One-Week-C-Primer (Day 2)

Day 2: Operators, Control Flow, and Loops

Note : Refer to the Attached Document along with this for the codes

Operators and Types

An operator is a symbol that operates on a value or a variable
Types:

Arithmetic Operators:

Arithmetic operators perform basic mathematical operations in C.

- Addition (`+`): Adds two operands.
- Subtraction (`-`): Subtracts the second operand from the first.
- Multiplication (`*`): Multiplies two operands.
- Division (`/`): Divides the first operand by the second.
- Modulus (`%`): Returns the remainder of the division of the first operand by the second.

```
#include <stdio.h>

int main() {
    int num1 = 10, num2 = 5;

    int addition = num1 + num2;          // 10 + 5 = 15
    int subtraction = num1 - num2;       // 10 - 5 = 5
    int multiplication = num1 * num2;    // 10 * 5 = 50
    int division = num1 / num2;          // 10 / 5 = 2
    int modulus = num1 % num2;           // 10 % 5 = 0

    printf("Addition: %d\n", addition);
    printf("Subtraction: %d\n", subtraction);
    printf("Multiplication: %d\n", multiplication);
    printf("Division: %d\n", division);
    printf("Modulus: %d\n", modulus);

    return 0;
}
```

Relational Operators:

Relational operators compare two values or expressions and return a Boolean value (1 for true and 0 for false).

- Equal to (`==`): Checks if two operands are equal.
- Not equal to (`!=`): Checks if two operands are not equal.
- Greater than (`>`): Checks if the first operand is greater than the second.
- Less than (`<`): Checks if the first operand is less than the second.
- Greater than or equal to (`>=`): Checks if the first operand is greater than or equal to the second.
- Less than or equal to (`<=`): Checks if the first operand is less than or equal to the second.

```
#include <stdio.h>

int main() {
    int num1 = 10, num2 = 5;

    printf("%d is equal to %d: %d\n", num1, num2, num1 == num2);     // 10 is equal to 5:
 0 (False)
    printf("%d is not equal to %d: %d\n", num1, num2, num1 != num2); // 10 is not equal to
5: 1 (True)
    printf("%d is greater than %d: %d\n", num1, num2, num1 > num2);  // 10 is greater than
5: 1 (True)
    printf("%d is less than %d: %d\n", num1, num2, num1 < num2);     // 10 is less than 5:
0 (False)
    printf("%d is greater than or equal to %d: %d\n", num1, num2, num1 >= num2); // 10 is
 greater than or equal to 5: 1 (True)
    printf("%d is less than or equal to %d: %d\n", num1, num2, num1 <= num2);    // 10 is
 less than or equal to 5: 0 (False)

    return 0;
}
```

Logical Operators:

Logical operators perform logical operations between two expressions and return a Boolean value.

- Logical AND (`&&`): Returns true if both operands are true.

- Logical OR (`||`): Returns true if at least one of the operands is true.
- Logical NOT (`!`): Returns the opposite of the operand's value.

```
#include <stdio.h>

int main() {
    int num1 = 10, num2 = 5;

    printf("(%d > 5) && (%d < 15): %d\n", num1, num1, (num1 > 5) && (num1 < 15)); // (10 >
5) && (10 < 15): 1 (True)
    printf("(%d > 5) || (%d > 15): %d\n", num2, num2, (num2 > 5) || (num2 > 15)); // (5 >
 5) || (5 > 15): 1 (True)
    printf("!(%d > %d): %d\\n", num1, num2, !(num1 > num2));                      // !(10
> 5): 0 (False)

    return 0;
}
```

Assignment Operators:

Assignment operators are used to assign values to variables.

