The Move Book

Control flow statements are used to control the flow of execution in a program. They are used to make decisions, repeat a block of code, or exit a block of code early. Move includes the following control flow statements (explained in detail below):

The if expression is used to make decisions in a program. It evaluates a boolean expression and executes a block of code if the expression is true. Paired with else , it can execute a different block of code if the expression is false.

The syntax for an if expression is:

Just like any other expression, if requires a semicolon if there are other expressions following it. The else keyword is optional, except when the resulting value is assigned to a variable, as all branches must return a value to ensure type safety. Let's examine how an if expression works in Move with the following example:

Let's see how we can use if and else to assign a value to a variable:

In this example, the value of the if expression is assigned to the variable y . If x is greater than 0, y is assigned the value 1; otherwise, it is assigned 0. The else block is required because both branches of the if expression must return a value of the same type. Omitting the else block would result in a compiler error, as it ensures all possible branches are accounted for and type safety is maintained.

Conditional expressions are among the most important control flow statements in Move. They evaluate user-provided input or stored data to make decisions. One key use case is in the assert! macro , which checks if a condition is true and aborts execution if it is not. We'll explore this in detail shortly.

Loops are used to execute a block of code multiple times. Move has two built-in types of loops: loop and while . In many cases they can be used interchangeably, but usually while is used when the number of iterations is known in advance, and loop is used when the number of iterations is not known in advance or there are multiple exit points.

Loops are useful for working with collections, such as vectors, or for repeating a block of code until a specific condition is met. However, take care to avoid infinite loops, which can exhaust gas limits and cause the transaction to abort.

The while statement executes a block of code repeatedly as long as a boolean expression evaluates to true. Just like we've seen with if , the boolean expression is evaluated before each iteration of the loop. Additionally, like conditional statements, the while loop is an expression and requires a semicolon if there are other expressions following it.

The syntax for the while loop is:

Here is an example of a while loop with a very simple condition:

Now let's imagine a scenario where the boolean expression is always true . For example, if we literally passed true to the while condition. This is similar to how the loop statement functions, except that while evaluates a condition.

An infinite while loop, or a while loop with an always true condition, is equivalent to a loop . The syntax for creating a loop is straightforward:

Let's rewrite the previous example using loop instead of while :

Infinite loops are rarely practical in Move, as every operation consumes gas, and an infinite loop will inevitably lead to gas exhaustion. If you find yourself using a loop, consider whether there might be a better approach, as many use cases can be handled more efficiently with other control flow structures. That said, loop might be useful when combined with break and continue statements to create controlled and flexible looping behavior.

As we already mentioned, infinite loops are rather useless on their own. And that's where we introduce the break and continue statements. They are used to exit a loop early, and to skip the rest of the current iteration, respectively.

Syntax for the break statement is (without a semicolon):

The break statement is used to stop the execution of a loop and exit it early. It is often used in combination with a conditional statement to exit the loop when a certain condition is met. To illustrate this point, let's turn the infinite loop from the previous example into something that looks and behaves more like a while loop:

Almost identical to the while loop, right? The break statement is used to exit the loop when x is 5. If we remove the break statement, the loop will run forever, just like in the previous example.

The continue statement is used to skip the rest of the current iteration and start the next one. Similarly to break , it is used in combination with a conditional statement to skip the rest of an iteration when a certain condition is met.

Syntax for the continue statement is (without a semicolon):

The example below skips odd numbers and prints only even numbers from 0 to 10:

break and continue statements can be used in both while and loop loops.

The return statement is used to exit a function early and return a value. It is often used in combination with a conditional statement to exit the function when a certain condition is met. The syntax for the return statement is:

Here is an example of a function that returns a value when a certain condition is met:

Unlike in many other languages, the return statement is not required for the last expression in a function. The last expression in a function block is automatically returned. However, the return statement is useful when we want to exit a function early if a certain condition is met.

Conditional Statements

The if expression is used to make decisions in a program. It evaluates a boolean expression and executes a block of code if the expression is true. Paired with else , it can execute a different block of code if the expression is false.

