Goal: Implement Complete & Efficient Types

Sean Parent | Principal Scientist

Background

- Chapter 1: Regular Types
  - Goal: Implement Complete & Efficient Types

- Chapter 2: Algorithms
  - Goal: No Raw Loops
- Chapter 4: Runtime Polymorphism
  - Goal: Shift Polymorphism to Point of Use
- Chapter 5: Concurrency
  - Goal: No Raw Synchronization Primitives

- See *C++ Seasoning*, http://channel9.msdn.com/Events/GoingNative/2013/Cpp-Seasoning

What is a Type?

- An *object* is a representation of an entity as a value in ***memory***
- A *type* is a pattern for storing and modifying objects[1]

[1]*Elements of Programming*, Section 1.3

What is a Type?

- An *object* is a representation of an entity as a value in ***memory***
- A *type* is a pattern for storing and modifying objects[1]

type is the interpretation of the bits

[1]*Elements of Programming*, Section 1.3

What is a Type?

- An *object* is a representation of an entity as a value in ***memory***
- A *type* is a pattern for storing and modifying objects[1]

type is the interpretation of the bits

structure and basis operations

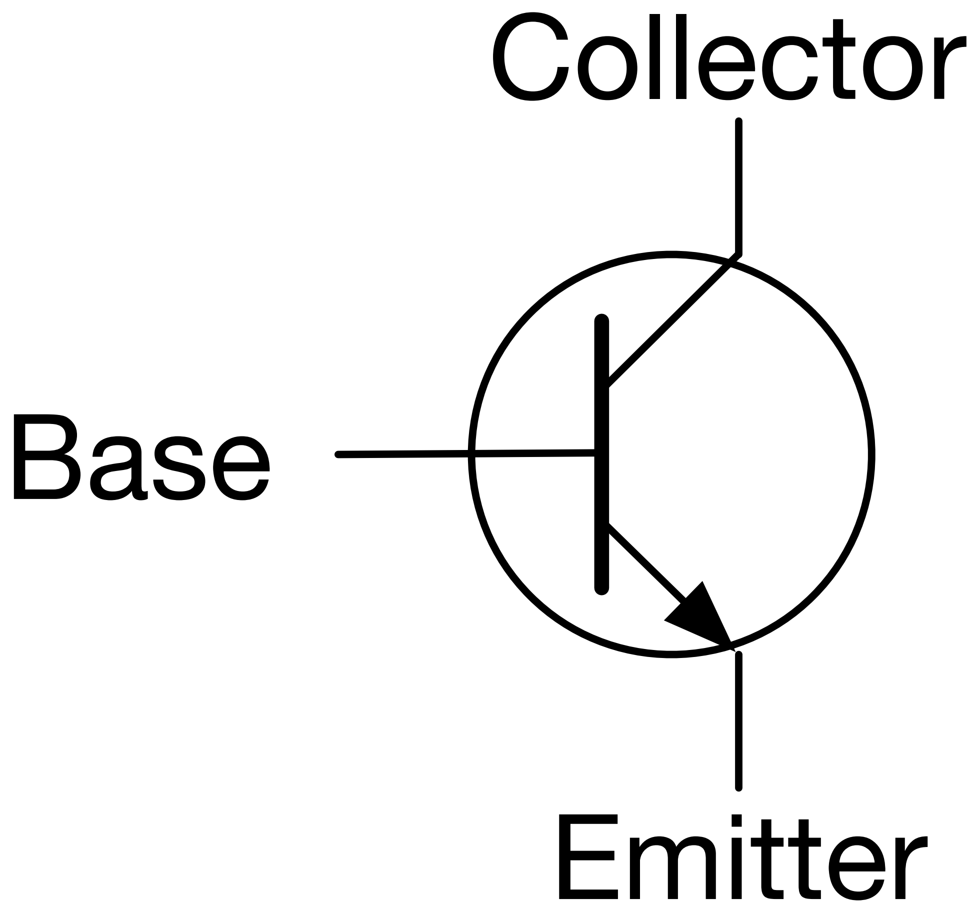
[1]*Elements of Programming*, Section 1.3

Objects are Physical Entities

- Physicality allows us to apply Philosophy, Logic, Mathematics, and Physics to Computer Science

