[CS 225] Advanced C/C++

Lecture 9: Move semantics

DigiPen
INSTITUTE OF TECHNOLOGY
SINGAPORE

Sławomir "Swavek" Włodkowski
(Summer 2020)

Agenda

- Continuing last lecture:
  - Class templates
  - TMP

Class templates

- Generic programming: “cookie-cutters” for classes.
- Support specialization
  - Full (explicit) specialization – customizing a base template for a full set of template arguments.
  - Partial specialization – customizing a base template for a given category of template arguments.

Class templates

```cpp
template<typename T, std::size_t N>
class my_array
{
    T _data[N];
public:
    T& operator[](std::size_t index)
        { return _data[index]; }
    T operator[](std::size_t index) const
        { return _data[index]; }
    T* data()
        { return _data; }
    std::size_t size() const
        { return N; }
};
```

```cpp
#include <iostream>


int main()
{
    my_array<int, 2> arr;
    arr[0] = 10;
    arr[1] = 20;

    for (std::size_t i=0; i<arr.size(); ++i)
    {
        std::cout << arr[i] << std::endl;
    }
}
```

Class templates

- Support class template argument deduction (CTAD)
  - CTAD is available since C++17.
  - Works in initialization, new expression, C++ style cast.
  - Relies on constructors using template arguments.

Understanding l-values and r-values

Expressions (an operator with its operands, a literal, a variable name) has a ***data type*** and a ***value category***.

<u>Primary value categories:</u>

- l-values
- r-values
    - pr-values
    - x-values

Introduction to TMP

Template meta-programming (TMP) is a programming technique, where templates are use for generating other compile-time code: constants, data structures, functions working on values or types.

Key observations:

- Most of the resulting template instances do not find their ways into an executable.
- Expressions generating types get replaced with generated types; expressions generating compile-time constants get replaced with generated constants.
- They are no longer “cookie-cutters” performing type substitution for run-time code (generic programming).

Class templates

Generic programming:

- Define classes parameterized for `Ts` and `Ns`.

- Template meta-programming:
  - Define custom scopes with compile-time definitions (static constants, enums, functions, type aliases, etc.).
  - Behave as if they are parameterized namespaces.
  - If not instantiated as an object or referred by their static member in runtime instruction, they result in no code.

Function templates

Generic programming:

- Define functions parameterized for `Ts` and `Ns`.

- Template meta-programming:
  - Define functions collapsing universal references.
  - Define compile-time recursive functions.
  - Define functions for calculating types (later this trimester).

Other templates

Generic programming:

- Define values and type aliases to simplify syntax.

- Template meta-programming:
  - Define templated abstractions to hide where compile-time values or types come from.

```
// This type is an alias defined as a dependent type of a class
typename remove_reference<MyType>::type example_of_bad_abstraction;

// This type alias comes from anywhere...
remove_reference_t<MyType>              example_of_good_abstraction;
```

Example

Fibonacci sequence
1, **1**, 2, 3, 5, 8, …

$$f_n = \begin{cases} 1, when\ n = 1 \\ 1, when\ n = 2 \\ f_{(n-1)} + f_{(n-2)} \end{cases}$$

```
#include <iostream>

template <unsigned int N> struct FibImpl
{
    static const unsigned int value =
        FibImpl<N - 1>::value +
        FibImpl<N - 2>::value;
};
template <> struct FibImpl<1>
{
    static const unsigned int value = 1;
};
template <> struct FibImpl<2>
{
    static const unsigned int value = 1;
};

template <unsigned int N>
const unsigned int Fib = FibImpl<N>::value;

int main()
{
    // Index: 1 2 3 4 5 6  7...
    // Value: 1 1 2 3 5 8 13...
    std::cout << Fib<7> << std::endl;
    // Code inside an executable file:
    // std::cout << 13 << std::endl;
}
```

Agenda

- New topics:
  - Understanding l-values and r-values
  - L-value and r-value references
  - Copy semantics vs. move semantics
  - STL
    - `std::move`
    - `std::remove_reference`

L-values

Locator values are objects that can be located in memory (have an address), and generally can be used on the left or the right side of an assignment operator.

