Increment and Decrement

• `++` is the increment operator. It adds one to a variable. val++; is the same as val = val + 1;

• `++` can be used before (prefix) or after (postfix) a variable:
  - `++val;` // increment, then use
  - `val++;` // use, then increment

• `--` is the decrement operator. It subtracts one from a variable. val--; is the same as val = val - 1;

• `--` can be used before (prefix) or after (postfix) a variable:
  - `--val;` // decrement, then use
  - `val--;` // use, then decrement

Notes on Increment, Decrement

• Can be used in expressions:
  - result = num1++ + --num2;

• Must be applied to something that has a location:
  - result = num1 + num2++;

• Can be used in relational expressions:
  - if (++num > limit)

• HIGH precedence, just under ( )

Increment and Decrement

While loop

• Loop: a control structure that causes a statement or statements to repeat

• General format of the while loop:
  - `while (expression) statement;`

• `statement;` can also be a block enclosed in `{}`:
  - `while (expression)
    
    statement; statement;

• Statements to repeat:

While Loop
The while loop

- Condition controlled loop
- Prime the loop (make sure it has a valid way to start)
- Change the condition inside the loop

Input Validation

- Input validation is the process of inspecting data that is given to the program as input and determining whether it is valid.
- The while loop can be used to create input routines that reject invalid data, and repeat until valid data is entered.

The following loop will never execute:
```cpp
int number = 6;
while (number <= 5)
{
    cout << "Hello\n";
    number++;
}
```

- The loop must contain code to make expression become false

The while loop

- Carefully decide what goes inside the loop and what goes outside the loop.
- The condition is tested each time at the beginning of the loop.
- When the condition is true all the instructions inside the loop are carried out before the condition is tested again.

Input Validation

- Input validation is the process of inspecting data that is given to the program as input and determining whether it is valid.
- The while loop can be used to create input routines that reject invalid data, and repeat until valid data is entered.
Input Validation

Here's the general approach, in pseudocode:

Read an item of input.
While the input is invalid
  Display an error message.
  Read the input again.
End While

cout << "Enter a number less than 10: ";
cin >> number;
while (number >= 10)
{
  cout << "Invalid Entry!
  Enter a number less than 10: ";
  cin >> number;
}

Counters

- Counter: a variable that is incremented or decremented each time a loop repeats.
- Can be used to control execution of the loop (also known as the loop control variable).
- Must be initialized before entering the loop.

Now you can simply use ++ or --. For example:

Program 5-6

```cpp
int score, count = 0; // initialize counter
int totalScore = 0; // prime the loop
cout << "Enter a score (enter -1 to quit): ";
cin >> score;
while (score != -1)
{
  totalScore += score;
  count++;
  cout << "Enter a score (enter -1 to quit): ";
  cin >> score;
}
double average = static_cast<double>(totalScore) / count;
cout << "The average score is " << average << endl;
```

End While

Read the input again.
Display an error message.
While the input is invalid.
Read an item of input.

Pseudocode:

Here's the general approach in

Input Validation
The do-while loop

- **do-while**: a posttest loop – execute the loop, then test the expression

- **General Format**:
  - loop always executes at least once
  - execution continues as long as expression is true, stops repetition when expression becomes false

- Useful in menu-driven programs to bring user back to menu to make another choice (see Program 5-8 in the book)

- **for Loop**
  - useful for counter-controlled loop
  - general format:
    ```
    for (initialization; test; update)
    {
      statement // or block in { }
    }
    ```
  - no semicolon after 3rd expression or after the `)

- **Loop always executes at least once**
- **Execution continues as long as expression is true**, stops repetition when expression becomes false
- Useful in menu-driven programs to bring user back to menu to make another choice (see Program 5-8 in the book)
- **Note that a semicolon is required after**
  ```
  while (expression)
  {
    statement // or block in { }
  }
  ```
- **always happens at least one time**
- execute one time because do-while is a posttest loop

Although the test expression is false, this loop will execute one time.
for Loop

will loop 19 times: 0 to 18

count = 0; counter = 0; counter++

for (count = 0; count <= 5; count++)
    cout << "Hello" << endl;

for Loop - Example

1) Perform initialization
2) Evaluate test expression
   - If false, terminate loop execution
   - If true, execute statement
3) Execute update, then re-evaluate test

{ } // or block in { }

for (counter=0;counter<18;counter++)
    cout<<"Hello world!\n";

for (int i=0;i<=18;i++)
    cout<<"Hello world!\n";
```cpp
for (int i = 0; i < 18; i += 2)
    cout << "Hello world!\n";
```

