12.2 — Value categories (lvalues and rvalues)

Before we talk about our first compound type (lvalue references), we're going to take a little detour and talk about what an `lvalue` is.

In lesson 1.10 -- Introduction to expressions, we defined an expression as "a combination of literals, variables, operators, and function calls that can be executed to produce a singular value".

For example:

```
#include <iostream>

int main()
{
    std::cout << 2 + 3 << '\n'; // The expression 2 + 3 produces the value 5

    return 0;
}
```

In the above program, the expression `2 + 3` is evaluated to produce the value 5, which is then printed to the console.

In lesson 6.4 -- Increment/decrement operators, and side effects, we also noted that expressions can produce side effects that outlive the expression:

```
#include <iostream>

int main()
{
    int x { 5 };
    ++x; // This expression statement has the side-effect of incrementing x
    std::cout << x << '\n'; // prints 6

    return 0;
}
```

In the above program, the expression `++x` increments the value of `x`, and that value remains changed even after the expression has finished evaluating.

Besides producing values and side effects, expressions can do one more thing: they can evaluate to objects or functions. We'll explore this point further in just a moment.

The properties of an expression

To help determine how expressions should evaluate and where they can be used, all expressions in C++ have two properties: a type and a value category.

The type of an expression

The type of an expression is equivalent to the type of the value, object, or function that results from the evaluated expression. For example:

```
int main()
{
    auto v1 { 12 / 4 }; // int / int => int
    auto v2 { 12.0 / 4 }; // double / int => double

    return 0;
}
```

For `v1`, the compiler will determine (at compile time) that a division with two `int` operands will produce an `int` result, so `int` is the type of this expression. Via type inference, `int` will then be used as the type of `v1`.

For `v2`, the compiler will determine (at compile time) that a division with a `double` operand and an `int` operand will produce a `double` result. Remember that arithmetic operators must have operands of matching types, so in this case, the `int` operand gets converted to a `double`, and a floating point division is performed. So `double` is the type of this expression.

The compiler can use the type of an expression to determine whether an expression is valid in a given context. For example:

```
#include <iostream>

void print(int x)
{
    std::cout << x << '\n';
}

int main()
{
    print("foo"); // error: print() was expecting an int argument, we tried to pass
in a string literal

    return 0;
}
```

In the above program, the `print(int)` function is expecting an `int` parameter. However, the type of the expression we're passing in (the string literal `"foo"`) does not match, and no conversion can be found. So a compile error results.

Note that the type of an expression must be determinable at compile time (otherwise type checking and type deduction wouldn't work) -- however, the value of an expression may be determined at either compile time (if the expression is constexpr) or runtime (if the expression is not constexpr).

The value category of an expression

Now consider the following program:

```
int main()
{
    int x{};

    x = 5; // valid: we can assign 5 to x
    5 = x; // error: can not assign value of x to literal value 5

    return 0;
}
```

One of these assignment statements is valid (assigning value `5` to variable `x`) and one is not (what would it mean to assign the value of `x` to the literal value `5`?). So how does the compiler know which expressions can legally appear on either side of an assignment statement?

The answer lies in the second property of expressions: the `value category`. The **value category** of an expression (or subexpression) indicates whether an expression resolves to a value, a function, or an object of some kind.

Prior to C++11, there were only two possible value categories: `lvalue` and `rvalue`.

In C++11, three additional value categories (`glvalue`, `prvalue`, and `xvalue`) were added to support a new feature called `move semantics`.

Author's note

In this lesson, we'll stick to the pre-C++11 view of value categories, as this makes for a gentler introduction to value categories (and is all that we need for the moment). We'll cover move semantics (and the additional three value categories) in a future chapter.

Lvalue and rvalue expressions

An **lvalue** (pronounced "ell-value", short for "left value" or "locator value", and sometimes written as "l-value") is an expression that evaluates to an identifiable object or function (or bit-field).

The term “identity” is used by the C++ standard, but is not well-defined. An entity (such as an object or function) that has an identity can be differentiated from other similar entities (typically by comparing the addresses of the entity).

Entities with identities can be accessed via an identifier, reference, or pointer, and typically have a lifetime longer than a single expression or statement.

```
int main()
{
    int x { 5 };
    int y { x }; // x is an lvalue expression

    return 0;
}
```

In the above program, the expression `x` is an lvalue expression as it evaluates to variable `x` (which has an identifier).