The syntax for an if expression is:

```
bash if (<bool_expression>) <expression>; if (<bool_expression>) <expression> else <expression>;
```

Just like any other expression, if requires a semicolon if there are other expressions following it. The else keyword is optional, except when the resulting value is assigned to a variable, as all branches must return a value to ensure type safety. Let’s examine how an if expression works in Move with the following example:

```bash

# [test]

fun test_if() { let x = 5;

// `x > 0` is a boolean expression.
if (x > 0) {
    std::debug::print(&b"X is bigger than 0".to_string())
};

} ```
```

Let's see how we can use if and else to assign a value to a variable:

```bash

# [test]

fun test_if_else() { let x = 5; let y = if (x > 0) { 1 } else { 0 };

assert!(y == 1);

} ```
```

In this example, the value of the if expression is assigned to the variable y . If x is greater than 0, y is assigned the value 1; otherwise, it is assigned 0. The else block is required because both branches of the if expression must return a value of the same type. Omitting the else block would result in a compiler error, as it ensures all possible branches are accounted for and type safety is maintained.

Conditional expressions are among the most important control flow statements in Move. They evaluate user-provided input or stored data to make decisions. One key use case is in the assert! macro , which checks if a condition is true and aborts execution if it is not. We’ll explore this in detail shortly.

Loops are used to execute a block of code multiple times. Move has two built-in types of loops: loop and while . In many cases they can be used interchangeably, but usually while is used when the number of iterations is known in advance, and loop is used when the number of iterations is not known in advance or there are multiple exit points.

Loops are useful for working with collections, such as vectors, or for repeating a block of code until a specific condition is met. However, take care to avoid infinite loops, which can exhaust gas limits and cause the transaction to abort.

The while statement executes a block of code repeatedly as long as a boolean expression evaluates to true. Just like we've seen with if , the boolean expression is evaluated before each iteration of the loop. Additionally, like conditional statements, the while loop is an expression and requires a semicolon if there are other expressions following it.

The syntax for the while loop is:

```
bash while (<bool_expression>) { <expressions>; };
```

Here is an example of a while loop with a very simple condition:

```
``bash // This function iterates over thexvariable until it reaches 10, the // return value is the number of iterations it took to reach 10. // // Ifxis 0, then the function will return 10. // If
```

x` is 5, then the function will return 5. fun while_loop(mut x: u8): u8 { let mut y = 0;

```
// This will loop until `x` is 10.
// And will never run if `x` is 10 or more.
while (x < 10) {
    y = y + 1;
    x = x + 1;
};

y

}
```

[test]

fun test_while() { assert!(while_loop(0) == 10); // 10 times assert!(while_loop(5) == 5); // 5 times assert!(while_loop(10) == 0); // loop never executed } ```

Now let's imagine a scenario where the boolean expression is always true . For example, if we literally passed true to the while condition. This is similar to how the loop statement functions, except that while evaluates a condition.

```bash

# [test, expected_failure(out_of_gas, location=Self)]

fun test_infinite_while() { let mut x = 0;

```
// This will loop forever.
while (true) {
    x = x + 1;
};

// This line will never be executed.
assert!(x == 5);

} ```
```

An infinite while loop, or a while loop with an always true condition, is equivalent to a loop . The syntax for creating a loop is straightforward:

```
bash loop { <expressions>; };
```

Let's rewrite the previous example using loop instead of while :

```bash

# [test, expected_failure(out_of_gas, location=Self)]

fun test_infinite_loop() { let mut x = 0;

```
// This will loop forever.
loop {
    x = x + 1;
};

// This line will never be executed.
assert!(x == 5);
```

} ```

Infinite loops are rarely practical in Move, as every operation consumes gas, and an infinite loop will inevitably lead to gas exhaustion. If you find yourself using a loop, consider whether there might be a better approach, as many use cases can be handled more efficiently with other control flow structures. That said, loop might be useful when combined with break and continue statements to create controlled and flexible looping behavior.

As we already mentioned, infinite loops are rather useless on their own. And that's where we introduce the break and continue statements. They are used to exit a loop early, and to skip the rest of the current iteration, respectively.

