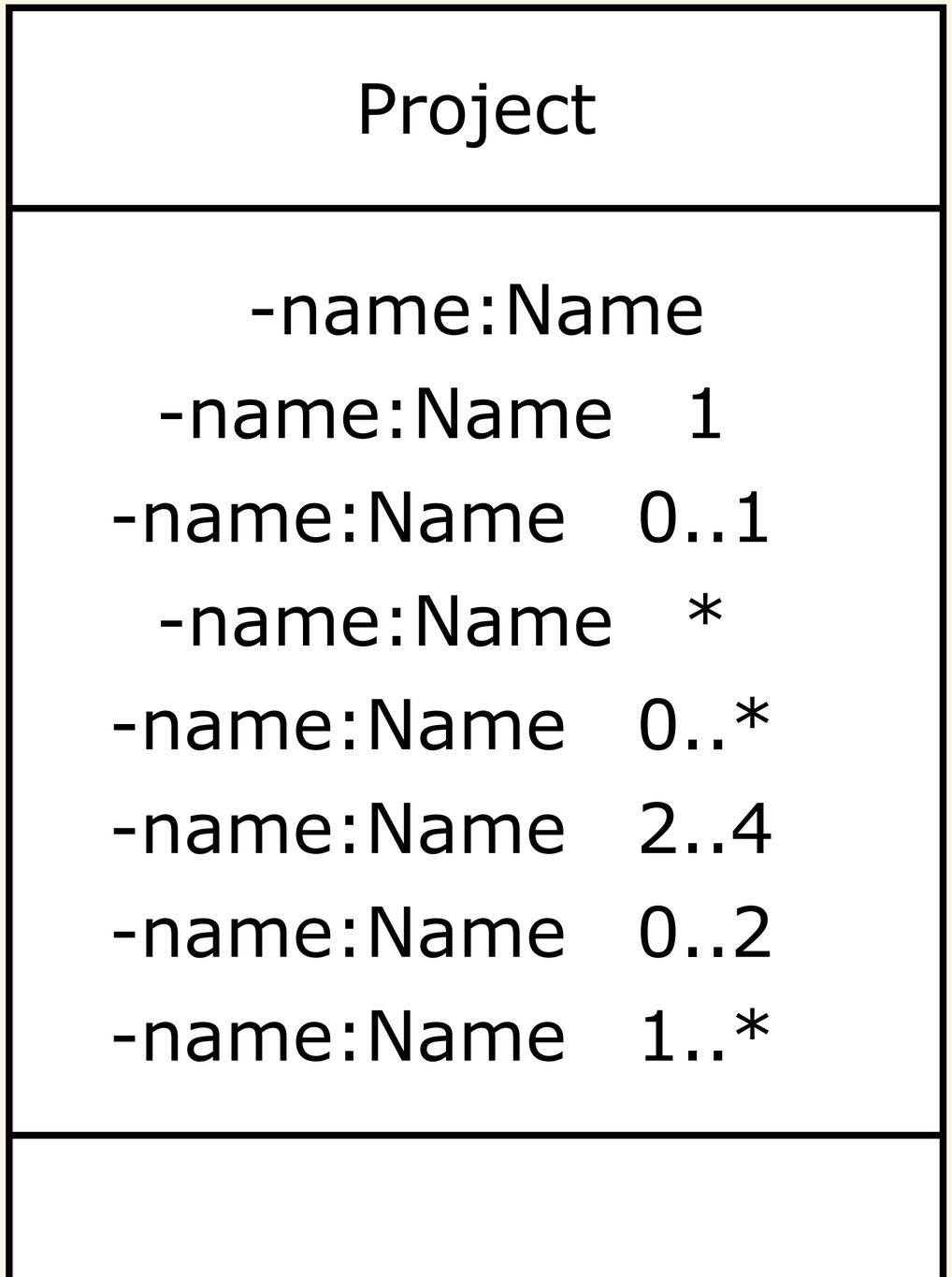
Object-Oriented Programming

UML Multiplicity

Michael L. Collard, Ph.D.

