, class T, class F>
void on_expiration_(S scheduler, T timer, F task) {
    auto remaining = timer();

    if (decltype(remaining){0} < remaining) {
        scheduler(remaining, [=] {
            on_expiration_(scheduler, timer, task);
        });
    } else {
        task();
    }
}
```

On Expiration

```
template <class S, class T, class F>
void on_expiration_(S scheduler, T timer, F task) {
    auto remaining = timer();

    if (decltype(remaining){0} < remaining) {
        scheduler(remaining, [=] {
            on_expiration_(scheduler, timer, task);
        });
    } else {
        task();
    }
}
```

On Expiration

```
template <class S, class T, class F>
void on_expiration_(S scheduler, T timer, F task) {
    auto remaining = timer();

    if (decltype(remaining){0} < remaining) {
        scheduler(remaining, [=] {
            on_expiration_(scheduler, timer, task);
        });
    } else {
        task();
    }
}
```

On Expiration

```
template <class S, class T, class F>
void on_expiration_(S scheduler, T timer, F task) {
    auto remaining = timer();

    if (decltype(remaining){0} < remaining) {
        scheduler(remaining, [=] {
            on_expiration_(scheduler, timer, task);
        });
    } else {
        task();
    }
}

template <class S, class T, class F>
void on_expiration(S scheduler, T timer, F task) {
    scheduler(timer(), [=] { on_expiration_(scheduler, timer, task); });
}
```

Architecture

Architecture

- By looking at the structure of the function we can design a better function

Architecture

- By looking at the structure of the function we can design a better function
- Note that on_expiration has no external dependencies

Architecture

- By looking at the structure of the function we can design a better function
- Note that on_expiration has no external dependencies
  - No **`std::chrono`**

Architecture

- By looking at the structure of the function we can design a better function
- Note that on_expiration has no external dependencies
  - No `std::chrono`
  - No `std::function`

Architecture

- By looking at the structure of the function we can design a better function
- Note that on_expiration has no external dependencies
  - No **`std::chrono`**
  - No **`std::function`**
  - Or **`invoke_after`** or **`_last_idle`**

Architecture

- By looking at the structure of the function we can design a better function
- Note that on_expiration has no external dependencies
  - No **`std::chrono`**
  - No **`std::function`**
  - Or **`invoke_after`** or **`_last_idle`**
- Requirements are the semantics of the operations and the relationship between arguments

Registry

Registry

- A registry is a container supporting the following operations

Registry

- A registry is a container supporting the following operations
  - Add an object, and obtain a *receipt*

Adobe

Registry

- A registry is a container supporting the following operations
  - Add an object, and obtain a *receipt*
  - Use the receipt to retrieve the object or remove it

Registry

- A registry is a container supporting the following operations
  - Add an object, and obtain a *receipt*
  - Use the receipt to retrieve the object or remove it
  - Operate on the objects in the registry

Registry

- A registry is a container supporting the following operations
  - Add an object, and obtain a *receipt*
  - Use the receipt to retrieve the object or remove it
  - Operate on the objects in the registry

  - Example: signal handler

Registry

```
template <class T>
class registry {
    unordered_map<size_t, T> _map;
    size_t _id = 0;
public:
    auto append(T element) -> size_t {
        _map.emplace(_id, move(element));
        return _id++;
    }

    void erase(size_t id) { _map.erase(id); }

    template <typename F>
    void for_each(F f) const {
        for (const auto& e : _map)
            f(e.second);
    }
};
```

Registry

```
template <class T>
class registry {
    unordered_map<size_t, T> _map;
    size_t _id = 0;
public:
    auto append(T element) -> size_t {
        _map.emplace(_id, move(element));
        return _id++;
    }

    void erase(size_t id) { _map.erase(id); }

    template <typename F>
    void for_each(F f) const {
        for (const auto& e : _map)
            f(e.second);
    }
};
```

Registry

```
template <class T>
class registry {
    unordered_map<size_t, T> _map;
    size_t _id = 0;
public:
    auto append(T element) -> size_t {
        _map.emplace(_id, move(element));
        return _id++;
    }

    void erase(size_t id) { _map.erase(id); }

    template <typename F>
    void for_each(F f) const {
        for (const auto& e : _map)
            f(e.second);
    }
};
```

Registry

```
template <class T>
class registry {
    unordered_map<size_t, T> _map;
    size_t _id = 0;
public:
    auto append(T element) -> size_t {
        _map.emplace(_id, move(element));
        return _id++;
    }

    void erase(size_t id) { _map.erase(id); }

    template <typename F>
    void for_each(F f) const {
        for (const auto& e : _map)
            f(e.second);
    }
};
```

Registry

```
template <class T>
class registry {
    unordered_map<size_t, T> _map;
    size_t _id = 0;
public:
    auto append(T element) -> size_t {
        _map.emplace(_id, move(element));
        return _id++;
    }

    void erase(size_t id) { _map.erase(id); }

    template <typename F>
    void for_each(F f) const {
        for (const auto& e : _map)
            f(e.second);
    }
};
```

Registry

```
template <class T>
class registry {
    unordered_map<size_t, T> _map;
    size_t _id = 0;
public:
    auto append(T element) -> size_t {
        _map.emplace(_id, move(element));
        return _id++;
    }

    void erase(size_t id) { _map.erase(id); }

    template <typename F>
    void for_each(F f) const {
        for (const auto& e : _map)
            f(e.second);
    }
};
```

Registry

```
template <class T>
class registry {
    unordered_map<size_t, T> _map;
    size_t _id = 0;
public:
    auto append(T element) -> size_t {
        _map.emplace(_id, move(element));
        return _id++;
    }

    void erase(size_t id) { _map.erase(id); }

    template <typename F>
    void for_each(F f) const {
        for (const auto& e : _map)
            f(e.second);
    }
};
```

Registry

```
template <class T>
class registry {
    unordered_
    size_t _id
public:
    auto appen
        _map.e
        return
    }

    void erase

    template <
    void for_e
        for (const auto& e : _map)
            f(e.second);
    }
};
```

Adobe

Russian Coat Check Algorithm

Russian Coat Check Algorithm

- Receipts are **ordered**

Russian Coat Check Algorithm

- Receipts are **ordered**
  - Coats always appended with stub

Russian Coat Check Algorithm

- Receipts are **ordered**
  - Coats always appended with stub
  - Binary search to retrieve coat by matching receipt to stub

Russian Coat Check Algorithm

- Receipts are **ordered**
  - Coats always appended with stub
  - Binary search to retrieve coat by matching receipt to stub
    - When more than half the slot are empty, compact the coats

Russian Coat Check Algorithm

- Receipts are **ordered**
  - Coats always appended with stub
  - Binary search to retrieve coat by matching receipt to stub
    - When more than half the slot are empty, compact the coats

- Coats are always ordered by receipt stubs

Russian Coat Check Algorithm

- Receipts are **ordered**
  - Coats always appended with stub
  - Binary search to retrieve coat by matching receipt to stub
    - When more than half the slot are empty, compact the coats

- Coats are always ordered by receipt stubs
- As an additional useful properties coats are always ordered by insertion

Russian Coat Check Algorithm

```
template <class T>
class registry {
    vector<pair<size_t, optional<T>>> _map;
    size_t _size = 0;
    size_t _id = 0;

public:
    //...
```

Russian Coat Check Algorithm

```
auto append(T element) -> size_t {
    _map.emplace_back(_id, move(element));
    ++_size;
    return _id++;
}
//...
```

Russian Coat Check Algorithm

Russian Coat Check Algorithm

0	1	2	3	4	5	6	7
a	b	c	d	e	f	g	h

Russian Coat Check Algorithm

```
void erase(size_t id) {
    auto p = lower_bound(
        begin(_map), end(_map), id,
        [](const auto& a, const auto& b) { return a.first < b; });

    if (p == end(_map) || p->first != id || !p->second) return;

    p->second.reset();
    --_size;

    if (_size < (_map.size() / 2)) {
        _map.erase(remove_if(begin(_map), end(_map),
                             [](const auto& e) { return !e.second; }),
                   end(_map));
    }
}
//...
```

Russian Coat Check Algorithm

```
void erase(size_t id) {
    auto p = lower_bound(
        begin(_map), end(_map), id,
        [](const auto& a, const auto& b) { return a.first < b; });

    if (p == end(_map) || p->first != id || !p->second) return;

    p->second.reset();
    --_size;

    if (_size < (_map.size() / 2)) {
        _map.erase(remove_if(begin(_map), end(_map),
                             [](const auto& e) { return !e.second; }),
                   end(_map));
    }
}
//...
```

Russian Coat Check Algorithm

```
void erase(size_t id) {
    auto p = lower_bound(
        begin(_map), end(_map), id,
        [](const auto& a, const auto& b) { return a.first < b; });

    if (p == end(_map) || p->first != id || !p->second) return;

    p->second.reset();
    --_size;

    if (_size < (_map.size() / 2)) {
        _map.erase(remove_if(begin(_map), end(_map),
                             [](const auto& e) { return !e.second; }),
                   end(_map));
    }
}
//...
```

Russian Coat Check Algorithm

```
void erase(size_t id) {
    auto p = lower_bound(
        begin(_map), end(_map), id,
        [](const auto& a, const auto& b) { return a.first < b; });

    if (p == end(_map) || p->first != id || !p->second) return;

    p->second.reset();
    --_size;

    if (_size < (_map.size() / 2)) {
        _map.erase(remove_if(begin(_map), end(_map),
                             [](const auto& e) { return !e.second; }),
                   end(_map));
    }
}
//...
```

Russian Coat Check Algorithm

```
void erase(size_t id) {
    auto p = lower_bound(
        begin(_map), end(_map), id,
        [](const auto& a, const auto& b) { return a.first < b; });

    if (p == end(_map) || p->first != id || !p->second) return;

    p->second.reset();
    --_size;

    if (_size < (_map.size() / 2)) {
        _map.erase(remove_if(begin(_map), end(_map),
                             [](const auto& e) { return !e.second; }),
                   end(_map));
    }
}
//...
```

Russian Coat Check Algorithm

