# lvalue and rvalue in C language

**L-value:** “l-value” refers to memory location which identifies an object. l-value may appear as either left hand or right hand side of an assignment operator(=). l-value often represents as identifier.

Expressions referring to modifiable locations are called “**modifiable l-values**“. A modifiable l-value cannot have an array type, an incomplete type, or a type with the **const** attribute. For structures and unions to be modifiable **lvalues**, they must not have any members with the **const** attribute. The name of the identifier denotes a storage location, while the value of the variable is the value stored at that location.

An identifier is a modifiable **lvalue** if it refers to a memory location and if its type is arithmetic, structure, union, or pointer. For example, if ptr is a pointer to a storage region, then **\*ptr** is a modifiable l-value that designates the storage region to which **ptr** points.

In C, the concept was renamed as **“locator value”**, and referred to expressions that locate (designate) objects. The l-value is one of the following:

1. The name of the variable of any type i.e, an identifier of integral, floating, pointer, structure, or union type.
2. A subscript ([ ]) expression that does not evaluate to an array.
3. A unary-indirection (\*) expression that does not refer to an array
4. An l-value expression in parentheses.
5. A **const** object (a nonmodifiable l-value).
6. The result of indirection through a pointer, provided that it isn’t a function pointer.
7. The result of member access through pointer(-> or .)

|  |
| --- |
| // declare a an object of type 'int'  int a;    // a is an expression referring to an  // 'int' object as l-value  a = 1;    int b = a; // Ok, as l-value can appear on right    // Switch the operand around '=' operator  9 = a;    // Compilation error:  // as assignment is trying to change the  // value of assignment operator |

**R-value**: r-value” refers to data value that is stored at some address in memory. A r-value is an expression that can’t have a value assigned to it which means r-value can appear on right but not on left hand side of an assignment operator(=).

|  |
| --- |
| // declare a, b an object of type 'int'  int a = 1, b;    a + 1 = b; // Error, left expression is             // is not variable(a + 1)    // declare pointer variable 'p', and 'q'  int \*p, \*q; // \*p, \*q are lvalue    \*p = 1; // valid l-value assignment    // below is invalid - "p + 2" is not an l-value  // p + 2 = 18;    q = p + 5; // valid - "p + 5" is an r-value    // Below is valid - dereferencing pointer  // expression gives an l-value  \*(p + 2) = 18;    p = &b;    int arr[20]; // arr[12] is an lvalue; equivalent                // to \*(arr+12)                // Note: arr itself is also an lvalue    struct S { int m; };    struct S obj; // obj and obj.m are lvalues    // ptr-> is an lvalue; equivalent to (\*ptr).m  // Note: ptr and \*ptr are also lvalues  struct S\* ptr = &obj; |

**Note**: The unary & (address-of) operator requires an lvalue as its operand. That is, &n is a valid expression only if n is an lvalue. Thus, an expression such as &12 is an error. Again, 12 does not refer to an object, so it’s not addressable. For instance,

|  |
| --- |
| // declare a as int variable and  // 'p' as pointer variable  int a, \*p;    p = &a; // ok, assignment of address          // at l-value    &a = p;    // error: &a is an r-value    int x, y;    (  x < y ? y : x) = 0; // It's valid because the ternary                    // expression preserves the "lvalue-ness"                   // of both its possible return values |

*Remembering the mnemonic, that****lvalues****can appear on the left of an assignment operator while****rvalues****can appear on the right*