## 5. Move Semantics

A close-up of a list of values

Description automatically generated

A screenshot of a computer program

Description automatically generated

# R-value and L-values

R-values and L-values are important concepts in C++ that determine the assignability and lifetime of expressions:

**1. Definition:**

* **L-value:** An L-value refers to an object that occupies some identifiable location in memory (i.e., it has an address). L-values can appear on the left-hand side of an assignment operation and can be assigned new values.
* **R-value:** An R-value refers to a temporary value that does not persist beyond the expression in which it is created. R-values typically appear on the right-hand side of an assignment operation and cannot be assigned new values.

**2. Examples:**

* **L-values:** Variables like **x**, **y**, and **z** are L-values because they have a name and can be assigned new values. For example, **x = 5;** assigns the value **5** to the variable **x**.
* **R-values:** Numeric literals like **5**, **10**, and **8** are R-values because they are temporary values and cannot be assigned new values. They cannot appear on the left-hand side of an assignment operation.
* Expressions like **(x + y)** also return R-values because the result is temporary and does not persist beyond the expression.

**3. Pre-increment Operator:**

* When the pre-increment operator (**++x**) is applied to a variable **x**, the expression returns an L-value because it refers to the variable itself. Thus, it can appear on the left-hand side of an assignment operation.

**4. Function Return Types:**

* Functions that return by value, like **Add()**, return R-values because they return temporary values that cannot be assigned new values.
* Functions that return by reference, like **Transform()**, return L-values because they return a reference to an object with an identifiable memory location. This means that the returned value can be assigned new values.

**5. Summary:**

* R-values are temporary values that cannot be assigned new values and typically appear on the right-hand side of assignments.
* L-values are objects with identifiable memory locations that can be assigned new values and can appear on the left-hand side of assignments.

Understanding the distinction between R-values and L-values is crucial for writing correct and efficient C++ code, especially when dealing with expressions, assignments, and function return types.

# Summary

1. **L-value:**

* An L-value represents an object that occupies some identifiable location in memory. It has a name.
* L-values can appear on the left-hand side of an assignment operator (**=**) and can be assigned new values.
* Variables, objects, and expressions that refer to memory locations are considered L-values.
* Example: Variables **x**, **y**, and **z** are L-values because they have names and can be assigned values.

1. **R-value:**

* An R-value represents a temporary value that does not persist beyond the expression in which it appears.
* R-values do not have a name or an identifiable memory location.
* R-values typically represent literal constants, temporary results of expressions, or unnamed temporaries created during expression evaluation.
* R-values cannot appear on the left-hand side of an assignment operator (**=**) because they do not represent assignable memory locations.

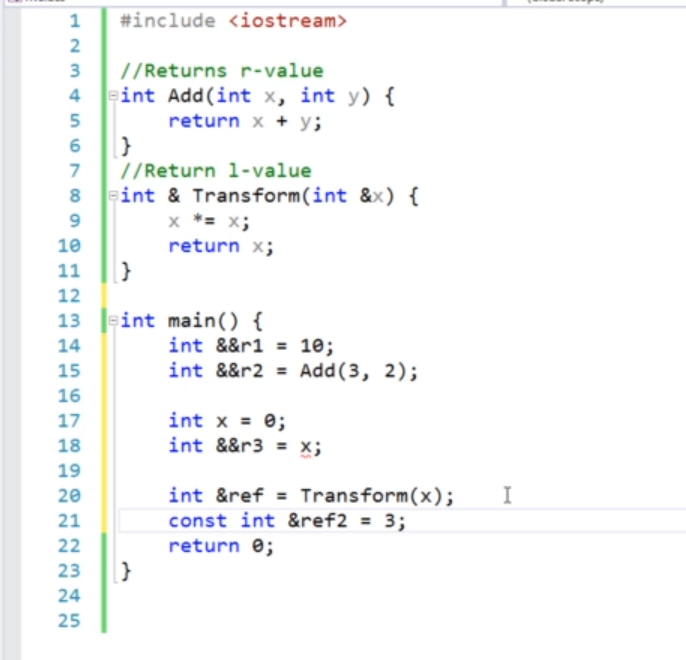
1. **Usage in Expressions:**
   * Some expressions return L-values, while others return R-values.
   * Expressions involving variables, pre-increment operators, and function calls that return by reference typically result in L-values.
   * Expressions that involve literals, arithmetic operations, or function calls that return by value typically result in R-values.

