CONTROL FLOW

Sercan Külcü | 21.06.2023

Contents

1 Decision-making statements

1.1 if statement

In this chapter, we'll dive into the fascinating realm of decision-making statements, specifically focusing on the "if" statement. Decision-making is a fundamental concept in programming, allowing us to create dynamic and intelligent programs. So, let's get started!

The "if" statement is a powerful tool in C programming that enables us to make decisions based on certain conditions. It allows our program to execute different sets of instructions depending on whether a particular condition evaluates to true or false. By utilizing the "if" statement, we can control the flow of our program and make it more responsive to different scenarios.

The syntax of the "if" statement is as follows:

```
if (condition)

{

   // Code block executed when condition is true

}
```

The condition is an expression that evaluates to either true or false. If the condition is true, the code block enclosed within the curly braces ({}) will be executed. If the condition is false, the code block will be skipped, and the program will proceed to the next statement after the closing brace.

The condition within the "if" statement often involves the use of relational operators. These operators compare the values of operands and return either true or false. Here are some commonly used relational operators:

- == (equal to): Checks if two values are equal.
- != (not equal to): Checks if two values are not equal.
- < (less than): Checks if the left operand is less than the right operand.
    - (greater than): Checks if the left operand is greater than the right operand.
- <= (less than or equal to): Checks if the left operand is less than or equal to the right operand.

- >= (greater than or equal to): Checks if the left operand is greater than or equal to the right operand.

Let's take a look at some examples to illustrate the usage of "if" statements:

Example 1:

```
int age = 25;

if (age >= 18)

{

   printf("You are eligible to vote!\n");

}
```

In this example, if the value of the variable age is greater than or equal to 18, the message "You are eligible to vote!" will be printed.

Example 2:

```
int num = 7;

if (num % 2 == 0)

{

   printf("The number is even.\n");

}

else

{

   printf("The number is odd.\n");

}
```

Here, we check if the value of num is divisible by 2. If it is, the message "The number is even" is printed; otherwise, the message "The number is odd" is printed.

We can also nest "if" statements within each other to handle multiple conditions. This allows us to create more complex decision-making structures. Here's an example:

```
int x = 10;

if (x > 0)

{

   if (x < 20)

   {

      printf("The number is positive and less than 20.\n");

   }

}
```

In this case, the inner "if" statement is only executed if the outer "if" condition is true. If both conditions are satisfied, the message "The number is positive and less than 20" will be displayed.

1.2 else if statement

In this chapter, we'll explore another essential aspect of decision-making statements: the "else if" statement. The "else if" statement allows us to handle multiple conditions and create more sophisticated decision-making structures. So, let's dive in!

The "else if" statement is an extension of the "if" statement that provides additional options for decision-making. While the "if" statement allows us to execute code when a condition is true, the "else if" statement enables us to test multiple conditions in a cascading manner. It provides a way to handle different scenarios by checking subsequent conditions when the previous ones are not satisfied.

The syntax of the "else if" statement is as follows:

```
if (condition1)

{

   // Code block executed when condition1 is true

}

else if (condition2)
```

```
{
    // Code block executed when condition2 is true
}
else if (condition3)
{
    // Code block executed when condition3 is true
}
// ...
else
{
    // Code block executed when none of the conditions are true
}
```

The "else if" statement follows the "if" statement and is followed by either another "else if" statement or an optional "else" statement. The conditions are evaluated one by one, and as soon as a condition evaluates to true, the corresponding code block is executed. If none of the conditions are true, the code block within the final "else" statement (if present) will be executed.

Let's consider an example to illustrate the usage of the "else if" statement:

Example:

```
int score = 85;
if (score >= 90)
{
    printf("You received an A grade!\n");
}
else if (score >= 80)
{
    printf("You received a B grade!\n");

}

else if (score >= 70)

{

    printf("You received a C grade!\n");

}

else if (score >= 60)

{

    printf("You received a D grade!\n");

}

else

{

    printf("You received an F grade. Please work harder!\n");

}
```

In this example, the program checks the value of the variable score against multiple conditions. Depending on the score, it prints out the corresponding grade. The "else if" statements provide a cascading effect, allowing the program to choose the appropriate code block based on the conditions.

When using "else if" statements, it's important to consider the order of conditions. The conditions are evaluated sequentially from top to bottom, and the first condition that evaluates to true will be executed. Therefore, if a condition is true earlier in the sequence, the subsequent conditions will be skipped. Be mindful of this behavior when structuring your decision-making statements.

