Object-Oriented Programming

Callbacks

Michael L. Collard, Ph.D.

Department of Computer Science, The University of Akron

Generalizing a Sort

```
void sort(std::vector<int>::iterator begin,
          std::vector<int>::iterator end,
          bool(*compare)(int, int)) {

    // sort [begin, end) using compare
    for (auto p = begin; p != end; ++p) {

        // locate smallest from [p, end)
        auto smallPos = p;
        for (auto q = p; q != end; ++q) {
            if (compare(*q, *smallPos))
                smallPos = q;
        }

        // swap the smallest from [p, end) to p
        std::swap(*smallPos, *p);

        // vector is now sorted from [begin, p)
        assert(std::is_sorted(begin, p));
    }
    assert(std::is_sorted(begin, end));
}
```

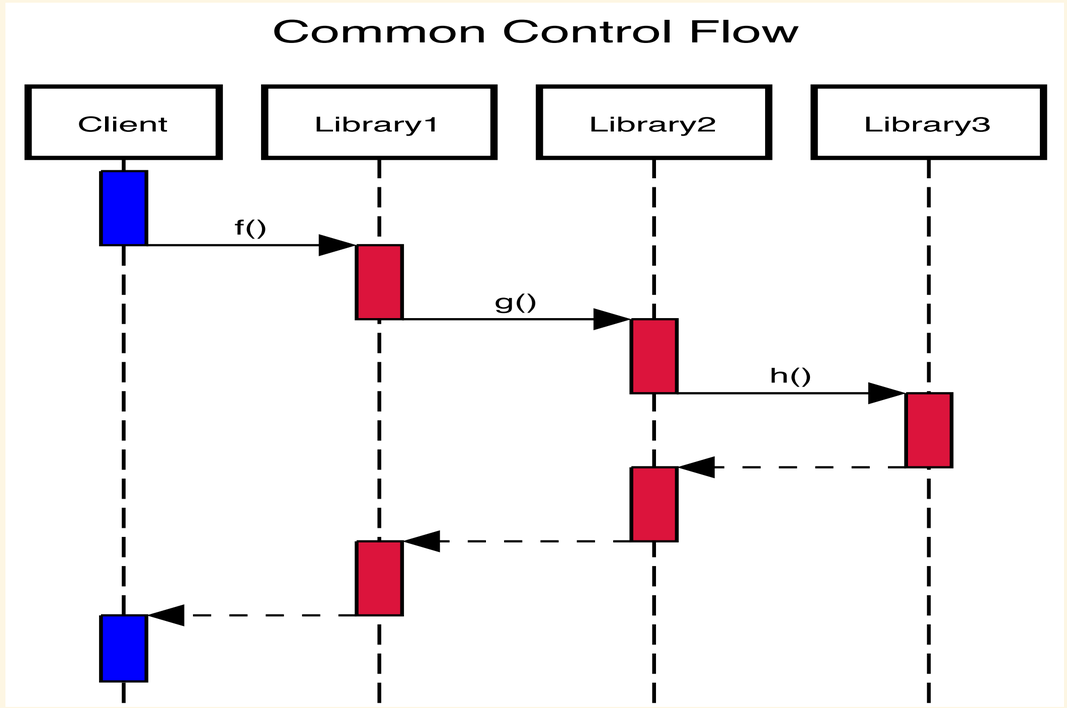
```
template <typename Compare>
void sort(std::vector<int>::iterator begin,
          std::vector<int>::iterator end,
          Compare compare) {

    // sort v
    for (auto p = begin; p < end; ++p) {

        // locate smallest from [p, end)
        auto smallPos = p;
        for (auto q = p; q < end; ++q) {
            if (compare(*q, *smallPos))
                smallPos = q;
        }

        // swap the smallest from [p, end) to p
        std::swap(*smallPos, *p);

        // vector is now sorted from [begin, p)
        assert(std::is_sorted(begin, p));
    }
    assert(std::is_sorted(begin, end));
}
```

C++ Library Equivalents

```c
void qsort(void *base,
           size_t nel,
           size_t width,
           int (*compar)(const void *, const void *));
```