Department of Computer Science, The University of Akron

UML Multiplicity



UML Multiplicity Categories

	Mandatory	Optional
Single Value	`[1]`	`[0..1]`
Multivalued	`[1..*]` `[2..4]`	`[*]` `[0..*]` `[0..2]`

Mandatory & Optional Multivalued

- Multiplicity: `[*]` `[0..2]` `[0..*]` `[1..*]` `[2..4]`
- Containers:

  `std::vector<int>`

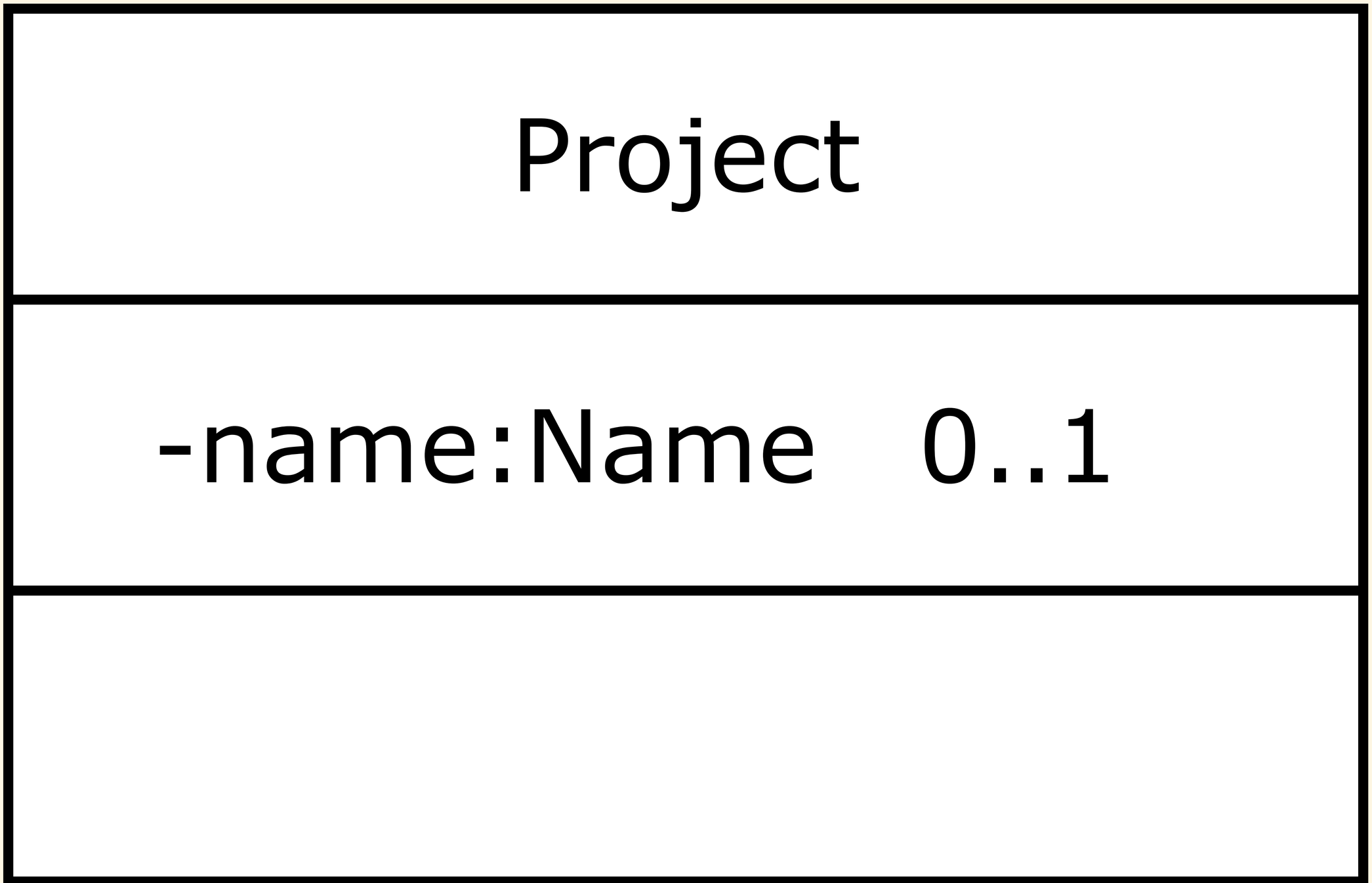
  `std::deque<int>`

  `std::list<int>`

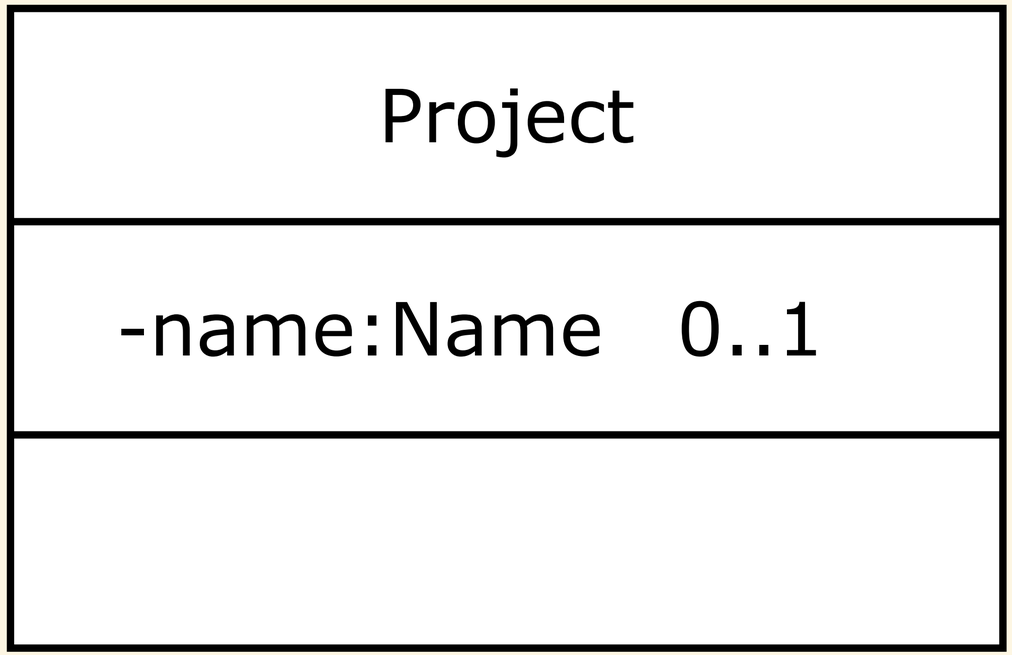
  `std::forward_list<int>` (C++11)

  `std::array<int, N>` (C++11) where N is a `constexpr`

Optional Single Value



Optional Single Value: C++ Pointer




```cpp
struct Record {
    char* name = nullptr;
    // ...
};
```

C++ Pointer Use

```cpp
struct Record {
    char* name = nullptr;
    // ...
};
```

```cpp
void output(const char* name) {
    // output name if it exists
    if (name) {
        std::cout << *name << '\n';
    }
}
```

3 states of `char*`

```cpp
// NOTE: Null state
char* name = nullptr;
output(name);

// NOTE: Empty value state
char* name = "";
output(name);

// NOTE: Value state
char* name = "Hello";
output(name);
```

std::string Record

```
struct Record {
    std::string name;
    // ...
};
```

2 States of `std::string`

```
// NOTE: Empty value state
std::string name;
output(name);

// NOTE: Value state
std::string name = "Hello";
output(name);
```

Solution: Flag

```cpp
struct Record {
    std::string name;
    bool hasName = false;
    // ...
};

// ...

if (hasName) {
    std::cout << name;
}
```

- For each optional, have to maintain another flag variable
- Have to remember the relationship between `name` and `hasName`

Solution: `std::optional`

```
std::optional<std::string> name;
assert(!name);

name = "Fred";
assert(name);

if (name)
    assert(*name == "Fred");