```
void erase(size_t id) {
    auto p = lower_bound(
        begin(_map), end(_map), id,
        [](const auto& a, const auto& b) { return a.first < b; });

    if (p == end(_map) || p->first != id || !p->second) return;

    p->second.reset();
    --_size;

    if (_size < (_map.size() / 2)) {
        _map.erase(remove_if(begin(_map), end(_map),
                             [](const auto& e) { return !e.second; }),
                   end(_map));
    }
}
//...
```

Russian Coat Check Algorithm

```
void erase(size_t id) {
    auto p = lower_bound(
        begin(_map), end(_map), id,
        [](const auto& a, const auto& b) { return a.first < b; });

    if (p == end(_map) || p->first != id || !p->second) return;

    p->second.reset();
    --_size;

    if (_size < (_map.size() / 2)) {
        _map.erase(remove_if(begin(_map), end(_map),
                             [](const auto& e) { return !e.second; }),
                   end(_map));
    }
}
//...
```

Russian Coat Check Algorithm

0	1	2	3	4	5	6	7
a	b	c	d	e	f	g	h

Russian Coat Check Algorithm

0	1	2	3	4	5	6	7
a	x	x	d	e	x	x	x

Russian Coat Check Algorithm

0	1	2	3	4	5	6	7
a	x	x	d	e	x	x	x

Russian Coat Check Algorithm

0	3	4	
a	d	e	

Russian Coat Check Algorithm

0	3	4	
a	d	e	

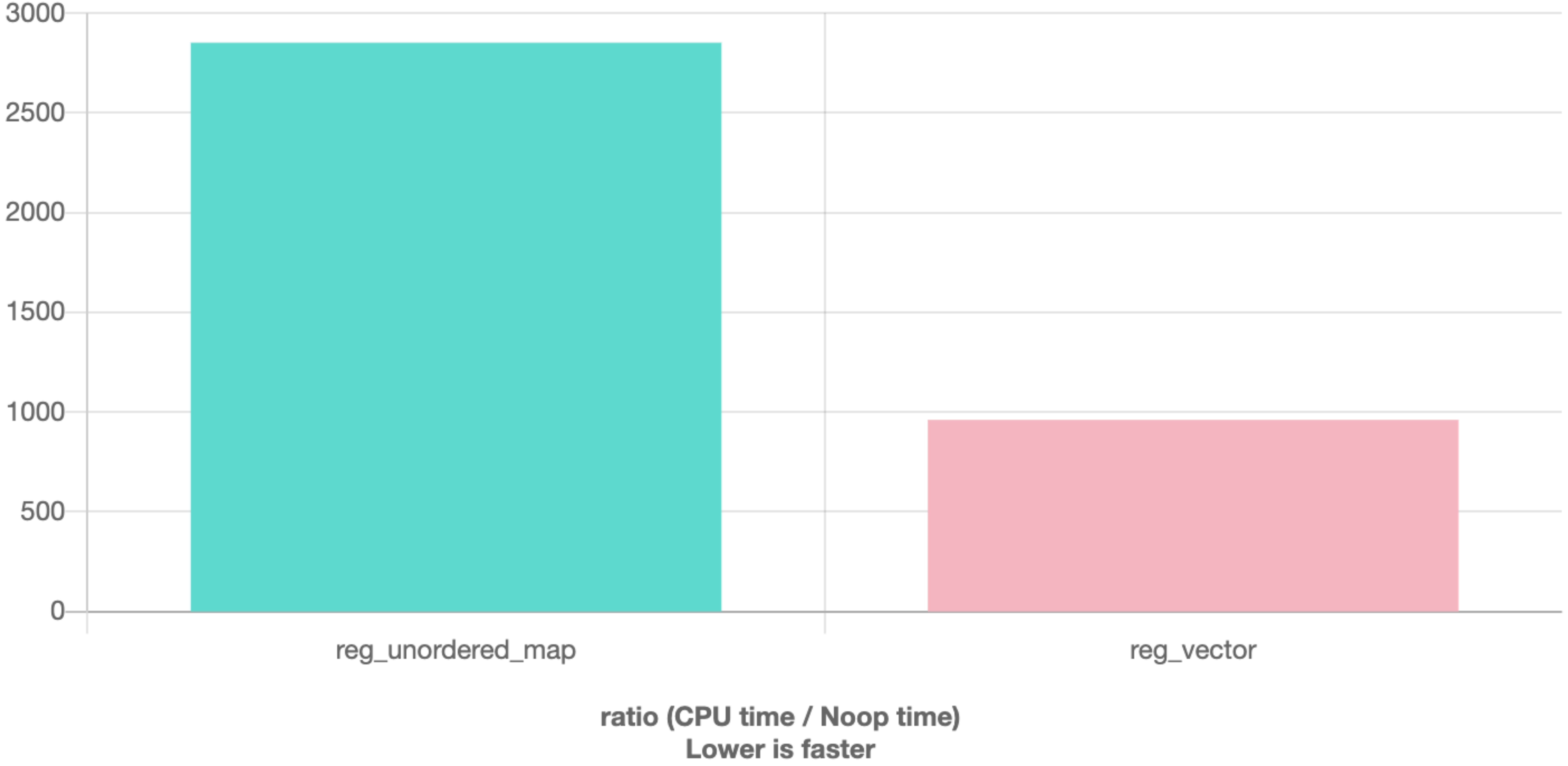
Russian Coat Check Algorithm

0	3	4	8	9	
a	d	e	i	j	

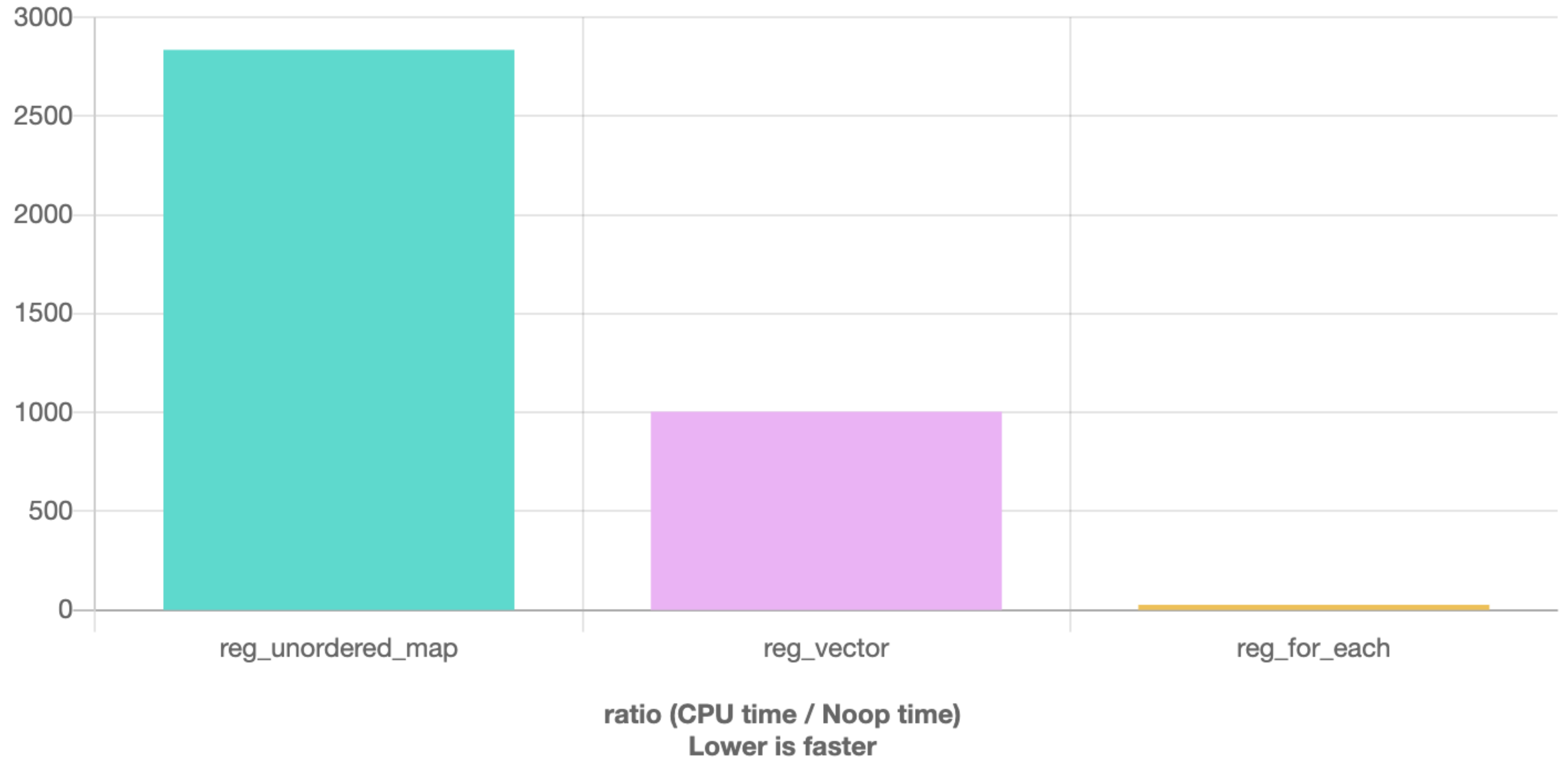
Russian Coat Check Algorithm