- Assignment (`=`): Assigns the value on the right to the variable on the left.
- Addition assignment (`+=`): Adds the value on the right to the variable on the left and assigns the result to the left variable.
- Subtraction assignment (`=`): Subtracts the value on the right from the variable on the left and assigns the result to the left variable.
- Multiplication assignment (`=`): Multiplies the value on the right with the variable on the left and assigns the result to the left variable.
- Division assignment (`/=`): Divides the variable on the left by the value on the right and assigns the result to the left variable.
- Modulus assignment (`%=`): Calculates the remainder of the division of the variable on the left by the value on the right and assigns the result to the left variable.

```
#include <stdio.h>

int main() {
    int num1 = 10, num2 = 5;

    // Assignment
```

```
    num1 = num2;           // num1 = 5
    printf("After assignment, num1 = %d\n", num1);

    // Addition assignment
    num1 += num2;          // num1 = num1 + num2 => 5 + 5 => 10
    printf("After addition assignment, num1 = %d\n", num1);

    // Subtraction assignment
    num1 -= num2;          // num1 = num1 - num2 => 10 - 5 => 5
    printf("After subtraction assignment, num1 = %d\n", num1);

    // Multiplication assignment
    num1 *= num2;          // num1 = num1 * num2 => 5 * 5 => 25
    printf("After multiplication assignment, num1 = %d\n", num1);

    // Division assignment
    num1 /= num2;          // num1 = num1 / num2 => 25 / 5 => 5
    printf("After division assignment, num1 = %d\n", num1);

    // Modulus assignment
    num1 %= num2;          // num1 = num1

 % num2 => 5 % 5 => 0
    printf("After modulus assignment, num1 = %d\n", num1);

    return 0;
}
```

Decision-making Statements

Conditional statements are essential constructs in programming that allow the execution of different blocks of code based on specific conditions. In C, conditional statements are implemented using the `if`, `else if`, and `else` statements. These statements help control the flow of a program and make decisions based on the evaluation of expressions. Conditional statements provide the ability to create branching logic, enabling programs to behave differently in various situations.

1. if Statement:
   The `if` statement is the fundamental conditional statement in C. It evaluates an expression inside parentheses and executes the code block (enclosed in curly braces) if the expression evaluates to true (non-zero).

Syntax:

```
if (expression) {
    // Code block to execute if the expression is true
}
```

Example:

```
#include <stdio.h>

int main() {
    int num = 10;

    if (num > 5) {
        printf("The number is greater than 5\n");
    }

    return 0;
}
```

1. if-else Statement:
   The `if-else` statement allows executing different code blocks based on whether the condition is true or false. If the expression inside the `if` evaluates to true, the code inside the `if` block is executed. Otherwise, the code inside the `else` block is executed.

Syntax:

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

Example:

```
#include <stdio.h>

int main() {
    int num = 10;

    if (num > 5) {
        printf("The number is greater than 5.\n");
```

```
    } else {
        printf("The number is less than or equal to 5.\n");
    }

    return 0;
}
```

1. else if Statement:
   The `else if` statement is used when there are multiple conditions to check. It allows testing multiple expressions and executing the corresponding block of code for the first true condition encountered.

Syntax:

```
if (expression1) {
    // Code block to execute if expression1 is true
} else if (expression2) {
    // Code block to execute if expression2 is true
} else {
    // Code block to execute if all previous expressions are false
}
```

Example:

```
#include <stdio.h>

int main() {
    int num = 10;

    if (num < 5) {
        printf("The number is less than 5.\n");
    } else if (num > 10) {
        printf("The number is greater than 10.\n");
    } else {
        printf("The number is between 5 and 10 (inclusive).\n");
    }

    return 0;
}
```

1. Nested if-else:
   Conditional statements can be nested inside each other to create complex decision-making logic.

Syntax:

```
if (expression1) {
    // Code block to execute if expression1 is true
    if (expression2) {
        // Code block to execute if both expression1 and expression2 are true
    } else {
        // Code block to execute if expression1 is true and expression2 is false
    }
} else {
    // Code block to execute if expression1 is false
}
```

Example:

```
#include <stdio.h>

int main() {
    int num = 10;

    if (num > 5) {
        printf("The number is greater than 5.\n");
        if (num <= 10) {
            printf("The number is less than or equal to 10.\n");
        } else {
            printf("The number is greater than 10.\n");
        }
    } else {
        printf("The number is less than or equal to 5.\n");
    }

    return 0;
}
```

Conditional statements are powerful tools for creating flexible and dynamic programs that respond to different scenarios. They play a crucial role in controlling program execution flow and enabling decision-making capabilities based on various conditions.

Loops

Loops are fundamental control structures in programming that allow repeating a block of code multiple times. In C, two commonly used loop types are the 'for' loop and the 'while' loop. These

loops help automate repetitive tasks and execute code based on specific conditions. The 'for' loop is ideal for tasks where the number of iterations is known beforehand, while the 'while' loop is useful for situations where the loop continues as long as a certain condition remains true.

