Poll Everywhere

pollev.com/naomila

Ok, we're pretty much done with C++. What was your favorite part?

(No sarcasm pls, I'm being earnest)

A. Printing things with the << operator
B. Destructors!
C. Constness all over the place
D. Smart pointers taking care of my bizness
E. STL containers just holdin' stuff
F. Overloading all the things << + () -= * ! !
G. none of these i hated it
H. none of these(it was something else and i promise i will email naomi what it was, naomila@uw.edu)

CSE333 Systems Programming

C++ Conversions and Casting
Sneak Peak: Networking

Instructors:

Naomi Alterman

Teaching Assistants:

Ann Baturytski	Barbora Buzkova	Mendel Carroll
Camden Harris	Hannah Hempstead	Rishabh Jain
Irene Lau	Jonathan Nister	Advay Patil
Aviel Wood	Angela Wu	Jennifer Xu

Administrivia

- 👏work 👏on 👏hw3 👏this 👏weekend 👏

Lecture Outline

- **C++ Conversions**
- C++ Casting
- Sneak Peak: Networking!

- Reference: *C++ Primer* §4.11.3, 19.2.1

Implicit Conversion

- The compiler tries to infer some kinds of conversions
  - When types are not equal and you don't specify an explicit cast, the compiler looks for an acceptable implicit conversion

```
void Bar(std::string x);

void Foo() {
  int x = 5.7;  // conversion, float -> int
  char c = x;   // conversion, int -> char
  Bar("hi");    // conversion, (const char*) -> string
}
```

Sneaky Implicit Conversions

- (`const char*`) to `string` conversion?
  - If a class has a constructor with a single parameter, the compiler will exploit it to perform implicit conversions
  - At most, one user-defined implicit conversion will happen
    - Can do `int` → `Foo`, but not `int` → `Foo` → `Baz`

```
class Foo {
 public:
  Foo(int xi) : x(xi) { }
  int x;
};

int Bar(Foo f) {
  return f.x;
}

int main(int argc, char** argv) {
  return Bar(5);  // equivalent to return Bar(Foo(5));
}
```

Avoiding Sneaky Implicits

- Declare one-argument constructors as `explicit` if you want to disable them from being used as an implicit conversion path
  - Usually a good idea

```
class Foo {
 public:
  explicit Foo(int xi) : x(xi) { }
  int x;
};

int Bar(Foo f) {
  return f.x;
}

int main(int argc, char** argv) {
  return Bar(5);  // compiler error
}
```

Lecture Outline

- C++ Conversions
- **C++ Casting**
- Sneak Peak: Networking!

- Reference: *C++ Primer* §4.11.3, 19.2.1

Explicit Casting in C

- Simple syntax: `lhs = (new_type) rhs;`
- Used to:
  - Convert between pointers of arbitrary type
    - Doesn't change the data, but treats it differently
  - Forcibly convert one primitive type to another
    - Actually changes the bit-level representation

- You *can* still use C-style casting in C++, but you *shouldn't*
  - C++ gives us tools to make your *intent* much more clear

Casting in C++

- C++ provides an alternative casting style that is more informative:
  - `static_cast<to_type>(expression)`
  - `dynamic_cast<to_type>(expression)`
  - `const_cast<to_type>(expression)`
  - `reinterpret_cast<to_type>(expression)`

- Always use these in C++ code
  - Intent is clearer
  - Easier to find in code via searching

static_cast

- `static_cast` can convert:
  - Pointers to classes **of related type**
    - Compiler error if classes are not related
    - Dangerous to cast *down* a class hierarchy
  - Casting between `void*` and `T*`
  - Non-pointer conversion
    - *e.g.*, `float` to `int`
- `static_cast` is checked at compile time

staticcast.cc

```
class A {
 public:
  int x;
};

class B {
 public:
  float x;
};

class C : public B {
 public:
  char x;
};
```

```
void Foo() {
  B b; C c;

  // compiler error
  A* aptr = static_cast<A*>(&b);
  // OK
  B* bptr = static_cast<B*>(&c);
  // compiles, but dangerous
  C* cptr = static_cast<C*>(&b);
}
```