Note: `const` values can be l-values, even though they can be put only on the right side of the assignment, because they can reside in memory and have a valid address.

L-values

<u>Examples:</u>

- The name of a variable, a function, a data member.
- A dereferenced pointer.
- A string literal (`"ABC"`).
- L-value reference result from a function.
- L-value reference variable or a parameter.
- R-value reference variable or a parameter.

"If a reference has a name, it behaves as an l-value."

R-values

All ***not l-values*** (by exclusion); generally, can be used on the right side of an assignment operator.

These values are special, because they are never reused unless explicitly stated; this makes them available for destructive optimizations.

Getting an r-value requires another function call, creating a copy of immutable object, or explicit reuse.

R-values

pr-value

“Pure r-value”

<u>Examples:</u>

- Literal values (immutable and copied)
- Non-reference result from a function.
- Result of built-in arithmetic, logical, comparison, address-of and some other operators.
- The `this` pointer.

R-values

x-value

“Expiring value” or “explicit r-value”

<u>Examples:</u>

- R-value reference result from a function.
- R-value expression without a name.
- Cast expression resulting in r-value.
- Data member of an object, or element of an array object where the object is r-value itself.

L-value and r-value references

A reference is an alias to an object. It is an abstraction; under-the-hood it can be represented by a raw pointer or completely eliminated by a compiler.

Key uses:

- Aliasing an object in a function.
- Passing parameters by reference.
- Returning results by reference.

L-value references

An l-value reference can bind to an l-value.

Constant l-value references can bind to l-values or r-values.

R-value references

An r-value reference can bind to an r-value
(pr-value or x-value).

Constant r-value references can bind to `const` r-values. They are often used to overload functions for r-values, but still support only the copy semantics.

Copy semantics vs. move semantics

Functions can be overloaded based only l-value or r-value reference nature of function arguments.

<u>Possibilities:</u>

- L-values may be used elsewhere; they **must** be copied.
- L-values or r-values that are `const` **must** be copied.
- R-values are destructible so their value **can** be swapped.

Copy semantics vs. move semantics

Example:

Constructors and assignment operators can be overloaded to be more efficient by swapping when possible.

