

# Rvalues and Move Semantics

---

**Chapter 17**



# Objectives

- Describe the difference between LValues and RValues.
- Use RValue reference operator.
- Compare overload methods based on reference types.
- Describe move semantics and its usefulness.
- Gain experience through code walk-throughs.
  - The example programs are in the [chapter directory](#).



# 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() {
    int a = 100;    // a is lvalue
    100 = a;        // 100 is a rvalue
    int c = FuncA(); // FuncA is rvalue
    FuncB() = a;    // FuncB is an lvalue.
}
```

# 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.

```
6
7 int main()
8 {
9     auto a = 1;
10    auto b = 5;
11
12    a = b + 10;
13    10 = a; // wrong
14
15
16 }
17
18
```

(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.

// Which overloaded method is called in the following code?

```
void Func(int &var) {  
    cout << "Lvalue reference" << endl;  
}  
void Func(int &&var) {  
    cout << "Rvalue reference" << endl;  
}  
int main() {  
    int a = 5;  
  
    Func(a); // Lvalue ref  
    Func(10); // Rvalue ref  
}
```



# 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)
{
    // "T &&" is a UNIVERSAL REFERENCE
    cout << t << endl;
}

int main() {
    int a = 5;
    func(a);    // Lvalue reference
    func(5);    // Rvalue reference
}
```



# Copy vs Move Semantics

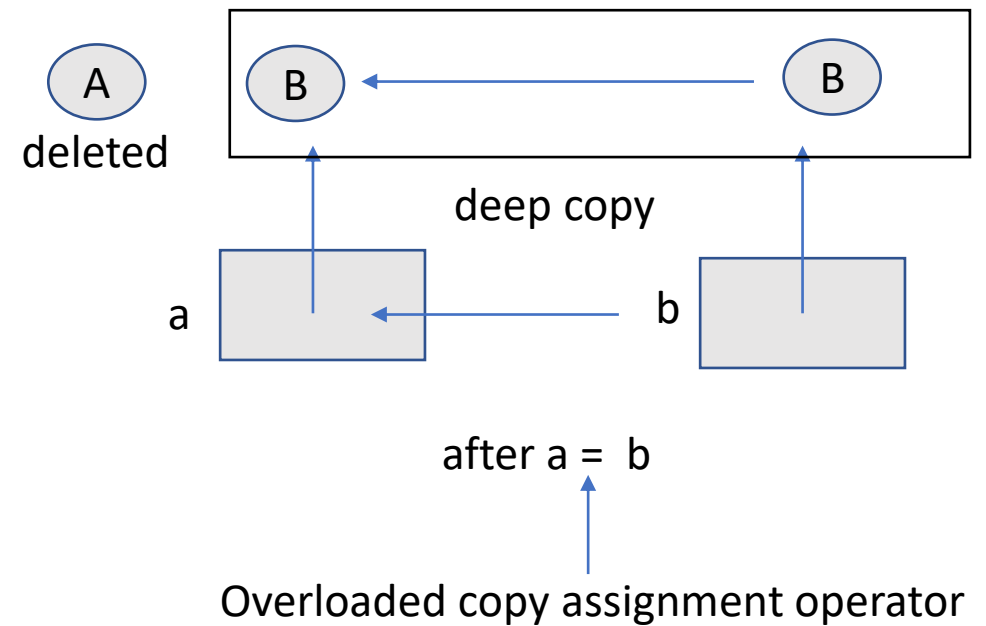
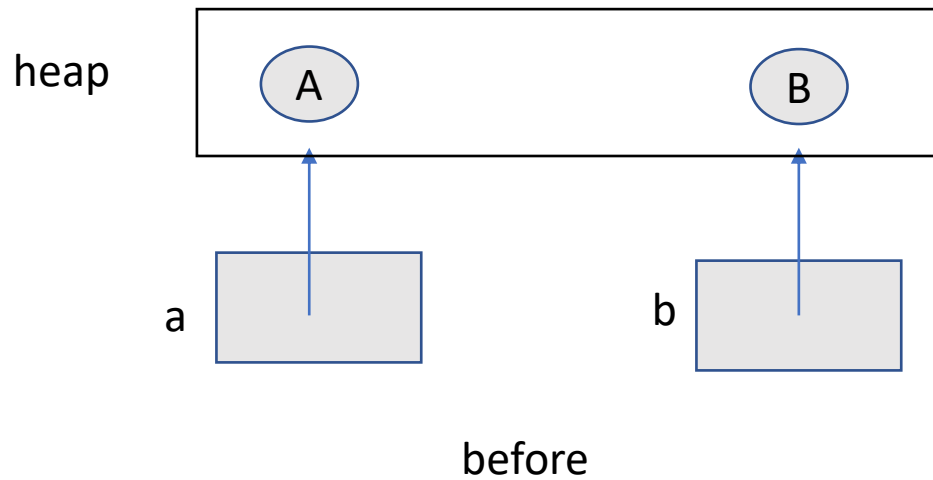
- **Copy and assignment can be expensive – especially temporary objects pointing to a lot of data. This can adversely impact performance and resource utilization.**

```
template <typename T>
swap(T& a, T& b) {
    T tmp(a);    // copy constructor
    a = b;        // copy assignment
    b = tmp;      // copy assignment
}
```