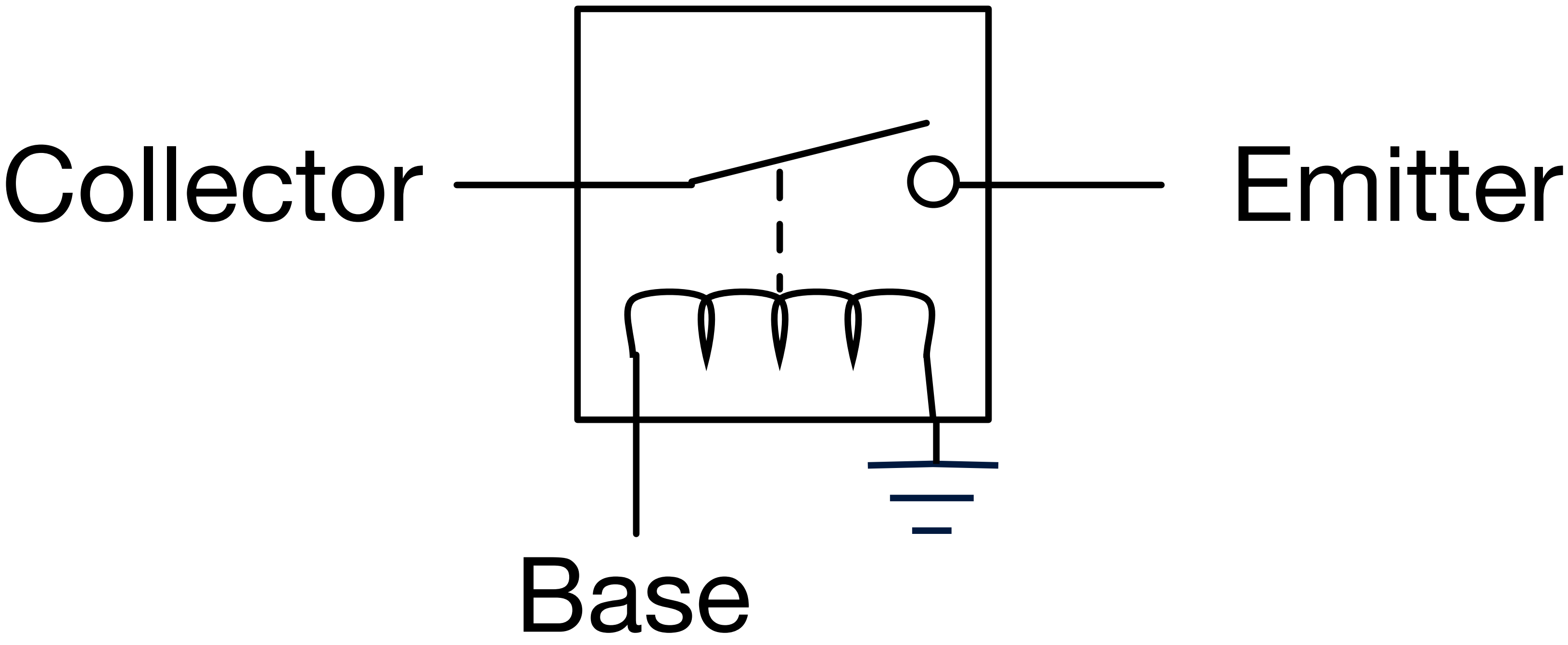
Objects are Physical Entities

- Transistors are solid-state switches



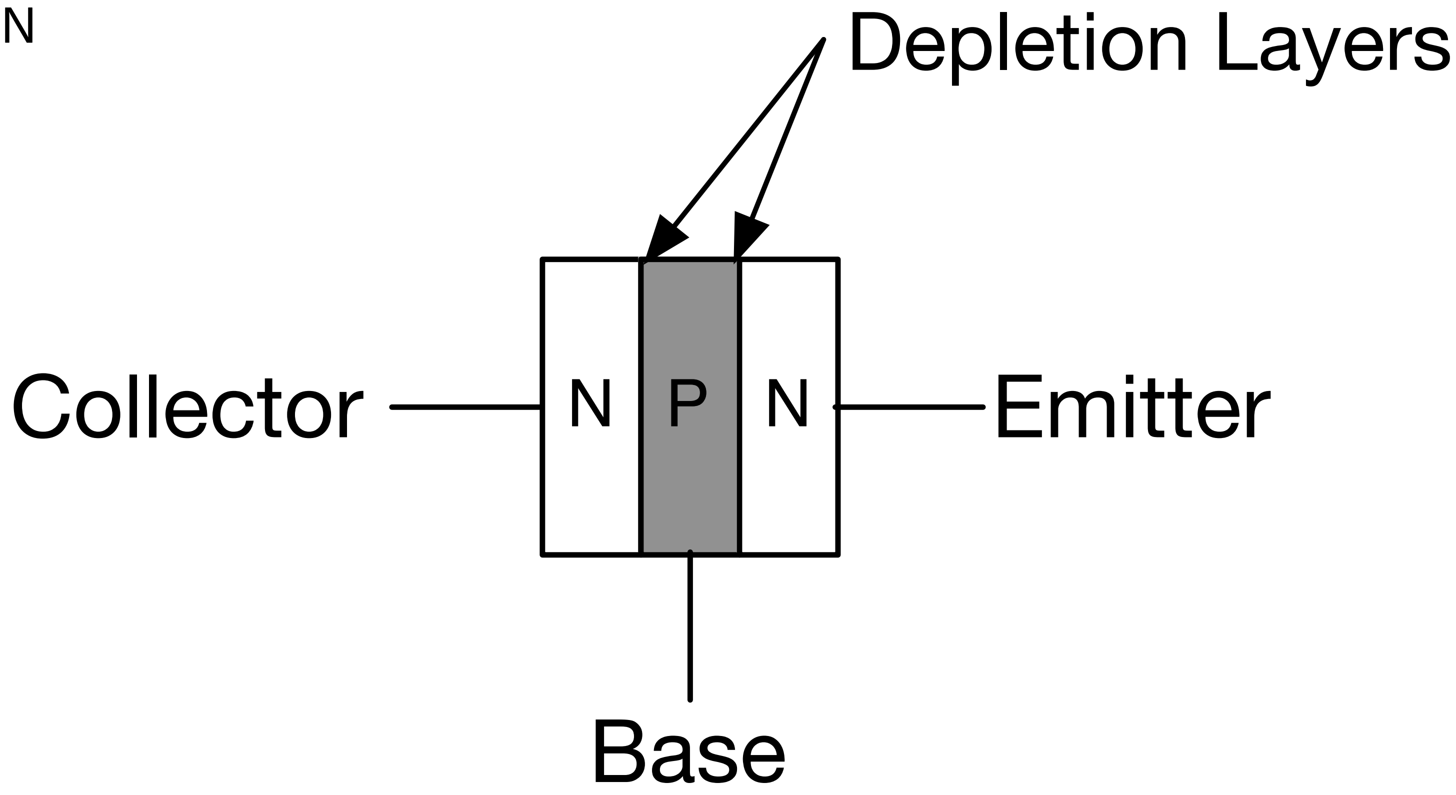
Objects are Physical Entities

- Just as a relay is an electrically controlled switch



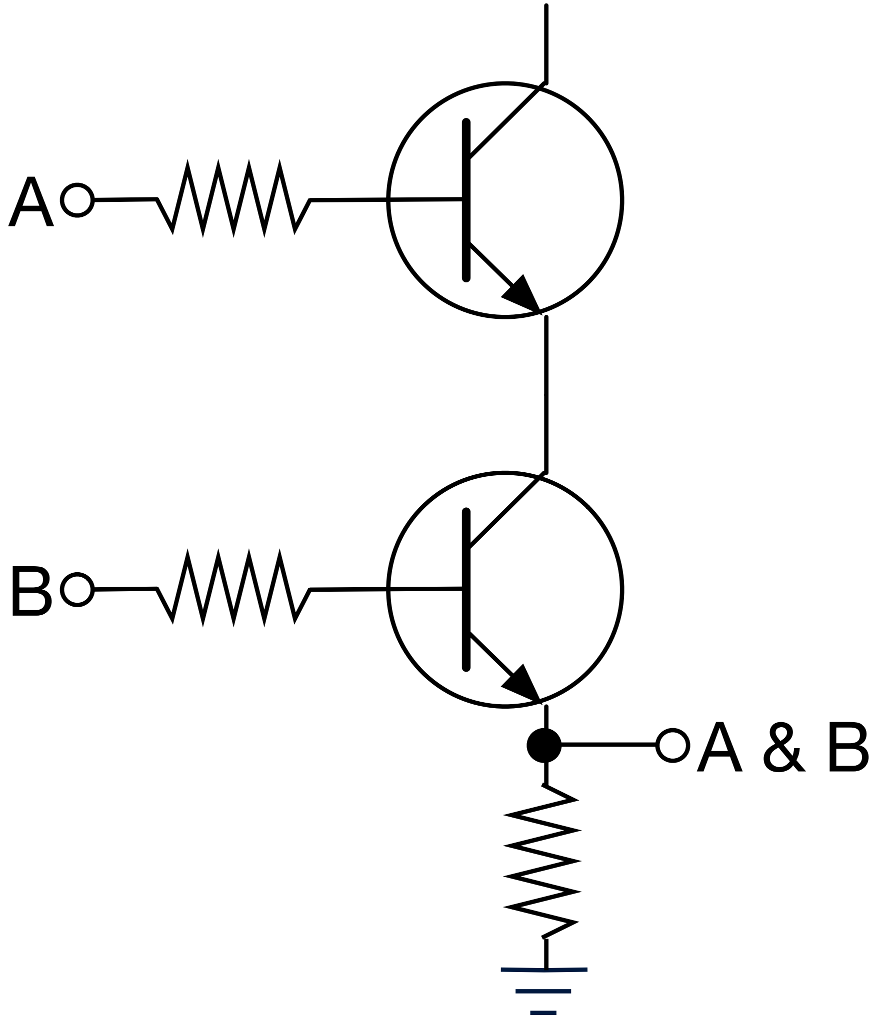
Objects are Physical Entities

- Silicon + Boron = P
- Silicon + Phosphorus = N



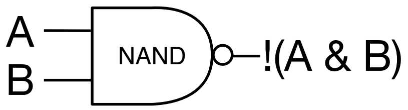
Objects are Physical Entities

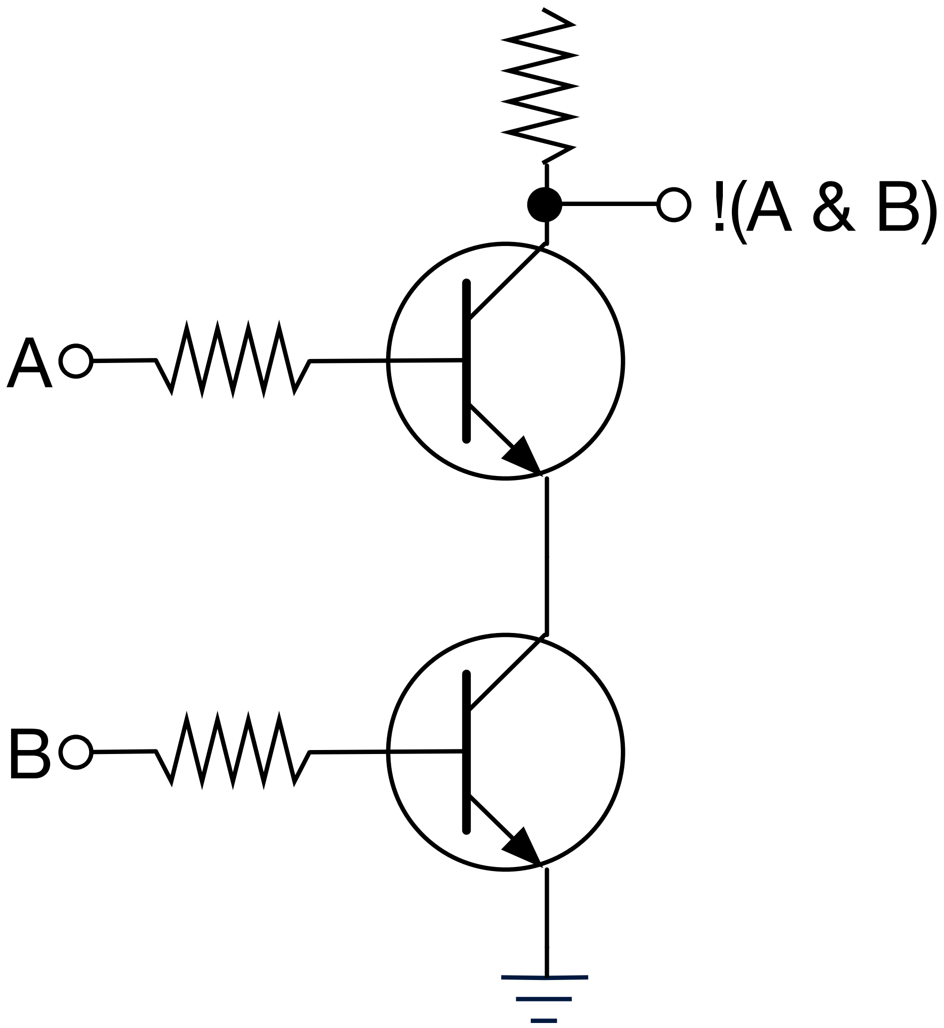
- An AND Gate



Objects are Physical Entities