```
    template <typename F>
    void for_each(F f) {
        for (const auto& e : _map) {
            if (e.second) f(*e.second);
        }
    }
};
```

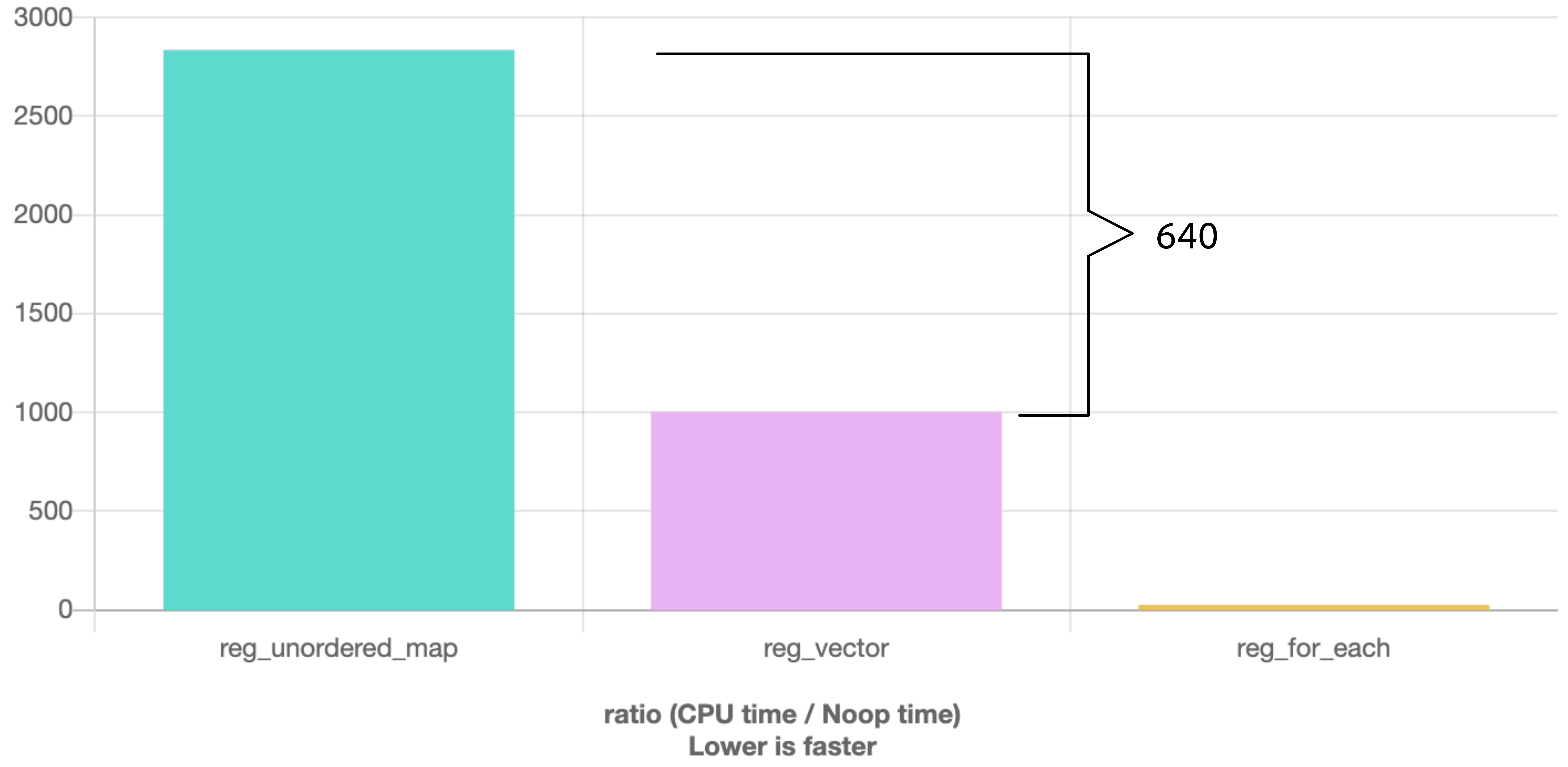
Russian Coat Check Algorithm



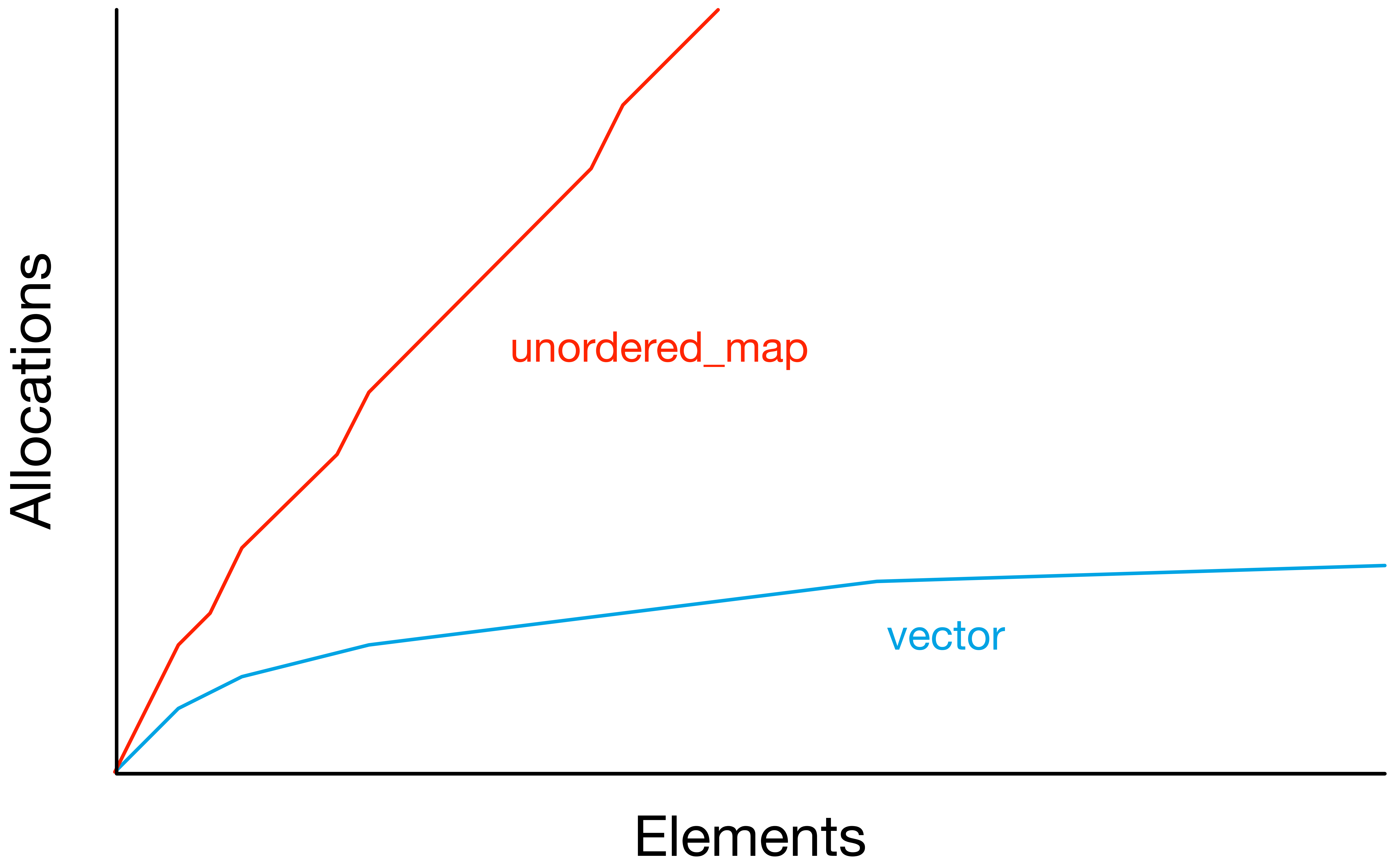
Russian Coat Check Algorithm



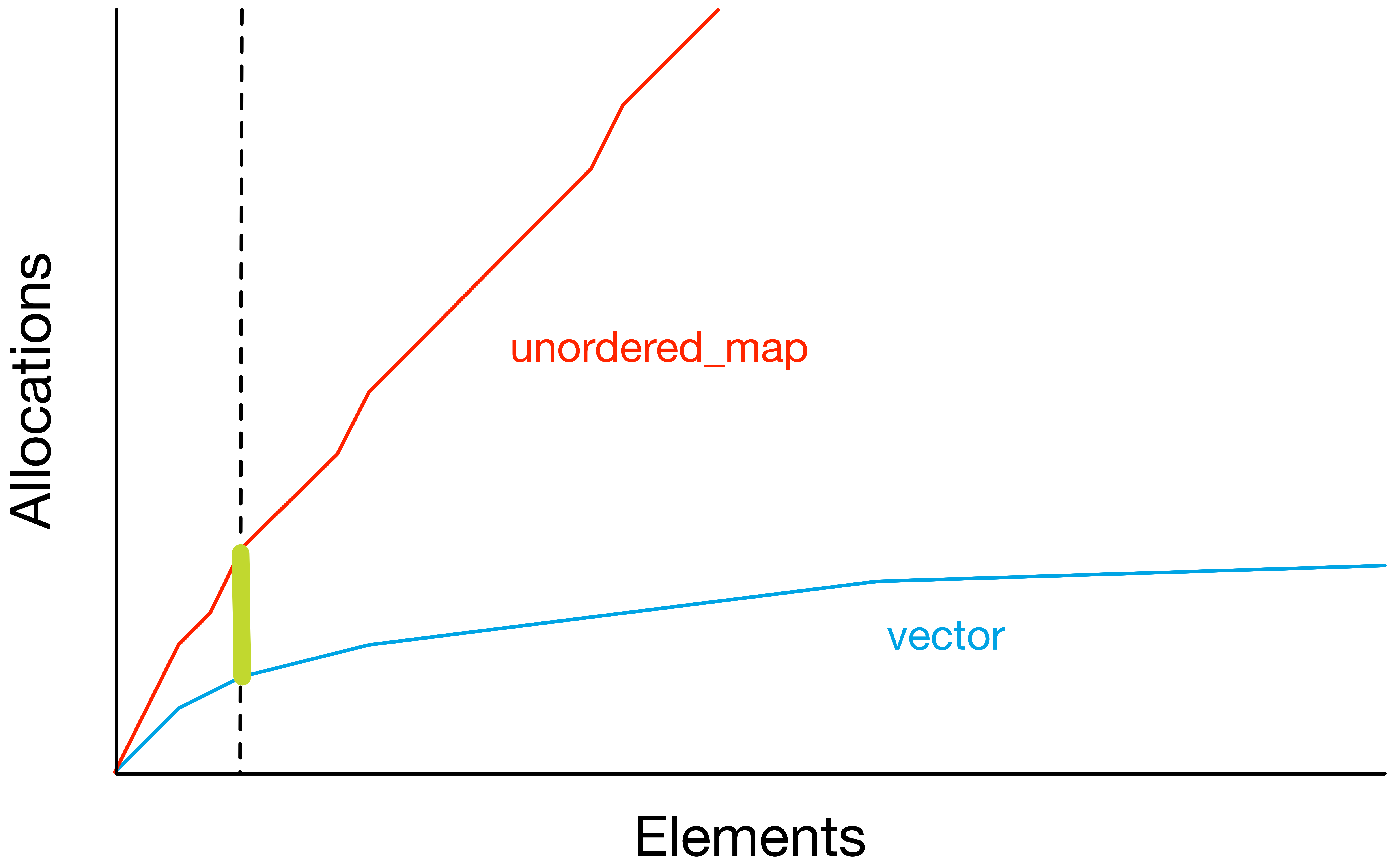
Russian Coat Check Algorithm



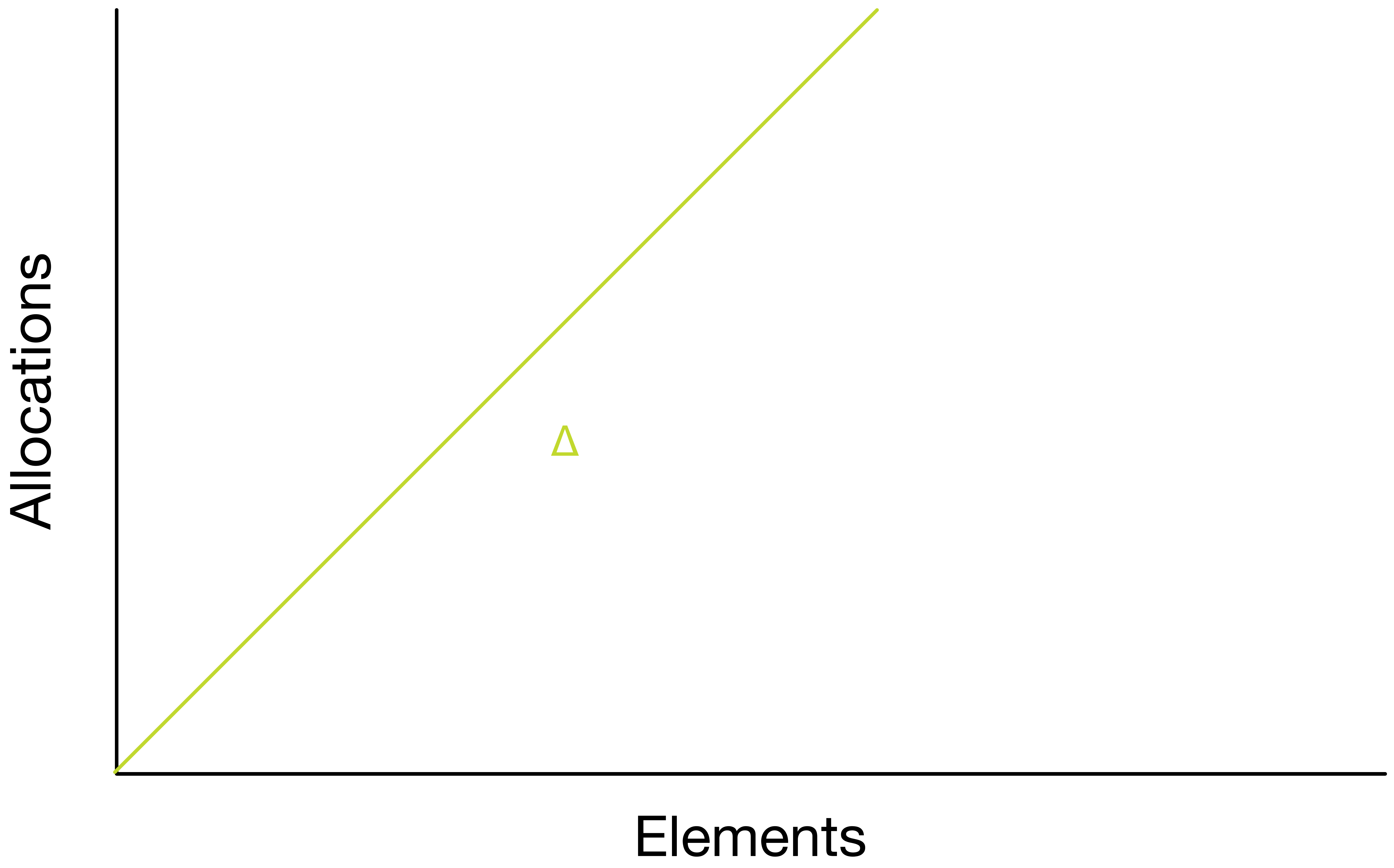
Russian Coat Check Algorithm



Russian Coat Check Algorithm



Russian Coat Check Algorithm



Architecture

- Relationships can be exploited for performance

Architecture

- Relationships can be exploited for performance
  - Understanding the relationship between the cost of operations is important

Goal: No Contradictions

#AdobeRemix
Hiroyuki-Mitsume Takahashi

Double-entry bookkeeping

Double-entry bookkeeping

- Double-entry bookkeeping is an accounting tool for error detection and fraud prevention

Double-entry bookkeeping

- Double-entry bookkeeping is an accounting tool for error detection and fraud prevention
- Relies on the accounting equation

$$assets = liabilities + equity$$

Double-entry bookkeeping

- Double-entry bookkeeping is an accounting tool for error detection and fraud prevention
- Relies on the accounting equation

$$assets = liabilities + equity$$

- An example of equational reasoning

Double-entry bookkeeping

- Double-entry bookkeeping is an accounting tool for error detection and fraud prevention
- Relies on the accounting equation

$$assets = liabilities + equity$$

- An example of equational reasoning
- Pioneered in the 11th century by the Jewish banking community

Double-entry bookkeeping

- Double-entry bookkeeping is an accounting tool for error detection and fraud prevention
- Relies on the accounting equation

$$assets = liabilities + equity$$

- An example of equational reasoning
- Pioneered in the 11th century by the Jewish banking community
  - Likely developed independently in Korea in the same time period

Double-entry bookkeeping

- Double-entry bookkeeping is an accounting tool for error detection and fraud prevention
- Relies on the accounting equation

$$assets = liabilities + equity$$

