2024-09-19

# Break and continue

## Conditional iteration

* Sometimes, you want to write a loop that will skip some iterations if a certain condition is met
* For example, you may be writing a for loop that iterates through an array of numbers, but you only want to use *even* numbers from the array
* One way to accomplish this is to nest an if statement inside the for loop that checks for the desired condition. For example:

int sum = 0;  
for(int i = 0; i < myArray.Length; i++)  
{  
 if(myArray[i] % 2 == 0)  
 {  
 Console.WriteLine(myArray[i]);  
 sum += myArray[i];  
 }  
}

Since the entire body of the for loop is contained within an if statement, the iterations where myArray[i] is odd will skip the body and do nothing.

## Skipping iterations with continue

* The continue keyword provides another way to conditionally skip an iteration of a loop
* When the computer encounters a continue; statement, it immediately returns to the beginning of the current loop, skipping the rest of the loop body
* Then it executes the update statement (if the loop is a for loop) and checks the loop condition again
* A continue; statement inside an if statement will end the current iteration only if that condition is true
* For example, this code will skip the odd numbers in myArray and use only the even numbers:

int sum = 0;  
for(int i = 0; i < myArray.Length; i++)  
{  
 if(myArray[i] % 2 != 0)  
 continue;  
 Console.WriteLine(myArray[i]);  
 sum += myArray[i];  
}

If myArray[i] is odd, the computer will execute the continue statement and immediately start the next iteration of the loop. This means that the rest of the loop body (the other two statements) only gets executed if myArray[i] is even.

* Using a continue statement instead of putting the entire body within an if statement can reduce the amount of indentation in your code, and it can sometimes make your code’s logic clearer.

## Loops with multiple end conditions

* More advanced loops may have multiple conditions that affect whether the loop should continue
* Attempting to combine all of these conditions in the loop condition (i.e. the expression after while) can make the loop more complicated
* For example, consider a loop that processes user input, which should end either when a sentinel value is encountered or when the input is invalid. This loop ends if the user enters a negative number (the sentinel value) or a non-numeric string:

int sum = 0, userNum = 0;  
bool success = true;  
while(success && userNum >= 0)  
{  
 sum += userNum;  
 Console.WriteLine("Enter a positive number to add it. "  
 + "Enter anything else to stop.");  
 success = int.TryParse(Console.ReadLine(), out userNum);  
}  
Console.WriteLine($"The sum of your numbers is {sum}");

* The condition success && userNum >= 0 is true if the user entered a valid number that was not negative
* In order to write this condition, we needed to declare the extra variable success to keep track of the result of int.TryParse
* We cannot use the condition userNum > 0, hoping to take advantage of the fact that if TryParse fails it assigns its out parameter the value 0, because 0 is a valid input the user could give

## Ending the loop with break

* The break keyword provides another way to write an additional end condition
* When the computer encounters a break; statement, it immediately ends the loop and proceeds to the next statement after the loop body
* This is the same break keyword we used in switch statements
* In both cases it has the same meaning: stop execution here and skip to the end of this code block (the ending } for the switch or the loop)
* Using a break statement inside an if-else statement, we can rewrite the previous while loop so that the variable success is not needed:

int sum = 0, userNum = 0;  
while(userNum >= 0)  
{  
 sum += userNum;  
 Console.WriteLine("Enter a positive number to add it. "  
 + "Enter anything else to stop.");  
 if(!int.TryParse(Console.ReadLine(), out userNum))  
 break;  
}  
Console.WriteLine($"The sum of your numbers is {sum}");

* Inside the body of the loop, the return value of TryParse can be used directly in an if statement instead of assigning it to the success variable
* If TryParse fails, the break statement will end the loop, so there is no need to add success to the while condition
* We can also use the break statement with a for loop, if there are some cases where the loop should end before the counter reaches its last value
* For example, imagine that our program is given an int array that a user *partially* filled with numbers, and we need to find their product. The “unused” entries at the end of the array are all 0 (the default value of int), so the for loop needs to stop before the end of the array if it encounters a 0. A break statement can accomplish this:

int product = 1;  
for(int i = 0; i < myArray.Length; i++)  
{  
 if(myArray[i] == 0)  
 break;  
 product \*= myArray[i];  
}

* If myArray[i] is 0, the loop stops before it can multiply the product by 0
* If all of the array entries are nonzero, though, the loop continues until i is equal to myArray.Length
* Note that in this example, we access each array element once and do not modify them, so we could also write it with a foreach loop:

int product = 1;  
foreach(int number in myArray)  
{  
 if(number == 0)  
 break;  
 product \*= number;  
}