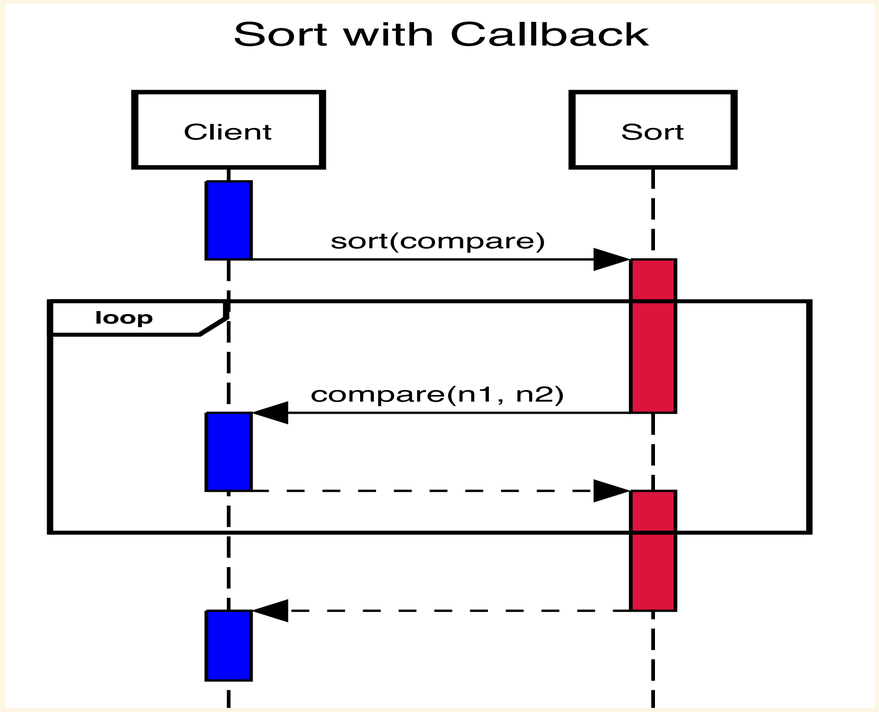
```c
template<class RandomIt, class Compare>
void std::sort(RandomIt first, RandomIt last, Compare comp);
```

Common Control Flow




```
void f() {
    g();
}

void g() {
    h();
}

void h() {
}

int main() {

    f();

    return 0;
}
```

Callback

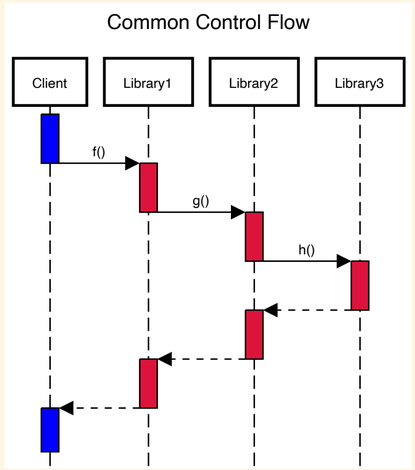


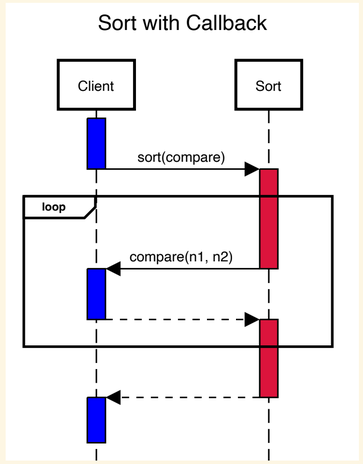
- A *callback* is a bit of computation from the *caller* that the *called function* or *callee function* can use whenever they need to in the implemented algorithm

- Callbacks permit customization of the computation in a function/method without changing the code in the computation

- Not a new programming mechanism

Control Flow Comparison





Callback Arguments

```
void output(int x) {
    std::cout << x << "\n";
}
// ...
traverse(output);
```

- free functions, *function pointers*

```
traverse([](int x) { std::cout << x << "\n"; });
```

