Object Oriented Programming using C++

STATEMENTS

What is a statement ?

A statement is a unit of a program that performs an action. It represents a command or instruction that can be executed to achieve a specific result.

Selection statements

- IF STATEMENT
- ITERATION STATEMENTS

If statement

General form of an If statement:

```
if (expression) {
   // Statements to execute if the expression is true
} else {
   // Statements to execute if the expression is false
}
```

If the expression evaluates to true, the statement or block associated with the `if` is executed. Otherwise, the statement or block associated with the `else` is executed.

Example :

```
#include <iostream>
using namespace std;
int main() {
   int a = 10, b = 15;
   if (a > b) {
      cout << "A is greater";
   } else {
      cout << "B is greater";
   }
   return 0;
}
```

Nested if

Here's a general form for a nested `if` statement:

```
if (expression1) {
    if (expression2) {
        // Statement executed if expression1 and expression2 are true
    } else {
        // Statement executed if expression1 is true and expression2 is false
    }
} else {
    // Statement executed if expression1 is false
}
```

In this structure:

- If `expression1` is true, the inner `if` is evaluated.

  - If `expression2` is also true, the corresponding statement inside the inner `if` block is executed.

  - If `expression2` is false, the statement inside the `else` block of the inner `if` is executed.

- If `expression1` is false, the statement in the `else` block of the outer `if` is executed.

Example :

```
#include <iostream>
using namespace std;
int main() {
   int a = 10, b = 15, c = 20;
   if (a > b) {
      if (a > c) {
         cout << "A is largest";
      } else {
         cout << "C is largest";
      }
   } else {
      if (b > c) {
         cout << "B is largest";
      } else {
         cout << "C is largest";
      }
   }
   return 0;
}
```

Else if ladder

In an if-else ladder, conditions are evaluated sequentially from top to bottom. As soon as a true condition is encountered, the corresponding statement is executed, and the remaining conditions are bypassed.

```
if (expression1) {
   statement1;
} else if (expression2) {
   statement2;
} else if (expression3) {
   statement3;
} else {
   statement;
}
```

Example:

```
#include <iostream>
using namespace std;
int main() {
   int score = 85;
   if (score >= 90) {
      cout << "Grade: A";
   } else if (score >= 80) {
      cout << "Grade: B";
   } else if (score >= 70) {
      cout << "Grade: C";
   } else {
      cout << "Grade: D";
   }
   return 0;
}
```

? Operator(Ternary)

The ternary `?` operator in C++ provides a concise way to replace simple `if-else` statements. It evaluates a condition and returns one of two values based on the result.

condition ? statement1 : statement2;

Example:

```
#include <iostream>
using namespace std;
int main() {
   int x = 10;
   // Using ternary operator to determine if x is positive or not
   string result = (x > 0) ? "x is positive" : "x is not positive";
   cout << result << endl;
   return 0;
}
```

Switch statement

The `switch` statement in C++ is used for multi-way branching based on the value of an expression. It simplifies the process of selecting among many possible execution paths.

```
switch(expression) {
   case constant1:
      statement1;
      break;
   case constant2:
      statement2;
      break;
   // Add more cases as needed
   default:
      defaultStatement;
}
```

Example:

```
#include <iostream>
using namespace std;
int main() {
  int day = 3;
  switch(day) {
    case 1:
      cout << "Monday" << endl;
      break;
    case 2:
      cout << "Tuesday" << endl;
      break;
    case 3:
      cout << "Wednesday" << endl;
      break;
    case 4:
      cout << "Thursday" << endl;
      break;
    case 5:
      cout << "Friday" << endl;
      break;
    case 6:
      cout << "Saturday" << endl;
      break;
    case 7:
      cout << "Sunday" << endl;
      break;
    default:
      cout << "Invalid day" << endl;
  }
  return 0;
}
```

Iteration statements

In C++, iteration statements (also known as loops) allow a set of instructions to be executed repeatedly based on a condition. They are useful for performing repetitive tasks efficiently. There are three primary types of iteration statements in C++:

For loop

The `for` loop in C++ is used to execute a block of code repeatedly based on a specified condition. It is especially useful when you know in advance how many times you want the loop to run.

for(initialization; condition; increment/decrement) {

- Initialization: This is an assignment statement that sets the initial value of the loop control variable.