- An example of equational reasoning
- Pioneered in the 11th century by the Jewish banking community
  - Likely developed independently in Korea in the same time period
- In the 14th century, double-entry bookkeeping was adopted by the Medici bank

Double-entry bookkeeping

- Double-entry bookkeeping is an accounting tool for error detection and fraud prevention
- Relies on the accounting equation

$$assets = liabilities + equity$$

- An example of equational reasoning
- Pioneered in the 11th century by the Jewish banking community
  - Likely developed independently in Korea in the same time period
- In the 14th century, double-entry bookkeeping was adopted by the Medici bank
  - Credited with establishing the Medici bank as reliable and trustworthy

Double-entry bookkeeping

- Double-entry bookkeeping is an accounting tool for error detection and fraud prevention
- Relies on the accounting equation

$$assets = liabilities + equity$$

- An example of equational reasoning
- Pioneered in the 11th century by the Jewish banking community
  - Likely developed independently in Korea in the same time period
- In the 14th century, double-entry bookkeeping was adopted by the Medici bank
  - Credited with establishing the Medici bank as reliable and trustworthy
    - Leading to the rise of one of the most powerful family dynasties in history

Double-entry bookkeeping

- Double-entry bookkeeping is an accounting tool for error detection and fraud prevention
- Relies on the accounting equation

$$assets = liabilities + equity$$

- An example of equational reasoning
- Pioneered in the 11th century by the Jewish banking community
  - Likely developed independently in Korea in the same time period
- In the 14th century, double-entry bookkeeping was adopted by the Medici bank
  - Credited with establishing the Medici bank as reliable and trustworthy
    - Leading to the rise of one of the most powerful family dynasties in history
- Double-entry bookkeeping was codified by Luca Pacioli (the Father of Accounting) in 1494

Luca Pacioli

Double-entry bookkeeping

Double-entry bookkeeping

- Every transaction is entered twice, into at least two separate accounts