Similar to "if" statements, we can nest "else if" statements within each other to handle more complex scenarios. By combining different conditions and nesting statements, we can create intricate decision-making structures to accommodate a variety of possibilities.

1.3 switch statement

In this chapter, we'll explore yet another powerful decision-making statement: the "switch" statement. The "switch" statement provides an elegant and efficient way to handle multiple possibilities and make decisions based on different cases. So, let's dive in and uncover the wonders of the "switch" statement!

The "switch" statement is a versatile tool that simplifies decision-making when we have multiple cases to consider. It provides an alternative approach to handling multiple conditions compared to long chains of "if" and "else if" statements. With the "switch" statement, we can evaluate the value of an expression and execute different blocks of code based on its matching case.

The syntax of the "switch" statement is as follows:

```
switch (expression)

{

  case constant1:

    // Code block executed when expression matches constant1

    break;

  case constant2:

    // Code block executed when expression matches constant2

    break;

  // ...

  default:

    // Code block executed when no case matches the expression

}
```

The "switch" statement starts with the keyword "switch," followed by the expression that will be evaluated. Each "case" represents a specific constant value that we want to match against the expression. When a match is found, the corresponding code block is executed. The "break" statement is used to exit the switch block after executing the corresponding case. If none of the cases match the expression, the code block within the "default" section (if present) will be executed.

Let's look at an example to illustrate the usage of the "switch" statement:

Example:

```
int day = 3;

switch (day)

{

   case 1:

      printf("Sunday\n");

      break;

   case 2:

      printf("Monday\n");

      break;

   case 3:

      printf("Tuesday\n");

      break;

   case 4:

      printf("Wednesday\n");

      break;

   case 5:

      printf("Thursday\n");

      break;

   case 6:

      printf("Friday\n");

      break;

   case 7:

      printf("Saturday\n");
```

```
        break;
    default:
        printf("Invalid day\n");
}
```

In this example, the program evaluates the value of the variable "day" and matches it against the cases. Since "day" has a value of 3, the code block under the case 3 label is executed, printing "Tuesday" to the console. The "break" statement ensures that the program exits the switch block after executing the matching case.

Each "case" block within the switch statement must end with a "break" statement. Without the "break" statement, the program would continue executing the code blocks of subsequent cases, leading to unexpected behavior. The "break" statement ensures that the program exits the switch block after executing the matching case.

The "default" case is optional and serves as a catch-all case when none of the other cases match the expression. It is useful for handling unexpected or invalid inputs. Including a "default" case ensures that the program always has a code block to execute, even if none of the other cases match.

Benefits of the "switch" Statement

- Readability: The "switch" statement enhances the readability of our code, especially when dealing with multiple cases. It allows us to present our decision-making logic in a clear and concise manner, making it easier for others (including our future selves) to understand and maintain the code.
- Efficiency: The "switch" statement can often be more efficient than long chains of "if" and "else if" statements. When evaluating the expression, the "switch" statement performs a direct jump to the matching case, avoiding the need to check each condition one by one. This can result in faster execution, especially when dealing with a large number of cases.
- Simplified Logic: Using the "switch" statement can simplify our decision-making logic by grouping related cases together. It allows us to handle different scenarios in a structured and organized way, making it easier to manage and modify our code as requirements evolve.

While the "switch" statement is a powerful tool, there are a few restrictions to keep in mind:

- Limited Expression Types: The expression within the "switch" statement must be of integral type, such as integers or characters. Floating-point numbers and strings are not allowed as expressions.
- Constant Cases: The cases within the "switch" statement must be constants. This means that the case labels must be known at compile-time and cannot be variables or expressions.
- Unique Cases: Each case label within a "switch" statement must be unique. Duplicate case labels are not allowed.

In C, the "switch" statement has a fall-through behavior. This means that if a case block does not end with a "break" statement, the program will continue executing the code blocks of subsequent cases until a "break" statement is encountered. Fall-through behavior can be useful in certain scenarios where multiple cases should execute the same code.

2 Looping statements

2.1 for loop

In this chapter, we'll explore one of the most important concepts in programming: looping. Looping allows us to repeat a set of instructions multiple times, enabling us to automate repetitive tasks and create efficient programs. We'll dive into the "for" loop, a versatile and widely used looping statement in C. So, let's get started and unlock the power of the "for" loop!

