Rvalues and Move Semantics

TOPICS

- LValues and RValues.
- RValue reference operator.
- Compare overload methods based on reference types.
- Describe move semantics and its usefulness.
- Create move constructor and move assignment operator
- Learn the benefits of perfect forwarding.

LVALUE VERSUS RVALUE

- Lvalue can appear on the left of an assignment.
- Lvalue can evaluate to an addressable value
- Rvalue can only appear on the right side of an assignment.
- Rvalue cannot be evaluated to an explicit address

```
int FuncA(){

  return 5;
}

int &FuncB() {

  static int b=10;
  return b;
}

void main() {


  // a is lvalue
  int a = 100;

  // 100 is a rvalue
  100 = a;

  // FuncA is an rvalue
  int c = FuncA();

  //  FuncB is an lvalue
  FuncB() = a;.

}
```

LITERAL CONSTANTS

- Literal constants are an example of Rvalue.
- They are not addressable.
- Any attempt to use a Rvalue as an Lvalue will cause a compiler error.

```
int main()
{
    auto a = 1;
    auto b = 5;

    a = b + 10;
    10 = a;  // wrong
```

(int)10

expression must be a modifiable lvalue

```
}
```

RVALUE REFERENCE OPERATOR

- & indicates an Lvalue reference.
- Lvalue reference is a constant pointer to another object. Lvalue reference can only reference a Lvalue.
- && indicates an Rvalue reference for referencing Rvalues.

```
int a = 5;

int &b = a;

int &&d = 10;
```

OVERLOAD REFERENCE TYPE

- You can overload based on lvalue versus rvalue reference.
- Compilers selects appropriate match.

```
void Func(int &var) {
  cout << "Lvalue reference"
       << endl;
}

void Func(int &&var) {
  cout << "Rvalue reference"
       << endl;
}

int main() {
    int a = 5;
    Func(10);
}
```

UNIVERSAL REFERENCE TYPE

- Reference type within a template can be deduced at compile time.
- Compilers selects appropriate match.
- Provides support for perfect forwarding (discussed later)

```
template <typename T>
void func(T &&t)
{
  cout << t << endl;
}
int main() {
  int a = 5;
  func(a); // Lvalue ref
  func(5); // Rvalue ref
}
```

MOVE SEMANTICS

- Reference type within a template can be deduced at compile time.
- Compilers selects appropriate match.
- Provides support for perfect forwarding (discussed later)

STD::MOVE()

- Objects get silently created and destroyed a lot.

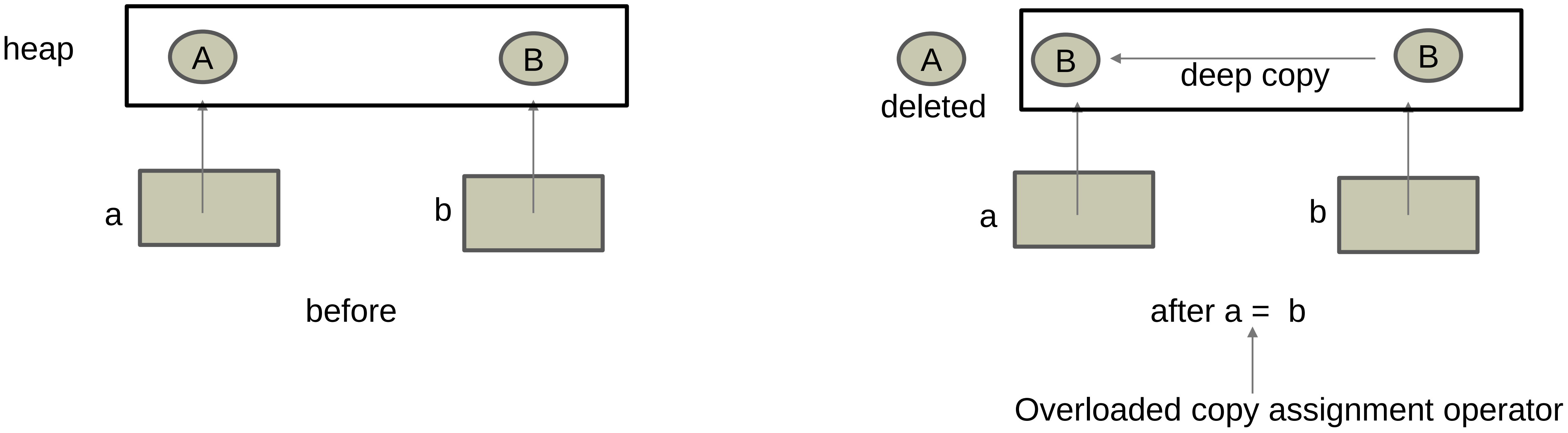
```
template <typename T>
swap(T& a, T& b) {
   T tmp(a);      // second copy of a
   a = b;         // second copy of b (and discarded a copy of a)
   b = tmp;      // second copy of tmp (and discarded a copy of b)
}
```

- Using move allows you to swap the resources instead of copying them.