dynamic_cast

- dynamic_cast can convert:
  - Pointers to classes **of related type**
  - References to classes **of related type**
- dynamic_cast is checked at both compile time and run time
  - Casts between unrelated classes fail at compile time
  - Casts from base to derived fail at runtime if the pointed-to object is not the derived type

dynamiccast.cc

```
class Base {
 public:
  virtual void Foo() { }
  float x;
};

class Der1 : public Base {
 public:
  char x;
};
```

```
void Bar() {
  Base b; Der1 d;

  // OK (run-time check passes)
  Base* bptr = dynamic_cast<Base*>(&d);
  assert(bptr != nullptr);

  // OK (run-time check passes)
  Der1* dptr = dynamic_cast<Der1*>(bptr);
  assert(dptr != nullptr);

  // Run-time check fails, returns nullptr
  bptr = &b;
  dptr = dynamic_cast<Der1*>(bptr);
  assert(dptr != nullptr);
}
```

const_cast

- const_cast adds or strips const-ness
  - Dangerous (!)
  - ***Doesn't*** allow you to modify const variables (writing to un-"const"ed variables results in undefined behavior
  - The *only* time this really makes sense to use is when you're passing data to a C API that doesn't specify constness in its types but otherwise assures it

```
#include "old-c-library.h"
// gives us:
// int FANCY_80s_HASH(char *, int);
int CalculateHash(const char* buf, int size) {
  return FANCY_80s_HASH(const_cast<char *>(buf), size);
}
int main(int argc, const char** argv) {
  cout << CalculateHash(argv[1], strlen(argv[1]));
  return EXIT_SUCCESS;
}
```

`reinterpret_cast`

- `reinterpret_cast` casts between *incompatible* types
  - Low-level reinterpretation of the bit pattern
  - *e.g.*, storing a pointer in an `int`, or vice-versa
    - Works as long as the integral type is "wide" enough
  - Converting between incompatible pointers
    - Dangerous (**!**)
    - This is used (carefully) in hw3
    - Comes up in networking code (because it's an API from the 70s)
  - Use any other C++ cast if you can!