For loop

Statement with a comma:
The test expression separate the
You can also have multiple statements in

* **For Loop - Modifications**

You can have multiple statements in the

• When to use for loop:
  - an update to occur at the end of each iteration
  - a false condition to stop the loop
  - an initialization
  - in any situation that clearly requires

* **For Loop**

The following loop will never iterate:

* The for loop initializes, tests, and updates in each iteration before each iteration. If it is a pretest loop, the for loop is a pretest loop.

```cpp
int x, y;
for (x = 1, y = 1; x <= 5; x++, y++)
    cout << x << " plus " << y << " equals " << (x + y) << endl;
```

For loop - Modifications

• You can have multiple statements in the initialization expression. Separate the statements with a comma:

```cpp
int x, y;
for (x = 1, y = 1; x <= 5; x++)
    cout << x << " plus " << y << " equals " << (x + y) << endl;
```
Loop - Modifications

• You can omit the initialization expression if it has already been done:

```c++
int sum = 0, num = 1;
for (; num <= 10; num++)
    sum += num;
```

• You can declare variables in the initialization expression:

```c++
int sum = 0;
for (int num = 0; num <= 10; num++)
    sum += num;
```

Sentinels

• sentinel: value in a list of values that indicates end of data

Keeping a running Total

• running total: accumulated sum of numbers from each repetition of loop
• accumulator: variable that holds running total
• remember to initialize:

```c++
int sum=0, num=1; // sum is the accumulator
while (num <= 10) // accumulator
{
    sum += num;
    num++;
}
```

Sentinels

• sentinel: value in a list of values that indicates end of data

Program 5.1

Program 5.12

Program 5.12: Accumulate the running total: accumulated sum of numbers from each repetition of loop

The scope of the variable num is the for loop:

```c++
sum = num;
for (int num = 0; num <= 10; num++)
    sum = 0;
```
Sentinels

Loop through a File

This function can be used to read to the end of a file:

- The stream extraction operator `>>` returns `true` when a value was successfully read, `false` otherwise.
- This function is true when you have tried to read past the last item in the file.
- The function `input.eof()` returns true when the end of file has been reached:

```cpp
while (input.eof()) ...
```

The file stream objects have a function that returns `true` when the end of file has been reached:

```cpp
ifstream input("data.txt");
if (!input) {
    cout<<"File open Error!!
return 1;
```

```
int value;
input>>value;//prime the loop
while (!input.eof()) {
    …
    input>>value;
    //change the value of the condition
    //inside the loop
```

The stream extraction operator `>>` returns `true` when a value was successfully read from the file.

```
while (input >> number) …
```

The file:

- Can be tested in a `while` loop to continue execution as long as values are read from the file.
- `else` otherwise:

```cpp
else {
    //else block
```

```
Program: 15
```
Loop through a File

Consider a file containing the contents like:

Kevin Jones
3769.23
Tony Blair
4235.94
Sarah Allan
3200.23

Loop through a File

To perform a certain number of times, use a while-loop when a loop is to continue
until a certain condition changes.

Which loops?

• while: pretest loop; loop body may not be executed at all
• do-while: pretest loop; loop body will always be executed at least once
• for: pretest loop with initialization and update expression; useful with counters, or if precise number of repetitions is needed

Programming Rules

• Use a for-loop when a loop is to be performed a certain number of times. Use a while-loop when a loop is to continue until a certain condition changes.

Nested Loops

A nested loop is a loop inside the body of another loop.

T2: Use a for-loop when a loop is to be performed a certain number of times.
Lines from Program 5-16 - Nested Loops

- Inner loop goes through all repetitions for each repetition of outer loop
- Inner loop repetitions complete sooner than outer loop
- Total number of repetitions for inner loop is product of number of repetitions of the two loops

{   cout<<endl;   cout<<endl; for (int i=0;i<height;i++)   {      for (int j=0;j<width;j++)         cout<<"*";      cout<<endl;   }   for (int i=1;i<=height;i++)   {      for (int j=0;j<i;j++)         cout<<"*";      cout<<endl;   }}

What does this print?

- Break

- Can use break to terminate execution of a loop
- Use sparingly if at all – makes code harder to understand and debug

- Continue

- Takes you out of that iteration of a loop and enters next iteration of same loop
- Use sparingly – like break, can make program logic hard to follow

- Total number of repetitions for inner loop is product of number of repetitions of the two loops
- Each repetition of outer loop goes through all repetitions for inner loop