Creative Software Design

Modern C++ Part 2

Yunho Kim
yunhokim@hanyang.ac.kr
Dept. of Computer Science

Topics Covered

- Lambda functions (or lambda expressions)

- Move semantics

Recall: Function Pointers

- What?
  - A pointer to a function

- Why?
  - Allows the same code to work in multiple instances
    - E.g. sort ascending and sort descending can use the same code with changi ng the function being used for the comparison

Function Pointer - Syntax

- `ret(*name)(arg, arg)`
  - `Ret` – return type
  - `Name` – the name of the variable representing the function being pointed to
  - `Arg` – the type of the arguments the function should take

Function Pointer Example

```
bool anonymousCompare(int x, int y, bool(*comp)(int, int)) {
   return comp(x, y);
}
bool lessThan(int x, int y) {
   return x < y;
}
bool greaterThan(int x, int y) {
   return x > y;
}
int main(int argc, char* argv[]) {
   anonymousCompare(3, 2, lessThan); //returns false
   anonymousCompare(3, 2, greaterThan); //returns true
}
```

Lambda Functions (stackoverflow.com)

- What are they?
  - Functions defined directly in source code (as opposed to being defined in a se parate function)
  - Functions not bound to an identifier

- Other names
  - Anonymous function
  - Functional literal
  - Lambda abstraction

Lambda Function Syntax

- Full syntax
  - `[ capture-list ] ( params ) -> ret { body }`

- Lambda function parts
  - `capture-list`
    - List of variables to pass to the function, like arguments (does not change the function signature), typically local variables
  - `params`
    - The parameters that are passed to the function (defines the function signature)
  - `body`
    - Where the code for your function goes
  - `ret`
    - The return type of the function
    - If not defined, return type is inferred using rules similar to auto

Lambda Function Syntax

- Full Syntax
  - `[ capture-list ] ( params ) -> ret { body }`
- Partial Syntax (still valid)
  - `[ capture-list ] ( params ) { body }`
- Partial Syntax (still valid)
  - `[ capture-list ] { body }`

Capture

- `[&](){}`
  - Implicitly capture the used automatic variables by reference

- `[=]() {}`
  - Implicitly capture the used automatic variables by copy

- `[&, x, y] {}`, (or `[=, &x, &y] {}`)
  - Implicitly capture the used automatic variables by reference (or copy) except for `x` and `y` captured by copy (or reference)

- `[x, &y, &z] () {}`
  - Capture `x` by copy and `y` and `z` by reference

Why Capture

- Why do we need capture?
  - We can pass local variables as parameters to the lambda function!

- Capture does not change the signature of a lambda function while parameters change

Lambda Function Examples

```
bool myfunction(int i, int j) { return (i<j); }
int main() {
    int myNumba = 123455.304f;
    auto lambda = [myNumba]() { cout << "lambda: "<< myNumba << endl; };
    lambda();//prints myNumba

    int myints[] = { 32,71,12,45,26,80,53,33 };
    std::vector<int> myvector(myints, myints + 8);

    std::sort(myvector.begin(), myvector.end(), [](int i, int j){
        return (i<j)
    });
    std::sort(myvector.begin(), myvector.end(), myfunction);
}
```

Quiz #1

- What is the expected output? (including compile/runtime error)

```
#include<iostream>
using namespace std;
int main()
{
    int a = 5;
    int b = 5;
    auto check = [&a]() {
        a = 10;
        b = 10;
    }
    check();
    cout<<"Value of a: "<<a<<endl;
    cout<<"Value of b: "<<b<<endl;
    return 0;
}
```

Lambda Functions vs. Function Pointers

- Lambda Function
- One shot
- Prevents bloating of header files
  - Moves it to the code
- Better in-line functionality
  - Has the potential to run faster
- Not bound to a trackable identifier
- Decreases Modularity
- Higher coupling (decreased separation of concerns)

- Function Pointers
- Passed around multiple times or multiple different functions
- Can increase clarity of code
  - When functions passed have meaningful names
- Bound to a trackable identifier
- Increased separation of concerns
- Increased modularity

