# Loops

**T1** Chapters:

1. **14-16**
2. **20-23**

* A **loop** is a statement that repeatedly executes a piece of code inside its **body**. In the C++ programming language there are three kinds of loops, each equivalent to each other but more suited in different contexts - while, do-while and for.
* Each time the loop body is executed (an **iteration** of the loop), the controlling boolean expression is evaluated, and the loop continues executing as long as this expression evaluates to True.
* The structures of the loops are as follows,
* while
* while (EXPRESSION)  
  {  
   BODY  
  }
* do-while
* do  
  {  
   BODY  
  } while (EXPRESSION)
* for
* for (EXPRESSION\_1; EXPRESSION\_2; EXPRESSION\_3)  
  {  
   BODY  
  }  
    
  // range-based for-loop  
  for (TYPE VAR : CONTAINER)  
  {  
   BODY  
  }

## while

In a while loop, the EXPRESSION is evaluated **before** the body is executed. It is useful when the body of a loop needs to be executed only when the condition expression evaluates to True and repeats as long this holds true.

For example,

while (i < n)  
{  
 i = i \* 2;  
}

Here, i and n are two variables such that i < n holds true for some fixed value of n. In this case, the body of the while loop is executed repeatedly until the condition i < n becomes False.

### Infinite while Loop

A special case of a while loop is of the infinite variety which executes forever, and whose conditional/control expression always evaluates to true.

while (true)  
{  
 BODY  
}

Here, this while loop executes the body without end, unless it contains one of break, goto, return statements or a function that causes the program to terminate like exit.

### Exercises

1. Using a while loop, write a program that prints a table of squares. The program will prompt the user to enter a number n representing the number of iterations. For each iteration i starting at i = 1 and i <= n, it will print i and its square 8 spaces apart. (E)

* This program prints a table of squares.  
  Enter number of entries in table: 5  
  1 1  
  2 4  
  3 9  
  4 16  
  5 25

1. Write a program that sums a series of integers entered by the user (M)

* This program sums a series of integers.  
  Enter integers (0 to terminate): 8 23 71 5 0  
  The sum is: 107

## do-while

In a do-while loop, the controlling expression is evaluated **after** the body is executed, but otherwise it is equivalent to the while loop. It is handy when a loop must be executed at least once.

For example,

i = 10;  
do  
{  
 printf("T minus %d and counting\n", i);  
} while (i > 0);

### Exercises

1. Write a program that calculates and displays the number of digits in an integer, using a do-while loop. (M)

* Enter a nonnegative integer: 60  
  The number has 2 digit(s).

## for

The for loop, though seemingly complex, is probably the most versatile and commonly used of the three loops, and usually used to repeat a piece of code for a specific number of iterations.

For example,

for (int i = 0; i < 10; i += 1)  
{  
 printf("i = %d\n", i);  
}

Here, the for loop will print the value of i, in increasing order, from 0 to 9.

From the structure of the for loop above, it can be seen that EXPRESSION\_1 corresponds to int i = 0 which defines a new variable i with an initial value of 0, and as long as the EXPRESSION\_2 corresponding to i < 10 holds true, the code in the body corresponding to the printf will be executed, and after each iteration i is incremented by 1 as given in EXPRESSION\_3 corresponding to i += 1.

The above loop is equivalent to the following while loop which illustrates the order of operations in the for loop.

int i = 0;  
while (i < 10)  
{  
 printf("i = %d\n", i);  
 i += 1;  
}

The difference in the two loops is related to the visibility of the i variable after the loop is finished execution. In the case of the for loop, i is not visible or accessible to the code after the loop, while in the case of the while loop, i is visible and accessible which can lead to subtle errors that can be hard to debug in a complex codebase. The technical term for visibility is **scope**, and is something that doesn't become particularly important we encounter **functions**.

**NOTE** All expressions in the for loop header are optional. An infinite while loop can be emulated by a for loop as follows,

for (;;)  
{  
 BODY  
}

Here, EXPRESSION\_1, EXPRESSION\_2 and EXPRESSION\_3 are omitted. Similarly, the previous while loop can be emulated using for loop as follows,

int i = 0;  
for (; i < 10;)  
{  
 printf("i = %d\n", i);  
 i += 1;  
}

### Exercises

1. Rewrite the table of squares program using a for loop, instead of a while loop. (E)
2. In the program above, use addition instead of multiplication to compute the squares. (M)
3. Using a for loop, write a program that asks the user to enter two integers, then calculates and displays their greatest common divisor (GCD). (H)

* Enter two integers: 12 28  
  GCD: 4

## Exiting from a Loop

* Aside from infinite loops, loops stop execution when their conditional or controlling expression does not hold True anymore. However, there will be situations when it will be necessary to exit from the loop early. An example of such a situation would be when we have found the largest number in a list or when the user has entered an invalid input.
* To handle such situations, the C++ programming language provides multiple facilities, listed below in decreasing order of frequency of use, of which we will only learn about the first two.
  + break
  + return
  + exit(ERROR\_CODE)
  + goto

## break

The break statement is used to exit from a loop that it is used in.

For example,

for (int i = 0; i < 10; i += 1)  
{  
 if (i > 5)  
 {  
 break;  
 }  
}

Here, when the value of i is greater than 5, the loop is exited.

**NOTE 1** The break statement can only be used inside the body of a loop, and when used inside a conditional statement, like in the example above, the conditional statement must be in the body of a loop.