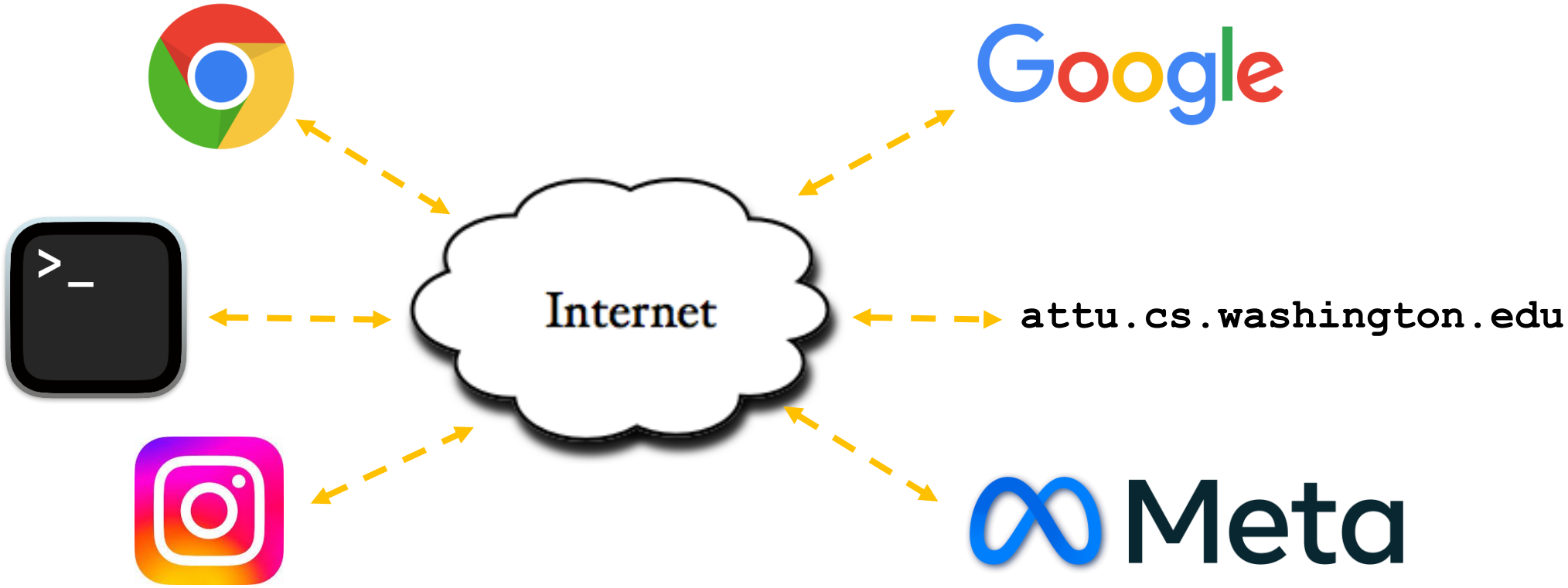
Casting Style Considerations

- From the "Casting" and "Run-Time Type Information (RTTI)" sections of the Google C++ Style Guide:
  - When the logic of a program guarantees that a given instance of a base class is, in fact, an instance of a particular derived class, then a `dynamic_cast` may be used freely on the object.
    - Usually one can use a `static_cast` as an alternative in such situations
  - Only use `reinterpret_cast` if you know what you are doing and you understand the aliasing issues
    - For *unsafe conversions* of pointer types to and from integer and other pointer types, including `void*`

