How many times does the code snippet given below display "Loop Execution"?

```
int i = 0;
while (i != 9)
{
   cout << "Loop Execution" << endl;
   i++;
}
```

A. Infinite times
B. 8 times
[✓ C. 9 times]
D. 10 times
E. 11 times

How many times does the loop run in the following code fragment?

```
int i;
for (i = 0; i < 60; i = i + 4)
{
   cout << i << endl;
}
```

A. 1
B. 14
[✓ C. 15]
D. 16

If you declare a variable inside a loop, which statement is not true?

A. It won't be accessible outside of the loop.
B. It is a different variable every time the loop runs.
C. The scope of the variable is one execution of the body.
[✓ **D.** The current value of the variable is 0 after the loop ends.]

- Strategies for processing user input
  - Input validation with if statements
  - Processing input with loop structures

5

Input Validation with if Statements

Apply if statements to detect whether user input is valid.

- When reading a value, check that it is within the required range.
- Use the `fail` function to test whether the input stream has failed.

7

Input Validation with if Statements

Example:

- Assume that the elevator panel has buttons labeled 1 through 20.
- The following are illegal inputs:
  - Zero or a negative number
  - A number larger than 20
  - A value that is not a sequence of digits, such as five
- In each of these cases, we will want to give an error message and exit the program.

8

Input Validation with if Statements

To ensure that the user doesn’t enter a number outside the valid range:

```
if (floor <= 0 || floor > 20)
{
   cout << "Error: "
        << " The floor must be between 1 and 20."
        << endl;
   return 1;
}
```

9

Input Validation with if Statements

The statement:

```
return 1;
```

immediately exits the **`main`** function and therefore terminates the program.

It is a convention to return with the value 0 if the program completes normally, and with a non-zero value when an error is encountered.

Input Validation with if Statements

Dealing with input *that is not a valid integer* is a more difficult problem.

What if the user does not type a number in response to the prompt?

‘F’ ‘i’ ‘v’ ‘e’ is not an integer response.

Input Validation with if Statements

When

```
cin >> floor;
```

is executed, and the user types in a bad input, the integer variable **`floor`** is not set.

Instead, the input stream **`cin`** is set to a failed state.

Input Validation with if Statements

You can call the **`fail`** member function to test for that failed state.

So you can test for bad user input this way:

```
if (cin.fail())
{
   cout << "Error: Not an integer." << endl;
   return 1;
}
```

Processing Input – When and How to Stop?

- We need to know, when getting input from a user, when they are done.
- One method is to hire a sentinel.

  or more correctly choose a *value* whose meaning is STOP!
- As long as there is a known range of valid data points, we can use a value not in it..

Processing Input – When and How to Stop?

Example: we will write code to calculate the average of some salary values input by the user.

How many will there be?

That is the problem. We can't know.

But we can use a sentinel value, as long as we tell the user to use it, to tell us when they are done.

Since salaries are never negative, we can safely choose **-1** as our **sentinel value**.

Processing Input – When and How to Stop?

- In order to have a value to test, we will need to get the first input before the loop. The loop statements will process each non-sentinel value, and then get the next input.
- Suppose the user entered the sentinel value as the first input. Because averages involve division by the count of the inputs, we need to protect against dividing by zero.

```
#include <iostream>
using namespace std;

int main()
{
   double sum = 0;
   int count = 0;
   double salary = 0;
   // get all the inputs
   cout << "Enter salaries, -1 to finish: ";
   while (salary != -1)
   {
      cin >> salary;
      if (salary != -1)
      {
         sum = sum + salary;
         count++;
      }
   }
   // process and display the average
   if (count > 0)
   {
      double average = sum / count;
      cout << "Average salary: " << average <<
endl;
   }
   else
   {
      cout << "No data" << endl;
   }

   return 0;
}
```

A program run:

```
Enter salaries, -1 to finish: 10 10 40 -1
Average salary: 20
```

Using Failed Input for Processing

- Sometimes it is easier and a bit more intuitive to ask the user to “Hit Q to Quit” instead of requiring the input of a sentinel value.

- Sometimes picking a sentinel value is simply impossible – if any valid number is allowed, which number could be chosen?

Using Failed Input for Processing

- In the previous slide, we used **`cin.fail( )`** to test if the most recent input failed.
- Note that if you intend to take more input from the keyboard after using failed input to end a loop,

  you must reset the keyboard with **`cin.clear()`**, to clear the stream's internal *error state flags* (the stream status back to normal, so that you can extract further data from the stream).

Using Failed Input for Processing

If we introduce a bool variable to be used to test for a failed input, we can use **cin.fail( )** to test for the input of a ‘Q’ when we were expecting a number:

```
cout << "Enter values, Q to quit: ";
bool more = true;
while (more)
{
   cin >> value;
   if (cin.fail())
   {
      more = false;
   }
   else
   {
      // process value here
   }
}
cin.clear(); //reset if taking more input
```

Using Failed Input for Processing

- Using a **`bool`** variable in this way is disliked by many programmers.
- **`cin.fail`** is set when **>>** fails
  It is not really a top or bottom test.

