Why Use Repetition?

- Repetition allows you to efficiently use variables.
- Can input, add, and average multiple numbers using a limited number of variables.
- For example, you can add five numbers together by:
  - declaring a variable for each number, inputting the numbers and adding the variables together.
  - creating a loop that reads a number into a variable and adds it to a variable that contains the sum of the numbers and looping until all numbers are read.
The while Loop

while (expression)
{
    statements
}

The expression provides an entry condition
The statement executes if the expression initially evaluates to true
The loop condition is then reevaluated
If it is still true, the statement executes again
The statement continues to execute until the expression is no longer true
An infinite loop continues to execute endlessly and can be avoided by making sure that the loop's body contains statement(s) that assure that the exit condition will eventually be false

An example:
int i=1;
while (i<=5)
{
    cout<<i<<endl;
    i=i+1;
}

Another example:
int i=0;
while (i<=20)
{
    i=i+5;
    cout<<i<<endl;
}
The while Loop

There are 4 kinds of while loop structures

- Counter-Controlled while Loops
- Sentinel-Controlled while Loops
- Flag-Controlled while Loops
- EOF-Controlled while Loops

1 Counter-Controlled while Loops

- An example
- Input: a set of integers started with the number of them
  3 10 20 30
- Output: their sum and average
  the sum of 3 numbers is 60
  the average is 20

```
Dev-C++.lnk
counter_controlled_while_1.cpp
```

1 Counter-Controlled while Loops

- Another example
- Input: a set of integers started with the number of them
  A sentinel variable is tested in the condition and the loop ends when the sentinel value is encountered
- Output: the number of zeros, positive numbers and negative numbers respectively

```
Dev-C++.lnk
counter_controlled_while_2.cpp
```

2 Sentinel-Controlled while Loops

- A sentinel variable is tested in the condition and the loop ends when the sentinel value is encountered
- The syntax is:
  ```
  cin>>variable;
  while (variable != sentinel)
  {
    cin>> variable;
    ...
  }
  ```
2 Sentinel-Controlled while Loops

Example 1
Input a set of integers ended by a sentinel (e.g. -9999)
1 2 3 -9999
Output their sum and average
The sum of 3 numbers is 6
The average is: 2

Example 2
Write a loop, input a letter, output corresponding telephone digit.
To end this program, please input '*'

Example 3
input a set of integers ended with -9999
Output how many zeros, positive numbers and negative numbers respectively

3 Flag-Controlled while Loops

A flag-controlled while loop uses a Boolean variable to control the loop
The flag-controlled while loop takes the form:

```cpp
found = false;
while(!found)
{
    ...
    if(expression)
    {
        found = true;
    }
    ...
}
```
3 Flag-Controlled while Loops

- An Example

- Input a set of integers, find the first negative number.

4 EOF-Controlled while Loops

- An example

- Input a set of integers ended with a letter
  10 20 30   A

- Output their sum and average
  the sum of 3 numbers is 60
  the average is 20

4 EOF-Controlled while Loops

- The eof Function

  - The function **eof** with an input stream variable can also be used to determine the end of file status
  - Like the I/O functions, such as get, ignore, and peek, the function **eof** is also a member of data type istream
  - The syntax to use the function **eof** is:
    istreamVar.eof()
  - where istreamVar is an input stream variable, such as cin
4 EOF-Controlled while Loops

The eof Function

- An example
- Read every character from a ‘in.txt’ file in the same directory as the source program, and output each character to the monitor, meanwhile calculate the length of the file.

Examples

```cpp
while_eof_example_1.cpp
```

write all the integers between 1 and 10 separated by a comma

1, 2, 3, 4, 5, 6, 7, 8, 9, 10

Program this with a while statement

Examples

```cpp
while_1.cpp
```

Input n, output n!