Recall `unique_ptr` and STL Example

```
int main(int argc, char **argv) {
  std::vector<std::unique_ptr<int> > vec;

  vec.push_back(std::unique_ptr<int>(new int(9)));
  vec.push_back(std::unique_ptr<int>(new int(5)));
  vec.push_back(std::unique_ptr<int>(new int(7)));

  int z = *vec[1];
  std::cout << "z is: " << z << std::endl;

  // compile error
  std::unique_ptr<int> copied = vec[1];

  // OK
  std::unique_ptr<int> moved = std::move(vec[1]);
  std::cout << "*moved: " << *moved << std::endl;
  std::cout << "vec[1].get(): " << vec[1].get() << std::endl;

  return EXIT_SUCCESS;
}
```

Copy Semantics

- Assigning values typically means making a copy
  - Usually, this is what you want
    - *e.g.* assigning a string to another makes a copy of its value
  - Sometimes this is wasteful
    - *e.g.* assigning a returned string goes through a temporary copy

```
std::string ReturnFoo(void) {
  std::string x("foo");
  return x;  // this return might copy
}

int main(int argc, char **argv) {
  std::string a("hello");
  std::string b(a);  // copy a into b

  b = ReturnFoo();   // copy return value into b

  return EXIT_SUCCESS;
}
```

Move Semantics (C++11)

- "Move semantics" move values from one object to another without copying ("stealing")
  - Useful for optimizing away temporary copies
  - A complex topic that uses things called "*rvalue references*"

```cpp
std::string ReturnFoo(void) {
  std::string x("foo");
  // this return might copy
  return x;
}

int main(int argc, char **argv) {
  std::string a("hello");

  // moves a to b
  std::string b = std::move(a);
  std::cout << "a: " << a << std::endl;
  std::cout << "b: " << b << std::endl;

  // moves the returned value into b
  b = std::move(ReturnFoo());
  std::cout << "b: " << b << std::endl;

  return EXIT_SUCCESS;
}
```

Transferring Ownership via Move

- `unique_ptr` supports move semantics
  - Can "move" ownership from one `unique_ptr` to another
    - Behavior is equivalent to the "release-and-reset" combination

```
int main(int argc, char **argv) {
  unique_ptr<int> x(new int(5));
  cout << "x: " << x.get() << endl;

  unique_ptr<int> y = std::move(x); // x abdicates ownership to y
  cout << "x: " << x.get() << endl;
  cout << "y: " << y.get() << endl;

  unique_ptr<int> z(new int(10));

  // y transfers ownership of its pointer to z.
  // z's old pointer was delete'd in the process.
  z = std::move(y);

  return EXIT_SUCCESS;
}
```

Why Move Semantics

- Performance (avoid unnecessary copies)
  - Mostly done automatically

- Explicitly transfer ownership
  - Example: `std::unique_ptr`

Example: Explicitly moving in code

- std::swap (presume T supports moving)

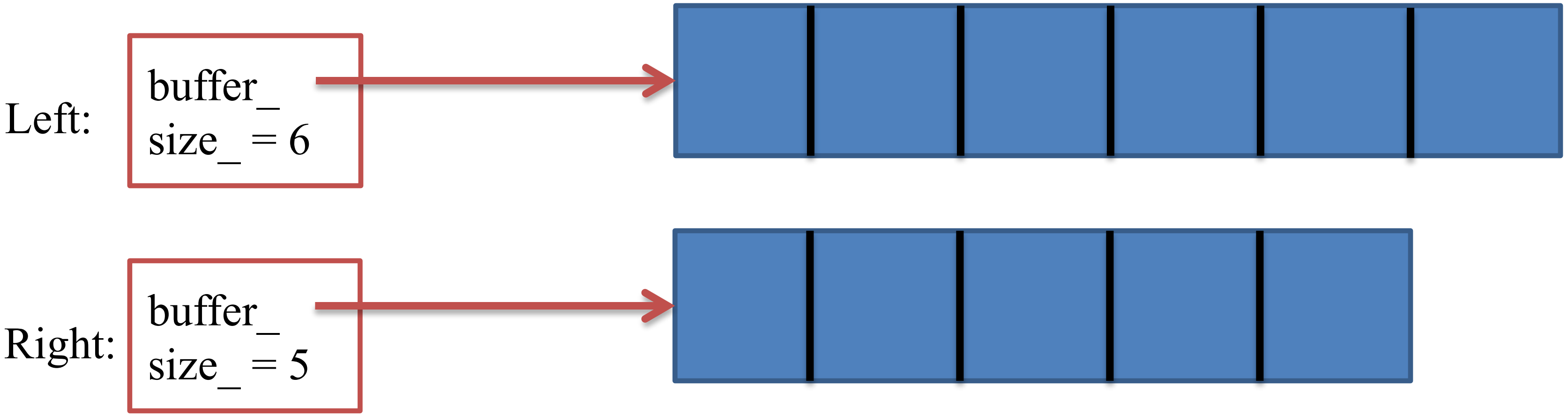
```
template<class T>
void swap(T& left, T& right) {
  T temp(left); // Copy left  to temp
  left = right; // Copy right to left
  right = temp; // Copy temp  to right
} // Destroy temp
```