- lambda functions, *anonymous functions*

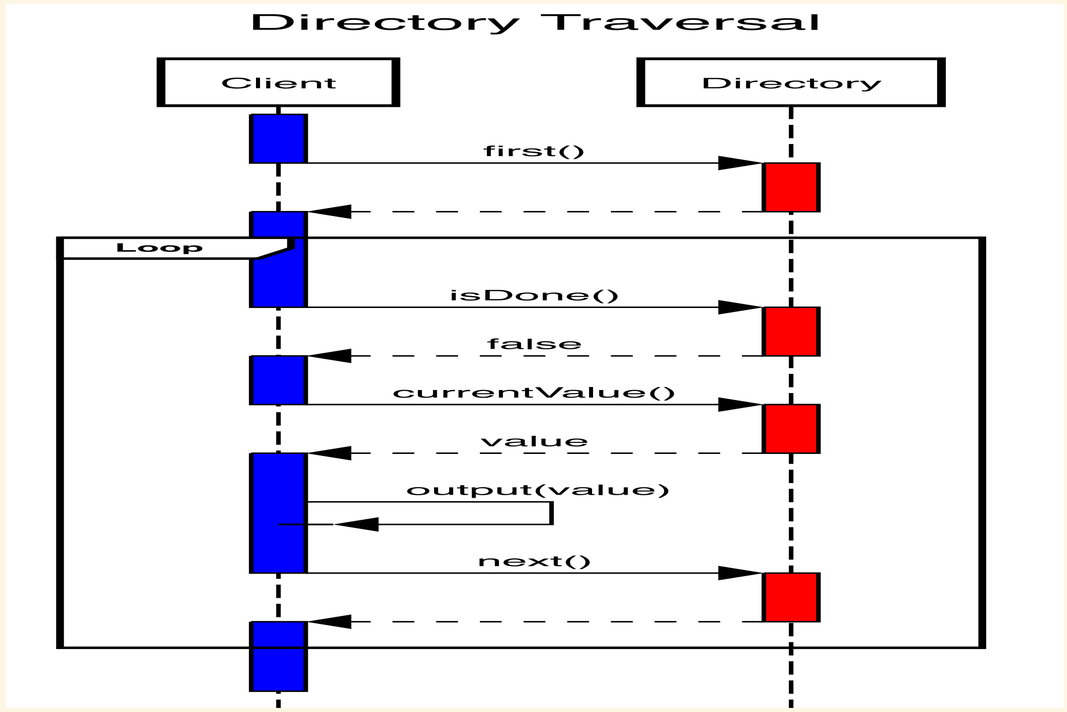
```
struct Functor {
    void operator()(int x) const { std::cout << x << "\n"; }
};

traverse(Functor());
```

- *functors*, class that acts like a function

```
class MyClass {
public:
    void output(int x) { std::cout << x << "\n"; }
};

MyClass obj;
traverse(std::bind(&MyClass::memberFunction, &obj, std::placeholders::_1));
```

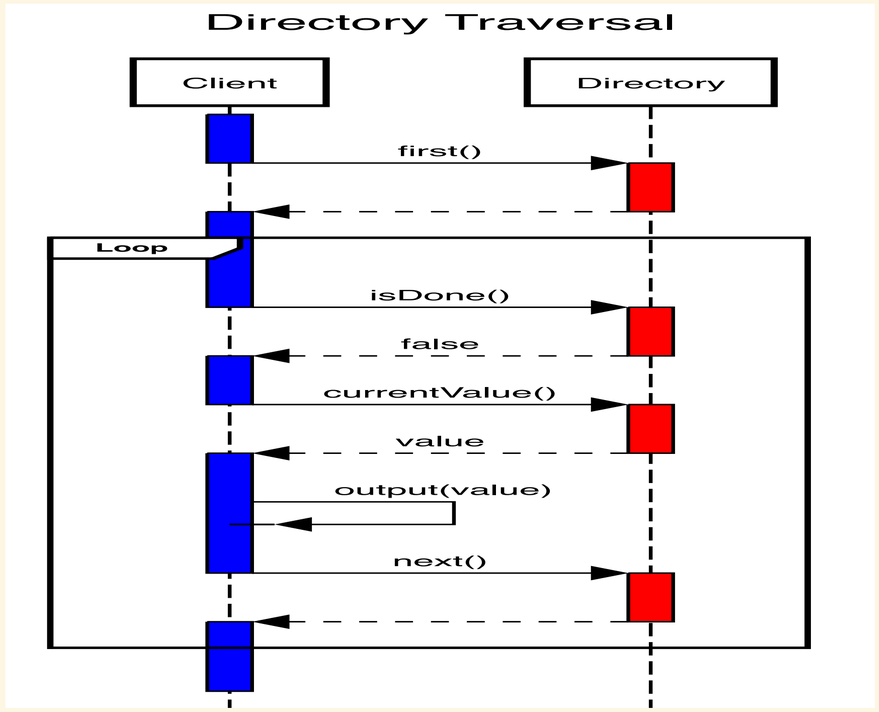
- methods, *pointers to member functions*