Syntax for the break statement is (without a semicolon):

```
bash break
```

The break statement is used to stop the execution of a loop and exit it early. It is often used in combination with a conditional statement to exit the loop when a certain condition is met. To illustrate this point, let's turn the infinite loop from the previous example into something that looks and behaves more like a while loop:

```bash

# [test]

fun test_break_loop() { let mut x = 0;

```
// This will loop until `x` is 5.
loop {
    x = x + 1;

    // If `x` is 5, then exit the loop.
    if (x == 5) {
        break // Exit the loop.
    }
};

assert!(x == 5);
```

} ```

Almost identical to the while loop, right? The break statement is used to exit the loop when x is 5. If we remove the break statement, the loop will run forever, just like in the previous example.

The continue statement is used to skip the rest of the current iteration and start the next one. Similarly to break , it is used in combination with a conditional statement to skip the rest of an iteration when a certain condition is met.

Syntax for the continue statement is (without a semicolon):

```
bash continue
```

The example below skips odd numbers and prints only even numbers from 0 to 10:

```bash

# [test]

fun test_continue_loop() { let mut x = 0;

```
// This will loop until `x` is 10.
loop {
    x = x + 1;

    // If `x` is odd, then skip the rest of the iteration.
    if (x % 2 == 1) {
        continue // Skip the rest of the iteration.
    };

    std::debug::print(&x);

    // If `x` is 10, then exit the loop.
    if (x == 10) {
        break // Exit the loop.
    }
};

assert!(x == 10); // 10
```

} ```

break and continue statements can be used in both while and loop loops.

The return statement is used to exit a function early and return a value. It is often used in combination with a conditional statement to exit the function when a certain condition is met. The syntax for the return statement is:

```
bash return <expression>
```

Here is an example of a function that returns a value when a certain condition is met:

``bash /// This function returnstrueifxis greater than 0 and not 5, /// otherwise it returnsfalse`. fun is_positive(x: u8): bool { if (x == 5) { return false };

```
if (x > 0) {
    return true
};

false
```

}

[test]

fun test_return() { assert!(is_positive(5) == false); assert!(is_positive(0) == false); assert!(is_positive(1) == true); } ```

Unlike in many other languages, the return statement is not required for the last expression in a function. The last expression in a function block is automatically returned. However, the return statement is useful when we want to exit a function early if a certain condition is met.

Repeating Statements with Loops

Loops are used to execute a block of code multiple times. Move has two built-in types of loops: loop and while . In many cases they can be used interchangeably, but usually while is used when the number of iterations is known in advance, and loop is used when the number of iterations is not known in advance or there are multiple exit points.

Loops are useful for working with collections, such as vectors, or for repeating a block of code until a specific condition is met. However, take care to avoid infinite loops, which can exhaust gas limits and cause the transaction to abort.

The while statement executes a block of code repeatedly as long as a boolean expression evaluates to true. Just like we've seen with if , the boolean expression is evaluated before each iteration of the loop. Additionally, like conditional statements, the while loop is an expression and requires a semicolon if there are other expressions following it.

The syntax for the while loop is:

```
bash while (<bool_expression>) { <expressions>; };
```

Here is an example of a while loop with a very simple condition:

``bash // This function iterates over thexvariable until it reaches 10, the // return value is the number of iterations it took to reach 10. // // Ifxis 0, then the function will return 10. // If`x` is 5, then the function will return 5. fun while_loop(mut x: u8): u8 { let mut y = 0;

```
// This will loop until `x` is 10.
// And will never run if `x` is 10 or more.
while (x < 10) {
    y = y + 1;
    x = x + 1;
};

y

}
```

[test]

fun test_while() { assert!(while_loop(0) == 10); // 10 times assert!(while_loop(5) == 5); // 5 times assert!(while_loop(10) == 0); // loop never executed } ```

Now let's imagine a scenario where the boolean expression is always true . For example, if we literally passed true to the while condition. This is similar to how the loop statement functions, except that while evaluates a condition.

```bash

# [test, expected_failure(out_of_gas, location=Self)]

fun test_infinite_while() { let mut x = 0;

```
// This will loop forever.
while (true) {
    x = x + 1;
};

// This line will never be executed.
assert!(x == 5);

} ```
```

An infinite while loop, or a while loop with an always true condition, is equivalent to a loop . The syntax for creating a loop is straightforward:

```
bash loop { <expressions>; };
```

Let's rewrite the previous example using loop instead of while :

```bash

# [test, expected_failure(out_of_gas, location=Self)]

fun test_infinite_loop() { let mut x = 0;

```
// This will loop forever.
loop {
    x = x + 1;
};

// This line will never be executed.
assert!(x == 5);

} ```
```

Infinite loops are rarely practical in Move, as every operation consumes gas, and an infinite loop will inevitably lead to gas

exhaustion. If you find yourself using a loop, consider whether there might be a better approach, as many use cases can be handled more efficiently with other control flow structures. That said, loop might be useful when combined with break and continue statements to create controlled and flexible looping behavior.

As we already mentioned, infinite loops are rather useless on their own. And that's where we introduce the break and continue statements. They are used to exit a loop early, and to skip the rest of the current iteration, respectively.

Syntax for the break statement is (without a semicolon):

```
bash break
```

The break statement is used to stop the execution of a loop and exit it early. It is often used in combination with a conditional statement to exit the loop when a certain condition is met. To illustrate this point, let's turn the infinite loop from the previous example into something that looks and behaves more like a while loop:

```bash

# [test]

fun test_break_loop() { let mut x = 0;

// This will loop until `x` is 5.
loop {
    x = x + 1;

    // If `x` is 5, then exit the loop.
    if (x == 5) {
        break // Exit the loop.
    }
};

assert!(x == 5);

} ```
```

Almost identical to the while loop, right? The break statement is used to exit the loop when x is 5. If we remove the break statement, the loop will run forever, just like in the previous example.

The continue statement is used to skip the rest of the current iteration and start the next one. Similarly to break , it is used in combination with a conditional statement to skip the rest of an iteration when a certain condition is met.

Syntax for the continue statement is (without a semicolon):

```
bash continue
```

The example below skips odd numbers and prints only even numbers from 0 to 10:

```bash

# [test]

fun test_continue_loop() { let mut x = 0;

// This will loop until `x` is 10.
loop {
    x = x + 1;

    // If `x` is odd, then skip the rest of the iteration.
    if (x % 2 == 1) {
        continue // Skip the rest of the iteration.
    };

    std::debug::print(&x);

    // If `x` is 10, then exit the loop.
    if (x == 10) {
        break // Exit the loop.
```

```
    }
};

assert!(x == 10); // 10

} ```

break and continue statements can be used in both while and loop loops.

The return statement is used to exit a [function] early and return a value. It is often used in combination with a conditional statement to exit the function when a certain condition is met. The syntax for the return statement is:

```
bash return <expression>
```

Here is an example of a function that returns a value when a certain condition is met:

```
``bash /// This function returnstrueifxis greater than 0 and not 5, /// otherwise it returnsfalse`. fun is_positive(x: u8): bool { if (x == 5) { return false };

if (x > 0) {
    return true
};

false

}
```

[test]

fun test_return() { assert!(is_positive(5) == false); assert!(is_positive(0) == false); assert!(is_positive(1) == true); } ```

Unlike in many other languages, the return statement is not required for the last expression in a function. The last expression in a function block is automatically returned. However, the return statement is useful when we want to exit a function early if a certain condition is met.

The

The while statement executes a block of code repeatedly as long as a boolean expression evaluates to true. Just like we've seen with if , the boolean expression is evaluated before each iteration of the loop. Additionally, like conditional statements, the while loop is an expression and requires a semicolon if there are other expressions following it.

The syntax for the while loop is:

```
bash while (<bool_expression>) { <expressions>; };
```

Here is an example of a while loop with a very simple condition:

```
``bash // This function iterates over thexvariable until it reaches 10, the // return value is the number of iterations it took to reach 10. // // Ifxis 0, then the function will return 10. // Ifx` is 5, then the function will return 5. fun while_loop(mut x: u8): u8 { let mut y = 0;