std::move facilitates moves

```
template<class T>
void swap(T& left, T& right) {
  T temp(std::move(left)); // Move left  to temp
  left = std::move(right); // Move right to left
  right = std::move(temp); // Move temp  to right
} // Destroy temp (probably has no “real” state anymore)
```

std::swap example for std::vector

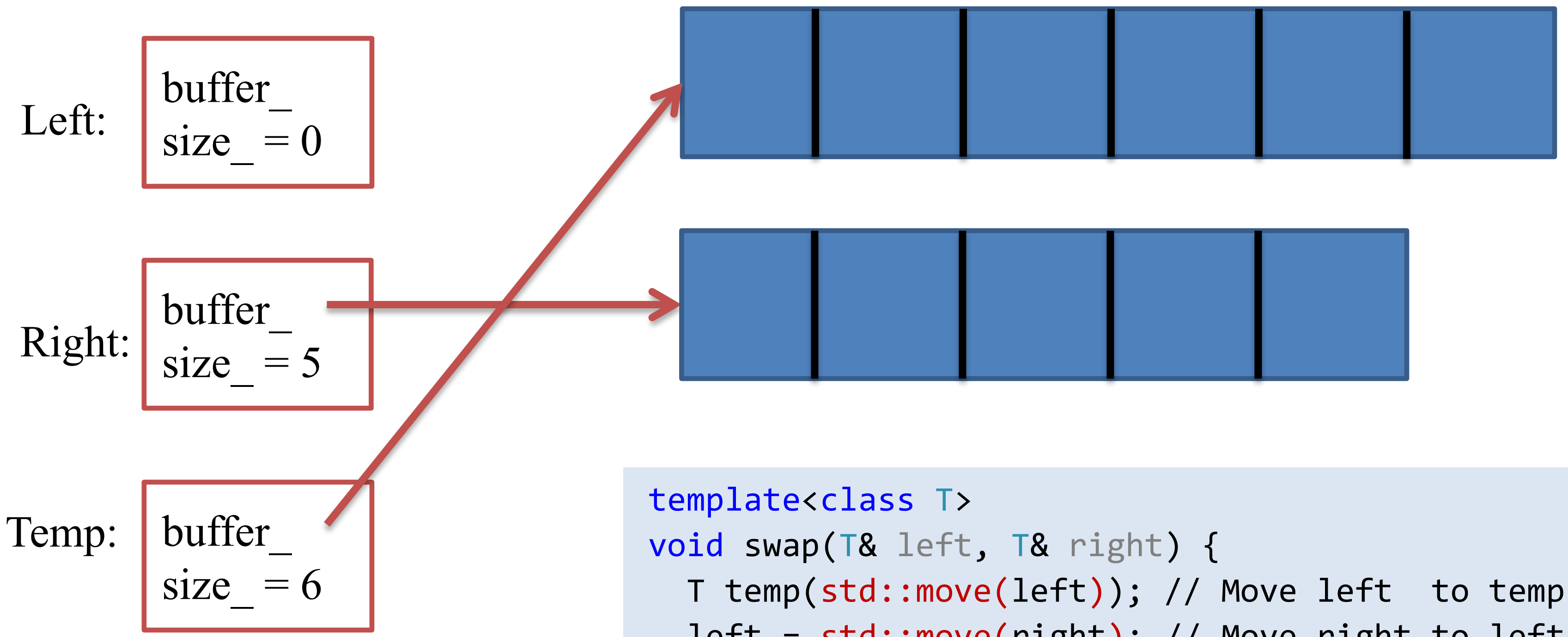
- Initial situation




```
template<class T>
void swap(T& left, T& right) {
  T temp(std::move(left)); // Move left  to temp
  left = std::move(right); // Move right to left
  right = std::move(temp); // Move temp  to right
} // Destroy temp (probably has no “real” state
anymore)
```

