Loops

Looping, or iteration, lets us repeat some code over and over again until a certain condition is met. A very common reason for doing this is if you have a list of objects, such as a list of all enemies in a level, and you want to do something for each one of them. Or perhaps you want to create a certain number of items, or even draw a series of lines where the logic is the same and only the position changes. Many, many things become simpler and easier when using loops.

# While Statement

A while statement looks just like an if statement, but replaces the if with the keyword while.

while (expression)

{

statement

}

Like an if statement, the code inside the curly brackets will only execute if **expression** evaluates to true. The difference is that **statement** will be executed repeatedly until **expression** becomes false.

The example below will execute the statement in the body of the loop five times. Once i becomes 5, the controlling expression is false, so the statement won’t be executed again.

int count = 5;

int i = 0;

while (i < count)

{

++i;

}

Note: It is a very common convention to use a lowercase i, standing for “iterator,” as the name of a variable which counts loops like this.

When writing loops it is important to make sure you have some condition that will make the controlling expression evaluate to false, otherwise you will end up with an **infinite loop** which never stops.

# Do While Statement

A while statement will only execute its code if the controlling expression initially evaluates to true, so in some cases the code will never run. If we want to make sure the statement is executed at least once, we can use a do while statement.

do

{

statement

}

while (expression);

This behaves just like a while statement except that it switches the order of when the expression is checked and when the statement is executed. In this case, the statement is run first, and then the expression is checked to see if it should be run again. This means that the code in the curly brackets will be executed at least once no matter what.

# For Statement

Although it looks different, a for statement does the same thing as a while statement. First let’s see what a for statement looks like:

for (initialize\_expression; controlling\_expression; increment\_expression)

{

statement

}

This is the sequence of what happens when the for statement is evaluated:

1. Execute **initialize\_expression**.
2. Evaluate **controlling\_expression**.
3. If the result is true, execute **statement**. If it is false, stop the loop.
4. Execute **increment\_expression**.
5. Go to step 2.

This process can be rewritten using an equivalent while statement:

initialize\_expression;

while (controlling\_expression)

{

statement

increment\_expression;

}

Any for loop can be written as a while loop and vice versa. The following loops both print the numbers 1 through 10, assuming a Print function that takes a number and prints it to the console.

|  |  |
| --- | --- |
| int i = 1;  while (i <= 10)  {  Print(i);  ++i;  } | for (int i = 1; i <= 10; ++i)  {  Print(i);  } |

Note that in C, since variables must be declared at the beginning of the code, you must declare int i before the for loop.

In this case we are simply adding one to the variable i after each loop, but you can do almost anything you want in that expression.

We can also put more than one expression in each section of the for loop by separating them with commas.

for (int i = 0, int value = 1; i < 10 && value < 100; ++i, value \*= 2)

{

Print(value);

}

We can also leave out some of the sections if we don’t need them.

int i = 1;

for (; i <= 10; ;)

{

Print(i++);

}

In the above example, we might as well use a while loop instead.

# Break and Continue

If we want to exit a loop prematurely, we can use the **break** statement. This will stop executing the loop and take us straight to the code that follows it.

The following example prints the numbers 1 through 10.

int i = 1;

while (true)

{

Print(i++);

if (i > 10)

{

break;

}

}

If we want to skip part of the loop’s statements but still keep on looping, we can use the **continue** statement. This will skip whatever code is left in the body of the loop. In a while loop it will go straight to evaluating the controlling expression to check if it is still true, and in a for loop it will execute the third expression in the parentheses and then evaluate the second expression to check if it is true.

The following example prints even numbers from 0 to 20.

for (int i = 0; i <= 20; ++i)

{

if (IsThisNumberEven(i) == false)

{

continue;

}

Print(i);

}

# Scope

Like we described in the previous chapter on Conditionals, all of these looping statements create a new block scope within their curly braces.