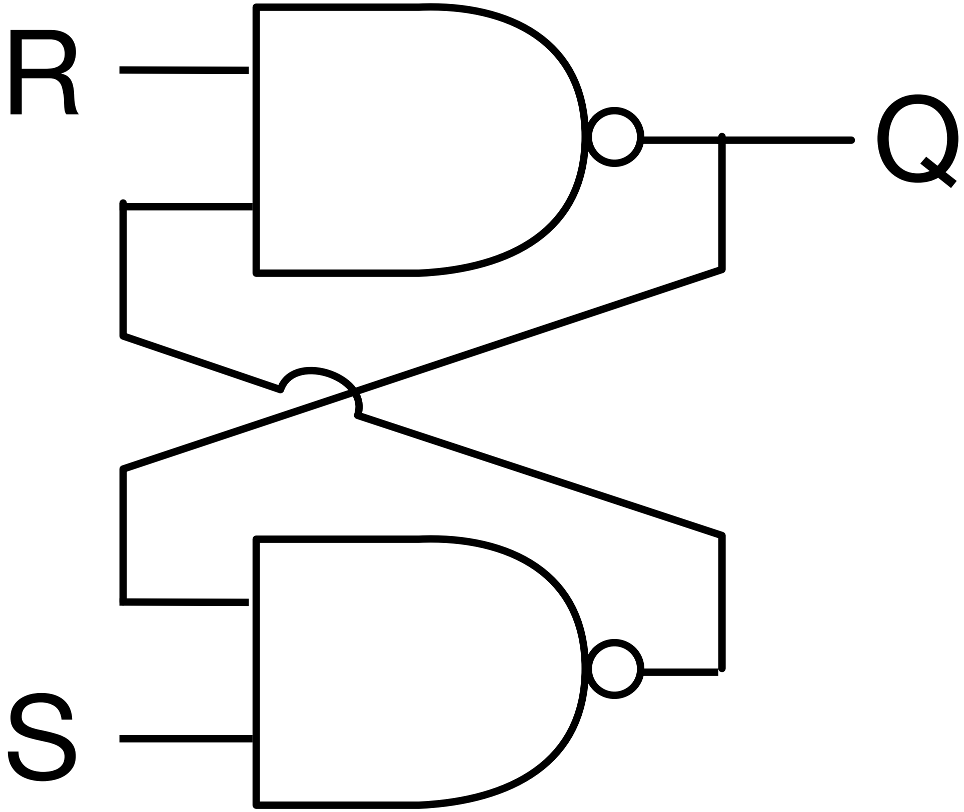
- A NAND gate





Objects are Physical Entities

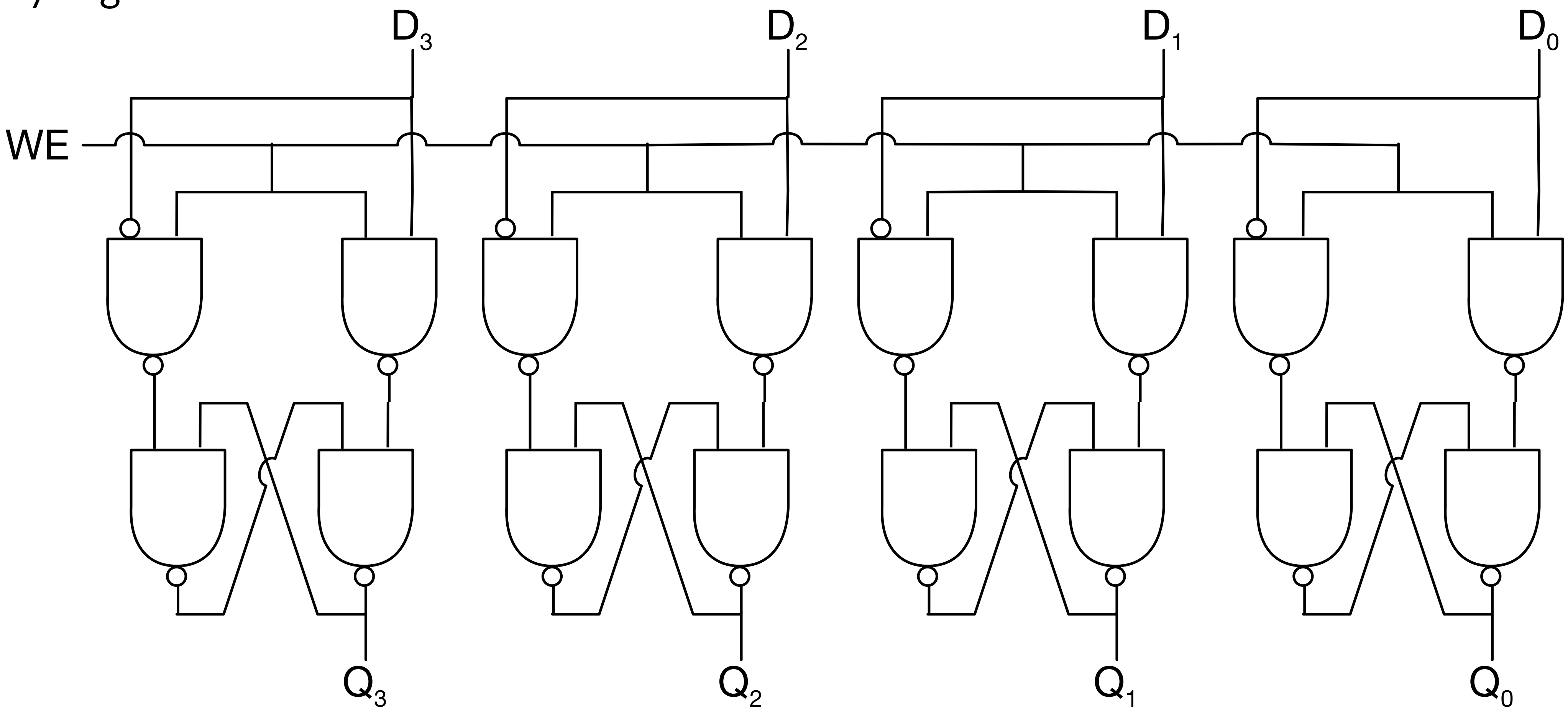
- Sequential Logic RS Latch



R	S	Q
0	1	1
1	0	0
1	1	Q'

Objects are Physical Entities

- Memory Register



Objects are Physical Entities

- With some additional control logic a collection of registers form a memory space
- Switches -> Gates -> Sequential Circuits -> Memory -> Processor

- Switches can be built in any number of ways (relay, vacuum tube, levers, gears, marbles, dominos…)

"There is a set of procedures whose inclusion in the computational basis of a type lets us place objects in *data structures* and use algorithms to *copy objects* from one data structure to another. We call types having such a basis regular, since their use guarantees regularity of behavior and, therefore, interoperability."

Elements of Programming Section 1.5

Equality

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

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

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

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

Equality

- If the representation of a value as an object is not unique, then the complexity of implementing equality can be arbitrarily complex
- If the representation is unique, the complexity is O(areaof(A)) worse case

- The expected complexity of equality is O(areaof(A)), when the complexity is significantly greater implement equality as representation equality

Representational Equality => Value Equality

Copy and Assignment of Objects

- A copy of an object is a new object equal to the operand
- Assigning to an object makes the object equal to the operand without modifying the operand

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

Copy and Equality

- Two objects of the same type with the same representation are equal
- It follows that any object is *copyable* by copying the representation

Copy and Equality

- Two objects of the same type with the same representation are equal
- It follows that any object is *copyable* by copying the representation