std::swap example for std::vector

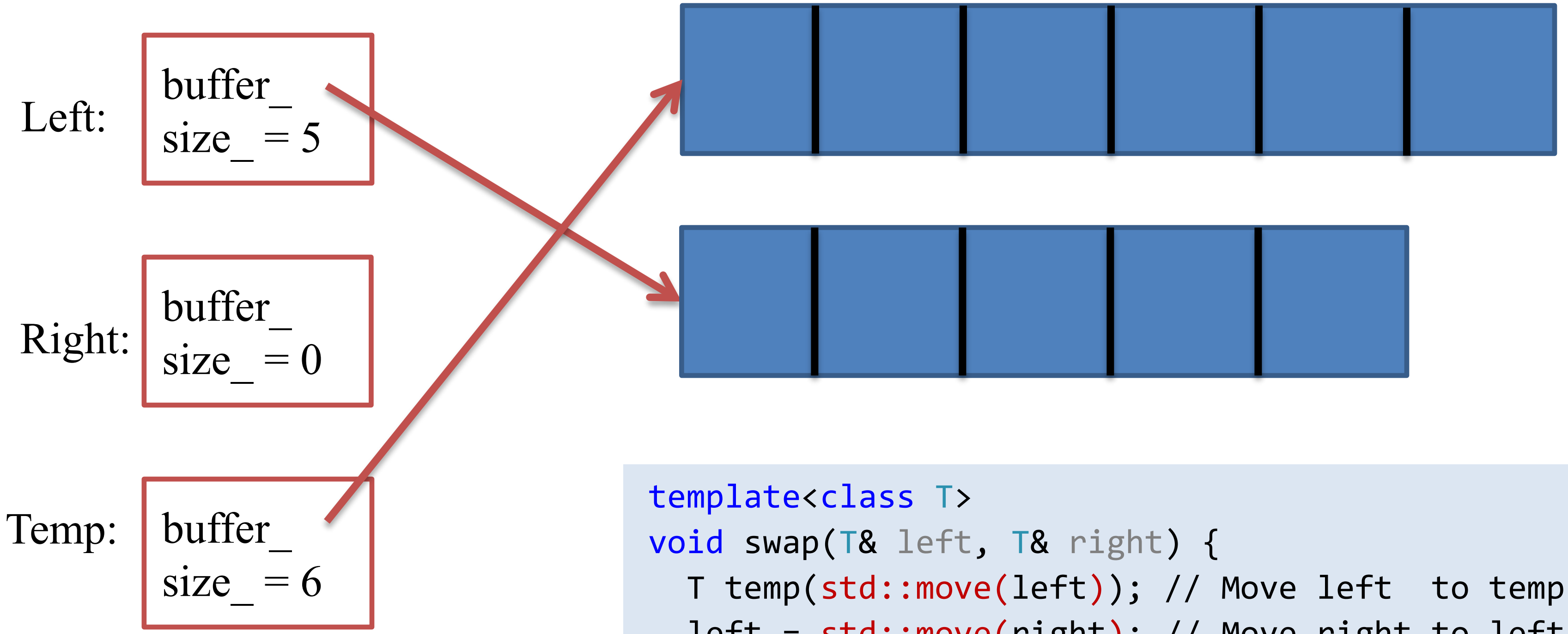
```
T temp(std::move(left)); // Move left  to temp
```




```
template<class T>
void swap(T& left, T& right) {
  T temp(std::move(left)); // Move left  to temp
  left = std::move(right); // Move right to left
  right = std::move(temp); // Move temp  to right
} // Destroy temp (probably has no “real” state
anymore)
```

