C++ References*

Jeffrey D. Oldham

2000 Mar 25

1 Caveat Reader!

I wrote this notes to prepare myself for lecture. The topic of returning references from functions is not well documented, even in the C++ language definition so I have used my experience to guess the rules.

2 Introduction

A C++ reference is used when:

- a value needs to be changed,
- copying a value is not permitted, or
- copying a value is not desirable, i.e., is inefficient.

3 Reference Parameters

This program illustrates these uses for reference parameters.

```
#include <iostream>
#include <string>

void foo(int & length,          // string's length placed here
         ostream & out,         // illegal to copy
         const string & s) {    // too expensive to copy
  length = s.size();
  out << "foo: The string's length is " << length << ".\n";
  return;
}

int main(int argc, char *argv[]) {
  cout << "This silly program prints the length of the first command-line argument.\n";
  if (argc == 1) {              // too few command-line arguments
    cerr << argv[0] << ": string\n";
```

```
    throw "too few command-line arguments";
  }

  int l;                        // initially has garbage value
  foo(argv[1], l);
  cout << argv[1] << " has " << l << " characters.\n";

  return EXIT_SUCCESS;
}
```

For `foo` to set the value of the first parameter it must be passed by reference.

`istream`s and `ostream`s may not be copied so the second parameter must be passed by reference.[1]

`foo`'s last parameter is passed by reference so the `string` need not be copied. Not copying the string saves time proportional to the number of characters in the string. Since we are passing the string by reference but we do not want the string to be copied, we mark it `const`. Thus, `foo` can inspect and use the string, but it may not change its contents. For more information, read the textbook, pp. 66–67.

4 Returning References

A function may return a reference for exactly the same reasons:

- the returned value may need to be changed,
- copying the return value is not permitted,
- copying the return value is not desirable, i.e., is inefficient.

This program illustrates these uses for reference parameters.

```
#include <iostream>
#include <string>
#include <algorithm>            // has copy()
#include <assert.h>             // has assert()
```

[1]Each `istream` object tracks the position of the next character to read. If a function was given a copy of an `istream` object, it could read from the stream. Then the `main` function could read the same characters again! Thus, helper functions to read input would be mostly useless. By using only references to `istream` objects, all functions refer to the same object; if one reads some characters changing the position, all change the position. Similar reasoning applies to `ostream`s.

```
class foo {
public:
  // constructor and destructor
  foo (const unsigned int sz) {
    size = sz;
    if (size == 0) size += 1;
    array = new string[size];
  }
  ~foo(void) { delete [] array; }

  // copying functions
  foo(const foo & f) {
    copier(f);
    return;
  }
  foo& operator=(const foo & f) {
    if (this != &f) {
      delete [] array;
      copier(f);
    }
    return *this;
  }

  // element access
  string& operator[](unsigned int i) {
    assert(i < size);
    return array[i];
  }
  const string& operator[](unsigned int i) const {
    assert(i < size);
    return array[i];
  }

  // output
  friend ostream& operator<<(ostream& out, const foo & f) {
    for (unsigned int i = 0; i < f.size; i += 1)
      out << f.array[i] << endl;
    return out;
  }

private:
  unsigned int size;
  string * array;

  void copier(const foo & f) {
```

```
    size = f.size;
    array = new string[size];
    copy(f.array, f.array + size, array);
    return;
  }
};


int main(void) {
  foo f(4);
  foo g(4);
  f[0] = f[2] = "hello";
  f[1] = (f[3] = "world");
  g = f;
  cout << "f:\n" << f;
  cout << "g:\n" << g;
  return EXIT_SUCCESS;
}
```

The `foo` class holds a dynamically-allocated array of strings. Thus, we need a destructor, a copy constructor, and an assignment operator. We also define two element access functions and an output operator.

The statement `f[2] = "hello"` illustrates changing the return value. Equivalent code with an explicit function call is `f.operator[](2) = "hello"`. The result of `f.operator[](2)` is assigned a value so it must be a reference, not a value.

The `operator[]` function obeys the two rules for functions returning references. Any function having return type `T &` must follow two rules:

- The `return` statement must return a value with type `T`.
- The value returned must not be a variable local to the function.

`operator[]` obeys the first rule because it returns a member of a `string` array. It obeys the second rule because it returns a reference to a position in the object's array, not the function's only variable `i`.

The `const` version of `operator[]` guarantees that the returned result is not changed. It returns a reference to avoid copying the return value since copying a very long string may be very expensive but marks it `const` so it is not changed. The second occurrence of `const` guarantees that any call to this function does not change the object's contents. Defining this `const` version is necessary so that functions with `const foo` variables or parameters can access their array elements.

Tip: Whenever defining a function returning a reference to a variable inside an object, define a `const` version that will also work with `const` objects.

As mentioned above, `ostream`s may not be copied. Thus, the output operator must return a reference so its return value is not copied. The expression it returns has `ostream` type and is defined outside the function so returning `out` is permissible.