Top of Form

# R-value references

A white background with black text

Description automatically generated



// R -value reference Can not bind to L -values

/\* L -value reference reference may bind to a temporary if the reference is constant. \*/

**1. Purpose of R-value References:**

* R-value references allow programmers to detect temporaries in expressions and provide special handling for them.
* They enable the creation of functions that can distinguish between temporary (R-value) and non-temporary (L-value) objects.

**2. Overloading Functions Based on References:**

* To demonstrate the usage of R-value references, we can write overloaded functions that accept different types of references.
* One function can accept an L-value reference, another can accept a constant L-value reference, and a third can accept an R-value reference.

**3. Invocation of Functions:**

* When calling these overloaded functions with arguments, the compiler will select the appropriate function based on the value passed.
* Temporaries (R-values) will bind to the function that accepts an R-value reference.
* Non-temporary objects (L-values) can bind to functions that accept L-value references or constant L-value references.

**4. Example:**

A screenshot of a computer code

Description automatically generated

**5. Behavior with Overloading:**

* When overloading functions, the temporary (R-value) argument will always bind to the function that accepts an R-value reference.
* It will not bind to the function that accepts a constant reference because the temporary can be efficiently moved or modified using the R-value reference.

A screenshot of a computer program

Description automatically generated

# Copy Semantics and Move Semantics

**Copy Semantics:**

* Copy semantics involve creating a copy of an object through its copy constructor.
* Copies can be either deep or shallow. Deep copy is necessary for classes containing pointers or other resources.
* In some cases, copies need to be created from temporaries, such as when a function returns a user-defined object by value.
* Move semantics offer an alternative to deep copying, allowing the state of a source object to be transferred to a target object without allocating new resources.

**Move Semantics:**

* In move semantics, the state of a temporary object (created as a result of some expression) can be moved into another object.
* Rather than performing a deep copy, the pointer in the target object can point to the same address as the pointer in the source object.
* This approach avoids issues such as dangling pointers and is generally faster than deep copying, as it doesn't involve allocating new memory.
* Move semantics are particularly useful for optimizing performance when dealing with temporary objects that need to be copied.

**Decision Criteria:**

* Whether to use move semantics or copy semantics depends on whether an expression yields a temporary that needs to be copied to another object.
* Temporary objects can be detected by implementing a constructor that accepts an R-value reference.
* A constructor that accepts an R-value reference is called a move constructor, and it automatically binds to temporary objects.
* Implementing move semantics can significantly improve performance compared to deep copying, especially for operations involving temporary objects.

**Implementation:**

* Implementation of move semantics involves stealing resources from the source object and transferring them to the target object.
* This is typically achieved by implementing a move constructor that performs a shallow copy and updates pointers accordingly.
* The move constructor is automatically invoked for temporary objects, facilitating efficient resource management.

Overall, understanding and leveraging move semantics can lead to more efficient resource management and improved performance in C++ programs, especially when dealing with temporary objects.

A white background with black text

Description automatically generated

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

**Example**

* In copy semantics, objects are copied using a copy constructor.
* Copies can be deep or shallow. Deep copy is necessary for classes with pointers or other resources.
* Sometimes, copies need to be made from temporaries, like when a function returns a user-defined object by value. In such cases, move semantics can be used to transfer the state from the source object to the target object.
* Consider an object obj1 with attributes v and ptr, where v is a value type and ptr is a pointer. To copy obj1 into obj2, the value of v is directly copied. For a deep copy of the pointer, new memory must be allocated and the value copied there.
* If obj1 is a temporary object created during the execution of an expression, and another object obj2 needs to be created, instead of deep copying, obj2's pointer can be made to point to the same address as obj1's pointer, akin to a shallow copy.
* However, this can lead to problems if one of the objects is destroyed and frees the memory, leaving the other with a dangling pointer.
* To mitigate this, after initializing obj2's pointer with the same address as obj1's, we assign NULL to obj1's pointer.
* This way, when obj1 is destroyed, its destructor will delete a null pointer, which is ignored by the runtime, effectively transferring resources from obj1 to obj2.
* This approach is faster than deep copying, as it avoids the allocation of new memory and copying of resources.