Example: Directory Traversal Iterator



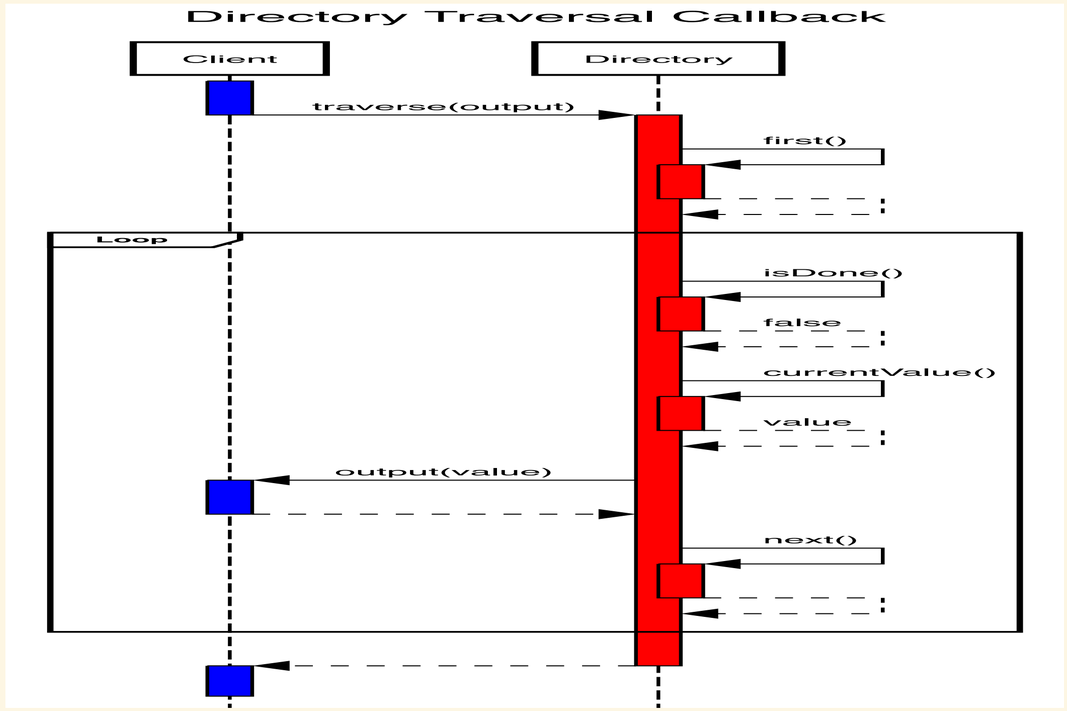

```cpp
/* @filename Filenames.cpp */
Directory files(path);

// output all the filenames in the directory tree
for (auto file = files.begin(); file != files.end(); ++file;) {

    // output filename
    std::cout << file->filename << '\n';
}
```