Double-entry bookkeeping

- Every transaction is entered twice, into at least two separate accounts
- There are 5 standard accounts, Assets, Capital, Liabilities, Revenues, and Expenses

Double-entry bookkeeping

- Every transaction is entered twice, into at least two separate accounts
- There are 5 standard accounts, Assets, Capital, Liabilities, Revenues, and Expenses
- This ensures the mechanical process of entering a transaction is done in two distinct ways

Double-entry bookkeeping

- Every transaction is entered twice, into at least two separate accounts
- There are 5 standard accounts, Assets, Capital, Liabilities, Revenues, and Expenses
- This ensures the mechanical process of entering a transaction is done in two distinct ways

- If the accounting equation is not satisfied, then we have a *contradiction*

Contradictions

Contradictions

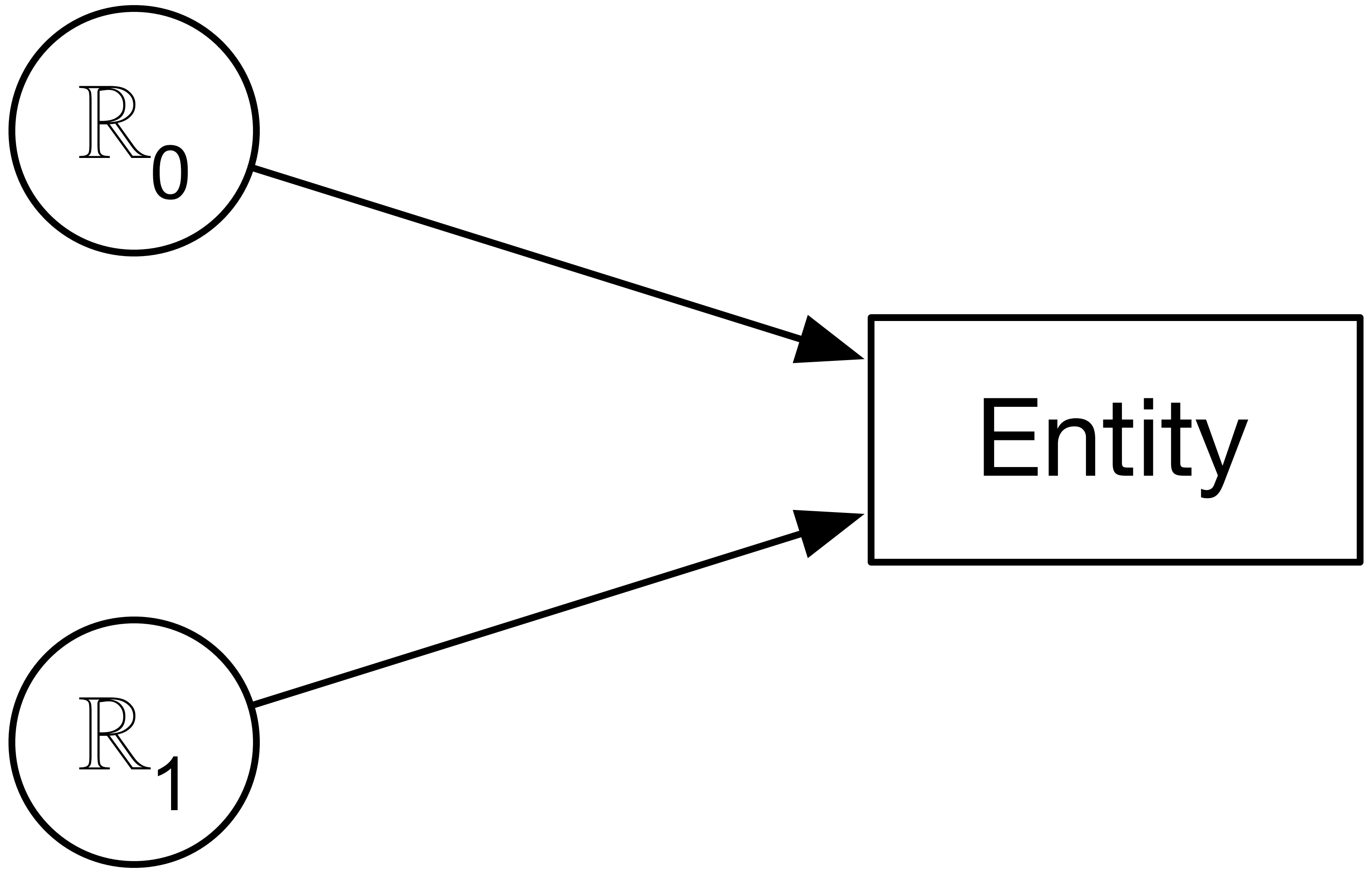
- When two relationships imply the same entity has different values

Contradictions

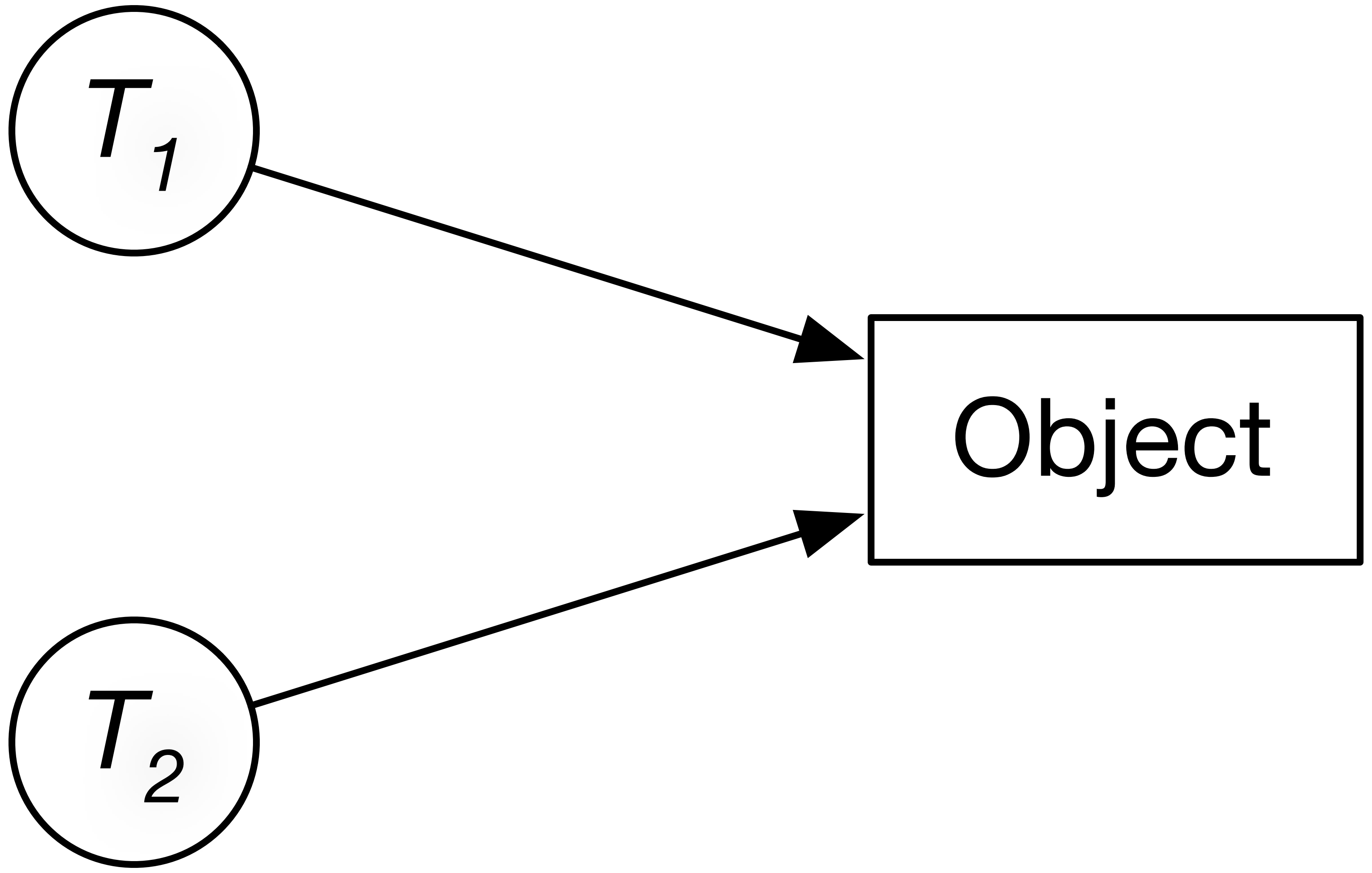
- When two relationships imply the same entity has different values
- Relationships are *consistent* if they imply the same entity has the same value

Contradictions

- When two relationships imply the same entity has different values
- Relationships are *consistent* if they imply the same entity has the same value

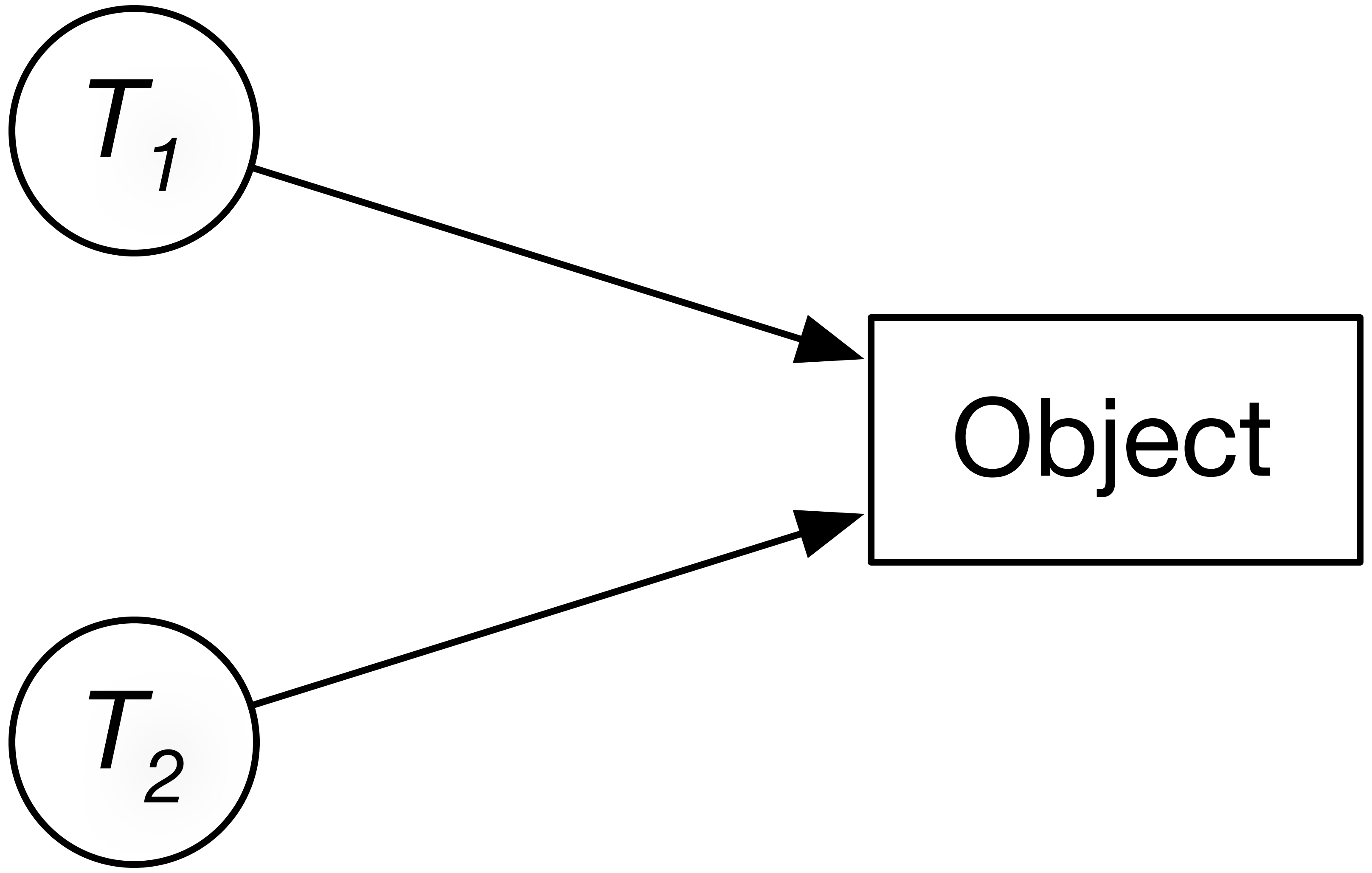


Data Race

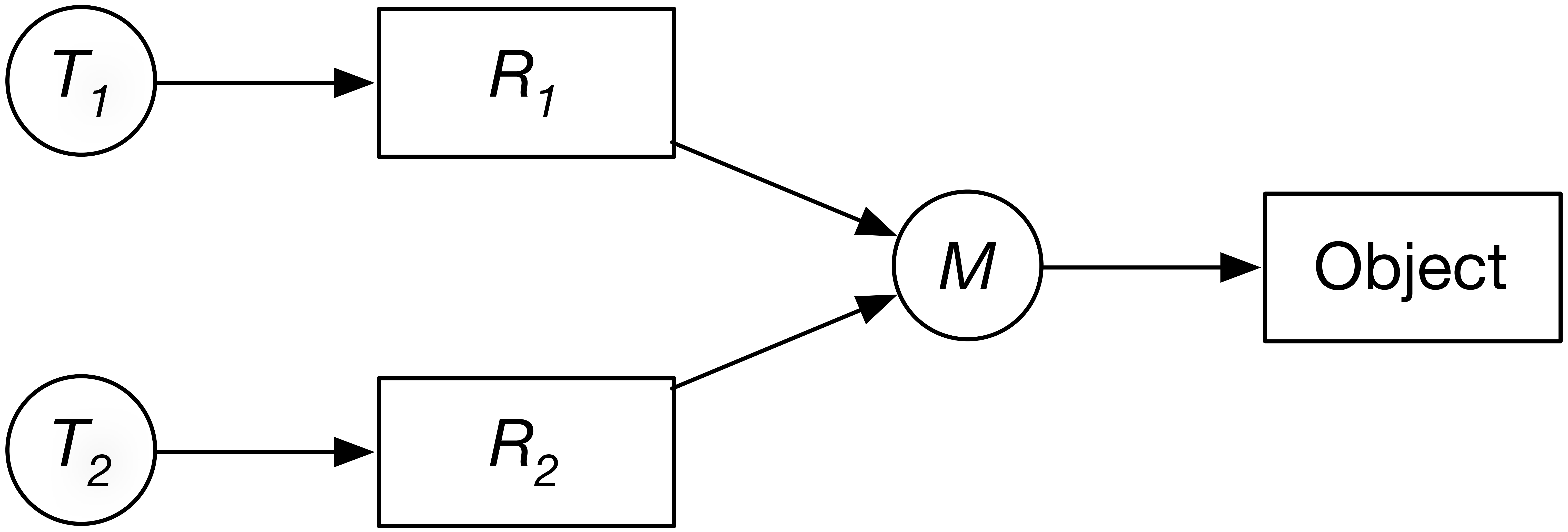


Data Race

- When two or more threads access the same object concurrently and at least one is writing

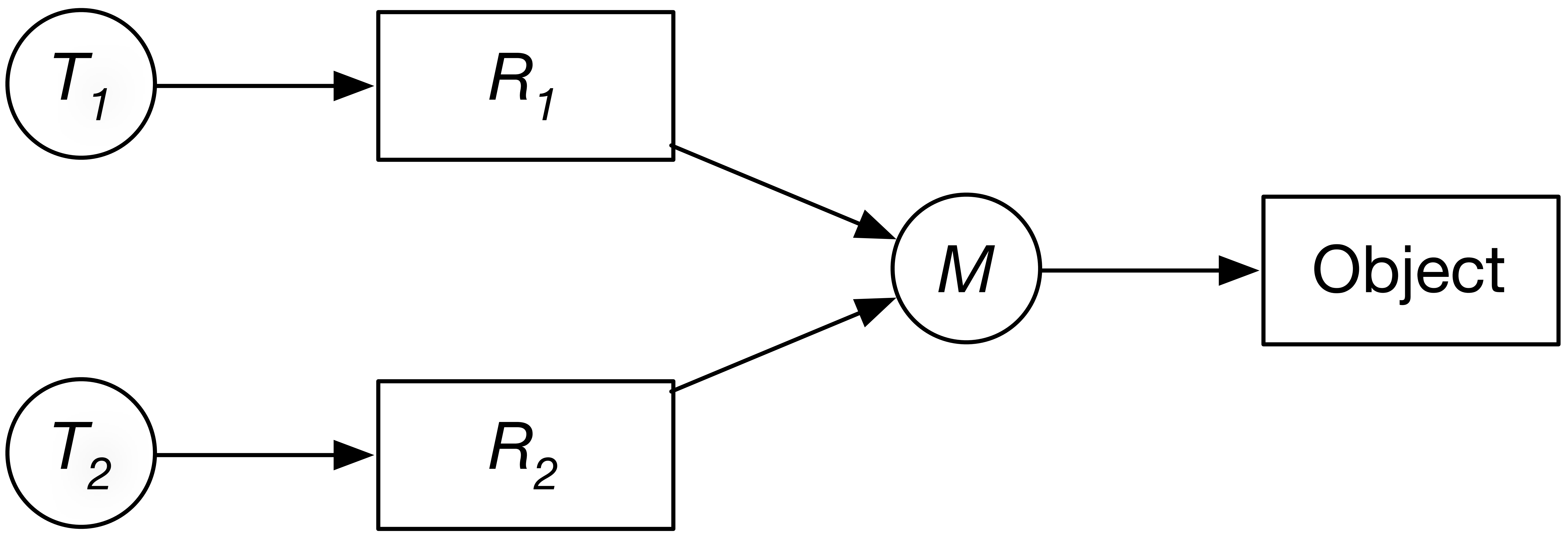


Data Race



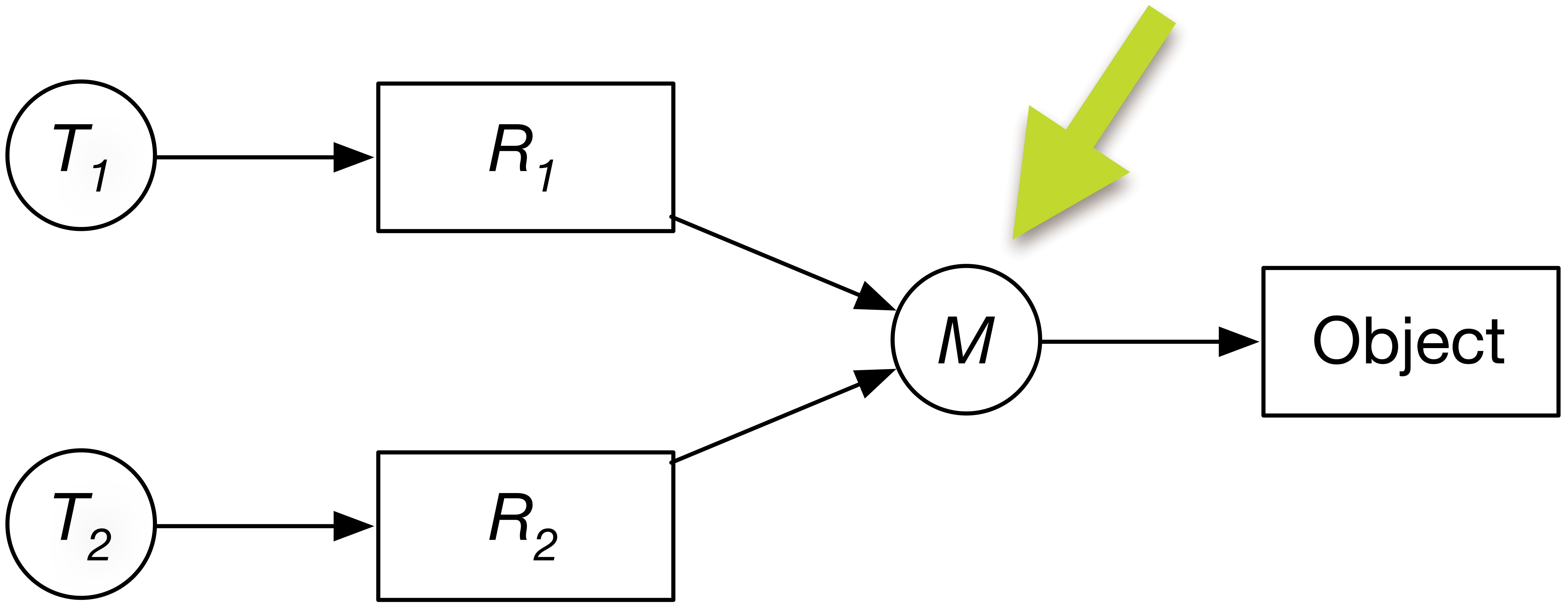
Data Race

- We can resolve the race with a mutex



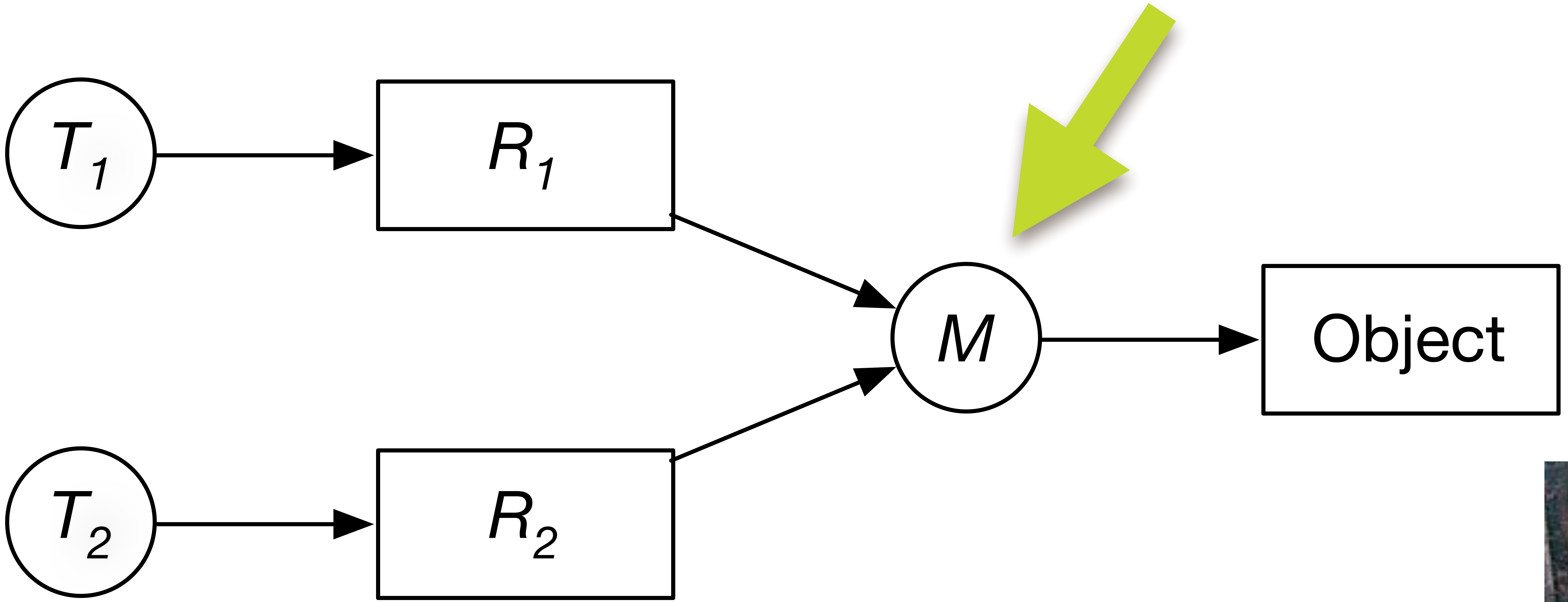
Data Race

- We can resolve the race with a mutex
- But what does it mean?



Data Race

- We can resolve the race with a mutex
- But what does it mean?



No Raw Synchronization Primitives

Null Pointer Dereference

- C++ Specification: dereferencing a null pointer is *undefined behavior*

Null Pointer Dereference

- C++ Specification: dereferencing a null pointer is *undefined behavior*