**Approach for deciding between move semantics and copy semantics in C++.**

1. **Consider Temporaries**:
   * If an expression generates a temporary object that needs to be passed to another object or function, move semantics can be beneficial.
   * Temporaries often occur in return statements or within expressions where objects are created on the fly.
2. **Detecting Temporaries**:
   * By default, when a temporary needs to be copied, the copy constructor is invoked.
   * You can detect temporaries by implementing a constructor that accepts an rvalue reference, denoted by **&&**. This constructor is known as a move constructor.
   * Temporaries automatically bind to move constructors, allowing you to implement efficient move operations.
3. **Move Semantics in Move Constructors**:
   * In move constructors, instead of copying the temporary, you move its resources, such as memory or ownership, to the destination object.
   * This operation can often be more efficient than copying, especially when dealing with large or resource-heavy objects.

# The ownership semantics

1. **Class without Ownership Semantics**:
   * In C++, a class without ownership semantics typically means that the class does not manage the lifetime of its member objects.
   * Such a class does not control the allocation and deallocation of its member variables, and it does not take ownership or responsibility for memory management.
2. **Ownership Semantics**:
   * Ownership semantics in programming refer to the rules and conventions regarding the responsibility for managing the lifecycle and memory of objects.
   * In languages like C++, where manual memory management is possible, ownership semantics are crucial to avoid memory leaks, dangling pointers, and other memory-related issues.
3. **Rule of Five**:
   * The Rule of Five states that if a class has ownership semantics (i.e., it manages resources like dynamically allocated memory), then it should implement or disable all functions related to resource management: the destructor, copy constructor, copy assignment operator, move constructor, and move assignment operator.
   * This ensures that resources are properly managed, and the class behaves correctly when copied or moved.
4. **Rule of Zero**:
   * If a class does not have ownership semantics and it does not acquire any resources, then there's no need to implement any of the functions from the Rule of Five.
   * Instead, this scenario follows the Rule of Zero, which means letting the compiler generate the default destructor, copy constructor, copy assignment operator, move constructor, and move assignment operator.

**In general, ownership semantics define the following aspects:**

1. Creation: It specifies who is responsible for creating an object and initializing its state. Ownership is typically established when an object is created.
2. Lifetime: It defines the duration for which an object remains valid and accessible. Objects are typically destroyed when they are no longer needed.
3. Destruction: It determines who is responsible for deleting an object and releasing any resources associated with it. Destruction typically happens when an object's lifetime ends.
4. Transfer of Ownership: It specifies how ownership of an object can be transferred from one entity to another. Ownership transfer typically involves passing an object from one scope, function, or class to another.

**Note:**

If you need to implement another constructor then delete constructors can be Provided by this method using default. This will cause the compiler to synthesize a default implementation of the move assignment. And implementation will internally call the move assignment of the Integer class. These are considered custom implementations

A screenshot of a computer program

Description automatically generated

A white document with black text

Description automatically generated

A white paper with black text

Description automatically generated

A white paper with black text

Description automatically generated

A table with text on it

Description automatically generated

**Rule of Five:**

1. **Definition**: The Rule of Five is a guideline in C++ that states if a class has ownership semantics, it must provide a user-defined destructor, copy constructor, copy assignment operator, move constructor, and move assignment operator.
2. **Purpose**: Ensures proper handling of resources (e.g., file handles, sockets, memory) acquired by a class, especially during object copying, moving, and destruction, to prevent leaks.
3. **Implementation**:

* Destructor releases the acquired resource.
* Copy constructor creates a copy of the underlying resource.
* Copy assignment operator performs similar tasks as the copy constructor.
* Move constructor moves the underlying resource from one object to another.
* Move assignment operator ensures proper resource transfer between objects.

**Rule of Zero:**

1. **Definition**: If a class does not have ownership semantics (i.e., it does not acquire any resource), then no functions from the Rule of Five need to be implemented.
2. **Implementation**: The compiler automatically synthesizes the necessary functions if no custom implementations are provided. This is referred to as the Rule of Zero.