The "for" loop is a fundamental construct in C programming that allows us to repeatedly execute a block of code as long as a certain condition remains true. It provides a concise and structured way to perform iterations, making it ideal for situations where we know the exact number of iterations required.

The syntax of the "for" loop is as follows:

```
for (initialization; condition; increment/decrement)

{

   // Code block executed in each iteration

}
```

The "for" loop consists of three components:

- Initialization: This component is executed before the loop starts and is typically used to initialize the loop control variable. It is executed only once at the beginning of the loop.
- Condition: The condition is evaluated before each iteration. If the condition evaluates to true, the loop's code block is executed. If it evaluates to false, the loop terminates, and the program continues with the next statement after the loop.
- Increment/Decrement: After each iteration, the increment or decrement statement is executed. It updates the loop control variable, bringing it closer to the termination condition.

Let's look at an example to illustrate the execution of a "for" loop:

Example:

```
for (int i = 1; i <= 5; i++)

{

   printf("Iteration %d\n", i);

}
```

In this example, the loop control variable "i" is initialized to 1. The condition "i <= 5" is evaluated before each iteration. As long as the condition remains true, the code block inside the loop is executed. After each iteration, the increment statement "i++" updates the value of "i" by adding 1. This process continues until "i" is no longer less than or equal to 5.

The output of this loop would be:

```
Iteration 1

Iteration 2

Iteration 3

Iteration 4

Iteration 5
```

The "for" loop is commonly used for iteration over a range of values. By utilizing the loop control variable and the termination condition, we can precisely control the number of iterations. This is particularly useful when working with arrays, performing calculations, or processing a set of data.

Example:

```
int sum = 0;

for (int i = 1; i <= 100; i++)

{

   sum += i;

}

printf("The sum of numbers from 1 to 100 is: %d\n", sum);
```

In this example, the loop iterates from 1 to 100, continuously adding the loop control variable "i" to the "sum" variable. At the end of the loop, the sum of numbers from 1 to 100 is printed.

Just like "if" and "else if" statements, "for" loops can also be nested within each other. This allows us to create more complex patterns and perform multidimensional iterations. By carefully managing the initialization, condition, and increment/decrement statements of the nested "for" loops, we can achieve precise control over the iterations.

Example:

```
for (int i = 1; i <= 5; i++)

{

   for (int j = 1; j <= i; j++)

   {

      printf("* ");

   }

   printf("\n");

}
```

In this example, we have two nested "for" loops. The outer loop controls the rows, while the inner loop controls the columns. With each iteration of the outer loop, the inner loop prints asterisks, creating a pattern of increasing stars in each row. The output would be:

```
*

* *

* * *

* * * *

* * * * *
```

The loop control variable plays a crucial role in the "for" loop. It determines the starting value, termination condition, and how the variable changes in each iteration. It's important to manage the loop control variable correctly to ensure that the loop behaves as expected and doesn't result in an infinite loop.

In some cases, a loop may inadvertently become an infinite loop, where the termination condition is never satisfied. This can lead to the program getting stuck in an endless loop, causing it to become unresponsive. To prevent this, we must carefully design our loops and ensure that the termination condition can be met based on the logic of the program.

In situations where we want to exit a loop prematurely, we can use the "break" statement. The "break" statement immediately terminates the loop and transfers control to the statement following the loop.

With the "for" loop in your arsenal, you have gained the ability to efficiently process data, traverse arrays, and perform iterative calculations. In the next chapter, we'll explore another powerful looping statement: the "while" loop. Get ready to further enhance your programming skills and unlock new possibilities. Keep up the great work!

2.2 while loop

This time, we'll explore the "while" loop, another powerful construct that allows us to repeat a set of instructions as long as a condition remains true. The "while" loop offers flexibility and adaptability, making it an essential tool in your programming toolkit. So, let's embark on this journey and uncover the wonders of the "while" loop!

The "while" loop is a fundamental looping construct in C programming. It allows us to execute a block of code repeatedly as long as a specified condition remains true. The "while" loop is particularly useful when the number of iterations is uncertain or when we want to repeat a certain task until a specific condition is met.

The syntax of the "while" loop is as follows:

```
while (condition)

{

   // Code block executed as long as the condition is true

}
```

The "while" loop begins with the keyword "while," followed by a condition in parentheses. The code block inside the loop is executed only if the condition evaluates to true. If the condition is false initially, the code block will not be executed at all.