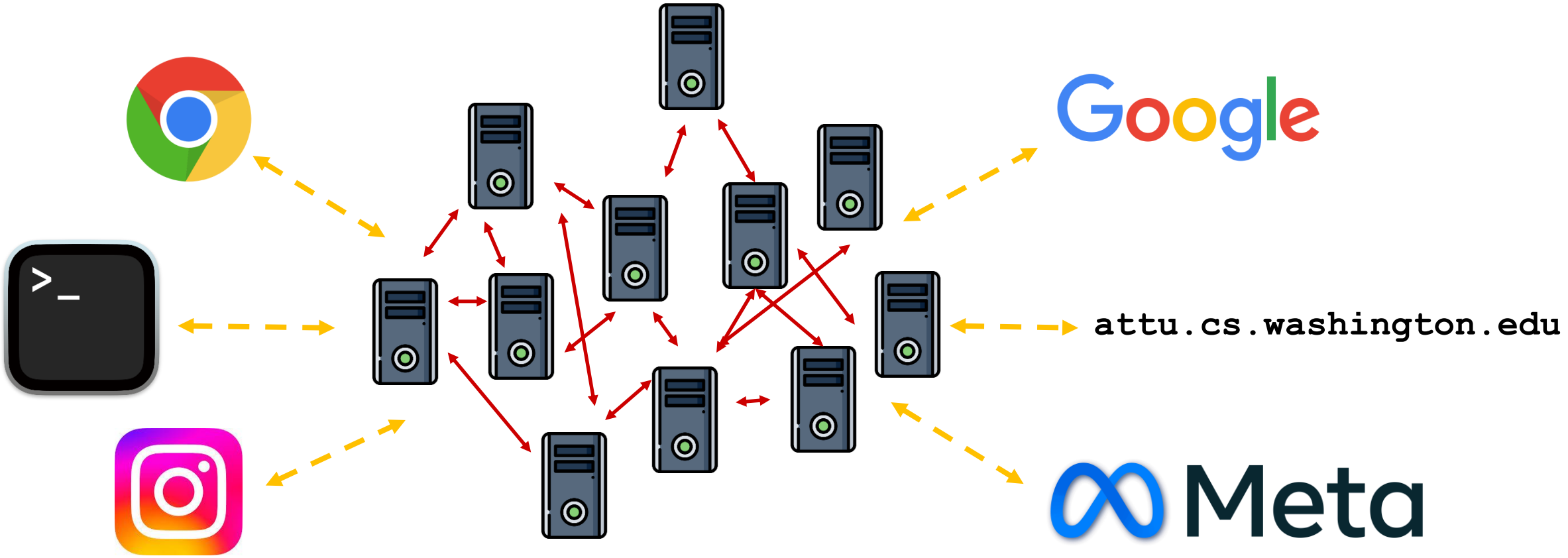
Lecture Outline

- C++ Conversions
- C++ Casting
- **Sneak Peak: Networking!**

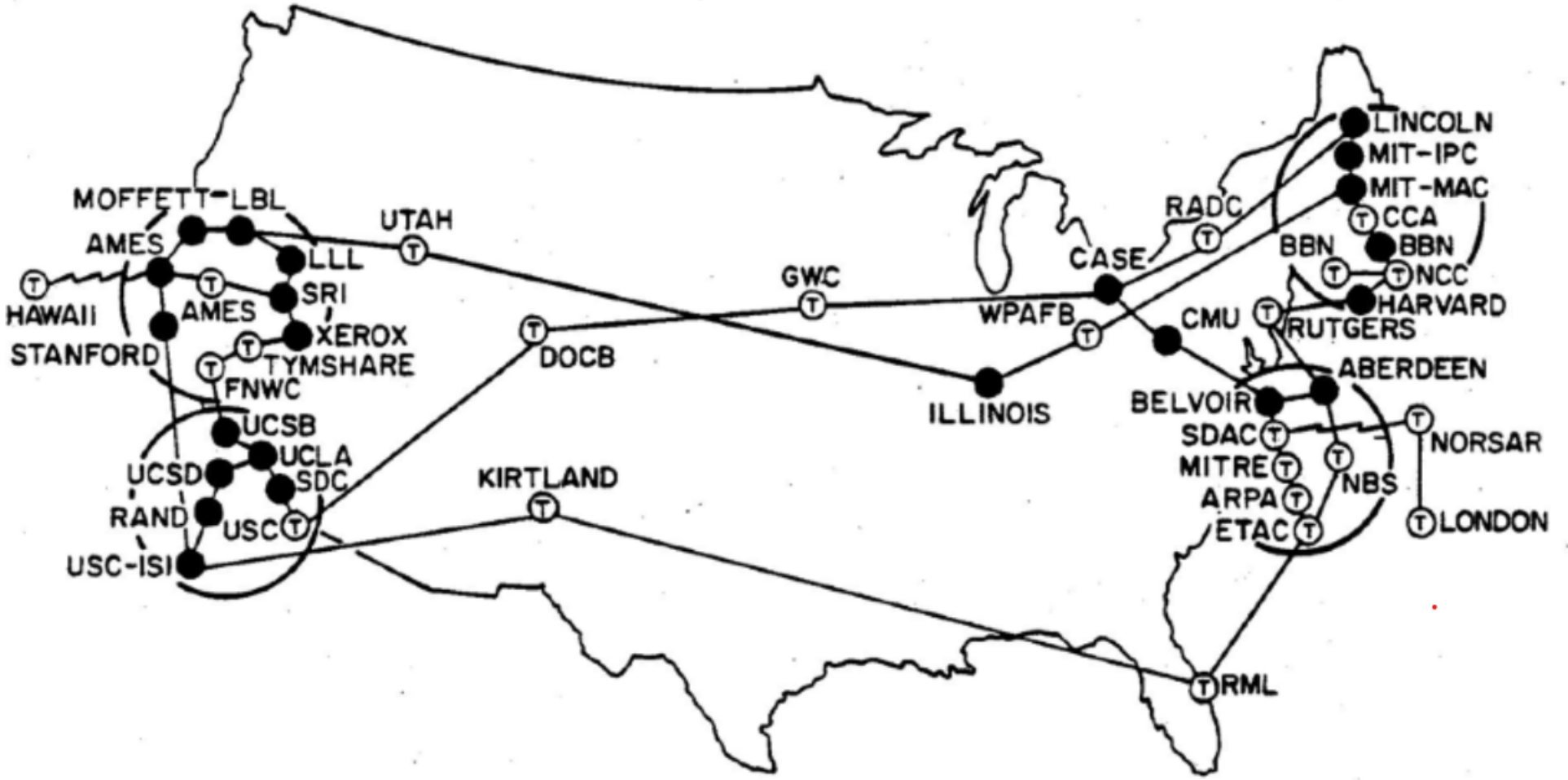
Networks From 10,000 ft



Networks From 10,000 ft (with really good eyes)