```
p->member();
```

Null Pointer Dereference

- C++ Specification: dereferencing a null pointer is *undefined behavior*

```
p->member();
```

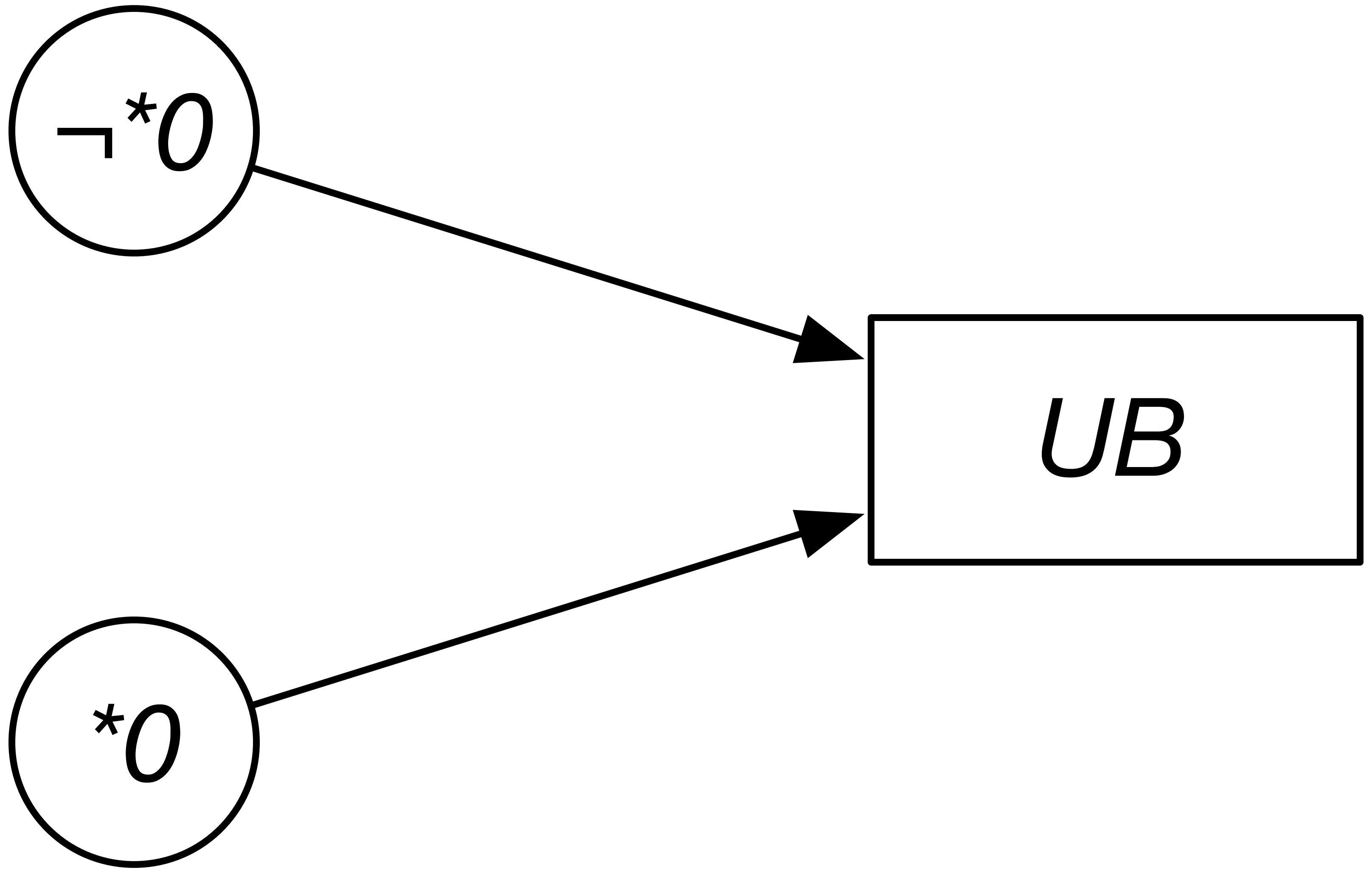
SIGABRT

Null Pointer Dereference

- C++ Specification: dereferencing a null pointer is *undefined behavior*

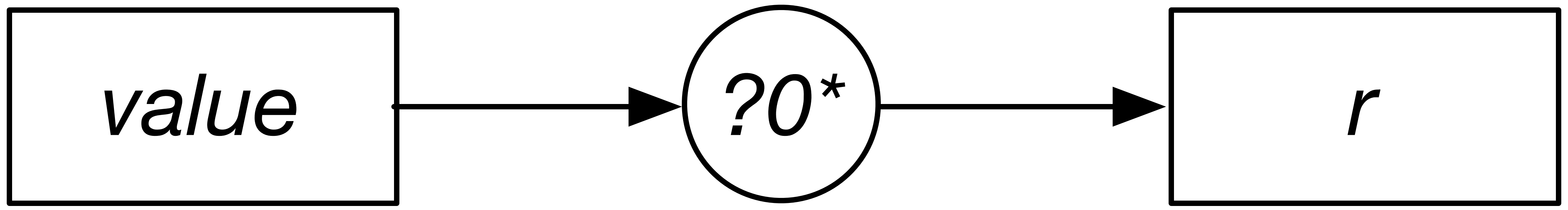
```
p->member();
```

SIGABRT



Null Pointer Dereference

- C++ Specification: dereferencing a null pointer is *undefined behavior*

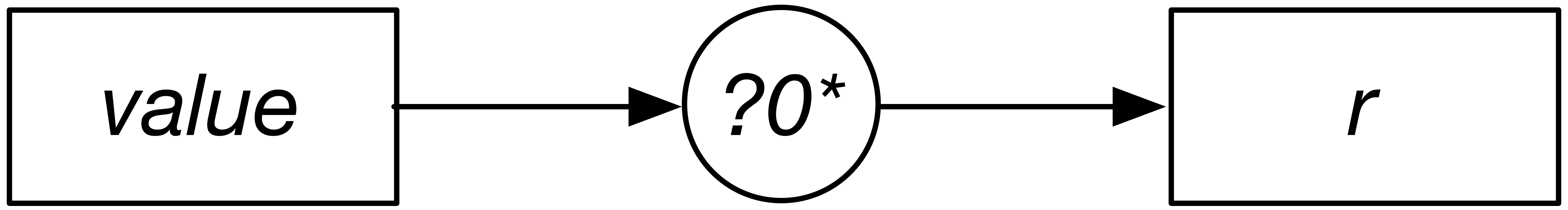


Adobe

Null Pointer Dereference

- C++ Specification: dereferencing a null pointer is *undefined behavior*