Since the introduction of constants into the language, lvalues come in two subtypes: a **modifiable lvalue** is an lvalue whose value can be modified. A **non-modifiable lvalue** is an lvalue whose value can’t be modified (because the lvalue is const or constexpr).

```
int main()
{
    int x{};
    const double d{};

    int y { x }; // x is a modifiable lvalue expression
    const double e { d }; // d is a non-modifiable lvalue expression

    return 0;
}
```

An **rvalue** (pronounced “arr-value”, short for “right value”, and sometimes written as `r-value`) is an expression that is not an lvalue. Rvalue expressions evaluate to a value. Commonly seen rvalues include literals (except C-style string literals, which are lvalues) and the return value of functions and operators that return by value. Rvalues aren’t identifiable (meaning they have to be used immediately), and only exist within the scope of the expression in which they are used.

```
int return5()
{
    return 5;
}

int main()
{
    int x{ 5 }; // 5 is an rvalue expression
    const double d{ 1.2 }; // 1.2 is an rvalue expression

    int y { x }; // x is a modifiable lvalue expression
    const double e { d }; // d is a non-modifiable lvalue expression
    int z { return5() }; // return5() is an rvalue expression (since the result is
returned by value)

    int w { x + 1 }; // x + 1 is an rvalue expression
    int q { static_cast<int>(d) }; // the result of static casting d to an int is an
rvalue expression

    return 0;
}
```

You may be wondering why `return5()`, `x + 1`, and `static_cast<int>(d)` are rvalues: the answer is because these expressions produce temporary values that are not identifiable objects.

Key insight

Lvalue expressions evaluate to an identifiable object.
Rvalue expressions evaluate to a value.

Now we can answer the question about why `x = 5` is valid but `5 = x` is not: an assignment operation requires the left operand of the assignment to be a modifiable lvalue expression, and the right operand to be an rvalue expression. The latter assignment (`5 = x`) fails because the left operand expression `5` isn't an lvalue.

```
int main()
{
    int x{};

    // Assignment requires the left operand to be a modifiable lvalue expression and
the right operand to be an rvalue expression
    x = 5; // valid: x is a modifiable lvalue expression and 5 is an rvalue
expression
    5 = x; // error: 5 is an rvalue expression and x is a modifiable lvalue
expression

    return 0;
}
```

Related content

A full list of lvalue and rvalue expressions can be found here. In C++11, rvalues are broken into two subtypes: prvalues and xvalues, so the rvalues we're talking about here are the sum of both of those categories.

Tip

If you're not sure whether an expression is an lvalue or rvalue, try taking its address using `operator&`, which requires its operand to be an lvalue. If `&(expression);` compiles, `expression` must be an lvalue:

```
int foo()
{
    return 5;
}

int main()
{
    int x { 5 };
    &x; // compiles: x is an lvalue expression
    &5; // doesn't compile: 5 is an rvalue expression
    &foo(); // doesn't compile: foo() is an rvalue expression
}
```

Key insight

A C-style string literal is an lvalue because C-style strings (which are C-style arrays) decay to a pointer. The decay process only works if the array is an lvalue (and thus has an address that can be stored in the pointer). C++ inherited this for backwards compatibility.

We cover array decay in lesson 17.8 -- C-style array decay.

Lvalue to rvalue conversion

Let's take a look at this example again:

```
int main()
{
    int x { 5 };
    int y { x }; // x is an lvalue expression

    return 0;
}
```

If `x` is an lvalue expression that evaluates to variable `x`, then how does `y` end up with value `5`?

The answer is that lvalue expressions will implicitly convert to rvalue expressions in contexts where an rvalue is expected but an lvalue is provided. The initializer for an `int` variable is expected to be an rvalue expression. Thus lvalue expression `x` undergoes an lvalue-to-rvalue conversion, which evaluates to value `5`, which is then used to initialize `y`.

We said above that the assignment operator expects the right operand to be an rvalue expression, so why does code like this work?

```
int main()
{
    int x{ 1 };
    int y{ 2 };

    x = y; // y is a modifiable lvalue, not an rvalue, but this is legal

    return 0;
}
```

In this case, `y` is an lvalue expression, but it undergoes an lvalue-to-rvalue conversion, which evaluates to the value of `y` (2), which is then assigned to `x`.

Now consider this example:

```
int main()
{
    int x { 2 };

    x = x + 1;

    return 0;
}
```

In this statement, the variable `x` is being used in two different contexts. On the left side of the assignment operator, `x` is an lvalue expression that evaluates to variable `x`. On the right side of the assignment operator, `x + 1` is an rvalue expression that evaluates to the value `3`.

Now that we've covered lvalues, we can get to our first compound type: the `lvalue reference`.

Key insight

A rule of thumb to identify lvalue and rvalue expressions:

- Lvalue expressions are those that evaluate to variables or other identifiable objects that persist beyond the end of the expression.
- Rvalue expressions are those that evaluate to literals or values returned by functions/operators that are discarded at the end of the expression.