Example: Directory Traversal Iterator



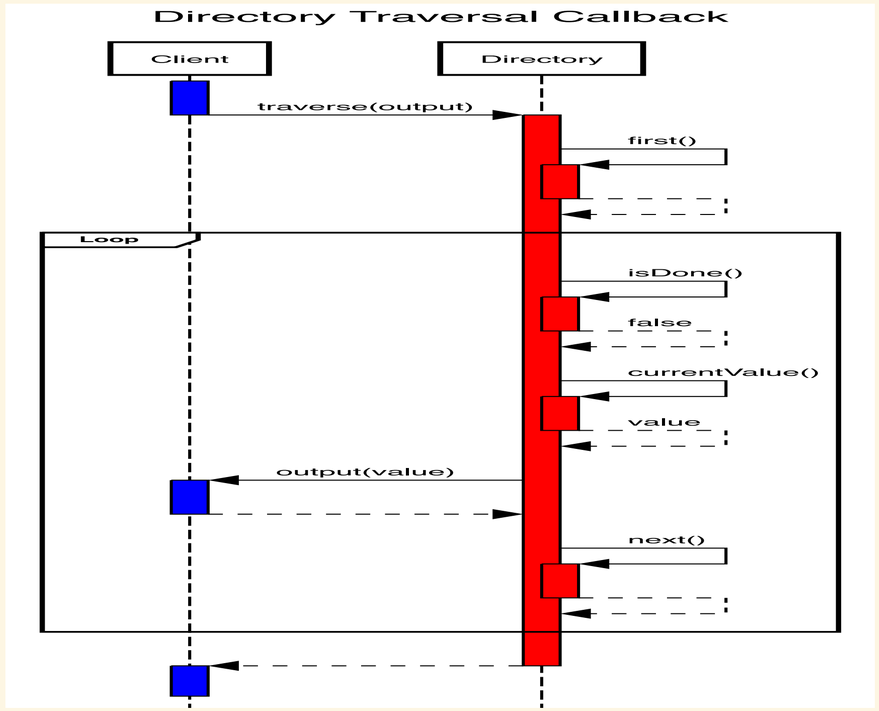
- Assume that `Directory` provides an iterator similar to `std::vector`
- The client has to know how to use the iterator
- Client calls `Directory` methods
- `Directory` methods **do not** call the client
- Have to know the methods available from the `Directory`

Example: Directory Traversal Callback



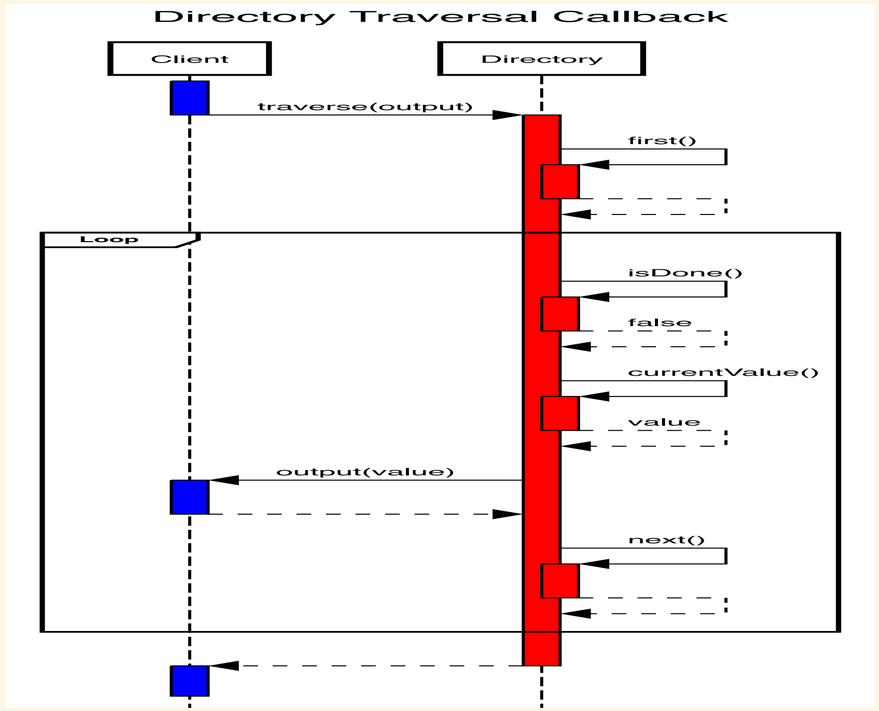

```cpp
/* @filename Directory.cpp */

// traverse the directory applying process to every file
void Directory::traverse(bool(*process)(std::string_view)) {

    for (auto file = files.begin(); file != files.end(); ++file) {

        process(file->filename);
    }
}

/* @filename Filenames.cpp */

// output filename
bool outputFilename(std::string_view filename) {
    std::cout << filename << '\n';
}

Directory files(path);

// output all the filenames in the directory tree
files.traverse(outputFilename)
```