  If only we could use the input itself to control the loop – we can!
- An input that does not succeed is considered to be **`false`** so it can be used in the **`condition`**.

Using Failed Input for Processing

Using the input attempt directly we have:

```
cout << "Enter values, Q to quit: ";
while (cin >> value)
{
   // process value here
}
cin.clear();
```

Read the input, which caused the input failure, into a dummy variable:

```
string sentinel;
cin >> sentinel;
```

Now you can go on and read more inputs.

Extract and Discard Input Characters

`cin.ignore(`*`n,delim`*`)` member function extracts characters from the input sequence and discards them, until either *n* characters have been extracted, or one compares equal to *delim*.

`cin.get()` member function extracts a single character from the input stream if calling it without any parameter.

```
#include <iostream>
using namespace std;

int main()
{
   char first, last;

   cout << "Please enter your first name
           followed by your last name: ";

   first = cin.get();
   cin.ignore(256, ' ');
   last = cin.get();

   cout << "Your initials are " << first
        << last << endl;
   return 0;
}
```

```
cout << "Please input an integer (click q or Q to quit)";
cin >> input;

while (cin.fail())
{
      cin.clear();
      if (input == "q" || input == "Q")
      {
            break;
      }
      else
      {
            cout << "Please input an integer (click q or Q
to quit)";
            cin >> input;
      }
}
```

It will generate endless loop if the user inputs a non-integer value, why???

```
cout << "Please input an integer (click q or Q to quit)";
cin >> input;

while (cin.fail())
{
	cin.clear();
	string input_to_check;
	cin >> input_to_check;
	if (input_to_check == "q" || input_to_check == "Q")
	{
		break;
	}
	else
	{
		cout << "Please input an integer (click q or Q
to quit)";
		cin >> input;
	}
}
```

It solves the problem of endless loop! Are we done yet?

28

```
cout << "Please input an integer (click q or Q to quit)";
cin >> input;

while (cin.fail())
{
	cin.clear();
	string input_to_check;
	cin >> input_to_check;
	if (input_to_check == "q" || input_to_check == "Q")
	{
		break;
	}
	else
	{
		cout << "Please input an integer (click q or Q
to quit)";
		cin >> input;
	}
}
```

What if the user types a floating-point number instead of integer?
What if the user types an integer, then something else, such as 23cats?

29

```
int times;
cin >> times;
```

What if the user types:
-5 ↲

Everything is good!
The variable named times will store -5.

```
int times;
cin >> times;
```

What if the user types:
eight ↲

The failbit error state flag is set for the stream: cin
The variable named times will not get any value.
You need to clear the error state flag to continue…

```
int times;
cin >> times;
```

What if the user types:
7ab ↵

The status of cin is good
The variable named times will store 7.
How about the leftover???
You need to get rid of what has been left in the input stream, then continue…

```
int times;
cin >> times;
```

What if the user types:
9 do I mean nine? ↵

The status of cin is good
The variable named times will store 9.
How about the leftover???
You need to get rid of what has been left in the input stream, then continue…

```
int times;
cin >> times;
```

What if the user types:
24.5aa and something ↵

The status of cin is good
The variable named times will store 24.
How about the leftover???
You need to get rid of what has been left in the input stream, then continue…

```
int times;
cin >> times;
```

What if the user types:
Q ↵

The **failbit** error state flag is set for the stream: **cin**

The variable named times will not get any value.

You need to clear the error state flag to continue…

And if you want to check what is it?

```
cin.clear();
string input_to_check;
cin >> input_to_check;
```

```
int main()
{
        int size;
        bool validSize = false;
        do {
                cout << "Please enter a positive integer as the size: ";
                cin >> size;

                if (cin.fail()) {
                        cout << "That is not even a number!" << endl;
                        // Clear the failing state.
                        cin.clear();
                } else {
                    if (size <= 0)
                        cout << "Need a positive number!" << endl;
                    else
                        validSize = true;
                }
                cin.ignore(256, '\n'); //skip any possible bad input
        } while(!validSize);

        for (int i=0; i < size; i++)
                cout << "*";
        cout << endl;
        return 0;
}
```

In-class Question 1?

Write a program that repeatedly reads an integer from the user input and decide if it is a prime number, until the user clicks “Q” or “q” to quit the program.

A prime number (or a prime) is a natural number greater than 1 that is not a product of two smaller natural numbers.

(Note that the user input is not guaranteed to be valid)

In-class Question 2?

Write a program that repeatedly reads a sequence of integers until the user clicks “Q” or “q” to quit the process, and then prints the average of all the integers read from the user.

(Note that the user’s input is not guaranteed to be valid)