- n=1, 1!=1
- n=2, 2!=1*2
- n=3, 3!=1*2*3
- ...
- n = n!=1*2*3*...*(n-1)*n

Examples

```cpp
while_factorial.cpp
```

Input n, output n! Solution 1

```cpp
let i=1 and factorial=1
input n
while (condition)
{
    factorial = factorial *i;
    i=i+1;
}
```

Examples

```cpp
while_factorial.cpp
```
Examples

Input $n$, output $n!$  Solution 2

```c
factorial = 1
input n
i = n
while (condition)
{
    factorial = factorial * i;
    i = i - 1;
}
```

while_factorial_2.cpp

Examples

Applications

1, 1, 2, 3, 5, 8, 13, 21, 34, 55,…

What will be the next number?

Answer: 89

This is the Fibonacci sequence

$$f_n = f_{n-2} + f_{n-1}$$

$$f_1 = f_2 = 1$$

while_fibonacci.cpp
Examples

Applications

2, 3, 5, 7, 11, 13, 17, 19, 23, 29, …

What is the particularity of these numbers?

Answer: They are primes
Their divisors are 1 and themselves only.

Prime numbers have many applications in computer science
For example they are used to exchange information secretly and for authentication

Write a program to test if a number is prime or not

Algorithm:

Read a number n
while_prime_number.cpp

Test if each number from 1 to n is a divisor of n

If the number of divisors of n is equal to 2
then n is a prime
otherwise
n is not a prime
// this is not the best solution

The for Loop

- The general form of the for statement is:
  for(initial statement; loop condition; update statement)
  statement

- The initial statement, loop condition, and update statement (called for loop control statements) that are enclosed within the parentheses control the body of the for statement
The for Loop

- The for loop executes as follows:
  - The initial statement executes
  - The loop condition is evaluated
    - if the loop condition evaluates to true
      - execute the for loop statement
      - execute the update statement (the third expression in the parentheses)
  - Repeat the previous step until the loop condition evaluates to false
- The initial statement initializes a variable
- The initial statement in the for loop is the first to be executed and is executed only once

If the loop condition is initially false, the loop body does not execute

The update expression, when executed, changes the value of the loop control variable which eventually sets the value of the loop condition to false

The for loop executes indefinitely if the loop condition is always true

Fractional values
- can be used for loop control variables of the double type or any real data type
- should be avoided because different computers can give different results for the variables

A semicolon at the end of the for statement (just before the body of the loop) is a semantic error, in this case, the action of the for loop is empty

If the loop condition is omitted it is assumed to be true!
The for Loop

In a for statement, all three statements—initial statement, loop condition, and update statement can be omitted.

The following is a legal for loop:

```cpp
for(;;)
    cout<<"Hello"<<endl;
```

It is an infinite loop.

An example

```cpp
for (i=1; i<5; i++)
    cout<<setw(3)<<i;
```

// 1 2 3 4

```cpp
i=1
while (i<5)
{
    cout<<setw(3)<<i;
}
```

An example

```cpp
for (i=1; i<=10; i++)
cout<<setw(3)<<(11-i);
```

Output: 10 9 8 7 6 5 4 3 2 1

```cpp
for (i=10; i>=1; i--)
cout<<setw(3)<<i;
```

Input: the the number of positive integers to be added (e.g., 10)

Output: The sum of first 10 positive integers (e.g., 55)
for Loop

Input n, output n!  Solution 1

```cpp
f=1;
cin>>n;
for(i=1; i<=n; i++)
{
    f=f*i;
}
```

for_factorial_1.cpp

Input n, output n!  Solution 2

```cpp
f=1;
cin>>n;
for(i=n; i>=1; i--)
{
    f=f*i;
}
```

for_factorial_2.cpp

The for Loop

- An example
- Input: 20 integers
- Output: The number of zeros, positives and negatives.

The for Loop

Applications

1, 1, 2, 3, 5, 8, 13, 21, 34, 55,…

What will be the next number?