- **Move semantics allows an object, under certain conditions, to take ownership of some other object's external resources.**
- **If an object does not manage at least one external resource (either directly, or indirectly through its member objects), move semantics will not offer any advantages over copy.**



# copy semantics review



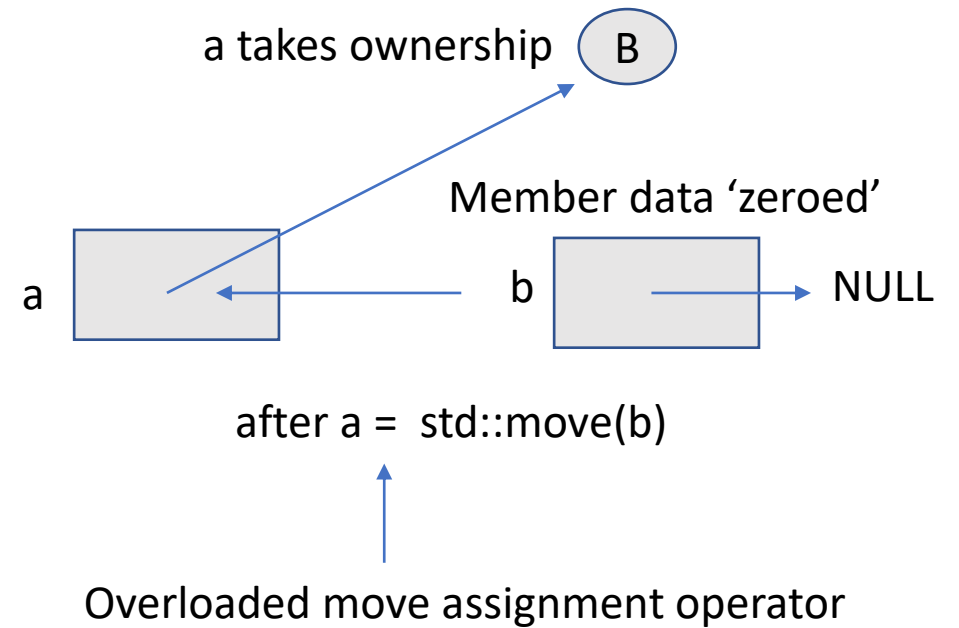
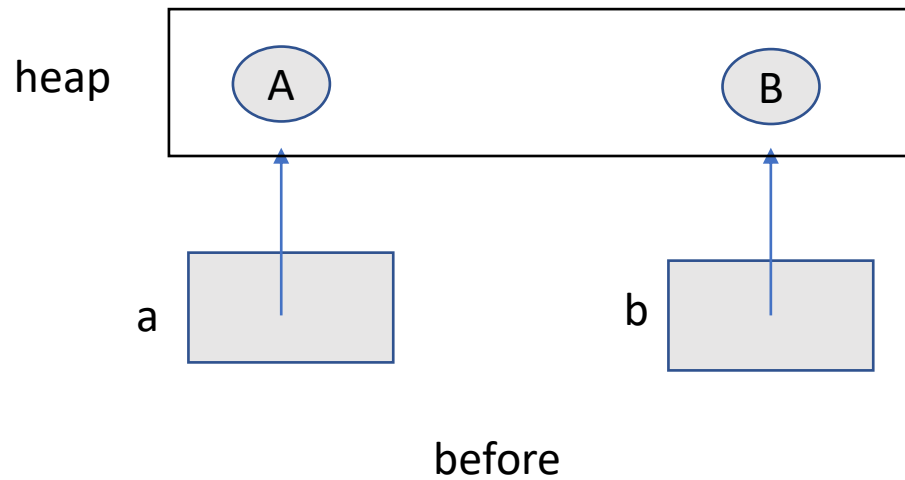


# std::move

- std::move() is a cast that produces an rvalue-reference to an object.
- Using std::move allows you to swap the resources instead of copying them.