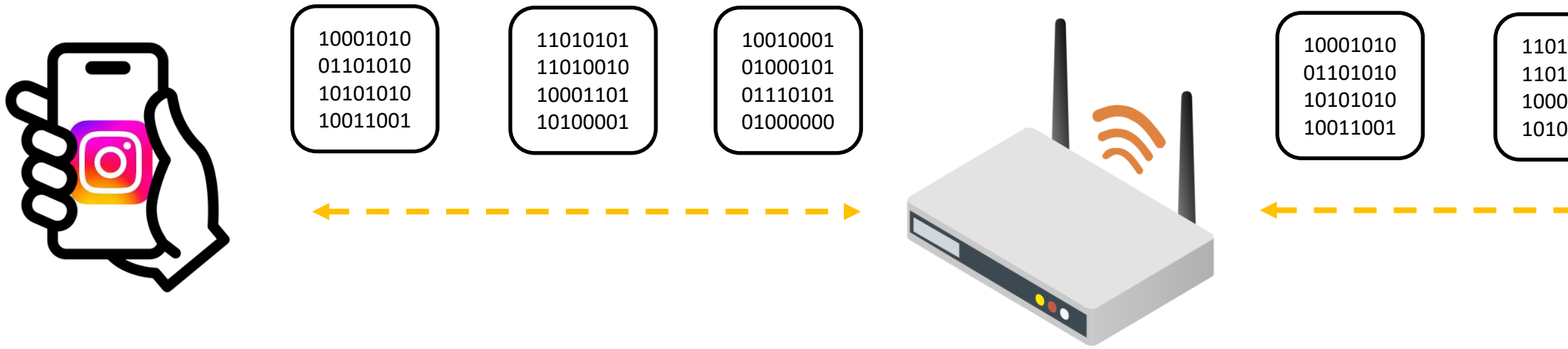


Networks From 10,000 ft (and a time machine)



https://en.wikipedia.org/wiki/ARPANET#/media/File:Arpanet_in_the_1970s.png

Networks From 1 ft

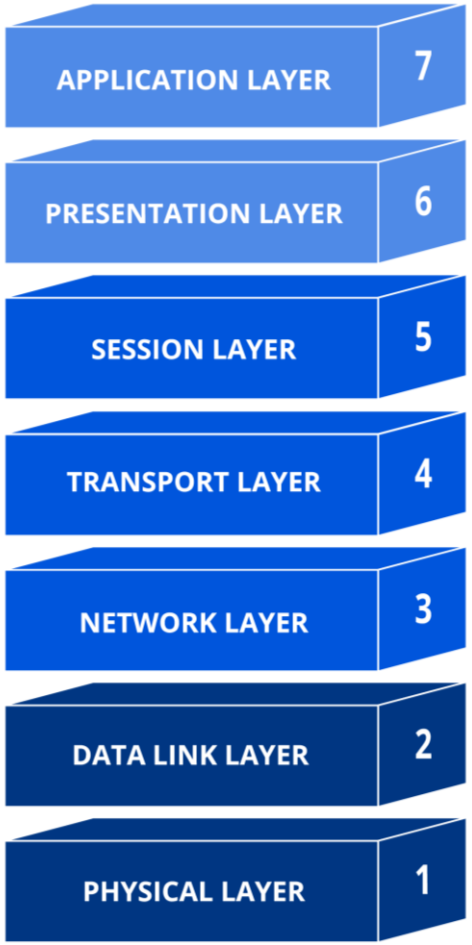


We send data from one software system to another by packaging it into **data packets** and then asking the **operating system** to transport them for us.

What's in those packets?

- Layers…
  - Upon layers…
    - Upon layers…
      - upon layers…
        - » upon layers…
          - ▨ upon layers…
            - ◊ …upon layers!

The OSI model



- The software (and hardware) that gets your packets from one side of the internet to the other is organized into "**layers**"
  - At each step of the way, each layer is present and doing some work to get your bytes to the next hop

The OSI model