1. The 'for' Loop:

The 'for' loop is used for executing a block of code repeatedly for a fixed number of times. It is widely used when you know the number of iterations beforehand. The loop consists of three parts: initialization, condition, and update.

Syntax:

```
for (initialization; condition; update) {
    // Code block to be executed iteratively
}
```

Explanation:

1. The 'initialization' part is executed only once, before the loop starts. It initializes a loop control variable.
2. The 'condition' is evaluated before each iteration. If the condition is true, the code block inside the 'for' loop is executed. If the condition is false, the loop terminates, and control moves to the next statement after the 'for' loop.
3. After each iteration, the 'update' part is executed. It updates the loop control variable, changing its value for the next iteration.
4. The loop continues until the 'condition' evaluates to false.

Example:

```
#include <stdio.h>

int main() {
    int i;

    // Loop from 1 to 5
    for (i = 1; i <= 5; i++) {
        printf("Iteration %d\n", i);
    }
```

```
    return 0;
}
```

Output:

```
Iteration 1
Iteration 2
Iteration 3
Iteration 4
Iteration 5
```

2. The 'while' Loop:

The 'while' loop is used for executing a block of code repeatedly as long as a specified condition remains true. It is suitable when the number of iterations is unknown, and the loop continues until the condition becomes false.

Syntax:

```
while (condition) {
    // Code block to be executed as long as the condition is true
}
```

Explanation:

1. The 'condition' is evaluated before each iteration. If the condition is true, the code block inside the 'while' loop is executed. If the condition is false initially, the loop will not be executed at all.
2. Inside the loop, it is crucial to update the variables involved in the 'condition' to ensure that the loop eventually terminates when the condition becomes false.

Example:

```
#include <stdio.h>

int main() {
    int num = 1;

    // Loop until num becomes greater than 5
    while (num <= 5) {
        printf("Value of num: %d\n", num);
```

```
        num++; // Increment num for the next iteration
    }

    return 0;
}
```

Output:

```
Value of num: 1
Value of num: 2
Value of num: 3
Value of num: 4
Value of num: 5
```

3. The 'do-while' Loop:

The 'do-while' loop is used for executing a block of code at least once, even if the condition is false initially. It is suitable when you want to ensure the code block runs at least once before checking the condition.

Syntax:

```
do {
    // Code block to be executed at least once
} while (condition);
```

Explanation:

1. The code block inside the 'do-while' loop is executed first, and then the 'condition' is evaluated. If the condition is true, the loop continues; otherwise, the loop terminates.
2. Similar to the 'while' loop, it is important to update the variables involved in the 'condition' inside the loop to ensure that the loop eventually terminates when the condition becomes false.

Example:

```
#include <stdio.h>

int main() {
    int num = 1;
```

```
    // Loop until num becomes greater than 5
    do {
        printf("Value of num: %d\n", num);
        num++; // Increment num for the next iteration
    } while (num <= 5);

    return 0;
}
```

Output:

```
Value of num: 1
Value of num: 2
Value of num: 3
Value of num: 4
Value of num: 5
```

In this example, the 'do-while' loop executes the code block at least once before evaluating the condition. If the condition is true, the loop continues; otherwise, it terminates.

These three loop types provide different mechanisms for controlling the repetition of code in C programming. The 'for' loop is used when you know the number of iterations beforehand, the 'while' loop for condition-controlled tasks, and the 'do-while' loop when you need to ensure at least one execution of the code block. Understanding these loop types empowers programmers to efficiently design and manage various looping scenarios in their programs.

Control Flow Statements

In C, 'break', 'continue', and 'goto' are control flow statements that allow you to manipulate the flow of a program in various ways. Here's an overview of each of these statements and how they can be used:

1. break:

The 'break' statement is used to immediately exit a loop (for, while, or do-while) or a switch statement. When encountered inside a loop, 'break' terminates the loop and continues execution with the next statement after the loop. In a switch statement, 'break' is used to exit the switch block.

Example: Using break in a loop

```
#include <stdio.h>

int main() {
    int i;

    // Loop from 1 to 10
    for (i = 1; i <= 10; i++) {
        printf("%d ", i);

        // Break the loop when i reaches 5
        if (i == 5) {
            break;
        }
    }

    return 0;
}
```

Output:

```
1 2 3 4 5
```

2. continue:

The 'continue' statement is used to skip the current iteration of a loop and continue with the next iteration. When encountered inside a loop, 'continue' skips the remaining code inside the loop for the current iteration and moves to the next iteration.