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

# move assignment



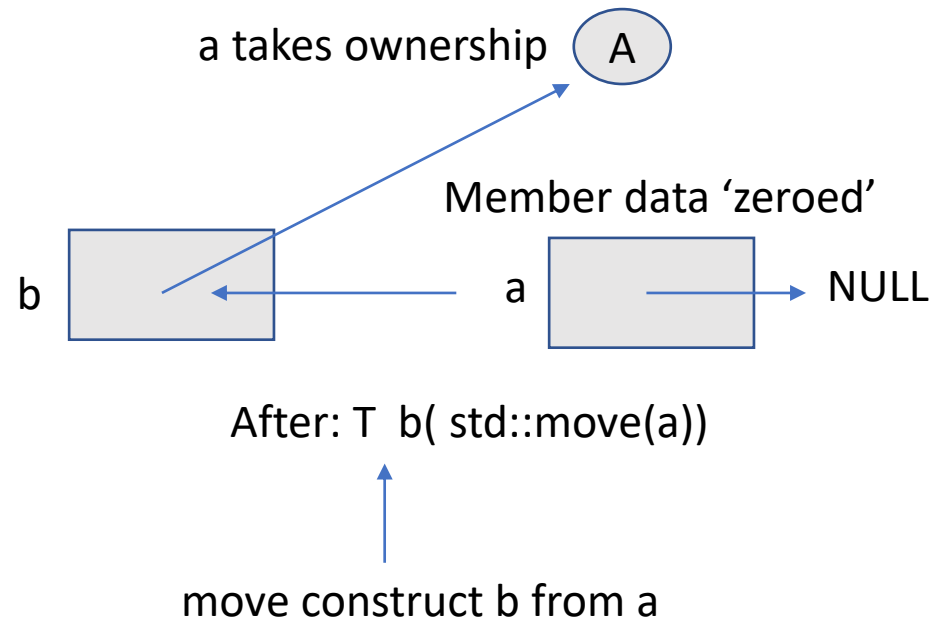
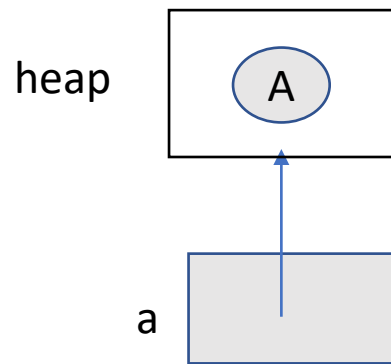


# Move assignment operator

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

```
Data& operator=(Data&& rhs) { // argument is not const
    if (this == &rhs) return *this; // don't destroy yourself!
    delete _data; // release what we hold
    // scavenge rhs
    _data = rhs._data;
    rhs._data = nullptr;
    return *this; // return this instance by ref
}
```

# Move construction





# Move constructor

- The purpose of a move constructor purpose is to transfer ownership of the managed resource from the source into the current object.
- The move constructor is typically called when an object is initialized (by direct-initialization or copy-initialization) from an rvalue
- 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.

```
/*  
You can eliminate redundant code by writing the move constructor to call the move  
assignment operator as shown in the example.  
*/  
Data(Data&& rhs) {  
    *this = std::move(rhs);  
}
```



# 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 is 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.
- Here three copy by value constructors are called.

```
class ClassA {  
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() {  
    ClassA 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
}
```



# Summary

- In C++11, in addition to copy constructors, objects can have move constructors.
- And in addition to copy assignment operators, they have move assignment operators.
- The move constructor is used instead of the copy constructor, if the object has type "rvalue-reference" (Type &&).
- `std::move()` is a cast that produces an rvalue-reference to an object, to enable moving from it.