name = std::nullopt;
assert(!name);
```

- C++17 feature, similar to `boost::optional`
- Can be used on any value type (non-pointer)
- Allows for lazy loading/allocation
- Complete replacement of having to use pointers for *optional* single-value attributes

Scalar Usage

```cpp
// NOTE: create empty
std::optional<int> size;
assert(!size);

// NOTE: create with value
std::optional<int> leftValue(10);
assert(leftValue);
assert(*leftValue == 10);

// NOTE: deduction create with value
std::optional rightValue(10); // deduction guides
assert(rightValue);
assert(*rightValue == 10);

// NOTE: copy constructor
auto leftValueCopy = leftValue;

// NOTE: assignment
leftValueCopy = leftValue;

// NOTE: auto
auto oDouble = std::make_optional(3.0);
```

Use Case: Represent Optional Types

```cpp
#include <optional>

struct Record {
    std::optional<std::string> name;
    // ...
};
```

```cpp
void output(std::optional<std::string> name) {
    // output name if it exists
    if (name) {
        std::cout << *name << '\n';
    }
}
```

```cpp
// NOTE: Null state
std::optional<std::string> name;

// NOTE: Empty value state
std::optional<std::string> name = "";
output(name);

// NOTE: Value state
std::optional<std::string> name = "Hello";
output(name);
```

Use Case: Error Return

```cpp
std::optional<std::string> getName() {
    if (/* invalid state or error */)
        return std::nullopt;

    std::string name;
    // ...


    return name;
}
```

```cpp
std::optional<std::string> name;
assert(!name);

name = "Fred";
assert(name);

if (name)
    assert(*name == "Fred");

name = std::nullopt;
assert(!name);
```

Takeaway

- Use `std::optional` where you are mapping from an Optional type in UML
- Use `std::optional` in all cases where pointers are used to represent optional values