**Synthesizing Functions:**

1. **Automatic Synthesis**: When no custom implementations are provided, the compiler synthesizes necessary functions like copy constructors, copy assignment operators, move constructors, move assignment operators, and destructors.
2. **Default Specifiers**: The **default** specifier can be used to instruct the compiler to synthesize default implementations of functions like move assignment, even if other functions have custom implementations.

**Impact of Custom Implementations:**

1. **Deleted Functions**: Providing custom implementations of certain functions can result in the deletion of other related functions (e.g., providing a move constructor may delete the copy operations).
2. **Manual Synthesis**: In cases where some functions are deleted due to custom implementations, the remaining functions can be manually synthesized using default specifiers.

**Overview of Function Synthesis:**

1. **Provided Implementation**: Indicates which functions have custom implementations.
2. **Impact on Synthesis**:

* Custom implementations affect the synthesis of other functions.
* Move operations are often deleted when custom implementations are provided.

**Conclusion:**

1. **Guidelines**: Developers should understand when to apply the Rule of Five and the Rule of Zero based on ownership semantics.
2. **Automatic Synthesis**: The compiler can automatically synthesize functions unless custom implementations are provided.
3. **Best Practices**: Implementing the Rule of Zero is ideal for classes without ownership semantics, minimizing manual implementation and potential errors.

# Copy Elision

* Copy Elision is a technique that is used by the compiler to eliminate temporary objects.
* Elide (verb): to omit or leave out
* Elision (noun): the act of omission
* Note that for copy elision to work, the class should have the copy and the move constructors.

A screenshot of a computer code

Description automatically generated

A screenshot of a computer

Description automatically generated

Copy elision is a fascinating optimization technique employed by C++ compilers to enhance the efficiency of code execution.

1. **Copy Elision Explained**: Copy elision is a strategy used by C++ compilers to avoid unnecessary copying of objects. When a temporary object is created and then copied elsewhere in the code, the compiler may skip the copy step altogether, resulting in improved performance.
2. **Example Illustration**: To illustrate copy elision, let's consider a scenario where a temporary object is created and then returned from a function. In traditional circumstances, the temporary object would be copied during the return process. However, with copy elision, the compiler may optimize this process and directly construct the object at its final destination, omitting the unnecessary copy operation.
3. **Automatic Implementation**: Some compilers, like Visual Studio, automatically implement copy elision for temporary objects. This means that developers using these compilers may not explicitly see copy operations being skipped, as the optimization is applied automatically by the compiler.
4. **Switch to Linux**: To demonstrate copy elision in a different environment, we switches to Linux. This emphasizes the universality of the copy elision optimization technique across different compiler platforms.
5. **Code Modification for Demonstration**: modifies the code to showcase how copy elision works in practice. By manipulating the code to involve the creation and return of temporary objects, we effectively demonstrates how copy elision optimizes the process.
6. **Explanation of Terminology**: The meaning of terms like "elide" and "elision," providing a foundational understanding for comprehending the concept of copy elision.
7. **Compiler Behavior**: The behavior of compilers with regard to copy elision may vary depending on optimization settings and compiler versions. Some compilers may automatically apply copy elision, while others may require specific optimization flags to enable it.
8. **Named Return Value Optimization (NRVO)**: Named Return Value Optimization (NRVO) is mentioned as another form of copy elision. This optimization technique specifically optimizes the creation and return of objects from functions by directly constructing them in their final destination, eliminating the need for unnecessary copies.
9. **Return Value Optimization (RVO)**: Return Value Optimization (RVO) is highlighted as another optimization technique related to copy elision. Like copy elision, RVO aims to improve code efficiency by optimizing object creation and return processes.
10. **C++17 Changes**: Changes related to copy elision in C++17, indicating ongoing improvements and optimizations in the C++ language standard.

In summary, copy elision is a powerful optimization technique that helps improve the efficiency and performance of C++ code by eliminating unnecessary object copies. By understanding and leveraging copy elision, developers can write more efficient and streamlined C++ programs.Top of Form

# Std move

1. **Using std::move to Force Move Semantics**: When you use **std::move**, you're explicitly indicating to the compiler that you want to use move semantics instead of copy semantics. This is beneficial when you no longer need the state of an object and want to transfer its resources to another object. It's also useful for objects that are non-copyable, such as file streams.
2. **Benefits of Using std::move**:
   * It moves the object to the destination instead of making a copy, which can be more efficient, especially for large or resource-intensive objects.
   * After moving, the original object is in a valid but unspecified state, typically set to null or some other default value, which prevents accessing it accidentally and causing runtime errors.
   * You can reassign a new value to the original object after moving its resources, but you need to ensure proper memory allocation if needed.