All types are copyable

Copy and Equality

- Two objects of the same type with the same representation are equal
- It follows that any object is *copyable* by copying the representation

All types are copyable *

Completeness & Efficiency

Completeness & Efficiency

- A type is *complete* if the set of provided basis operations allow us to construct and operate on any valid, representable value
- A type is *efficient* if the set of basis operations allow for any valid operation to be performed in the most efficient way possible for the chosen representation

Completeness & Efficiency

- A type is *complete* if the set of provided basis operations allow us to construct and operate on any valid, representable value
- A type is *efficient* if the set of basis operations allow for any valid operation to be performed in the most efficient way possible for the chosen representation

- By simply making all data members public, you provide, by definition, an efficient basis for the type

Completeness & Efficiency

- A type is *complete* if the set of provided basis operations allow us to construct and operate on any valid, representable value
- A type is *efficient* if the set of basis operations allow for any valid operation to be performed in the most efficient way possible for the chosen representation

- By simply making all data members public, you provide, by definition, an efficient basis for the type
- However, you may fail to protect the invariants of the type, making the approach *unsafe*

Safety and Validity

Safety and Validity

- A *safe* operation is one that when, the preconditions are satisfied, leaves an object in a *valid* state, containing a representable value
- An unsafe operation may leave an object in an invalid state, requiring additional operations to restore the object invariants

Safety and Validity

- A *safe* operation is one that when, the preconditions are satisfied, leaves an object in a *valid* state, containing a representable value
- An unsafe operation may leave an object in an invalid state, requiring additional operations to restore the object invariants

- Sometimes unsafe operation are required to provide an efficient basis

Copy and Equality

- * If the extent of a type is not know either statically or encoded as part of the type, then equality and copy cannot be implemented as a function of only the type
- Such a type is *constructionally incomplete*

Copy and Equality

- * If the extent of a type is not know either statically or encoded as part of the type, then equality and copy cannot be implemented as a function of only the type
- Such a type is *constructionally incomplete*

```
class incomplete_int_array {
    unique_ptr<int[]> data_;
public:
    explicit incomplete_int_array(size_t size) : data_(new int[size]()) { }
};
```

Copy and Equality

- If any value of an object can be distinguished through the public interface then equality can be implemented as a non-member, non-friend function
- Such a type is *equationally complete*

equationally complete => constructionally complete

Copy and Equality

- Copy and equality are *composed* properties

Two objects are equal iff only if their *essential* parts are equal
An object is copyable iff the *essential* parts are copyable

Copy and Equality

- An *essential* part of an object is a part that contributes to its value and is not simply part of the representation

Equality of Functions

Equality of Functions

- Two functions are equal if given the same argument they return the same value

Equality of Functions

- Two functions are equal if given the same argument they return the same value
- In C, we fall back to a representational equality through identity

```
assert(log2f != log10f);
```

Equality of Functions

- Two functions are equal if given the same argument they return the same value
- In C, we fall back to a representational equality through identity

```
assert(log2f != log10f);
```

- Unfortunately in C++ function objects (including lambdas) do not define equality

Equality of Functions

- Two functions are equal if given the same argument they return the same value
- In C, we fall back to a representational equality through identity

```
assert(log2f != log10f);
```

- Unfortunately in C++ function objects (including lambdas) do not define equality
- Functions objects are copyable and copies are equal, however they are equationally incomplete

Copy and Equality

- Expected complexity of copy is O(areaof(T)) worst case

```
class int_array {
    size_t size_;
    unique_ptr<int[]> data_;
public:
    explicit int_array(size_t size) : size_(size), data_(new int[size]()) { }
    int_array(const int_array& x) : size_(x.size_), data_(new int[x.size_])
    { copy(x.data_.get(), x.data_.get() + x.size_, data_.get()); }

    int_array& operator=(const int_array& x); // **

    const int* begin() const { return data_.get(); }
    const int* end() const { return data_.get() + size_; }
    size_t size() const { return size_; }
};

bool operator==(const int_array& x, const int_array& y)
{ return (x.size() == y.size()) && equal(begin(x), end(x), begin(y)); }
```

Relationships

Relationships

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

Relationships

- As soon as we have two objects we have implicit relationships
  - A memory space is a container object

Relationships

- As soon as we have two objects we have implicit relationships
  - A memory space is a container object