// This will loop until `x` is 10.
// And will never run if `x` is 10 or more.
while (x < 10) {
    y = y + 1;
    x = x + 1;
};

y

}
```

[test]

fun test_while() { assert!(while_loop(0) == 10); // 10 times assert!(while_loop(5) == 5); // 5 times assert!(while_loop(10) == 0); //
```

loop never executed } ```

Now let's imagine a scenario where the boolean expression is always true . For example, if we literally passed true to the while condition. This is similar to how the loop statement functions, except that while evaluates a condition.

```bash

# [test, expected_failure(out_of_gas, location=Self)]

fun test_infinite_while() { let mut x = 0;

```
// This will loop forever.
while (true) {
    x = x + 1;
};

// This line will never be executed.
assert!(x == 5);
```

} ```

An infinite while loop, or a while loop with an always true condition, is equivalent to a loop . The syntax for creating a loop is straightforward:

```
bash loop { <expressions>; };
```

Let's rewrite the previous example using loop instead of while :

```bash

# [test, expected_failure(out_of_gas, location=Self)]

fun test_infinite_loop() { let mut x = 0;

```
// This will loop forever.
loop {
    x = x + 1;
};

// This line will never be executed.
assert!(x == 5);
```

} ```

Infinite loops are rarely practical in Move, as every operation consumes gas, and an infinite loop will inevitably lead to gas exhaustion. If you find yourself using a loop, consider whether there might be a better approach, as many use cases can be handled more efficiently with other control flow structures. That said, loop might be useful when combined with break and continue statements to create controlled and flexible looping behavior.

As we already mentioned, infinite loops are rather useless on their own. And that's where we introduce the break and continue statements. They are used to exit a loop early, and to skip the rest of the current iteration, respectively.

Syntax for the break statement is (without a semicolon):

```
bash break
```

The break statement is used to stop the execution of a loop and exit it early. It is often used in combination with a conditional statement to exit the loop when a certain condition is met. To illustrate this point, let's turn the infinite loop from the previous example into something that looks and behaves more like a while loop:

```bash

# [test]
```

fun test_break_loop() { let mut x = 0;

```
// This will loop until `x` is 5.
loop {
    x = x + 1;

    // If `x` is 5, then exit the loop.
    if (x == 5) {
        break // Exit the loop.
    }
};

assert!(x == 5);
```

} ```

Almost identical to the while loop, right? The break statement is used to exit the loop when x is 5. If we remove the break statement, the loop will run forever, just like in the previous example.

The continue statement is used to skip the rest of the current iteration and start the next one. Similarly to break , it is used in combination with a conditional statement to skip the rest of an iteration when a certain condition is met.

Syntax for the continue statement is (without a semicolon):

```
bash continue
```

The example below skips odd numbers and prints only even numbers from 0 to 10:

```bash

# [test]
```

fun test_continue_loop() { let mut x = 0;

```
// This will loop until `x` is 10.
loop {
    x = x + 1;

    // If `x` is odd, then skip the rest of the iteration.
    if (x % 2 == 1) {
        continue // Skip the rest of the iteration.
    };

    std::debug::print(&x);

    // If `x` is 10, then exit the loop.
    if (x == 10) {
        break // Exit the loop.
    }
};

assert!(x == 10); // 10
```

} ```

break and continue statements can be used in both while and loop loops.

The return statement is used to exit a function early and return a value. It is often used in combination with a conditional statement to exit the function when a certain condition is met. The syntax for the return statement is:

```
bash return <expression>
```

Here is an example of a function that returns a value when a certain condition is met:

``bash /// This function returnstrueifxis greater than 0 and not 5, /// otherwise it returnsfalse`. fun is_positive(x: u8): bool { if (x == 5) { return false };

```
if (x > 0) {
    return true
};
```

```
false

}
```

[test]

fun test_return() { assert!(is_positive(5) == false); assert!(is_positive(0) == false); assert!(is_positive(1) == true); } ```

Unlike in many other languages, the return statement is not required for the last expression in a function. The last expression in a function block is automatically returned. However, the return statement is useful when we want to exit a function early if a certain condition is met.