Answer: 89

This is the Fibonacci sequence

\[
\begin{align*}
    f_n &= f_{n-2} + f_{n-1} \\
    f_1 &= f_2 = 1
\end{align*}
\]
for Loop

Algorithm:
Read previous1, previous2 and n from the user
for(....)
{
current = previous1 + previous2;
previous1 = previous2;
previous2 = current;
}

for Loop

Write a program to test if a number is prime or not

Algorithm:
Read a number n
Test if each number from 1 to n is a divisor of n
If the number of divisors of n is equal to 2
then n is prime
otherwise
n is not prime

do...while

The general form of a do...while statement is:
do
statement
while(expression);
The statement executes first, and then the expression is evaluated
If the expression evaluates to true, the statement executes again
As long as the expression in a do...while statement is true, the statement executes

do...while

To avoid an infinite loop, the loop body must contain a statement that ultimately makes the expression false and assures that it exits
The statement can be a simple or a compound statement
If it is a compound statement, it must be enclosed between braces
Because the while and for loop has an entry condition, it may never activate, whereas, the do...while loop, which has an exit condition, always goes through at least once
Difference of while and do-while

The block in do-while will be executed at least once (1+ times)

In while-loop, the block will be executed for 0+ times

Example: output 0 5 10 15 20 25

```cpp
int i=0;
do{
    cout<<setw(4)<<i;i=i+5;
}while(i<=25);
```
Break & Continue Statements

- A **break** and **continue** statement alters the flow of control
- The **break** statement, when executed in a **switch** structure, provides an immediate exit from the switch structure
- The **break** statement can be used in while, for, and do...while loops
- When the **break** statement executes in a repetition structure, it immediately exits from these structures

The **break** statement is typically used for two purposes:

1. To exit early from a loop
2. To skip the remainder of the switch structure

After the **break** statement executes, the program continues to execute with the first statement after the structure

The use of a break statement in a loop can eliminate the use of certain (flag) variables

---

```cpp
int sum=0, num;
bool isNegative=false;
cin>>num;
while(cin&&!isNegative)
{
    if(num<0){
        cout<<"Negative number found in the data."<<endl;
        isNegative=true;
    }
    else{
        sum=sum+num;
        cin>>num;
    }
}
// end of while
cout<<"The sum is "<<sum<<endl;
```
Break & Continue Statements

```cpp
int sum=0, num;
cin>>num;
while(cin){
    if(num<0)
    {
        cout<"Negative number found in the data."<<endl;
        break;
    }
    else
    {
        sum=sum+num;
cin>>num;
    }
}  // end of while
```

```
cout<"The sum is "<<sum<<endl;
```

Nested Control Structures

Suppose we want to create the following pattern

```
*         //line i=1, star number j=1
**        // i=2, j=2
***       // i=3, j=3
****      // i=4, j=4
*****     // i=5, j=5
```

In the first line we want to print one star, in the second line two stars and so on

Since five lines are to be printed, we start with the following for statement
```
for(i = 1; i <= 5 ; i++)
```

The value of `i` in the first iteration is 1, in the second iteration it is 2, and so on

We can use the value of `i` as the limiting condition in another `for` loop nested within this loop to control the number of stars in a line

```
for(i = 1; i <= 5 ; i++)
{
    for(j = 1; j <= i; j++)
        cout<"*";
    cout<<endl;
}
```

The syntax would be:
```
for(i = 1; i <= 5 ; i++)
{
    for(j = 1; j <= i; j++)
        cout<"*";
    cout<<endl;
}
```
Nested Control Structures

What pattern does the code produce if we replace the first for statement with the following?

```
for(i = 5; i >= 1; i--)
```

i.e.

```
for(i = 5; i >= 1; i--)
{
    for(j = 1; j <= i; j++)
        cout<<"*";
    cout<<endl;
}
```

Answer:

```
*****
****
***
**
*
```

An example:

```
const int n = 5;
```

Output:

```
*  // i=1, j=1, blanks=2
*** // i=2, j=3, blanks=1
***** // i=3, j=5, j=2*i-1, blanks=0
     // blanks=3-i
```

stars_1.cpp