Assignment 5: Move Semantics

C++ Programming Course, Summer Term 2018

5-0 Have a Cookie

- Have a look at cppinsights.io.
- Understand what the tool does, experiment with it a bit.
- Rejoice.

5-1 Containers Revisited

References:

- Session 5 notes and discussed example
- Common misconception with C++ move semantics

5-1-0 CppCon and Chill

Watch these talks:

- Don't Help the Compiler
- The strange details of std::string at Facebook

These talks address a rather experienced (5+ years in industry) audience. Pay close attention, pause, think, rewatch. It will be overwhelming. That's okay, speakers at CppCon are beasts.

5-1-1 list Move Semantics

- Add a test case to the test suite of `List<T>` from assignment 3 to validate moving / copy elision of temporary `List<T>` objects.

- Validate that your test case in fact creates / copies / assigns temporary `List<T>` instances (gdb, cppinsights.io, logging ... anything that works for you).
- Extend your implementation of `List<T>` from assignment 3 by move semantics.
- Validate that temporary instances are moved (use techniques from step 2).

5-1-2 sparse_array Move Semantics

Same as 5-1-1 for `sparse_array<T,N>`:

- Add a test case to the test suite of `sparse_array<T,N>` from assignment 4 to validate moving / copy elision of temporary `sparse_array<T,N>` objects.
- Validate that your test case in fact creates / copies / assigns temporary `sparse_array<T,N>` instances (gdb, cppinsights.io, logging ... anything that works for you).
- Extend your implementation of `sparse_array<T,N>` from assignment 4 by move semantics.
- Validate that temporary instances are moved (use techniques from step 2).

5-2 Compiler Explorer

Analyze the following simplified variant of the example discussed in session 5 in compiler explorer.

How does it help you to check if move semantics of `ArrayWrapper` work as expected?

(Hint: You don't need to read assembly)

```
#include <string>

#define LOG(scope, msg) do { \
} while(0)


template <class T>
class ArrayWrapper {
  typedef ArrayWrapper<T> self_t;

public:
  typedef T   value_type;
  typedef T * iterator;
```

```
public:

  ArrayWrapper()
  : _data(new T[64]),
    _size(64),
    _name("d") {
    LOG("ArrayWrapper()",
        "ooo --- default construct " << _name);
  }

  ArrayWrapper(int n, std::string name)
  : _data(new T[n]),
    _size(n),
    _name(name) {
    LOG("ArrayWrapper(n,s)",
        "*** --- create " << _name);
  }

  // move constructor
  ArrayWrapper(self_t && other)
      : _data(other._data),
        _size(other._size),
        _name(std::move(other._name)) {
    LOG("ArrayWrapper(self &&)",
        "((( --- move  * <-- " << _name);
    other._data = NULL;
    other._size = 0;
  }

  // copy constructor
  ArrayWrapper(const self_t & other)
  : _data(new T[other._size]),
    _size(other._size),
    _name(other._name) {
    LOG("ArrayWrapper(const self &)",
        "=== --- create copy of " << _name);
    for (int i = 0; i < _size; ++i) {
      _data[i] = other._data[i];
    }
    LOG("ArrayWrapper(const self &)",
        "=== --- copied " << _size << " values *_*'");
  }

  ~ArrayWrapper() {
    LOG("~ArrayWrapper()",
```

```
        "xxx --- destroy " << _name << (
          ((NULL != _data) ? " and free data" : " and go home")));
    delete[] _data;
  }

  iterator begin() const {
    return _data;
  }

  iterator end() const {
    return _data + _size;
  }

  int size() const {
    return _size;
  }

 private:
  T               * _data;
  int               _size;
  const std::string _name;
};


ArrayWrapper<int>
return_array_by_value(
  int         size,
  std::string name) {
  if (size % 2 == 0) {
    return ArrayWrapper<int>(size / 2, name);
  } else {
    return ArrayWrapper<int>(size * 2, name);
  }
}

int
accept_array_by_value(
  ArrayWrapper<int> a) {
  ArrayWrapper<int> mine(std::move(a));
  mine.begin()[0] = 345;
  return mine.begin()[0];
}

int main() {
  accept_array_by_value(
     return_array_by_value(234, "X")
```

```
  );

  return 0;
}
```