Infinite

Now let's imagine a scenario where the boolean expression is always true . For example, if we literally passed true to the while condition. This is similar to how the loop statement functions, except that while evaluates a condition.

```bash

# [test, expected_failure(out_of_gas, location=Self)]

fun test_infinite_while() { let mut x = 0;

```
// This will loop forever.
while (true) {
    x = x + 1;
};

// This line will never be executed.
assert!(x == 5);
```

} ```

An infinite while loop, or a while loop with an always true condition, is equivalent to a loop . The syntax for creating a loop is straightforward:

```
bash loop { <expressions>; };
```

Let's rewrite the previous example using loop instead of while :

```bash

# [test, expected_failure(out_of_gas, location=Self)]

fun test_infinite_loop() { let mut x = 0;

```
// This will loop forever.
loop {
    x = x + 1;
};

// This line will never be executed.
assert!(x == 5);
```

} ```

Infinite loops are rarely practical in Move, as every operation consumes gas, and an infinite loop will inevitably lead to gas exhaustion. If you find yourself using a loop, consider whether there might be a better approach, as many use cases can be handled more efficiently with other control flow structures. That said, loop might be useful when combined with break and continue statements to create controlled and flexible looping behavior.

As we already mentioned, infinite loops are rather useless on their own. And that's where we introduce the break and continue statements. They are used to exit a loop early, and to skip the rest of the current iteration, respectively.

Syntax for the break statement is (without a semicolon):

```
bash break
```

The break statement is used to stop the execution of a loop and exit it early. It is often used in combination with a conditional statement to exit the loop when a certain condition is met. To illustrate this point, let's turn the infinite loop from the previous example into something that looks and behaves more like a while loop:

```bash

# [test]

fun test_break_loop() { let mut x = 0;

// This will loop until `x` is 5.
loop {
    x = x + 1;

    // If `x` is 5, then exit the loop.
    if (x == 5) {
        break // Exit the loop.
    }
};

assert!(x == 5);

} ```
```

Almost identical to the while loop, right? The break statement is used to exit the loop when x is 5. If we remove the break statement, the loop will run forever, just like in the previous example.

The continue statement is used to skip the rest of the current iteration and start the next one. Similarly to break , it is used in combination with a conditional statement to skip the rest of an iteration when a certain condition is met.

Syntax for the continue statement is (without a semicolon):

```
bash continue
```

The example below skips odd numbers and prints only even numbers from 0 to 10:

```bash

# [test]

fun test_continue_loop() { let mut x = 0;

// This will loop until `x` is 10.
loop {
    x = x + 1;

    // If `x` is odd, then skip the rest of the iteration.
    if (x % 2 == 1) {
        continue // Skip the rest of the iteration.
    };

    std::debug::print(&x);

    // If `x` is 10, then exit the loop.
    if (x == 10) {
        break // Exit the loop.
    }
};

assert!(x == 10); // 10

} ```
```

break and continue statements can be used in both while and loop loops.

The return statement is used to exit a function early and return a value. It is often used in combination with a conditional statement to exit the function when a certain condition is met. The syntax for the return statement is:

```
bash return <expression>
```

Here is an example of a function that returns a value when a certain condition is met:

``bash /// This function returnstrueifxis greater than 0 and not 5, /// otherwise it returnsfalse`. fun is_positive(x: u8): bool { if (x == 5) { return false };

```
if (x > 0) {
    return true
};

false

}
```

[test]

fun test_return() { assert!(is_positive(5) == false); assert!(is_positive(0) == false); assert!(is_positive(1) == true); } ```

Unlike in many other languages, the return statement is not required for the last expression in a function. The last expression in a function block is automatically returned. However, the return statement is useful when we want to exit a function early if a certain condition is met.

Exiting a Loop Early

As we already mentioned, infinite loops are rather useless on their own. And that's where we introduce the break and continue statements. They are used to exit a loop early, and to skip the rest of the current iteration, respectively.