Overall, **std::move** is a powerful tool for efficient resource management and avoiding unnecessary copying of object

A computer screen shot of code

Description automatically generated

* Will move the object to val in (Print(std::move(a));) instead of copy, as we will not use a again, and after val is destroyed at the end of the function it will release the resource.
* After that can not read from a. This will lead to a run time error because when we perform a move the constructor will assigned to NULL in the move constructor execution.
* But can reassign a new value to a but you should make sure that in the member function, you will allocate new memory for that object.

The essence of using move semantics rather than copy semantics lies in optimizing resource management:

1. Efficient Resource Transfer: Move semantics allow objects to transfer their resources (such as dynamically allocated memory, file handles, or unique pointers) from one object to another, without the need for costly deep copying. This transfer is typically done by "stealing" the resources from the source object and leaving it in a valid but unspecified state.
2. Avoiding Redundant Copies: In situations where a copy of an object is only needed temporarily or when the source object is about to be discarded, move semantics can significantly reduce unnecessary copying overhead. Instead of making a duplicate copy of the object's resources, move semantics enable efficient transfer of those resources to a new object.
3. Performance Optimization: By eliminating unnecessary copying of large or expensive-to-copy objects, move semantics can lead to performance improvements, especially in scenarios where objects are frequently passed or returned by value.
4. Enabling Move-Only Types: Move semantics enable the creation of move-only types, which cannot be efficiently copied but can be efficiently moved. This allows for the design of types that represent unique resources or states that should not be duplicated, such as unique locks, unique file handles, or unique network connections.

In summary, the essence of using move semantics is to optimize resource management and improve performance by efficiently transferring ownership of resources between objects, avoiding redundant copies, and enabling the creation of move-only types for representing unique resources.

1. **Primitive Types and Parameter Passing**:
   * When passing parameters to functions or methods, they can be passed by value (copying the parameter) or by reference (passing a reference or pointer to the parameter).
   * Parameter types can include primitive types (integers, floats, etc.), user-defined types (classes, structs), or references/pointers to these types.
2. **Redundancy of Move on Parameter Types**:
   * Since parameter types typically don't own resources (they are just copies or references to existing objects), applying move semantics to them is redundant.
   * Even if you attempt to move a parameter type, it will still be copied because there's nothing to move.
3. **Optimizing Resource Management**:
   * Move semantics can optimize resource management by transferring ownership, which is beneficial for classes or types that manage resources such as dynamically allocated memory.
   * However, applying move semantics to parameter types doesn't yield any benefits because parameters are typically just copies or references to existing objects, rather than owning their resources.

In summary, move semantics on parameter types is redundant because parameters usually don't own resources. It's more relevant and beneficial for types that manage resources and need to transfer ownership efficiently.

Top of Form

* **Purpose of std::move**:
  + **std::move** is a standard library function in C++ used to indicate that an object's value should be treated as an rvalue, enabling move semantics instead of copy semantics.
* **Difference from Copy Semantics**:
  + While regular copy operations create duplicate objects, move semantics transfer ownership of resources from one object to another, typically more efficiently.
* **Example and Explanation**:
  + The example demonstrates a scenario where an object **E** is copied into object **B**, but the intention is to move its state instead of creating a copy.
  + By default, due to function overload resolution, the compiler chooses copy semantics, resulting in unnecessary copies.
  + To force move semantics, **std::move** is applied to the object, indicating the intention to move its state rather than copying.
  + While a custom type cast could achieve the same effect, **std::move** provides clearer intent to readers and maintains readability.
* **Named Return Value Optimization (NRVO)**:
  + Mentioned the optimization technique NRVO, where the return value of a function is constructed directly in the caller's context, avoiding unnecessary copies or moves.
* **Usage with Non-Copyable Objects**:
  + **std::move** is especially useful with non-copyable objects, which cannot be duplicated using copy semantics.
  + It facilitates transferring ownership of resources without triggering copy operations, enhancing performance and resource management.
* **Conclusion**:
  + **std::move** is a crucial tool in C++ for enabling move semantics, optimizing resource management, and improving code efficiency.
  + It should be applied judiciously, especially with non-copyable objects and situations where move semantics are preferred over copy semantics.

