Object Oriented Programming using C++

Operators and Expressions

C++ provides a rich set of built-in operators, which can be categorized into four main classes:

1. Arithmetic Operators: Used for performing basic arithmetic operations.
   - Examples: `+`, `-`, `*`, `/`, `%`
2. Relational Operators: Used for comparing values.
   - Examples: `==`, `!=`, `<`, `>`, `<=`, `>=`
3. Logical Operators: Used for performing logical operations.
   - Examples: `&&` (logical AND), `||` (logical OR), `!` (logical NOT)
4. Bitwise Operators: Used for performing operations on bits and bit patterns.
   - Examples: `&` (bitwise AND), `|` (bitwise OR), `^` (bitwise XOR), `~` (bitwise NOT), `<<` (left shift), `>>` (right shift)

The Assignment Operator

In C++, the assignment operator (`=`) is used to assign a value to a variable. The general form of the assignment operator is:

target = expression;

- Target (Left-hand side): This must be a variable or a pointer. It cannot be a function or a constant.

- Expression (Right-hand side): This can be as simple as a single constant or as complex as a full expression.

For example:

```
int a;       // 'a' is an lvalue
a = 10;      // 10 is an rvalue
```

In this case, `a` is an lvalue because it represents a variable that can be assigned a value, while `10` is an rvalue because it represents a value that is assigned to `a`.

Type Conversion in Assignments

When variables of different types are mixed in an expression, type conversion takes place to ensure that the operation is valid. This process is known as type conversion or type casting.

Here's how it works:

Implicit Conversion (Automatic Type Conversion):

The compiler automatically converts the value of the right-hand side (expression side) to match the type of the left-hand side (target variable). This ensures that the types are compatible for the assignment or operation.

For example:

```
int a = 5;
double b = 10.5;
a = b;  // Implicit conversion: double b is converted to int and assigned to a.
```

Here, `b` is a `double`, but it is automatically converted to `int` before being assigned to `a`.

Explicit Conversion (Type Casting):

You can also manually convert between types using type casting. This gives you control over the conversion process.

For example:

```
double b = 10.5;
int a = static_cast<int>(b);  // Explicit conversion: double b is explicitly cast to int.
```

In both cases, the value of the right side (expression side) is converted to the type of the left side (target variable) to ensure that the operation or assignment is carried out correctly.

Multiple Assignments

In C++, you can assign the same value to multiple variables in a single statement using chained assignments. Here's how it works:

```
int x, y, z;
x = y = z = 0;
```

In this example, the assignment is evaluated from right to left:

1. `z = 0`: First, the value `0` is assigned to `z`.
2. `y = z`: Next, `y` is assigned the value of `z`, which is `0`.
3. `x = y`: Finally, `x` is assigned the value of `y`, which is also `0`.

As a result, all three variables (`x`, `y`, and `z`) end up with the value `0`.

Arithmetic Operators

```
#include <iostream>
int main() {
    int x = 5, y = 2;
    std::cout << x / y << std::endl;  // Performs integer division, displays 2
    std::cout << x % y << std::endl;  // Calculates the remainder of the division, displays 1
    y = 2;
    std::cout << x / y << " " << x % y << std::endl;  // Displays "2 1"
    return 0;
}
```

In this example:

- `x / y` performs integer division, which divides `x` by `y` and returns the quotient (in this case, `5 / 2` results in `2`).

- `x % y` computes the remainder of the division (in this case, `5 % 2` results in `1`).

So:

- `std::cout << x / y` prints `2`

- `std::cout << x % y` prints `1`

When `y` is assigned the value `2`, the output of `x / y` and `x % y` remains `2` and `1`, respectively.

Diff b/w prefix and postfix forms

In C++, the increment (`++`) and decrement (`--`) operators can be used in both prefix and postfix forms. The difference between these two forms affects when the operation is performed relative to when the value is used in an expression. Here's how they work:

Prefix Form (`++x` or `--x`)

- Operation: The increment or decrement is performed first, before the value is used in the expression.

- Example:

```
int x = 10;
int y = ++x;  // x is incremented to 11 first, then y is assigned the value 11
```

In this case, `x` becomes 11 before `y` is assigned, so `y` gets the value 11.

Postfix Form (`x++` or `x--`)

- Operation: The current value of the variable is used in the expression first, and then the increment or decrement is performed.

- Example:

```
int x = 10;
int y = x++;  // y is assigned the value 10 first, then x is incremented to 11
```

Here, `y` gets the value 10 because the increment happens after `y` is assigned. `x` becomes 11 after the assignment.

Summary

- Prefix Example:

```
int x = 10;
int y = ++x;  // x becomes 11, y is assigned 11
```

- Postfix Example:

```
int x = 10;
int y = x++;  // y is assigned 10, x becomes 11
```

In both cases, `x` will end up as 11. The difference is in the value assigned to `y` and when the increment or decrement actually occurs.

Note on Parentheses

Parentheses can be used to alter the precedence of operations, forcing certain operations to be evaluated before others. For example:

```
int x = 5;

int y = (x++ + 3);  // x++ is evaluated first, then 3 is added to the original value of x
```

Here, `x++` evaluates to 5, and then 3 is added to give `y` a value of 8. After this, `x` becomes 6.

Relational and Logical operators

Relational Operators:

These operators compare values to determine their relationships, such as equality, inequality, and order.

Examples include:

- `==` (equal to)
- `!=` (not equal to)
- `>` (greater than)
- `<` (less than)
- `>=` (greater than or equal to)
- `<=` (less than or equal to)

Logical Operators:

These operators connect or negate relational expressions to create more complex conditions.

