# Lecture 3 Notes

## Comparison operators

Comparison operators are used for comparing two values. The resulting value of a comparison expression is a boolean: either `true` or `false`.

| **Operator** | **Description** | **Usage** |
|---|---|---|
| `==` | Is equal to | `a == b` |
| `!=` | Is not equal to | `a != b` |
| `<` | Is less than | `a < b` |
| `>` | Is greather than | `a > b` |
| `<=` | Is less than or equal to | `a <= b` |
| `>=` | Is greater than or equal to | `a >= b` |

Comparison operators can be used with any of the primitive data types, including `int` and `double`.

You can also use the comparison operators with strings; the less than and greater than operators can compare strings by their alphabetical ordering.

## `if` statements

Comparison expressions are usually used within an `if` statement. An `if` statement selects the next statement(s) to execute based on a boolean expression. For example:

Listing 1: Example of an `if` statement

```
int amount;
cout << "Enter donation amount: "s;
cin >> amount;
if (amount < 5) cout << "Donation too small.\n"s;
```

The `if` statement can be used in this form:

`if (` *boolean* `)` *statement*

or this form:

`if (` *boolean* `)` *statement* `else` *statement*

In the first form, the `if` statement executes it's inner statement if it's inner boolean expression equals `true`.

The second form of the `if` statement is similar to the first form, with an additional statement to execute if the inner boolean expression happens to equal `false`.

The second statement for an `if` statement's `else` clause can be another `if` statement. Thus, you can perform a selection based on multiple boolean expressions:

Listing 2: Example of an `if` statement with multiple conditions

```
int age;
cout << "Enter your age: "s;
cin >> age;
```

```
if (age < 21) cout << "Too young.\n"s;
else if (age < 0) cout << "Not born yet.\n"s;
else cout << "You are old enough.\n"s;
```

`if` statements are not limited to executing a single statement, they can execute multiple statements if you provide a compound statement:

Listing 3: Example of an `if` statement and a compound statement

```
if (order_type == "combo"s) {
   cout << "Input size: "s;
   cin >> size;
   cout << "Input drink: "s;
   cin >> drink;
}
```

A compound statement is a sequence of statements treated as one statement. You can group statements together into a compound statement by surrounding them with { and }.

## Logical operators

You can use logical operators to combine multiple comparisons into one expression:

| **Operator** | **Usage** |
|---|---|
| `and` | `a and b` |
| `or` | `a or b` |
| `not` | `not a` |

Using these operators, you can form more complex boolean expressions:

```
bool should_hire = ((knows_cpp and knows_english) or
   can_clean_toilets) and age >= 18 and not a_felon;
```

The `and` operator returns `true` only if both of it's operands equal `true`. Here are all the possible operands and results for the `and` operator:

| `c = a and b` | | |
|---|---|---|
| **a** | **b** | **c** |
| `false` | `false` | `false` |
| `false` | `true` | `false` |
| `true` | `false` | `false` |
| `true` | `true` | `true` |

The `or` operator returns `false` only if both of it's operands equal `false`. Here are all hte possible operands and results for the `or` operator:

| `c = a or b` | | |
|---|---|---|
| **a** | **b** | **c** |
| `false` | `false` | `false` |
| `false` | `true` | `true` |
| `true` | `false` | `true` |
| `true` | `true` | `true` |

The `not` operator only requires one operand; it returns the inverse of it's boolean operand:

| b = not a | |
|---|---|
| **a** | **b** |
| `false` | `true` |
| `true` | `false` |

## Assignment operators

As you know, the simple assignment operator copies and stores the value from it's right operand into the variable in it's left operand. `a = b;` is an example of an assignment expression. Here are other types of assignment operators:

| **Operator** | **Usage** | **Simple assignment equivalent** |
|---|---|---|
| `+=` | `a += b` | `a = a + b` |
| `-=` | `a -= b` | `a = a - b` |
| `*=` | `a *= b` | `a = a * b` |
| `/=` | `a /= b` | `a = a / b` |
| `%=` | `a %= b` | `a = a % b` |

These are known as the arithmetic assignment operators; they all perform a calculation between a variable and a value, and then they store the result into the variable.

## Increment/Decrement operators

C++ supports short-hand ways to increment/decrement a variable:

| **Operator name** | **Usage** | **Description** |
|---|---|---|
| Prefix Increment | `++x` | Increment x and return x |
| Prefix Decrement | `--x` | Decrement x and return x |
| Postfix Increment | `x++` | Increment x and return the old value of x |
| Postfix Decrement | `x--` | Decrement x and return the old value of x |

You shouldn't worry too much about the postfix increment/decrement operators, they will become more useful to you as you get more experienced. Just remember that `++x` is the same as `x += 1` which is also the same as `x = x + 1`. `--x` is the same as `x -= 1` which is also the same as `x = x - 1`.

## `while` statements

A `while` statement is used for statement iteration; it will repeatedly execute a statement while a boolean expression returns `true`. The syntax of the `while` statement is:

`while (` *boolean* `)` *statement*

The boolean expression is evaluated first; if it's value is `true`, then the statement or compound statement is executed. This process repeats while the boolean expression evaluates to `true`.

Listing 4: Example of a while statement

```
int n = 0;
string s;
while (n < 100) {
   s += "A"s;
   ++n;
}
```

The above `while` statement used a compound statement, you may also use a single statement within a `while` statement:

Listing 5: Example of a while statement without using a compound statement

```
while (x < y) x *= 2;
```

## `do while` statements

A `do while` statement is similar to the `while` statement. The syntax is:

`do` *statement* `while (` *boolean* `);`

The inner statement is executed first, then the boolean expression is evaluated. The process repeats while the boolean expression evaluates to `true`.

Listing 6: Example of a `do while` statement

```
do {
   cout << "Are you guilty? "s;
   cin >> answer;
} while (answer == "no"s);
```

## `for` statements

Another variation of the `while` statement is the `for` statement:

`for (` *expr1* `;` *expr2* `;` *expr3* `)` *statement*

According to the above syntax, a `for` statement will do the following:

1. Execute *expr1*
2. Evaluate *expr2*
3. If *expr2* is `true`, then *statement* gets executed, otherwise the `for` statement aborts.
4. *expr3* gets executed.
5. Go back to step 2

Listing 7: Example of a `for` statement that counts from 0 to 9

```
int i;
for (i = 0; i < 10; ++i) {
   cout << i << endl;
}
```

This code outputs:

```
0
1
2
3
4
5
6
7
8
9
```

`endl` is a stream manipulator that writes a new line to the output stream. `cout << endl` has the same effect as `cout << '\n'` and `cout << "\n"s`.

Here is another example of using a `for` statement:

Listing 8: Iterating backwards

```
int i;
for (i = 9; i >= 0; --i) {
   cout << i << endl;
}
```

The code above outputs:

```
9
8
7
6
5
4
3
2
1
0
```

## `break` statement

`break` unconditionally breaks out of an iteration. It can be used inside a `for` statement, `while` statement, or `do while` statement.

```
int i;
for (i = 0; i < 10; ++i) {
   cout << i << endl;
   if (i == 5) break;
}
```

This code outputs:

```
0
1
2
3
4
5
```

## `continue` statement

`continue` jumps to the next iteration, skipping the remaining statements before the end of the iteration.

```
int i;
for (i = 0; i < 10; ++i) {
   if (i % 2 != 0) continue;
   cout << i << endl;
}
```

This code outputs:

```
0
2
4
6
8
```

– *Mark Swoope*