- When an object is copied, any relationship that object was involved in is either *severed* or *maintained*

Reified Relationships

- A reified relationship is a relationship represented by an object
  - As an object, a reified relationship is copyable and equality comparable
  - When a reified relationship is copied, the relationship is either maintained, severed, or *invalidated*

  - We may choose not to implement copy for relationships

Managing Relationships

- Chapter 2: Algorithms
  - Goal: No Raw Loops
  - Managing positional relationships
- Chapter 4: Runtime Polymorphism
  - Goal: Shift Polymorphism to Point of Use
  - Managing owned relationship by transforming to whole-part relationship
- Chapter 5: Concurrency
  - Goal: No Raw Synchronization Primitives
  - Managing relationships between objects and the thread of execution

Whole-Part Relationship

- A part which is referred to indirectly is a *remote part*
- An object with remote parts can be *moved*
  - Moving an object only requires storage for the local parts
  - Any reified relationship can be maintained and *moved*

Whole-Part Relationship

- A part which is referred to indirectly is a *remote part*
- An object with remote parts can be *moved*
  - Moving an object only requires storage for the local parts
  - Any reified relationship can be maintained and *moved* ***

Move

- Move an object by moving all the local essential parts and moving the relationship to any remote essential part

$$a = b, a \rightharpoonup c \Rightarrow c = b$$

(move is value preserving)

Move

- Complexity of move is O(sizeof(T))

```
int_array(int_array&& x) noexcept = default;
int_array& operator=(int_array&& x) noexcept = default;
```

Move

```
class int_array {
    size_t size_;
    unique_ptr<int[]> data_;
public:
    explicit int_array(size_t size) : size_(size), data_(new int[size]()) { }
    int_array(const int_array& x) : size_(x.size_), data_(new int[x.size_])
    { copy(x.data_.get(), x.data_.get() + x.size_, data_.get()); }

    int_array(int_array&& x) noexcept = default;
    int_array& operator=(int_array&& x) noexcept = default;

    int_array& operator=(const int_array& x);  // **

    const int* begin() const { return data_.get(); }
    const int* end() const { return data_.get() + size_; }
    size_t size() const { return size_; }
};
```

Move

Move

- A moved from object is *partially formed*
  - assigned to
  - destructible

Move

- A moved from object is *partially formed*
  - assigned to
  - destructible

- A moved from object does not represent a value

Move

- A moved from object is *partially formed*
  - assigned to
  - destructible

- A moved from object does not represent a value
- Move is an unsafe operation

**Assignment

- Copy and Move provide transactional assignment

```
int_array& operator=(const int_array& x)
{ int_array tmp = x; *this = move(tmp); return *this; }
```

***I Lied

***I Lied

- Any reified relationship can be maintained and *moved*

***I Lied

- Any reified relationship can be maintained and *moved*
    - Unless the relations is a part-whole relationship

***I Lied

- Any reified relationship can be maintained and *moved*
  - Unless the relations is a part-whole relationship

- Don’t invert the whole-part relationship

***I Lied

- Any reified relationship can be maintained and *moved*
  - Unless the relations is a part-whole relationship

- Don’t invert the whole-part relationship
- Or understand that you must stay within the same whole

Move Efficiency

- C++ Move is *not* efficient

```
int_array(int_array& x, unsafe) : size_(x.size_), data_(x.data_.get()) { }
```

Move Efficiency

- C++ Move is *not* efficient

Move Efficiency

- C++ Move is *not* efficient

```
template <typename T>
void move_unsafe(T& x, void* raw) { new (raw) T(x, unsafe()); }

template <typename T>
void move_unsafe(void* raw, T& x) { new (&x) T(*static_cast<T*>(raw), unsafe()); }

void swap(int_array& x, int_array& y)
{
    aligned_storage<sizeof(int_array)>::type tmp;

    move_unsafe(x, &tmp);
    move_unsafe(y, &x);
    move_unsafe(&tmp, y);
}
```

Other operations on regular types

- Default Construction
- Representations Ordering
- Serialization
- Hash
- Area

Chapter Conclusions

- Understanding the physical nature of objects provides a framework for thinking about objects and types

- Careful consideration of providing efficient basis operations is important to reuse
- Sometimes the most efficient basis operations are unsafe

Adobe