Examples include:

- `&&` (logical AND)
- `||` (logical OR)
- `!` (logical NOT)

Expression Results: Expressions with relational and logical operators return `0` for false and `1` for true.

Operator	Meaning	Expression	Value
>	is greater than	6 > 4	true
>=	is greater than or equal to	7>=6	true
<	is less than	9 < 8	false
<=	is less than or equal to	5 <= 5	true
==	is equal to	3 == -3	false
!=	is not equal to	8.0 != 8	fasle

Precedence

Operator	Associativity	Type
+, −	right to left	unary arithmetic
*, /, %	left to right	binary arithmetic
+, −	left to right	binary arithmetic
<, <=, >, >=	left to right	binary relational
==, !=	left to right	binary relational

Bitwise Operator

Bitwise operations involve manipulating the individual bits of integer data types. These operations include testing, setting, and shifting bits in bytes or words. They are applied directly to the binary representations of the operands.

The bit shift operators, `>>` (right shift) and `<<` (left shift), move all bits in a value to the right or left, respectively, by the specified number of positions. The general form for the right shift is:

value >> number_of_bits

The general form for the left shift is:

value << number_of_bits

When bits are shifted out of one end, zeros are introduced at the other end. Note that shifting is not a rotation; bits that are shifted out do not wrap around to the other end. Instead, they are lost.

The ternary operator

The ternary operator `?` has the following general form:

Exp1 ? Exp2 : Exp3;

Here's how it works:

- `Exp1` is evaluated first.
- If `Exp1` is true, `Exp2` is evaluated and becomes the value of the entire expression.
- If `Exp1` is false, `Exp3` is evaluated and becomes the value of the entire expression.

For example:

```
int x = 10;
int y = (x > 5) ? 100 : 200;
```

In this case, since `x > 5` is true, `y` will be assigned the value 100.

*The & and * pointer operators*

A pointer is essentially the memory address of an object.

A pointer variable is specifically declared to store the address of an object of a particular type.

The `&` operator, which is a unary operator requiring only one operand, returns the memory address of its operand.

For example, in the statement:

m = &count;

`m` is assigned the memory address of the variable `count`. This address represents the internal location of `count` in memory and is distinct from the value stored in `count`. In this context, `&` signifies "the address of," so the statement can be understood as "m receives the address of count."

In the above eg assume that the variable count is at memory location 2000. Also assume that count has the value 100, m will have the value 2000.

The second pointer operator is `*`, which complements the `&` operator.

The `*` operator, also a unary operator, retrieves the value of the variable located at the address it precedes.

For example, if `m` contains the memory address of the variable `count`, the following statement:

q = *m;

will place the value stored at the address `m` into `q`. If `count` holds the value `100` and its address is stored in `m`, then `q` will end up with the value `100`. Here, `*` signifies "at address," so the statement can be understood as "q receives the value at the address stored in m."

Here's a corrected version of the program that uses the `*` and `&` operators to assign the value `10` to a variable called `target`:

```
#include <stdio.h>
int main(void) {
   int target, source;
   int *m;
   source = 10;    // Assign 10 to source
   m = &source;    // m now holds the address of source
   target = *m;    // target is assigned the value at the address m (which is 10)
   printf("%d", target); // Print the value of target
   return 0;
}
```

The dot(.) and arrow(->) operator

In both C and C++, the `.` (dot) and `->` (arrow) operators are used to access elements of structures, unions, and classes.

- Class members:
  - Dot Operator (`.`): Used to access a member of a class when you have a direct instance of it.
    - Example: `employee.wage = 123.34;`
  - Arrow Operator (`->`): Used to access a member of a class through a pointer.
    - Example: `employee_ptr->wage = 123.34;`

In summary:

- Use the dot operator (`.`) when you are working with a direct object or instance.
- Use the arrow operator (`->`) when you are working with a pointer to an object or instance.

Expressions

In programming, expressions are formed from operators, constants, and variables. An expression is any valid combination of these elements that yields a result.

- Operators perform operations on variables and constants.
- Constants are fixed values that do not change.
- Variables are storage locations identified by names that hold data which can be modified during program execution.

For example, in the expression `x = 5 + 3`, `+` is an operator, `5` and `3` are constants, and `x` is a variable.

Type conversion in expressions

When constants and variables of different types are used together in an expression, they are converted to a common type to ensure consistent operations. This process involves type promotion.

- Type Promotion: The compiler converts all operands to the type of the largest operand to ensure that operations are performed accurately.

- Integral Promotion: Specifically, all `char` and `short` integer values are automatically promoted to `int` before the expression is evaluated.

This ensures that operations are performed with the most appropriate precision and that the results are consistent across different types.

Casts

You can enforce a specific data type for an expression using a cast. The general form of a cast is:

(type) expression

Here, `type` represents the data type you want to cast to.

For example, to ensure that the division `x / 2` is treated as a floating-point division and not integer division, you can use:

(float) x / 2

In this case, casting `x` to `float` ensures that the division operation yields a floating-point result, preserving any fractional part. Casts are unary operators and have the same precedence as other unary operators.

Compound assignments

A variation of the assignment statement, known as compound assignment, simplifies certain types of assignment operations. For example:

Instead of writing:

x = x + 10;

You can use the compound assignment operator:

x += 10;

In this case, `+=` tells the compiler to add `10` to the current value of `x` and then assign the result back to `x`.

Compound assignment operators exist for all binary operators. Generally, a statement like:

var = var operator expression;

can be rewritten using compound assignment as:

var operator= expression;

Here are some examples:

- `x -= 5` is equivalent to `x = x - 5;`
- `y *= 3` is equivalent to `y = y * 3;`
- `z /= 4` is equivalent to `z = z / 4;`
- `w %= 2` is equivalent to `w = w % 2;`