```
if (p) p->member();
```



Null Pointers or Optional Objects

Null Pointers or Optional Objects

- The graceful handling of *nothing* as a limit is important

Null Pointers or Optional Objects

- The graceful handling of *nothing* as a limit is important
  - empty ranges, 0, etc.

Null Pointers or Optional Objects

- The graceful handling of *nothing* as a limit is important
  - empty ranges, 0, etc.
- Removing sections of code to avoid a crash is likely only moving the contradiction

Null Pointers or Optional Objects

- The graceful handling of *nothing* as a limit is important
  - empty ranges, 0, etc.
- Removing sections of code to avoid a crash is likely only moving the contradiction

Pro Tip

- Use strong preconditions to move the issue to the caller

Pro Tip

- Use strong preconditions to move the issue to the caller

```
void f(type* p) {
    //...
    if (p) p->member();
    //...
}
```

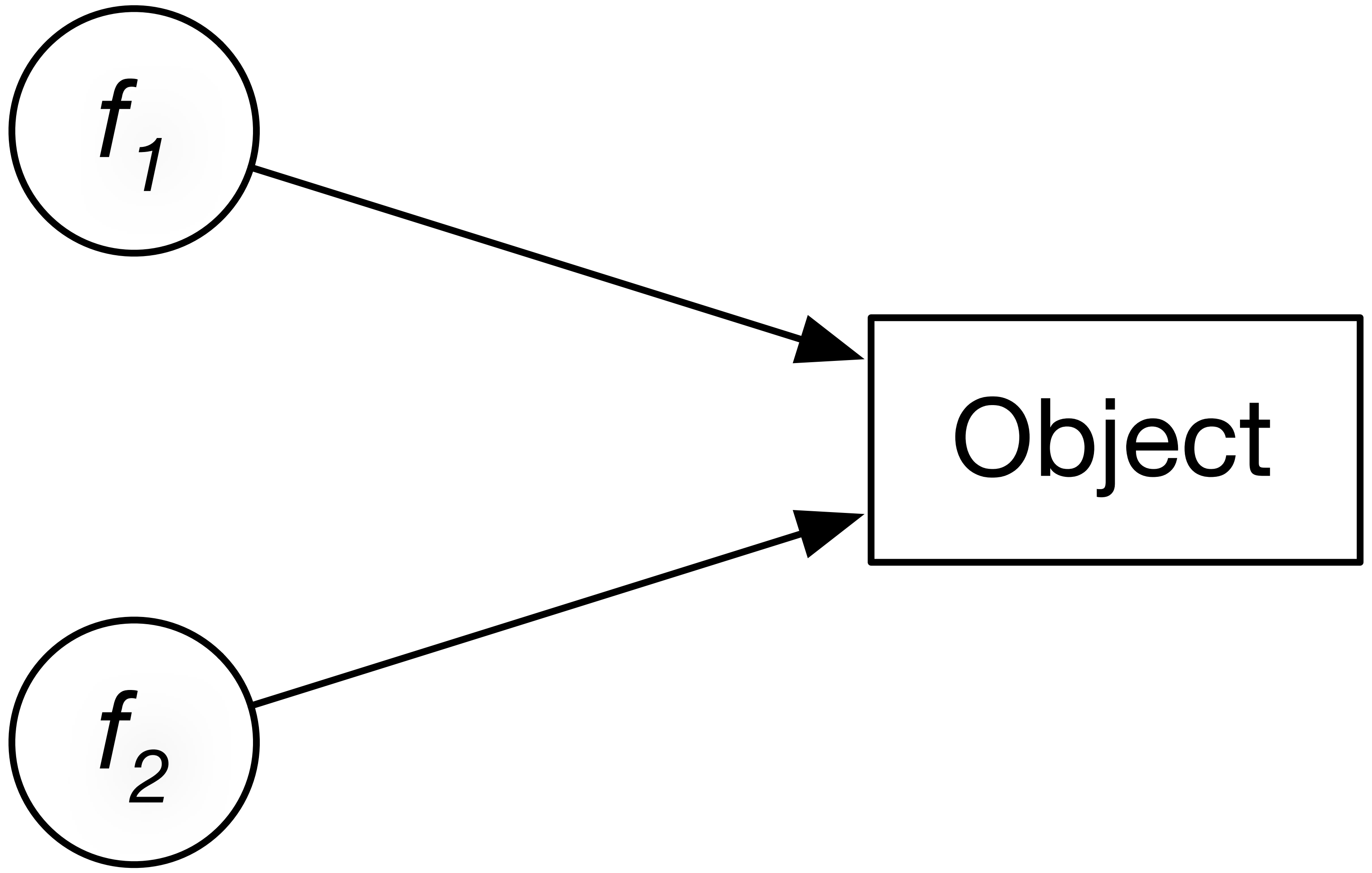
Pro Tip

- Use strong preconditions to move the issue to the caller