Example: Directory Traversal Callback



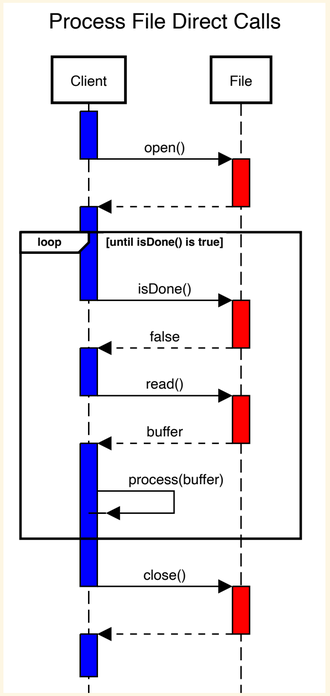
- Only one call to traverse the directory
- Client calls `traverse()`
- The method `traverse()` calls the client's `output()`
- The callback is provided the filename via a parameter that the `traverse()` calls each pass
- The whole process repeats until the directory is completely traversed

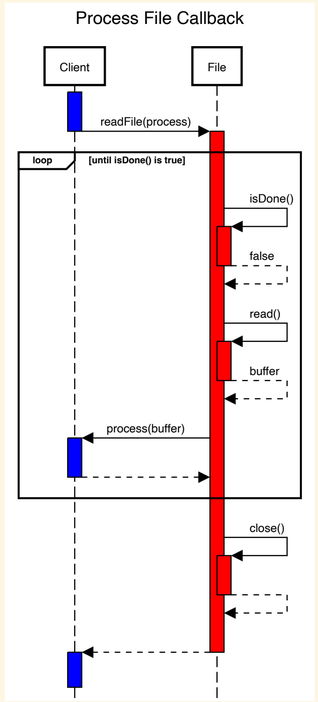
Example: Directory Traversal Callback



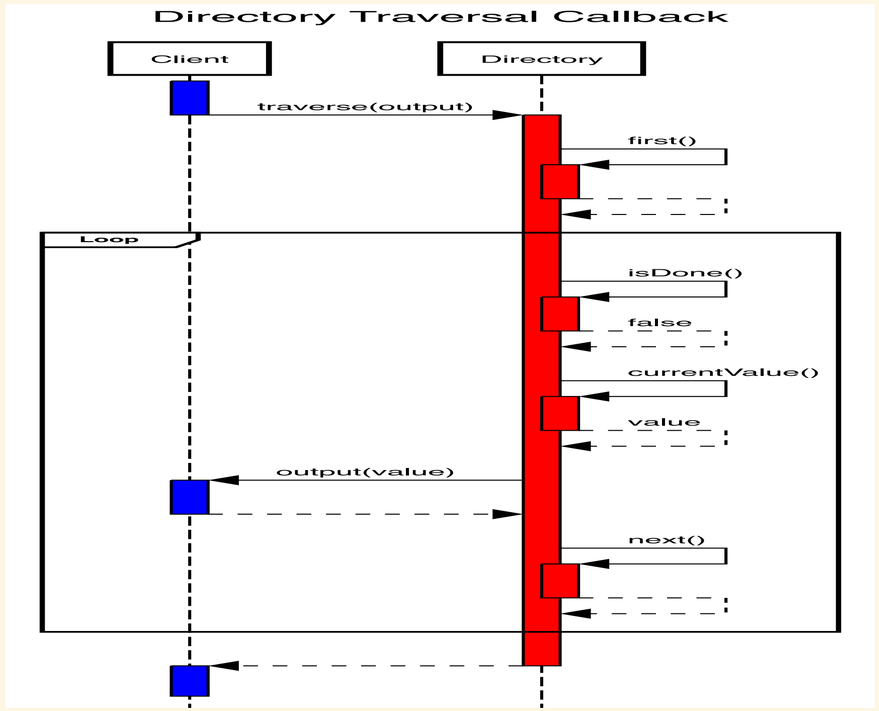
- Only the parameters, e.g., `value`, are available to the callback
- Each call to `traverse()` can use a different callback
- Very useful when processing is quite complex, and it may be inefficient to save state and return each step
- The name of the callback, `process()`, is typically general as we don't know what it will be used for
- Processing of a directory is much more complex than this

Example: Process File





Callbacks



- Can be used with any level of complex processing
- Details of processing hidden inside the callback-enabled processor
- Client only has to provide the code to run when a particular *event* occurs
- Essential for *asynchronous* processing