**NOTE 2** The effect of a break statement inside a loop is different from the one inside a switch statement, in that, the one inside a loop exits out of the loop even if inside a conditional statement. However, a break in a switch statement inside a loop will only exit the switch statement and not the loop.

For example,

while (expr)  
{  
 switch (v)  
 {  
 case x:  
 break;  
 case y:  
 break;  
 case z:  
 break;  
 default:  
 }  
}

the loop above will continue executing until expr becomes False, even though there is a break inside the switch statement. This is because the break in the switch only works to prevent from checking the next case block, thus causing an exit out of the switch.

## return

The return statement, in the context of loops, is typically used when a loop is inside a **function** and is used to exit not only from the loop but also from the **function** itself. For example,

for (int i = 0; i < 10; i += 1)  
{  
 if (i > 5)  
 {  
 return i;  
 }  
}

In the loop above, the return exits the loop and also the function in which the loop exists with the value of i when i > 5.

## Skipping Iterations

There will also be situations where an iteration might need to be skipped when a particular condition is met, and to handle these situations the continue statement is provided. One of can think of the continue statement as complimentary to the break statement which stops the loop immediately, while the continue makes the loop skip an iteration.

For example,

for (int i = 0; i < 10; i += 1)  
{  
 if (i < 5)  
 {  
 continue;  
 } else {  
 printf("i = %d\n", i);  
 }  
}

the loop above will only print the values of i that are greater than or equal to 5, and in all other cases will skip to the next iteration. If the continue statement was not available and rewritten using a while loop, it is equivalent to the following:

int i = 0;  
while (i < 10)  
{  
 if (i >= 5)  
 {  
 printf("i = %d\n", i);  
 }  
 i += 1;  
}

### Exercises

1. Write a program that prints a one-month calendar. The user specifies the number of days in the month and the day of the week on which the month begins. You must use the break statement. (H)

* Enter number of days in month: 31  
  Enter starting day of the week (1=Sun, 7=Sat): 3  
   1 2 3 4 5  
   6 7 8 9 10 11 12  
  13 14 15 16 17 18 19  
  20 21 22 23 24 25 26  
  27 28 29 30 31

# Arrays I

So far, the kinds of objects that we have dealt with are all **scalar**. Meaning the variables hold only one value, however, there will be situations where we want a variable to refer to multiple values of the same type. This is where arrays come in.

Declaring an array is similar to a variable,

TYPE NAME[N] = {0};

The only new syntax that you have to remember is the addition of the square brackets, the size of the array N and the initialization expression on the right-hand side. In this case, we're initializing the contents of the array to be the default value of the TYPE.

For instance,

int x[3] = {0};

declares an array x with 3 elements, all initialized to 0.

int x[3];

declares an array x with 3 elements. The values of the elements is undefined.

int x[3] = {1};

declares an array x with 3 elements, whose first element is 1, but the remaining elements are 0.

## Index and Position

An **index** is a variable that holds a **position** value and used to access elements in an array. For instance,

int i = 3;
  
int x[4] = {1,2,3,4};
  
printf("The last element of x is %d\n", x[i]);

the snippet above will display the element 4 at position i = 3 on the console.

When **index** is used as a verb, as in the statement *index into the array*, it refers to the process of using an index variable to access an element at the position indicated by the value held by the index.

The terminology is important to familiarize yourself with, especially in the context of using arrays within loops.

## Arrays and Loops

#include <cstdio>
  
  
int main()
  
{
  
 constexpr int N = 5;
  
 int x[N] = {0};
  
 for (int i = 0; i < N; i += 1)
  
 {
  
 printf("x[%d] = %d\n", i, x[i]);
  
 }
  
}

In the above program, we have a simple for-loop that's iterating over the array x and printing out the value of each element. Here, the index variable is i that's used to index into x.

The same loop can be used to index into another array if the index is within bounds.

#include <cstdio>
  
  
int main()
  
{
  
 constexpr int N = 5;
  
 int x[N] = {0,1,2,3,4};
  
 int y[N] = {5,6,7,8,9}
  
 for (int i = 0; i < N; i += 1)
  
 {
  
 printf("x[%d] = %d, y[%d] = %d\n", i, x[i], i, y[i]);
  
 }
  
}

**NOTE**

1. Remember that once an array has been declared, its size cannot be modified. If the size of an array must be increased, a new array be declared with the desired size and the elements from the original array must be copied over.
2. When an array is declared, the elements are always laid out, in memory, side by side or **contiguously**, making them extremely efficient for sequential and random-access.
3. In the C/C++ programming languages, you cannot print an array the same way you would print a scalar variable. You need to use a loop to achieve this.
4. Similarly, there are no functions that tell you the number of elements in an array.
5. Array variables cannot be assigned like scalar variables.

# Exercises

1. Print the even and odd elements of an array. (E)
2. Modify the elements of an array such that they are raised to the power of two. (M)
3. Swap the elements of an array such that the array is reversed. (M)
4. Write a program that increases the size of the array when it is full, and shrinks it when it is empty. The elements to fill the array must be obtained from the user and must remove elements for display when requested by the user. (H)

# Homework

1. Chapters:
   1. **24-26**
2. Write a program that asks the user to enter a list of numbers, store them in an integer array and display them to the console using a for loop. (E)
3. Write a program that finds the largest element of an array using a for loop. The program should display both the index and the value of this element in the array. (M)
4. Modify the above program to use the range-based for-loop. (H)