Let's look at an example to illustrate the execution of a "while" loop:

Example:

```
int count = 0;

while (count < 5)

{

   printf("Count: %d\n", count);

   count++;

}
```

In this example, we have initialized a variable named "count" to 0. The condition "count < 5" is evaluated before each iteration. As long as the condition remains true, the code block inside the loop is executed. Within the loop, we print the current value of "count" and then increment it by 1 using the "count++" statement. This process continues until the condition becomes false, at which point the program exits the loop.

The output of this loop would be:

```
Count: 0

Count: 1

Count: 2

Count: 3

Count: 4
```

One common use of the "while" loop is for input validation. We can repeatedly ask the user for input until they provide valid input that satisfies certain conditions.

Example:

```
int number;
```

```
printf("Enter a positive number: ");
scanf("%d", &number);


while (number <= 0)
{
    printf("Invalid input! Please enter a positive number: ");
    scanf("%d", &number);
}
printf("You entered: %d\n", number);
```

In this example, we prompt the user to enter a positive number. If the user provides a non-positive number, the loop continues to execute, displaying an error message and asking for input until a positive number is entered.

The condition in the "while" loop is crucial, as it determines whether the loop's code block will be executed or not. It's essential to ensure that the condition is appropriately defined and that it will eventually become false to prevent infinite loops.

Similar to the "for" loop, a "while" loop can also become an infinite loop if the condition is always true. This can result in the program getting stuck and becoming unresponsive. To avoid this, we must carefully design the loop's condition to allow for termination.

To exit a "while" loop prematurely, we can use the "break" statement. The "break" statement immediately terminates the loop and transfers control to the statement following the loop.

Example:

```
int number;
while (1)
{
    printf("Enter a number (0 to exit): ");
    scanf("%d", &number);


    if (number == 0)
    {
        printf("Exiting the loop...\n");
        break;
    }


    printf("You entered: %d\n", number);
}
```

In this example, the loop runs indefinitely until the user enters 0. If the user enters 0, the loop is terminated using the "break" statement, and the program continues to the statement following the loop.

The choice between a "while" loop and a "for" loop depends on the specific requirements of the program. While the "for" loop is commonly used when the number of iterations is known, the "while" loop is preferred when the exact number of iterations is uncertain or when we need to repeatedly execute a block of code until a specific condition is met.

2.3 do-while loop

In this chapter, we'll explore another powerful looping construct: the "do-while" loop. The "do-while" loop is similar to the "while" loop, but with one key difference: the code block is executed at least once, regardless of the condition. Join me as we dive into the fascinating world of the "do-while" loop and unlock its potential!

The "do-while" loop is a versatile looping construct that allows us to repeat a block of code as long as a specified condition remains true. It is particularly useful when we want to ensure that the code block executes at least once before checking the condition. With the "do-while" loop, you have the power to create robust and adaptable programs.

The syntax of the "do-while" loop is as follows:

```
do

{

   // Code block executed at least once

}

while (condition);
```

In the "do-while" loop, the code block is executed first, and then the condition is evaluated. If the condition evaluates to true, the loop continues, and the code block is executed again. If the condition evaluates to false, the loop terminates, and the program continues with the next statement after the loop.

Let's look at an example to illustrate the execution of a "do-while" loop:

Example:

```
int count = 0;

do

{

   printf("Count: %d\n", count);

   count++;

}

while (count < 5);
```

In this example, we initialize a variable named "count" to 0. The code block inside the loop is executed first, which prints the current value of "count." Then, the variable "count" is incremented by 1 using the "count++" statement. The condition "count < 5" is evaluated after each iteration. As long as the condition remains true, the loop continues. Once the condition becomes false, the loop terminates.

The output of this loop would be:

```
Count: 0
```

```
Count: 1

Count: 2

Count: 3

Count: 4
```

One common use of the "do-while" loop is in menu-driven programs. In such programs, we repeatedly display a menu of options to the user, and the user selects an option until they choose to exit.

Example:

```
int choice;

do

{

    printf("Menu:\n");

    printf("1. Option 1\n");

    printf("2. Option 2\n");

    printf("3. Exit\n");

    printf("Enter your choice: ");

    scanf("%d", &choice);


    switch (choice)

    {

        case 1:

            printf("Option 1 selected.\n");

            break;

        case 2:

            printf("Option 2 selected.\n");
            break;

        case 3:

            printf("Exiting the program...\n");

            break;

        default:

            printf("Invalid choice. Please try again.\n");

            break;

    }

}

while (choice != 3);
```

In this example, we display a menu to the user and prompt them to enter their choice. The code block inside the loop processes the user's choice using a switch statement. If the user chooses option 1 or option 2, the corresponding message is displayed. If the user chooses option 3, the loop is terminated using the condition choice != 3, and the program exits.