Example: Using continue in a loop

```
#include <stdio.h>

int main() {
    int i;

    // Loop from 1 to 10
    for (i = 1; i <= 10; i++) {
        // Skip printing even numbers
        if (i % 2 == 0) {
            continue;
        }
        printf("%d ", i);
    }
```

```
    return 0;
}
```

Output:

```
1 3 5 7 9
```

3. goto:

The 'goto' statement allows jumping to a specified label in the code. It is generally discouraged due to its potential to create spaghetti code (hard-to-read and maintain code). However, in some situations, it can be useful for breaking out of nested loops or handling error conditions.

Syntax:

```
goto label;

// ...

label:
// Code to be executed after the goto statement
```

Example: Using goto to handle an error condition

```
#include <stdio.h>

int main() {
    int num;

    // Ask the user for a positive number
    printf("Enter a positive number: ");
    scanf("%d", &num);

    // Check if the number is positive
    if (num <= 0) {
        printf("Error: Entered number is not positive.\n");
        goto end; // Jump to the end of the program
    }

    printf("The square of the number is: %d\n", num * num);

    end:
    printf("Program ends here.\n");
```

```
    return 0;
}
```

Output 1 (when a positive number is entered):

```
Enter a positive number: 5
The square of the number is: 25
Program ends here.
```

Output 2 (when a non-positive number is entered):

```
Enter a positive number: -2
Error: Entered number is not positive.
Program ends here.
```

In summary, 'break', 'continue', and 'goto' are control flow statements that provide ways to modify the flow of a program. 'break' is used to exit a loop or switch, 'continue' skips the current iteration of a loop, and 'goto' allows jumping to a specified label. While 'break' and 'continue' are commonly used in loops for control flow, 'goto' is not recommended and should be used with caution to avoid creating complex and hard-to-maintain code.

Switch Case

The switch-case statement in C is used for multiway branching, providing an elegant way to choose one of several possible options based on the value of an expression. It simplifies the code and makes it more readable when dealing with multiple conditions. Here's an overview of the switch-case statement and how it works:

Overview:

The switch-case statement is designed to handle multiway branching. It takes an expression as input and evaluates it against a series of constant values specified as case labels. If there is a match, the code block corresponding to that case is executed. If no case matches the expression, an optional default case can be used to provide a fallback action.

Syntax:

```
switch (expression) {
    case constant_value1:
        // Code block to be executed if expression matches constant_value1
        break;
    case constant_value2:
        // Code block to be executed if expression matches constant_value2
        break;
    // More cases can be added as needed
    default:
        // Code block to be executed if no case matches the expression
}
```

Explanation:

1. The 'expression' is evaluated, and its value is compared with the constant values specified in each case label.
2. If a case label matches the value of the 'expression', the corresponding code block is executed. After executing the code in that case block, the 'break' statement is used to exit the switch statement, preventing fall-through to other cases.
3. If no case matches the 'expression', the code inside the 'default' block (if present) is executed. The 'default' case acts as a fallback and is optional. If there is no 'default' case and no matching case, the switch statement simply does nothing and proceeds to the next statement in the program.

Example: Using switch-case for multiway branching

```
#include <stdio.h>

int main() {
    char grade;

    printf("Enter your grade (A, B, C, D, or F): ");
    scanf(" %c", &grade);

    switch (grade) {
        case 'A':
            printf("Excellent!\n");
            break;
        case 'B':
        case 'C':
            printf("Well done!\n");
            break;
```

```
        case 'D':
            printf("You passed, but you can do better.\n");
            break;
        case 'F':
            printf("Sorry, you failed.\n");
            break;
        default:
            printf("Invalid grade entered.\n");
    }

    return 0;
}
```

Output:

```
Enter your grade (A, B, C, D, or F): B
Well done!
```

In this example, the user is asked to enter their grade, and the switch-case statement matches the input grade to different cases. The appropriate message is then printed based on the grade entered. If the entered grade doesn't match any case, the default case handles the situation.

The switch-case statement is particularly useful when you have multiple options to choose from based on the value of a single expression. It enhances the readability of the code and makes it easy to understand the branching logic. However, it's important to use break statements to prevent unintended fall-through between cases.