std::swap example for std::vector

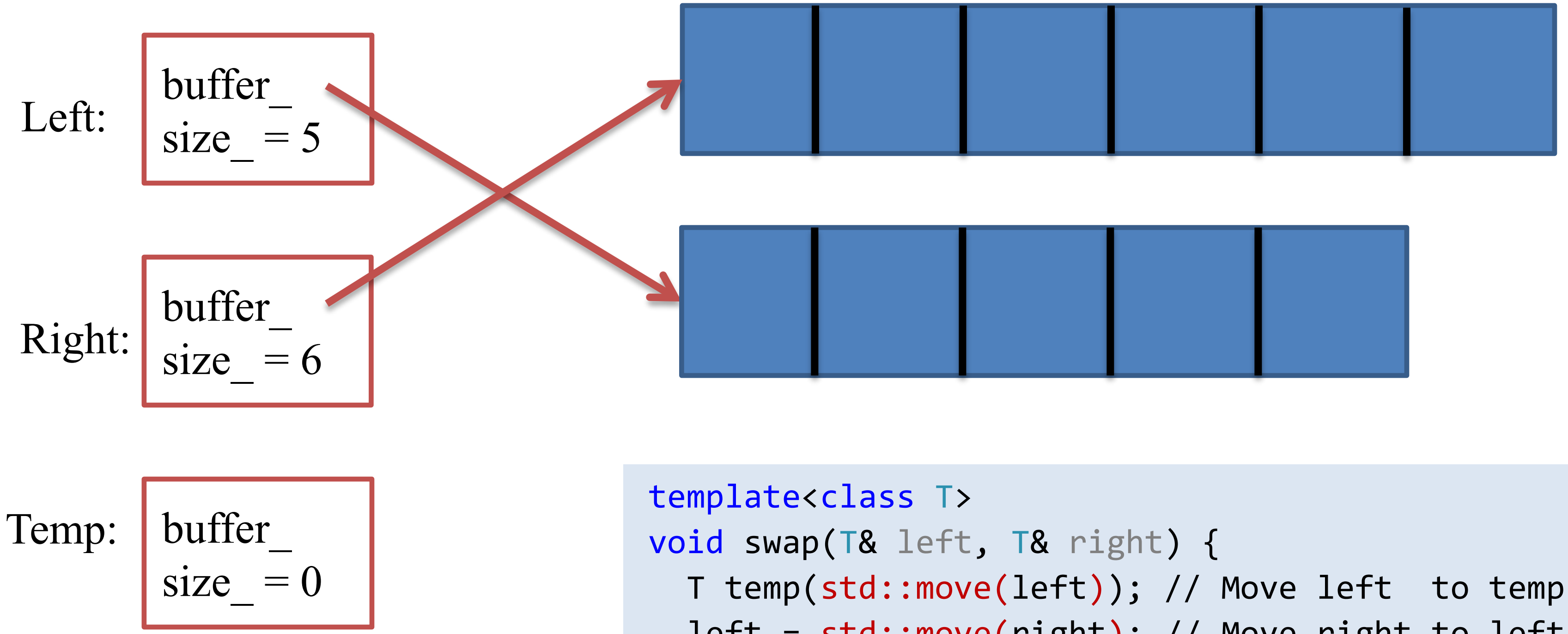
```
left = std::move(right); // Move right to left
```




```
template<class T>
void swap(T& left, T& right) {
  T temp(std::move(left)); // Move left  to temp
  left = std::move(right); // Move right to left
  right = std::move(temp); // Move temp  to right
} // Destroy temp (probably has no “real” state anymore)
```