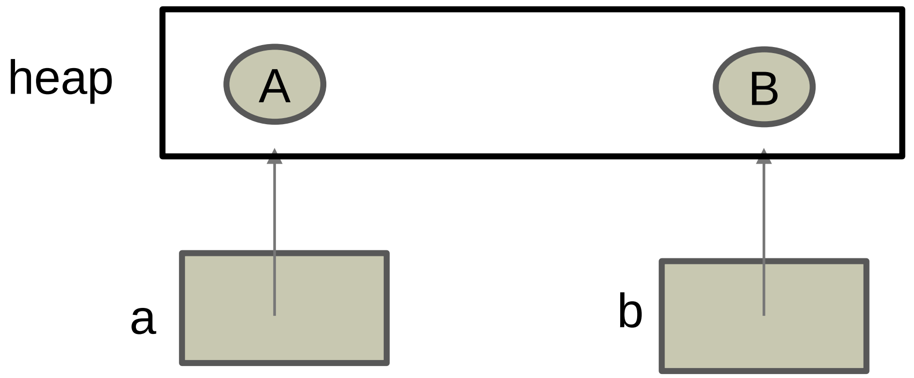
```
template <typename T>
swap(T& a, T& b) {
   T tmp(std::move(a));
   a = std::move(b);
   b = std::move(tmp);
}
```

- std::move() is exactly equivalent to a static_cast to an rvalue reference type.
    - It doesn't move anything.

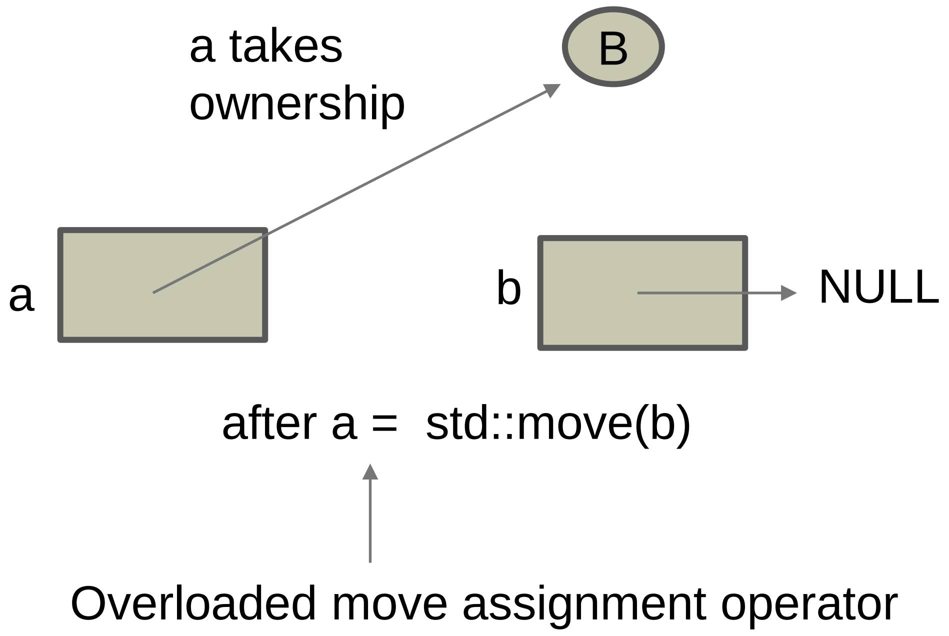
COPY SEMANTICS REVIEW



MOVE SEMANTICS



before



MOVE ASSIGNMENT OPERATOR

- Move assignment operators typically "steal" the resources held by the argument (e.g. pointers to dynamically-allocated objects)

```
Big(Big &&rhs)
{
  if (this != &rhs) {
    // Free existing resource.
    delete _data;
    // Copy ptr and length from source object.
    _data = rhs._data;
    _length = rhs._length;
    // Release ptr from the source object.
    rhs._data = nullptr;
    rhs._length = 0;
   }
return *this;
}
```

MOVE CONSTRUCTOR

- The move constructor transfers ownership of the managed resource from the source into the current object.
- It is typically called when an object is initialized from an rvalue

```
Big(Big &&rhs)
{
   *this = std::move(rhs);
}
```

COMPILER DEFAULTS

- The compiler will implicitly declare a move constructor if all of the following are true:
  - there are no user-declared copy constructors;
  - there are no user-declared copy assignment operators;
  - there are no user-declared move assignment operators;
  - there is no user-declared destructor.

- The compiler will implicitly declare a move assignment operator if all the following are true:
  - there are no user-declared copy constructors
  - there are no user-declared move constructors
  - there are no user-declared copy assignment operators
  - there is no user-declared destructor

THE RULE OF FIVE

- The rule of three specifies that if a class implements any of the following functions, it should implement all of them:
  - copy constructor
  - copy assignment operator
  - destructor
- The rule of five identifies that it usually appropriate to also provide the following functions to allow for optimized copies from temporary objects:
  - move constructor
  - move assignment operator

PERFECT FORWARDING

- Like move semantics, perfect forwarding reduces overhead associated with a function call. Often, a function call is essentially a delegate to another function.
- Calling FuncB is essentially a call to FuncA. However, there is additional overhead of two pass by value calls instead of one pass by value call. If obj is a heavy object, the additional overhead could be considerable.

```
/*
In this example 3 copy by value constructors
are called
*/
class Foo {
public:
    Foo() { cout << "Regular ctor" << endl; }
    Foo(const ClassA & obj) {
        cout << "Regular ctor" << endl;
    }
};

void FuncA(ClassA obj) { }
void FuncB(ClassA obj) { FuncA(obj); }

int main() {
    Foo obj;
    FuncB(obj);

    return 0;
}
```

STD::FORWARD()

- Perfect forwarding removes the potential additional overhead of functions that are thin wrappers for delegating to another function.
- Perfect forwarding is accomplished with a combination of move semantics and std::forward to forward parameters through a thin wrapper.

```
template<typename T>
void Func(T b) {
    std::cout << "Func " << b.data() << std::endl;
}

template<typename T>
void Wrapper(T&& b) {
    Func<T>(std::forward<T>(b));  // Forward as lvalue or
as rvalue, depending on T
}
```