Copy semantics

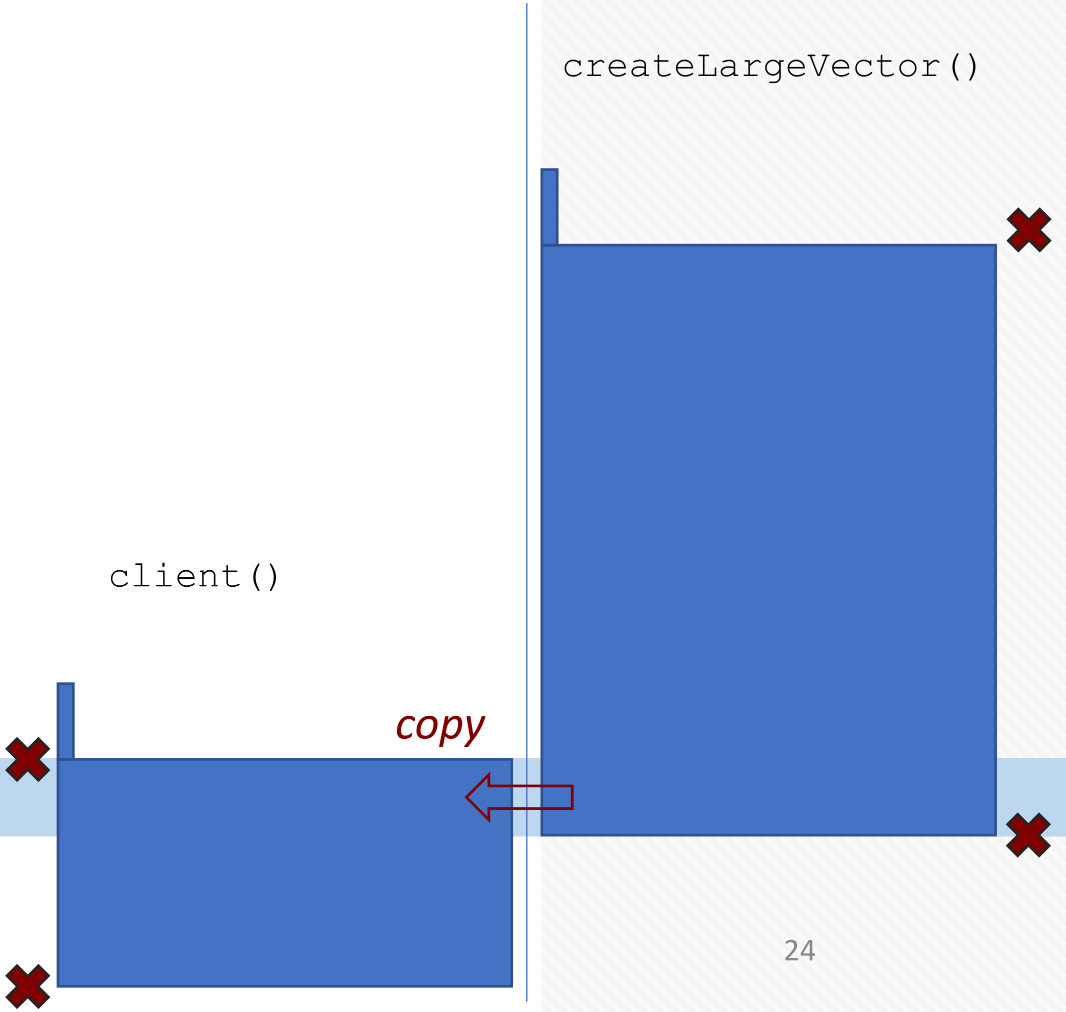
```
vector(const vector<T>& v) : _data{new T[v._capacity]},
    _size{v._size}, _capacity{v._capacity}
{
    std::copy(_data, _data + size, v._data);
}

vector<T>& operator=(const vector<T>& v)
{
    if (this != &v)
    {
        T* temp = new T[v._capacity];
        std::copy(v._data, v._data + v.size, temp);
        delete[] _data;
        _data = temp;
        _size = v._size;
        _capacity = v._capacity;
    }
    return *this;
}
```

Copy semantics

```
using V = std::vector<int>;
V createLargeVector()
{
    V v;
    v.resize(largeValue);
    return v;
}

void client()
{
    V v;
    v = createLargeVector();
    // Important stuff
}
```