While the "do-while" loop guarantees the execution of the code block at least once, it's crucial to ensure that the loop condition is correctly defined to prevent unexpected behavior. The condition determines whether the loop should continue executing or terminate.

As with any loop, it's possible for a "do-while" loop to become an infinite loop if the condition is always true. This can lead to the program getting stuck and becoming unresponsive. To prevent this, ensure that the loop condition will eventually become false to allow for termination.

If you need to exit the "do-while" loop prematurely, you can use the "break" statement, just as in the "for" and "while" loops. The "break" statement immediately terminates the loop and transfers control to the statement following the loop.

Both the "do-while" loop and the "while" loop allow for repetitive execution of code based on a condition. However, the key difference lies in when the condition is evaluated. In the "while" loop, the condition is checked before the code block is executed, whereas in the "do-while" loop, the code block is executed first and then the condition is evaluated. This ensures that the code block is executed at least once in the "do-while" loop.

3 Break and continue statements

In this chapter, we'll explore two powerful control flow statements: the "break" and "continue" statements. These statements offer us greater control over loops, allowing us to alter their behavior and flow. Join me as we uncover the capabilities of the "break" and "continue" statements and learn how to harness their power!

3.1 "break" Statement

The "break" statement is a control flow statement that allows us to immediately exit a loop or switch statement, transferring control to the statement following the loop or switch. It is useful when we want to terminate a loop prematurely or break out of a switch statement once a condition is met.

Let's explore how the "break" statement works in loops. When the "break" statement is encountered inside a loop, the loop is immediately terminated, and program execution continues with the statement following the loop.

Example:

```
int i;

for (i = 1; i <= 10; i++)

{

   if (i == 5)

       break;

   printf("%d ", i);

}
```

In this example, the loop iterates from 1 to 10. However, when the value of "i" becomes 5, the "break" statement is encountered. This causes the loop to terminate, and the program continues executing the statement following the loop. As a result, the output will be:

```
1 2 3 4
```

The "break" statement is commonly used in switch statements to terminate the execution of the switch block. After encountering a "break" statement, program control transfers to the statement following the switch.

Example:

```
int option = 2;

switch (option)

{

   case 1:

      printf("Option 1 selected.\n");

      break;

   case 2:

      printf("Option 2 selected.\n");

      break;

   case 3:

      printf("Option 3 selected.\n");

      break;

   default:

      printf("Invalid option.\n");

}
```

In this example, based on the value of "option," the corresponding case is executed. After executing the code block for "option 2," the "break" statement is encountered, causing the switch statement to terminate. The program then continues with the statement following the switch.

3.2 "continue" Statement

The "continue" statement is another control flow statement that allows us to skip the remaining code in a loop and continue with the next iteration. It is useful when we want to bypass specific iterations of a loop based on a condition.

Example:

```
int i;

for (i = 1; i <= 10; i++)

{

    if (i % 2 == 0)

        continue;

    printf("%d ", i);

}
```

In this example, the loop iterates from 1 to 10. However, when the value of "i" is divisible by 2 (i.e., an even number), the "continue" statement is encountered. This causes the remaining code in the loop to be skipped, and the next iteration begins. As a result, only the odd numbers are printed:

```
1 3 5 7 9
```

By combining the "break" and "continue" statements strategically, we can achieve even more precise control over the flow of our programs. These statements allow us to tailor the execution of loops to meet specific requirements. Let's take a look at an example that demonstrates the combined use of "break" and "continue" statements:

```
int i;

for (i = 1; i <= 10; i++)

{

    if (i == 5)

        break;
    if (i % 2 == 0)
        continue;

    printf("%d ", i);
}
```

In this example, we have a loop that iterates from 1 to 10. When the value of "i" is 5, the "break" statement is encountered, causing the loop to terminate. Additionally, if "i" is divisible by 2, the "continue" statement is encountered, and the remaining code within the loop is skipped, moving to the next iteration.

As a result, the output will be:

```
1 3
```

Here, we skipped the numbers 2 and 4 due to the "continue" statement, and the loop terminated when "i" became 5 due to the "break" statement.