Top of Form

A close-up of a computer code

Description automatically generated

Both copy constructors and move constructors are special member functions in C++ that are used for initializing objects. However, they serve different purposes and are invoked under different circumstances:

1. **Copy Constructor (Integer(const Integer& obj))**:

* A copy constructor is used to create a new object as a copy of an existing object.
* It takes a reference to a constant object (**const Integer& obj**) as its parameter, indicating that the object is being copied from and should not be modified during the copy process.
* It performs a deep copy of the object's data, allocating new memory if necessary, to ensure that the new object is independent of the original.
* It is invoked when:
* An object is passed by value as a function argument.
* An object is returned by value from a function.
* An object is explicitly copied using the copy initialization syntax (**Integer obj2 = obj1;**).

1. **Move Constructor (Integer(Integer&& obj) noexcept(true))**:
   * A move constructor is used to transfer the resources (such as dynamically allocated memory) owned by a temporary or expiring object to another object.
   * It takes an rvalue reference to an object (**Integer&& obj**) as its parameter, indicating that the object is an expiring temporary and its resources can be moved from.
   * It typically performs shallow copying of data and transfers ownership of resources (e.g., pointers) from the source object to the destination object.
   * It is invoked when:
     + An object is explicitly moved using the move constructor or move assignment operator (**Integer obj2 = std::move(obj1);**).
     + An object is returned by value from a function, and the return statement involves a temporary object (e.g., creating a temporary object within the return statement).
     + An object is being constructed from an rvalue expression (e.g., a temporary object or the result of a function call that returns by value).

In summary, while both copy constructors and move constructors are used for object initialization, they serve different purposes: copy constructors create independent copies of existing objects, while move constructors transfer resources from temporary objects or objects that are about to be destroyed. This distinction is important for understanding resource management and optimizing performance, especially when dealing with large or dynamically allocated data.

In C++, **Integer&& obj** and **Integer& obj** are both reference types, but they have different meanings and implications:

1. **Integer&& obj (Rvalue Reference)**:
   * **Integer&& obj** denotes an rvalue reference, which is a reference to a temporary object or an object that is about to be moved from.
   * It is typically used to bind to temporary objects that are expiring or to identify objects that can be moved from efficiently.
   * Rvalue references are commonly used in move semantics, allowing for the creation of move constructors and move assignment operators to efficiently transfer resources (e.g., dynamically allocated memory) from one object to another.
   * Example usage: **Integer&& temp = createTempInteger();**
2. **Integer& obj (Lvalue Reference)**:
   * **Integer& obj** denotes an lvalue reference, which is a reference to an object that persists beyond the current expression and can be modified.
   * It is typically used to bind to non-temporary objects, allowing for the modification of their state.
   * Lvalue references are commonly used for passing objects to functions by reference, enabling the function to modify the original object.
   * Example usage: **Integer& ref = existingInteger;**

**Key Differences**:

* **Usage**:
  + Rvalue references (**&&**) are commonly used in move semantics and perfect forwarding, enabling efficient resource management and optimization of function calls.
  + Lvalue references (**&**) are commonly used for passing objects by reference and allowing functions to modify the original objects.
* **Binding**:
  + Rvalue references can bind to temporary objects (rvalues) and are often used to identify objects that can be moved from.
  + Lvalue references can bind to non-temporary objects (lvalues) and allow for the modification of their state.
* **Lifetime**:
  + Rvalue references can bind to temporary objects that may expire after the current expression, so they are suitable for short-term use or transferring resources from expiring objects.
  + Lvalue references bind to objects that persist beyond the current expression, allowing for continued access and modification.

In summary, the choice between **Integer&& obj** and **Integer& obj** depends on the intended use and the characteristics of the objects being referenced. Rvalue references are typically used for move semantics and temporary objects, while lvalue references are used for accessing and modifying non-temporary objects.