Move semantics

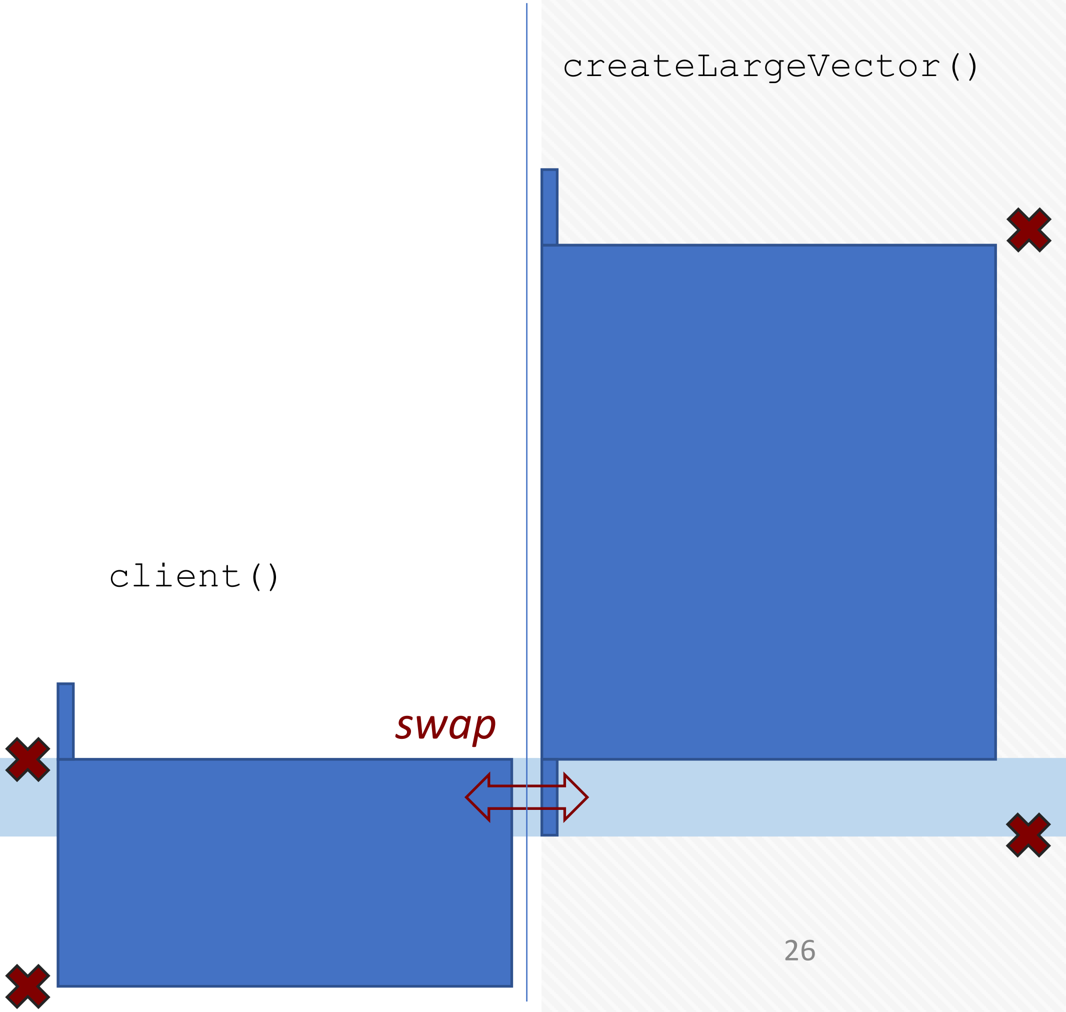
```
vector(vector<T>&& v) : _data{v._data},
    _size{v._size}, _capacity{v._capacity}
{
    v._data = new T[0];
    v._size = 0;
    v._capacity = 0;
}

vector<T>& operator=(vector<T>&& v)
{
    if (this != &v)
    {
        std::swap(v._data, _data);
        std::swap(v._size, _size);
        std::swap(v._capacity, _capacity);
    }
    return *this;
}
```

Move semantics

```
using V = std::vector<int>;
V createLargeVector()
{
    V v;
    v.resize(largeValue);
    return v;
}

void client()
{
    V v;
    v = createLargeVector();
    // Important stuff
}
```



Move semantics

Copy and move constructors

```
vector(const vector<T>& v);
vector(vector<T>&& v);
```

Copy and move assignment operators

```
vector<T>& operator=(const vector<T>& v);
vector<T>& operator=(vector<T>&& v);
```

std::move

```
int&& std::move(int& t)
{
    return static_cast<int&&>(t);
}


int&& std::move(int&& t)
{
    return static_cast<int&&>(t);
}
```

`std::move`

Since the implementation of `std::move()` looks the same for `int&` and `int&&`, we should replace it with a single function template.

Problems:

1. Should the argument be `T&` or `T&&`?
2. For a template argument `T`, `T&&` is not r-value reference!?