- Condition: A relational expression that determines if the loop should continue or exit. The loop runs as long as this condition is true.

- Increment/Decrement: Defines how the loop control variable changes after each iteration of the loop.

Infinite Loop:

You can create an infinite loop using the `for` loop with no condition:

```
for(;;) {
   // Statements
}
```

Example:

Here’s a simple example of a `for` loop that prints numbers from 0 to 9:

```
#include <iostream>
using namespace std;
int main() {
    for(int i = 0; i < 10; i++) {
        cout << i << "\t";
    }
    return 0;
}
```

Output:

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

While loop

The `while` loop in C++ allows you to execute a block of code repeatedly based on a condition. It is used when the number of iterations is not known beforehand and depends on some condition evaluated at runtime.

```
while(condition) {
   // Statement(s)
}
```

- Condition: An expression that is evaluated before each iteration. If the condition is true, the loop continues; if false, the loop terminates.

Example:

Here’s an example of a `while` loop that prints numbers from 0 to 9:

```
#include <iostream>
using namespace std;
int main() {
    int i = 0;
    while (i < 10) {
        cout << i << "\t";
        i++;
    }
    return 0;
}
```

Output:

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

Do while

The `do-while` loop is an exit-controlled loop in C++, meaning that the condition is tested after the loop body is executed. This guarantees that the loop body will be executed at least once.

```
do {
   // Statement(s)
} while (condition);
```

- Statements: The block of code that will be executed.

- Condition: An expression evaluated after the loop body is executed. If the condition is true, the loop continues; if false, the loop terminates.

Example:

Here’s an example of a `do-while` loop that prints numbers from 25 to 20:

```
#include <iostream>
using namespace std;
int main() {
   int a = 25;
   do {
      cout << a << "\t";
      a--;
   } while (a > 20); // Continue the loop while 'a' is greater than 20
   return 0;
}
```

Output:

```
25   24   23   22   21   20
```

Jump Statements

C++ provides several statements for performing unconditional branching within a program. These statements allow you to alter the flow of control in various ways.

return

- Purpose: Exits from the current function and optionally returns a value.
- Usage: Can be used anywhere within a function.
- Example:

```
int add(int a, int b) {
    return a + b; // Exits the function and returns the result
}
```

goto

- Purpose: Jumps to a labeled statement within the same function.
- Usage: Can be used anywhere in the function but should be used cautiously as it can make code harder to follow.
- Example:

```
#include <iostream>
using namespace std;
int main() {
   int a = 5;
   if (a == 5) goto label;
   cout << 'This will be skipped' << endl;
label:
   cout << 'Jumped to label' << endl;
   return 0;
}
```

break

- Purpose: Exits from the nearest enclosing loop or switch statement.
- Usage: Commonly used within loops and switch statements to terminate execution prematurely.
- Example:

```
#include <iostream>
using namespace std;
int main() {
    for (int i = 0; i < 10; i++) {
        if (i == 5) break; // Exits the loop when i is 5
        cout << i << ' ';
    }
    return 0;
}
```

continue

- Purpose: Skips the remaining statements in the current iteration of a loop and proceeds to the next iteration.
- Usage: Used within loops to bypass certain parts of the loop body based on a condition.
- Example:

```
#include <iostream>
 using namespace std;
 int main() {
    for (int i = 0; i < 10; i++) {
        if (i % 2 == 0) continue; // Skips even numbers
        cout << i << ' ';
    }
    return 0;
 }
```

The exit() function

Breaking Out of a Program

- Function: `exit()`

- Purpose: Causes immediate termination of the entire program, returning control to the operating system.

- General Form: `void exit(int return_code);`

  - The `return_code` value is returned to the calling process, typically the operating system.

- Usage: Can be used to exit from anywhere in the program.