std::swap example for std::vector

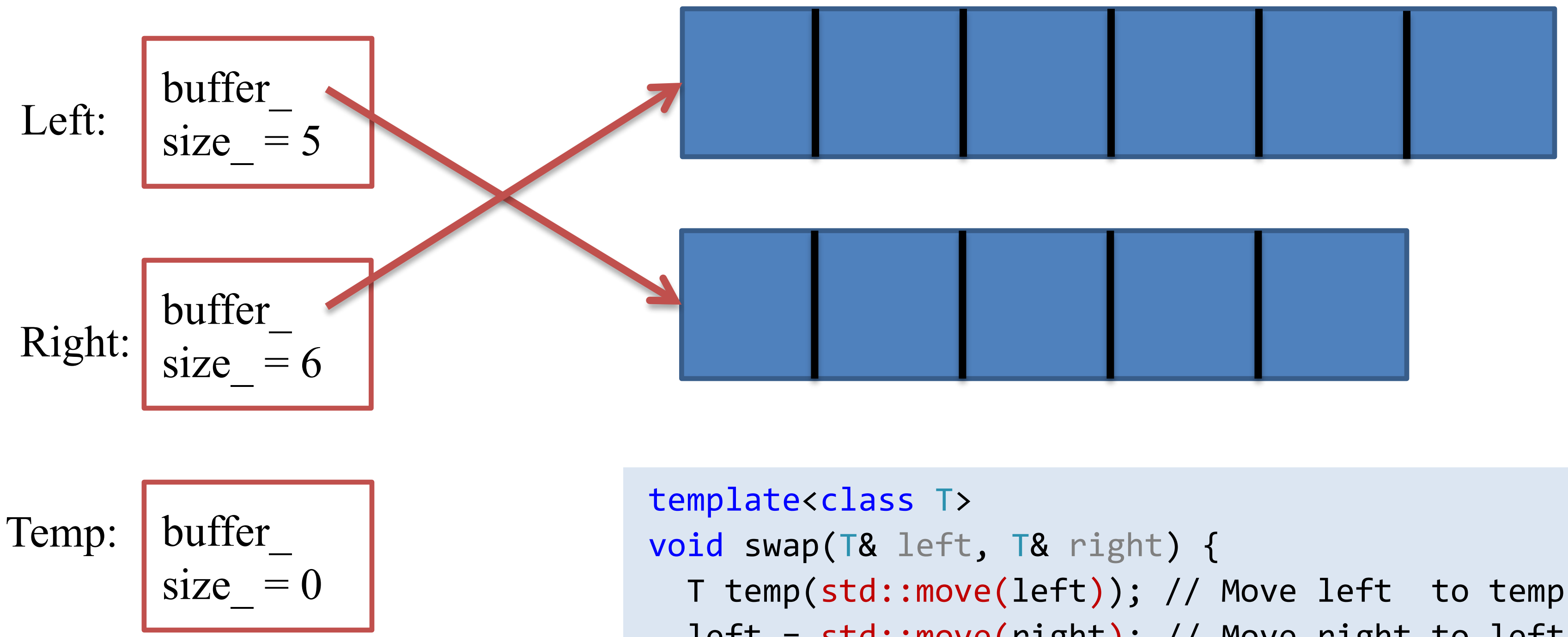
```
right = std::move(temp); // Move temp  to right
```




```
template<class T>
void swap(T& left, T& right) {
  T temp(std::move(left)); // Move left  to temp
  left = std::move(right); // Move right to left
  right = std::move(temp); // Move temp  to right
} // Destroy temp (probably has no “real” state
anymore)
```

std::swap example for std::vector

```
} // Destroy temp (probably has no “real” state anymore)
Move itself not destructive
```




```
template<class T>
void swap(T& left, T& right) {
  T temp(std::move(left)); // Move left  to temp
  left = std::move(right); // Move right to left
  right = std::move(temp); // Move temp  to right
} // Destroy temp (probably has no “real” state
anymore)
```

Importance? What moved? Enabling Move Semantics?

- Moving is important when
  - Copying is expensive
  - Object has data on heap
  - Copying is impossible: `std::ofstream, std::unique_ptr`

- What is moved?
  - Temporary variables
  - Other objects: using std::move
  - If class has move support

- How can we enable move semantics
  - Based on rvalues and rvalue references
  - Move constructor & move assignment operator

Lvalues ⇔ rvalues
Lvalues references ⇔ rvalues references

- Definition rather complicated (with lvalues, glvalues, rvalues, prvalues and xvalues)

- (Not exact but) Workable in practice:
  - Variable has name => lvalue
  - Rvalues: temporary variables without a name

Lvalues ⬄ Rvalues
Lvalues References ⬄ Rvalues References

- Lvalue references: int&, const int&

```
std::string a("test"); // a: lvalue
```

```
std::string& la = a;    // la: lvalue reference to a
```

- Rvalue references: int&&, const int&&

```
std::string f();
f(); // f returns rvalue
```

```
const std::string&& rf = f();
  // rf: const rvalue reference to rvalue
```

- Lvalues can’t bind with rvalues references

Lvalues ⇔ Rvalues – Binding of Variables

Argument type: Function signature	lvalue	const lvalue	rvalue	const rvalue
`f(const std::string&)`	OK	OK	OK	OK
`f(const std::string&&)`	NOK	NOK	OK	OK
`f(std::string&)`	OK	NOK	NOK	NOK
`f(std::string&&)`	NOK	NOK	OK	NOK

- Lvalues can only bind to lvalue references, not rvalue references
- Rvalues can bind to rvalue references and const lvalue references
  - by default to rvalue references

Lvalues ⇔ Rvalues

- We want to move temporaries (rvalues)
  - And nothing else but rvalues

Argument type: Function signature	lvalue	const lvalue	rvalue	const rvalue
`f(const std::string&)`	OK	OK	OK	OK
`f(const std::string&&)`	NOK	NOK	OK	OK
`f(std::string&)`	OK	NOK	NOK (*)	NOK
`f(std::string&&)`	NOK	NOK	OK	NOK