```
void f(type& p) {
    //...
    p.member();
    //...
}
```

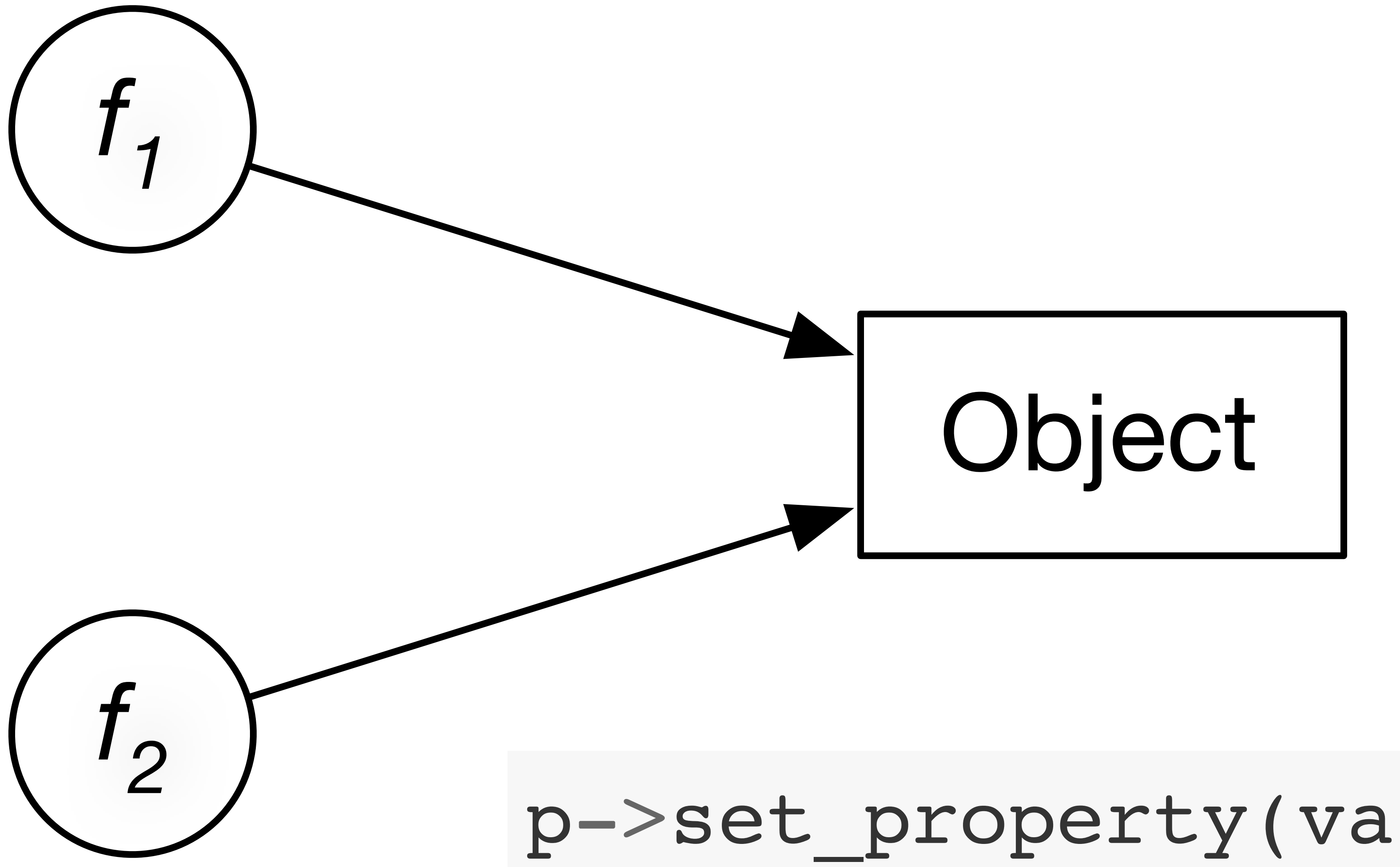
Setting a Property

- Two functions setting the same value through a shared pointer



Setting a Property

- Two functions setting the same value through a shared pointer




```
p->set_property(value);

// Someplace else...
p->set_property(other_value);
```

Setting a Property

Setting a Property

- Possible meanings:

Setting a Property

- Possible meanings:
  - Code is redundant

Adobe

Setting a Property

- Possible meanings:
  - Code is redundant
  - Different aspects of the same relationship, represented in disparate sections of code
    - **`value`** `is a * b when a changes`
    - **`other_value`** `is a * b when b changes`

Setting a Property

- Possible meanings:
  - Code is redundant
  - Different aspects of the same relationship, represented in disparate sections of code
    - `value is a * b when a changes`
    - `other_value is a * b when b changes`
  - Different, mutually exclusive, relationships with non-local control

Setting a Property

- Possible meanings:
  - Code is redundant
  - Different aspects of the same relationship, represented in disparate sections of code
    - **value** is a * b when a changes
    - **other_value** is a * b when b changes
  - Different, mutually exclusive, relationships with non-local control
  - Implied “last in wins” relationship

Setting a Property

- Possible meanings:
  - Code is redundant
  - Different aspects of the same relationship, represented in disparate sections of code
    - **`value`** `is a * b when a changes`
    - **`other_value`** `is a * b when b changes`
  - Different, mutually exclusive, relationships with non-local control
  - Implied “last in wins” relationship
  - An incidental algorithm - property will converge to the correct value

Setting a Property

- Possible meanings:
  - Code is redundant
  - Different aspects of the same relationship, represented in disparate sections of code
    - **`value is a * b when a changes`**
    - **`other_value is a * b when b changes`**
  - Different, mutually exclusive, relationships with non-local control
  - Implied “last in wins” relationship
  - An incidental algorithm - property will converge to the correct value
  - Property is not a simple property but a stream, trigger, or latch

Setting a Property

- Possible meanings:
  - Code is redundant
  - Different aspects of the same relationship, represented in disparate sections of code
    - `value is a * b when a changes`
    - `other_value is a * b when b changes`
  - Different, mutually exclusive, relationships with non-local control
  - Implied “last in wins” relationship
  - An incidental algorithm - property will converge to the correct value
  - Property is not a simple property but a stream, trigger, or latch
  - Or, it is just wrong

Setting a Property

- Possible meanings:
  - Code is redundant
  - Different aspects of the same relationship, represented in disparate sections of code
    - `value is a * b when a changes`
    - `other_value is a * b when b changes`
  - Different, mutually exclusive, relationships with non-local control
  - Implied “last in wins” relationship
  - An incidental algorithm - property will converge to the correct value
  - Property is not a simple property but a stream, trigger, or latch
  - Or, it is just wrong

No Raw Pointers

Play the Game

Play the Game

- Consider the *essential* relationships

Play the Game

- Consider the *essential* relationships
- Learn to see structure

Play the Game

- Consider the *essential* relationships
- Learn to see structure
- Architect code

sean-parent.stlab.cc

photoshopishiring.com

Adobe

Adobe