Syntax for the break statement is (without a semicolon):

```
bash break
```

The break statement is used to stop the execution of a loop and exit it early. It is often used in combination with a conditional statement to exit the loop when a certain condition is met. To illustrate this point, let's turn the infinite loop from the previous example into something that looks and behaves more like a while loop:

```bash

# [test]

fun test_break_loop() { let mut x = 0;

```
// This will loop until `x` is 5.
loop {
    x = x + 1;

    // If `x` is 5, then exit the loop.
    if (x == 5) {
        break // Exit the loop.
    }
};

assert!(x == 5);
```

} ```

Almost identical to the while loop, right? The break statement is used to exit the loop when x is 5. If we remove the break statement, the loop will run forever, just like in the previous example.

The continue statement is used to skip the rest of the current iteration and start the next one. Similarly to break , it is used in combination with a conditional statement to skip the rest of an iteration when a certain condition is met.
```

Syntax for the continue statement is (without a semicolon):

```
bash continue
```

The example below skips odd numbers and prints only even numbers from 0 to 10:

```bash
# [test]

fun test_continue_loop() { let mut x = 0;

// This will loop until `x` is 10.
loop {
    x = x + 1;

    // If `x` is odd, then skip the rest of the iteration.
    if (x % 2 == 1) {
        continue // Skip the rest of the iteration.
    };

    std::debug::print(&x);

    // If `x` is 10, then exit the loop.
    if (x == 10) {
        break // Exit the loop.
    }
};

assert!(x == 10); // 10

} ```

break and continue statements can be used in both while and loop loops.

The return statement is used to exit a function early and return a value. It is often used in combination with a conditional statement to exit the function when a certain condition is met. The syntax for the return statement is:

```
bash return <expression>
```

Here is an example of a function that returns a value when a certain condition is met:

``bash /// This function returnstrueifxis greater than 0 and not 5, /// otherwise it returnsfalse`. fun is_positive(x: u8): bool { if (x == 5) { return false };

if (x > 0) {
    return true
};

false

}

[test]

fun test_return() { assert!(is_positive(5) == false); assert!(is_positive(0) == false); assert!(is_positive(1) == true); } ```

Unlike in many other languages, the return statement is not required for the last expression in a function. The last expression in a function block is automatically returned. However, the return statement is useful when we want to exit a function early if a certain condition is met.

Skipping an Iteration

The continue statement is used to skip the rest of the current iteration and start the next one. Similarly to break , it is used in combination with a conditional statement to skip the rest of an iteration when a certain condition is met.
```

Syntax for the continue statement is (without a semicolon):

```
bash continue
```

The example below skips odd numbers and prints only even numbers from 0 to 10:

```bash
# [test]

fun test_continue_loop() { let mut x = 0;

// This will loop until `x` is 10.
loop {
    x = x + 1;

    // If `x` is odd, then skip the rest of the iteration.
    if (x % 2 == 1) {
        continue // Skip the rest of the iteration.
    };

    std::debug::print(&x);

    // If `x` is 10, then exit the loop.
    if (x == 10) {
        break // Exit the loop.
    }
};

assert!(x == 10); // 10

} ```

break and continue statements can be used in both while and loop loops.

The return statement is used to exit a function early and return a value. It is often used in combination with a conditional statement to exit the function when a certain condition is met. The syntax for the return statement is:

```
bash return <expression>
```

Here is an example of a function that returns a value when a certain condition is met:

``bash /// This function returnstrueifxis greater than 0 and not 5, /// otherwise it returnsfalse`. fun is_positive(x: u8): bool { if (x == 5) { return false };

if (x > 0) {
    return true
};

false

}

[test]

fun test_return() { assert!(is_positive(5) == false); assert!(is_positive(0) == false); assert!(is_positive(1) == true); } ```

Unlike in many other languages, the return statement is not required for the last expression in a function. The last expression in a function block is automatically returned. However, the return statement is useful when we want to exit a function early if a certain condition is met.

Early Return

The return statement is used to exit a function early and return a value. It is often used in combination with a conditional statement to exit the function when a certain condition is met. The syntax for the return statement is:

```
bash return <expression>
```
```

Here is an example of a function that returns a value when a certain condition is met:

``bash /// This function returnstrueifxis greater than 0 and not 5, /// otherwise it returnsfalse`. fun is_positive(x: u8): bool { if (x == 5) { return false };

```
if (x > 0) {
    return true
};

false
```

}

[test]

fun test_return() { assert!(is_positive(5) == false); assert!(is_positive(0) == false); assert!(is_positive(1) == true); } ```

Unlike in many other languages, the return statement is not required for the last expression in a function. The last expression in a function block is automatically returned. However, the return statement is useful when we want to exit a function early if a certain condition is met.