- Rvalue references identify what can be moved

⇒ Overload "copy constructor" and = for non-const rvalue references (const == don't change me)

Add Move Support in Class

```
class MyString
{
public:
  MyString(const char* string)
      : string_(string) {}
private:
  std::string string_;
};

MyString(const MyString&);
MyString& operator=(const MyString&);
MyString(MyString&&) noexcept;
MyString& operator=(MyString&&) noexcept;
```

- Add the followings
  - Move constructor
  - Move assignment operator
- `noexcept`
  - Some STL functions don’t accept throwing move operations

Implementing Move Semantics

```
MyString (MyString&& string) noexcept :
  string_(std::move(string.string_))
{
}
```

std::move facilitates moves

```
MyString& operator=(MyString&& string) noexcept
{
  string_ = std::move(string.string_);
  return *this;
}
```

Implementing Move Semantics - Note

- Moved object has to remain in a (unspecified) **valid** state

```
MyString (MyString&& string)
```

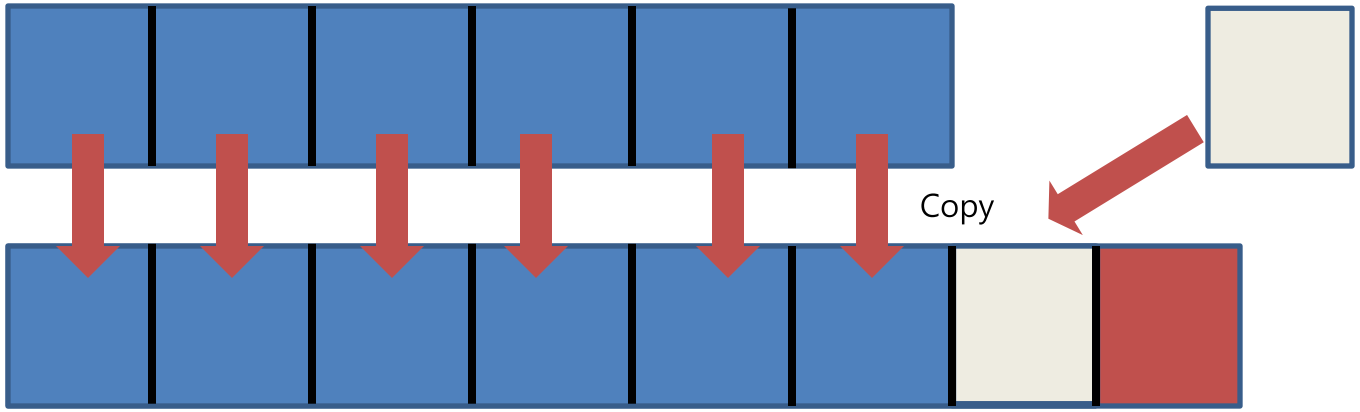
  – string is an lvalue (it has a name)
  – `std::move` has to be called!

```
MyString (MyString&& string)
    : string_(std::move(string.string_))
{
}
MyString& operator=(MyString&& string)
{
  string_ = std::move(string.string_);
  return *this;
}
```

Without `std::move` => Copy!

  – Even with `std::move` might still copy (move operations not (or not correctly) implemented)

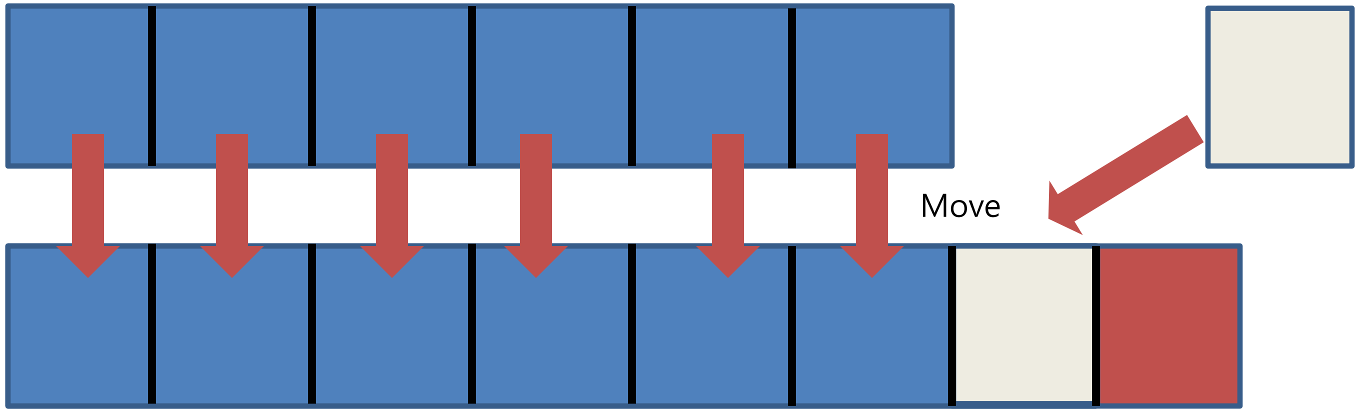
Behavior of std::vector::push_back in C++98



1. if Size == capacity: new buffer
2. Copy elements from existing buffer
3. Copy new element
4. Delete old buffer

- If exception during (1, 2 or 3) => original buffer not changed
  - Strong exception safety guarantee

Erroneous behavior of std::vector::push_back Using Move Semantics



1. if Size == capacity: new buffer
2. **Move** elements from existing buffer
3. **Move**/copy new element
4. Delete old buffer

Implementations should NOT unconditionally move!

- If exception during 2 or 3 => original buffer changed (elements already moved!)
  - Regression w.r.t. C++98

Behavior of std::vector::push_back in C++11

- std::vector::push_back
- Doesn’t call std::move
- But std::move_if_noexcept
  - Only moved if std::move does NOT throw
    - Make move constructor/move assignment operator `noexcept` (if possible)
    - Not copy-constructible => still moved (no regression)

std::move facilitates moves

Quiz #2

- What is the expected output? (including compile/runtime error)

```cpp
#include <iostream>
#include <vector>
using namespace std;
class Move {
    int* data;
public:
    Move() {
        data = new int;
    }
    Move(int d){
        data = new int;
        *data = d;
        cout << "Constructor is called for " << d << endl;
    };
    Move(const Move& source) {
        cout << "Copy Constructor is called -"
             << "Deep copy for " << *source.data
             << endl;
        data = new int;
        *data = *(source.data);
    }
```

```cpp
    Move(Move&& source) : data(source.data) {

        cout << "Move Constructor for "
             << *source.data << endl;
        source.data = nullptr;
    }
};

int main()
{
    Move m(10);
    vector<Move> vec(0);

    vec.push_back(m);
    vec.push_back(Move(20));
    return 0;
}
```

std::move

- Converts lvalues into rvalues references
  - A simple cast
- Doesn’t move anything
  - A a; a = std::move(b);
  - = does the move
- std::move doesn't necessarily lead to a move operation
  - std::move (const object)
  - Class might not implement move operations
  - Move operations possibly not generated

Something Wrong?

```
MyStringVector getDataTrue() {
  MyStringVector result;
  …
  return std::move(result); // move to avoid copy
}
```

```
MyStringVector getDataFalse() {
  return std::move(getReverseData()); // move to avoid copy
}
```

Do NOT Return std::move(…)

```
MyStringVector getDataTrue() {
  MyStringVector result;
  …
  return result;
}
```

```
MyStringVector getDataFalse() {
  return getReverseData();
}
```

Do NOT Return std::move(…)

```
MyStringVector getDataTrue() {
  MyStringVector result;
  …
return std::move(result);
}
```

```
MyStringVector getDataFalse() {
return std::move(getReverseData());
}
```

Move not necessary (*)

- Move is done implicitly if local values are returned

RVO and NRVO cannot be applied with std::move

– Not (obviously) a temporary

* There are a limited number of advanced use-cases where it is useful to return with std::move.

Something Wrong?

```
const MyStringVector getData()
{
  MyStringVector result;
  …
  return result;
}
```

Don’t Return by const Value

```
const MyStringVector getData()
{
  MyStringVector result;
  …
  return result;
}
```

- const means “Do NOT change me”
  - Can’t be moved
  - Compiles, but copy instead of move

```
MyStringVector& operator=(const MyStringVector& vector)
MyStringVector& operator=(MyStringVector&& vector)
```

Something Wrong?

```
std::string a(“Test”)
std::string b = std::move(a);
…
a += “test”;
```

Do NOT Use a Value That Has Been Moved

- Object is in an unspecified valid state

```
std::string a(“Test”)
std::string b = std::move(a);
…
a += “test”;
```

a **might be anything:**
“Test”, “”, “Invalid String”, …
Most likely: “”
Unspecified, but valid